POLITECNICO DI TORINO

Department of Control and Computer Engineering

Master of Science in Computer Engineering

Master Degree Thesis

Designing an end-to-end Pipeline for Developing and Deploying IoT Solutions on Embedded Neuromorphic Platforms

Advisor

Prof. Gianvito Urgese

Co-Advisor:

Prof. Giacomo Indiveri

Vittorio Fra, Ph.D.

Candidate

Marco Bramini

2022 - 2023

Abstract

The primary objective of this thesis is to explore, develop, and evaluate solutions for building and training SNN-based models that are immediately compatible for deployment on state-of-the-art neuromorphic embedded systems.

To extend the study to real use cases, this work specifically addresses the task of Human Activity Recognition (HAR). HAR involves the detection of human actions by analyzing motion data collected from sensors within the Inertial Measurement Unit (IMU) of smartphones and smartwatches.
In particular, the thesis focuses on the revised version of the Wireless Sensor Data Mining (WISDM) dataset, comprising recordings from the accelerometer and gyroscope sensors embedded in IMUs of smartphones and smartwatches.

Historically, the feasibility of HAR has been affected by the limited computational and power resources available in portable devices, making real-time data processing impractical.
The ultra-low-power and ultra-low-latency capabilities of Spiking Neural Networks (SNNs), which can operate on non-von Neumann neuromorphic devices, offer a potential solution to these challenges.

SNNs model their behavior on biologically inspired computation, mimicking the mechanisms of organic brains. This approach is of high interest due to its impressive computational capabilities, low latency, and minimal energy consumption. SNNs consist of spiking neurons that operate in a sparse mode, in contrast to the continuous activity of neurons in traditional Artificial Neural Networks. Each neuron processes sparse input events, known as spikes, and remains inactive in the absence of input, making them unparalleled in terms of energy efficiency.

This thesis specifically targets two neuromorphic embedded systems: DYNAP-SE2, developed by the Institute for Neuroinformatics of Zürich, and Xylo, developed by SynSense. DYNAP-SE2 is a mixed-signal asynchronous chip that utilizes analog spiking neurons and synapses for computation. Xylo is a fully digital synchronous chip tailored for processing low-dimensional input signals.
The overall design, development, and training of solutions for these devices pose considerable challenges, mainly due to the unique characteristics and limitations of each platform.

This thesis proposes an end-to-end multi-step pipeline for generating models compatible with the target devices, that can be generalized to work across a wide range of applications.

The ultimate goal of the thesis is to gather information about the capabilities of target devices by evaluating their performance under varying levels of input data complexity.

Both devices are benchmarked across a series of eight tasks of increasing complexity, based on different subsets of the full HAR task, to discover their behavior near their operational limits.

This work introduces critical techniques, such as Synapse Pruning and Quantization Tuning, which enhance the utilization of onboard chip resources and overall model performance. In particular, Quantization Tuning has proven to be necessary to ensure consistent performances after hardware deployment.

The proposed pipeline can produce well-trained, hardware-ready models that can be directly deployed on the target devices.
DYNAP-SE2 and Xylo have proven to be mature enough for simple and medium-difficulty tasks, but struggled with more complex ones primarily due to limited software support.
Improving the software support will be critical for unlocking their full potential and extending their applicability in real-world scenarios.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Neuromorphic computing, a concept originally theorized and first applied in the late 1980s [1, 2], is a computational paradigm that mimics the behavior and structure of biological nervous systems found in organic brains.

In recent years, with the growth of interest and applications of artificial neural networks (ANN), neuromorphic computing has gained increasing attention in the scientific community. It is often regarded as a more efficient substitute for traditional neural networks, offering advantages in terms of power efficiency, robustness, and computational speed [3].

These advantages can primarily be attributed to the abandonment of the von Neumann architecture, which suffers from a big penalty on the latency and power consumption point of view caused by the need to transfer large amounts of data between its internal components [4].

Neuromorphic computing leverages an innovative asynchronous event-based paradigm, which relies on computational nodes with complex internal dynamics, the spiking neurons, that are interconnected to form a network. These networks are called Spiking Neural Networks (SNN). Each neuron is able to independently process sparse input events, called spikes, and remain inactive when there is no input. This behavior diverges from the traditional neural network paradigm in which each network unit processes a continuous input signal.

To make an analogy with biology, neurons are connected through synapses. These act as communication channels between neurons: when a neuron emits a spike, it sends encoded information from its axon to the dendrites of downstream neurons. The strength and quantity of these synaptic connections model the overall behavior and information processing capabilities of biological neural networks [5].

The most common neuron model is the Integrate-and-Fire (IF) model, particularly its leaky variant known as the Leaky Integrate and Fire (LIF) model [6].

In this model, each spike that reaches a specific neuron causes an increase in its membrane potential. When the potential reaches and surpasses a certain level, the membrane threshold voltage, the neuron emits a spike that propagates to the output neurons.

In LIF neurons, the membrane potential tends to decay exponentially and resets back to its resting state over time.

The working behavior of the LIF neuron model, with a particular focus on the membrane potential dynamics, is described by differential equation 1.1.

$$\tau_m \frac{dV_m}{dt} = -(V_m(t) - V_{\text{rest}}) + R_m \cdot I \tag{1.1}$$

In the equation, τ_m stands for the membrane time constant, $V_m(t)$ is the membrane potential as a function of time, V_{rest} is the resting potential of the membrane, R_m is the membrane resistance and I is the input current. The leakage of the membrane potential is described by the term $-(V_m(t) - V_{\text{rest}})$.

Therefore, the behavior of a LIF neuron depends on the input current I, which is calculated as a weighted sum of the input spiking contributions from upstream neurons. Adjusting these weights alters the behavior of the neural network, which can then be trained, as is the case with ANNs, to perform complex pattern recognition tasks.

Generally, neuromorphic devices are digital, analog, or mixed VLSI circuits, in which the physical artificial neurons are realized by employing transistors, and memristors, among other techniques.

Neuromorphic devices find applications across a wide spectrum, spanning from scientific studies to practical classification and regression tasks.

Scientific applications primarily concentrate on the ambitious goal of simulating and understanding the behavior of the human brain. These applications enable researchers to gain insights into the mechanism of neural computation, contributing to the advancement of the neuroscience [7].

In contrast, classification tasks, ideally, represent the field of application in which neuromorphic devices truly shine.
In fields such as image and speech recognition, autonomous robotics, or medical applications, like prosthetic limbs control or anomaly classification in EEGs and ECGs, neuromorphic computing has the potential to revolutionize performance, enabling rapid and energy-efficient decision-making [8].

Overall, the ability to process sensor data in real-time, leveraging their exceptional power efficiency and low-latency capabilities, makes neuromorphic devices suitable for practical, real-use cases where low power consumption and high-speed processing are critical.

In practice though, there are significant limitations that affect the usability and applicability of this kind of device. These limitations can be identified in various aspects, including programming complexity, compatibility with the available software ecosystem of libraries and languages, and hardware constraints.

Moreover, training spiking neural network devices can be a lot more challenging if compared to their traditional counterparts: the non-differentiable nature of spiking neurons makes the standard back-propagation algorithm inapplicable.

To address these challenges several methods have been proposed, involving adaptations of training techniques traditionally employed for Artificial Neural Networks (ANNs) as well as exploring novel approaches [9].

One of the proposed methods is based on supervised learning, where an ANN is initially trained using continuous input signals. Subsequently, artificial neurons are replaced with spiking neurons, leading to enhanced inference efficiency [10]. However, this technique has a possibility of incurring in an accuracy loss due to the conversions and approximations used [9].

Another family of methods employs surrogate activation functions as an approximation of spiking activation to enable back-propagation with input spike events also for SNNs. These techniques are very expensive from the computational point-of-view and require a large amount of data, making it impossible to accomplish efficient on-chip learning. [11].

Furthermore, some neuromorphic systems rely on unsupervised learning techniques, like spike-timing-dependent plasticity (STDP), which is a biologically plausible mechanism that instantaneously adjusts the synaptic weights based on the temporal correlations between pre- and post-synaptic spike timings [12].

These constraints must be taken into consideration when evaluating the suitability of neuromorphic devices for general-purpose applications. While these can be generally acceptable for specific low-complexity tasks, enabling to fully take advantage of the neuromorphic benefits, it could become prohibitive for complex and broad tasks.

In the last 15 years, a wide set of large-scale neuromorphic devices has been developed by various research groups [8].

Starting from 1998, the Advanced Processor Technologies Research Group from the University of Manchester has been conceiving and designing a neuromorphic supercomputer based on the digital paradigm, SpiNNaker [13]. It achieved the significant milestone of simulating one million neurons in 2018 [14].

From 2011 to 2015, research groups coming from the most important European universities collaborated in the development of a hybrid analog neuromorphic supercomputer called BrainScaleS [15].

Both SpiNNaker and BrainScaleS have been funded by the European Union's Human Brain Project, with the objective of enabling brain simulations for neuroscience research.

In 2019, a group of researchers from Tsinghua University introduced Tianjic [16], a chip capable of supporting both spiking and traditional neural networks. This versatility makes it usable in a large range of applications, as demonstrated by its employment in a self-driving bike, able to avoid obstacles, navigate autonomously, and respond to voice commands.

Recently, large companies such as IBM, Intel, and others, have shown a keen interest in the commercialization of these devices.

In 2014, IBM introduced TrueNorth [17], followed by Intel's release of Loihi [18] in 2017. These digital neuromorphic devices share a common objective: to extend the advantages inherent to the neuromorphic paradigm to consumer edge devices.

In 2021, Intel introduced the successor of Loihi, called Loihi2 [19], which is more capable, supports various neuron models, and enables finer and more precise configurations.

This thesis focuses on the DYNAP-SE2 and Xylo neuromorphic chips.

DYNAP-SE2 is a mixed-signal chip that implements its computational capabilities through analog spiking neurons and synapses. It was developed and built by the Institute of Neuroinformatics, affiliated with the UZH and ETH universities of Zürich.

Xylo is a fully digital neuromorphic chip tailored for processing low-dimensional input signals, introduced during 2023 by SynSense, the world's leading supplier of neuromorphic solutions.

The overall design, development, and training of solutions for these devices pose considerable challenges, mainly due to the unique characteristics and limitations of each platform.

The objective of this thesis is to explore the capabilities and limitations of neuromorphic devices, assessing their performance at various levels of input data complexity.

Furthermore, this thesis intends to build upon prior research conducted in a previously published work [20], in which the authors explored the suitability of the neuromorphic approach for on-edge IoT applications.

The final goal is to evaluate their suitability for practical applications, relying on off-the-shelf available consumer-grade sensors, without the need for specialized, high-precision, or costly hardware.

To extend the study to a real use case the DYNAP and Xylo platforms are benchmarked using a classification task based on the Human Activity Recognition (HAR) problem. The main objective of the HAR task is to identify human actions by analyzing motion data captured through wearable sensors. The task will be described in more detail in Section 2.1.

Both target devices are tested across a series of tasks of increasing complexity, based on different subsets of the full HAR task, having different numbers and combinations of classes, to discover their behavior near their operational limits.

In conclusion, this thesis attempts to reveal the potential and limitations of state-of-the-art, currently available neuromorphic devices, seeking to contribute with valuable insights into the growing field of neuromorphic computing and its potential role in the future of edge computing and real-time, low-power data analysis.

Chapter 2

Background

2.1 Human Activity Recognition

In certain applications, achieving a deep understanding of the user's environment is necessary to accomplish the system's objectives.

Specifically, the recognition of user behavior and activities opens doors to smarter and more effective human-computer interactions. It also paves the way for the development of a new generation of applications in domains ranging from medicine and related fields to self-driving vehicles or security surveillance. For instance, the ability to recognize human actions or behavior could play an important role in the control of prosthetic limbs, conducting fitness analyses of individuals, preventing illnesses, and more.

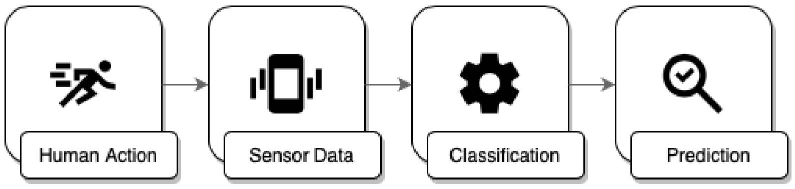


Figure 2.1: Human Activity Recognition task diagram. The motion data, produced by sensors during an activity of a human subject, is processed through a trained classifier. The system's objective is to identify the activity.

This is precisely the objective of the HAR task: to identify and categorize human actions by analyzing data collected from motion sensors applied to the subject.

Historically, this task was very challenging due to the unavailability of off-the-shelf, precise sensors and to the limited computational capabilities of portable devices.
In the past, capturing the required data necessitated the use of uncomfortable and impractical full-body sensor suits, resulting in redundant and inefficient data collection. Furthermore, processing the data in real-time on handheld devices was unfeasible within humanly acceptable times and without draining the device's battery.

However, recent advancements in sensor technology and the widespread adoption of wearable devices, such as smartwatches, have presented an unprecedented opportunity.
These modern devices are equipped with sensors that offer significantly higher precision than their predecessors. Their computational capabilities are excellent in relation to their form factor and their battery is perfectly able to last for more than a day.
Additionally, they have been widely accepted as both practical and fashionable accessories, typically worn throughout the day. This ensures a continuous stream of data that can be exploited to adapt and fine-tune recognition systems for individual users, enhancing their accuracy. Moreover, this opens the door to background analyses that were unfeasible in the past.

These developments in the field of HAR could positively impact a wide range of related applications and pave the way for groundbreaking innovations.
Such innovations have the potential to mark a significant milestone in the integration of humans and machines.

2.1.1 Wireless Sensor Data Mining Dataset

In recent years, researchers groups generated a very large number of datasets tailored for the HAR task.
These datasets typically contain data samples captured through motion sensors, such as accelerometers and gyroscopes, commonly found within Inertial Measurement Units (IMUs) integrated into smartphones, smartwatches, or dedicated hardware devices.

Among the most widely recognized datasets are:

- Opportunity Dataset [21]: Developed for the Opportunity Project in 2010, this dataset includes recordings from various types of sensors and cameras. Its primary objective is to achieve highly precise activity recognition within a breakfast scenario.

- Human Activity Recognition Using Smartphones Dataset [22]: Generated from the recordings of 30 subjects performing activities of daily living, wearing a smartphone on their waist.

- Wireless Data Mining Dataset (WISDM) [23]: Generated in controlled laboratory conditions by recording only smartphone's accelerometer data from 29 subjects performing daily activities, such as walking, jogging, climbing stairs, sitting and standing.

- Actitracker Dataset [24]: This is considered the "real world" version of the WISDM dataset. It is composed of data collected through the homonymous smartphone application, this dataset focuses on monitoring and improving users' health by setting and tracking activity goals.

- PAMAP2 Physical Activity Monitoring Dataset [25]: Generated by recording 9 subjects wearing a specialized network of 3 IMUs on the wrist, ankle and chest, plus a heart rate monitor.

This thesis centers its attention on the updated version of the WISDM dataset [26], which was released in 2019.
In its revised form, WISDM relies not only on data from smartphones, but also from smartwatches. Moreover, in contrast to the original version, which solely relied on accelerometer data, it also incorporates readings from the devices' gyroscope.
From now on, the acronym "WISDM" will refer to the revised dataset.

Label	Class	Label	Class	Label	Class
0	Walking	6	Brushing Teeth	12	Kicking Soccer
1	Jogging	7	Eating Soup	13	Catch Tennis
2	Stairs	8	Eating Chips	14	Dribbling Basketball
3	Sitting	9	Eating Pasta	15	Writing
4	Standing	10	Drinking	16	Clapping
5	Typing	11	Eating Sandwich	17	Folding Clothes

Table 2.1: Summary of the classes included in the WISDM dataset, along with their corresponding labels.

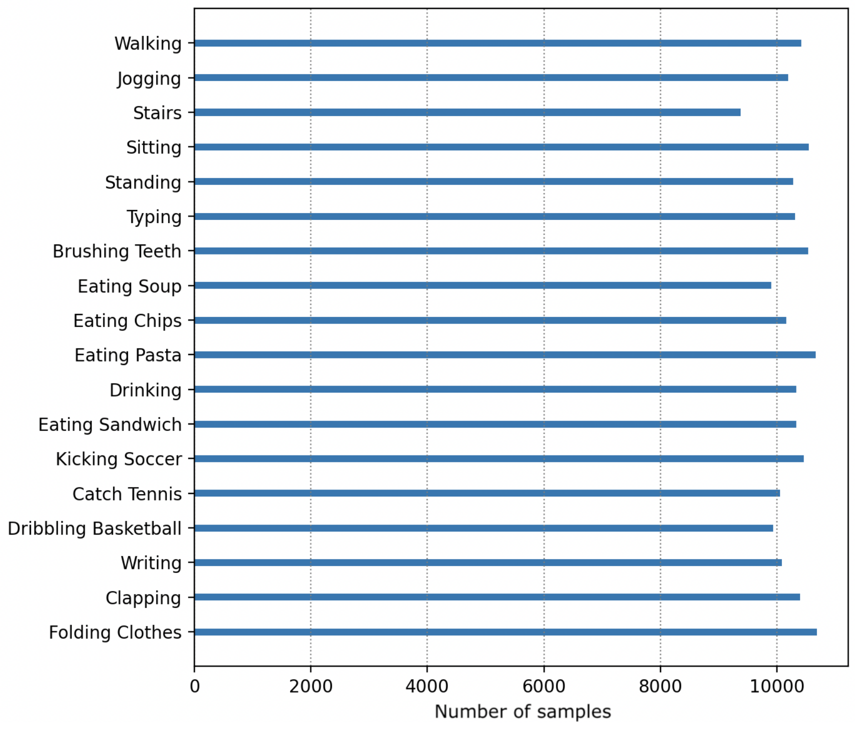


Figure 2.2: Overview of the WISDM class distribution. The x axis shows the number of samples for each class. The classes in the dataset are mostly balanced.

Samples were collected by recording 51 subjects while performing a predefined

set of activities. Each activity had a fixed duration of 3 min, recorded at an acquisition rate of 20 Hz (every 50 ms).
Consequently, the dataset contains 18 nearly balanced classes that are associated with hand-oriented and non-hand-oriented human actions. By comparison, the previous version contained only 6 classes: this shows the extent of the revision, which aimed to expand the dataset, potentially making it the most important for the HAR task.

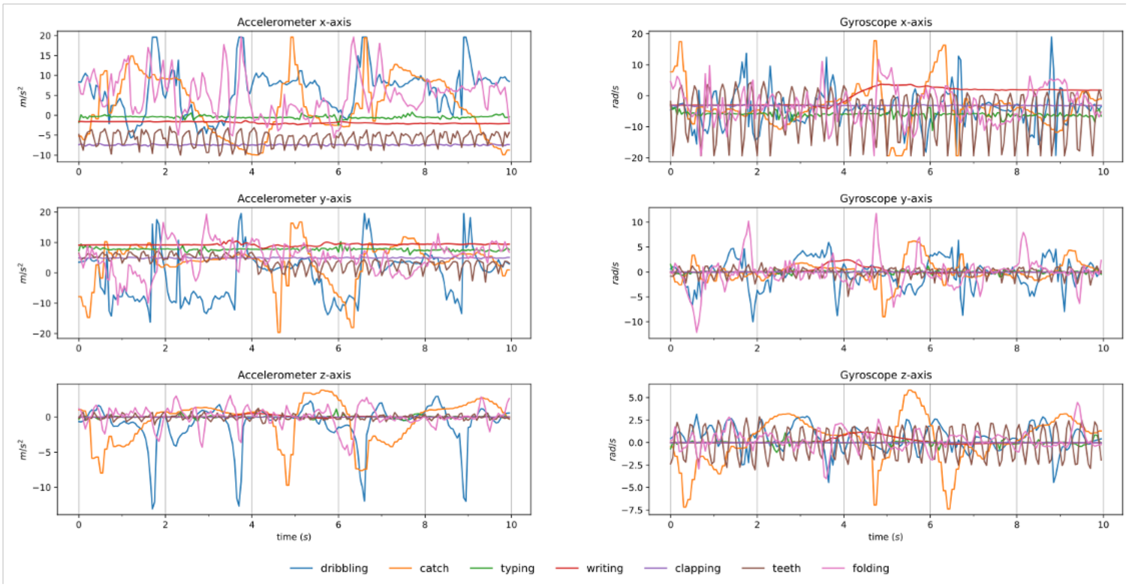


Figure 2.3: Plot of seven randomly selected samples representing different classes from the WISDM dataset. The figure displays the time series data associated with the accelerometer and gyroscope axes.

As mentioned earlier, data consists of sensor measurements from the IMU of either a smartphone or a smartwatch, obtained by recording human subjects performing a specific action.
In particular, each sample is composed of 6 readings for each time step, each associated with the IMU's accelerometer and gyroscope axes.

The dataset used in this thesis was further processed, splitting the 3-minute samples into windows of two seconds. Therefore, each sample is a time series containing 40 timesteps, given the original acquisition rate of 20 Hz.

The WISDM dataset is an ideal choice for developing action recognition or biometric identification models.

2.2 Neuromorphic Devices

2.2.1 DYNAP-SE2

The DYnamic Neuromorphic Asynchronous Processor — ScalablE 2 (DYNAP-SE2) neuromorphic chip is the latest addition to the DYNAP family of neuromorphic chips.
It was developed by the Institute for NeuroInformatics (INI) in Zürich, which is affiliated with ETH Zürich and the University of Zürich (UZH).

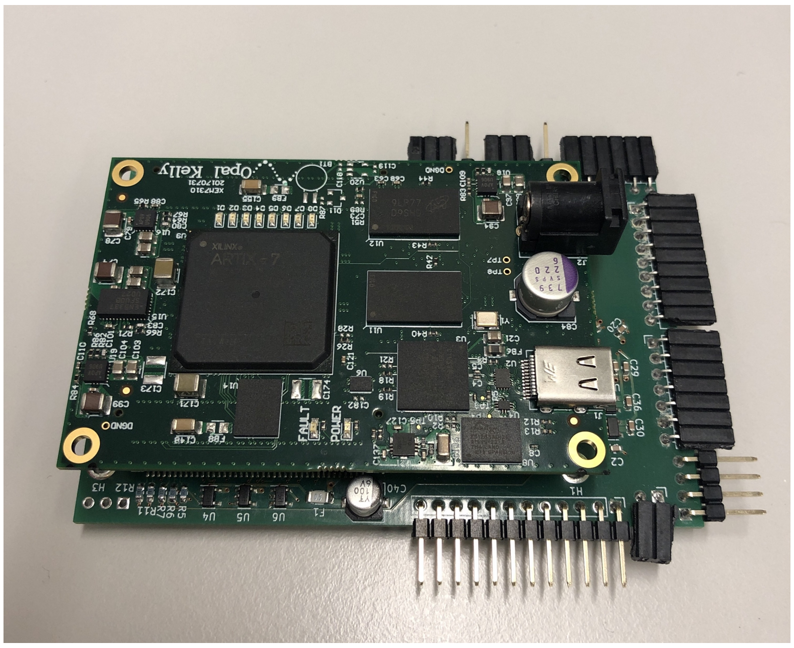


Figure 2.4: A DYNAP-SE2 board. The control FPGA is assembled as a shield.

The primary goal of the DYNAP-SE2 is to emulate the behavior of biological brains and nervous systems.
Its neurons and synapses can be configured to form complex networks, capable of generating intelligence and, therefore, possessing computational capabilities.
This chip is compatible with a wide range of common network models, spanning from feedforward to reservoir and recurrent networks.

The chip is based on a mixed-signal architecture and possesses remarkable ultra-low-power and ultra-low-latency properties.
All the computational units, including neurons and synapses, are analog structures designed to work in the sub-threshold domain, resulting in exceptional power efficiency.
Furthermore, the digital event-based inter-neuron communications rely on a novel and patented asynchronous routing system [27].
This communication scheme diverges from the traditional von Neumann architecture, as memory is effectively distributed across all system components. This design choice results in an in-memory computing architecture that remains unaffected by the limitations of traditional architectures.
This ensures low latency, even for inter-chips communications, and dramatic system scalability.

While the sub-threshold design allows for great power efficiency, it also makes the DYNAP-SE2 more susceptible to the hardware manufacture variability [28], which is commonly present between each produced item. This is reflected in small but chronic inequalities across the behavior of the internal components of the same chip. This phenomenon is defined as "mismatch" [29].

The presence of mismatch can be advantageous for the objective of emulating a biological brain, where natural variations exist among neurons and synapses. However, it can pose significant challenges for certain classification tasks. In practical terms, this mismatch complicates the process of configuring the board to generate the desired behavior.

Going into the chip's specifications, it comprises four neural cores, with each core containing 256 Adaptive Exponential Integrate and Fire (AdExpIF) neurons, making a total of 1024.
Each neuron supports up to 64 input synapses, allowing it to receive information from 64 other neurons, including itself through recurrent connections.

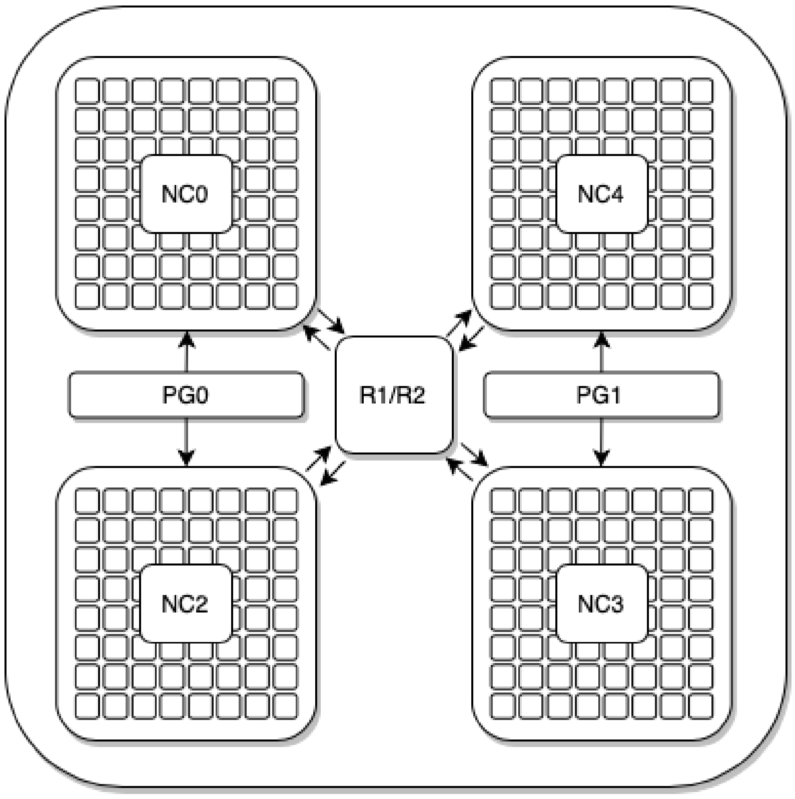


Figure 2.5: A schematic representation of the internal structure of a DYNAP-SE2 chip. The chip is composed of four neural cores (NC), each containing 256 neurons. Two parameter generators (PG) propagate the user-defined configuration to the cores. The intra-core and, eventually, inter-chip is handled by a patented hierarchical routing scheme.

Moreover, each neuron has two dedicated memory components responsible for storing I/O connection configurations:

- Content Addressable Memory (CAM): Holds the configuration details for the input connections. In particular, each entry corresponds to an input synapse and contains information about the source neuron identification tag, plus type, weight, and other settings (such as delay, short-term plasticity, and more) of the synapse.

- Static Random Access Memory (SRAM): Stores the neuron-specific configuration for output events routing. Each entry contains information about the specific chip and core the output events must be forwarded to.

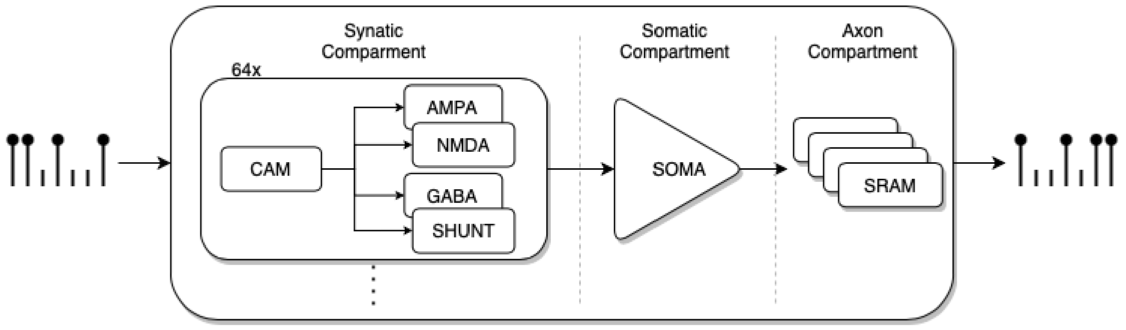


Figure 2.6: A schematic representation of the internal structure of a DYNAP-SE2 neuron. The input events stream is processed asynchronously by up to 64 configurable synapses. Synapses inject current into the neuron's soma, causing a variation of the membrane potential. If the latter surpasses a threshold, the neuron fires, and an output event is generated.

The DYNAP-SE2 is highly configurable and exposes a set of 70 parameters to customize the behavior of neurons and synapses. These parameters include synaptic weights, time constants, and more.
Like in other similar neuromorphic platforms, this high configurability comes with some constraints.
Specifically, the DYNAP-SE2 does not allow for the individual configuration of neurons and synapses but operates at the group level. Generally, users can only define a per-core configuration. Thus, the parameters are shared between the neurons and synapses that reside in the core the parameters were specified for.
This approach also extends to the synaptic weights: only four weight values can be specified per core, and each synapse has a bit mask to enable one or a combination of these values.

A parameter generator, or bias generator, is responsible for converting the digital value of every single parameter to an analog current, that is subsequently supplied to the respective components.

The system is used in conjunction with an FPGA, which serves as the interface to external systems for input and output functions.

Each input event is encapsulated into an Address-Event Representation (AER) and sent to the FPGA, where it is temporarily stored in internal buffers. At the right timestep, the event is propagated in the chip through a series of virtual neurons, that are connected to the physical neurons.

When an input event stream is present, each synapse integrates the incoming events, each with a behavior defined by its type, and injects current into the neuron membrane.
The processor supports four types of synapses: AMPA, NMDA, GABA, and SHUNT. AMPA and NMDA are excitatory synapses and, thus, produce an increment in the membrane potential. On the contrary, GABA and SHUNT are inhibitory synapses, causing a decrement of the membrane potential [30]. AMPA and SHUNT synapses' effects are fast but short-lasting, while NMDA and GABA produce a slow but sustained effect.

When the membrane potential exceeds a specific value, known as the firing threshold, the neuron emits an event.
Each time a neuron fires, it triggers a reset mechanism which is responsible for returning the membrane potential back to the resting state: following each firing event, the neuron enters a refractory period, having a configurable but brief duration.
During the refractory period the membrane current is flushed, and the neuron ignores any input and, therefore, is incapable of firing.

The output event is handled by the routing system, which sends it to the designated output neurons by broadcasting it across the chip and core as configured in the SRAM of the firing neuron.

The FPGA has the capability to intercept the events emitted by the neurons, enabling users to visualize and analyze neural activity.

The DYNAP-SE2 chip stands as one of the most advanced examples of cutting-edge neuromorphic technology, promising unparalleled energy efficiency and enabling both research-driven exploration and broader practical applications [30].

2.2.2 Xylo

Xylo describes a family of neuromorphic devices designed and commercialized by SynSense. The company is considered the world's largest supplier in neuromorphic technologies [31], with a wide range of ultra-low-power and ultra-low-latency neuromorphic solutions, specializing in applications related to vision and audio processing.
SynSense's primary mission is the commercialization of neuromorphic devices, developed relying on two decades of experience coming from associated universities: ETH Zürich and the University of Zürich (UZH).

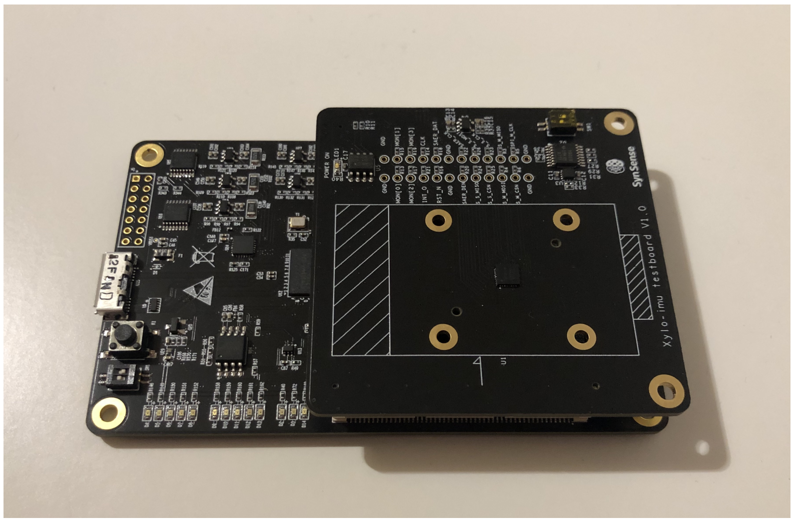


Figure 2.7: A SynSense's Xylo IMU board.

Among all Synsense's offers, Xylo is specifically designed for low-dimensional signal processing. For that reason, it is perfectly suitable for the Human Activity Recognition task.
Xylo's applications range from keyword spotting and environmental recognition to medical analyses and smart security [32, 33].

At the time of writing this thesis, the Xylo family comprehends several models, such as Xylo Audio and Xylo IMU, specialized in various applications.

The primary difference among them can be identified in the type of input sensor they are equipped with: Xylo Audio has an analog/digital microphone, while Xylo IMU contains a three-axis accelerometer.

Considering the potential of directly utilizing the onboard IMU for the HAR task, it becomes evident that Xylo IMU would have been the most suitable choice for the objectives of this thesis. Unfortunately, the Xylo IMU was only released in October 2023 and this thesis work was at too late a stage. We leave the exploration of Xylo IMU solutions to later work. For that specific reason, this work focuses on the Xylo Audio chip.

The on-chip input sensor enables Xylo to be used as an all-in-one solution, which integrates both the sensorial input and neuromorphic computational capacity: data generated from the sensorial compartment is converted to a stream of spiking events by the analog front-end and directly processed by Xylo's SNN core.

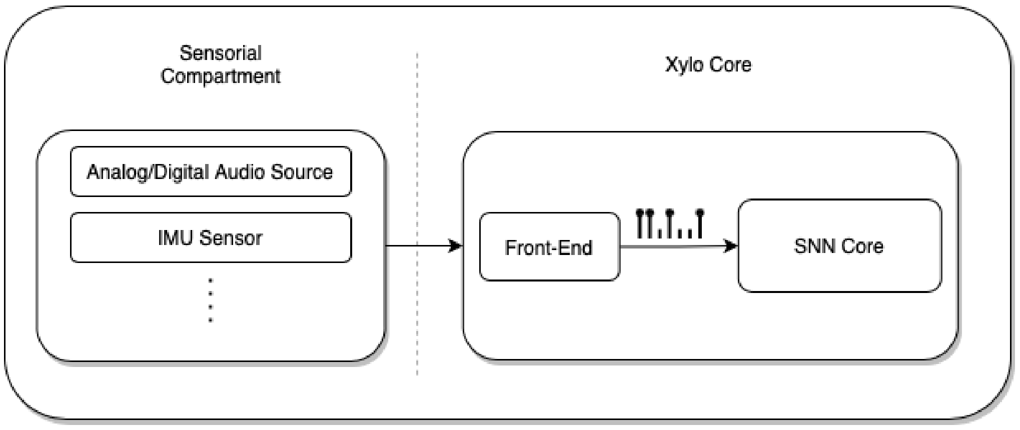


Figure 2.8: A high-level representation of the internal structure of a Xylo device. The input data obtained from the sensorial compartment is processed by Xylo's front-end. The front-end filters and encodes it producing a stream of events. The input spike stream is then processed by Xylo's SNN core.

At the core of the Xylo devices is a fully digital synchronous neuromorphic processor, with a very wide configurability, that supports the most common spiking neural networks, such as feed-forward, recurrent, reservoir, or other complex custom neural network structures.

Xylo V1 and Xylo Audio, the first released, support up to 16 input channels, 1000 hidden neurons and 8 readout channels, conveniently available for classification purposes.
The number of hidden neurons is lower for the Xylo IMU variant, supporting up to 496 hidden neurons, but offering a higher amount of readout channels, that is 16. However, the higher hidden neuron count in Xylo Audio comes with a very strong limitation on the neuron's fan-out, which is limited to 64 per neuron, against the 512 maximum output connection for the Xylo IMU.

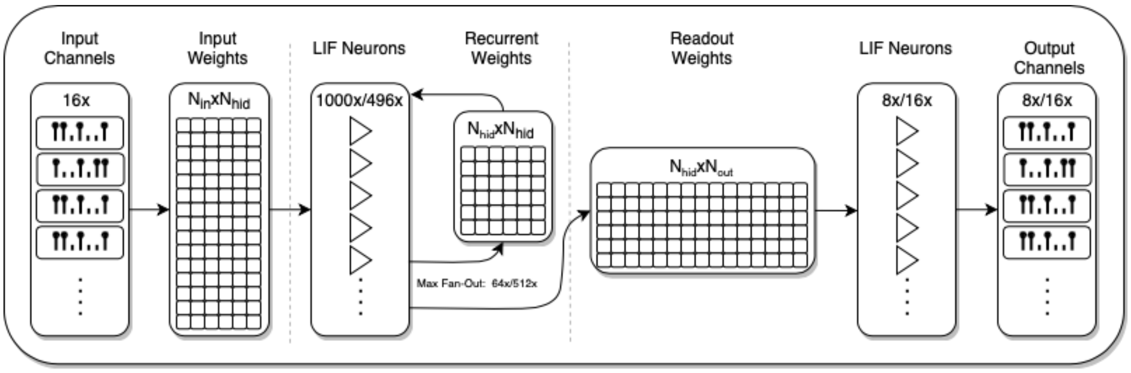


Figure 2.9: A schematic of the logical architecture of a Xylo device. It is designed to support a large set of network architectures. Each component in the schematic shows the amount available for each resource, with the distinction between Xylo Audio and Xylo IMU platforms.

The hidden neuron model built in Xylo is the Current-Based Leaky Integrate-and-Fire (CuBa-LIF) spiking neuron.

Input events from pre-synaptic neurons are weighted by multiplying them with the 8-bit synaptic weights.
Xylo is able to handle multiple events at the same time: each input channel can transport up to 15 events per time-step, while each neuron can fire a maximum of 31 times per time-step, generating the same amount of output events.
The 16-bit synaptic state stores the global sum of all the input contributions.
Some of the Xylo models support a secondary synaptic compartment, which is unsupported by the current version of the software library but allows for future expansions.

The synaptic states are added together with the neuron state, causing an increase in the membrane potential. The latter is encoded, as for the synaptic states, using 16-bit.

If the membrane potential goes above the neuron-specific configured threshold, the neuron fires and generates one or multiple output events.
The membrane reset is triggered at each firing event and causes the reduction of the neuron's membrane potential. This operation is implemented as a subtraction of a specific configurable value.

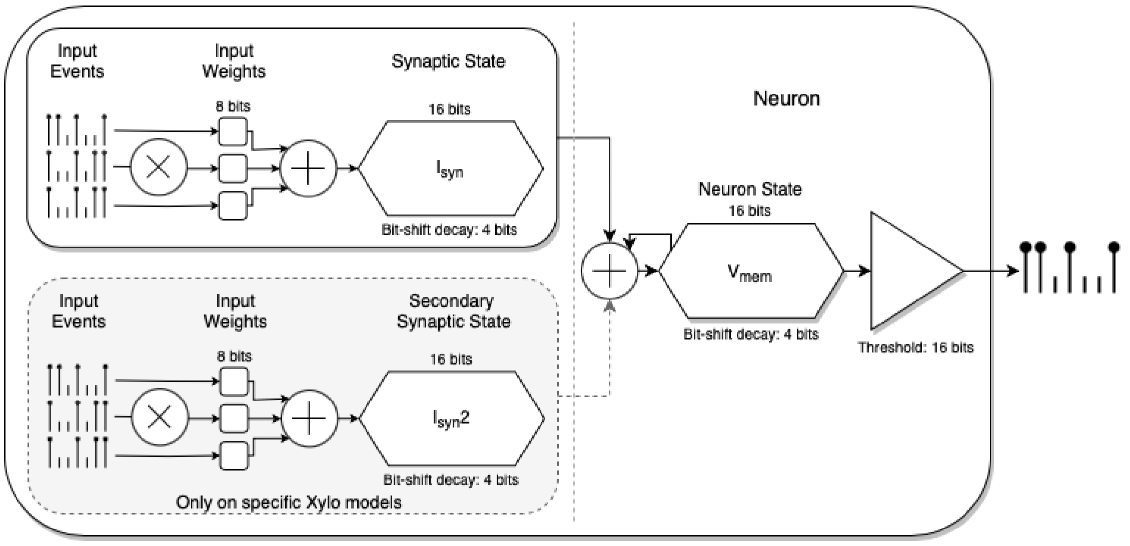


Figure 2.10: A schematic representation of a Xylo's neuron. The pre-synaptic neuron events are multiplied by the synaptic weights. All the input contributions are added together and produce an increase in the neuron's membrane potential. If this value surpasses the threshold value, one or multiple events are emitted.

The leaky trait of the neuron is implemented through an approximation of exponential decay, called bit-shift decay. Executing bit-shifts is way more efficient on digital devices than computing exponentiations.
This way, Xylo can keep the power requirements very low, but at the same time is able to effectively approximate the expected neuron leakage behavior.

Xylo allows for complete configurability of each neuron and synapse, by tuning their parameters individually (e.g. time constants, neuron threshold, decay behavior, and more).

In conclusion, Xylo is an advanced commercial chip, likely the first of its kind to reach the market. It delivers high performance while maintaining its ultra-low-power and ultra-low-latency characteristics, potentially offering new solutions for everyday applications.

2.3 Tools

2.3.1 Rockpool

Rockpool is an open-source software library developed by SynSense.
It provides a robust framework for efficiently building, training, simulating, and deploying both traditional and spiking neural networks.

Rockpool supports the gradient-descent training paradigm out-of-the-box, relying on standard back-propagation. It exploits surrogate gradient functions to extend this paradigm to the spiking domain [32].

This framework offers support for various simulation backends, and it includes essential modules for composing complex networks.
As of now, Rockpool is compatible with Torch, JAX, and Numpy, and it leverages their GPU acceleration capabilities, ensuring reliability and high-speed simulations.

Notably, Rockpool seamlessly supports both the Xylo and DYNAP-SE2 platforms.
Xylo was the first device to be compatible with Rockpool. The latter offers a bit-accurate simulator that allows users to test network performance without the need for physical hardware.

To make a network designed within Rockpool compatible with hardware, it must be processed through a conversion pipeline. This process involves mapping the network's modules to specific hardware resources, quantizing the weights to match hardware characteristics, and ultimately converting the network into a hardware configuration that can be directly deployed on the chip.

During 2023, Rockpool's compatibility was extended to include the DYNAP-SE2 chip.
This expansion was based on a research [34] that outlines a novel network training and deployment pipeline tailored to the DYNAP-SE2 platform. The paper explores an unsupervised weight quantization method to optimize network parameters and

proposes a training approach aimed at achieving robust models that can handle hardware mismatch, a characteristic of the DYNAP-SE2.

Like in the case of Xylo, Rockpool incorporates a dedicated simulator for the complex neuron dynamics of the DYNAP-SE2, as well as all the necessary tools for network conversion and deployment.
This ensures smooth integration with these neuromorphic platforms, making it a powerful tool for researchers and developers.

2.3.2 Tonic

Tonic [35] is an open-source software library developed with the support of SynSense. It provides access to publicly available event-based audio and vision datasets, offering a wide range of functions for event transformation.

Notable features include the ability to optimize data loading using caching techniques, easily batch and slice event-based datasets, and support for advanced event-based data manipulation.

This includes operations such as probabilistic event dropping, time cropping, noise addition or removal, and downsampling. Additionally, Tonic enables users to convert static images into various event-based representations.

Tonic seamlessly integrates with Rockpool, extending its functionality to incorporate the features mentioned above.

2.3.3 Samna

Samna, a software library developed by SynSense, is a crucial component in interfacing with SynSense's hardware platforms, aiming to provide a comprehensive and powerful toolset for users.

Samna is compatible with the entire range of SynSense products, including those from the DYNAP series, such as DYNAP-CNN and DYNAP-SE2. It gives users the ability to manage the board configurations, update firmware, deploy network models, and monitor the power consumption during operation.

Furthermore, Samna offers an extensive suite of tools necessary for not only controlling but also observing the behavior of connected devices. Users can send and receive events from the board, compose complex event processing graphs, and leverage a dedicated graphical user interface for visualizing these events effectively. This versatility makes Samna an indispensable resource for optimizing and controlling the functionality of SynSense's hardware offerings.

2.3.4 Neural Network Intelligence

NNI, which stands for Neural Network Intelligence, is an open-source toolkit designed to support researchers, engineers, and data scientists working in the field of machine learning.
Developed by Microsoft, NNI is a comprehensive tool that aims to automate the processes and simplify the complexities involved in neural network architecture search (NAS) and hyperparameter optimization (HPO), making it easier to design and fine-tuning effective deep learning models [36].

In the context of this thesis, NNI assumes a pivotal role, primarily focusing on hyperparameter optimization. In fact, it can be used to automatize the complex task of searching for optimal hyperparameters, by providing an extensive set of optimization techniques.

The tool configuration for the HPO task is straightforward. The user has to define the hyperparameters search space, which defines the interval of validity for the parameters that must be optimized, and to configure the experiment, for example by choosing the tuning algorithm, the maximum experiment duration, in time or in number of trials, and other relevant settings. Moreover, the user needs to provide a default metric, i.e. model accuracy, that NNI will try to maximize or minimize, according to the user configuration.
Running the experiment starts the optimization process: the tuning algorithm wisely chooses the parameters to test in each trial, trying to find an optimal solution based on the model behavior.

Noteworthy functionalities of NNI include model compression through quantization and model pruning, as well as comprehensive support for feature engineering.

NNI provides a user-friendly interface and seamless integration with popular machine learning frameworks such as TensorFlow, PyTorch, and Keras.

Moreover, NNI can be used on local machines, or to orchestrate the operations within distributed environments. The capability to parallelize tasks across multiple machines significantly enhances efficiency, saving valuable time, especially when working with very large models.

In the end, NNI simplifies the complex process of developing and tuning high-performance neural networks. Its user-friendly design and comprehensive functionality make it a valuable asset in the research and development of machine learning models.

Chapter 3

Materials and methods

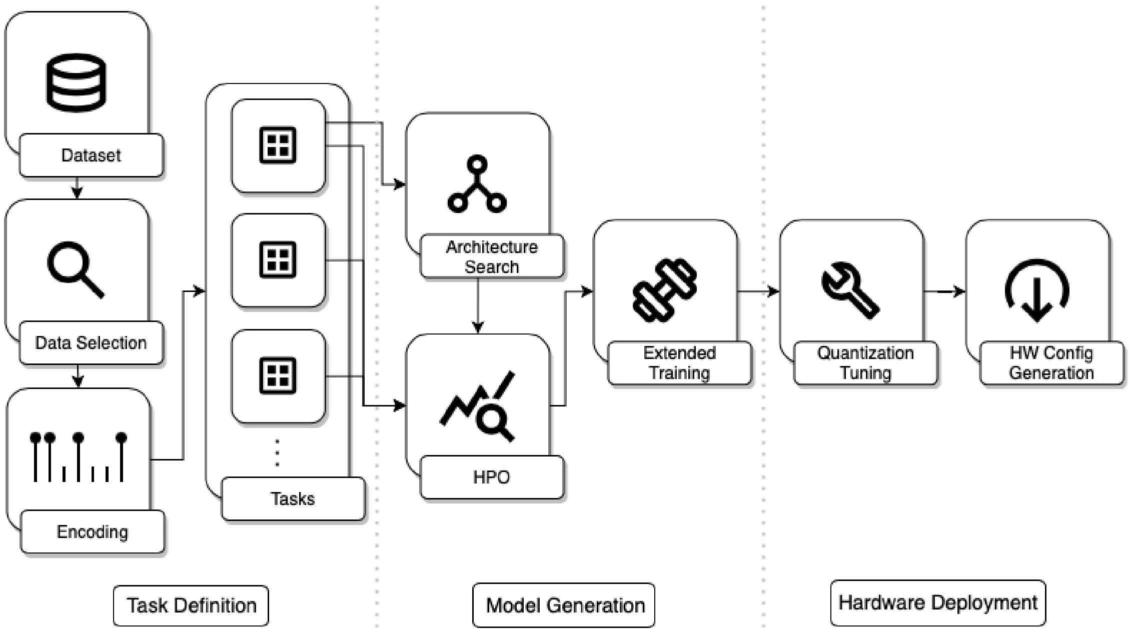


Figure 3.1: A schematic showcasing the proposed end-to-end pipeline. It comprises three macro-sections: 1) Task Definition, including the Data Selection and Encoding stages, 2) Model Generation, including Architecture Search, Hyperparameter Optimization (HPO), and Extended Training stages, and 3) Hardware Deployment, including Quantization Tuning and Hardware Configuration Generation.

The goal of this thesis, namely the exploration of the capabilities and limitations of target neuromorphic devices, requires the production of optimized models that can be easily implemented and deployed on hardware.

Discovering optimal parameter configurations, running complete and extensive training sessions, and tuning HW configuration generation are mandatory steps to extract the best performance possible from each generated model: the incorrect behavior of each of these processes could cause dramatic performance loss that could heavily affect the accuracy of the final model and, thus, the reliability of the experiment.

To achieve a consistent and repeatable state of optimal model generation, this thesis proposes an end-to-end pipeline consisting of several steps, as depicted in Figure 3.1:

- Data Selection & Task Definition: As the first step, it is responsible for defining the tasks that will be used for the benchmark. Subsets of the raw dataset, composed of combinations of classes of various sizes and, therefore, with various levels of complexity, are selected by employing a metric that can measure the separability of the classes in the selected combination. Each selected subset of the dataset will be associated with a single task.

- Encoding: The subset of data from each task is encoded as sparse, binary events using an encoding technique. This is a requirement when working with SNNs.

- Architecture Search: This step is responsible for selecting the best neural network architecture for a specific task. The best architecture will be used for the rest of the pipeline's steps.

- Hyperparameter Optimization: The best model hyperparameter configuration for each task is sought during this step.

- Extended Training: The final model for each task is obtained by training over a longer period the best configuration identified by the HPO experiment. This should ensure that the best possible model accuracy is achieved.

- Quantization Tuning: The trained model weights are manipulated to ensure the best performance after quantization.

- Hardware Configuration Generation: Finally, the network parameters are quantized, translating them into the device parameter domain, and a deployable configuration is generated.

3.1 Data Selection

Given the primary objective of this thesis, which is to explore the performance of neuromorphic devices as the complexity of the processed data changes, the first step was to obtain a collection of tasks of controlled and increasing complexity.

The WISDM dataset is extensive and contains data from both smartphones and smartwatches. In any case, the data from smartphones are expected to be less accurate and have a lower dynamic range than the information extracted from smartwatches.
This difference does not depend on the different IMU units supplied with the devices, but is mainly due to the radically different positioning of the device: while the smartphone used to record the subjects' movements was positioned on the waist, the smartwatch was positioned on the wrist, whose movements are more pronounced and have greater variability.
Therefore, a classifier will potentially be unable to distinguish between the eating actions, which have a high degree of similarity if seen from the smartphone's perspective.
For that reason, I only considered the smartwatch-generated samples, to get a more compact and homogeneous starting dataset and to slightly simplify the classification task.

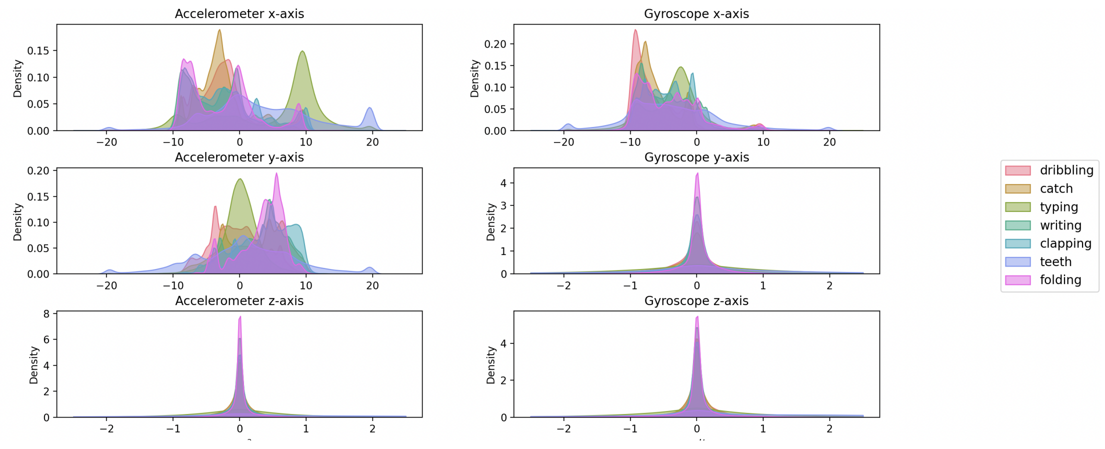


Figure 3.2: KDE visualization for the 7 classes belonging to the general, hand-oriented subset of the WISDM dataset. As clearly visible, classes are heavily overlapping.

The different classes of which the WISDM dataset is composed are generally highly overlapping, as shown by the Kernel Density Estimation analysis in Figure 3.2.

This phenomenon causes certain groups of classes to be harder to classify if compared to others.
Therefore, the complexity of a task can be selected by finding different combinations of a certain number of classes and measuring their separability through a metric that can produce observable and consistent estimations. The data belonging to the classes in each combination is then joined to form multiple subsets, which have a known complexity relative to the original dataset.

Thus, each task is associated with a specific subset of classes and its difficulty level is determined by calculating the separability score of its classes.

The data selection process began with the creation and analysis of methods for calculating the separability of various classes in a dataset, to find one suitable for the task.

This led to the development of two metrics, respectively based on exploiting the Dynamic Time Warping (DTW) distance metric and the Küllback-Leibler Divergence (KLD), also used as a distance metric.

3.1.1 Dynamic Time Warping Class Separability Metric

The first metric tested was the one based on the DTW distance. Dynamic Time Warping [37] is a distance metric that allows for better performance in time series comparison, against simple Euclidean distance [38, 39].

Euclidean distance calculates the distance between two points as the length of the linear segment that connects them. While it performs well in most of the low-dimensional tasks, it is slightly imprecise when applied to high-dimensional data. In particular, in the case of time series, the Euclidean distance metric is not able to adapt to desynchronized or compressed/dilated data.

Dynamic Time Warping tries to take into account these factors when calculating distance. In fact, applying DTW is equivalent to minimizing Euclidean distance between aligned time series under all admissible temporal alignments [40].

This enables the metric to be able to adapt to misaligned data, even when the sampling rate changes dynamically.

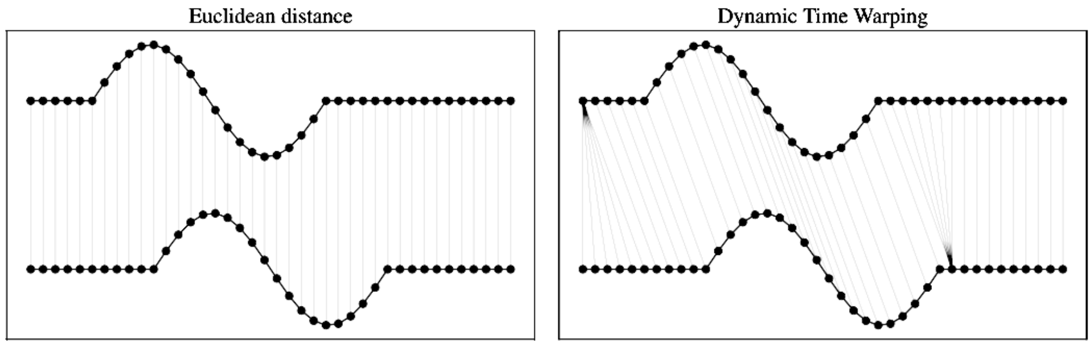


Figure 3.3: Comparison between the behavior of Euclidean and Dynamic Time Warping distance metrics when applied to misaligned time series. DTW can adapt to data desynchronization and obtain more coherent measurements. Source: [40].

The class separability metric based on DTW works as follows:

1. A per-class average sample is obtained by calculating the median values for each time step, for each of the class samples. This results in a single sample containing 40 timesteps and six dimensions.

2. A class distance matrix is generated by calculating the distance between the different classes' median samples, comparing them two by two using DTW over all six dimensions.

3. After selecting the value of the variable n, which controls the size in terms of classes in the sets to analyze, the system generates all the possible combinations of n class labels.

4. The final separability score for each combination is then calculated as the sum of the distance values between each pair of classes it contains. High values mean good separability.

While, theoretically, this approach could yield sufficiently accurate measurements, it is heavily susceptible to the lack of synchronization between samples belonging to the same class.
This issue, inherent in the WISDM dataset, was exacerbated by the windowing applied to the dataset, dividing the 3-minute samples into 2-second time series. This misalignment causes the dimensions of the average sample to be straight lines. Therefore, most of the information is contained in the amplitude of such signals, while the information based on the shape of the signal is almost completely lost.

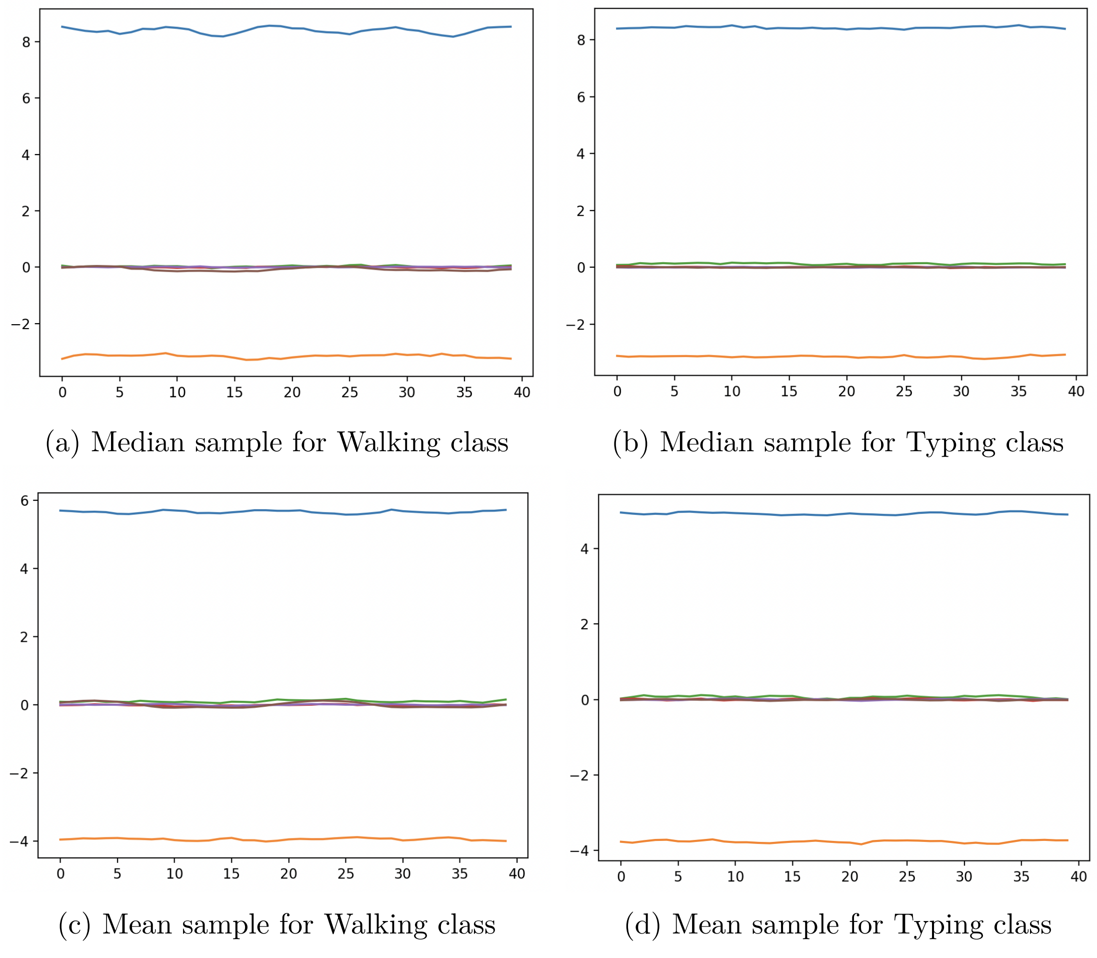


Figure 3.4: Comparison between the average samples for two random classes obtained using the median and mean techniques. The average sample calculated using the median seems to keep some of the shape information and obtain a better amplitude dynamic range.

A possible solution to this problem is to calculate the inter-class distance as the sum of the distances between all their samples, thus exploiting better the capacity of DTW to work on desynchronized signals. However, this approach was discarded because of its cost in terms of computational time and resources.

3.1.2 Küllbach-Leibler Divergence Class Separability Metric

The DTW-based metric issues resulted in the necessity to explore other techniques. For this reason, the second approach tested, based on the Küllbach-Leibler Divergence, deviates from a purely geometrical strategy and focuses on the statistical perspective, analyzing the sample distributions of each class.

Originating from Information Theory, Küllbach-Leibler Divergence (KLD) [41] is a type of statistical distance.
In fact, it is a measure of how much a probability distribution is different compared to another one, taken as a reference.
Given two probability distributions, P and the reference Q, a simple interpretation of KLD is the measure of the information lost when Q is used as an approximation for P.

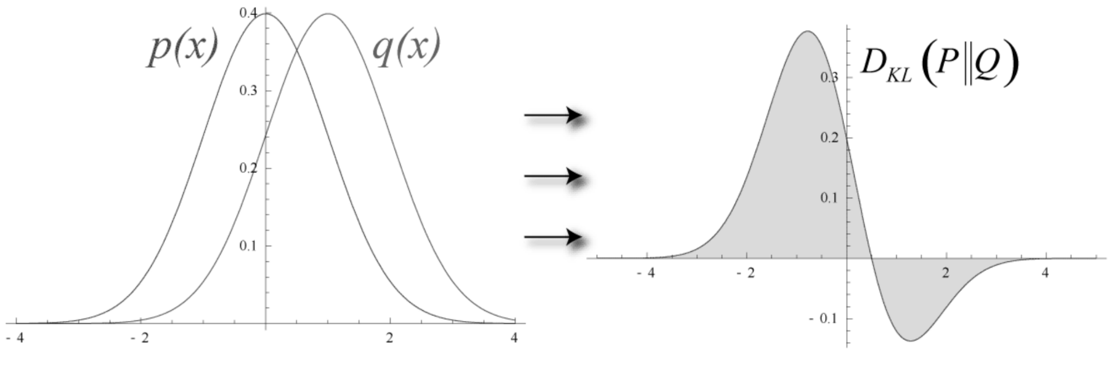


Figure 3.5: Example of the Kullback-Leibler divergence applied on Gaussian probability distributions. The left figure shows two example Gaussian PDFs. Integrating the highlighted area, in the right figure, returns the KLD value. Source: [41].

KLD cannot be applied directly as a distance metric because of its non-symmetry:

calculating the divergence of P from Q returns a different value than the one obtained by calculating it in the opposite direction, of Q from P.
For such specific reason, the algorithm for the class separability handles the distance calculation in a slightly different way, compared to the one based on DTW: when measuring the distance between two classes, the KLD values between their dimensions are calculated in both directions and then added together.

Moreover, this algorithm makes use of Kernel Density Estimation (KDE) [42], which is a way to estimate the probability density function of a certain set of data. KDE produces continuous functions, in contrast with the discrete histograms, obtained with the binning technique.

The algorithm starts by substituting each data sample with a specific and configurable continuous kernel function. Successively, these kernel functions are added together, producing a close and smooth estimation of the real probability density function for the input data.

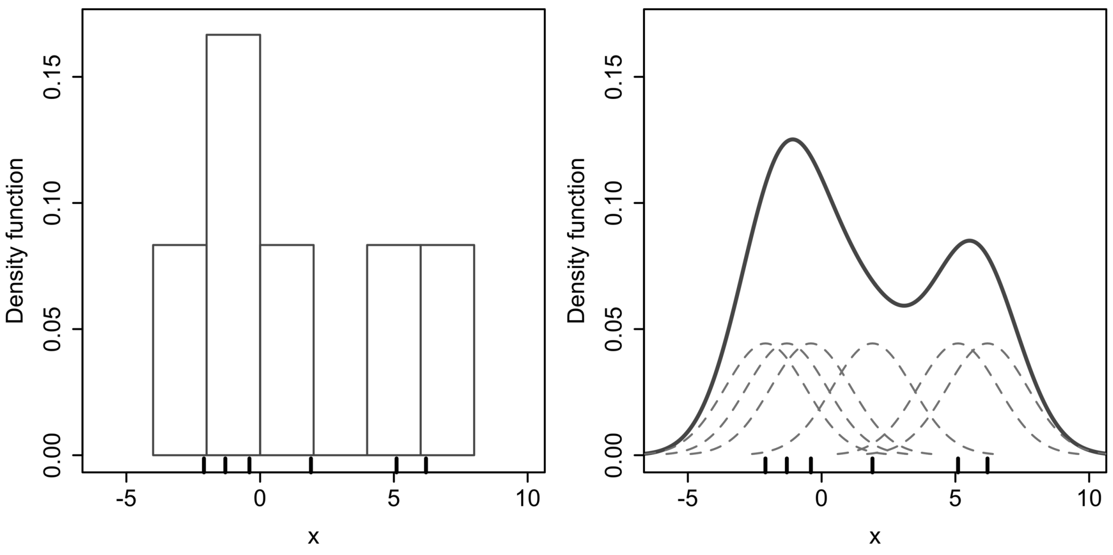


Figure 3.6: Comparison between discrete histograms and Kernel Density Estimation to visualize data distribution. The figures show the histogram and the KDE calculated on the input data, represented as small ticks. Source: [42].

The KLD-based class separability metric is calculated as follows:

1. To make the process efficient, the algorithm starts by calculating the KDEs for each class and all the data dimensions, finally storing them in a map.
2. A class distance matrix is generated by calculating the distance between the KDEs of each pair of classes using KLD, considering all the data dimensions. Given the non-symmetry of KLD, the final score between each pair is calculated in both directions and added together.
3. After choosing the value of the variable n, which controls the size in terms of classes in the sets to analyze, the system generates all the possible combinations of n class labels.
4. The final separability score for each combination is then calculated as the sum of the distance values between each pair of classes it contains. High values mean good separability.

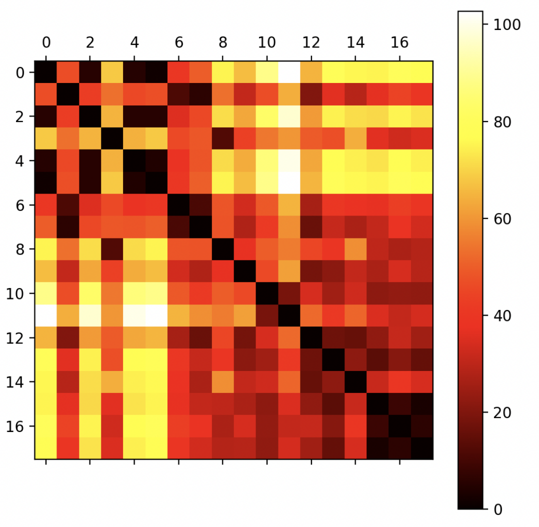
3.1.3 Metrics Comparison

The next step was to compare the identified metrics, evaluating their effectiveness and selecting the most accurate one to use for defining tasks.

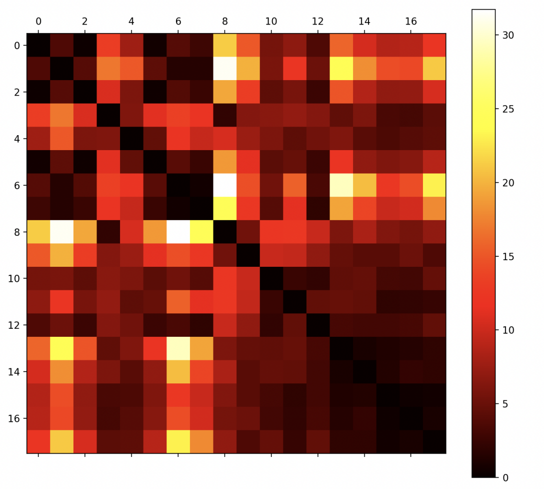
Figure 3.7 shows the distance matrix that aggregates all the measurements for each pair of classes.

Both of the metrics seem able to identify some common patterns.
In fact, they agree on the very high overlap between classes 0 and 2, associated with "Walking" and "Stairs".
Moreover, in both of the matrices, the bottom right section is darker than the rest, highlighting a very high similarity between each other the classes from label 12 to 17 (in order "Kicking Soccer", "Catch Tennis", "Dribbling Basketball", "Writing", "Clapping", "Folding Clothes").

However, they disagree on the most separable class pair: the DTW-based method identifies the pairs (0: "Walking", 11: "Eating Sandwich") and (5: "Typing",



(a) DTW



(b) KLD

Figure 3.7: Distance matrices generated employing the class separability metrics based on DTW and KLD. The class-label associations are listed in table 2.1. Lower scores mean higher class overlap, thus lower separability. DTW tends to overshoot the measurements, but the metrics share some common patterns.

11: "Eating Sandwich") as the best, while the KLD-based method selects the pairs (1: "Jogging", 8: "Eating Chips") and (6: "Brushing Teeth", 8: "Eating Chips").

Generally, the matrix obtained with the DTW-based method seems to show higher separability values, compared to the KLD-based measurements.

As a further comparison between the two methods, I decided to directly test them for their original purpose, that is to define the data subsets associated with the tasks for this thesis work. The two algorithms have been tested searching for the best and worst separable class combinations for the task with two classes.

In particular, the accuracy of their measurements has been verified by visual analysis of the KDE plots, calculated on the data subset relative to the task.

The two-class task provides great information on the metrics performance with the big advantage of still being human-intelligible.

The tables 3.1 and 3.2 compare the 5 best and worst combinations generated by the DTW-based and KLD-based algorithms.

The scores produced by the two metrics are not directly comparable, as they

DTW		KLD	
Class Combination	Score	Class Combination	Score
(Typing, Eat. Sandwich)	**102.703**	(Brush. Teeth, Eat. Chips)	**31.733**
(Walking, Eat. Sandwich)	102.625	(Jogging, Eat. Chips)	31.245
(Standing, Eat. Sandwich)	100.171	(Brush. Teeth, Catch Tennis)	29.402
(Stairs, Eat. Sandwich)	97.422	(Eat. Soup, Eat. Chips)	24.451
(Typing, Drinking)	88.586	(Jogging, Catch Tennis)	24.288

Table 3.1: Ranking of the five most separable class combinations for the two classes task. Separability decreases from top to bottom.

DTW		KLD	
Class Combination	Score	Class Combination	Score
(Walking, Typing)	**1.237**	(Walking, Stairs)	**0.330**
(Writing, Folding Clothes)	2.890	(Stairs, Typing)	0.433
(Standing, Typing)	4.278	(Writing, Clapping)	0.465
(Walking, Standing)	4.843	(Writing, Folding Clothes)	0.511
(Walking, Stairs)	5.513	(Brushing Teeth, Eating Soup)	0.554

Table 3.2: Ranking of the five least separable class combinations for the two classes task. Separability increases from top to bottom.

measure different things. So, it only makes sense to compare scores generated by the same algorithm.

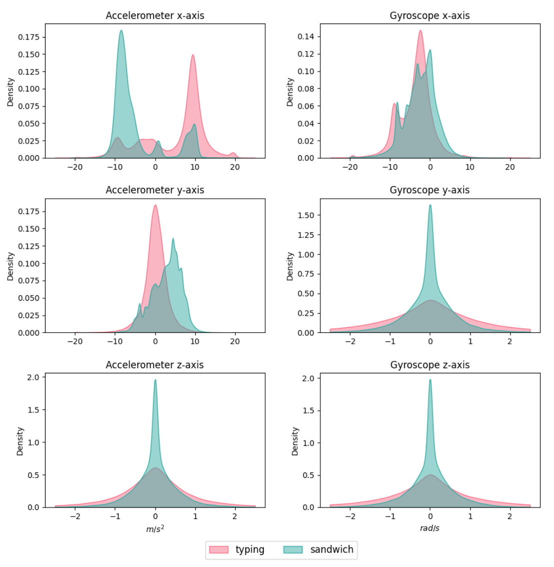
We can notice the two algorithms agree on some of the combinations for the worst case: they share the pairs (Walking, Typing) and (Writing, Folding Clothes) in the top five of the less separable combinations detected.

Figure 3.8 shows the KDE for the most and least separable class combinations that have been selected by the algorithms.

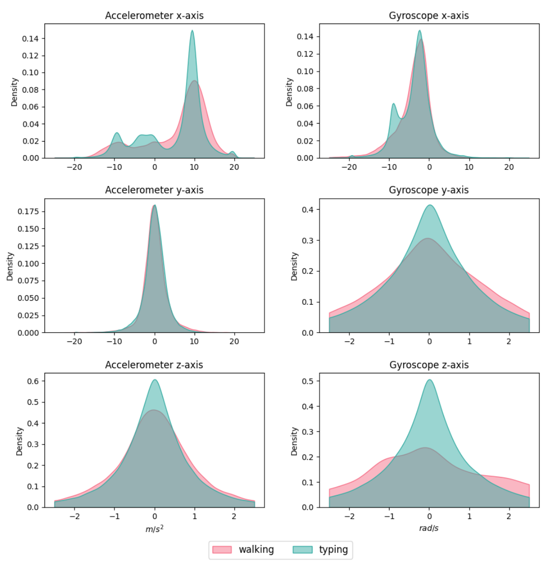
While both methods produce good results, from a simple visual analysis emerges that the combinations selected by the KLD-based method are likely more accurate, compared with the ones from the DTW-based algorithm. The latter is paying a heavy price because of the approximation based on using the class median for comparison. Anyway, this was a forced choice, being it was the only way to make the algorithm computationally feasible.

In particular, the worst-case combination obtained with KLD is far better than the other: the class distributions for most of the accelerometer's and gyroscope's axes almost match perfectly between the two identified classes.

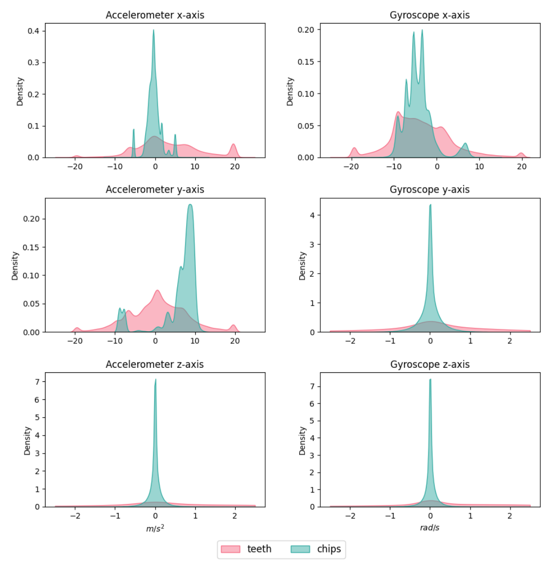
Given the better performance generated by the KDL-based metric, the task definition process fully relied on it.



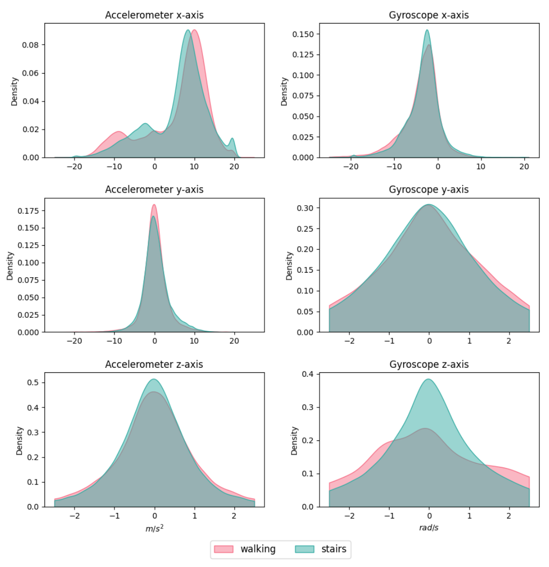
(a) DTW Best Separable



(b) DTW Worst Separable



(c) KLD Best Separable



(d) KLD Worst Separable

Figure 3.8: KDE comparison between DTW-based and KLD-based best and worst separable class combinations, for the two classes task. The KLD-based metric was able to find better combinations.

3.1.4 Tasks Definition

The final phase of the Task Definition macro-area of the pipeline consists of defining tasks based on the results of the data selection process.

As a recall, our objective was to define tasks of various and controlled complexity that will be then used to benchmark the target neuromorphic devices.

Figure 3.9 provides a summary of the task definition process.
The process starts with the generation of all the possible n-sized class combinations. A class separability score is assigned to such combinations using the KLD-based metric algorithm, as explained in the section 3.1.2.
Finally, class combinations are ranked by sorting them based on their scores. The best and worst combinations are selected as tasks.

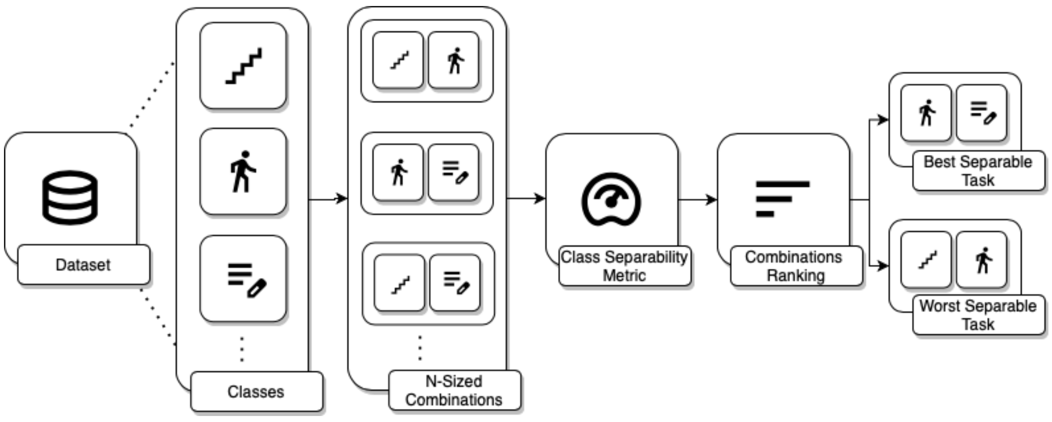


Figure 3.9: Overview of the task definition processes. All the fixed-size combinations of classes are generated and ranked exploiting a class separability metric. The best and worst combinations are selected as tasks.

On the one hand, analyzing the performance of a model on the worst separable combination of classes should cover the worst-case scenario for any classification task, with very complex and overlapping datasets.
On the other hand, the best separable class combination should represent the best scenario, with simple datasets, making the classification task easier.
The size of the class combinations should be able to control the intensity of each situation, with task difficulty linearly increasing with the increase of the number of

classes.

Finally, running the benchmarks on the best and worst combinations of classes should provide useful insights into neuromorphic devices' performance comparing them on the classification of the full range of data distribution that can be found in general usage.

To get a good coverage of various difficulty levels, I considered four sizes of class combinations, that should mimic the size of common classification tasks.

Overall, the task definition process generated eight tasks, associated with the best and worst separable combination of two, three, four, and seven classes.

Table 3.3 shows the class combinations selected for each task employing the KLD-based metric.

Best Separable Combinations	
Task	Combination
2CB	Brushing Teeth, Eating Chips
3CB	Brushing Teeth, Eating Chips, Catch Tennis Ball
4CB	Jogging, Brushing Teeth, Eating Chips, Catch Tennis Ball
7CB	Jogging, Brushing Teeth, Eating Soup, Eating Chips, Catch Tennis Ball, Dribbling Basketball, Folding Clothes

Worst Separable Combinations	
Task	Combination
2CW	Walking, Stairs
3CW	Walking, Stairs, Typing
4CW	Catch Tennis Ball, Writing, Clapping Hands, Folding Clothes
7CW	Eating Sandwich, Kicking Soccer, Catch Tennis Ball, Dribbling Basketball, Writing, Clapping Hands, Folding Clothes

Table 3.3: Summary of the generated tasks. The task acronym consists of the size of the task, in terms of the number of classes, and a letter (B, W) that defines whether the task is associated with the best or worst combination of separable classes.

3.2 Spike Encoding

One of the most advantageous neuromorphic computing features is sparsity: instead of relying on continuous signals propagated throughout the network's layers, spiking neurons communicate using sparse events. While this is great from the power efficiency perspective, it makes it impossible to directly process the readings from most of the modern sensors. In order to use traditional continuous time series with neuromorphic hardware, they must be converted to collections of events through an encoding process.

This process is called spike encoding and its purpose is to generate, from any input signal, a set of events or spikes, that can be processed by the spiking neurons that compose neuromorphic devices [43].

Among the wide collection of available methods for spike generation, we can highlight:

- Rate encoding: Generates a Poisson spike train with a frequency that depends on the intensity of the input.
- Delta encoding: Generates spikes only when the difference between two subsequent time steps exceeds a threshold value.
- Time To First Spike (TTFS) encoding: Encodes the information in the latency between the input propagation and the first spike of the neuron.
- Phase encoding: Encodes the information as the time difference between a spike and the oscillation of a global reference. This simulates what happens in the hippocampus and olfactory system, where the generated spike patterns are correlated with periodic, internally generated phase oscillations.
- Burst encoding: Simulates the behavior of neurons thalamus cortex or auditory system that seems to communicate through bursts of spikes.

All the briefly presented encoding techniques have strengths and weaknesses that heavily depend on the use case in which they are employed.

The analysis of the performance of most of these was the principal objective of the research published in [44].

The researchers compared a large number of encoding types by employing them for two very specific use cases, one of which involves a classification task on the WISDM dataset, the same dataset selected for this thesis project.

Phase encoding, compared with the others by utilizing Mutual Information and Efficiency metrics, was found to be the best for the WISDM dataset, being able to convey the most information most efficiently.

The dataset used in this thesis was kindly encoded and provided by the authors of the research cited above.

Figure 3.10 shows two random samples, belonging respectively to the "Walking" and "Stairs" classes.

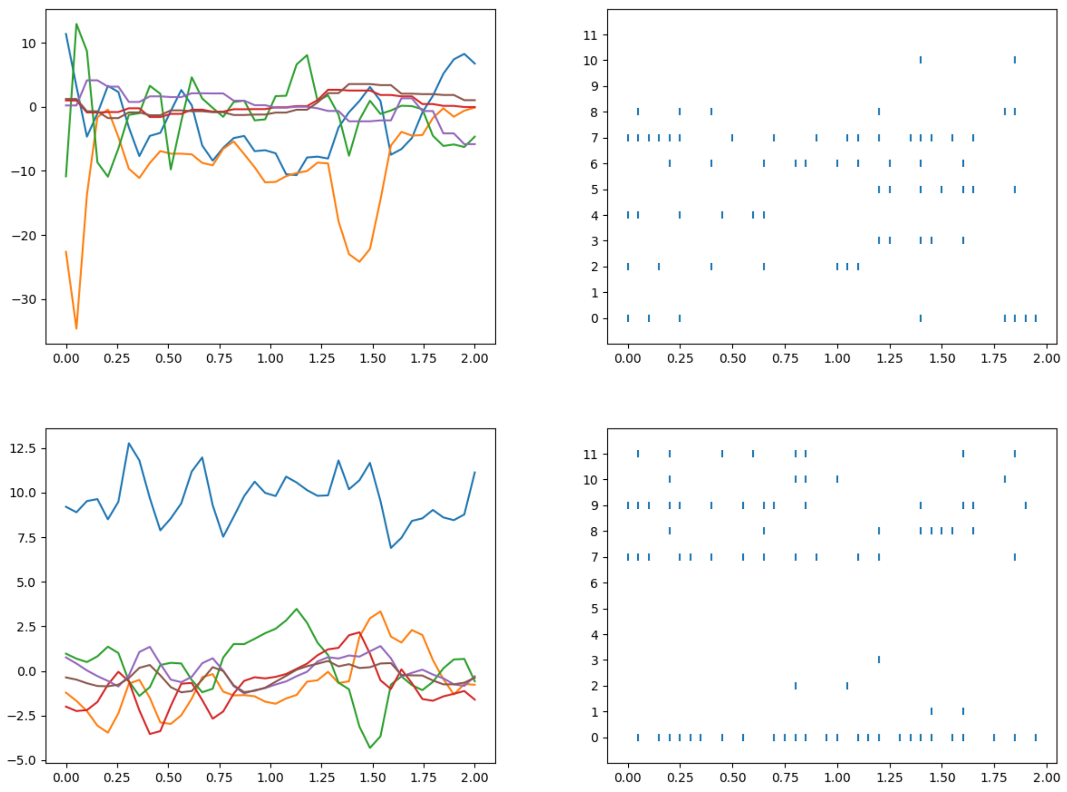


Figure 3.10: Two random phase encoded samples, belonging to the walking (above) and stairs (bottom) classes.

The PHASE encoding produced two channels per data dimension, associated with the increasing and decreasing data changes (UP and DOWN). Therefore, the resulting encoded dataset contains 12 dimensions, double the amount of the raw dataset.

Each sample of the encoded dataset is composed of two lists of values that describe the spiking events: each event is identified by its channel and the timestep at which it occurs.

Searching for and selecting the best encoding technique specifically for the target data, is a mandatory step for any spiking classification task. Relying on the extensive and comprehensive benchmark discussed in the research cited above should ensure optimal performance on the target task of this thesis work.

3.3 Training Pipeline

At the heart of the proposed pipeline, we can find a sub-pipeline devoted to training consistent and effective models, the training pipeline.

It was specifically thought to be reused in multiple points of the end-to-end pipeline and, therefore, was designed to be fully configurable. In fact, the training pipeline is the kernel of the Architecture Search, Hyperparameter Optimization, and Extended Training steps.
Given the importance of this component, a lot of attention and time was put into its development, optimization, and testing.

The pipeline exploits the Rockpool framework, which allows for efficient training of models, that are compatible for conversion and deployment to the target hardware.

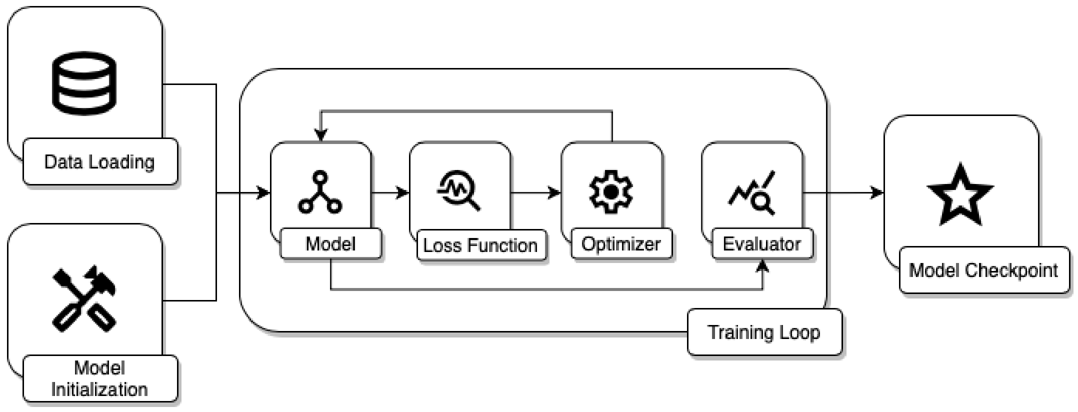


Figure 3.11: Overview of the training pipeline. A data loading step prepares the input data to be used for training. Then, a model initialization step builds the model that will be trained. The training loop, based on the backpropagation algorithm, optimizes the model parameter iteratively. The best model, evaluated on the validation set, is selected and a checkpoint is saved. The checkpoint contains all the information needed to restore the model afterwards, for inference.

While sharing the same high-level procedure, DYNAP-SE2 and Xylo use two different training pipelines that diverge slightly depending on the Rockpool backend used.
In fact, the DYNAP-SE2 toolkit is based on the JAX Rockpool backend, while Xylo uses Torch.

Specifically, the two pipelines diverge on the basic modules used to build the networks, the loss function used, and platform-specific operations such as mismatch generation and Synapse Pruning, which are only available for DYNAP-SE2 and Xylo, respectively.

The process starts with the loading of the input data. The data loading step takes the spike-encoded dataset as input and works as follows:

1. Each input sample is converted to be compatible for processing with Rockpool. In particular, samples are converted to their rasterized form, where each data sample represents a specific timestep. Events are represented in a boolean matrix, where the first axis corresponds to the timestep and the second axis to the channel.

2. Training, validation, and test subsets are extracted by shuffling and splitting the original data following a fixed ratio (60 % training, 20 % validation, 20 % test).

3. Datasets are batched and formatted by using data handling utilities from Tonic, and data loading techniques supported by Torch.

The next step of the training pipeline involves building a trainable model from a specific network architecture.
Rockpool makes this step straightforward by offering a set of basic modules that can be used to compose any common network architecture. Among them, we can highlight the *LIF* module, which represents a neuron population that implements the LIF neuron dynamics, and the *Linear* module, which holds a linear weights matrix and must be used to form multi-layer architectures.
The two most useful structural modules are *Sequential*, which allows to stack of basic computational layers sequentially, and *Residual*, which adds skip connections between layers and can be used to build residual networks.

In Rockpool, each network must begin with a *Linear* layer, which contains the weights for the input connections: to feed input data into the chip, both DYNAP-SE2 and Xylo use a set of virtual input neurons; these weights are associated with the connections between the virtual input neurons and the physical neurons on the chip.
In general, for classification tasks, the last layer of the network should consist of a population of neurons (e.g., *LIF*), in which each neuron represents an output channel and is associated with a specific label. The firing rate of neurons can be analyzed to obtain the prediction.

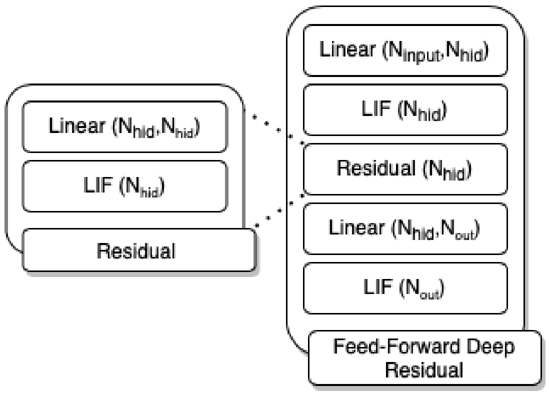


Figure 3.12: Example of a residual network compatible with Rockpool.

This step, which involves building the model to be trained, marks the first point of divergence between the DYNAP-SE2 and Xylo pipelines.

DYNAP-SE2 support is based on the *DynapSim* module: a simulator developed using JAX that can simulate the behavior of a DYNAP-SE2 board. The *DynapSim* layers can be stacked, interposing a linear weights layer between each pair, by using the *LinearJax* module, to form complex networks.
For computational feasibility purposes, the simulator is not numerically precise with respect to the hardware, but it uses approximations to be fast enough to be used for training models.
While the device and the simulator do not react exactly the same to the same input, this also happens between two physical chips because of the device mismatch

problem. For that reason, the imprecise behavior of the simulator can be positively considered as a way to add robustness during model training, which should produce consistent cross-chip performance after deployment.
DynapSim also provides a mismatch generation feature that can be used for the same purpose, that is to add robustness to the trained models, by varying mid-training the model parameter randomly within a specified interval of values.

Differently, Xylo models can be built by using the standard *LIFTorch* and *LinearTorch* layers offered by the Torch backend in Rockpool.

Finally, after the data loading and model building steps, we can find the actual training loop, that is responsible for the iterative training of the model.

The training loop is based on the backpropagation algorithm, which has been adapted by Rockpool developers for use with the spiking paradigm, by using a surrogate function [32]. Both DYNAP-SE2 and Xylo training pipelines are based on this technique.
Rockpools fully takes care of this process, automatically substituting the activation function in the forward and backward passes during backpropagation: the forward pass uses the non-differentiable spiking activation function, while a differentiable approximation, the surrogate function, is used for the backward pass.

The training loop works as follows:

1. For each epoch, the model is fed all the batches of data one by one, processes them, and generates output activity. For a single sample, the model output activity can be visualized as a boolean 2D matrix having the model's output channels on one axis and the timesteps on the other. The value one means that a spiking event occurred, while zero means no activity.

2. The loss function is used to calculate the batch loss, starting from the output activity and ground truth for each of the batch's samples.
   The DYNAP-SE2 and Xylo pipelines' behavior also diverges in this step. Xylo uses the cross-entropy loss function calculated between the output activity,

after applying softmax, and the expected one-hot formatted label. DYNAP-SE2 employs mean squared error loss, still calculated on the output activity, but comparing it to an artificial target signal in which the output neuron associated with the correct label fires at every timestep.

3. Next, the gradients are calculated and fed to the optimizer, which updates the parameters of the network.
   Both DYNAP-SE2 and Xylo make use of the Adam optimizer.

4. Finally, after each epoch, the model is evaluated using the validation set and compared, in terms of accuracy, with the best score: if the model obtains a better score than the previous one, the pipeline saves a checkpoint of the model, containing all its metadata and parameters that can be used to restore it later.

3.3.1 Synapse Pruning

As outlined in the previous chapter, Xylo imposes a strict constraint on the fan-out of its neurons, set at 64 for the Xylo Audio model, our designated target device.
By default, each single layer adopts an all-to-all connections policy, connecting every single neuron in a layer to all the neurons in the next layer. While this policy facilitates a reduction in complexity during the model deployment process, especially in mapping the model onto the hardware architecture, it simultaneously imposes significant constraints on the ability to fully leverage Xylo's resources.
In fact, the fan-out constraint together with the all-to-all policy limits the number of neurons per layer to a maximum of 64.

Xylo IMU is not affected by the fan-out limitation, offering up to 512 output connections per neuron. However, some recurrent networks might have issues with the maximum number of employable weights of 31744.

This thesis proposes Synapse Pruning, a technique that aims to alleviate these constraints by adapting incompatible networks and thus making better use of chip

resources.
Synapse Pruning is a two-stage technique that is applied both at training time and during the hardware deployment phase.

The first phase occurs during training: periodically, after a configurable number of epochs, defaulting to 10, the training loop triggers a routine that is tasked with adapting the model to be compatible with the device. This is achieved by dropping, for each neuron in every layer of the network, a precisely calculated amount of connections by setting their weights to zero.

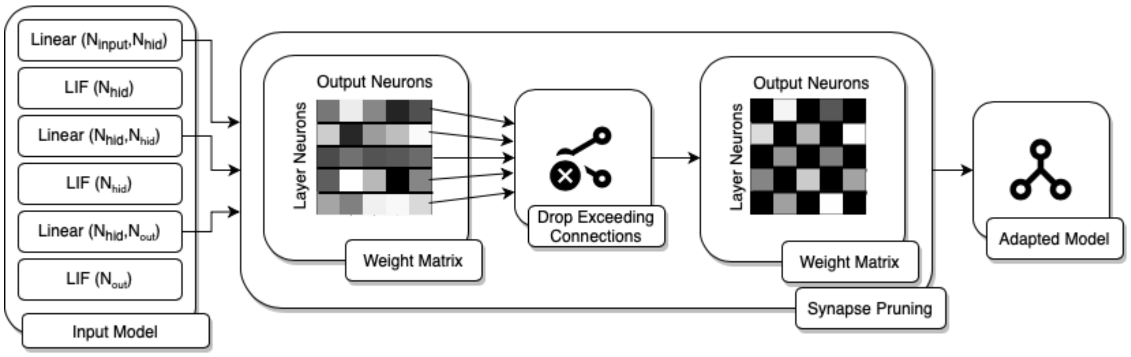


Figure 3.13: Overview of the proposed Synapse Pruning technique. A model employing an incompatible network architecture is adapted by dropping the less significant output connections for each neuron in every layer of the model.

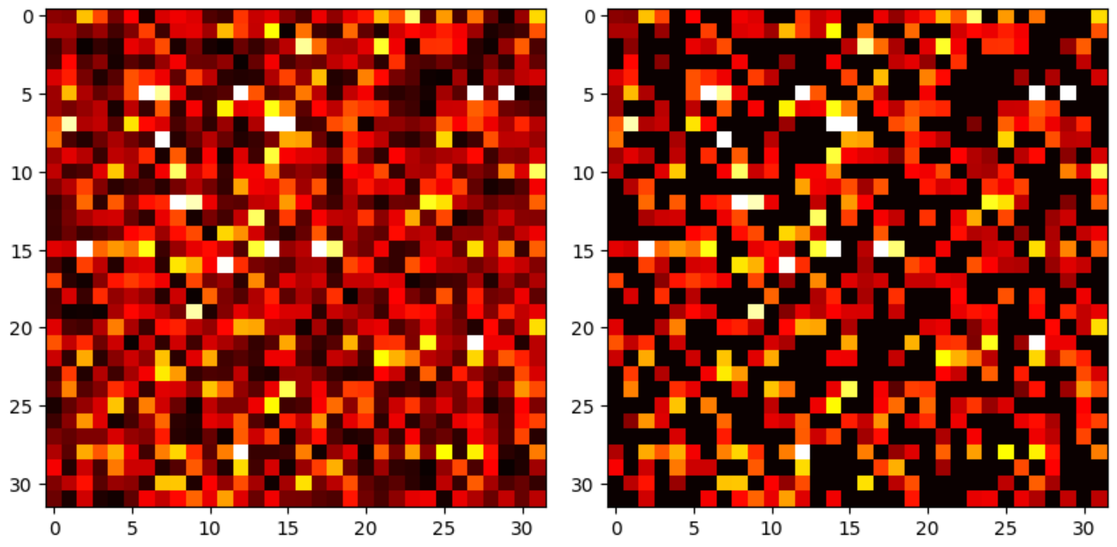


Figure 3.14: Comparison of the weights matrix of a Feed-Forward Deep architecture with N_{hid} of 32. Synapse Pruning was applied to fit a hypothetical neuron fan-out of 16.

The amount of connections to be dropped is calculated taking into account both simple and recurring connections, to make various families of networks compatible with the hardware requirements. Therefore, Synapse Pruning works for most of the architectures supported by Rockpool, including residual and recurrent networks.

The connections to eliminate are selected as the less significant ones, that is the ones having the smallest weights. This should ensure minimal disturbance in the behavior of the model.

Performing Synapse Pruning periodically during training allows the network to adapt to work with fewer resources with little, if any, performance loss.

The second phase is performed during the deployment process, before the quantization of the model parameters: Synapse Pruning is applied again after the model has been reinitialized from the training checkpoint to ensure compatibility with the target platform for the deployment process.

3.3.2 Hyperparameter Optimization

This pipeline largely exploits the advantages offered by automated hyperparameter optimization. It is used as the foundation for most of the training and fine-tuning related steps, such as, for example, Architecture Search or Quantization Tuning.

In addition to the standalone mode, the training pipeline offers an HPO working mode, exploiting the NNI tool.

NNI, a tool for automated HPO, described in the section 2.3.4, requires some integration changes in the training pipeline code to be able to control it during the process. Principally, these changes allow the training pipeline to access the NNI context, for operations such as fetching the parameters configuration to use or reporting the model accuracy after the training ended.

The HPO process is executed through a startup script that generates a new NNI experiment, which runs in the background and offers a web-based GUI for checking the progress in real-time.
The experiment can be heavily configured, setting its maximum time duration or

number of trials, the search space for each hyperparameter, the level of concurrency to use, and more.

During the experiment, the training pipeline is executed various times, each time using a different parameter configuration. The parameter configuration to use for each trial is automatically selected by a configurable tuner, that exploits a specific hyperparameter discovery strategy.
To be able to run, the experiment must be configured with the sampling interval for each hyperparameter, called search space: the tuner will only sample values within the provided range. Carefully selecting the search space is the basis of an effective and efficient HPO process.

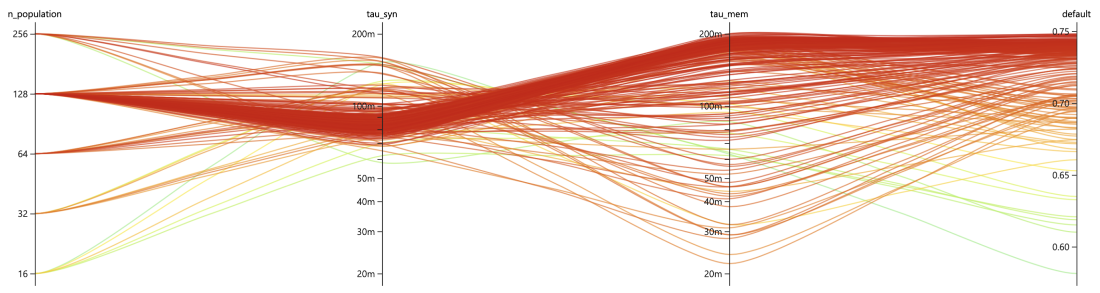


Figure 3.15: Visualization of hyperparameter configurations exploration during an HPO experiment, using the Anneal tuner.

NNI offers a wide range of tuners, having different working behaviors. This thesis employs the Anneal tuner for all the HPO experiments, which showed the best results during practical usage. The Anneal strategy starts from random sampling, but converges over time to hyperparameter configurations closer and closer to the best ones observed.

The training pipeline continuously reports the model loss and accuracy, evaluated on the validation set, through intermediate results. When the training finishes, the pipeline saves a checkpoint for the best model, and its accuracy is reported to NNI as the final result. These scores are visualized on the GUI, enabling the user to check the status of the process.

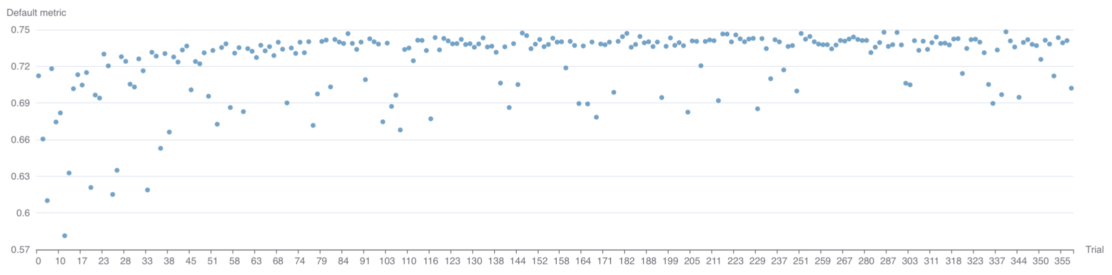


Figure 3.16: Visualization of trials' final accuracy scores during an HPO experiment.

In some cases, some hyperparameter configurations may produce a performance that is obviously insufficient, therefore it would be unnecessary to finish the evaluation.
For these situations, NNI offers early stopping techniques, using algorithms called assessors. The assessors monitor the intermediate results of each trial, stopping trials that are visibly under-performing, leaving space to test other more promising configurations.
Early stopping techniques are very useful when the available resources for the experiments are limited, as in the case of this thesis work. The proposed training pipeline implements the Medianstop assessor, which evaluates whether a configuration performs worse than the median and, if so, terminates the test.

At the end of the HPO process, the best-performing hyperparameter configuration is selected as the one having a higher accuracy score on the validation dataset.

3.4 Architecture Search

The Data Selection and Encoding steps led to the definition of a set of tasks for the benchmark. The next step of the pipeline, the Architecture Search phase, consists of discovering the best network architecture to train a model that could solve such tasks, aiming for the best accuracy possible.

This stage is based on the training pipeline described in the previous section and exploits the NNI tool for automated hyperparameter optimization.

The process employed in this work can be summarized as follows:

1. A set of promising network architectures is manually designed, taking into consideration the hardware and software limitations of the target devices.
2. The best set of hyperparameters, including structural and neuron-related settings, is then found by HPO. This step aims to extract the best performance possible, in terms of accuracy, while analyzing the behavior of the model at the variation of the structural parameters.
3. The best architecture is selected as the best performing between the predefined ones, in combination with the structural settings found.

As happened for the training pipeline, the architecture search process was affected by platform limitations, so DYNAP-SE2 and Xylo were treated differently. In fact, it turned out that it is impossible to construct a deployable multilayer network for DYNAP-SE2 because of a bug in Rockpool.
Specifically, the bug consisted of neurons contained in the output layer not being mapped into the hardware configuration, which is then loaded onto the chip. Therefore, there was no way to extract and record the output activity of the network.
The bug has been notified to SynSense through Github, opening an Issue `https://github.com/synsense/rockpool/issues/10`).

Another important limitation on the DYNAP-SE2 side derives from the hardware mismatch, mentioned in the second chapter and in the previous section: its effect exponentially increases with the depth of the networks, making it impossible to train a deep model that operates similarly between simulator and real devices.

For these reasons, I couldn't run any form of architecture search on the DYNAP-SE2 and was constrained to only use a single-layer network.
This is a severe limiting factor for the performance of the board, which, despite having one of the most advanced hardware architectures, suffers from poor software support.
In any case, this work should not be seen as a competition between the DYNAP-SE2 and Xylo neuromorphic platforms, but as an analysis of the individual capabilities of the currently available neuromorphic devices and the state of software support for these.

The network employed for every experiment in this thesis work on DYNAP-SE2 matches the standard network proposed by Rockpool, called *DynapSimNet*, composed of a *Linear* layer and a *DynapSim* layer.
The linear layer contains the weights of the connection between the virtual input neurons and the physical neurons. Specifically, the number of input neurons matches the data channels, that is 12.
Being the output layer, the *DynapSim* layer contains a number of neurons equal to the number of classes of the task we are considering.

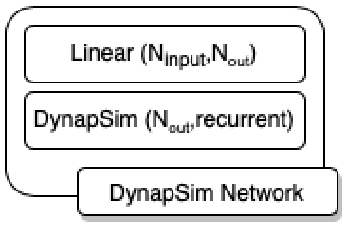

Linear (N_{input},N_{out})
DynapSim (N_{out},recurrent)
DynapSim Network

Figure 3.17: Overview of the DynapSimNet, a fixed architecture used for DYNAP-SE2. Only a single layer is supported due to software issues.

On the other side, Xylo allows for much more freedom in terms of compatible networks.
The architecture search for Xylo followed the original plan, described above, therefore is a sort of semi-automated process. In fact, the shape of the network is selected

between a set of six manually designed, predefined architectures. The proposed architectures include feed-forward, recurrent and residual networks, exploring their shallower and deeper versions.
However the number of hidden units (N_{hid}), and therefore the size of the network, was treated as a hyperparameter and automatically optimized together with the neuron's time constants (*tau_mem* and *tau_syn*).

Figure 3.18 describes the structure of the six network architectures tested.

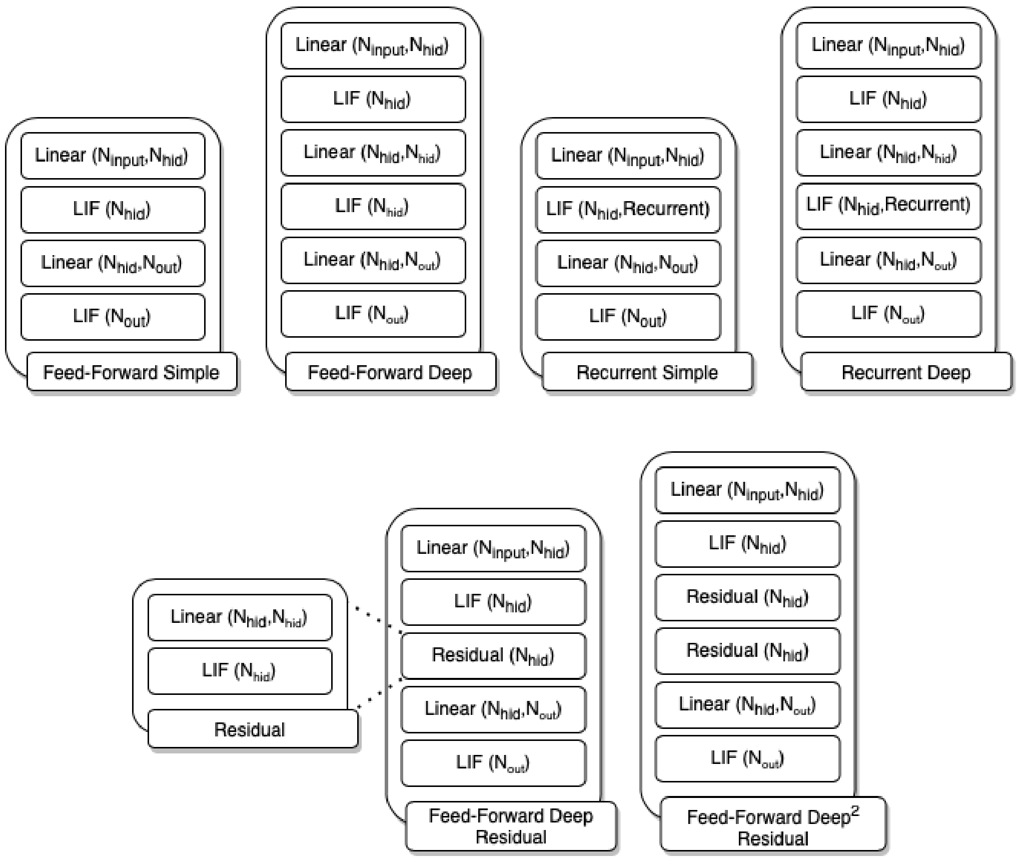


Figure 3.18: Overview of the architectures compared during Xylo's architecture search. The number of hidden neurons is variable between 16 and 256.

Task 7CB, based on the seven most separable classes in the WISDM dataset, was selected for the architecture search process.
Being a very difficult task, it should be more sensitive to performance variations and hopefully expose the different potentials of the various architectures tested,

better than the other tasks can.

The six experiments for the Architecture Search stage had a fixed duration of three days and followed the same HPO procedure described in the last section: each trial involved training a model on a specific hyperparameters configuration for 250 epochs, evaluating the accuracy of the model on the validation set and saving the checkpoint for the best one.
The duration of the experiments was generally enough to reach an optimized score, delineated by a performance plateau, allowing for a fair and objective comparison between the proposed architectures.

3.5 Hyperparameter Optimization & Extended Training

After the Architecture Search step, which selected the best neural network architecture for the job, the focus shifted to training the most accurate models achievable for each task.

This process is identical between the DYNAP-SE2 and Xylo platforms, and consists of two sequential jobs repeated for each of the benchmark tasks:

1. The first job consists of performing Hyperparameter Optimization (HPO), searching for the best configuration of parameters possible for each specific task.

2. Finally, a model is trained for a larger number of epochs, using the optimal configuration found in the previous phase.

While the Architecture Search stage specifically targeted the 7CB task, this process addresses all the tasks, selecting the best configuration of hyperparameters without having to deal with the additional complexity generated by architectural parameters.
In this phase, HPO focuses precisely on the neuron's time constants, trying to obtain the most suitable configuration for each task. In particular, the process optimizes the *tau_mem* and *tau_syn*, controlling the leakage behavior of the neuron's membrane and the synapses.

The HPO procedure adopts the default technique with the same duration settings used for architecture search: each experiment lasts for 3 days with trials of 250 epochs.
The number of epochs, selected for computational and time constraints, proved again to be sufficient for the optimization job. However, analyzing the behavior of the loss curve of values, it appeared that the resulting models still had some space for improvement.

This issue is addressed by a final Extended Training step: a model, dedicated to each task, is extensively trained for a very large number of epochs, to extract the best performance possible.
In detail, each final model was trained for 1500 epochs, configured with the optimal parameters obtained during the HPO step.

Specifically for the DYNAP-SE2 case, mismatch generation, which was not used for the HPO phase, was enabled for the final extended training, seeking to improve the robustness of the final model to the hardware mismatch phenomenon. Mismatch generation was applied every 100 epochs.

3.6 Hardware Deployment

The final stages of the pipeline involve the fine-tuning and deployment of the trained models on the target devices. The hardware deployment process, seen from a high-level perspective, is identical between the DYNAP-SE2 and Xylo platforms.

Rockpool offers a model conversion procedure for both DYNAP-SE2 and Xylo.

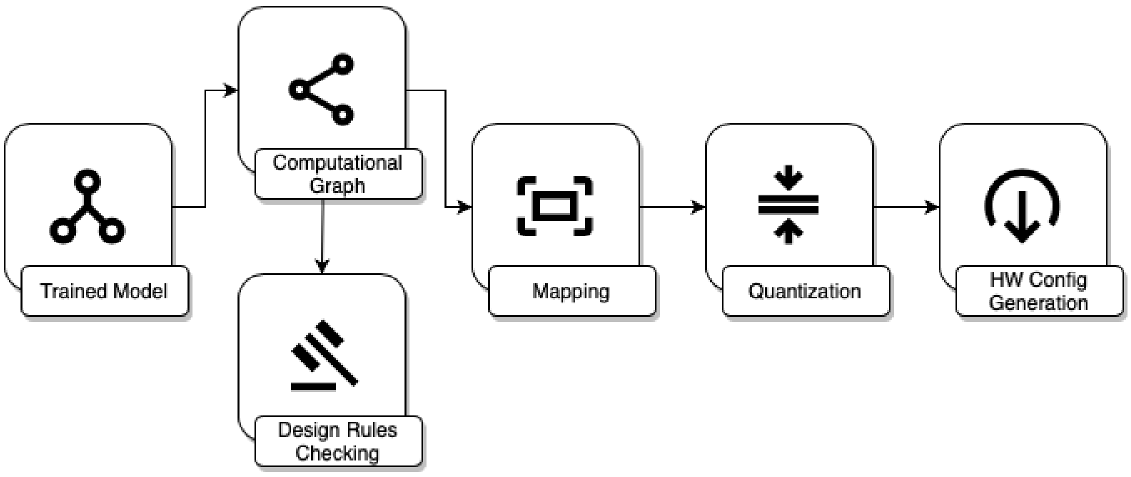


Figure 3.19: Overview of the Rockpool's hardware configuration generation procedure. The trained model is converted into a computational graph, that is then tested against DRC rules. A mapping function fits the computational graph into the hardware architecture. Later, during the quantization process, model parameters are converted to a format compatible with the target device. Finally, a deployable hardware configuration is generated.

It starts with the extraction of a computational graph from the model. The latter is then tested against a predefined set of design rules, different for each platform, which verify the compliance with the constraints imposed by the device architecture. This process is called design rules checking (DRC).

Next, a mapping tool tries to fit the computational graph to the device's hardware architecture, allocating resources and assigning hardware IDs to entities like neurons, weights, input, and outputs. The result of this mapping procedure is a hardware specification object, describing the model converted for target hardware.

The specification object contains all the parameters of the original model, extracted from the computational graph. However, these parameters are likely incompatible with the hardware requirements of the target devices: in fact, given the need to keep the power consumption and design complexity very low, devices might support only specific representations for parameter values.

For instance, DYNAP-SE2 employs a parameter representation based on two integers, *COARSE* and *FINE*. Changing the 3-bit *COARSE* value of a specific parameter has a very impactful effect on the device behavior, on the other hand modifying the 8-bit *FINE* value causes small and controlled variations.
Moreover, the device does not allow setting a dedicated weight for each synaptic connection: DYNAP-SE2 only allows to set four base weight values globally for each core; it is then possible to select one or a sum of the weight values for each connection by using a 4-bit boolean bit-mask.

Xylo offers much more freedom on that side, allowing to set a different weight for each connection. Anyway, it only supports 8-bit integer values for the model parameters, instead of the floating point values used during training.

The quantization process deals with these issues by converting the model parameter values to be compatible with the format and parameter range allowed by the target device.

Finally, the quantized specification object, containing the converted model structure and compatible parameter values, is processed to generate a hardware configuration object. The output of this process is an object compatible with Samna, the toolkit used for managing hardware devices, and can be directly deployed on the target device.

Going back to the pipeline proposed in this thesis, the Hardware Deployment section endorses the Rockpool procedure described above and includes two main processes:

1. Quantization Tuning: A novel technique to deal with the losses of parameters quantization. In this phase, the parameters of the extensively trained models are manipulated to reduce the quantization losses.

2. Hardware Configuration Generation: The step responsible for the generation of the deployable hardware configuration.

Figure 3.20 shows a schematic description of the Hardware Deployment section of the pipeline.

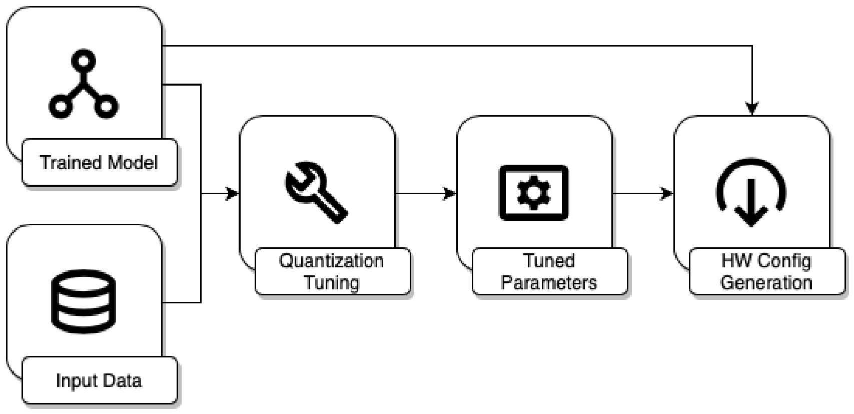


Figure 3.20: Overview of the Hardware Deployment macro-area of the proposed end-to-end pipeline.

3.6.1 Quantization Tuning

Quantization refers to the process of converting parameters, belonging to a pre-trained model, to ensure compatibility with the constraints of the target device.
It is a lossy process that can sometimes be detrimental to network performance. In particular, on some occasions network activity may be too low or too high, thus producing no activity or saturating hardware capacity. Either possibility can cause a loss of performance or completely impede inference.

Rockpool already offers quantization techniques that often produce good results, but sometimes it suffers from the problem stated above.

While experimenting with the deployment of trained models, I noticed that quantization disruptions can be minimized by manipulating the quantized parameters (in particular, the weights), tweaking them globally, or altering the values associated with individual neurons in the readout layer.

For this reason, I propose the Quantization Tuning technique, aiming to rebalance the neural activity of the quantized model, thus reducing or eliminating the side effects of quantization.

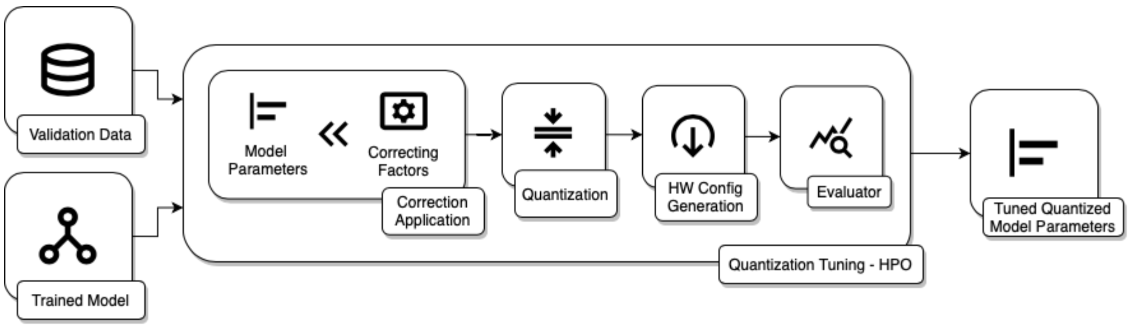


Figure 3.21: Overview of the Quantization Tuning process. Based on automated HPO, it aims to find the best correcting factors configuration to re-balance the network activity after parameter quantization.

Figure 3.21 describes the proposed Quantization Tuning procedure.

The tuning process is controlled by two types of parameters:

- *global_activity_factor*: A value that is multiplied by all the weights and controls globally the network activity magnitude.

- *c*_activity_factor*: A set of values (one for each class) that control the balance of the output activity for a specific class.

Quantization Tuning involves an HPO process to find the optimal values of these parameters for which the quantized model performs the best.
The HPO process starts from the quantized form of the model, tests a large set of configurations for the tuning parameters, and evaluates the performance on the validation set.
In particular, the system tests 1000 configurations of parameters, using the same process employed for the Architecture Search and HPO steps of the pipeline, described in the section 3.3.2.

Tuned quantized parameters will be applied during the Hardware Configuration Generation step, following the modality described in the next section.

Applying Quantization Tuning should result in well-tuned post-quantization network activity, which positively affects the deployed model performance, allowing to recover some of the loss caused by the quantization process.

3.6.2 Hardware Configuration Generation

The final step of the pipeline consists of generating the deployable hardware configuration.

This process follows the same procedure proposed in Rockpool, previously described in this section, with small changes due to the introduction of Quantization Tuning.
To summarize, it works as the following:

1. The model to be deployed is restored from the checkpoint. The proposed algorithm can automatically reconstruct a working DYNAP-SE2 and Xylo model from its checkpoint files, without the need for any additional input from the user.

2. The equivalent computational graph is extracted from the model and is mapped to the hardware resources as in the original Rockpool process, generating a specification object.

3. If Quantization Tuning was performed for the model intended for deployment, the system directly updates the specification object with the tuned quantized parameters, eliminating the need for additional quantization.
   Otherwise, the parameters in the specification object are quantized without applying any tuning.

4. The quantized specification object is converted into a hardware configuration object, that is directly deployable to the target hardware using Samna.

3.7 Source Code

The source code for the proposed pipeline can be found in a dedicated, open-source repository on GitHub.
GitHub is a free cloud platform that offers versioning control features based on Git. Also, it provides project management functionalities such as access control, bug tracking, task planning, and continuous integration.

This thesis's source code can be found at the following URL:
`https://github.com/MarcoBramini/master_thesis_ce_polito`.

The repository contains the scripts implementing the various processes described in this chapter, plus the checkpoints of the optimized models together with metadata including performance scores.

In particular, the *task-definition* folder contains all the scripts needed for the Data Selection and Task Definition steps, including the separability metrics, KDE calculation, class combinations generation, and some utility scripts for data visualization.

The *training-pipeline* folder holds all the scripts needed for the Architecture Search, HPO, Extended Training, and Quantization Tuning steps.

The *dynapse2-deploy* and *xylo-deploy* folders include the Python notebooks that contain scripts for the evaluation, simulation, and deployment of the models. These notebooks implement the Hardware Configuration Generation step of the pipeline.

Chapter 4

Results and discussion

This chapter presents the results obtained by evaluating the proposed pipeline and assessing its capacity to produce effective models.

The evaluation process will start by analyzing the outcomes of the architecture search, then proceeding to the deeper training stages, and finally investigating the models' behavior using the target devices' simulator.
Special emphasis will be placed on the newly proposed Synapse Pruning and Quantization Tuning techniques.
Each step's intermediate results will be discussed to offer insights into the process's efficacy and the impact of the implemented techniques.

The intention of this analysis is not to compare the devices under testing, given the high inequality in terms of software support, but to produce a benchmark on a generally plausible classification task, that could directly provide useful insights about the capabilities of readily available standalone neuromorphic devices.

4.1 Architecture Search

The evaluation process starts with the Architecture Search step, responsible for selecting the best architecture for the task.

This process directly targets the 7CB and is based on the analysis of different, manually selected architectures. HPO is employed to retrieve the best parameters for each architecture, including the size of the hidden population and the neuron parameters.

Unfortunately, due to software issues affecting the DYNAP-SE2, already described in section 3.4, it was impossible to perform this step for the DYNAP platform. In fact, due to a bug, only single-layer networks can be used without running into training and deployment issues when developing DYNAP-SE2 models in Rockpool.

Table 4.1 shows the results of the Architecture Search process on Xylo, evaluated on the validation and test sets associated with task 7CB.

Architecture	Best N_{hid}	Validation	Test
Feed-Forward Simple	128	73.4%	72.3%
Feed-Forward Deep	256	76.7%	75.7%
Feed-Forward Deep Residual	256	77.6%	76.5%
Feed-Forward Deep^2 Residual	256	**78.6%**	**77.5%**
Recurrent Simple	256	76.3%	74.9%
Recurrent Deep	256	77.3%	76.2%

Table 4.1: Results of the Architecture Search process on Xylo. The performance is measured in terms of accuracy. All the experiments were conducted on the task 7CB. N_{hid} is a structural parameter, optimized during the HPO process, identifying the number of neurons in each of the network's hidden layers. The Feed-Forward Deep^2 Residual architecture, employing 256 hidden neurons per layer, emerges as the best one.

From the results, we can extrapolate an interesting trend: the model accuracy grows linearly with respect to the deepness of the network. The 7CB task is complex and the usage of a deep network during classification is highly beneficial.

Another interesting aspect can be highlighted when comparing residual and standard feed-forward networks: the residual variants seem to perform better than the standard models, as can be seen by observing the Feed-Forward Deep and Feed-Forward Deep Residual.

The Feed-Forward Deep^2 Residual structure turned out to be the best one, followed by its shallower version, the Feed-Forward Deep Residual.
This architecture is able to generate decent performance on one of the hardest tasks.

Unfortunately, the winning network uses most of Xylo's resources, requiring a total of 768 hidden neurons, making it impossible to extend the search to even deeper architectures.

The need for a deeper network represents a big issue for the DYNAP-SE2 platform, constrained by the software issues cited above. Almost certainly, its performance will be heavily affected by the inability to use deeper networks, and consequently, a larger slice of its computational resources.

To conclude, these results look promising, given that this is still the very beginning of the training process.

4.1.1 Synapse Pruning

We now analyze the effects of the proposed Synapse Pruning technique, by comparing the performance of the tested architectures as their size changes in terms of neuron population.

Figure 4.1 shows the performance comparison between the tested architectures, at the variation in size of the hidden neuron population. Again, the performance was analyzed in terms of accuracy, on the validation set relative to the 7CB task.

As clearly visible, Synapse Pruning proved beneficial in almost every case: exploiting the larger variants of the proposed architectures resulted in a general improvement in the accuracy of the models.

Feed-forward networks seem to benefit the most from Synapse Pruning. We can see a marked trend of improvement throughout the experiment.

This is not the case for recurrent networks, which seem not to digest the configuration with the largest number of neurons.

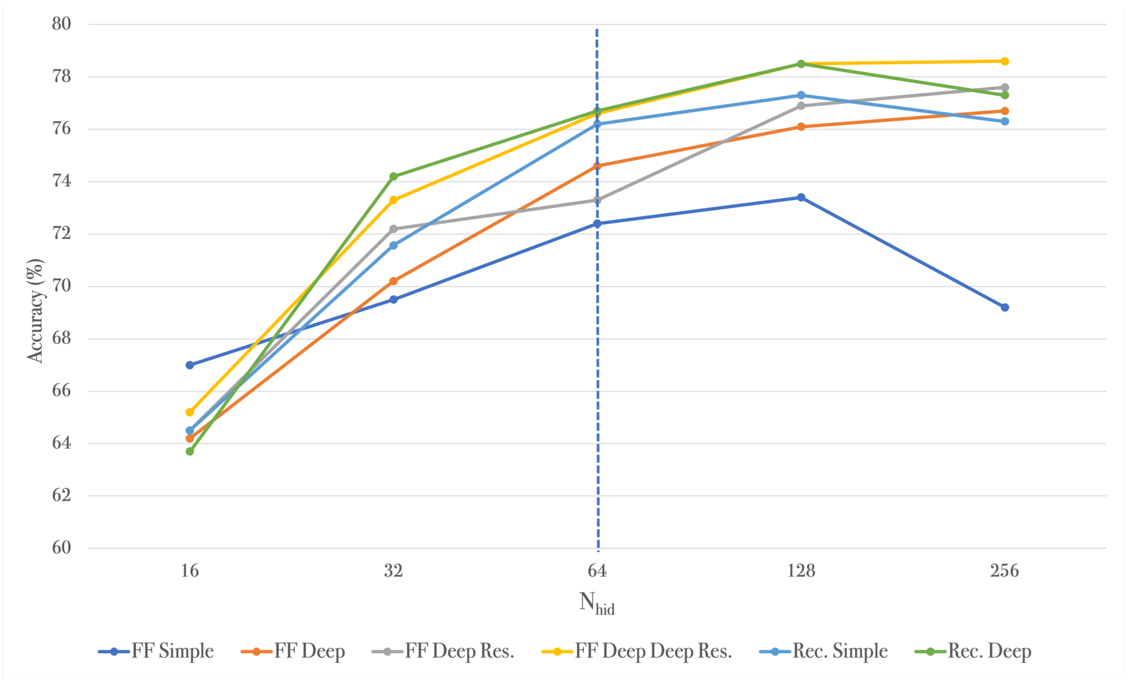


Figure 4.1: Visual comparison of the accuracy scores generated by evaluating various architecture structures during architecture search. N_{hid} identifies the number of neurons in the hidden population. The scores at the right of the dashed lines have been produced employing the Synapse Pruning technique.

Anyway, this was somewhat expected: in the proposed recurrent structures, each layer is not only connected with the previous and next ones, following the all-to-all strategy, but also with itself. Applying Synapse Pruning on such networks causes the removal of a huge number of connections, possibly causing problems in the model behavior.
Therefore, recurrent networks are almost certainly affected by this issue, which is a strong limitation in the scalability of such architectures.

Moreover, we can also identify an anomaly within the feed-forward family of architectures: the Feed-Forward Simple network seems to work very badly with 256 neurons, producing an even worse accuracy than the one obtained with just 32 neurons.
This phenomenon can probably be explained by looking at the layout of the architecture: being the smallest network, the Feed-Forward Simple architecture consists

of only one layer and is less interconnected; removing connections in this structure affects network activity more heavily than in the other structures.

Table 4.2 explores in detail the accuracy scores obtained by the two best architectures.

	FF Deep^2 Res.		Rec. Deep	
N_{hid}	Val.	Test	Val.	Test
No Synapse Pruning				
16	65.2%	63.2%	63.7%	62.2%
32	73.3%	71.8%	74.2%	72.7%
64	76.6%	75.4%	76.7%	75.2%
Synapse Pruning				
128	78.5%	77.1%	78.5%	76.9%
256	**78.6%**	**77.5%**	77.3%	76.2%

Table 4.2: Analysis of Synapse Pruning effects on the two best architectures at the variation of the number of neurons in the hidden population (N_{hid}). The performance is measured in terms of accuracy.

We can notice how the performance of both of the architectures is very similar: with 128 neurons they produce almost the same score.
However, as seen previously, it is clear that the Recurrent Deep network activity cannot sustain the disruption caused by the removal of too many connections.

On the contrary, the Feed-Forward Deep^2 Residual architecture can better adapt to the larger amount of neurons, although only producing a very small improvement in accuracy.

Generally, Synapse Pruning enabled the usage of larger and more complex networks, producing advantages in terms of performance and allowing better use of Xylo's onboard resources.

4.2 Hyperparameter Optimization

We continue the discussion by evaluating the two final stages of the Model Generation macro-area of the proposed pipeline: the Hyperparameter Optimization and Extended Training steps.

After the selection of the optimal architecture, the HPO process consists of specifically optimizing the neuron parameters for each of the tasks. The output of this process is a set of models, one for each task, that is specifically optimized for that task.

Starting from the DYNAP-SE2 platform, figure 4.2 shows the large number of configurations tested during the optimization process, while table 4.3 describes the results of the HPO process.

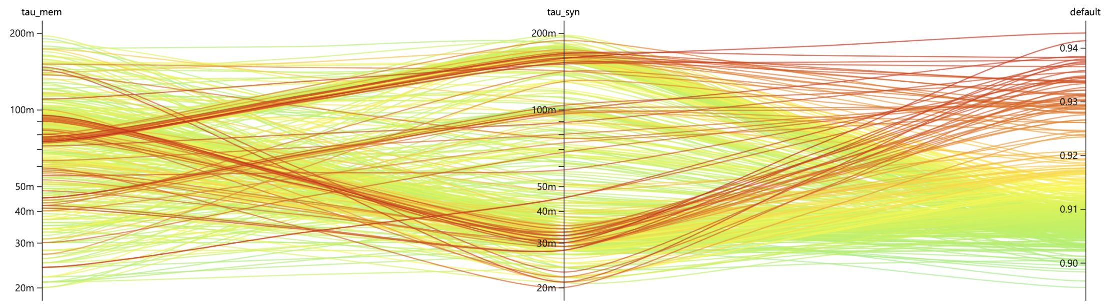


Figure 4.2: Visualization of the hyperparameter configurations tested by NNI during the HPO process on the 2CB task, for the DYNAP-SE2 platform.

Task	Validation	Test
2CB	94.3%	93.5%
2CW	61.0%	60.0%
3CB	79.3%	78.7%
3CW	44.7%	43.7%
4CB	62.0%	61.4%
4CW	36.6%	35.8%

Table 4.3: Results of the HPO process on DYNAP-SE2. The performance is measured in terms of accuracy.

As expected the performance on the DYNAP-SE2 platform is heavily affected by the software limitations cited before.

The single-layer network can generate acceptable accuracy when applied to easier tasks, such as 2CB and 3CB, associated with the most separable combination of two and three classes.
However, the platform experiences a dramatic drop in scores with the increase of task complexity: tasks 4CB and all the tested worst separable combinations travel near or below the 60 % accuracy threshold.

Furthermore, the DYNAP-SE2 training pipeline could not produce a working model for the tasks with the highest number of classes, 7CB and 7CW. During training, the loss value for the models associated with these tasks was stuck at a very high value, without any learning happening in the network.
Unfortunately, I was not able to find a solution to this problem.

Proceeding, figure 4.3 shows the behavior of the optimization process on Xylo.

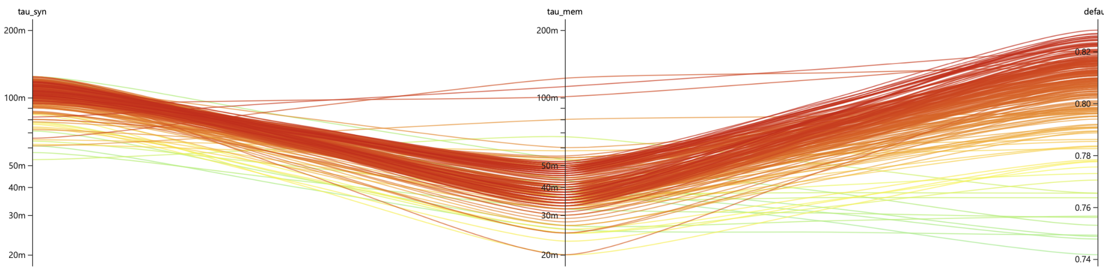


Figure 4.3: Visualization of the hyperparameter configurations tested by NNI during the HPO process on the 2CW task, on Xylo.

We can notice that Xylo is far less sensible to the variation of parameters if compared to DYNAP-SE2.
Presumably, this characteristic can be associated with the different network structures employed on the two devices: the depth of the network employed on Xylo, and its ability to map more complex patterns, improves tolerance to a wider range of parameter values, while the shallow network of the DYNAP-SE2 needs very specific values to perform well.

Table 4.4 shows the results for the HPO process on Xylo.

Task	Validation	Test
2CB	99.2%	99.1%
2CW	81.5%	81.5%
3CB	91.2%	91.4%
3CW	69.5%	70.2%
4CB	94.0%	93.1%
4CW	56.3%	55.2%
7CB	78.1%	77.3%
7CW	58.3%	58.6%

Table 4.4: Results of the HPO process on Xylo. The performance is measured in terms of accuracy.

The optimized models generate an accuracy of over 90 % on the simpler tasks, that is the 2CB, 3CB, and 4CB tasks.
However, they suffer on the harder tasks with the only exception of task 2CW.

One notable result is the one obtained for the 7CB task, where Xylo almost reaches 80 % accuracy value.

The results mostly satisfy the prior expectations, with the trained models doing well on easier tasks and struggling on the ones deemed as more complex, providing a validation of the proposed data selection method. The latter is generally able to correctly generate tasks of increasing complexity.
The DYNAP-SE2's models performance perfectly reflects this, with a steady and linear decrement in accuracy that follows the increase in task complexity.
This is also valid in the Xylo case but with two exceptions, that can be identified by comparing the accuracy of the tasks 3CB/4CB and 4CW/7CW: in these cases, the trained model performed better on tasks predicted as more difficult.

To conclude, the HPO process was able to maximize the performance in every situation, adapting the network behavior to the requirements of each task.

4.3 Extended Training

The next step, the Extended Training, is the last phase of the pipeline's macro-section relative to model generation.
Its objective is to extract the best performance possible from each model by training extensively, with the best architecture and parameters detected by the previous steps.

Starting again from the DYNAP-SE2, we want to analyze the behavior of the models during the Extended Training step.
This step introduces mismatch generation, as explained in the section 3.5, to enhance the robustness against the hardware mismatch phenomenon.

Figure 4.4, shows the loss and accuracy trends for a randomly selected model, as an indication of the goodness of the process.

The training process seems to work nominally. It can be seen that the loss did not stop decreasing, which might indicate that the model could improve further with longer training. However, this was not possible due to time and computational constraints.

Table 4.5 contains the scores obtained by each model during the Extended Training step.

Task	HPO	Validation	Test
2CB	94.3%	94.1%	93.2%
2CW	61.0%	58.0%	57.3%
3CB	79.3%	79.4%	79.2%
3CW	44.7%	45.4%	44.4%
4CB	62.0%	62.9%	62.7%
4CW	36.6%	36.4%	35.6%

Table 4.5: Results of the Extended Training process on DYNAP-SE2. The performance is measured in terms of accuracy.

Analyzing these results, we can notice that the DYNAP-SE2 process shows an unexpected behavior: performance levels for almost all tasks appear to be stationary when compared with the HPO results, not following the expected growth

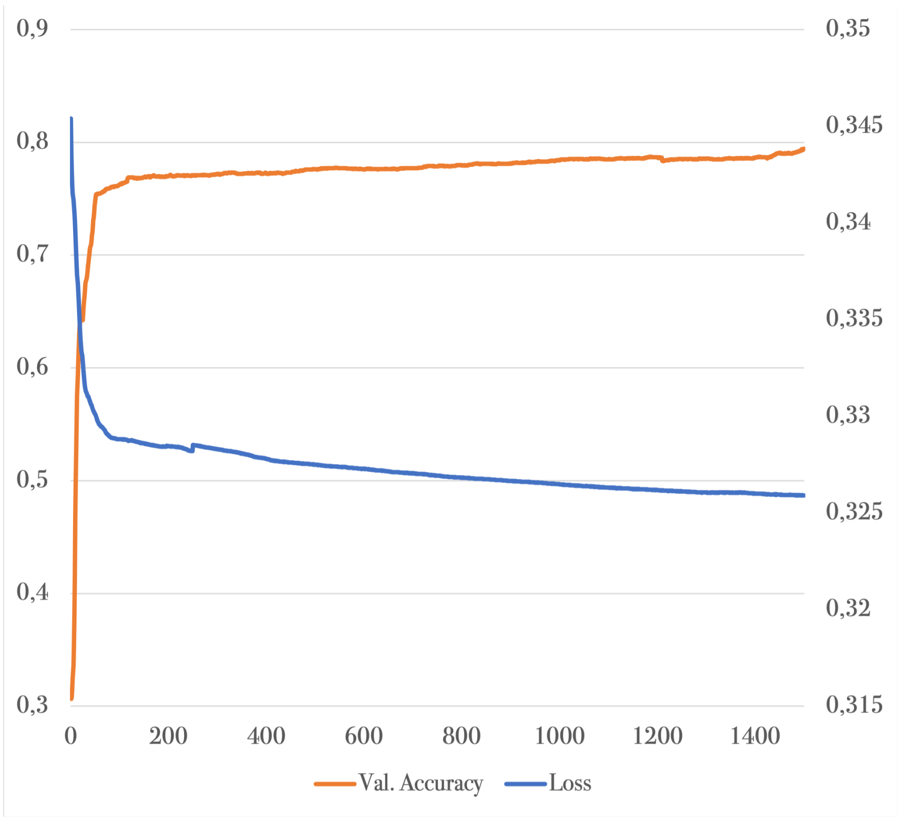


Figure 4.4: Behavioral analysis of the model under training for the task 3CB on DYNAP-SE2, during the Extended Training step.

generated by increased training time.
The cause of this phenomenon can be identified in the introduction of mismatch generation. This can be regarded as a price to pay for a gain in performance robustness.

We conclude this section with the analysis of Extended Training outcomes for Xylo. Following the same process employed for the DYNAP-SE2, we start by visualizing the behavior of a random model during the training process.
Figure 4.5 presents the Extended Training statistics for task 3CB.

The line plot shows a considerable amount of variability in the loss and accuracy values, more than what has been observed during the DYNAP-SE2 process.
The cause of this behavior can be identified in Synapse Pruning: triggered every 10 epochs, this technique is responsible for generating some disturbance in the training

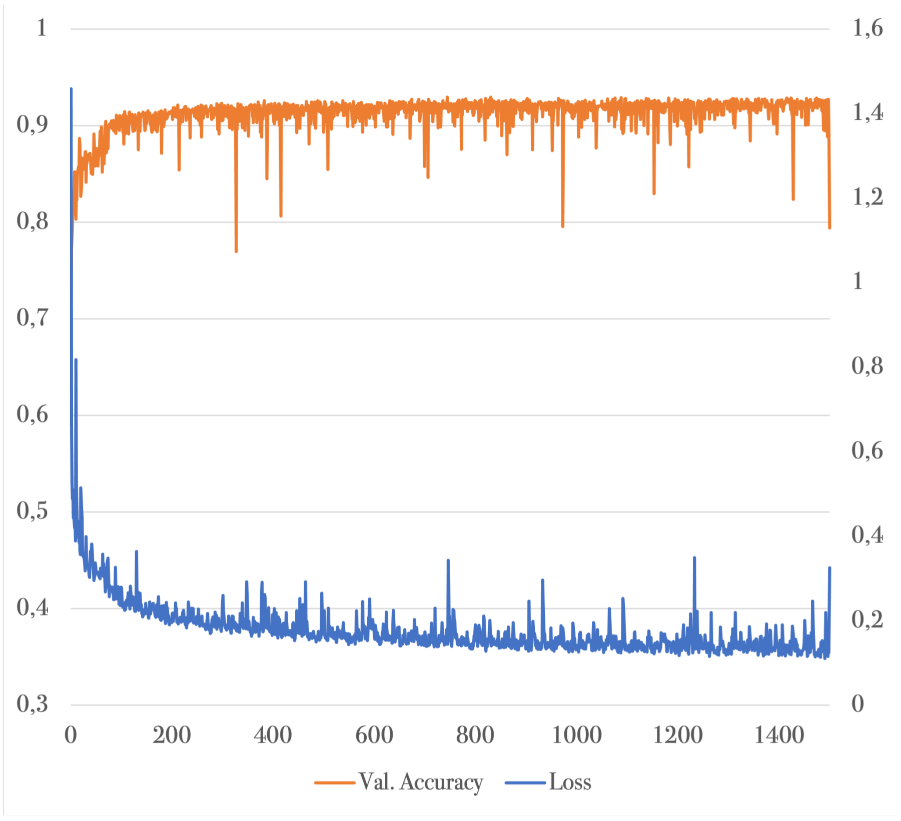


Figure 4.5: Behavioral analysis of the model under training for the task 3CB on Xylo, during the Extended Training step.

process.
Still, synapse pruning proved to be beneficial for the training process, thus this aspect can be safely ignored.

Overall, the training works as expected, and the model learns correctly how to classify data from the task it is associated with.

Table 4.6 describes the final scores obtained by the extensively trained Xylo's models.

Extended Training was very impactful on Xylo's models, providing a good performance boost among all the tasks.
In particular, it was extremely effective when applied to the most difficult tasks, where this procedure improved accuracy by up to 5 %.

Task	HPO	Validation	Test
2CB	99.2%	99.5%	99.3%
2CW	81.5%	86.3%	86.2%
3CB	91.2%	93.0%	92.7%
3CW	69.5%	74.8%	73.5%
4CB	94.0%	94.8%	93.7%
4CW	56.3%	59.7%	59.0%
7CB	78.1%	80.8%	79.9%
7CW	58.3%	59.0%	59.4%

Table 4.6: Results of the Extended Training process on Xylo. The performance is measured in terms of accuracy.

Furthermore, this process reduced or resolved the data selection process anomalies identified during the analysis of the HPO step's results, with the trained model behaving better on harder tasks for the couples 3CB/4CB and 4CW/7CW. Training for a longer period fixed the issue for tasks 4CW/7CW and reduced the accuracy distance between tasks 3CB and 4CB.

Overall, I have shown that introducing the Extended Training step was extremely beneficial to the good behavior of the models, providing a good foundation for the next phase, related to hardware implementation.

4.4 Quantization Tuning

This section enters the final pipeline macro-area, which is responsible for generating the hardware configuration from the trained models.

In particular, this section will cover the evaluation of the Quantization Tuning process, which tries to handle the loss of performance that the quantization procedure might cause, and recover the correct pre-quantization behavior of the model.

As described in section 3.6.1, Quantization Tuning is based on a hyperparametric optimization process having the goal of finding the best values for a set of parameters that affect the model's operational behavior.

Each tuning parameters configuration has been evaluated by testing each model on the validation set, using the simulator offered by Rockpool for the DYNAP-SE2 and Xylo platforms. Figure 4.6 shows an overview of the parameter configurations tested during HPO, for task 3CB.

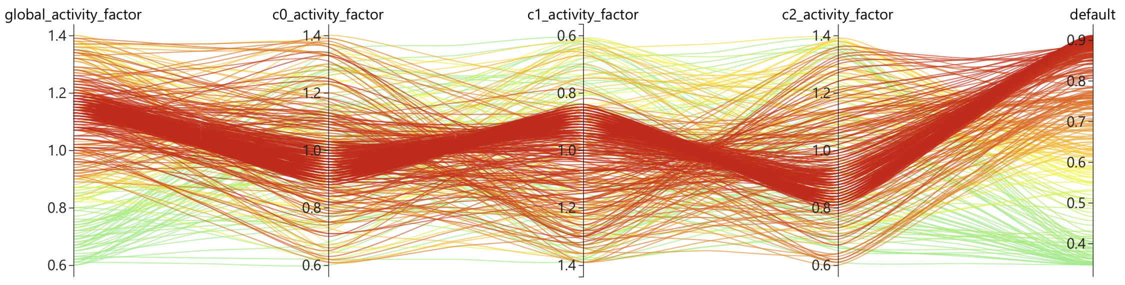


Figure 4.6: Visualization of the hyperparameter configurations tested by NNI during the Quantization Tuning process on Xylo's 3CB task.

The tables 4.7 describe the accuracy scores obtained on each task, analyzing the impact of Quantization Tuning.
The DYNAP-SE2 models suffered an average loss of 4% due to the quantization of network parameters.
On Xylo, the average performance loss was even higher, standing at around 16 percent and peaking at 36.6 percent for the 3CW task.

Moreover, the models associated with the more complex tasks seem to be largely more affected by this phenomenon.

Task	Pre Quantization	Post Quantization	Tuning Applied
2CB	94.1%	89.0%	**94.8%**
2CW	58.0%	55.1%	**62.0%**
3CB	79.4%	73.1%	**81.2%**
3CW	45.4%	37.3%	**45.7%**
4CB	62.9%	65.4%	**67.5%**
4CW	36.4%	30.8%	**36.8%**

DYNAP-SE2

Task	Pre Quantization	Post Quantization	Tuning Applied
2CB	99.5%	97.2%	**99.4%**
2CW	86.3%	53.5%	**80.2%**
3CB	93.0%	81.5%	**91.1%**
3CW	74.8%	38.2%	**66.1%**
4CB	94.8%	91.1%	**92.9%**
4CW	59.7%	44.3%	**53.5%**
7CB	80.8%	65.7%	**71.4%**
7CW	59.0%	43.8%	**48.5%**

Xylo

Table 4.7: Analysis of the Quantization Tuning effects on the HW Configuration Generation step. Models are evaluated on the validation set using devices' simulators. The performance is measured in terms of accuracy.

An interesting finding concerning the DYNAP-SE2 platform is that the results, obtained following the application of the proposed Quantization Tuning technique, turned out to be better than the pre-quantization scores.
This is probably explained by the presence of some problems in Rockpool's DYNAP-SE2 training procedure, which was unable to better optimize the model parameters. Quantization Tuning found a better balance in the model parameters by brute-forcing the problem with the help of HPO, thus achieving better performance.

Thanks to this technique Xylo's models were able to get close to their pre-quantization scores.

As emerges from the results, Quantization Tuning is incredibly beneficial in every tested case, allowing the recovery of a large slice of the performance lost due to quantization of the model parameters.

4.5 Hardware Configuration Generation & Deployment

In the final stage of the pipeline, models generated during the training phases, targeting the tasks defined during the Task Definition process, are converted into equivalent hardware configurations that can be deployed on the target device.

After this process, the final models were tested to evaluate their performance after deployment, assessing how much performance, if any, was lost because of this conversion step.
Furthermore, evaluating the behavior of the deployed models allows to identify the training process's effectiveness in real device execution.

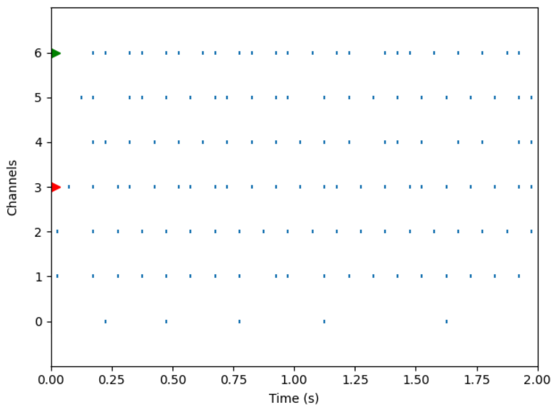
Unfortunately, conducting these tests in real-time directly on the target hardware was unfeasible. With each sample lasting 2 seconds, running each individual sample on the devices would have taken at least as long as the duration of the sample, adding some time for feeding data into the chip and recovering the output network activity.
Considering the substantial number of samples, exceeding 10 thousand even for the smallest 2CB and 2CW tasks, the required time would extend into days.

For this reason, the models were tested using the device's simulators, which replicate the internal working behavior of the devices. In the simulated environment, they can significantly reduce the time required to process each sample, offering a convenient means for conducting model evaluations.
Consequently, the hardware configurations were virtually deployed and evaluated on the DYNAP-SE2's and Xylo's simulators, provided by the Rockpool library.

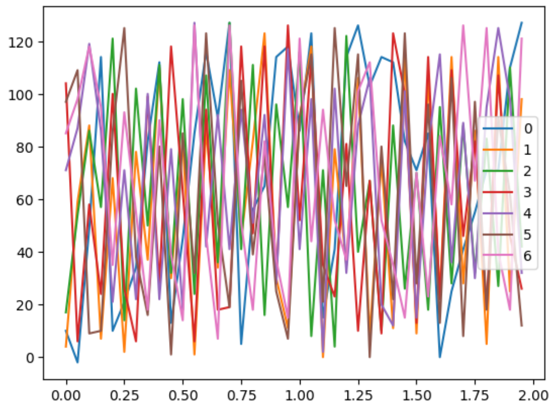
This process began by analyzing the behavior of the simulators to assess the efficacy of the simulation and fidelity to the hardware operation.

Commencing with Xylo, I utilized the Xylo IMU board at my disposal to conduct this comparison.

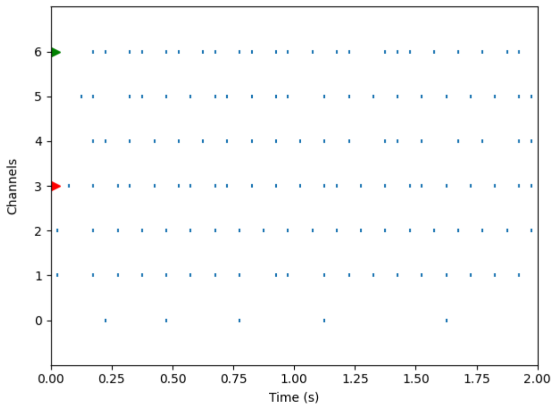
Figure 4.7 illustrates the behavior of both the simulator and the real device, configured with the same model and after being fed with the same data. They are compared based on the produced output events and membrane potentials.



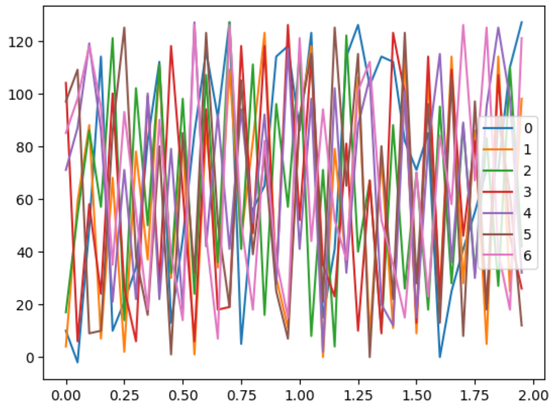
(a) Simulated output activity



(b) Simulated membrane potential



(c) Xylo IMU output activity

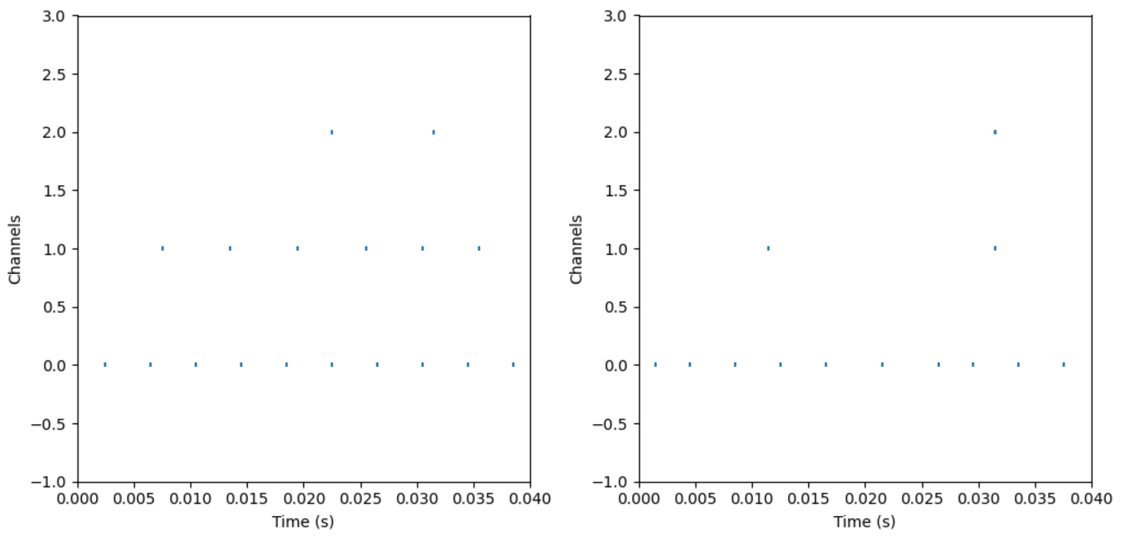


(d) Xylo IMU membrane potential

Figure 4.7: Comparison between the simulator and the real hardware for the Xylo platform. The simulator provided by Rockpool is extremely accurate.

The simulator and hardware behave in an indistinguishable way, producing identical output and membrane potential responses. This test verifies the claims made in the Rockpool documentation that Xylo's simulator is capable of providing a bit-accurate simulation of the hardware.

The outcomes of this comparison are significantly different considering the DYNAP-SE2 simulator. While it effectively simulates the theoretical behavior of the device, the hardware mismatch phenomenon introduces a variability factor. Figure 4.8 compares the output activities generated by the simulator and the real device. As in the Xylo case, both were configured with the same model and fed with the same data.



Simulated output activity DYNAP-SE2 output activity

Figure 4.8: Comparison between the simulator and the real hardware for the DYNAP-SE2 platform. The phenomenon of hardware mismatch introduces variability that prevents achieving a perfectly accurate simulation.

Simulated models may behave differently than they would on the real hardware, with the difference depending on the degree of hardware mismatch, representing how much each specific DYNAP-SE2 board deviates from the ideal behavior.

However, the proposed pipeline utilizes all available methods, offered by the Rockpool library, to ensure mismatch robustness for DYNAP-SE2-specific trained models. Therefore, even if the simulator is not perfectly accurate compared to the hardware, the model trained using the pipeline should be able to function correctly on the real device.

Finally, having validated our testing environment, we can proceed with the analysis of the conclusive results.
Table 4.8 provides an overview of the final accuracy scores for the trained models,

tested on the devices' simulators.

Task	After Training - Test	Validation	Test
2CB	93.2%	94.8%	94.0%
2CW	57.3%	62.0%	61.5%
3CB	79.2%	81.2%	81.0%
3CW	44.4%	45.7%	44.8%
4CB	62.7%	67.5%	67.5%
4CW	35.6%	36.8%	36.1%

DYNAP-SE2

Task	After Training - Test	Validation	Test
2CB	99.3%	99.4%	99.0%
2CW	86.2%	80.2%	80.9%
3CB	92.7%	91.1%	90.4%
3CW	73.5%	66.1%	64.6%
4CB	93.7%	92.9%	92.3%
4CW	59.0%	53.5%	51.5%
7CB	79.9%	71.4%	70.0%
7CW	59.4%	48.5%	48.3%

Xylo

Table 4.8: Results of the Hardware Configuration Generation process. Models are evaluated utilizing the devices' simulator. The performance is measured in terms of accuracy. The "After Training" column refers to the results generated from the extensively trained models.

The results on the device simulators confirm what was seen in the analysis of the previous pipeline steps.
The accuracy scores on the test set generally show a minimal drop if compared to the ones obtained on the validation set. This aspect validates the goodness of the proposed training process, proving that the models should be able to perform correctly on unseen data.

Furthermore, despite the performance loss incurred during the deployment process, the models' behavior did not experience significant disruptions. This is particularly true for easiest tasks, such as 2CB, 3CB, 4CB.
Specifically on Xylo, the scenario changes for more challenging tasks, where, in certain instances, the loss in accuracy exceeded 10%. The performance loss seems

to grow linearly with the difficulty of the task.

Finally, following the validation of the model performance on simulators, I attempted to deploy them on the target devices.

Figure 4.9 shows the behavior of the model associated with task 3CB, deployed on the DYNAP-SE2, for two correctly classified random samples.

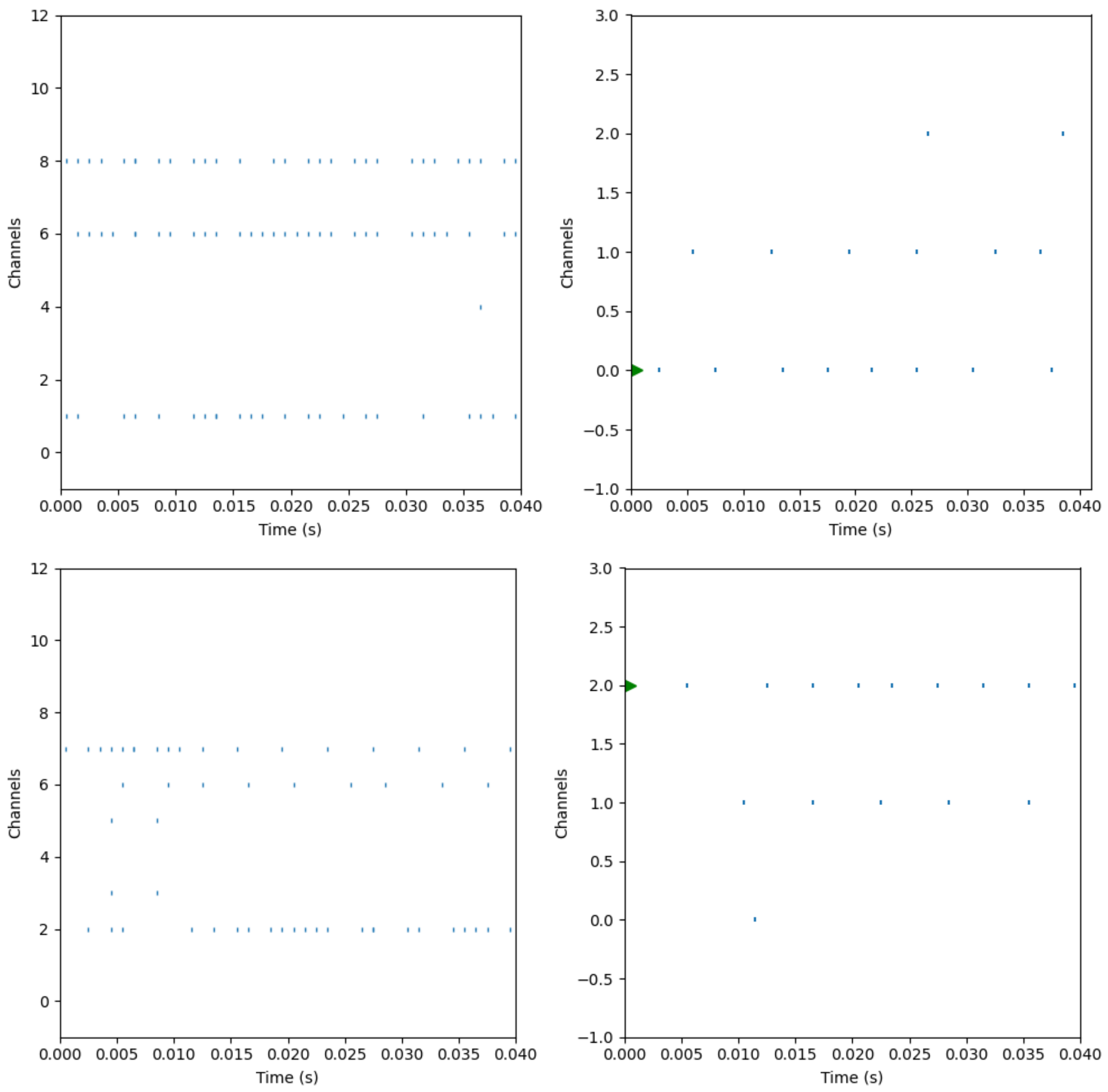


Figure 4.9: DYNAP-SE2 hardware activity on two random samples from task 3CB.

The device was able to correctly classify some randomly selected samples.

However, the output activity from the device proved to be very unstable, necessitating numerous attempts to achieve the desired performance. The temperature of the device appeared to influence its behavior during operation, making the deployment process challenging and not straightforward. This, coupled with the encountered bug during the Architecture Search phase, may indicate the immaturity of the software support for use in a production environment.

Unfortunately, I did not have the opportunity to acquire a Xylo Audio model, which was the specific variant targeted in this thesis work.
Testing any of the final models on a real Xylo device was not possible due to the utilization of a network architecture incompatible with the available device, namely the Xylo IMU. The Feed-Forward Deep^2 Residual network architecture employs 768 hidden neurons, whereas the Xylo IMU only has 496 neurons, as described in section 2.2.2.
However, taking into account the results of the Xylo simulator testing conducted at the beginning of this final evaluation process, it can be reasonably assumed that the deployed model would exhibit behavior on the real hardware similar to what was observed on the simulator.

Chapter 5

Conclusion

The objective of this thesis work was the exploration of the capabilities of immediately available neuromorphic devices, assessing their performance on classification tasks of controlled complexity. This study aims to show the maturity status of neuromorphic hardware and software development tools.

Specifically, this work targets two promising neuromorphic embedded systems: the DYNAP-SE2 and Xylo.

This thesis proposes an end-to-end multi-step pipeline for generating models compatible with the target devices, that can be generalized to work across a wide range of applications.

The proposed pipeline can be subdivided into three macro-areas, which are Task Definition, Model Generation, and Hardware Deployment.

The Task Definition macro-area has the goal of defining a set of tasks for the benchmark, covering a range of complexity levels.
Two metrics have been employed to define the separability of the different classes within the dataset, which is closely tied to the task complexity.
The complexity of a task can be controlled by altering the number of classes it is based on, and their combination, providing insights into how the devices perform in relation to the difficulty of the task.

Next, the Model Generation macro-area aims at building and training the best models possible for each task.
Two device-specific training pipelines have been developed using the available tools for the two platforms, provided with Rockpool, a software library developed and maintained by SynSense.
The generation of the models for the benchmark happens in multiple steps.
At first, the system searches for the best architecture available for the benchmark.
Later, a hyperparameter optimization step searches for the optimal parameters configuration to employ for each task.
Finally, the models are trained extensively for a large number of epochs, looking at extracting the best performance possible on each task.

The last macro-area involves the deployment on the target hardware, in which the models, trained during the previous phases, are mapped to the end devices' hardware architecture, quantized, and converted into equivalent hardware configurations.

In particular, I propose two novel techniques, Synapse Pruning and Quantization Tuning.
Employed during the model training, Synapse Pruning enables the usage of larger networks on Xylo, allowing the adaptation of incompatible network architectures to the device's hardware characteristics and constraints, exploiting better its onboard resources.
Instead, Quantization Tuning finds its importance in the parameter quantization step, during the deployment phase: it alleviates the effects of quantization of the model's parameters, impacting the model behavior, and, thus, causing a performance loss, by re-balancing the network activity.

To provide a real and commonly encountered use case, the proposed pipeline was tested on the Human Activity Recognition classification task.
The benchmark is based on a collection of eight tasks of controlled and increasing complexity, defined by subsetting the Wireless Sensor Data Mining dataset using

the proposed method.
The chosen dataset was an ideal fit for this benchmark. Its complexity did not impose limitations on the Task Definition process, allowing for the generation of tasks spanning a broad range of difficulty levels.

The models generated by the pipeline were subsequently evaluated using the available device simulators and tested on real hardware.

Analyzing the results, I assessed that the proposed pipeline was able to produce well-trained, hardware-ready models, independently from the task they were linked to.
The model performed well even after deployment, proving the effectiveness of the proposed training and fine-tuning techniques. In fact, Synapse Pruning and Quantization Tuning played an important role in reaching these performance levels.

Overall, the proposed procedures were effective in producing a meaningful benchmark.
The target platforms were able to perform greatly on the easiest tasks but ended up struggling on the hardest ones.
These results provide an important snapshot of the current capabilities of the targeted neuromorphic devices.

The DYNAP-SE2 platform employs a very promising hardware architecture, based on ultra-low-power analog neurons and synapses, and on an advanced routing scheme.
However, it was heavily disadvantaged due to poor software support.
The impossibility of using multi-layer architectures, caused by the bug cited before, and by the difficulty of obtaining consistent behavior with multi-layer networks due to the hardware mismatch phenomenon, was very detrimental, badly limiting the maximum performance level reachable by the device.

The hardware mismatch is very impactful and becomes a problem if seen from a classification task point of view, making it difficult to foresee a large-scale application of the device, without providing ways to alleviate it.

A way could be to identify and map the mismatch inherent in each board, building some correcting techniques able to make each device behave ideally.

Xylo is very mature, as it is its training procedure proposed in Rockpool.
The provided bit-precise simulator is very useful for developing models, safely assuming that the simulated network activity will be identical after hardware deployment.
This device, already commercialized, proved to be ready to be employed in production environments, and applied to general-purpose use cases.

Generally, SynSense-developed tools, such as Rockpool, Tonic, and Samna were fundamental for this benchmark. These libraries contain the most advanced generally available methods to develop and train models for the Xylo and DYNAP-SE2 platforms.
Being fairly recent and still undergoing active and continuous development, I encountered some bugs, promptly reported on GitHub, that negatively influenced some of the technical choices I made during this work.
Fixing these should be a priority to ensure a better user experience and extract the best performance possible from the supported devices.

In conclusion, this benchmark highlighted the maturity levels of the targeted neuromorphic devices.
Considering the significant advantages inherent in the neuromorphic paradigm, particularly in terms of power-to-accuracy ratio, these devices demonstrate the capability to achieve accurate classification on mobile devices.

The evolving neuromorphic paradigm, supported by continuous improvements in both hardware devices and software support, has the potential to become a new standard solution for ultra-low-power and ultra-low-latency applications.

Acknowledgements

I want to express my gratitude to several people and entities who played a crucial role in the success of my project.

First and foremost, I would like to thank Prof. Gianvito Urgese, my relator, for his invaluable guidance and insights into the realm of neuromorphic computing. His prompt assistance significantly contributed to my understanding of this fascinating field.

I extend my appreciation to Vittorio Fra Ph.D. and Riccardo Pignari for their fundamental support in troubleshooting issues that arose during the development phase. Their expertise proved instrumental in overcoming challenges and refining the project.

Special thanks to the EDA group of "Politecnico di Torino" for providing access to the computational resources essential for conducting experiments. Part of the investigation activity has been funded by the Ebrains-Italy project (CUP B51E22000150006).

I would also like to acknowledge the Institute of Neuroinformatics of Zürich, for providing me with an environment conducive to the conception and implementation of important phases of the project.

In particular, I am grateful to Prof. Giacomo Indiveri for guiding me during my stay at the INI, for his generosity in sharing knowledge about DYNAP-SE2 and for granting access to the device.

I would like to acknowledge SynSense and Dr. Dylan Muir for their pivotal role in lending the Xylo board and offering assistance in addressing project-related issues.

Finally, a shout-out to Uğurcan Çakal for his invaluable assistance with both the DYNAP-SE2 and Xylo platforms throughout the development of the project.

Each of these people and entities has significantly enriched my project experience, and I am sincerely thankful for their collaboration and support.

Infine, un ringraziamento speciale va a tutta la mia famiglia ed ai miei amici.

Ad Arianna, per essere al mio fianco da più di cinque anni, supportandomi in tutte le mie scelte, per essere la migliore compagna di viaggio che avessi mai potuto desiderare e per rendere tutto il mio mondo infinitamente più colorato.

A mia madre Maria Grazia, per il suo supporto totale e per essere la colonna portante della mia vita, grazie alla quale sono diventato la persona che sono ora.

A mia sorella Alessandra, per essere una fonte di ispirazione ed un punto fisso della mia vita, anche a centinaia di chilometri di distanza.

A mia nonna Marcella e mia zia Gabriella, per essere le fondamenta della mia famiglia e per il loro instancabile sostegno durante l'intera durata dei miei studi.

Ad Andrea e Ilo, per donare affetto alle persone più significative nella mia vita.

Ai miei amici di una vita, che ci sono sempre stati, donandomi spensieratezza anche nei momenti più bui.

Infine a mio nonno Antonio, che, seppur involontariamente, è stato la mia ispirazione per diventare l'ingegnere che sono oggi.

Bibliography

[1] C. Mead. «Neuromorphic electronic systems». In: *Proceedings of the IEEE* 78.10 (1990), pp. 1629–1636. DOI: 10.1109/5.58356.

[2] Carver Mead. «How we created neuromorphic engineering». In: *Nature Electronics* 3.7 (2020), pp. 434–435.

[3] Qing Wan et al. «2022 roadmap on neuromorphic devices and applications research in China». In: *Neuromorphic Computing and Engineering* 2.4 (Dec. 2022), p. 042501. DOI: 10.1088/2634-4386/ac7a5a. URL: https://dx.doi.org/10.1088/2634-4386/ac7a5a.

[4] «Beyond von Neumann». In: *Nature Nanotechnology* 15.7 (July 2020), pp. 507–507. ISSN: 1748-3395. DOI: 10.1038/s41565-020-0738-x. URL: https://doi.org/10.1038/s41565-020-0738-x.

[5] Wikipedia contributors. *Biological neuron model — Wikipedia, The Free Encyclopedia.* [Online; accessed 2-November-2023]. 2023. URL: https://en.wikipedia.org/w/index.php?title=Biological_neuron_model&oldid=1179916565.

[6] Louis Lapicque. «Recherches quantitatives sur l'excitation électrique des nerfs traitée comme une polarisation». In: *J. Physiol. Pathol. Gen.* 9 (1907), pp. 620–635.

[7] Robert Patton et al. «Neuromorphic Computing for Scientific Applications». In: *2022 IEEE/ACM Redefining Scalability for Diversely Heterogeneous Architectures Workshop (RSDHA)*. 2022, pp. 22–28. DOI: 10.1109/RSDHA56811.2022.00008.

[8] Catherine D. Schuman et al. «Opportunities for neuromorphic computing algorithms and applications». In: *Nature Computational Science* 2.1 (Jan. 2022), pp. 10–19. ISSN: 2662-8457. DOI: 10.1038/s43588-021-00184-y. URL: https://doi.org/10.1038/s43588-021-00184-y.

[9] Zexiang Yi et al. «Learning rules in spiking neural networks: A survey». In: *Neurocomputing* 531 (2023), pp. 163–179. ISSN: 0925-2312. DOI: https://doi.org/10.1016/j.neucom.2023.02.026. URL: https://www.sciencedirect.com/science/article/pii/S0925231223001662.

[10] Eric Hunsberger and Chris Eliasmith. «Spiking Deep Networks with LIF Neurons». In: *CoRR* abs/1510.08829 (2015). arXiv: 1510.08829. URL: http://arxiv.org/abs/1510.08829.

[11] Emre O. Neftci, Hesham Mostafa, and Friedemann Zenke. «Surrogate Gradient Learning in Spiking Neural Networks: Bringing the Power of Gradient-Based Optimization to Spiking Neural Networks». In: *IEEE Signal Processing Magazine* 36.6 (2019), pp. 51–63. DOI: 10.1109/MSP.2019.2931595.

[12] Timothée Masquelier, Rudy Guyonneau, and Simon Thorpe. «Competitive STDP-Based Spike Pattern Learning». In: *Neural computation* 21 (May 2009), pp. 1259–76. DOI: 10.1162/neco.2008.06-08-804.

[13] Steve B. Furber et al. «Overview of the SpiNNaker System Architecture». In: *IEEE Transactions on Computers* 62.12 (2013), pp. 2454–2467. DOI: 10.1109/TC.2012.142.

[14] www.datacenterdynamics.com. *SpiNNaker brain simulation project hits one million cores on a single machine.* [Online; accessed 23-October-2023]. 2018. URL: https://www.datacenterdynamics.com/news/spinnaker-brain-simulation-project-hits-one-million-cores-single-machine/.

[15] University of Heidelberg. *BrainScaleS project.* [Online; accessed 23-October-2023]. URL: `https://brainscales.kip.uni-heidelberg.de`.

[16] Lei Deng et al. «Tianjic: A Unified and Scalable Chip Bridging Spike-Based and Continuous Neural Computation». In: *IEEE Journal of Solid-State Circuits* PP (Feb. 2020), pp. 1–19. DOI: `10.1109/JSSC.2020.2970709`.

[17] Paul A. Merolla et al. «A million spiking-neuron integrated circuit with a scalable communication network and interface». In: *Science* 345.6197 (2014), pp. 668–673. DOI: `10.1126/science.1254642`. eprint: `https://www.science.org/doi/pdf/10.1126/science.1254642`. URL: `https://www.science.org/doi/abs/10.1126/science.1254642`.

[18] Mike Davies et al. «Loihi: A Neuromorphic Manycore Processor with On-Chip Learning». In: *IEEE Micro* 38.1 (2018), pp. 82–99. DOI: `10.1109/MM.2018.112130359`.

[19] Garrick Orchard et al. *Efficient Neuromorphic Signal Processing with Loihi 2*. 2021. arXiv: `2111.03746 [cs.ET]`.

[20] Vittorio Fra et al. «Human activity recognition: suitability of a neuromorphic approach for on-edge AIoT applications». In: *Neuromorphic Computing and Engineering* 2.1 (Feb. 2022), p. 014006. DOI: `10.1088/2634-4386/ac4c38`. URL: `https://dx.doi.org/10.1088/2634-4386/ac4c38`.

[21] Paul Lukowicz et al. «Recording a Complex, Multi Modal Activity Data Set for Context Recognition». In: Jan. 2010, pp. 161–166.

[22] Jorge Reyes-Ortiz et al. *Human Activity Recognition Using Smartphones.* UCI Machine Learning Repository. DOI: https://doi.org/10.24432/C54S4K. 2012.

[23] Jennifer R. Kwapisz, Gary M. Weiss, and Samuel A. Moore. «Activity Recognition Using Cell Phone Accelerometers». In: *SIGKDD Explor. Newsl.* 12.2 (Mar. 2011), pp. 74–82. ISSN: 1931-0145. DOI: `10.1145/1964897.1964918`. URL: `https://doi.org/10.1145/1964897.1964918`.

[24] Gary Weiss et al. «Actitracker: A Smartphone-Based Activity Recognition System for Improving Health and Well-Being». In: Oct. 2016, pp. 682–688. DOI: 10.1109/DSAA.2016.89.

[25] Attila Reiss. *PAMAP2 Physical Activity Monitoring*. UCI Machine Learning Repository. DOI: https://doi.org/10.24432/C5NW2H. 2012.

[26] Gary Weiss. *WISDM Smartphone and Smartwatch Activity and Biometrics Dataset*. UCI Machine Learning Repository. 2019. DOI: https://doi.org/10.24432/C5HK59.

[27] Saber Moradi et al. «A Scalable Multicore Architecture With Heterogeneous Memory Structures for Dynamic Neuromorphic Asynchronous Processors (DYNAPs)». In: *IEEE Transactions on Biomedical Circuits and Systems* 12.1 (Feb. 2018), pp. 106–122. DOI: 10.1109/tbcas.2017.2759700. URL: https://doi.org/10.1109%2Ftbcas.2017.2759700.

[28] Dmitrii Zendrikov, Sergio Solinas, and Giacomo Indiveri. «Brain-inspired methods for achieving robust computation in heterogeneous mixed-signal neuromorphic processing systems». In: *bioRxiv* (2022). DOI: 10.1101/2022.10.26.513846. eprint: https://www.biorxiv.org/content/early/2022/10/27/2022.10.26.513846.full.pdf. URL: https://www.biorxiv.org/content/early/2022/10/27/2022.10.26.513846.

[29] Julian Büchel et al. «Supervised training of spiking neural networks for robust deployment on mixed-signal neuromorphic processors». In: *Scientific Reports* 11.1 (Dec. 2021), p. 23376. ISSN: 2045-2322. DOI: 10.1038/s41598-021-02779-x. URL: https://doi.org/10.1038/s41598-021-02779-x.

[30] Ole Richter et al. *DYNAP-SE2: a scalable multi-core dynamic neuromorphic asynchronous spiking neural network processor*. 2023. arXiv: 2310.00564 [cs.NE].

[31] SynSense. *SynSense*. 2023. URL: https://www.synsense.ai.

[32] Hannah Bos and Dylan Muir. «Sub-mW Neuromorphic SNN audio processing applications with Rockpool and Xylo». In: *Embedded Artificial Intelligence: Devices, Embedded Systems, and Industrial Applications* (2023), p. 69.

[33] Philipp Weidel and Sadique Sheik. «WaveSense: Efficient Temporal Convolutions with Spiking Neural Networks for Keyword Spotting». In: *arXiv preprint arXiv:2111.01456* (2021).

[34] Uğurcan Çakal, Ilkay Ulusoy, and Dylan R. Muir. *Training and Deploying Spiking NN Applications to the Mixed-Signal Neuromorphic Chip Dynap-SE2 with Rockpool.* 2023. arXiv: `2303.12167 [cs.ET]`.

[35] Gregor Lenz et al. *Tonic: event-based datasets and transformations.* Version 0.4.0. Documentation available under https://tonic.readthedocs.io. July 2021. DOI: `10.5281/zenodo.5079802`. URL: `https://doi.org/10.5281/zenodo.5079802`.

[36] Microsoft. *Neural Network Intelligence.* Version 2.0. Jan. 2021. URL: `https://github.com/microsoft/nni`.

[37] Donald J. Berndt and James Clifford. «Using Dynamic Time Warping to Find Patterns in Time Series». In: *Proceedings of the 3rd International Conference on Knowledge Discovery and Data Mining.* AAAIWS'94. Seattle, WA: AAAI Press, 1994, pp. 359–370.

[38] Byoung-Kee Yi, H.V. Jagadish, and C. Faloutsos. «Efficient retrieval of similar time sequences under time warping». In: *Proceedings 14th International Conference on Data Engineering.* 1998, pp. 201–208. DOI: `10.1109/ICDE.1998.655778`.

[39] John Aach and George Church. «Aligning Gene Expression Time Series With Time Warping Algorithms». In: *Bioinformatics (Oxford, England)* 17 (July 2001), pp. 495–508. DOI: `10.1093/bioinformatics/17.6.495`.

[40] Romain Tavenard. *An introduction to Dynamic Time Warping.* `https://rtavenar.github.io/blog/dtw.html`. 2021.

[41] Wikipedia contributors. *Kullback–Leibler divergence — Wikipedia, The Free Encyclopedia.* [Online; accessed 22-October-2023]. 2023. URL: https://en.wikipedia.org/w/index.php?title=Kullback%E2%80%93Leibler_divergence&oldid=1175951168.

[42] Wikipedia contributors. *Kernel density estimation — Wikipedia, The Free Encyclopedia.* [Online; accessed 23-October-2023]. 2023. URL: https://en.wikipedia.org/w/index.php?title=Kernel_density_estimation&oldid=1169216425.

[43] Gianvito Urgese et al. «Powering the next-generation IoT applications: new tools and emerging technologies for the development of Neuromorphic System of Systems». In: *Frontiers in Neuroscience* 17 (2023), p. 1197918.

[44] Evelina Forno et al. «Spike encoding techniques for IoT time-varying signals benchmarked on a neuromorphic classification task». In: *Frontiers in Neuroscience* 16 (2022). ISSN: 1662-453X. DOI: 10.3389/fnins.2022.999029. URL: https://www.frontiersin.org/articles/10.3389/fnins.2022.999029.

[45] Wikipedia contributors. *Neuromorphic engineering — Wikipedia, The Free Encyclopedia.* 2023. URL: https://en.wikipedia.org/w/index.php?title=Neuromorphic_engineering&oldid=1177452745.

[46] SynSense. *DYNAP-SE2 — Rockpool.* 2023. URL: https://rockpool.ai/devices/DynapSE/dynapse-overview.html.

[47] SynSense. *Xylo — Rockpool.* 2023. URL: https://rockpool.ai/devices/xylo-overview.html.

[48] SynSense. *Tonic Website.* 2023. URL: https://tonic.readthedocs.io/en/latest/.

[49] SynSense. *Samna Website.* 2023. URL: https://synsense-sys-int.gitlab.io/samna/index.html.