POLITECNICO DI TORINO

Master degree course in Computer Engineering

Master Degree Thesis

Optimization of Spiking Neural Networks execution on low-power microcontrollers

Advisors
Prof. Gianvito Urgese
Dr. Andrea Pignata
Dr. Vittorio Fra

Candidate
Simone Delvecchio

December 2025

Abstract

The rapid proliferation of AI has prompted researchers to direct significant attention towards the development of novel and innovative solutions that optimise its performance and power consumption. Spiking Neural Networks (SNN) represent a particular type of neural network that emulates the behaviour of the biological brain to enhance neural computation. This results in advantages such as low-power consumption, effective memory-processing colocation, and event-driven execution.

The potential benefits of neuromorphic computing could be realised through the utilisation of optimised neuromorphic hardware, such as SpiNNaker 2 and Intel Loihi 2. However, these accelerators are difficult to obtain and often expensive due to their experimental nature.

The present work investigates a potential solution to be implemented on microcontroller units (MCUs) to run SNN in small and low-power systems. To this end, SNN models are deployed leveraging HW modules such as DSP and memory access accelerators, available in the target architectures.

The development focused on reproducing Leaky-Integrate and Fire (LIF) and Izhikevich neurons, implementing fully-connected, one-by-one, and recurrent connectivity. This effort led to the implementation of the snn2mcu C library which currently supports two neuron models, three connectivity patterns, and two target MCUs.

The first target of the snn2mcu library is the general-purpouse ARM Cortex M7 core from STMicroelectronics. Quantization has been implemented converting floating point network parameters to fixed-point data types compatible with the ARM DSP library (CMSIS) that provides an optimised implementation of common math operations and vectorised functions. The second target is the open source GAP8 architecture with Risc-V processors. Optimization have been implemented by exploiting the multi-level memory layout and the multiple cores available in the system.

snn2mcu also supports SNNs described with the Neuromorphic Intermediate Representation (NIR) framework. An automatic tool has been designed to generate optimized firmware starting from the high level NIR description of the SNN, thus removing the need for manual coding and facilitating development on embedded systems.

To evaluate the usability of the library, two SNN models were deployed on the GAP 8 and ARM core. The first is a SNN designed to classify seven Braille characters using input spikes from a robotic finger that produces twelve signal channels. The second is a classifier for handwritten numbers taken from the MNIST dataset. The Braille SNN model on the ST-ARM was executed 6x faster than floating point SNN simulators achieving the same accuracy of 91.43% over a 140-sample test dataset. The second model achieved an average energy consumed per sample of 10.54 mJ and an execution time of 187.82 ms per sample, and power consumption in line with a specific optimised FPGA design. In summary, the present work demonstrates that the utilisation of optimisation techniques, such as fixed-point mathematics with reduced bit-depth, DSP vectorised functions, multilevel cache, and code efficiency, can enable an effective deployment of SNNs on a wide variety of commercial grade MCUs.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The rapid diffusion of artificial intelligence into everyday devices has created a strong demand for solutions that can run sophisticated models directly on low-power, resource-constrained hardware at the edge, such as microcontroller-based boards and small system-on-chip platforms. Traditional deep neural networks are typically executed on GPUs, high-end CPUs, or specialised accelerators, making them very difficult to integrate into embedded systems that must operate with strict limits on energy, memory, and real-time responsiveness. Neuromorphic computing and, in particular, Spiking Neural Networks(SNNs) offer a compelling alternative because they model information processing using discrete spikes and event-driven dynamics, enabling sparse activity, local state, and a natural fit for low-power, always-on sensing tasks. However, bridging the gap between advanced neuromorphic models and widely avaiable microcontrollers remains challenging due to the lack of dedicated hardware support and standard, reusable software workflows.
Current approaches to neuromorphic computing have largely focused on specialised hardware platforms such as SpiNNaker 2 and Intel Loihi 2, which integrate custom accelerators, advanced memory hierarchies, and event-driven architectures to execute SNNs with high efficiency. These platforms demonstrate excellent performance and energy efficiency but are often expensive, hard to access, and not suitable for large-scale deployment in commercial embedded products. At the same time, software frameworks such as snnTorch and the Neuromorphic Intermediate Representation (NIR) have significantly simplified the design, training, and representation of SNN models at a high level, but they do not natively provide optimised, turnkey support for deployment on generic low-power microcontrollers. Other edge-oriented toolchains, like PULP-NN for GAP-based SoCs and ST Edge AI Core for STM32 devices, strongly optimise classical quantised neural networks yet offer little or no direct support for SNN-specific primitives and spike-based execution. As a result, there is a clear gap between high-level SNN research tooling and practical, reusable workflows for deploying SNNs on common MCU platforms such as ARM Cortex-M7 and RISC-V based systems like GAP-8.

This thesis addresses that gap by proposing and implementing a complete workflow and software library for executing SNNs on low-power microcontrollers using only widely avaiable hardware and open or standardised software components. The work centres on the design and deployment of the snn2mcu C library, which supports two neuron models, Leaky-Integrate-and-Fire (LIF) and Izhikevich neurons, and multiple connectivity patterns, such as fully connected, recurrent, and one-by-one topologies, targeting both ARM Cortex-M7 based STM32H757I-EVAL board and the GAP-8 RISC-V SoC. High-level SNNs are first designed and trained in Python using snnTorch; then, through a quantisation and optimisation process, they are translated into efficient fixed-point implementations that exploit DSP extensions, vectorised arithmetic, and multi-level memory structures available on the target MCUs. The library is further made compatible with NIR, so that SNN models described in NIR can be automatically converted into optimised embedded code, enabling a hardware-agnostic, reproducible pipeline from research prototypes to deployable firmware. The methodology is validated using two representative case studies: a Braille tactile classifier and a handwritten digit classifier based on MNIST, demonstrating that commercial MCUs can achieve accuracy and energy efficiency comparable to more specialised solutions when properly optimised.
The methodology followed to develop this system is briefly described as follows:

- **Research and analysis**: A review of all the tools used has been conducted, including common SNN models, neuron dynamics, training strategies, and state-of-the-art neuromorphic hardware and frameworks. This was done to identify gaps in SNN support on widely available MCUs.
- **Single-neuron modelling**: The neuron equations are then replicated in hardware platforms using smart quantisation and optimised operations. The LIF neuron dynamics and Izhikevich implementation are adapted for fixed-point arithmetic.
- **Single-network implementation**: Trials have been conducted on very small networks, and the Braille classifier has been reproduced on both the ST Board and the GAP-8 SoC, with validation against the snnTorch reference simulations.
- **NIR-to-C parser**: A translator has been developed that converts SNNs described in NIR into the corresponding C code and headers, which are compatible with snn2mcu and, in particular, the ST Board.
- **Parser benchmarking on a real use case**: Ultimately, the parser was evaluated using a real-world SNN. In this case, it was a MNIST model that was generated using pre-existing tools: Spiker + combined with the NIR exporter. Correctness checks were performed, and the execution time and energy per inference were benchmarked on the ST Board.

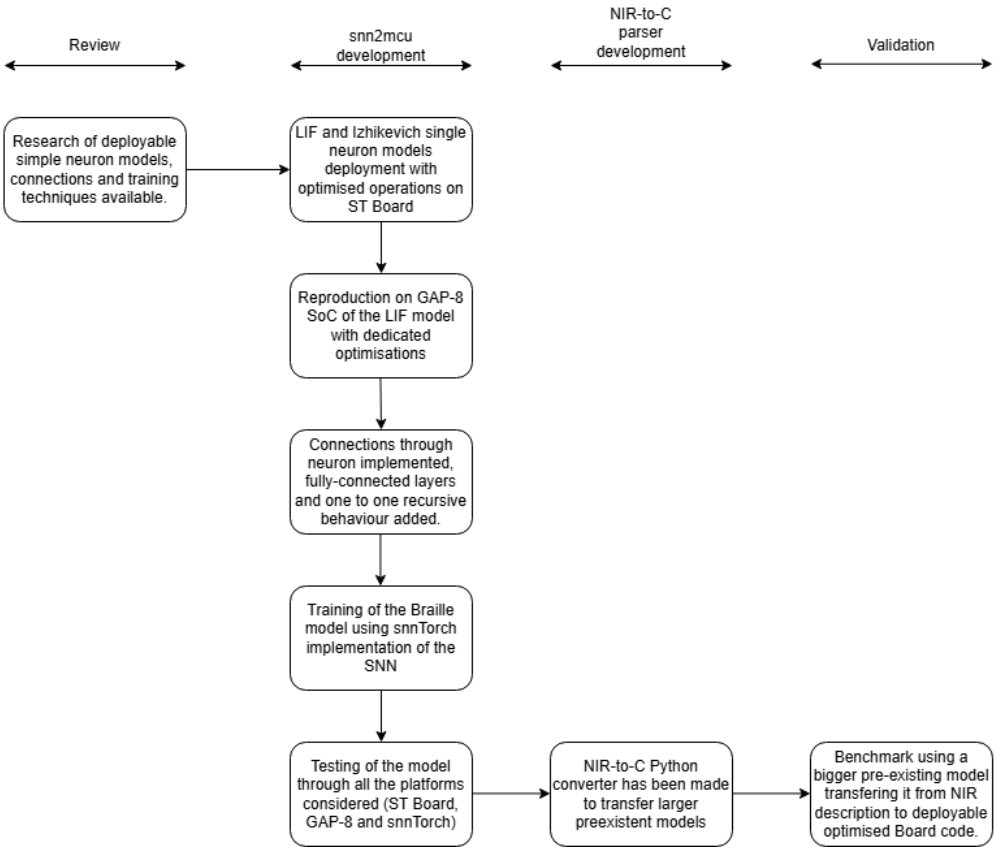

Figure 1.1: Workflow followed for the thesis

Chapter 2

Background

One of the most challenging aspects of this project is identifying and utilising novel solutions to optimise SNNs on edge devices. The first section will explain why SNN inferences on the edge are an effective solution for neuromorphic computing, providing a detailed description of the features and characteristics that define this approach. Subsequently, the underlying equations of the neuron model will be examined in more detail. This analysis will also emphasise how the model is employed and why it outperforms conventional neural networks. Several existing SNN implementations will also be described to determine the primary fields in which these architectures are used. The use of specific accelerators in academic settings will be examined, focusing on their effectiveness in executing SNN operations. The investigation will delve into the intricacies of these accelerators, elucidating their strengths and limitations. Furthermore, the chapter will examine frameworks for developing SNNs at a higher level using high-level languages such as Python. It will provide descriptions and details of the hardware used and explore its potential applications in general neural network inference. The analysis will present specific tools and libraries, demonstrating how the research in this thesis fills a significant gap in the field of SNNs. Several specific tools for developing edge SNNs have already been developed, and these will be described in the following section. Finally, the chapter will conclude with an explanation of the importance of all the previously described research and projects to this contribution.

2.1 SNN: overview and neuromorphic computing over the Edge

Among the best ways to deploy intelligence at the edge, i.e., directly on a hardware platform, is surely the use of neuromorphic computing. As described in [1], neuromorphic computing exploits methods for developing intelligent systems inspired by the biological brain. This particular implementation allows for greater efficiency,

scalability, and adaptability in intelligent applications.

Neuromorphic computing features Traditional Von Neumann architectures are based on the hard decoupling of computing units (CPUs) and memory units (typically RAM). This results in high energy consumption and speed bottlenecks caused by the transfer of data between memory and processing units. The idea of neuromorphic computing, on the other hand, is to couple memory and computation, shortening the distance between them using different types of architecture. This is achieved by incorporating neurons into intelligent architectures that endow applications with smart behaviour simply by preserving a state, generating spikes, and processing weights.

Another important feature of neuromorphic computing is sparsity: indeed, neurons in a network can perform actions sparsely, i.e. not all neurons are active throughout the execution of each step, but only some of them. This enables energy consumption to be kept low, with an average overall neuron activity that is sparse for each timestep.

Traditional AI systems typically rely on global backpropagation during learning, which causes high energy consumption when updating all the weights across all the layers of the network. Instead, neuromorphic systems could improve this learning process by focusing on local pre- and postsynaptic activity. By exploiting these rules, the system can update weights based on local activity features, favouring continuous learning.
Neuromorphic systems can also perform actions across a wide range of timesteps, operating with fast synaptic iteration as well as slower training, and have high temporal processing capabilities. This feature is fundamental when different sensed data need to be processed in a short amount of time for real-time applications.

Another key feature of neuromorphic systems is their brain-inspired hierarchical organisation, which, alongside layer structure, enables the easier processing of large and complex sensory information while reducing redundancy, a characteristic of normal non-hierarchical structures.

Ultimately, brain-inspired technology is also easy to deploy on hardware systems with strict memory limitations. This is because it uses spike event-driven technology instead of computing specific inputs, as is the case with normal deep learning models. The intrinsic parallelisation of the networks also allows systems to be developed that exploit the parallel execution of different neurons in a layer altogether to improve overall performance.

SNN models and training In order to facilitate a more profound comprehension of the Spiking Neural Network behaviour and its characteristic dynamics, a concise presentation of the aforementioned networks and the manner in which they are exploited is herein provided. As stated in the review by Nguyen (2021) [2], neurons are inspired by actual brain biological behaviours. However, it is specified here that the Hodgkin-Huxley model is the most accurate model in terms of brain behaviour, but that it is complex to use. In this review, other simpler neuron models are briefly described. One such example is the Izhikevic model, which, due to its dynamics and non-linear behaviour, represents an optimal trade-off between computational power and feasibility. Notwithstanding, the most frequently employed neuron models in accordance with the cited literature are the Leaky-Integrate and Fire (LIF) model and the more elementary Integrate and Fire (IF) model. The behaviour of these models is predicated on the accumulation of weight from preceding spikes, in conjunction with the dynamic generation of spikes at the level of spike generations. The membrane potential of the neuron attains a threshold value, thus instigating the aforementioned behaviour. Furthermore, the membrane potential undergoes a leakage loss over time, irrespective of the occurrence of spikes. In the event of a spike being performed, it is within the capabilities of the modeler to place the neuron in a recovery phase that temporarily disables it. The review elucidates that a number of models have been subjected to trials for the implementation of LIF (Leaky Integrate-and-Fire) or more straightforward models, which have demonstrated commendable performance. However, contemporary researchers are endeavouring to identify methodologies for the incorporation of computationally expensive neurons, such as Hodgkin-Huxley, within hardware devices. This objective is being pursued through the utilisation of numerical procedures, including the Euler method, for the purpose of simplifying these models.

In this review, the function of the synapse in SNNs is elucidated. The synapse is defined as a connection between neurons in the network, and it has been demonstrated that whenever a neuron spikes, the spike affects the neuron linked with a positive (excitatory) or a negative (inhibitory) behaviour on the state of the connected neuron. The management of this process is facilitated by the synaptic weights. In contrast to conventional artificial neural networks, these weights demonstrate plasticity over time, adapting to the behaviour of spikes originating from pre- and post-synaptic neurons.

SNNs differ from conventional artificial intelligence in that they can be trained in various ways, making them more adaptable to different inferences. As previously mentioned, one training method exploits the adjustment of strengths according to the timing difference between pre- and postsynaptic spikes. This is known as spike-timing-dependent plasticity (STDP). In this context, the temporal proximity of spikes modulates the strength of connections. Reduced temporal proximity

enhances strength, while increased temporal proximity reduces it. However, this method is not employed for comprehensive learning due to its inability to coordinate complex learning across different layers in large networks. A more prevalent learning method is supervised learning, which is based on analysing the gradients used to minimise the distance between the expected and produced outputs. Spike generation is a discrete process involving non-differentiable activation functions. To overcome this issue, surrogate gradient activation functions are employed to convert the spikes into continuous real values, thus facilitating gradient flow during training. This approach has been shown to deliver levels of accuracy comparable to those of a standard artificial neural network. An alternative method involves converting previously trained artificial neural networks (ANNs) into spiking neural networks (SNNs). While this approach is effective, encompassing all ANN training methods, there is a caveat: not all ANNs can be translated into SNNs. In most cases, the optimisations made for ANNs cannot be transferred directly to SNNs. This can result in concerns regarding power consumption and a potential increase in complexity.

A deeper look at LIF model In [3], is provided a detailed explanation of what a LIF neuron is, how it works and the key equations and elements that describe its behaviour. As previously mentioned, this is one of the most common neuron models used in typical spiking neural networks (SNNs), and it has some core elements that describe its dynamics during execution.

- **Membrane voltage integration**: neuron integration of incoming current represented as input **J(t)**.
- **Leakiness**: During execution, there is a leakage towards resting potential governed by a membrane time constant RC, where R stands for resistance and C stands for capacitance.
- **Firing threshold**: when the membrane voltage v(t) reaches threshold V_{th} the neuron generates a spike.
- **Reset and refractory behavior**: the neuron resets to resting potential for a refractory period indicated by t_{ref} after a spike; this is used to give the model more time awareness. Among the most commonly used reset state behaviours there are the reset to zero and the subtractive reset. The former assigns the neuron a membrane potential of 0, while the latter, when triggered by a spike, subtracts a potential amount equivalent to the threshold value. This process affords the neuron model enhanced temporal dynamics.

If we want to represent the neuron model dynamics using one single equation, which is time-continue, in the same paper, the equation is described like:

$$RC\frac{dv(t)}{dt} = -v(t) + J(t)$$

From the same document, there is also a description of the Steady-State firing rate equation, often used to predict the spiking dynamic in a normal execution when J(t) = j is a constant current:

$$r(j) = \frac{1}{t_{ref} + RClog(\frac{1}{1-\frac{V_{th}}{j}})}$$

if j is grater then V_{th}, 0 otherwise.
This equation is useful for training with this specific neuron model. As mentioned in previous research, the spike pattern must be 'smoothed' to make it compatible with common training techniques, such as back propagation. To achieve this, the following soft function is often used:

$$r(j) = \frac{1}{t_{ref} + RClog(\frac{1}{1-\frac{V_{th}}{\rho(j,\gamma)}})}$$

where the element $\rho(j, \gamma)$ is formed by:

$$\rho(j, \gamma) = \gamma log(1 + exp(\frac{j}{\gamma})$$

where exp stands for the exponential function, and the factor γ is a smoothing parameter used to smooth the response curves made by LIF neurons for differentiability.
These are all the useful pieces for working with LIF neurons in the best way. Section 3.1.2 will describe the neuron dynamics used in this thesis, namely the LIF dynamic equation in the context of a discrete-time application. Indeed, if we discretise the aforementioned equation from the cited paper, we obtain:

$$V_{T+1} = V_{reset} + \beta(V_T - V_{reset}) + (1 - \beta)J(t)$$

Where the β will later be described as a factor derived from RC in the equation (RC $= \tau$ there), and the input current $(1 - \beta)J(t)$ corresponds directly to the strengths summed together during each executive cycle in that section.

2.2 Use cases of SNNs

In this section, a couple of use cases will be analyzed in order to be more aware of the current state-of-the-art architectures.

2.2.1 Neu-BrAuER

The first use case that deserves a mention is Neu-BrAuER [4]. The workflow is briefly summarised in Figure 2.1. This work focuses on the classification and audio reading of Braille letters from sensory touch data. Neu-BrAuER can interpret signals detected by commercial capacitive pressure sensors and enable the correct pronunciation of letters according to the classification after sensing. The network structure consists of a three-layer, fully connected recurrent spiking neural network (RSNN) that uses LIF neurons. The model senses data from 12 different channels and can be implemented directly on commercial edge devices. To optimise inference execution over the edge, quantisation is performed to reduce all neuron states to 4- and 8-bit precision within different network layers, achieving gains in memory and execution efficiency without significantly degrading learning dynamics. Training of the network was performed using GPU accelerators and SNNtorch, and the model was deployed as an ONNX model for use on commercial MPUs. The resulting system is fast and optimised for real-time scenarios involving the acquisition of numerous samples without degradation.
This work also reports excellent results. Indeed, a median classification accuracy of 73.09% is achieved, with a standard deviation of 1.08%, across 50 test runs on a balanced dataset containing 27 classes. Some letters achieved 100% accuracy, but others, such as O, P and Q, were more difficult to classify. The network was executed on an STM32MP157F board and the median inference time was 264 ms per sample, with a peak of 400 ms. These results were also achieved with thousands of sample executions. The energy consumption is 313 mJ per inference, which is suitable for wearable or battery-powered devices. These results demonstrate that good results can be achieved when executing SNNs, even with small bit-widths and commercial tools.

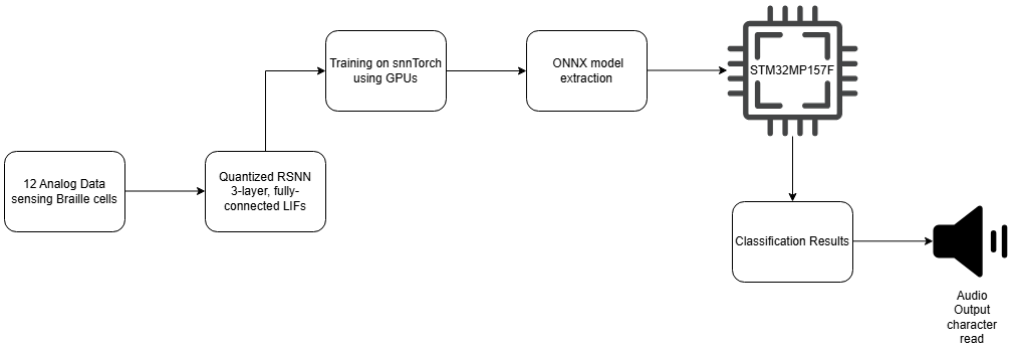

Figure 2.1: Workflow of Neu-BrAuER development

2.2.2 SNN-based HAR on Commercial Edge devices

To understand the power of SNN current implementations, it has been considered another use case presented in [5]. This work focuses on human activity recognition (HAR) using a neuromorphic architecture called L2MU, which is a spiking variant of the Legendre Memory Unit (LMU) model. This work enhances the LMU model's capabilities by employing a LIF neuron structure that focuses on both simple LIF models and their more advanced synaptic variants. This architecture is designed to take raw sensor data; therefore, no layer is needed to convert that data, and it can be implemented directly on edge devices as well as on specialised neuromorphic hardware. The architecture incorporates an encoding module capable of transforming continuous sensed data into spiking activity using a multi-layer LIF-based neural population. The model also supports hyperparameter optimisation for network architecture and specific neuron dynamics, which is useful for selecting the optimal structure required for high HAR task recognition accuracy.
This model uses a dataset (WISDM) comprising smartwatch sensor data that represents various actions performed by multiple subjects. The samples are segmented into two-second windows and the network specialises in recognising hand-related activities, categorising them into seven different values. The data is divided into three sets: 60% for training, 20% for validation and 20% for testing. L2MU translates ordinary LMU equations by exploiting only LIF neuron populations and spike patterns, ensuring full neuromorphic flow.
Adaptations to embedded and real-time applications are made by exploiting weight pruning (via the Sconce Python library) and retraining. This is an innovative way to compress SNN models for hardware with resource constraints, resulting in only a small loss of accuracy. Trials were conducted using three different commercial platforms: STM32MP157F-DK2, Raspberry Pi 3B and Raspberry Pi 4B. These boards run standard Linux-based operating systems and SNN inference is conducted using ONNX Runtime with PyTorch-based conversion for snnTorch.
Overall, both leaky and synaptic neurons performed very well, with median test accuracy above 93%. However, LIF neurons performed slightly better, especially after compression, demonstrating superior accuracy, stability and greater efficiency in memory usage.
The same model, which uses LIF neurons, is deployed on each commercial board, achieving the following results:

- STM32MP1 board has used 65.7 MB of RAM with 0.13 s as Mean Inference Time and 215.1 mJ as Mean Energy per Inference.
- Raspberry Pi 3B board has used 77.8 MB of RAM with 0.06 s as Mean Inference Time and 268.8 mJ as Mean Energy per Inference.
- Raspberry Pi 4B board has used 77.4 MB of RAM with 0.03 s as Mean Inference Time and 153.9 mJ as Mean Energy per Inference.

The accuracy achieved is 93.91%, which is consistent across all models. Synaptic accuracy is slightly lower, but RAM consumption is consistently higher across all devices. Overall, this work demonstrates that, if correctly designed and implemented, simple neuron models such as LIF can achieve high enough accuracy to justify their usage, even for commercial devices.

2.3 HW designed for neuromorphic applications

SNNs provide good results when executed on commercial platforms, but SNNs perform best when designed specifically for hardware devices optimised for neuromorphic operations. This section will describe two different types: Spinnaker 2 and Loihi 2 by Intel.

2.3.1 Spinnaker 2

Spinnaker 2 introduced several innovations in the field of neuromorphic computing, with a focus on energy efficiency, scalability and hybrid computation for neural networks. As stated in [6], the chip uses advanced 22 nm FDSOI technology with adaptive body biasing to reduce energy consumption and address process variation. It efficiently performs at low voltages of around 0.5 V to ensure lower energy consumption. Dynamic voltage and frequency scaling (DVFS) connects power supplies and clocks directly to computations, primarily neural spiking, and can adapt power consumption dynamically and coherently in response to the workload. This results in around 60% less power consumption than using a static power supply for typical SNN operations.
The processor architecture is the ARM Cortex-M4F core with single-precision floating-point computation, an improvement on the previous Spinnaker design. There are also integrated accelerators within each processing element (PE), which boost typical neural network computations. These consist of dedicated MAC arrays (4x16 8-bit units) target matrix and convolution operations, executing in parallel independently from the main core to speed up neuromorphic operations. Other accelerators focus on exponentials, logarithmic operations and random number generation to facilitate neuromorphic simulations.
A quad-processing-element (QPE) modular unit is used to improve the architecture. Each unit incorporates four PEs and a Network-on-Chip (NoC) router in a Globally Asynchronous Locally Synchronous (GALS) layout, which minimises complexity and power demands. The NoC system incorporates two different interlaced structures for data and configuration, using two asynchronous FIFOs, logic to handle various neural events, error correlation features and out-of-order buffers to increase speed and enable scalable communication between multiple PEs. Local and distributed memory architectures separate SRAM banks, minimising access

contention to enable full parallel processing across elements.
So, Spinnaker 2 uses a hybrid approach, combining common processor architecture with a set of accelerators that optimise common, repetitive neural operations. This makes the system perform faster while consuming minimal energy. Its architecture is also optimised for event-driven MAC usage, enabling efficient multi-bit signal processing to mimic biological graded neuron responses, or it could be exploited for complex learning mechanisms.
To effectively demonstrate the model's enhanced performance, standard benchmarks were conducted. SpiNNaker 2 achieves efficiency rates of approximately 16–20 CoreMarks per MHz and 1.47–1.75 TOPS/W for 8-bit operations, all at low supply voltages, when running standard benchmarks. The accelerators made it possible to execute convolutional and matrix operations using the MAC arrays, achieving speedups of up to 610x and energy consumption reductions of up to 652x for convolutional layers compared to executing the same kind of operations with the ARM core only. This hybrid approach enables support for the neural engineering framework (NEF) via the ARM core, while the accelerators handle all the repetitive, energy-intensive neuromorphic operations. Overall, this system's scalability and architecture model make SNN development and execution much easier and faster.

2.3.2 Intel Loihi 2

Another important system that is highly optimised for neuromorphic operations, and more precisely, SNNs, is the Intel Loihi 2. The main features and characteristics of this model are described in detail in [7]. Its architecture comprises microprocessor cores and up to 128 asynchronous neuron cores, which are linked together with a network-on-chip (NoC). The neuron cores specialise in executing high-speed spiking neuron computations and spiking logic.
The neurons are programmable using custom microcode, which allows various custom neuron models to be implemented, not just common models such as LIFs. Improvements have been made on the previous model and this chip is now capable of computing spikes with integer-valued payloads, improving precision while maintaining the sparsity introduced by spiking behaviour. Loihi 2 can handle the latest learning algorithms, such as backpropagation, and its architecture can improve synaptic and neuronal density. It exploits advanced memory partitioning, convolutional, factorised and stochastic connectivity, and compressed synapse encodings to increase efficiency by up to 160x. As previously mentioned, performance is dramatically boosted with up to 10x faster spike generation, 5x faster synaptic operations and 2x faster neuron updates compared with previous models. Timesteps are executed at a speed of 200 ns to outperform real biological circuits. Thanks to Ethernet and GPIO ports, it is also easy to integrate with other systems to communicate with different sensors and environments.

The Loihi 2 offers many features for customising and optimising neuron behaviour. Unlike the previous model, it also has soft-partitioned, asynchronous memories to speed up memory access and neuron dynamics. In terms of results, this model also consumes less than 1 watt in real-world applications, which is lower than general-purpose GPUs or CPUs, which could consume 10 to 100s of watts for similar workloads. It also supports Sigma-Delta Neural Networks, achieving 10x faster speeds and lower energy consumption than previous neuron models. All other neuron updates are also improved. A single Loihi 2 chip can support up to one million neurons and 120 million synapses by exploiting compression and flexible resource allocation. Unlike other models, the Loihi 2 is supported by Lava, an open-source, community-driven software framework that makes this system easier to use and handle. Lava supports simulation and profiling across CPU, GPU and neuromorphic hardware, promoting broader adoption and facilitating experimentation. This model's strengths are undoubtedly its open-source software access and high programmability.

2.4 Frameworks to develop SNNs

As mentioned in previous sections of this chapter, SNNs produce excellent results in neuromorphic systems simply through their use. Frameworks are designed to facilitate and streamline their development. SNNTorch is a PyTorch-based framework for SNN development. SNNs can be represented using the Neuromorphic Intermediate Representation (NIR), which makes them easily portable between different development environments.

2.4.1 snnTorch

The main features of this Python framework are detailed in [8]. This framework optimizes spike representation through the networks using single-bit values, which greatly reduces computational costs, particularly when deployed on actual hardware platforms. The framework now supports LIF neurons, current-based neurons, recurrent structures and advanced models such as spiking LSTMs and spiking Transformers. SNNs can interpret and produce spikes using rate coding (encoding information in spike counts), latency coding (using the timing of spikes), delta modulation (responding to input changes) and population coding (distributing information across groups), providing significant compatibility with systems that use time-dependent inputs, such as vision, sound and biosignals.
SNNs have huge compatibility with deep learning tools simply by being expressed as discrete-time equations. Like these tools, SNNs can benefit from features such as batch normalisation, residual connections and automatic differentiation.
Supported training techniques include:

- **Surrogate Gradient Descent**: it overcomes the non-differentiability of spikes using a smooth, differentiable surrogate function during the backward pass, making spiking neural networks (SNNs) compatible with backpropagation through time (BPTT). Various surrogate functions can be employed, such as sigmoid, arctan and triangular, and these can be used as hyperparameters of the network.

- **Objective Functions**: there is also support for different loss/objective settings, such as spike-rate targets and spike-timing (latency) objectives. Objectives that operate directly on the membrane potential are also supported, enabling compatibility with both rate- and latency-based learning schemes.

- **Local and Online Learning Algorithms**: Alongside traditional backpropagation with temporal unrolling, localized, biologically inspired approaches are also supported, with methods such as e-prop, Decolle and event-based plasticity (three-factor rules) implemented to enhance biological plausibility.

SNNs can be defined and structured using a familiar deep learning syntax. Layers, synapses and neurons are also fully compatible with standard PyTorch modules. Following normal ANN workflows, training is eased by simply adding time-stepped simulations and spike-based activities. Optimizers such as Adam and SGD are also available. Parameters such as the decay factor, thresholds, weights, biases, and other typical parameters can be set to be learnable, thereby enhancing the network's plasticity. The framework can easily bridge the gap between traditional ANN methods and new SNN methods, demonstrating that SNNs can be trained directly using deep learning methodologies while retaining brain-like characteristics. Along with the paper, the documentation is easily available online and contains ready-to-use tutorials for developing SNNs. Ultimately, the framework is excellent for easily training SNN models and has the tools to perform simulations on the models that can later be easily reproduced on specific target platforms.

2.4.2 Neuromorphic Intermediate Representation

The paper [9] describes the main features of the first standardised reference for neuromorphic computing, which is a powerful tool for future progress in this field. As NIR is completely platform-independent, it can be used to describe neuromorphic structures and computations at a high level of abstraction, facilitating the porting of the same model across different tools. NIR has definitive and fixed primaries that correspond to the standard features of neurons and connections between layers. These primaries can capture both continuous and discrete dynamics, enabling a complete description of SNNs, which can be deployed more easily and faithfully over hardware platforms. SNN development focuses solely on the model itself, not its specific implementation, enabling faster reproducibility. As the model

description is graph-based, the primitives of NIR are represented as nodes in this graph (e.g. leaky integrator, integrate-and-fire neurons, linear layers, convolutions and spikes), with edges representing signal flow. This makes it easy to manage the complexity of SNNs in simple structures, which also makes model development easier. NIR is purely declarative and completely machine-readable, and it also allows for the future extension of the primitive set as technology evolves and new structures and neurons emerge. NIR already supports seven neuromorphic simulators and four contemporary digital neuromorphic hardware platforms, including Intel Loihi 2, SynSense Speck, SpiNNaker2 and Xylo. This enables faster testing and benchmarking of the same model across different platforms, facilitating performance analysis and the identification of optimal solutions. Three representative tasks were used to conduct comprehensive experiments: a leaky integrate-and-fire neuron (basic), a spiking convolutional neural network (vision), and a spiking recurrent neural network (temporal processing). These models were then executed on all available platforms to examine qualitative and quantitative similarities in computational outcomes. In terms of accuracy, the Sequential Convolutional Neural Network (SCNN) model trained on neuromorphic MNIST achieved a mean test accuracy of 97.7%, with a standard deviation of 0.9%, resulting in the reliable preservation of the model across different simulators and processors. In summary, NIR occupies the same position in SNN development as ONNX or MLIR in deep learning, paving the way for a comprehensive, standardised representation of SNNs that has the potential to incorporate additional platforms.

2.5 Analysis of low power HW

One of the issues with the neuromorphic hardware previously described is that it is often difficult to acquire, expensive, and unavailable for purchase. This section will analyse two alternatives to traditional hardware that could be used to develop SNNs by exploiting the power of neuromorphic models. These are the STM32H757I-EVAL ST Board and the GAP-8 based on PULP.

2.5.1 STM32H757I-EVAL

The characteristics of the STM32H757I-EVAL ST Board (Figure 2.2[1] are detailed in the STMicroelectronics documentation [10]. It is clear from this documentation that the board is designed to deliver robust computational performance while optimising energy efficiency, making it suitable for demanding applications as well as power-sensitive deployments. The board has a dual-core setup with a high-speed

[1] `https://www.st.com/en/evaluation-tools/stm32h757i-eval.html`

Arm Cortex-M7 and Cortex-M4 (optimised for low power and real-time signal processing), and provides 2 MB of flash memory and 1 MB of RAM for storing code and data. It also has a variety of peripherals that can be used to integrate developed applications with other environments, such as USB OTG HS/FS, Ethernet, CAN FD, audio interfaces (DAC, ADC and SAI), digital MEMS microphones and extensive memory support (SDRAM, SRAM, NOR and Quad-SPI flash, as well as a microSD card). The board also features a 4" 480×800 TFT colour touchscreen display with a MIPI DSI interface, as well as a hardware cryptographic accelerator, which is absent from the STM32H74xI variants. To enhance system compatibility with other environments, support for multiple connectivity interfaces has been enabled, including USB ports, Ethernet and CAN FD for high-speed networking, I2C and RS-232, as well as extension connectors. Complete debugging and programming support is also provided through the use of STLINK-V3E. The system also supports custom voltage supply to cores; in particular, the microcontroller's core voltage can be supplied by either the internal, high-efficiency, DC/DC switch-mode power supply (SMPS), or a linear regulator (LDO). The SMPS is set by default and is generally best for low power, but the LDO could also be used, or a hybrid approach could be employed. The board can also be configured using specific jumpers to select the desired features for particular applications. For example, a jumper can be used to select one of six independent power supply options. STLINK USB, multiple USB OTG ports, an external 5 V DC adapter or power from daughterboard connectors, enabling the board to be used in almost every scenario. Current limitation and overcurrent detection are also in place to protect the board itself and any linked daughter boards, and the board status can easily be seen through the use of embedded LEDs.
All of the described interfaces also have low-power modes. For example, the USB and Ethernet PHY interfaces can be configured for low-power states when not in use, and in specific scenarios, the RTC with backup battery can be used for ultra-low-power timekeeping. The board is equipped with various peripherals, including a potentiometer, a joystick, a tamper switch, a wake-up button, a reset button, multiple general-purpose LEDs and an LCD touchscreen, to facilitate interactive prototyping and HMI projects. The MFX IO expander provides additional GPIOs and peripheral control. The board is supported by many major development environments, such as IAR, KEIL MDK-ARM and STM32CubeIDE. Numerous example programs in the free STM32Cube MCU package demonstrate how to use the peripherals, along with the relevant libraries. These libraries allow you to create fully customised and optimised systems. Now, a deeper explanation of the main core will be given to demonstrate the power of this architecture in low-power scenarios.

Arm Cortex-M7 All the main characteristics of the M7 core featured in the board are gathered in [11]. It adopts the Armv7E-M instruction set and boasts a

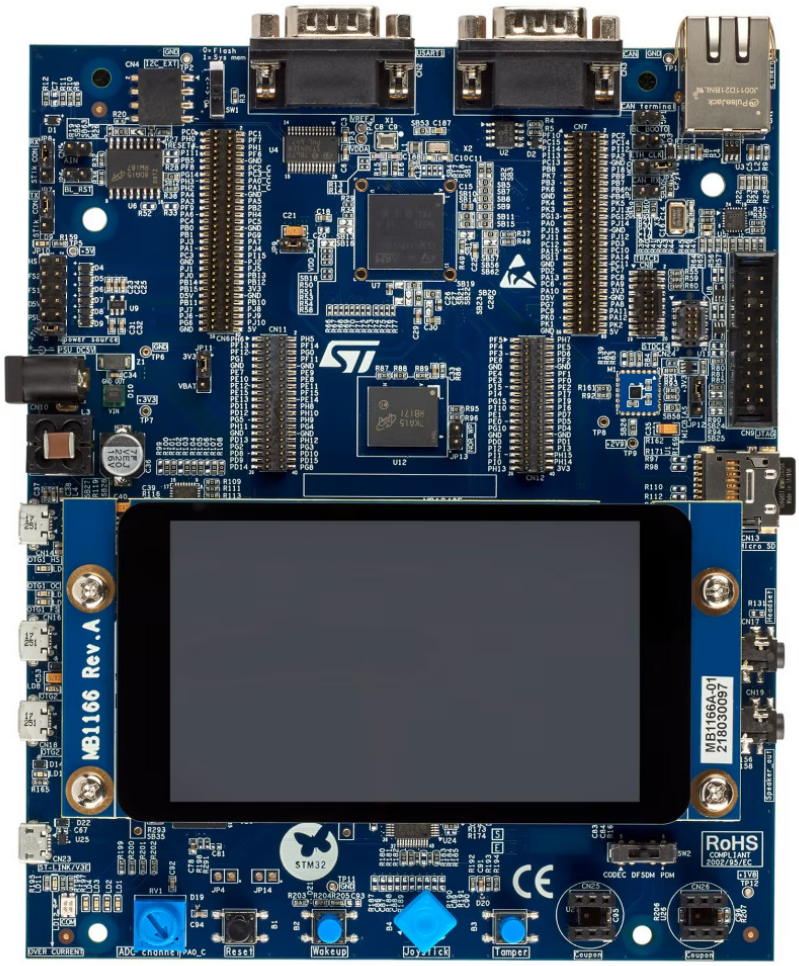

Figure 2.2: STM32H757I-EVAL ST Board

6-stage, dual-issue superscalar pipeline. This enables the parallel execution of multiple instructions, thereby increasing throughput in computationally intensive scenarios. It also has hardware branch prediction capabilities and integrated tightly-coupled memories (TCMs), giving the system high compatibility with real-time decision-making scenarios. Furthermore, it is described as outperforming previous models by achieving outstanding benchmark scores of 5.01 CoreMarks/MHz. To minimise memory latency, the system also has built-in instruction and data caches for memory-bound scenarios. Up to 16 MB of instruction and data TCMs

are designed to provide deterministic performance in real-time scenarios with predictable access times. Secure access control is also granted through the use of a Memory Protection Unit (MPU). One of the most important features of this core architecture is the advanced digital signal processor (DSP) extension, which features single-cycle multiply-accumulate (MAC) instructions and SIMD (single instruction, multiple data) arithmetic for parallel data processing (eight 16-bit operations per cycle). The hardware divide unit is particularly useful for the fast execution of mathematical operations. Operations such as multiply-accumulate, division and square root are supported by an optional floating-point unit (FPU) for single and double precision. The architecture is built around a high-speed AMBA4 AXI and AHB interconnect supporting wide (64-bit) communication with external memories and other system on a chip (SoC) peripherals. Advanced debugging, tracing and external system management are also facilitated through interfaces such as APB, ATB and private peripheral buses. Optional elements such as the embedded trace macrocell (ETM), the data watchpoint and trace (DWT) and the instrumentation trace functionality provide precise real-time debugging and trace analysis, which is particularly important in real-time scenarios. The Nested Vectored Interrupt Controller (NVIC) enables rapid interrupt handling and supports up to 240 configurable external interrupts and 256 priority levels, providing comprehensive customisable interrupt management for critical systems. To minimise the board's energy consumption when the system must remain inactive, the optional Wake-up Interrupt Controller (WIC) and low-power sleep modes could be used to allow dynamic power consumption that is consistent with the workload.
Now, we will take a closer look at DSP extension to enhance the optimisations made to speed up mathematical operations.

DSP [12] indicates that the Digital Signal Processor (DSP) extension of the M7 processor comprises hardware and instruction set enhancements that enable it to perform optimally in signal processing fields such as audio, sensor data processing, telecommunications and control systems. This is achieved by combining high-speed arithmetic, parallel data processing and specialised memory access patterns to accelerate arithmetic operations beyond the capabilities of the standard core. In addition to the previously mentioned optimisations, there is also wide support for a variety of data types, such as integers, fractional formats and floating-point numbers, which provides flexibility for fixed- and floating-point computations. DSP MAC instructions can multiply and sum values in one step, which is a frequent operation in algorithms such as digital filters or fast Fourier transforms (FFTs). Operations are parallelised using SIMD instructions, which allow multiple smaller data elements to be processed simultaneously. For example, two 16-bit or four 8-bit calculations can be performed in parallel within a 32-bit register. Overheads and bottlenecks are optimised using techniques such as loop unrolling and efficient memory access (e.g. using FIFO buffers to emulate circular addressing), which

significantly reduces the cycle count of processors without a DSP. These optimisations, which are embedded directly in the M7 core, remove the need for a separate DSP chip, keeping the overall cost of boards with these cores low and making them more accessible.

2.5.2 GAP-8

The main features of the GAP8 SoC are detailed in [13]. The overall structure is synthetized in Figure 2.3. Its architecture focuses on optimising local data processing, minimising data transmission needs and enabling low battery consumption, making it well-suited to the IoT domain. The GAP8 comprises a dual-domain system-on-chip fabricated using a 55 nm low-power CMOS process. It is formed by an energy-efficient microcontroller called the Fabric Controller, which has a computing cluster. Based on an enhanced RISC-V core, it can manage various general device operations and interfaces, directly exploiting all the on-chip peripherals, such as QSPI, I2C, I2S, the camera and PWM. This facilitates real-time, multi-modal data acquisition.
The cluster operates in different voltage and frequency domains and is made up of eight identical RISC-V cores, with the same architecture of the controller. These cores are supplemented by a hardware convolution engine (HWCE), which is dedicated to accelerating convolutional neural network (CNN) inference. Using a flexible shared memory, it can easily parallelise all normal NN tasks and speed up all common executions. The memory hierarchy is organised to enable easy memory parallelisation access by all eight cores and is supported by a multi-channel DMA system designed for low-latency data transfers. The system can rely on 512 KB of on-chip memory and expandable external memory to easily extend hardware capabilities according to model demands.
The system's software environment is built around GCC toolchains and offers native support for parallel programming via OpenMP. Developers can use explicit tools to automate memory tiling and scheduling, making it easier to manage the memory hierarchy. This approach facilitates the deployment of NN models by enabling them to be developed through high-level frameworks such as TensorFlow and by using explicit core generators for the target architecture. GAP-8 uses aggressive power optimisation mechanisms such as integrated DC/DC converters and deep retention modes. In deep sleep mode, power consumption can be reduced to as little as 3.6 µW, and peak computational loads can be handled with just 75 mW. Overall, this permits longer execution times using standard batteries.
The system architecture is designed to easily adapt to different workloads. Nearly linear acceleration is observed when parallelising across its eight cores. The observed benchmarks are impressive: the SoC achieves up to 10 GMACs for CNN computations at 90 MHz, with an energy efficiency of 600 GMACs/W. Common NN tasks, such as CIFAR-10 image classification, can be completed in around 650

µs at 15 mW. This enables the execution of 10 frames per second for almost two years on a single AAA battery, guaranteeing continuous, intelligent sensing at the edge. The system can efficiently optimise operations such as FFTs, matrix multiplications, convolutions and advanced signal analytics, using both SIMD extensions to the RISC-V cores and the HWCE accelerator to achieve 10x better energy efficiency than software-only execution.
Using this architecture, systems that were previously thought to be deployed in cloud environments can now be easily deployed on an actual hardware platform, achieving faster speeds for real-time scenarios and dramatically reducing energy consumption. This architecture is excellent for easily sensing at the edge and directly computing fast, typical NN operations.

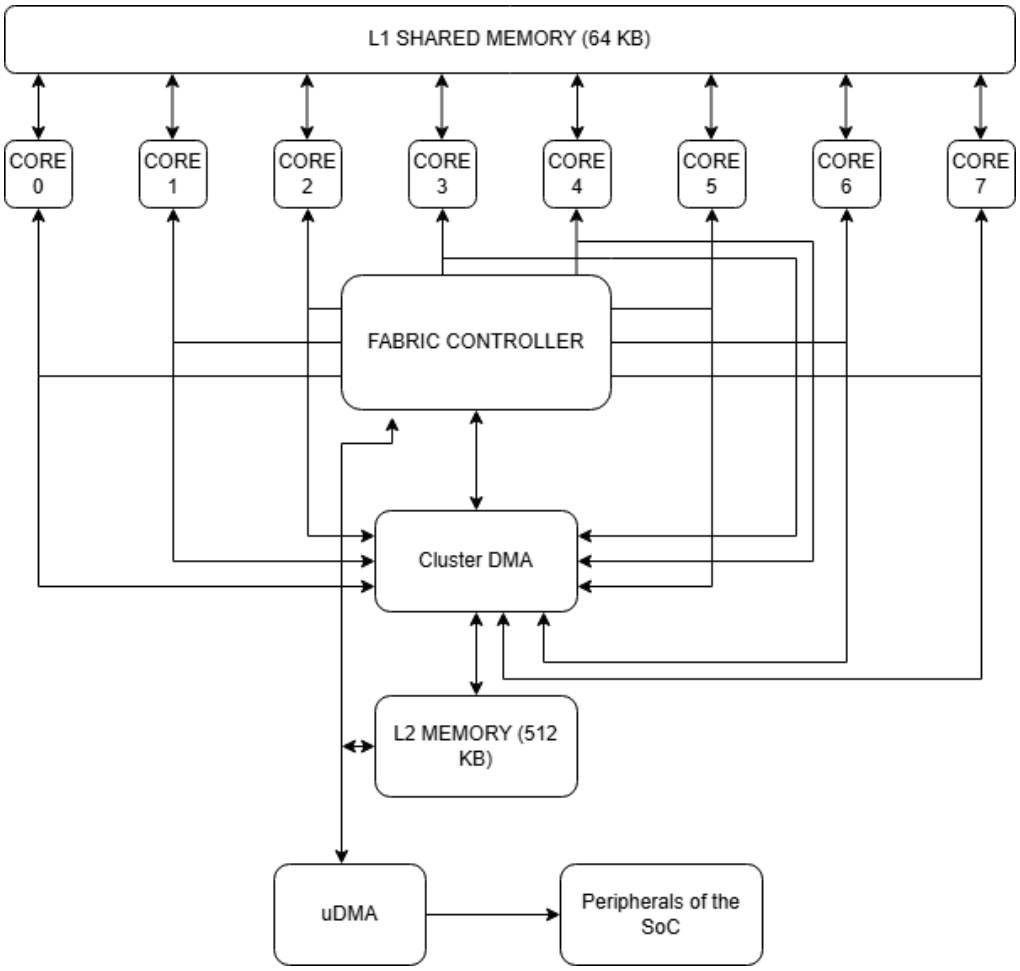

Figure 2.3: GAP-8 structure scheme

2.6 Tools to support NN at the Edge

In the previous section, some common hardware platforms were described to illustrate the characteristics that are useful for building Edge AI applications. These platforms also offer excellent support for the implementation of typical NNs through the use of ready-to-use dedicated tools and libraries: PULP-NN is used for the GAP-8 SoC, and the ST Edge AI Core is used for commercial ST boards.

2.6.1 PULP-NN

As described in [14], PULP-NN is an optimised, open-source software library designed for the efficient inference of quantised neural networks (QNNs) on multi-core, ultra-low-power RISC-V processors. Target architectures include the Parallel Ultra-Low Power (PULP) architecture and commercial implementations such as GAP-8. The library is suited to high-throughput applications with low power requirements. It supports aggressive quantisation using data types such as INT-8, INT-4, INT-2 and INT-1 for weights and activations. It contains optimized kernels, such as convolution, fully-connected, ReLU and max-pooling, which are tailored for reduced precision arithmetic. Binary kernels can be easily handled by exploiting the logical XNOR and bit-count instructions available in the instruction set architecture (ISA).
To drastically speed up typical neural network (NN) operations, the DSP instruction set architecture (ISA) extensions present in PULP-based RISC-V cores are used extensively, such as for SIMD operations (e.g. sdotp4 for vector dot products), hardware loops, and efficient memory access. The cluster of eight cores is also exploited to parallelise workloads and achieve a near-linear speedup in mathematical operations. The library uses the HWC (height-width-channel) data layout for activations and weights to optimise memory bandwidth and target the optimisation of operations such as matrix multiplication. im2col routines and custom packing/unpacking functions for sub-byte data are implemented to make the operation more compatible with SIMD units and minimise the memory footprint. These optimisations achieve very good results in terms of throughput, delivering up to 15.5 multiply-accumulate operations per cycle (MACs/cycle) on INT-8 kernels when running optimised matrix multiplication. The library outperforms common NN libraries for commercial platforms such as ARM CMSIS-NN, delivering a speedup of up to 63× over the baseline sequential RV32IMC and significantly higher energy efficiency and performance than STM32L4 and STM32H7 MCUs when operating at maximum frequency. The library also enables GAP-8 to run a CIFAR-10 network in 19.6× fewer cycles than CMSIS-NN on STM32H7, offering 14.1× better energy efficiency than STM32L4.
The open-source nature of the system ensures adaptability and makes it open to further updates to improve performance as technology advances. It also permits

the efficient development of real-time applications, especially compared to other commercially available tools. This system paves the way for the deployment of highly quantised networks in environments with limited resources, demonstrating that aggressive quantisation (even down to 1 bit) is feasible on programmable parallel architectures with minimal loss of accuracy, provided that proper training techniques are employed. While this system could deliver very high performance, it undoubtedly requires a high degree of expertise in the field.

2.6.2 ST Edge AI Core

The main feature of the ST Edge AI Core tool could be extracted from the datasheet provided by ST in [15] and the cited web page. This freely available desktop software suite, developed by STMicroelectronics, is used to optimise, compile and deploy artificial intelligence (AI) models on a wide variety of ST hardware platforms, including STM32 and Stellar microcontrollers and microprocessors, as well as smart sensors equipped with ISPU and MLC functionalities. The tool's primary function is to automatically transform pre-trained neural networks and classical machine learning models into highly optimised C code, facilitating the deployment of Edge AI applications from a standard model. It supports several AI frameworks, such as Keras and TensorFlow Lite, as well as any framework with ONNX export, such as PyTorch and MATLAB. This gives complete flexibility and compatibility for modern model workflows.
The tool further enhances computational efficiency by leveraging the generation of libraries that use dedicated hardware acceleration through the ST Neural-ART NPU wherever available. If the NPU does not support neural network operations, the tool seamlessly schedules these tasks to execute on the CPU, ensuring optimal use of resources. The software also enables users to assess RAM and flash usage during deployment, providing a more customisable environment. Optimisation options are provided to balance inference speed and binary size, catering for diverse application constraints. STEdgeAI-Core also features verification tools, allowing benchmarking and validation on host systems (Windows, MacOS and Linux) and actual ST hardware, ensuring reliability during edge AI development. The tool is accessible via a graphical interface and a command-line utility, and comes with comprehensive documentation to ease access and usage. STEdgeAI-Core is a key component of the ST Edge AI Suite, which aims to streamline every stage of AI deployment, from initial data management to final hardware integration, to create a robust, developer-friendly ecosystem. However, neither this tool nor the previously described one has direct SNN support. This is an issue that developers must overcome for their implementations with these tools to be as straightforward as standard NN deployment.

2.7 SNNs for widely available HW platforms

Following the analysis of commonly available, ready-to-use neural network (NN) tools for deployment on the edge, this paper will describe two solutions for developing spiking neural networks (SNNs) directly on widely available hardware devices. The first solution is Spiker+, which is specific to FPGA development, and the second is for deployment on RISC-V microcontrollers.

2.7.1 Spiker+

All Spiker+ features are fully described in [16]. It is a comprehensive framework designed to generate efficient hardware accelerators for spiking neural networks (SNNs) that are deployed on field programmable gate arrays (FPGAs) for edge inference applications. The framework is designed to efficiently integrate artificial neural networks (ANNs) at the edge for common critical real-time applications. The multi-layer architectures produced by this framework support both fully connected feed-forward (FF-FC) and recurrent (FC-R) SNN architectures, enabling the deployment of a large number of typical SNN models. The tool supports various Leaky Integrate and Fire (LIF) neuron model implementations (first- and second-order), with options for hard or subtractive reset mechanisms. Enhancements focus on minimising resource usage while maintaining computational accuracy. All network characteristics, such as network architectures, neuron types and input encoding schemes, can be described using a user-friendly Python interface. This makes the system more usable at a higher level of abstraction.
The hardware structure has been engineered to support high levels of both parallelism and modularity, while also ensuring a low-area footprint. The system utilises a start-ready handshake protocol for the purpose of synchronising and communicating between network components, thus optimising speed and power efficiency. The training framework utilises existing SNN training tools that support backpropagation through time (BPTT) with surrogate gradients. The process of optimisation and quantisation is also automatic, and is performed on neuron states such as synaptic weights and neuron parameters. This is done in order to balance accuracy and hardware constraints. The tool is also enriched by an automatic VHDL code generator with configurable memory storage options for synaptic weights. The system has been evaluated using two different datasets. In the first experiment, on the MNIST dataset, the feed-forward I-order LIF SNN was used, and Spiker achieved an accuracy of 93.85% with a classification latency of 780 microseconds per input image. This experiment also demonstrated that the system could consume only 180 mW of power on a low-end Xilinx XC7Z020 FPGA. In the second experiment, it was demonstrated that, using modest hardware resources, a compact and energy-efficient design suitable for resource-constrained edge devices could be achieved. In the context of audio classification, the Spiking Heidelberg Dataset (SHD) has

been found to be a particularly effective resource. Utilising recurrent architectures and II-order LIF neurons, the SHD has been demonstrated to achieve an accuracy of 72.99%, accompanied by a latency of 540 microseconds and a power consumption of 430 mW. This scenario encompasses a greater number of complex tasks, thus necessitating a greater quantity of memory and logic resources. However, it is notable that the results obtained demonstrate a favourable power-to-performance ratio in comparison to a significant number of contemporary FPGA SNN accelerators. It is evident that the system is capable of operating with minimal power consumption while achieving competitive levels of accuracy. It exhibits minimal utilisation of FPGA resources, thereby facilitating its deployment on compact, power-constrained devices. Additionally, it is equipped with a Python interface, which expedites the development of Edge SNNs. The impact of input spike activity and encoding schemes on power and latency is significant. Clock-driven update policies have been shown to be effective for small to medium-scale accelerators. Quantisation of neuron and synapse parameters has been demonstrated to have minimal impact on accuracy, but a significant impact on power savings. This facilitates larger implementable network sizes. However, it should be noted that memory capacity on the FPGA could be a limiting factor when implementing large and complex models. This tool is great for deploying SNNs on hardware, but it is limited to FPGAs, which are often expensive and generally require a high level of digital design expertise for deployment.

2.7.2 SNN decoder for Implantable Brain Machine Interfaces

All the details regarding this specific implementation over RISC-V microcontrollers are present in [17]. The system is a SNN designed as a neural decoder for regression tasks in implantable brain-machine interfaces (BMI). It is a low-complexity architecture comprising LIF neurons with trainable decay factors that are specific to each neuron rather than being uniform across the network. This design is flexible with regard to different temporal dependencies, which is an essential consideration for time-series neural decoding. The model uses 'spiking band power' (SBP), extracted from 96 channels, as the input feature. SBP uses a band-pass filter (300–1000 Hz) to reduce the data rate while retaining predominantly information from individual neural spikes — a choice that balances power efficiency and predictive accuracy. The architecture uses fully connected spiking layers, each containing 256 neurons, except for the output layer which predicts two-finger velocities. This process is handled entirely by neuron states, rather than exploiting any multi-step temporal convolutions, to optimise performance. Each neuron uses a custom reset-by-subtract scheme: whenever a neuron spikes, the membrane potential is decremented by the threshold value, giving the neurons more temporal

awareness. The network is trained using an enhanced spatio-temporal backpropagation (STBP) method. This method exploits trainable decay factors per neuron and noise injection before data normalisation for regularisation. Dropout is applied to the spatial dimensions during training (probability = 0.2) to combat overfitting. Batch normalisation uses threshold-dependent scaling and is fused into the weights and biases for deployment. Quantisation-aware training enables very low bit widths (down to 4 bits for the weights and 3 bits for the decays) with minimal loss of performance.
The input data used for training is prepared accurately to enhance the system's reliability and accuracy. Indeed, sequences are generated using a sliding window of 10 time steps and 9-step overlaps to increase the effective sample size via noise-based augmentation. The system is deployed on a RISC-V-based SoC (System on a Chip), the GAP9, and optimisations are made using DMA (Direct Memory Access) access for faster weight transfer and SIMD (Single Instruction Multiple Data) instructions to parallelise computations of neuron states and spikes. Distinct strategies for data movement and execution are applied to the input and spiking layers to minimise memory footprint (total storage of around 160 KB) and transfer overhead. When tested on two non-human primate benchmark datasets, the SNN achieved superior performance, with correlation coefficients of 0.783 and 0.624 for finger velocity prediction in datasets A and B respectively. This surpasses the performance of state-of-the-art Kalman filter (KF) and artificial neural network (ANN)-based decoders for offline inference. The system also achieved excellent efficiency results, consuming only 1.88 µJ of energy and 0.50 mW of power per inference via duty-cycled operation, with an average inference time of 0.12 ms. Using only 4-bit weights and higher-precision membrane potentials for stability has not resulted in relevant accuracy losses.
The results achieved here are excellent, but this work focused solely on one implementation using one target architecture, which makes it difficult to deploy different applications using the same optimisation approach.

2.8 Proposed solution

After analysing all state-of-the-art systems that exploit SNNs, their implementation advantages, and how they are used to optimise neuromorphic applications, this thesis will briefly describe the tools and approach used to solve some of the aforementioned solutions' problems, proposing a more standardised system that could be used to optimally deploy a simple SNN architecture over standard MCUs.
The system merges the strenghts of the previously mentioned tools to achieve:

- Optimised SNN execution on classic, affordable, commercial microcontrollers.

- Two different hardware systems are targeted to perform custom-specific optimisations for specific hardware architectures.
- The library is generated directly from the NIR description to provide greater flexibility for the proposed solution in different SNN applications for one of the target systems that uses LIF neurons.

Chapter 3

Materials and methods

After analyzing previous trials in this field of research and after analyzing the hardware where the system must be implemented, in this chapter, I present the work I carried out to develop the final system solution, composed of a ready-to-work SNN across different platforms and hardwares. In particular, the focus is on:

- SNN model development using the snnTorch framework, with training and testing.
- Deployment of the same model in an ST Board with optimized features.
- Deployment of the model into GAP8 hardware platform.
- NIR-to-C translator capable of translating NIR described SNN with a similar architecture as the proposed one into a .h/.c file bootable on the board.
- Only on ST Board, an half-optimised version of the Izhikevich neuron model.

The chapter contains the implementation details of the solution with all the tools used to make the system work.

3.1 SNN Model Architecture

In this first section, I will discuss about the model of neurons used in the architecture and on how the neurons are linked to each other to make the system work.

3.1.1 The model and its purpose

I decided to implement an SNN capable of making a classification of 7 different classes. The goal of such a model is to translate a signal input taken by a robotic hand when passing its finger over a Braille character [4].

The input comprises 12 channels from 12 analog sources. Each of these channels is monitored for temporal variations. When a channel's analog signal exceeds a predetermined threshold in either direction, a value of 1 is assigned to indicate a significant change. Conversely, periods during which the signal remains within the threshold result in the assignment of a value of 0, denoting stability. This threshold-based approach transforms the continuous analogue input into a binary sequence, where spikes represent moments of change and zeros indicate periods with constant signal values. This encoding method allows relevant dynamic information to be efficiently extracted from analog signals for subsequent classification tasks in the model.
From these 12 inputs ready to be computed, the system is capable of showing at the output through a single neuron spike what the braille character recognized by the robotic hand among 7 different values, which represent 7 different characters (classes), that corresponds to Space, A, E, I, O, U, Y.
So, the input of the neural network is composed of 12 different channels of binary values, the output is composed of 7 spike values, which are produced by 7 single neurons in a one-hot encoding style to identify the class that is recognized, the so-called output layer.
The architecture has also an hidden layer which is composed by 38 neurons, these neurons are directly fed by the 12 inputs in a all to all connection (fully connected), so each neuron receives at the input spikes from all the 12 input channels to compute a value and to spike, each neuron in the hidden layer has also a recurrent connection to itself, so whenever a neuron in the hidden layer spikes, it affects also the input of the next timestep. The output layer, instead, is only fully connected to the hidden layer without recurrences. Both layers are made by LIF neurons (Leaky-integrate and fire), with the only difference that the hidden layer has also recursive links.

3.1.2 LIF neuron structure

The LIF neuron used in this model is a first-order LIF neuron, which, in the end, corresponds to an implementation of the following equation:

$$V = V_{\text{reset}} - (V_{\text{current}} - V_{\text{reset}}) \cdot \beta + I_{\text{weighted}}$$

Where:

- V: Membrane potential at the end of each timestep (evaluated just before spike decision)
- $V_{\mathbf{reset}}$: Resting membrane potential value after neuron reset
- $V_{\mathbf{current}}$: Current membrane potential value at timestep T

- β: Decay factor constant computed as:

$$\beta = \exp\left(-\frac{1}{\tau}\right)$$

  where τ is the membrane time constant, ensuring $0 < \beta < 1$

- $I_{\textbf{weighted}}$: Weighted input sum calculated as:

$$I_{\text{weighted}} = \sum_i w_i \cdot x_i(t)$$

  where w_i are synaptic weights and $x_i(t)$ are input spikes at time t

This could, in general, be done when the R resistance of the neuron is equal to 1 ohm, so the value of current in Amperes and Volts is numerically the same, and, in the end, the weighted input accumulated in the equation is a tension with the same numerical value as the current.
The spike decision is made by a comparison against a particular membrane value denominated $V_{threshold}$, and in our so-called reset to zero implementation, the spiking behaviour is the following:

$$S = \begin{cases} 0, & \text{if } V < V_{\text{threshold}} \\ 1, & \text{if } V \geq V_{\text{threshold}} \end{cases}$$

And whenever the spike occurs, the value of the membrane is set to 0 from the next timestep after a spike.
So, if my implementation is reset to zero, we can write an easier implementation of the previous equation, which consists only in:

$$V = (V_{\text{current}}) \cdot \beta + I_{\text{weighted}}$$

For this equation remains the consideration on the R resistance of the neuron equal to 1.
This is the no recurrent variant of the LIF; the recurrent one is slightly similar to this one, but in my implementation, each neuron of the hidden layer also has another input generated by the spikes of the same neuron on itself. So, in the recurrent one, each neuron has a specific link to itself with a specific weight to have also more temporal memory through the computation of the samples. The typical LIF behaviour is schematized in Figure 3.1.

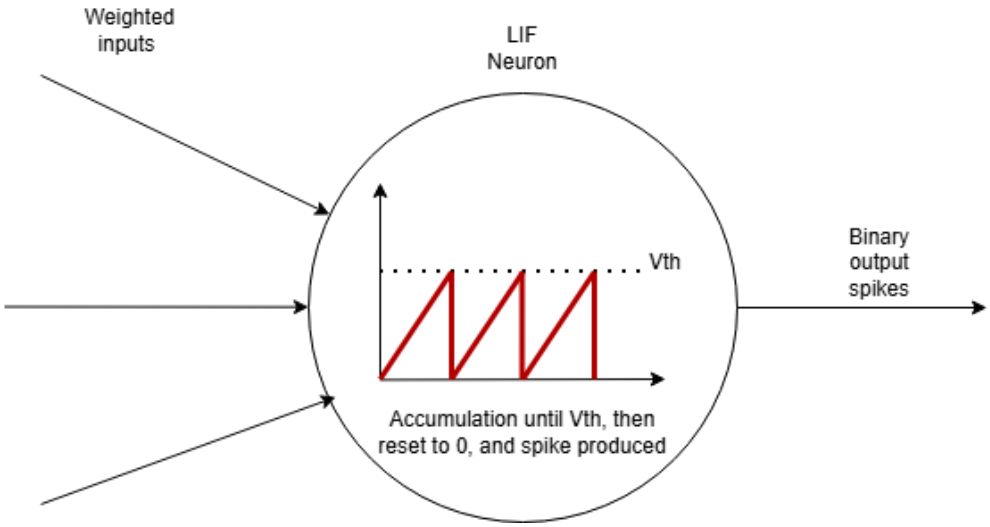

Figure 3.1: LIF Behaviour scheme.

3.2 SNNTorch Implementation

In this section, I will discuss about how I have implemented the previously discussed SNN structure using the framework SNNTorch.
First of all, i've identified what are the models of neurons in SNNTorch, which I can use to make the system work as expected. I found out that for the hidden layer, I can use the **RLeaky** neuron and for the output layer, I can use the **Leaky** one with the following settings:

- **RLeaky**: reset mechanism = "zero", all to all = false, reset delay = false.
- **Leaky**: reset mechanism = "zero", reset delay = false.

For all other settings of the network, I've taken some values from pre-existing networks already implemented using another kind of neurons (Section 2.2.1), and I will go into it in the training subsection.

3.2.1 Training and Validation of the model

To make the whole network work properly, I had to train the network to assign the right weights to maximise the correct class predictions.
So, I have used three different files of tensor datasets, one for training, one for validation and one for testing, already used in another previous work [4].
Together with these files, I have used fixed values for betas and for thresholds from the beginning of the training, beta1 = 0.90, beta2 = 0.55 and the threshold equal for all neurons equal to 1.0.

The following paragraphs explain the steps required to train a spiking neural network, adapting it to the specific custom model that I must train to compute the right weights of the network.
In my model, I need two weight matrices, one for the fully connected input-hidden layer stage and one for the hidden layer-output stage, always fully connected. I need another vector of weights for the recurrent connections, which is composed by 38 values in a 1 to 1 binding with the hidden layer structure (1 weight for each layer).
For the normal weights matrix, the system already uses initial values of weights following these equations retrieved from SNNTorch Documentation:

$$W \sim \mathcal{U}\left(-\sqrt{\frac{1}{\text{input\_channels}}},\ \sqrt{\frac{1}{\text{input\_channels}}}\right)$$

So, I have also used this kind of initialisation for the vector of recurrent weights because this was not done in the previous work implementations of training.
After these elements are set in initialisation, the training loop uses these elements to discover the right weights to maximise the right class prediction given the input samples:

- **Lost Function Selection**: To make the network aware of temporal integration patterns, a cross-entropy loss is applied to spike count over time so the system doesn't rely on instantaneous spike events.
- **Gradient Handling**: The fast sigmoid surrogate gradient function can provide smooth gradients during the backpropagation steps and mantains hard thresholds in forward steps, ensuring correct behaviour of the network.

So, the weights after each training loop are updated following this approach:

- A clear of previous computed gradients is performed.
- A store of the new gradients is made.
- An optimisation of weights is made using Adam optimiser.

After the training loop, validation loops are performed, where the model is rerun, at each epoch of training and, using a different dataset for validation, is able to perform the loss computation using the same weights as that training loop, and at each step also accuracy is also computed.
In the case of Validation, no actions were made on weights, only loss and accuracy computations.
In the end, the dataset test is used to perform the testing over the best values of weights among all the epochs, which is chosen by seeing the epoch where the net has the best validation accuracy among all. In our case, the epochs are 500.

3.3 SNN for ST Board (snn2mcu)

After the developement of the SNN model using SNNTorch, in this section is described how that model is imported into a .c implementation which can be executed by an ST Board.
The target board which i've used is an STM32H757I-EVAL, and the developement has focused on three main steps of work:

- Generation of initial setup code using cubeMX software.
- Realisation of a custom SNN model which mimics one-by-one the behaviour of the SNNTorch model using some optimisation to speed up some operations.
- Print some outputs of the model using UART peripheral of the board first, and then print also on the LCD of the board to see the results directly from the board itself.

3.3.1 Initialisation of the environment

First of all, to ease the initial setup of all the peripherals which I've used to perform all the SNN operation I've used the cubeMX[1] program to setup all the hardware peripherals and to setup the clock frequencies of those peripherals in a way that the operations can be coherently executed and then the outputs can be correctly shown.
So, in the first place, I've started a new project from an ST Board, STM32H757I-EVAL is chosen, and then the initial config is chosen at first. Together with the default settings, usart1 peripheral is enabled in my project, so I can track and see all the operations done by just setting up a UART receiver using PUTTY. The settings used for UART are:

- 115200 bits/s for the Baud rate.
- 8 bits of word length without parity bit.
- 1 stop bit implementation.

Then, for all other peripherals, the default settings are maintained so a generation of the code to be opened by STM32cubeIDE is executed.
At this point, I want to use the full optimisations to make a faster model and program the LCD peripheral more easily. Some libraries provided by ST must be imported, even though they are not directly included in the CubeMX project. In

[1] `https://www.st.com/en/development-tools/stm32cubemx.html`

particular, I've imported the DSP[2] and NN extensions to unlock the full optimisations over the mathematical operations of the SNN, and I've also imported the BSP[3] functions to handle the LCD configuration with some utilities to operate with the same device.
To import, a path to the include directories to the respective libraries is explicitly pointed out to the GCC compiler, among the others already present for the common CMSIS and HAL libraries, which were standard generated by cubeMX. Then, the source code for the NN is directly included in the project in the common folder, for DSP, instead, I've used the already given .a files so the libraries can be imported directly from there.
For the BSP, a set of files containing the useful libraries and peripherals for performing LCD operations is imported, based on an existing example.
And after that, all the peripherals used by the SNN are ready to use.

3.3.2 SNN execution and optimisations

At this point, the implementation on the board could be realised. The core of the SNN is described through two files (.h/.c), which contain neuron structures, weight vectors and functions that are used to initialise the network, perform simulation steps, display the results of the operations and clear the neural network status. This allows independent classification of samples in a single run, for example.
The main function, apart from the initialisation functions generated by cubeMX, can perform the following operations on the SNN, calling the respective functions:

- The initialisation function to load the model on memory.
- The function to give the correct input vectors before each simulation timestep.
- the function to clear the SNN status if we want to give other independent samples to the network.

Data types used

The main feature of this custom network is the use of some particular variables on 8 and 16 bits. The **q7_t** type of variable is an 8 bit variable, which I use just to give/take/propagate spikes through the network, so I can represent spikes by only writing ones on this 8 bit values instead of using 32 bit variables, which could result in a lot of wasted bits unused. The **q15_t** is instead used to represent weights, perform operations using them, represent membrane status and other neuron features like the threshold voltage and decay factor. The q15_t is on 16 bits, can

[2] `https://arm-software.github.io/CMSIS_5/DSP/html/index.html`

[3] `https://github.com/STMicroelectronics/stm32h747i-eval-bsp`

represent only numbers in the range of [-1, 1], these are effectively integer numbers, but represent in a quantised manner all the values between -1 and 1 representable using 16 bits, the first used only to show if the number is negative or positive. The differences from the normal floating-point data type are emphasised in Figure 3.2. The absence of an exponent field means that q15_t and q7_t can only represent values between -1 and 1. Fewer mantissa bits also mean that less precision can be represented overall.

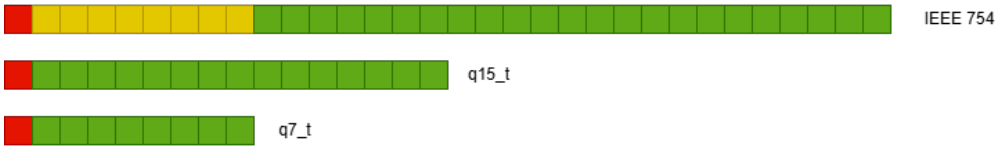

Figure 3.2: Comparing data types, standard, fixed q15 and fixed q8. Red is sign, Yellow is exponent, Green is mantissa.

Neuron structure and optimised execution

Other than that, this type of data are useful also because the DSP extension of CMSIS has some functions optimised to perform operations using these variables. The neuron in this implementation is a C struct; it has 4 different values represented in q15_t, which represent the threshold, the reset value, the membrane potential and the decay factor (the beta in the equation). Now I'll go into a deeper explanation of the code in this particular case, in the NIR Chapter 3.6, instead, there is also a description of what could be generated and how the code is modified accordingly to different situations.
The main functions of the SNN implementation on board are:

- **usart1_print**: Takes a string and uses HAL function to transmit over usart1 peripheral of the board to be read.

- **print_float**: Takes a floating number in input, the number is converted in character strings to be then represented in the right way on uart and then calls usart1 print to print the value. This function has been used mainly to perform debug prints in order to check the specific neurons behaviour after the cycles of simulation. I left the implementation there so anyone could also check the behaviour of the network.

- **LIFNeuron_init**: Takes in input a pointer to neuron struct, the threshold and the reset values. So, the neuron variables are initialised.

- **LIFNeuron_Layer_Update_Vectorized**: This function is one of the main ones used for performing neural operations within a layer. In particular, it is used for recurrent layers and takes the following as inputs: a pointer to the layer (an array of neurons), the input spikes, the weights vector, the number of inputs, the number of neurons in the layer, the recurrent spikes and weights, and a flag variable that indicates whether the function is one-to-one. After the operations, the function fills the output array, which is passed to the function as a pointer, so that the results can be displayed or propagated.

  Firstly, all the neuron parameters are extracted from the structs into different arrays to perform operations later. The weighted input accumulator is initialised and filled with zeros. This is used to update the values of each neuron based on the inputs received by each neuron. Each entry corresponds to the input received by a specific neuron at a given timestep. By checking the flag, we can see if the layer is fully or one-to-one connected to the previous layer. This allows us to use the arm_add_q15 function in two different ways.

  - In one-to-one, each input spike is checked in a for loop, but only one entry is updated each cycle time. Therefore, even though this function could perform vectorised operations, this is not exploited here.
  - If fully connected, all the entries in the weight vectors referring to that specific input neuron are added to all the entries of the weighted_input matrix in parallel. This means that there is only one loop at the inputs, and vectorised accumulation is exploited using the vectorised function. This is possible because the weights in the vector are ordered according to the pattern of inputs to neurons (in0->n0, in0->n1, in0->n2, ... in1->n0, ... until the end).

  The presence of values in recurrent inputs and the recurrent matrix is checked, and if any are found, the update follows the one-to-one approach like it was described above.

  After these accumulations, the other operations are standard for all the neurons in the layer following the update functions already described in previous chapters, so variations of the add function are exploited: arm_sub_q15, arm_mult_q15, the latter for decay computation.

  In the end, the current membrane value for each neuron is compared with the threshold value; if above, the membrane is reset, and a spike is generated; if not, a 0 is generated in the output spike.

- **LIFNeuron_Layer_Update_Vectorized_NoRecurrent**: Same exact behaviour as the previous function, but has no recurrency checking or update, used for no recurrent layers only.

- **Load_NIR_Weights**: This function is used to initialise the Weights of the network, and then these are scaled to a smaller values to prevent overflow on using q15_t variables. Then, from floating point (float), these values are converted into ready-to-use q15_t values to match the standard of the custom architecture.

- **SNN_Init**: This function is used to initialise the networks. Values for threshold, reset and betas are chosen, scaled with the same scaling factor as the weights and then the LIFNeuron init is called for all the neurons of the layer. The scaling factor is not applied to betas for two reasons, the first is that is not necessary given the fact that are always numbers < 1 but the other reason is that the betas must be multiplicated to the current membrane value to perform the decay, if scaled, the beta multiplication becomes no more consistent with the no-scaled architecture, because the scaling works fine and without problems only if accumulation or subtractions are performed. During multiplications with both membranes and betas scaled, the result of the multiplication is no longer equivalent to the normal network behaviour. After the neuron initialisation, the Load weights function is called here, and all the spikes for recurrent connections are set to zero.

- **SNN_Run_Timestep**: takes as input pointers to input and output spikes, performs in order the execution of the operations in the layers using the update functions, propagates the spikes from one layer to the other and gives also back the spikes for the recurrent layers.

- **SNN_Reset_State**: this function resets the state of the network, putting all the membranes to reset values and spikes to 0.

3.3.3 LCD implementation

All the results of the SNN execution are printed in some functions in the main. Here, we will focus on how the functions are used to print something on the LCD before and after the neural network execution.
First of all, using MPU_Config() function, the MPU attributes are configured as Write Through for SDRAM, then SDRAM and UART are initialised. Then LCD is initialised, parameters of LCD are configured as the active layer, the pixel format and the resolution of the LCD of the board. After all the init steps to initialise the LCD, two custom functions are used to print the initialisation of a simulation and the end of the simulation. At this point, the use of UTIL function is exploited to easily set fonts, write strings, and set text and background colours to read results directly from the board.
All the stuff related to results and optimisations is deeply described later in the results chapter.

3.4 Izhikevich on ST Board

To avoid focusing solely on one neuron implementation, the Izhikevich neuron model is also supported. This model is slightly less optimised due to its more complex nature. Firstly, a brief description of the neuron model will be given, and the differences from the previous ST implementation will be emphasised at the end. This model is only developed for the ST Board.

3.4.1 Izhikevic neuron model

The Izhikevich model is widely recognised as a SNN neuron model for its biological plausibility and computational efficiency. In this implementation, each neuron is defined by a set of state variables and parameters, namely the membrane potential (V), the recovery variable (u) and the parameters (a, b, c, d) that determine the spiking and adaptation dynamics. The discretised form can be described as follows:

$$V = V_{\text{current}} + \Delta t(0.04V_{\text{current}}^2 + 5V_{\text{current}} + 140 - u_{\text{current}} + I_{\text{weighted}}$$

$$u = u_{\text{current}} + \Delta t \cdot a(bV_{\text{current}} - u_{\text{current}})$$

And the spike behaviour is the following:

$$S = \begin{cases} 0, & \text{if } V < V_{\text{threshold}} \\ 1, V = c, u = u + d & \text{if } V \geq V_{\text{threshold}} \end{cases}$$

The parameters used in this equation means:

- V: Membrane potential at the end of each timestep (evaluated just before spike decision)
- $V_{\textbf{current}}$: Current membrane potential value at timestep T
- Δt: is the discretization timestep set to 1 for simplicity.
- a: Time scale of the recovery variable u; typically determines how quickly u responds to chaingings in membrane potential.
- b: Sensitivity of the recovery variable to the membrane potential; controls the coupling between u and V

- c: Post-spike reset value for membrane potential (V); sets the new voltage immediately after a spike.
- d: Post-spike increment for recovery variable (u).
- $I_{\textbf{weighted}}$: Weighted input sum calculated as:

$$I_{\text{weighted}} = \sum_i w_i \cdot x_i(t)$$

  where w_i are synaptic weights and $x_i(t)$ are input spikes at time t
- $V_{\textbf{threshold}}$: It is the voltage threshold which is used to spike like in LIF model.

This has resulted in a different approach to optimisation starting from previously discribed model for LIF neurons.

3.4.2 Izhikevich for ST Board

The main difference between this model and the previous one is that only the weights and spikes are quantised here to optimise the operations. This is because the Izhikevich neuron's more complex state update dynamics require more precision than the q15_t datatype can provide. I kept the float32_t datatype because this datatype is also supported by the DSP ARM Extension, ensuring that operations are always performed in an optimised way.
The main differences in the code are as follows:

- **IzhikevichNeuron_Init**: In this function, the neuron is initialised with all the parameters which are different from LIF (a,b,c,d,v_init and u). u is initialised as b*v_init.
- **IzhikevichNeuron_Update**: now in this implementation, the neuron update state is done here, right after weight accumulation, exploiting the f32 variants of the arm CMSIS DSP functions (arm_mult_f32, arm_add_f32, etc.)
- **IzhikevichNeuron_Layer_Update_Vectorized**: It behaves initially like before for normal weight accumulation, supporting also recurrency, then the weight accumulated is translated into the float32_t datatype in order to make weight accumulation compatible with floating computation of the update function.

For the rest, it follows the same scheme of execution supporting also recurrency for each node, like in the previous LIF implementation.

3.5 SNN for GAP8 processor

After the hardware board implementation, the development of the network on another kind of platform has been done.
The target architecture now is not a single-core architecture, so a multicore approach is exploited, and also the memory management is optimized with smart accesses to L1 and L2 shared memories. Together with that, the operation of the program and of the network are executed exploiting the 8-core cluster to speedup all the initialization and execution processes.
The project in this case is not executed on physical hardware; instead, the program is executed on the GVSOC simulator (GreenWaves[4] Virtual System on Chip), which is a virtual platform that runs upon Ubuntu Operating System and is used to simulate the instruction set architecture of a specific SoC with GAP8 RISC-V processor.

3.5.1 Program flow and Implementation

The target model is always the same as the other platforms, the Braille classificator, but on this specific implementation, I've exploited all the extra features of this platform to enhance the SNN performances.
From now on, I will describe each of the functions present on the program to emphasize how the cluster implementation and the memory optimizations are performed:

- **NeuronOpt**: a C struct with floats potential, threshold, reset, and alpha (the beta decay factor already pre-computed), and an int spiked, which is used as a flag for spikes.

- **LayerInstanziationOpt**: a C struct with the identification int like the layer id, the number of neurons in the layer, the number of inputs to the layer, a pointer to the array of neurons, a pointer to output, and a pointer to input values.

- In Main, the **pmsis_kickoff()** function is called, responsible for kernel initialization, all SoC hardware setup, Cluster preparation, and launches the main function, which is passed here as an argument.

- The main function for the network initialization and execution is **neuron_instanziation_opt()**, here the cluster is initialized, then configured and opened. If there are no problems, the system is correctly initialized. Then the initialization phase begins, so each layer is initialized one after the

[4] `https://github.com/GreenWaves-Technologies/gap_sdk`

other in an L1 buffer, and later the data is sent to L2 using the DMA acceleration of the SoC. Then other operations like neuron initialization, weight assignment, and execution happen here, and the classification and testing of this specific model are also done. The general functions to make the entire system work are now described.

- **pi_cluster_send_task_to_cl()** is the function used in the main to send to the 8 cores cluster all the functions that we want to execute in a optimized way, each basic function of the SNN is called by pi_cl_team_fork(), a function call inside the specific cluster_delegate functions, and this is used to select the specific core that should manage a particular task. In my implementation here, I send the task to all 8 cores to optimize, and further decisions are made to keep data coherent among the system and to optimize the speed.

- **cluster_layer_init_opt()** is the main function used to initialize all the neurons inside a layer and the layer itself. Thanks to the previously mentioned code, all the 8 cores in a parallelized way access this function, and to keep track of the operations, the core id of each core and the total number of cores are stored in variables. Using a Round Robin scheduling made by a for loop and using core IDs and total number of cores in a wise way, a parallel initialization of neurons is made, so 8 neurons per time could be initialized, and if in the cluster there are more than 8 neurons, the remaining ones are initialized in the next cycle of the for loop. The neurons assigned to the cores in this way could be initialized independently wisely and coherently. In the for loop, the **initialize_neuron_opt()** is called to initialize the layer and the neurons inside it, setting features like threshold, reset values, and so on in a similar way to the other implementations. Here also the output of each neuron is set to 0. The result is a 1-to-1 matching between the neuron number of the layer and the core assignment (core 0 initializes neuron 0, core 1 initializes neuron 1, and so on, after the seventh, the 0 is assigned to the eighth... until all the neurons are initialized). As previously mentioned, the init phase of layers is on L1, so a pi_cl_team_barrier() is called to wait here until all neurons are initialized. Now the L1 initialized buffer is moved using the DMA to the L2 respective variables. This is done in a parallelized way similar to before, but in a one-only operation, the core that has initialized the specific neurons is also the one that here performs the DMA transfer to keep the data coherent, and without using for loops, the DMA transfer is called independently by each core in a parallelized way. A barrier is present after DMA functions to stall ready cores until all of them end their operations. In the end also the layer is also sent to the respective L2 variable only using core 0, so the operations are safe.

- Following a similar approach, the **cluster_weights_instanziation_opt()** function is called; here, the initialization of the weights is done only using core 0, the initialization is done in flash memory because the data is huge, but then a transfer to L1 buffer is done in the initialize_weights_opt_L1 function. A DMA transfer is performed to store the weights in L2, as the weight data is typically too large to fit entirely in the L1 buffers. This process is carried out in parallel, following the approach of initialising the neurons, whereby each core is assigned specific weights and sends them independently to L2.

- After the entire network is initialized, a function to reset all the neuron states could be useful, so **cluster_reset_neurons_opt()** is called by only using core 0, and here L2 data is moved to L1, then the state is reset, then again moved to L2.

- So now the network is ready, only core 0 each timestep gives to the network the inputs using cluster_load_input_dma(). In a first implementation, also here the data is managed by DMA, but, because the input data could be made by huge datasets which could not fit in L1 or L2 memories, just straightforward accesses to the input data are made.

- With the input data loaded, the execution of the network is done using **cluster_simulation_opt()** function. This function behaves almost like the other, but the operations are controlled in a way to parallelize safe operation and to keep the execution only on core 0 if there could be unsafe operations to keep data coherent. Here, core 0 first of all transfers neuron states from L2 to L1, here in a wise way from the weight matrices from L2, seeing active neurons using global variables, only the weights that came from active inputs are stored in the L1 buffer, so, on weights, only necessary data from L2 is taken each timestep to perform actions on neurons. After a barrier, all the neurons are now executed in a parallelized way using all the 8 cores, buffers on L1 are used to speed up the operations with the prefiltered weights. The neuron assignment to perform the actions is done as the other pre-described functions. So, the accumulation of weights is done, and the LIF dynamics are applied like in the other network implementations. Then the threshold checking is done to determine whether to generate spikes or not. If there is a recurrence in the weights accumulation phase before the LIF update dynamics, the recurrent buffer is also used with the other default one.

- This is all the execution of the network, eventually it could be performed classification recordings or other types of output management accordingly to the specific implementation. In the end, the cluster is closed, and the exit function is called.

3.6 NIR-to-C translator

In the end, after all the network implementations among different platforms, a Python program capable of taking as input a .NIR description of the network and giving as output the full working .c/.h network compatible with the ST device is realised.
The process of making the translator has been done in two steps:

- Translation of a known model into a NIR description from SNNTorch.
- Developement of the NIR-to-C translator.

3.6.1 From SNNTorch to NIR

First of all, the SNNTorch of the previously described network is taken.
Theoretically, by just using an export function from the NIR Python library the translation of the network is straightforward. But in this specific case, using the recurrent connections, the RLeaky neuron is not compatible with the export tool, as that specific neuron is not supported by now. So, a manual implementation of the SNNTorch network in NIR has been described.

- Weights, betas and thresholds are taken from the SNNTorch description, a nir dictionary of nodes has been made to set all the network characteristics, input using nir.Input, Output using nir.Output, the connections between layers using nir.Affine (even though no bias, for easier further improvements) and the layer neurons are described using the primitive nir.LIF.
- The parameters for the LIF are specified, from the standard, the tau from the beta is used to describe the decay behaviour, V threshold is inserted as the other networks, V leak is set as zero because we don't use it in our implementations, V reset to zero, and membrane resistance to 1, to mimic other network behaviour.
- The recurrent layer is made as an affine, which is inserted between the outputs of the layer and connected back again to the same layer.
- This is a custom implementation, a way to describe the recurrency, and the matrix of the affine is here a diagonal one to mimic the one-to-one recurrency.
- The edges are described using input, output, lifx (x, the number of the layer), fcx (to determine the connections among layers) and recx inserted on the edge graph like before to determine a recurrency in a specific layer.
- So the nir graph has been made using the specific primitive and the nodes dictionary, and the edges graph is passed as arguments, and consequently a .nir file is generated.

- The names of the edges are important; indeed, they are used to generate the C code later in the translator.

3.6.2 From NIR to C

The translator supports only LIF neurons; one-to-one or full connections between layers are also supported. The recurrency is supported on the layers, but only the one-to-one implementations; bias is not supported, v leak is not used, and the r must be 1 (in any case, the translator ignores this value). So, a list of supported characteristics has been made.
The nir description could be given by modifying the code or as an argument from the command line. If something is not supported, an error message is shown during code generation.

- As the class of the generator is initialised, the input nir graph is read, an output prefix for the name is inserted as a variable and a scaling factor is inserted (60 by default).
- Then, the nir network is analysed, exploring the edges, and an adjacency list is made. After that, as the first thing, the input node in the node dictionary is searched, and the value is put into a variable.
- From Input, the number of inputs is extracted.
- From input, all the edges are explored, and a tracking of the visiting nodes is also performed to prevent infinite loops (important for the recurrent nodes).
- So using adjacency list, each node is explored, and an affine node variable is assigned if a node is an affine, or a lif node variable is assigned if a node is a lif layer. In this first phase recurrency is ignored, and the node is marked as visited.
- In the end of the current node analysis, if a rec is found on the next node, recurrent weights are stored, a flag is set to true, and a checking on if it is diagonal has been made. If it's diagonal, then the description is accepted; if not, an error is raised.
- Then info from the layer is extracted, also if it is one-to-one connected or not, for normal Affines, biases different from 0 are just ignored, and layer info is stored as a dictionary style.
- Then, if all the values across the layer are the same for all the neurons in the layer, a flag is set to true so the code generation is optimised for uniform

values; if not, later the code is realised to generate values for each neuron, increasing the lines of code generated in the C file.

- From now on, checking on the written dictionary, C code is generated in a wise way by appending strings.
- The .h is firstly generated and is always the same; the code checks the network characteristics to adapt the generated C code to the network behaviour.
- The output here is pretty similar to the one previously described for the specific implementation and dynamic number of outputs, inputs, neurons and layers are allowed, but, in the end, only similar architectures are allowed to be generated directly from NIR.
- It is important that the NIR structure uses the specific structure present in the previous subsection because only with that format, the translator is able to translate.
- The last thing generated is an example C code to run the network, which must be embedded in the main code for the ST board.

3.7 Used tools

The development part of this work has been realized by using professional tools that are widely used. In particular:

- Firmware Development: STMcubeMX to generate base .h/.c code firmware, STM32CubeIDE
- Python framework snnTorch and NIR library.
- GAP8 implementation using GVSOC simulator for Ubuntu Local Fossa.
- Version Control System: GIT

Chapter 4

Results and discussion

Ultimately, I ran tests and experiments using all available tools to evaluate the impact of the optimisations on various SNN models.

Firstly, I measured the accuracy of the aforementioned Braille model on the same dataset using three different platforms: SNNtorch, ST Board, and GAP8, to compare the performance of the different architectures.

Then, using the NIR to C parser I had previously created, I reproduced an existing SNN model used to recognize images from MNIST Dataset for training, model definition, and later I deployed it on the board, passing through NIR and analyzing execution speed and energy consumption. The energy and time per inference of this implementation have been compared to a similar implementation on FPGA.

4.1 Braille Model Results

This first section will present all the results of measurements made using this model across the three different platforms. All of these models' characteristics are fully described in Chapter 3.
Firstly, two examples of the behaviour of the output layer will be shown, with a comparison between the snnTorch model and the ST Board implementation. Secondly, a comparison has been made between the three platforms to emphasise the differences when running a dataset of 140 samples named ds_test on all the platforms.

4.1.1 LIF neuron Behaviour: snnTorch VS Board

As previously mentioned, two different sets of spikes were given to the ST model and SNNTorch to test the LIF neuron behaviour, mainly to observe the effects of quantisation, performing random operations on the two models.
As mentioned in Chapter 3, the ST model uses weights trained in snnTorch and

exploits the same SNN behaviour to maintain coherent results at the output. The first spike pattern has ten timesteps. For each timestep, a new input value is given to the input layer to analyse the membrane behaviour in the output layer. The inputs are essentially arrays that form a matrix when the dimension T of the timesteps is included. The values are as follows:

Table 4.1: Input1 Pattern Across Timesteps (Neurons 0 to 11)

Timestep	n0in	n1in	n2in	n3in	n4in	n5in	n6in	n7in	n8in	n9in	n10in	n11in
1	1	0	1	0	0	0	0	0	0	0	1	0
2	0	0	0	1	0	0	0	0	0	0	0	0
3	0	0	0	0	1	0	0	0	0	0	0	0
4	0	0	1	0	0	1	0	0	0	0	0	0
5	0	0	0	0	0	0	1	0	0	0	0	0
6	0	0	0	0	0	0	0	1	0	0	0	0
7	0	0	0	1	0	0	0	0	1	0	0	0
8	0	0	0	0	0	0	0	0	0	1	0	0
9	0	1	0	0	0	0	1	0	0	0	1	0
10	0	0	0	1	1	0	0	0	0	0	0	1

Now, graphs will be displayed showing the differences in membrane values between the two implementations, giving also reasons behind this specific behaviour.

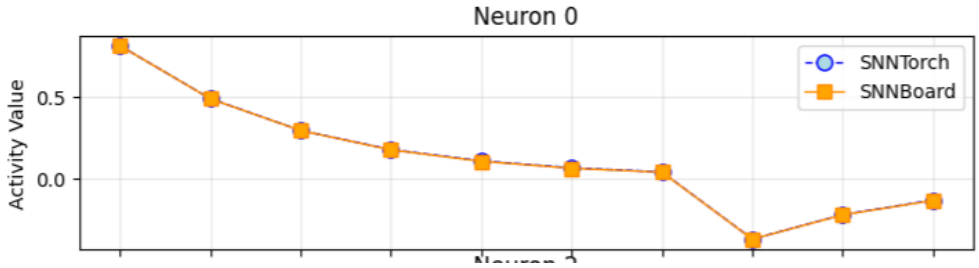

Figure 4.1: N0 behaviour for Input1 pattern across the timesteps.

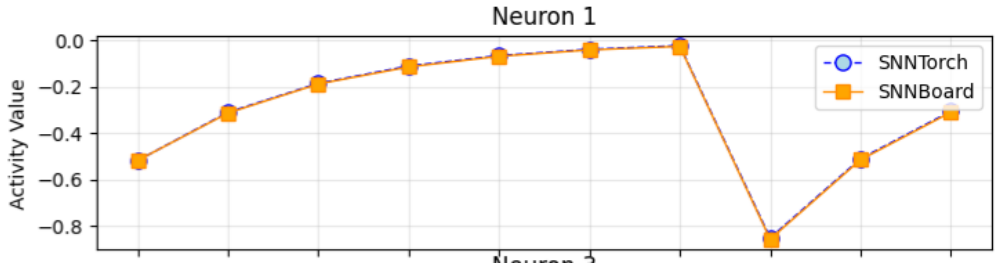

Figure 4.2: N1 behaviour for Input1 pattern across the timesteps.

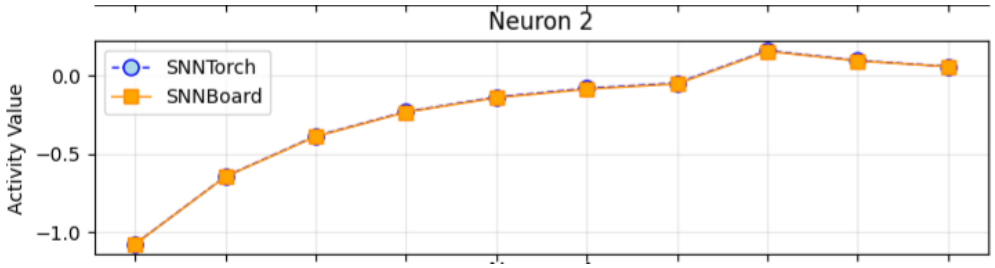

Figure 4.3: N2 behaviour for Input1 pattern across the timesteps.

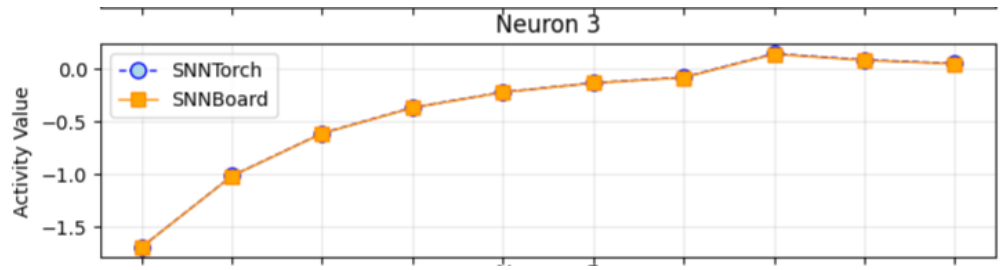

Figure 4.4: N3 behaviour for Input1 pattern across the timesteps.

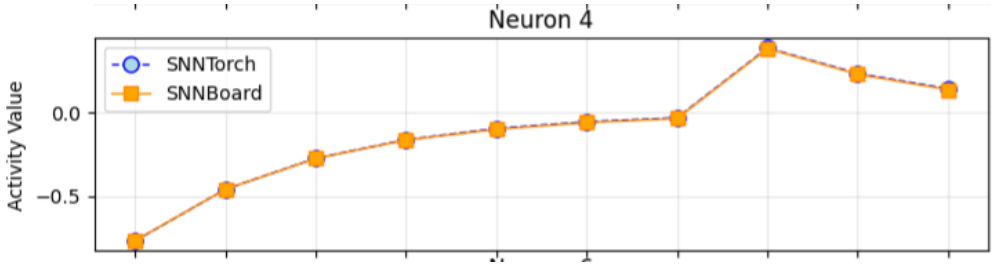

Figure 4.5: N4 behaviour for Input1 pattern across the timesteps.

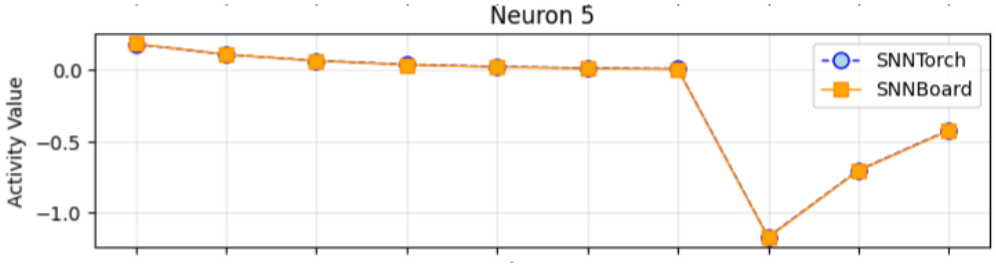

Figure 4.6: N5 behaviour for Input1 pattern across the timesteps.

Analysing the graphs shown in Figure 4.1, Figure 4.2, Figure 4.3, Figure 4.4, Figure 4.5, Figure 4.6, Figure 4.7, it can easily be seen that, even though the values in the ST Board are quantised, the results in terms of membrane values are almost the same. Indeed, the only difference is that the values from snnTorch are slightly

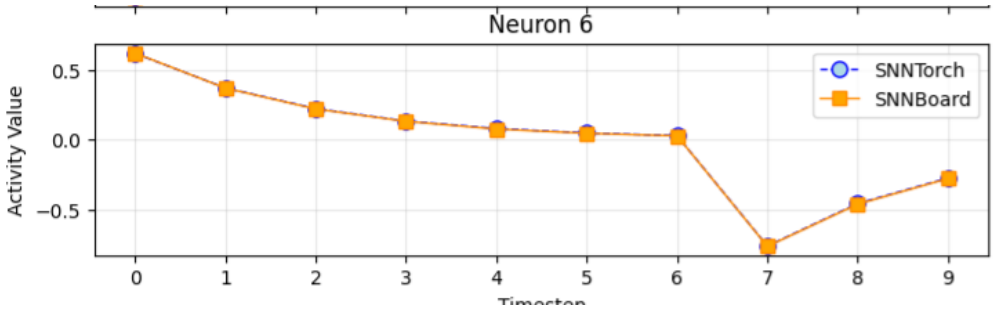

Figure 4.7: N6 behaviour for Input1 pattern across the timesteps.

higher. This is because the fixed-point type exploited by the board introduces errors when quantising due to the use of less precise values; the values are 16-bit rather than 32-bit. However, the results are very good: slightly less precise values, but for models that exploit spike patterns instead of precise membrane values, the results are usually the same. This can also be seen by analysing the other pattern. The second input pattern is always 10 timesteps long, but the number of ones given to the input is increased to observe how the model behaves when a spike pattern forces more dynamics onto the neurons. The values are as follows:

Table 4.2: Input2 Pattern Across Timesteps (Neurons 0 to 11)

Timestep	n0in	n1in	n2in	n3in	n4in	n5in	n6in	n7in	n8in	n9in	n10in	n11in
1	1	1	1	0	1	0	1	1	1	0	1	1
2	1	0	1	1	1	0	1	0	1	0	1	1
3	0	1	1	0	1	1	0	1	1	0	1	0
4	1	0	1	1	0	1	0	1	0	0	0	0
5	0	1	0	1	1	1	1	0	1	0	0	1
6	1	0	1	0	1	1	1	0	1	1	0	1
7	1	1	1	1	0	1	0	1	1	0	1	0
8	1	0	1	0	1	0	1	1	1	1	0	1
9	0	1	1	1	1	1	1	0	1	0	1	0
10	1	1	0	1	1	0	1	1	0	1	0	1

The dynamics of each neuron with Input2 as input are then represented in Figure 4.8, Figure 4.9, Figure 4.10, Figure 4.11, Figure 4.12, Figure 4.13 and Figure 4.14

It can then be observed that, even though the spiking input pattern introduces more dynamics, the behaviour follows the same approach as the previous input pattern. This makes the ST Board implementation both cheaper and highly reliable.

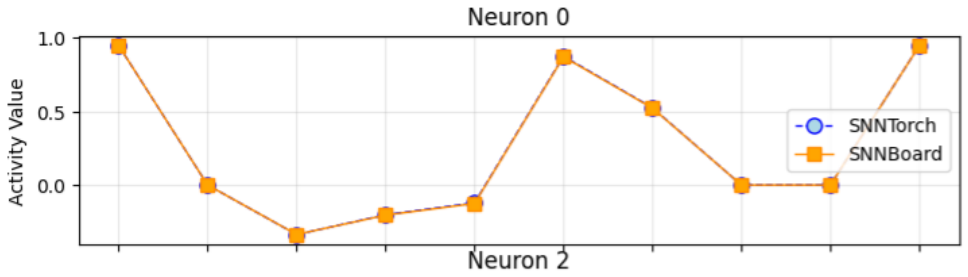

Figure 4.8: N0 behaviour for Input2 pattern across the timesteps.

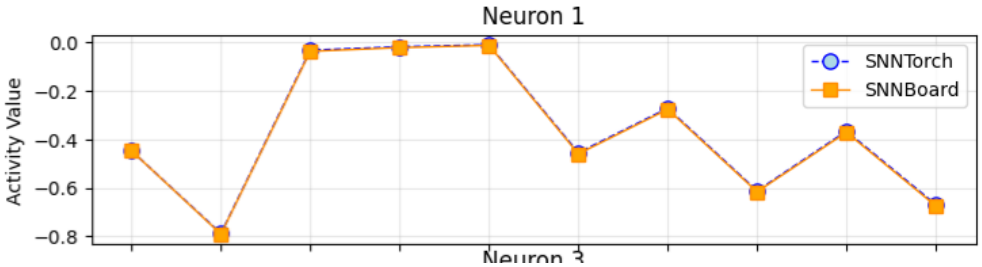

Figure 4.9: N1 behaviour for Input2 pattern across the timesteps.

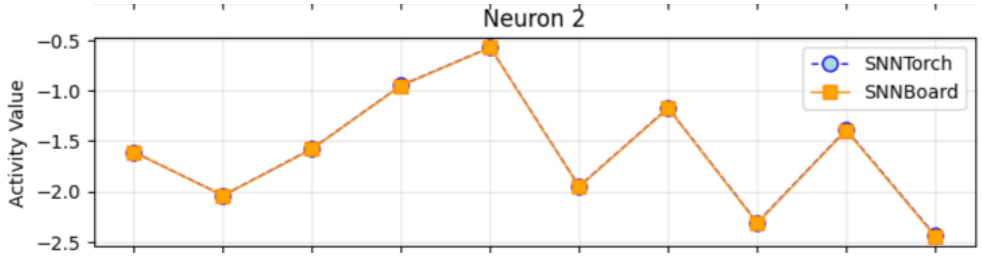

Figure 4.10: N2 behaviour for Input2 pattern across the timesteps.

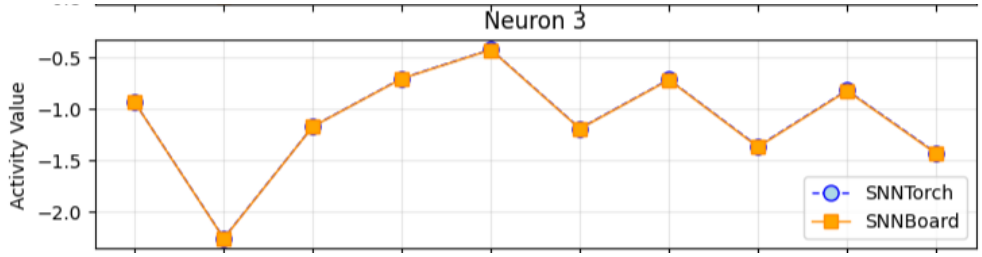

Figure 4.11: N3 behaviour for Input2 pattern across the timesteps.

4.1.2 ds_test across different platforms

Once it has been confirmed that the ST Board model is usable, a formal test is carried out using a dataset comprising 140 samples.
On this dataset, each sample is formed by 256 timesteps; in other words, a sample

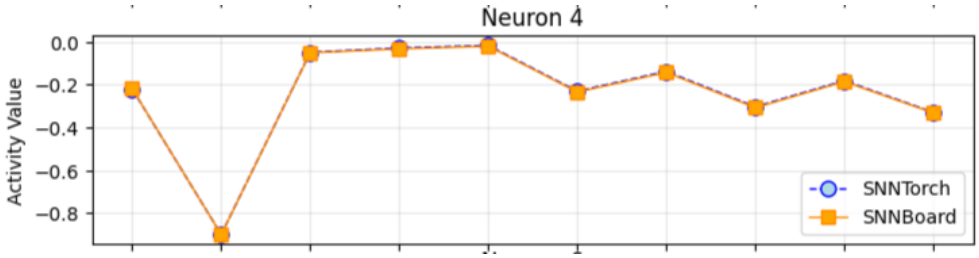

Figure 4.12: N4 behaviour for Input2 pattern across the timesteps.

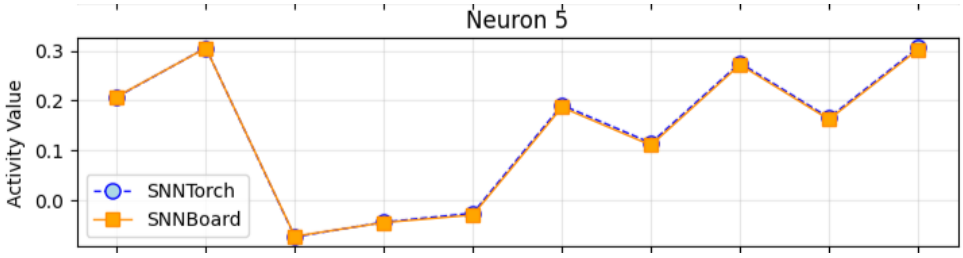

Figure 4.13: N5 behaviour for Input2 pattern across the timesteps.

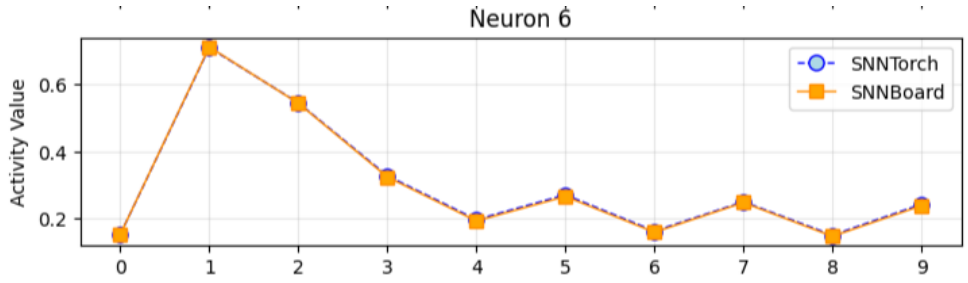

Figure 4.14: N6 behaviour for Input2 pattern across the timesteps.

is ready when 256 different input spike arrays have been received. To recognise the correct class, the spikes are accumulated into a classification array for each timestep. After executing the 256 timesteps and accumulating the output spikes for each timestep, the class value is the neuron that has spiked the most during all 256 timesteps. After classifying a single sample, the state of the SNN is reset so that the next classification is independent and unbiased from the previous one. The dataset also contains the expected class value, enabling the accuracy of the network to be computed and the performance of the models to be assessed.
The results of all the 140 samples analyzed are written in the Table A.1:

After executing on all three platforms, the accuracy of the models can be extracted as a ratio of matches to total samples. The overall result is 91.43%. The reliability of the ST Board implementation can be seen in this result, even though the match pattern is not the same as the GAP 8 and snnTorch models. Indeed,

the failed tests in these two models are 2, 4, 13, 15, 35, 55, 56, 60, 86, 95, 113 and 138. However, in the ST model, tests 13 and 113 are correct, while tests 23 and 79 are missed only in this implementation. This is because the spike patterns of GAP 8 and snnTorch are identical; they perform the same actions on 32 bits without error. The uniform quantisation errors turn two matches into misses and vice versa because the spike patterns are slightly different. This can be seen in the Figure 4.15, Figure 4.16, Figure 4.17, Figure 4.18, Figure 4.19, Figure 4.20 and Figure 4.21 graphs, which compare the spike accumulation of a specific neuron for each sample.

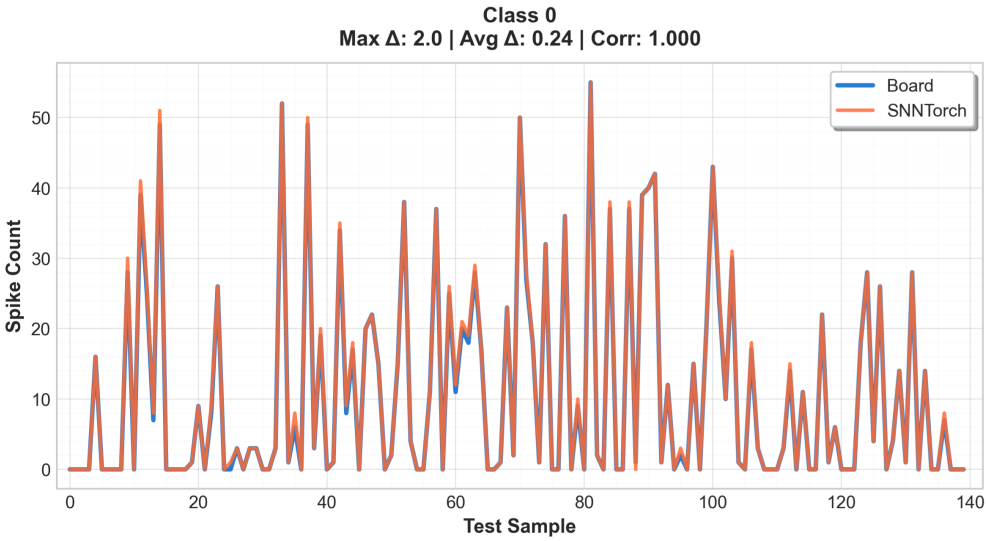

Figure 4.15: N0 spike accumulation ds_test

Small analyses were performed to demonstrate that the spike pattern is undoubtedly different, with a maximum of seven spikes present in the N6 neuron (Figure 4.21) and a minimum of two spikes for the N0 and N3 neurons (Figure 4.15 and Figure 4.18). The average spike variation is the mean of all variations across all 140 samples; this value is consistently below 1. The correlation factor, on the other hand, is:

$$r_{X,Y} = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}}$$

where:

- x_i: The i-th element of the spikeset X (first array).
- y_i: The i-th element of the spikeset Y (second array).
- $\bar{x}$: The mean (average) value of all elements in X, calculated as $\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$.

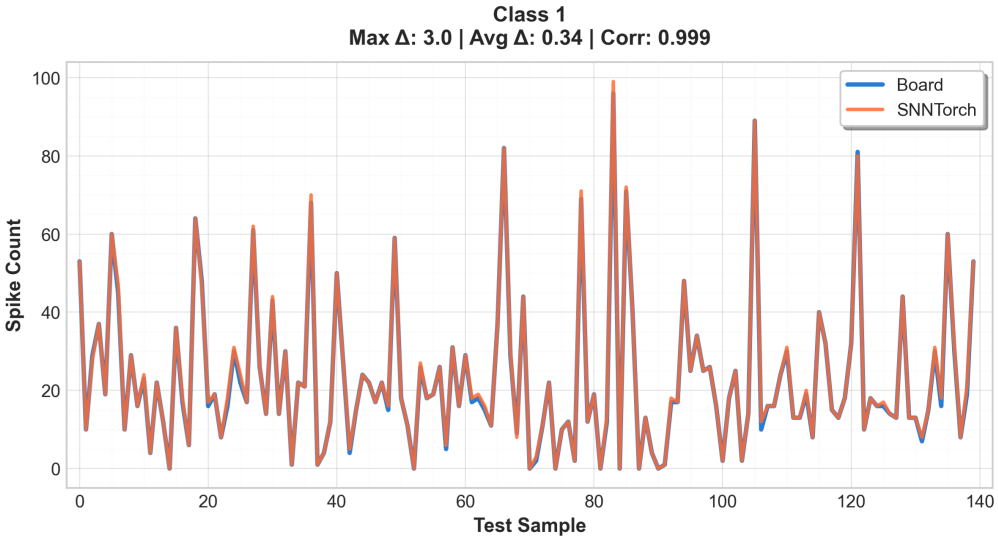

Figure 4.16: N1 spike accumulation ds_test

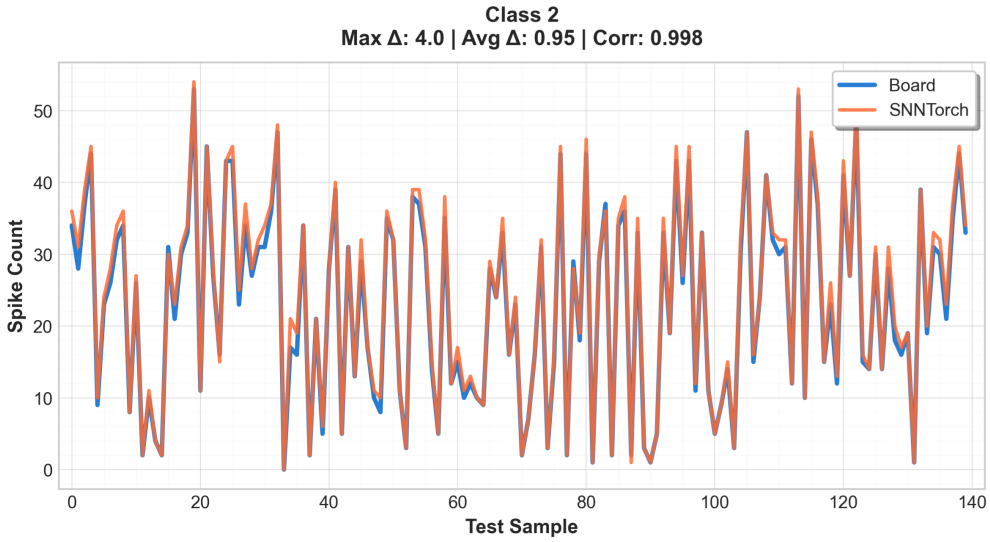

Figure 4.17: N2 spike accumulation ds_test

- $\bar{y}$: The mean (average) value of all elements in Y, calculated as $\bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i$.
- n: The total number of elements in each spikeset.
- $\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})$: The sum of the products of deviations of x_i and y_i from their respective means.
- $\sum_{i=1}^{n}(x_i - \bar{x})^2$ and $\sum_{i=1}^{n}(y_i - \bar{y})^2$: The sum of squared deviations, representing the variance for each spikeset.

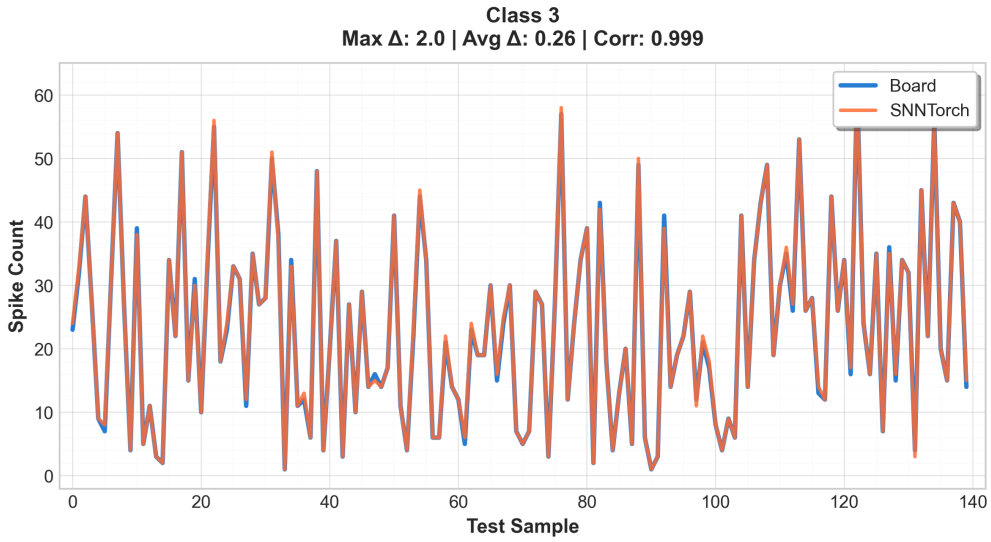

Figure 4.18: N3 spike accumulation ds_test

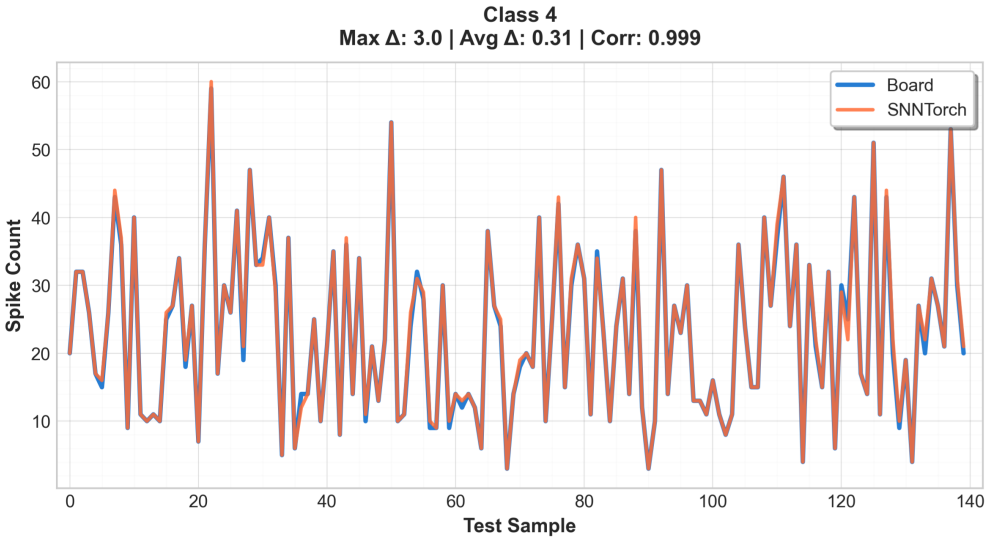

Figure 4.19: N4 spike accumulation ds_test

- $\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2}$ and $\sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}$: The standard deviation for each spike-set.

This factor is used to determine whether the spike values are correlated. The result is 1 if the values are equal. Across all neurons, the value is almost 1, which guarantees the reliability of the model.
For the GAP 8 model, I used the GVSOC simulator, so the execution time is meaningless because real hardware must be used to see the real values. Therefore,

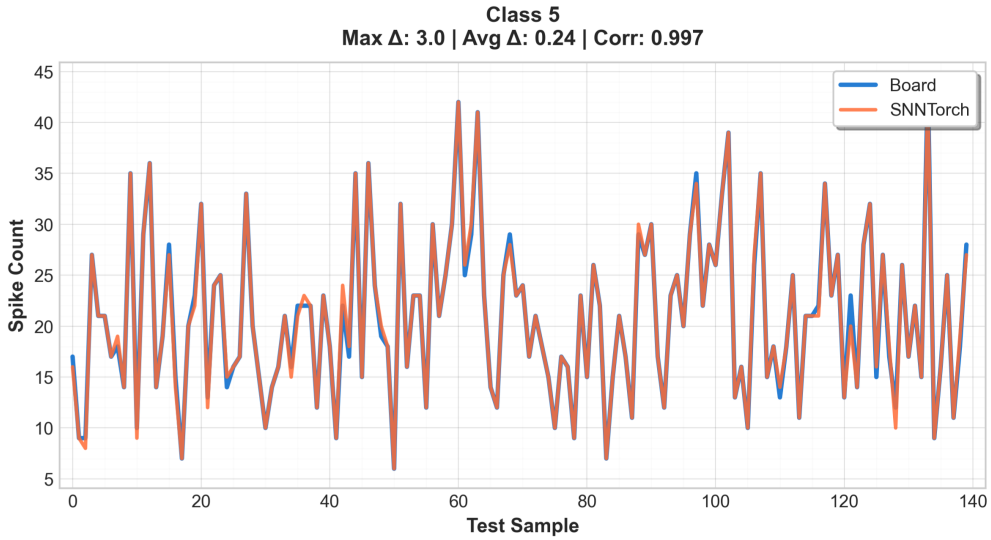

Figure 4.20: N5 spike accumulation ds_test

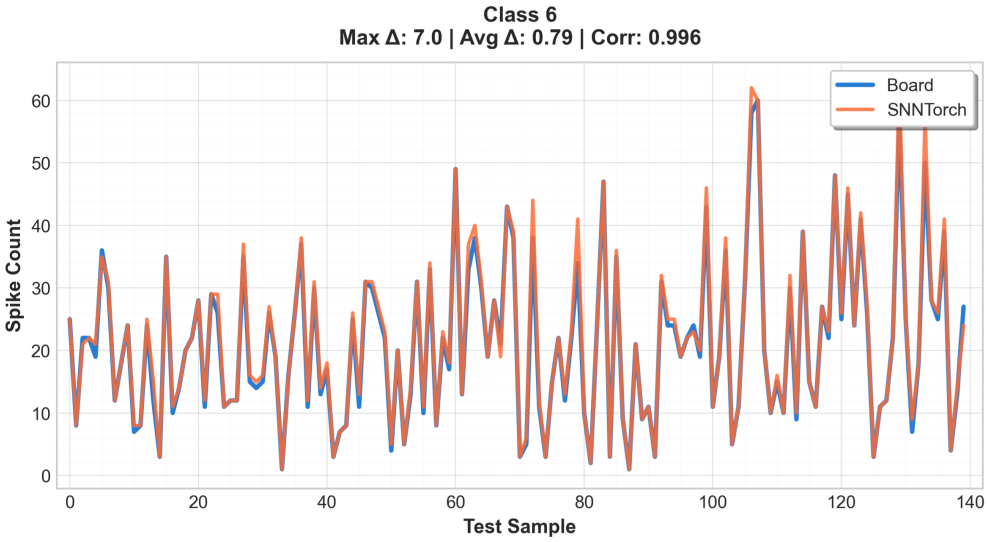

Figure 4.21: N6 spike accumulation ds_test

only the execution times of snnTorch and Board are analysed. The total execution time in milliseconds is measured using the **HAL_GetTick()** function on the ST Board and the average is then calculated over 140 samples. For snnTorch, the **time.perf_counter()** function from the time library is used to calculate the execution time and mean time per sample in the same way. The results were an execution time of 2.216 seconds on the Board versus 18.213964 seconds on snnTorch, with an average time per sample of 15.83 ms on the Board versus 0.130100 seconds on snnTorch. These results emphasise the power of an optimised hardware model

over a standard snnTorch model, which is extremely slower.
Ultimately, these results are printed directly onto the LCD mounted over the board. A first screen is shown during execution (Figure 4.22) and a second screen provides a recap of the test execution results (Figure 4.23).

Figure 4.22: Startup screen LCD

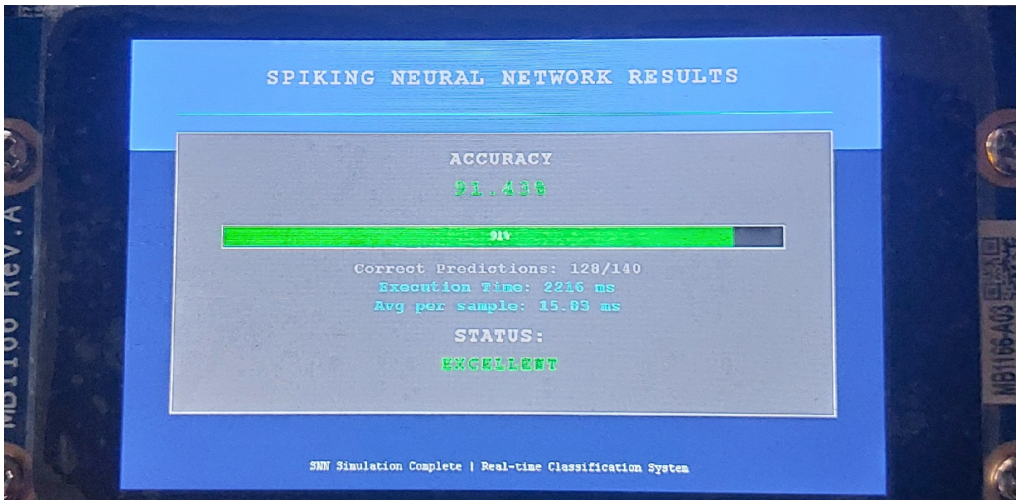

Figure 4.23: Results over LCD

4.2 MNIST Benchmark using Spiker+ and NIR generator

This section describes how the NIR-to-C, as described in Chapter 3, could be used to implement larger SNN architectures in practice.
All the steps required to make the system work are described:

- Using the Spiker+ framework [16], which was originally designed to perform SNN inference on FPGAs, a definition and training has been developed for a MNIST-adapted SNN model.
- The NIR-described model has been extracted using the export_nir function from the standard NIR Python library. Then, using the NIR-to-C translator with minimal changes, the model has been made deployable on board.
- Then, due to memory limitations, five random samples from the MNIST dataset were used to perform benchmarks, including those for execution time and energy consumption.

4.2.1 Spiker+ SNN description and training

The Spiker+ framework has already implemented an example LIF model for performing MNIST classification for FPGA inferences.
The framework uses snnTorch for model definition and training, so the only modification I made was to the synchronous loading of data from the MNIST dataset using the pre-defined functions in the framework, which relies on the Torch Python library.
The conversion of MNIST images to spike trains is achieved in several structured steps, producing a time-varying representation suitable for spiking neural networks.

- **Preprocessing**
  - Start with a 28×28 grayscale MNIST image.
  - Normalise all pixel values to the interval $[0, 1]$, so 0 is darkest, and 1 is brightest.
  - Flatten the image into a one-dimensional vector of length 784, preserving spatial order.
- **Rate-based Spike Encoding**

  For each time step in a simulation window of $T = 100$ steps:
  - Each vector element (pixel) determines the *firing probability* of its corresponding input neuron.

  - Spike generation uses a Bernoulli process:
    * Probability equals the normalised pixel value.
    * For example, a value of 1.0 produces a spike at each time step; 0.5 spikes about half the time; 0 produces no spikes.

- **Spike Train Representation**
  - The outcome is a binary matrix (tensor) of size 100×784.
  - Each entry is 1 if a spike occurred at that time for that pixel, 0 otherwise.
  - Higher original pixel intensities produce more frequent spikes; lower intensities produce fewer or none.

- **Functional Significance**

  This method achieves several advantages:
  - Encodes spatial intensity patterns as temporally distributed spike rates.
  - Provides input compatible with biological and artificial spiking neural networks.
  - Closely follows rate coding as observed in biological sensory systems, where information is represented by neurons' average firing rate over a window of time.

Then, the automatic network builder of the Spiker+ framework was used and the network was left almost the same, with only minor changes, such as resetting to zero and spike accumulation for classification, in order to maintain the same structure as the previously defined models. The SNN is a fully connected two-layer LIF with 728 inputs, a hidden layer of 128 neurons and an output layer of 10 neurons (numbers from 0 to 9). After defining the network, the SNNTorch network builder was used to build it. The training tool of the same framework was also used to perform training through back-propagation in a manner similar to that described in Chapter 3, relying on the Adam optimiser and the default surrogate gradient ATan to perform back-propagative updates to the weights. After training over 20 epochs using the MNIST dataset, the accuracy was 97.42% over the 10,000 MNIST test samples.

4.2.2 extract_nir and translation to C

The model in snnTorch is now trained and ready to use. In Chapter 3, there was a problem with the RLeaky, which is not supported by the extract_nir function of the Python NIR library. This problem does not arise here, as the SNN model

only uses classical LIF and has full connectivity between layers. The NIR-to-C generator has been updated to support the primitive Linear, as well as the Affine (which has the same full connection, but without bias, which is ignored as before). Some minor changes have been made, as the Beta-to-Tau conversion is linear for the extract_nir function, and is performed as follows:

$$dt = 1e - 4$$

$$tau\_mem = dt/(1 - \beta)$$

$$r = tau\_mem/dt.$$

There is no problem with r; it has simply been ignored, as the behaviour of both snnTorch and snn2mcu does not rely on it. Just an adaptation has been made, coherently with this tau calculation, to extract the correct beta from snnTorch when passing from NIR. The new NIR-to-C parser has also been updated to define static constants when defining weights, in order to define static values in FLASH memory, which makes the system compatible with memory limitations. Using the parser, the respective .c and .h code has been generated and a small Python program has been written to extract five random samples from the 10,000-sample dataset for further benchmarking on the board.

4.2.3 Benchmark over ST Board

In the end, the system is now ready to be used. Firstly, it was checked that the system worked as expected with five random values. It classified all five random values correctly, which is only meaningful in showing that the system actually works because the memory is too small to perform an accuracy benchmark.
So, the focus of this implementation was on energy consumption and execution time per sample. To make the benchmark meaningful, the same five samples were iterated 100 times to see the results clearly. Then, all the unnecessary peripherals and the M4 processor were deactivated, as only the M7 is used by the SNN.
Figure 4.24 shows the voltage of the chip while executing a C program. It is 3.3 V and remains constant. Figure 4.25 shows a sample of current consumption during the Benchmark execution period. The UART output when running the classifier is shown in Figure 4.26. The average current consumption is close to this value, as it fluctuates slightly with each sample. For simplicity, the average value of 17 mA has been used here. In terms of execution time, an average of 187.82 ms per sample was recorded during execution, enabling the energy consumption per sample to be calculated.

$$P = V \times I = 56.1mW$$

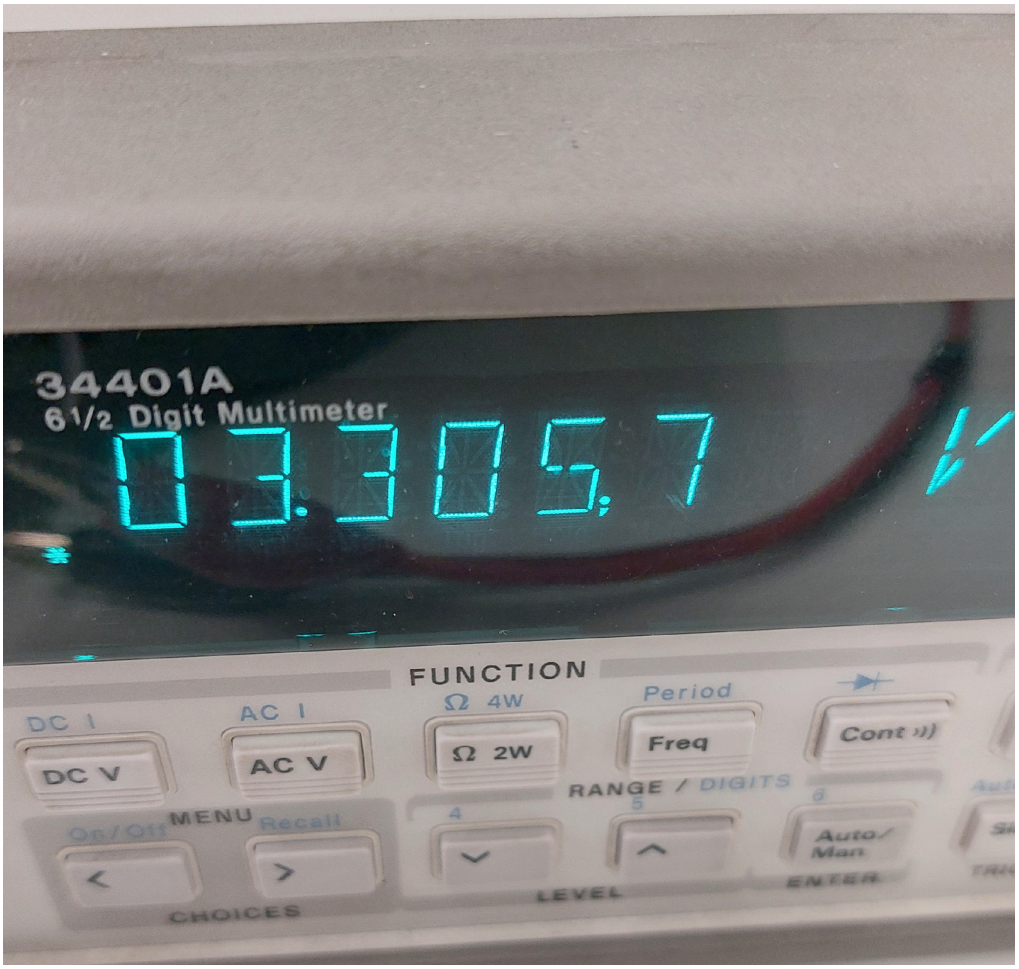

Figure 4.24: Tension of the board while executing.

$$I_{\mathrm{A}} = \frac{17}{1000} = 0.017 \text{ A}$$

$$t_s = \frac{187.82}{1000} = 0.18782 \text{ s}$$

$$E = P \times t_s$$

$$E = 3.3 \times 0.017 \times 0.18782$$

$$E \approx 0.01054 \text{ J}$$

$$E_{\mathrm{mJ}} = E \times 1000$$

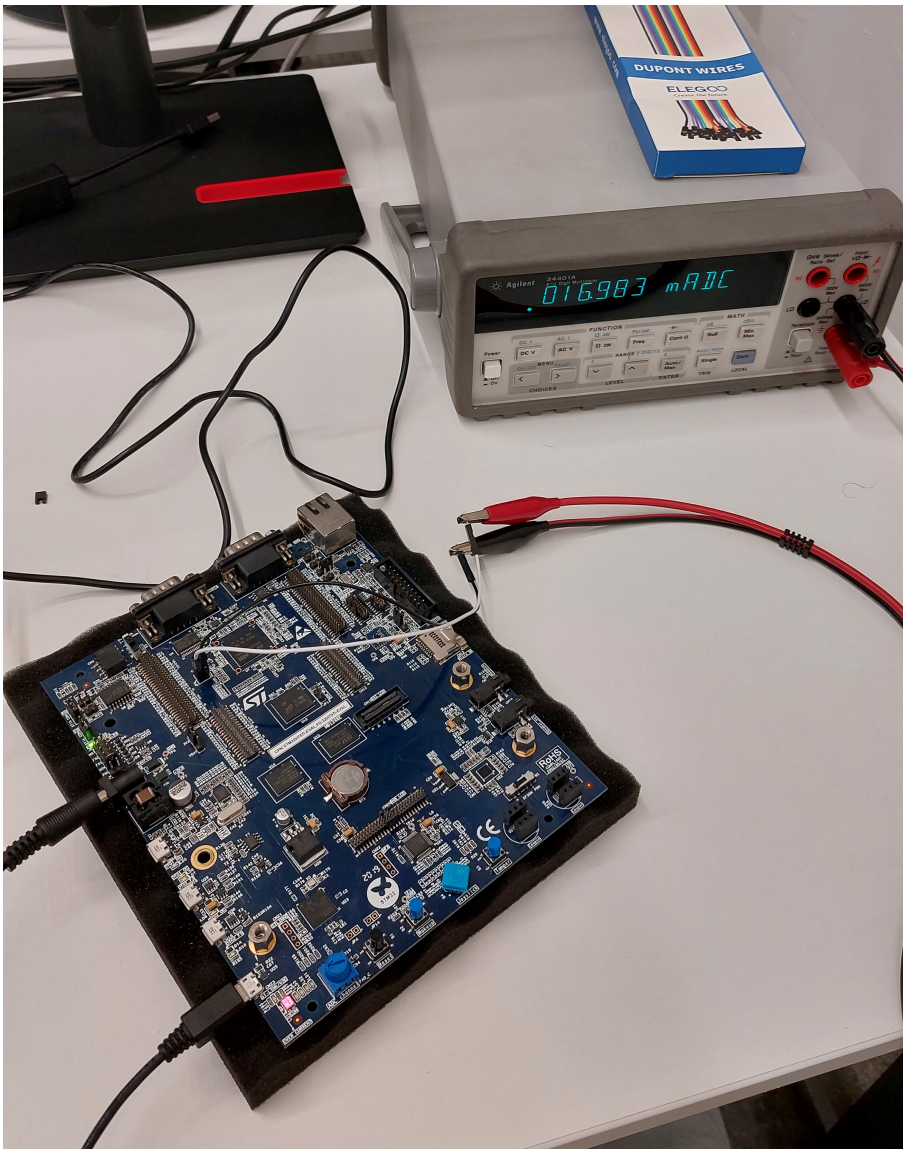

Figure 4.25: One sample of the current value when executing MNIST

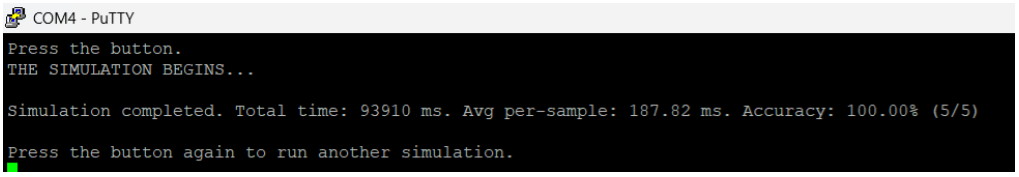

Figure 4.26: Output UART MNIST classifier

$$E_{\mathrm{mJ}} \approx 10.54 \text{ mJ}$$

This is the energy consumed per sample by the system.
These are great results, given that only commonly available MCU ST libraries have been exploited here, and that the M7 processor is not optimised for neuromorphic implementations.

4.2.4 Comparison against FPGA

The results achieved here are then compared with the FPGA implementation for MNIST classification present in [16]. The results are compared in Table 4.3. Comparing these results, we can surely tell that an FPGA implementation has faster execution time per inference and also less energy consumption because the system developed on FPGA has specific architectures designed to implement that specific SNN at its best, and uses less bit bandwidth, which also justifies such results. The commercial ST Board could not compete in those fields because there is no specific hardware designed to optimise the SNN execution, but the proposed one, with only firmware adjustment, is sufficient to efficiently execute inferences of real-time applications with MCU low power platforms. Indeed, in terms of power, the proposed custom implementation has a slightly lower value compared to Spiker+.

	snn2mcu implementation	Spiker +
T_{lat}/img [ms]	187.82	0.78
E/img [mJ]	10.54	0.14
Power [W]	0.0561	0.18

Table 4.3: Comparison of snn2mcu and spiker+

Chapter 5

Conclusion

This thesis addresses the challenge of deploying Spiking Neural Networks (SNNs) on commercial, low-power Microcontroller Units (MCUs), thereby bridging the gap between neuromorphic computing research and practical edge applications. In Figure 5.1, there is a summary of the system realized. The work demonstrates that SNNs, despite their biological inspiration and event-driven nature, can be efficiently implemented on widely available hardware processors such as the ARM Cortex-M7 and the RISC-V of the GAP-8 SoC. By leveraging hardware acceleration features (DSP extensions, multi-level memory, and parallelism), the custom snn2mcu library enables the execution of advanced neuron models (Leaky-Integrate-and-Fire and Izhikevich) and supports multiple connectivity patterns (fully connected, recurrent, one-by-one). This approach makes SNNs accessible for real-world applications, including embedded robotics, wearable sensors, and IoT devices, without requiring expensive or proprietary neuromorphic hardware.

A key achievement is the developement of a systematic workflow for SNN deployment. The thesis introduces a pipeline that starts with high-level SNN design in Python(using snnTorch), proceeds through quantization and optimization, and culminates in automated code generation for embedded targets. The integration of the Neuromorphic Intermediate Representation (NIR) framework enables portable, hardware-agnostic model descriptions, which are then translated into optimized C code for the target MCU. This automation streamlines the translation from research prototypes to deployable firmware, reducing mabnual coding effort and minimizing the risk of implementation errors. The result is a flexible, extensible framework that can be adapted to new neuron models, network architectures, and hardware platforms with minimal overhead.

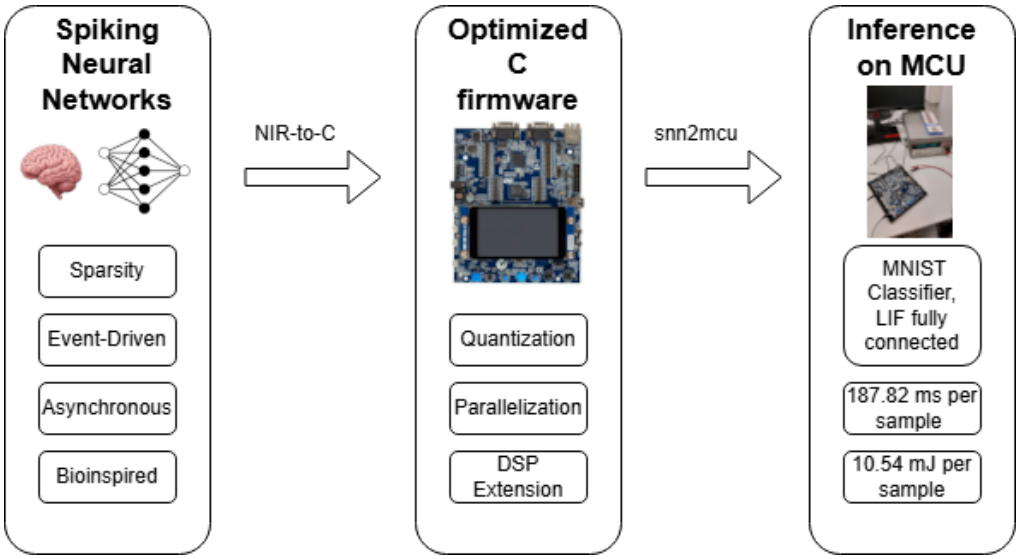

Figure 5.1: Summary of the complete System

5.1 Validation and Performance Analysis

The experimental campaign provides robust validation of the proposed methodology. Two distinct SNN models were implemented and benchmarked: a Braille character classifier for robotic tactile sensing and a handwritten digit classifier using the MNIST dataset. The Braille classifier, based on a three-layer recurrent SNN with LIF neurons, achieved a classification accuracy of 91.43% on a 140-sample test set when deployed on the STM32H757I-EVAL board. This performance matches that of the reference snnTorch simulation, confirming the fidelity of the embedded implementation. The execution time was reduced by a factor of six compared to floating-point software execution, highlighting the effectiveness of quantization and DSP optimizations. The model's energy consumption per sample was also measured, demonstrating its suitability for battery-powered edge devices.
The MNIST classifier, a two-layer fully connected SNN with LIF neurons, achieved an accuracy of 97.42% on the standard test set using snnTorch. When deployed on the ARM Cortex-M7, the system consumed an average of 10.54 mJ per sample, with an execution time of 187.82 ms per sample. These results are comparable with those of specialized FPGA accelerators [16], underscoring the potential of commercial MCUs for SNN inference. The energy efficiency and computational performance are further enhanced by the use of fixed-point arithmetic, vectorized DSP operations, and efficient memory management. The thesis also demonstrates the feasibility of deploying SNNs on the GAP-8 RISC-V SoC, leveraging its multi-core architecture and hardware convolution engine to achieve near-linear speedup for parallelizable tasks, as stated in their research papers introduced in chapter 2.5.2.

5.2 Challenges and Limitations

Despite these results, the thesis identifies several challenges and limitations that must be addressed in future work. One major challenge is the trade-off between accuracy and efficiency when quantizing SNN parameters. While aggressive quantization can significantly reduce memory footprint and computational cost, it may introduce quantization errors that degrade classification accuracy. The thesis shows that careful, smart quantization techniques can mitigate and, in some specific cases, erase degradation effects, so further research is needed to develop robust, automated quantization strategies for SNNs.
Another limitation is the lack of support for advanced learning algorithms on embedded platforms. Most SNNs are trained offline using GPU-accelerated frameworks, and the resulting models are then deployed on MCUs for inference. However, the ability to perform online learning, adapting network weights in response to new data or changing environments, would greatly enhance the practical utility of SNNs for edge applications. Implementing local learning rules on resource-constrained hardware remains a significant challenge, requiring efficient algorithms and hardware support for dynamic weight updates.
The thesis also highlights the importance of hardware-software co-design. While the snn2mcu library is optimized for the ARM Cortex M7 and GAP-8, its performance may vary when using such optimizations on other MCUs. Future work should focus on extending the library to support a broader range of hardware platforms, including those with specialized AI accelerators or novel memory hierarchies. Additionally, the integration of SNNs with other edge AI technologies could enable more complex, multimodal applications.

5.3 Future Research Directions

The findings of this thesis pave the way for several promising research directions, such as:

- Development of more sophisticated neuron models, like Hodgkin-Huxley or Synaptic, could enhance the biological realism and computational power of SNNs. These models are computationally expensive, but advances in numerical methods and hardware acceleration may make them feasible for embedded deployment. The thesis demonstrates that the Izhikevich model can be partially optimized for the ARM Cortex-M7, suggesting that further optimizations could enable full support for advanced neuron dynamics.

- Integration of online learning algorithms into the snn2mcu framework would enable SNNs to adapt to changing environments and learn from streaming

data. This could be achieved by implementing local learning rules or by leveraging hardware features such as dedicated memory for synaptic weights. The ability to perform continuous learning on edge devices would open up new applications in robotics, autonomous systems, and personalized healthcare.

- Extension of the NIR-to-C translator to support additional network architectures and hardware platforms would facilitate broader adoption of SNNs in the embedded domain. This could include support for recurrent neural networks (all-to-all connections), convolutional SNNs, and hybrid architectures that combine SNNs with traditional deep learning models. The development of a standardized, open-source ecosystem for SNN deployment would encourage collaboration and innovation in the field.

- This thesis suggests that future work should focus on real-world applications of SNNs in edge computing. This could include the deployment of SNNs in wearable sensors, autonomous robots, and smart home devices, where low power consumption and real-time performance are critical. The integration of SNNs with other edge AI technologies, such as computer vision, natural language processing, and reinforcement learning, could enable more intelligent, adaptive systems that operate efficiently in resource-constrained environments.

- At this moment, the snn2mcu library is able only to support optimized LIF neuron with fully connected layers, one-to-one connections, and one-to-one recurrence. The system could be improved to support other and more complex neuron models and could also be improved to support all-to-all connections in recurrent layers and convolutional SNN over hardware boards. Convolutions replace fully connected or one-to-one connections by using local, weight-shared filters that slide over the input, instead of giving each output neuron its own separate weight to every input neuron. This drastically reduces parameters and enforces that the same feature (e.g., an edge or texture) can be detected anywhere in the input, which is ideal for spatial data like images or event maps in spiking networks.

5.4 Final Reflections

In summary, this thesis demonstrates that SNNs can be efficiently deployed on commercial, low-power microcontrollers, enabling a new generation of intelligent edge devices. The snn2mcu library and the associated workflow provide a robust foundation for SNN research and development, bridging the gap between neuromorphic computing and practical applications. The experimental results validate the effectiveness of quantization, DSP optimizations, and hardware acceleration for

SNN inference, while also highlighting the challenges and opportunities for future work. By addressing these challenges and exploring new research directions, the field of neuromorphic computing can continue to advance, bringing the benefits of brain-inspired intelligence to a wider range of applications and users.

Appendix A

Inference Results tables

Test	Expected	T	Results (T)	B	Results (B)	G	Results (G)
0	1	1	Match	1	Match	1	Match
1	3	3	Match	3	Match	3	Match
2	2	3	Miss	3	Miss	3	Miss
3	2	2	Match	2	Match	2	Match
4	6	5	Miss	5	Miss	5	Miss
5	1	1	Match	1	Match	1	Match
6	1	1	Match	1	Match	1	Match
7	3	3	Match	3	Match	3	Match
8	4	4	Match	4	Match	4	Match
9	5	5	Match	5	Match	5	Match
10	4	4	Match	4	Match	4	Match
11	0	0	Match	0	Match	0	Match
12	5	5	Match	5	Match	5	Match
13	5	6	Miss	5	Match	6	Miss
14	0	0	Match	0	Match	0	Match
15	2	1	Miss	1	Miss	1	Miss
16	4	4	Match	4	Match	4	Match
17	3	3	Match	3	Match	3	Match
18	1	1	Match	1	Match	1	Match
19	2	2	Match	2	Match	2	Match
20	5	5	Match	5	Match	5	Match
21	2	2	Match	2	Match	2	Match
22	4	4	Match	4	Match	4	Match
23	6	6	Match	0	Miss	6	Match
24	2	2	Match	2	Match	2	Match
25	2	2	Match	2	Match	2	Match
26	4	4	Match	4	Match	4	Match
27	1	1	Match	1	Match	1	Match
28	4	4	Match	4	Match	4	Match

29	4	4	Match	4	Match	4	Match
30	1	1	Match	1	Match	1	Match
31	3	3	Match	3	Match	3	Match
32	2	2	Match	2	Match	2	Match
33	0	0	Match	0	Match	0	Match
34	4	4	Match	4	Match	4	Match
35	5	6	Miss	6	Miss	6	Miss
36	1	1	Match	1	Match	1	Match
37	0	0	Match	0	Match	0	Match
38	3	3	Match	3	Match	3	Match
39	5	5	Match	5	Match	5	Match
40	1	1	Match	1	Match	1	Match
41	2	2	Match	2	Match	2	Match
42	0	0	Match	0	Match	0	Match
43	4	4	Match	4	Match	4	Match
44	5	5	Match	5	Match	5	Match
45	4	4	Match	4	Match	4	Match
46	5	5	Match	5	Match	5	Match
47	6	6	Match	6	Match	6	Match
48	6	6	Match	6	Match	6	Match
49	1	1	Match	1	Match	1	Match
50	4	4	Match	4	Match	4	Match
51	5	5	Match	5	Match	5	Match
52	0	0	Match	0	Match	0	Match
53	2	2	Match	2	Match	2	Match
54	3	3	Match	3	Match	3	Match
55	4	3	Miss	3	Miss	3	Miss
56	5	6	Miss	6	Miss	6	Miss
57	0	0	Match	0	Match	0	Match
58	2	2	Match	2	Match	2	Match
59	5	5	Match	5	Match	5	Match
60	5	6	Miss	6	Miss	6	Miss
61	5	5	Match	5	Match	5	Match
62	6	6	Match	6	Match	6	Match
63	5	5	Match	5	Match	5	Match
64	6	6	Match	6	Match	6	Match
65	4	4	Match	4	Match	4	Match
66	1	1	Match	1	Match	1	Match
67	2	2	Match	2	Match	2	Match
68	6	6	Match	6	Match	6	Match
69	1	1	Match	1	Match	1	Match
70	0	0	Match	0	Match	0	Match
71	0	0	Match	0	Match	0	Match
72	6	6	Match	6	Match	6	Match

73	4	4	Match	4	Match	4	Match
74	0	0	Match	0	Match	0	Match
75	3	3	Match	3	Match	3	Match
76	3	3	Match	3	Match	3	Match
77	0	0	Match	0	Match	0	Match
78	1	1	Match	1	Match	1	Match
79	6	6	Match	4	Miss	6	Match
80	2	2	Match	2	Match	2	Match
81	0	0	Match	0	Match	0	Match
82	3	3	Match	3	Match	3	Match
83	1	1	Match	1	Match	1	Match
84	0	0	Match	0	Match	0	Match
85	1	1	Match	1	Match	1	Match
86	2	1	Miss	1	Miss	1	Miss
87	0	0	Match	0	Match	0	Match
88	3	3	Match	3	Match	3	Match
89	0	0	Match	0	Match	0	Match
90	0	0	Match	0	Match	0	Match
91	0	0	Match	0	Match	0	Match
92	4	4	Match	4	Match	4	Match
93	6	6	Match	6	Match	6	Match
94	1	1	Match	1	Match	1	Match
95	3	2	Miss	2	Miss	2	Miss
96	2	2	Match	2	Match	2	Match
97	5	5	Match	5	Match	5	Match
98	2	2	Match	2	Match	2	Match
99	6	6	Match	6	Match	6	Match
100	0	0	Match	0	Match	0	Match
101	5	5	Match	5	Match	5	Match
102	5	5	Match	5	Match	5	Match
103	0	0	Match	0	Match	0	Match
104	3	3	Match	3	Match	3	Match
105	1	1	Match	1	Match	1	Match
106	6	6	Match	6	Match	6	Match
107	6	6	Match	6	Match	6	Match
108	3	3	Match	3	Match	3	Match
109	2	2	Match	2	Match	2	Match
110	4	4	Match	4	Match	4	Match
111	4	4	Match	4	Match	4	Match
112	6	6	Match	6	Match	6	Match
113	3	2	Miss	3	Match	2	Miss
114	6	6	Match	6	Match	6	Match
115	2	2	Match	2	Match	2	Match
116	2	2	Match	2	Match	2	Match

117	5	5	Match	5	Match	5	Match
118	3	3	Match	3	Match	3	Match
119	6	6	Match	6	Match	6	Match
120	2	2	Match	2	Match	2	Match
121	1	1	Match	1	Match	1	Match
122	3	3	Match	3	Match	3	Match
123	6	6	Match	6	Match	6	Match
124	5	5	Match	5	Match	5	Match
125	4	4	Match	4	Match	4	Match
126	5	5	Match	5	Match	5	Match
127	4	4	Match	4	Match	4	Match
128	1	1	Match	1	Match	1	Match
129	6	6	Match	6	Match	6	Match
130	3	3	Match	3	Match	3	Match
131	0	0	Match	0	Match	0	Match
132	3	3	Match	3	Match	3	Match
133	6	6	Match	6	Match	6	Match
134	3	3	Match	3	Match	3	Match
135	1	1	Match	1	Match	1	Match
136	6	6	Match	6	Match	6	Match
137	4	4	Match	4	Match	4	Match
138	3	2	Miss	2	Miss	2	Miss
139	1	1	Match	1	Match	1	Match

Table A.1: 140 sample ds_test, T stands for snnTorch, B for ST Board, and G for GAP8 SoC. More details in 4.1

Acknowledgements

Bibliography

[1] Dhireesha Kudithipudi et al. «Neuromorphic computing at scale». In: *Nature* 637 (2025), pp. 801–812. DOI: 10.1038/s41586-024-08253-8.

[2] Duy-Anh Nguyen, Xuan-Tu Tran, and Francesca Iacopi. «A Review of Algorithms and Hardware Implementations for Spiking Neural Networks». In: *Journal of Low Power Electronics and Applications* 11.2 (2021), p. 23. DOI: 10.3390/jlpea11020023.

[3] Eric Hunsberger and Chris Eliasmith. «Spiking Deep Networks with LIF Neurons». In: *arXiv preprint arXiv:1510.08829* (2015).

[4] Vittorio Fra et al. «Neu-BrAuER: A Neuromorphic Braille Letters Audio-Reader for Commercial Edge Devices». In: *ECML PKDD*. 2025, pp. 51–60. DOI: 10.1007/978-3-031-74643-7_5.

[5] Vittorio Fra et al. «Natively Neuromorphic LMU Architecture for Encoding-Free SNN-Based HAR on Commercial Edge Devices». In: *Artificial Neural Networks and Machine Learning – ICANN 2024*. Ed. by Michael Wand et al. Cham: Springer Nature Switzerland, 2024, pp. 377–391. ISBN: 978-3-031-72359-9.

[6] Sebastian Hoppner et al. «The SpiNNaker 2 Processing Element Architecture for Hybrid Digital Neuromorphic Computing». In: *arXiv preprint arXiv:2103.08392v2* (Aug. 2022). URL: https://arxiv.org/abs/2103.08392v2.

[7] Intel Labs. *Taking Neuromorphic Computing to the Next Level with Loihi 2*. Technology Brief. Accessed: 2025-11-07. Intel Labs, 2021. URL: https://www.intel.com/content/www/us/en/research/neuromorphic-community.html.

[8] Jason K Eshraghian et al. «Training Spiking Neural Networks Using Lessons From Deep Learning». In: *Proceedings of the IEEE* 111.9 (2023), pp. 1016–1046. DOI: 10.1109/JPROC.2023.3308088.

[9] Jens E. Pedersen et al. «Neuromorphic intermediate representation: A unified instruction set for interoperable brain-inspired computing». In: *Nature Communications* 15 (2024), p. 8122. DOI: 10.1038/s41467-024-52259-9.

[10] STMicroelectronics. *Evaluation boards with STM32H747XI and STM32H757XI MCUs*. UM2525 Rev 6. User manual. STMicroelectronics. Apr. 2023. URL: https://www.st.com/resource/en/user_manual/um2525-evaluation-boards-with-stm32h747xi-and-stm32h757xi-mcus-stmicroelectronics.pdf.

[11] Arm Ltd. *Arm Cortex-M7 Processor Datasheet*. Accessed: November 8, 2025. Arm Limited. Cambridge, United Kingdom, 2020. URL: https://www.arm.com/-/media/Arm%20Developer%20Community/PDF/Processor%20Datasheets/Arm-Cortex-M7-Processor-Datasheet.pdf.

[12] Thomas Lorenser. «The DSP capabilities of arm cortex-m4 and cortex-m7 processors». In: *ARM White Paper* 29 (2016), pp. 1–19.

[13] Eric Flamand et al. «GAP-8: A RISC-V SoC for AI at the Edge of the IoT». In: *2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 2018, pp. 1–4. DOI: 10.1109/ASAP.2018.8445101.

[14] Angelo Garofalo et al. «PULP-NN: accelerating quantized neural networks on parallel ultra-low-power RISC-V processors». In: *Philosophical Transactions of the Royal Society A* 378.2168 (2019), p. 20190155. DOI: 10.1098/rsta.2019.0155.

[15] STMicroelectronics. *STEdgeAI-Core: Artificial intelligence AI optimizer technology for STMicroelectronics products*. Data brief DB5290, Rev 2. Accessed November 2025. 2024. URL: https://www.st.com.

[16] Alessio Carpegna, Alessandro Savino, and Stefano Di Carlo. «Spiker: a framework for the generation of efficient Spiking Neural Networks FPGA accelerators for inference at the edge». In: *arXiv preprint arXiv:2401.01141* (2024). URL: https://arxiv.org/abs/2401.01141.

[17] Jiawei Liao et al. «A Spiking Neural Network Decoder for Implantable Brain Machine Interfaces and its Sparsity-aware Deployment on RISC-V Microcontrollers». In: *arXiv preprint arXiv:2405.02146* (2024). arXiv: 2405.02146 [eess.SP].