POLITECNICO DI TORINO

Master degree course in Computer Engineering

Master Degree Thesis

Bridging Neuromorphic Platforms for Customized Recurrent Spiking Neural Networks: Human Activity Recognition from snnTorch to Intel Loihi 2

Relatori
Gianvito Urgese
Vittorio Fra

Candidato
Francesco Guarino

December 2024

Abstract

In recent years, conventional Artificial Neural Networks (ANNs) have become essential in research and industry, serving as the primary solution for a wide range of applications. Simultaneously, there is growing interest in Spiking Neural Networks (SNNs), which offer a more biologically plausible model by emulating the human brain's structure and function. SNNs excel in sparse and parallel processing, associative memory and low power consumption. To fully leverage these advantages, specialized neuromorphic hardware is required, shifting from traditional von Neumann architectures to event-driven, asynchronous computation.

This thesis presents a modular approach for designing SNNs suitable for deployment on Intel's Loihi 2 neuromorphic hardware through Intel's own framework, Lava. I utilized mature frameworks like snnTorch and Brevitas to address challenges related to fixed-point arithmetic, weight quantization, and internal state variable quantization inherent in Loihi 2's architecture. My pipeline enables the definition and training of SNNs with recurrent structures, which are crucial for time-series classification tasks.

The focus of this work is on Human Activity Recognition (HAR) using the Wireless Sensor Data Mining (WISDM) dataset, which includes accelerometer and gyroscope data from smartphones and smartwatches across 18 activity classes. To ensure comparability with prior studies, I selected subsets of seven classes based on their separability using the Kullback–Leibler divergence metric. The network architecture comprises an input layer, an output layer, and a hidden layer with two neuron populations. This design incorporates recurrence by feeding back spikes from the output of the hidden layer back to its input, passing through a second inhibitory population, allowing the network to retain the memory of previous time steps.

I addressed data conversion challenges by using the input neuron population as an encoding layer, converting floating-point data into discrete spike outputs distributed over time. For weight quantization, *Brevitas* was employed to train directly with 8-bit quantized weights, sharing the same quantization range across all connection layers. Internal state quantization was managed by leveraging functions within the Lava framework, which converted floating-point decays and thresholds into fixed-point representations suitable for Loihi 2.

The training process was enhanced using the Neural Network Intelligence (NNI), framework for hyperparameter optimization. Sparsity was promoted through dropout layers and a loss function that encouraged low activation frequencies without homogenizing neuron behavior. Results demonstrated that quantization and sparsity enforcement did not adversely affect training; the network achieved a high validation accuracy of 96.5%, comparable to ANNs results on the task. While converting states led to some clipping of system dynamics near the quantization range limits, this saturation of internal states had only a minor impact on performance, lowering the accuracy to 94.8%.

Future work will focus on enhancing the generalizability of the pipeline across various tasks and network architectures. This includes improving quantization techniques to further minimize state saturation effects. Additionally, expanding compatibility with other neuromorphic frameworks, such as Neuromorphic Intermediate Representation (NIR). This will serve to validate the pipeline's applicability and contribute to the broader adoption of SNNs in practical applications.

Contents

List of Figures

List of Tables

Glossary

AHPC After-Hyperpolarizing Compartment. 44, 45

ANN Artificial Neural Network. 3, 4, 13, 14, 18, 22, 34, 43

CUBA LIF CUrrent BAse Leaky Integrate-and-Fire. 54

HAR Human Activity Recognition. 3, 14, 15, 18, 26, 27, 36, 38, 43, 44

HPO Hyperparameter optimization. 8, 56–59, 61, 65, 66, 71

KDE Kernel Density Estimation. 7, 35–40

KLD Küllbach-Leibler Divergence. 7, 35–40

LIF Leaky Integrate and Fire. 14, 19, 42, 44–46, 54

NNI Neural Network Intelligence. 4, 7, 35, 57–59, 65

RSTDP Reward-modulated Spike-Timing-Dependent Plasticity. 85

SNN Spiking Neural Network. 3, 4, 13–15, 17, 18, 20–23, 25, 26, 30, 34, 40, 42–44, 50, 56, 58, 59, 65, 70, 86, 87

STDP Spike-Timing-Dependent Plasticity. 7, 14, 17, 23, 24, 85

WISDM Wireless Sensor Data Mining. 3, 7, 27, 36, 37

Chapter 1

Introduction

The rapid evolution of artificial intelligence and machine learning technologies has ushered in a renewed interest in computational paradigms inspired by biological systems. Neuromorphic computing, a concept theorized in 1997 by *Wolfgang Maass* in [1], is one such paradigm that mimics the behavior and structure of the brain, offering a potential pathway to more efficient and powerful artificial neural networks. Unlike traditional ANNs, which rely on continuous signal processing, SNNs leverage an event-driven model that better reflects the functionality of biological neurons. This architecture allows for the simultaneous representation of both spatial and temporal dependencies, enhancing the network's ability to tackle complex tasks.

In SNNs, information transmission is sparse and asynchronous, relying on discrete spike events rather than continuous activations. Each neuron in an SNN remains inactive until it receives a sufficient accumulation of spikes to exceed a threshold, at which point it generates a spike of its own. This approach aligns closely with how neurons in the brain operate, where communication between neurons is triggered by spikes, encoded as electrical pulses. Unlike traditional ANNs, which require continuous floating-point calculations, SNNs operate using binary spikes, allowing for greater power efficiency and enabling computation with minimal energy use. This distinctive approach not only captures spatial dependencies among neurons but also inherently encodes temporal information, making SNNs particularly suited to applications involving time-sensitive or sequential data.

One of the key strengths ofSNNs lies in their ability to process information in real-time. By only activating in response to incoming spikes, SNNs can significantly reduce the amount of computation required, which directly translates into energy savings. This event-driven nature makes SNNs an attractive solution for tasks where power efficiency and low-latency processing are crucial, such as mobile and wearable devices. However, these benefits come with challenges, particularly in the realm of training. The non-differentiable nature of spiking neurons complicates the use of conventional backpropagation (as SLAYER [2]), the backbone of training

in traditional ANNs. Instead, training SNNs often requires alternative approaches, such as surrogate gradient methods [3], conversion techniques from ANNs [4], or biologically inspired methods like STDP [5] These methods, while promising, frequently involve compromises in training efficiency or accuracy, highlighting an area of ongoing research and development.

In recent years, advancements in neuromorphic hardware have brought SNNs closer to real-world applications [6]. Unlike general-purpose processors, which rely on the von Neumann architecture, neuromorphic chips incorporate specialized circuits designed to emulate the operation of biological neurons and synapses. These chips integrate both computational and memory resources, enabling local processing of information in a manner similar to synaptic connections in the brain. Notable examples of such hardware include IBM's TrueNorth [7], SpiNNaker [8] which pioneered large-scale neuromorphic computing; SynSense's Xylo [9], optimized for low-dimensional inputs; and Intel's Loihi [6] [10] and Loihi 2 [11] [12] [13] and spiNNaker 2 [14] which offer flexibility in neuron models and support for complex network topologies. These devices enable the hardware-level implementation of SNNs, allowing researchers to explore their potential outside of purely software-based simulations.

Intel's Loihi 2, in particular, represents a significant step forward in the field. It provides a versatile platform that supports various neuron models, including the commonly used Leaky Integrate and Fire (LIF) model. The design of this chip places an emphasis on energy efficiency and configurability, rendering it an optimal choice for edge computing applications. Loihi 2's architecture facilitates the deployment of SNNs for practical tasks such as object detection, signal processing, and HAR, where real-time responsiveness and low power consumption are paramount. Loihi 2's capacity to integrate with other computing components also positions it as a viable accelerator for offloading specific tasks from the central processing unit, extending the functionality of neuromorphic systems to broader, more practical use cases.

The potential of neuromorphic computing is further amplified by its suitability for battery-powered devices, where energy constraints are a significant concern. In the domain of HAR, for instance, traditional deep learning models, while effective, often struggle with power efficiency when deployed on mobile devices. By leveraging the inherent advantages of SNNs, neuromorphic chips can facilitate HAR on devices like smartphones and wearables, using sensor data to classify activities such as walking, running, and sitting with minimal power draw. This capability opens up new possibilities for mobile applications, enabling continuous monitoring and real-time analysis without the energy burden typical of conventional models.

The advent of hardware platforms like Loihi 2 signals a shift from theoretical exploration to practical deployment of SNNs. Despite their promise, the deployment of SNNs on neuromorphic hardware involves several hurdles. These include

the challenge of encoding input data into spike-based formats, the need for specialized training methods that accommodate the event-driven nature of SNNs, and the intricacies of programming for neuromorphic architectures. Addressing these challenges requires a nuanced understanding of both the software and hardware aspects of neuromorphic computing, as well as careful consideration of the application context to maximize performance and efficiency.

This thesis aims to define, train, and evaluate a custom recurrent Spiking Neural Network on the Intel Loihi 2 platform for HAR The goal is to test the SNN's performance in this field, examining its power efficiency, robustness, and overall suitability for real-time applications. By deploying the SNN on Intel Loihi 2, this work also seeks to address the practical challenges of programming, training, and optimizing SNNs for neuromorphic hardware, contributing to the broader understanding of the applicability of SNNs for edge computing in real-world scenarios. Ultimately, this thesis will provide insights into the viability of a low-power alternative to traditional deep learning models in mobile and wearable technology applications.

Chapter 2

Background

2.1 From classic ANNs to SNNs

Spiking Neural Networks (SNNs) represent a class of artificial neural networks that attempt to mimic the way biological neurons communicate through discrete spikes, as opposed to the continuous activations used in traditional neural networks. SNNs were first theorized in the late 1980s and early 1990s as researchers sought to model more biologically realistic neural processes [15]. They differ from conventional neural networks by encoding information through spikes' timing, allowing them to process temporal data more naturally and with greater energy efficiency.

The early theoretical foundation of SNNs can be traced back to work by neuroscientists like Alan Hodgkin and Andrew Huxley in [16], who described the biophysical mechanisms of spiking neurons in 1952. However, it wasn't until 1997 that the formal concept of SNNs was introduced by Wolfgang Maass in [1] giving significant contributions by defining the computational capabilities of SNNs and showing that they are computationally more powerful than traditional neural networks for certain types of tasks. One important milestone in the evolution of SNNs came in 1997 with the introduction of STDP, a learning rule that adjusts the strength of synaptic connections based on the precise timing of spikes. This discovery, led by Henry Markram and colleagues, was significant because it aligned closely with how synaptic learning occurs in the brain, advancing both AI and neuroscience.

In the 2000s, it gained momentum with the development of neuromorphic hardware—specialized chips designed to run SNNs efficiently. In particular, IBM's TrueNorth [7] and Intel's Loihi [17] are notable examples of neuromorphic processors built to leverage the sparse, event-driven nature of SNNs for low-power applications. These chips have been demonstrating the practical advantages of SNNs, particularly in power-sensitive applications like mobile devices and IoT. Today, SNNs are an active area of research, with advancements in both algorithms and hardware driving new applications in areas such as real-time sensory processing, robotics, and edge computing. As neuromorphic technology continues to improve,

SNNs are poised to play an increasingly prominent role in AI, particularly for tasks that require temporal precision and energy efficiency.

The shift from conventional (ANNs) to (SNNs) reflects the pursuit of more energy-efficient, temporally dynamic, and biologically inspired AI. SNNs, which use discrete spikes to transmit information, are designed to mimic the brain's neural processes more closely than traditional ANNs. With the emergence of neuromorphic hardware, SNNs are increasingly being adopted for applications that demand real-time processing, low power consumption, and enhanced adaptability. Here are the key reasons driving this transition:

1. **Energy Efficiency**: SNNs only process information when spikes occur, unlike conventional ANNs, which operate on continuous signals and require constant updates. This event-driven approach significantly reduces energy consumption, especially when idle.

2. **Temporal Dynamics**: SNNs inherently model time, allowing them to handle sequential and time-sensitive data more effectively than ANNs, which require architectures like LSTMs or GRUs. This makes SNNs well-suited for tasks such as HAR, where understanding temporal patterns is essential.

3. **Spike Timing and Processing**: By using the precise timing of spikes, SNNs can represent and process temporal information directly, enabling them to respond to dynamic changes in data streams more naturally than conventional networks.

4. **Biological Plausibility**: SNNs simulate the behavior of biological neurons by using spikes for information transfer, closely resembling the way neurons in the brain communicate. This approach is not only more interpretable for studying neural mechanisms but also aligns well with neuroscience research.

5. **Scalability and Parallelism**: Neuromorphic hardware enables SNNs to scale with a high degree of parallelism, as each neuron can operate independently. This is beneficial for complex, large-scale tasks requiring substantial computational resources.

6. **Sparse Connectivity**: Unlike ANNs, which often require dense connections and extensive data exchange, SNNs rely on sparse connections. This reduces communication overhead, allowing SNNs to scale efficiently and maintain performance as the network size grows.

7. **Resilience to Noise**: The event-driven, spike-based processing in SNNs makes them more tolerant to noise and random perturbations in input data, as they do not continuously update based on every small change.

2.1.1 Leaky Integrate and Fire (LIF)

LIF 2.1 neurons serve as essential models in computational neuroscience, reflecting the behavior of biological neurons while providing a framework for mathematical analysis. Biologically, neurons integrate incoming electrical signals through their dendrites, where the membrane, acting as a boundary filled with intracellular saline, generates an action potential when ion channels, such as sodium (Na^+), facilitate the flow of ions. This movement creates the voltage changes that trigger communication with other neurons.

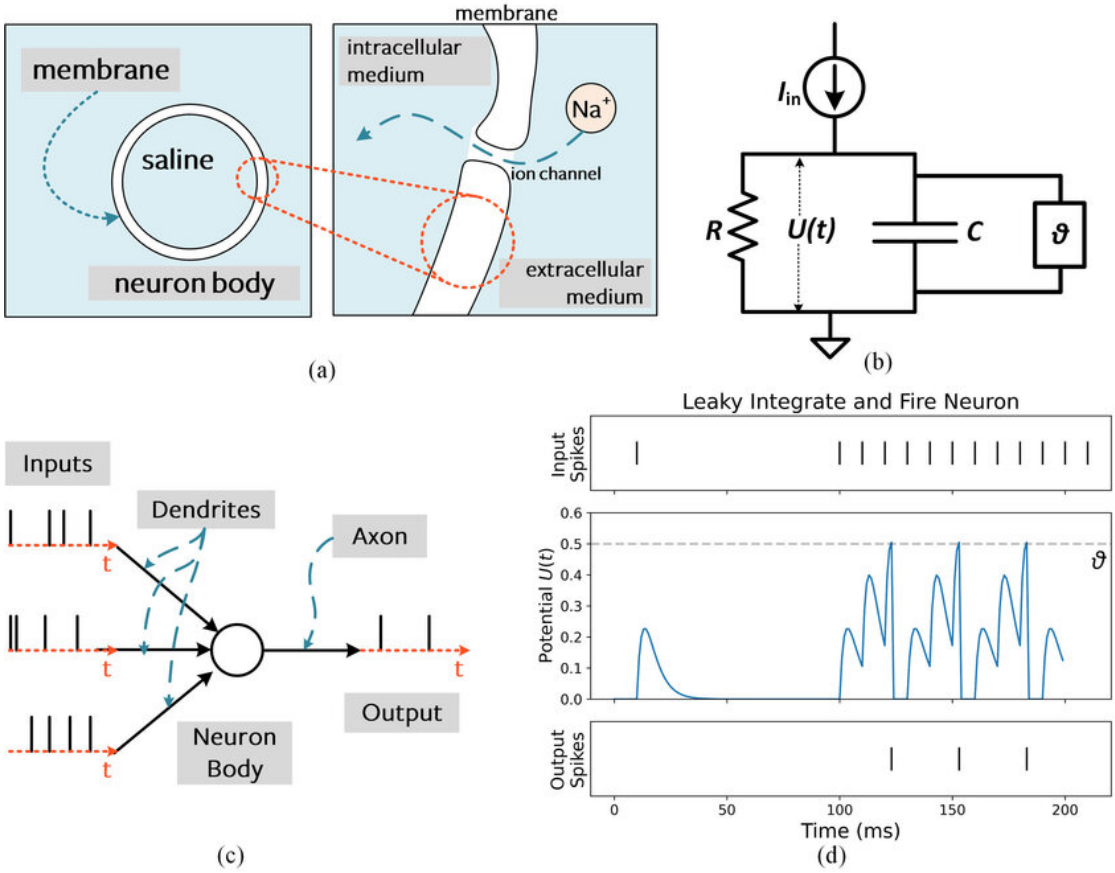


Figure 2.1: a) Biological visualization of membrane potential. b) Representation of equivalent circuit. c) Visual representation of elements constituting the neuron model. d) Evolution of membrane potential and spikes over time. Image from [18]

In the LIF model [19], this process can be represented as an electrical circuit (Figure 2.1.b) where incoming current I_{in} charges the membrane potential $U(t)$, constrained by membrane resistance R and capacitance C. This simple representation captures the integration of input and the natural decay of potential over time due to leakage. When $U(t)$ reaches a defined threshold ϑ, a spike is generated,

mirroring the firing behavior of biological neurons.

The behavior of LIF neurons can be modeled with mathematical functions 2.1 and 2.2 that describe how synaptic currents and membrane potentials evolve.

$$I_{\text{syn}}[t+1] = \alpha I_{\text{syn}}[t] + WX[t+1] \quad \text{where} \quad \alpha = e^{-\Delta t/\tau_{\text{syn}}} \tag{2.1}$$

$$U[t+1] = \beta U[t] + I_{\text{syn}}[t+1] - R[t] \quad \text{where} \quad \beta = e^{-\Delta t/\tau_{\text{mem}}} \tag{2.2}$$

The spiking condition is given by Equation 2.3.

$$S_{\text{out}}[t] = \begin{cases} 1, & \text{if } U[t] > U_{\text{thr}}, \\ 0, & \text{otherwise.} \end{cases} \tag{2.3}$$

where:

- $I_{\text{syn}}[t]$ is the synaptic current at time step t,
- $U[t]$ is the membrane potential at time step t,
- W represents the weight matrix,
- $X[t+1]$ is the input at time step $t+1$,
- $R[t]$ is the reset term after spiking,
- α and β are decay factors related to the synaptic and membrane time constants (τ_{syn} and τ_{mem}, respectively),
- Δt is the time step size,
- U_{thr} is the threshold for spiking.

2.1.2 Data Encoding

Data encoding in Spiking Neural Networks (SNNs) translates continuous input information into spike-based representations, crucial for conveying information through discrete spiking events. Common encoding methods include rate coding, latency coding, and delta modulation, each with applications tailored to specific aspects of sensory processing and neural computation [20].

1. **Rate Coding**: This method encodes input intensity as a firing rate or spike count, where higher input intensities are represented by higher spike frequencies. Rate coding is widely used in SNNs for applications such as image recognition [21], where pixel intensity can be mapped to neuron firing rates, allowing networks to process visual information similarly to traditional deep

learning networks. In recurrent SNNs, rate coding can carry information over timesteps, making it suitable for time-dependent processing tasks. This encoding scheme is also applied in speech processing [22] to represent phonetic intensity over time.

2. **Latency (Temporal) Coding**: Here, the timing of spikes represents input intensity, with earlier spikes corresponding to stronger inputs. Latency coding is effective for applications requiring precise timing, such as sound localization [23], where the relative spike pathways can pinpoint sound direction. Additionally, latency coding is employed in tactile sensing systems [24], where touch intensity is encoded by the spike timing, allowing rapid detection of physical pressure on neuromorphic sensors.

3. **Delta Modulation**: Also known as "threshold crossing," delta modulation encodes input by generating spikes only when there is a substantial change in input intensity. This encoding method is commonly used in event-based vision systems [25], where spikes are triggered only by changes in pixel intensity, drastically reducing data and power consumption in stable conditions. Delta modulation is also applied in anomaly detection [26], where infrequent but significant changes in sensor data trigger spikes, allowing the network to focus only on relevant dynamic information.

2.1.3 Training spiking neural network

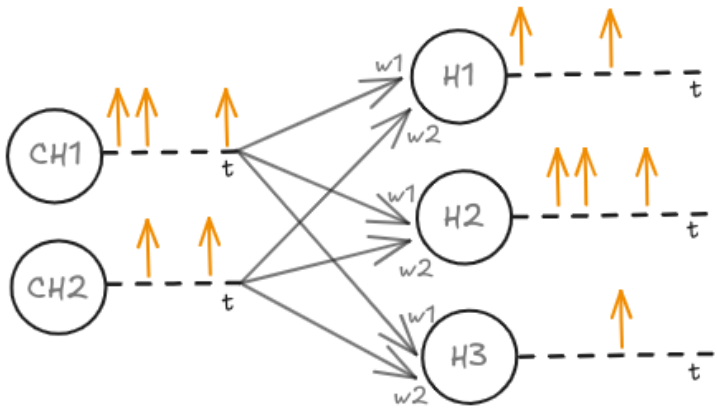


Figure 2.2

In SNNs 2.2, communication between neurons relies solely on binary signals, represented as ones and zeros, generated according to the neuron activation function. This discrete signaling approach mirrors biological neurons, where spikes represent the only form of data exchange across the network.

However, the spiking activation function is inherently non-differentiable, posing significant challenges for traditional training methods such as back-propagation. The standard back-propagation technique relies on gradient-based optimization, which necessitates differentiability in order to compute weight updates in an efficient manner. Consequently, the direct application of back-propagation in SNNs is not a viable approach without either altering the network's behaviour during training or selecting an alternative method of updating the models' weights. The standard back-propagation technique relies on gradient-based optimisation, which necessitates differentiability in order to compute weight updates in an efficient manner. Consequently, the direct application of back-propagation in SNNs is not a viable approach without either altering the network's behavior during training or selecting an alternative method of updating the models' weights.

A comprehensive account of the methodologies that have yet to be explored is provided in [18]. The following is a brief summary of these methodologies.

1. **From already existing ANNs**: Shadow training is a technique for training SNNs by first training an equivalent non-spiking ANN and then converting it into an SNN [4]. In this approach, the activations of the ANN are interpreted as spike-based metrics, such as firing rate or spike timing, allowing SNNs to benefit from conventional deep learning methods.

   In shadow training, the ANN is trained on the target task, and its activations are mapped to spike-based representations in the SNN:

   - **Firing Rate Mapping**: ANN activations are interpreted as firing rates, where the activation of a neuron in the ANN corresponds to the rate of spikes generated by the corresponding neuron in the SNN.
   - **Spike Timing Mapping**: Alternatively, ANN activations can be mapped to specific spike times, where higher activation translates to earlier spiking.

   This approach allows SNNs to leverage well-established deep learning techniques and architectures without requiring complex spike-based backpropagation. It also provides a way to apply recent advancements in ANN-based learning directly to SNNs.

   The utilization of shadow training represents an effective methodology for the initialization of SNNs. This approach facilitates the translation of ANN knowledge into spike-based models, thereby enabling SNNs to inherit the sophisticated capabilities inherent to ANNs for tasks such as classification, detection, and recognition.

   This technique has key drawbacks for temporal dependency tasks. By mapping static ANN activations to SNNs, it fails to capture precise timing dependencies, limiting the SNN's native temporal precision. Additionally, ANNs

process in frames rather than asynchronously, which mismatches the event-driven strengths of SNNs. This approach also increases latency and reduces efficiency, as it doesn't leverage the sparse, spike-based computations that SNNs are optimized for. Consequently, shadow-trained SNNs are less effective for tasks requiring temporal patterns, such as speech or gesture recognition, where direct spike-based learning is better suited.

2. **Local learning rules**: Weight updates are a function of signals that are spatially and temporally local to the weights, rather than from a global signal as in error backpropagation. Into this family of rules we can reference STDP [5]. STDP is a biologically inspired learning rule in SNNs where synaptic weight changes depend on the relative timing of spikes between pre- and post-synaptic neurons.

   If a pre-synaptic neuron fires just before a post-synaptic neuron, the connection weight W is strengthened (**LTP** - Long-Term Potentiation). Conversely, if the post-synaptic neuron fires first, the weight is weakened (**LTD** - Long-Term Depression). This is captured by the STDP update rule as equation 2.4.

$$\Delta W = \begin{cases} A_+ e^{-\Delta t/\tau_+}, & \text{if } \Delta t > 0 \\ -A_- e^{\Delta t/\tau_-}, & \text{if } \Delta t < 0 \end{cases} \tag{2.4}$$

   where:

   - $\Delta t = t_{\text{post}} - t_{\text{pre}}$ is the time difference between post- and pre-synaptic spikes.
   - A_+ and A_- are scaling factors for potentiation and depression.
   - τ_+ and τ_- are time constants controlling the decay rate of potentiation and depression.

   This asymmetric rule encourages causal spike timing: weights increase when pre-synaptic spikes precede post-synaptic spikes, reinforcing pathways that lead to timely activation.

   STDP enables unsupervised learning by strengthening or weakening synapses based on spike timing, capturing temporal correlations in the input data, and promoting efficient synaptic organization within the network.

3. **Backpropagation using spikes**: Backpropagation through spikes in SNNs allows training by approximating gradients for non-differentiable spike events [3]. Spikes are triggered when membrane potential U crosses a threshold θ, represented as equation 2.5.

$$S = H(U - \theta), \tag{2.5}$$

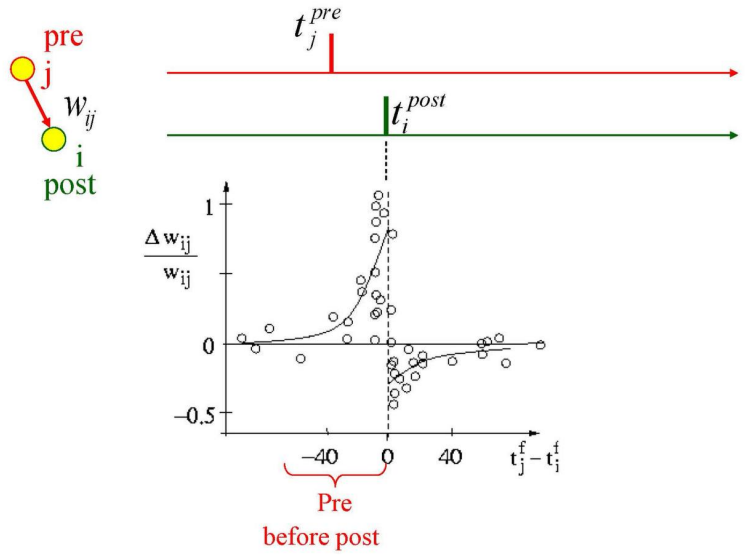


Figure 2.3: Spike-Timing Dependent Plasticity (schematic): The STDP function shows the change of synaptic connections as a function of the relative timing of pre-and postsynaptic spikes after 60 spike pairings. Schematically redrawn after Bi and Poo (1998). From [5].

where H is the Heaviside function. Since $\frac{dS}{dU} = 0$ for all $U \neq \theta$, gradients are zero except at the threshold, causing the **dead neuron problem**—neurons that do not spike receive any gradient and cannot learn.

To enable gradient flow, surrogate gradients replace $H(U - \theta)$ with smooth approximations. Common surrogate functions are:

- **Sigmoid:**
$$\frac{dS}{dU} \approx \sigma(U - \theta) = \frac{1}{1 + e^{-k(U-\theta)}}$$
- **Arctangent:**
$$\frac{dS}{dU} \approx \frac{1}{\pi} \arctan(k(U - \theta)) + 0.5$$

These provide non-zero gradients, even without spikes, enabling neurons to receive "credit" and overcome the dead neuron issue.

The loss gradient with respect to weights W is computed as in equation 2.6:

$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial S} \cdot \frac{\partial S}{\partial U} \cdot \frac{\partial U}{\partial W}, \tag{2.6}$$

where $\frac{\partial L}{\partial S}$ is W base on the Loss, $\frac{\partial S}{\partial U}$ is the surrogate gradient, while $\frac{\partial U}{\partial W}$ is the spiking function of the neurons taken in consideration. This allows effective learning by propagating gradients regardless of neuron firing status, minimizing the overall loss.

4. **SLAYER 2.0** [2]: Advanced surrogate gradient base backpropagation used specifically by Intel's developer inside Lava-dl framework, for training SNNs. Offers significant advantages over conventional surrogate gradient methods by enhancing efficiency, stability, and hardware compatibility. It optimizes temporal unfolding to manage SNNs' time dependencies, reducing redundant operations and improving memory efficiency, which is crucial for sequential tasks. SLAYER 2.0 also employs specialized surrogate gradients, such as the arctangent, for better gradient stability, avoiding issues like vanishing or exploding gradients and achieving faster convergence.

2.1.4 snnTorch

The ***snnTorch*** [18] framework is a Python library built on PyTorch that enables easy implementation and training of SNNs. One of its strengths lies in its seamless integration with PyTorch, which makes it accessible for users already familiar with deep learning workflows. With snnTorch, users can construct SNNs by defining spiking layers like *snn.LIF* (Leaky Integrate-and-Fire) and *snn.IF* (Integrate-and-Fire) neurons directly within the PyTorch *nn.Sequential* or functional API, maintaining a familiar setup and reducing the learning curve.

snnTorch supports surrogate gradient methods to enable backpropagation through spikes, allowing users to select from various approximations like sigmoid and arctangent, simplifying the process of training SNNs. This feature lets users easily experiment with different gradient approximations to improve model training performance. Moreover, *snnTorch* supports hybrid networks, combining both conventional and spiking layers for more flexible architectures.

snnTorch is also highly portable due to its roots in PyTorch, which allows it to run on a diverse range of hardware, including CPUs, GPUs, and even neuromorphic devices compatible with this framework. This portability ensures that models developed on standard hardware can later be transferred to specialized environments, supporting efficient experimentation and deployment. The compatibility with PyTorch libraries, such as *torchvision* and optimization tools, further enhances snnTorch's versatility, enabling integration with a vast ecosystem of machine learning resources.

2.1.5 NeuroBench

NeuroBench [27] is a benchmarking suite designed specifically for evaluating (SNNs) on a variety of neuromorphic tasks. It provides a standardized set of tasks and metrics, enabling consistent and objective comparisons between different SNN models, algorithms, and hardware implementations. *NeuroBench* includes benchmarks that focus on real-time, event-driven processing, aligning well with the capabilities of SNNs and neuromorphic hardware.

The suite supports a range of applications, from sensory processing tasks to temporal sequence learning, helping researchers assess model performance in real-world scenarios that require high efficiency and low latency. By offering a unified framework for evaluation, *NeuroBench* promotes reproducibility and aids in identifying the most effective architectures and training strategies for neuromorphic computing.

2.2 Human Activity Recognition

A deep understanding of the user's surroundings is essential for many systems to achieve their goals. Recognizing user behavior and activities is particularly crucial, as it enables more sophisticated and effective human-computer interactions.

This understanding has broad implications across various fields, including healthcare, autonomous driving, and security surveillance. For example, identifying human actions could enhance the functionality of prosthetic limbs, support fitness assessments, aid in illness prevention, and much more [28].

HAR aims to classify and understand human actions by analyzing motion data from sensors.

This data is processed through a trained classifier, which then identifies the activity. Historically, HAR posed significant challenges due to the lack of precise, commercially available sensors and the limited processing power of portable devices.

Gathering necessary data once required bulky, full-body sensor suits that were cumbersome and inefficient. Real-time data processing on handheld devices was also impractical, as it was too slow and rapidly depleted battery life.

Today, however, advancements in sensor technology and the widespread use of wearable devices, such as smartwatches, have transformed HAR. Modern wearables are equipped with highly accurate sensors and boast impressive processing capabilities for their size, with batteries that can easily last more than a day.

These devices have gained popularity as practical and stylish accessories, often worn continuously throughout the day. This constant usage provides a steady flow of data, enabling systems to adapt and tailor their recognition models to individual users, thereby improving accuracy. Furthermore, it allows for background analysis that was previously unfeasible.

The progress in HAR technology has the potential to drive a new wave of innovations that can significantly enhance human-machine integration across various applications.

2.2.1 WISDM dataset

The WISDM dataset at [29], created by the WISDM Lab at Fordham University, is a rich resource for human activity recognition and biometric research. This dataset was designed to capture and analyze data from the accelerometers and gyroscopes of both smartphones and smartwatches as participants engaged in 18 main activities (2.1. Specifically, 51 subjects, each equipped with a smartphone in their pocket and a smartwatch on their dominant hand, performed 18 distinct activities. These activities encompassed both ambulatory movements, like walking and jogging, and hand-centric tasks, such as brushing teeth, folding clothes, and various eating activities. Each activity was performed for three minutes, accumulating 54 minutes of data per participant and yielding an extensive dataset with over 15 million sensor readings.

Index	**Activity**
0	walking
1	jogging
2	stairs
3	sitting
4	standing
5	typing
6	brushing teeth
7	eating soup
8	eating chips
9	eating pasta
10	drinking
11	eating sandwich
12	kicking soccer
13	catch tennis
14	dribbling basketball
15	writing
16	clapping
17	folding clothes

Table 2.1: Complete List of Activities

Data collection was facilitated by a custom application that ran on both the smartphone and smartwatch, ensuring consistent data acquisition across devices.

The dataset was recorded at a frequency of 20 Hz (one reading every 50 ms), producing time-series data from four separate sensors: the smartphone's accelerometer and gyroscope, and the smartwatch's accelerometer and gyroscope. The smartphone models used were the Google Nexus 5/5X or Samsung Galaxy S5, running Android 6.0, while the smartwatch was the LG G Watch on Android Wear 1.5. This setup allowed for the simultaneous recording of accelerations and rotational movements, which were then tagged with an activity label and a unique subject identifier for ease of analysis.

The data are not only labeled with the specific activity being performed but are also segmented by subject, enabling the use of the dataset for both activity recognition and biometric identification. This dual purpose is further supported by the inclusion of subject identifiers in the dataset, allowing for personalized and user-specific analyses.

In addition to the raw sensor data, the dataset includes processed versions, with examples transformed using a sliding window approach, typical in activity recognition. The sliding window method segments the raw time-series data into non-overlapping windows (in this case, 10-second segments), with each window containing high-level features derived from the raw data. This transformation facilitates machine learning and data mining applications, as many algorithms require fixed-size, feature-rich input rather than raw time-series data. Subsequent works identify a better way to define the overlapping windows, in fact, Vittorio Fra in [30] was able to recognize with high accuracy each sample using 2-second segments, giving a total window of 40 timesteps for each sample. In Table 2.2, are represented some key features of the dataset.

Dataset	**Samples**	**Time Steps**	**Features**
Training	55,404	40	6
Validation	18468	40	6
Testing	18469	40	6
Calibration	100	40	6

Table 2.2: Dataset Sizes for Training, Validation, Testing, and Calibration

Overall, the WISDM dataset provides a comprehensive foundation for researchers exploring human activity recognition, with applications spanning health monitoring, personal fitness, biometric security, and beyond. Its structure and breadth make it a versatile dataset that can support a wide range of analytical techniques and model development efforts, particularly in fields leveraging wearable sensor data for real-time user monitoring and interaction. The dataset is publicly accessible via the UCI Machine Learning Repository[31], promoting further innovation in wearable sensor-based research.

2.3 Intel Loihi 2

Intel's **Loihi 2** represents the latest innovation in neuromorphic computing, building upon the foundational architecture of its predecessor, Loihi 1, with enhanced flexibility, computational power, and resource efficiency. Neuromorphic computing is a technology that emulates the neural networks of the brain to facilitate real-time, adaptive processing at low power consumption, rendering it an optimal choice for artificial intelligence applications in robotics, embedded systems, and edge devices.

2.3.1 Architecture and Key Features

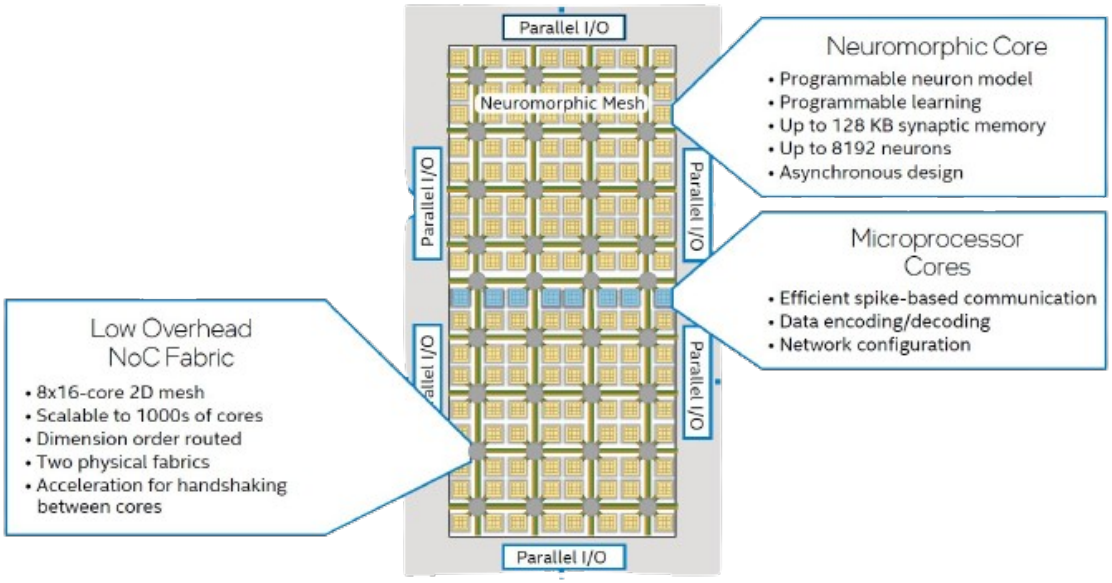


Figure 2.4

Loihi 2 introduces several architectural enhancements as shown in Figure 2.4, making it significantly more versatile and powerful than Loihi 1:

- **Programmable Neuron Models**: Loihi 2 provides fully programmable neuron models that support a wide variety of neural behaviors, allowing the implementation of customized spiking dynamics to match specific computational needs.
- **Graded Spike Events**: Loihi 2's support for graded spikes enables each spike to carry up to a 32-bit integer payload, increasing data precision while retaining the benefits of event-driven communication.

- **Enhanced Learning Rules**: Loihi 2 adds localized "third factors" for on-chip learning, enabling it to perform more complex training algorithms, including approximations of backpropagation. This is essential for supporting deep learning tasks that require gradient-based learning methods.

- **Increased Resource Density**: Built on Intel's 4-nm process, Loihi 2 achieves a 2x increase in resource density, with a die size of just 31 mm^2. This improvement enables up to 1 million neurons and 120 million synapses on a single chip, allowing for larger and more complex SNN models.

- **Onboard x86 Processor**: Loihi 2 incorporates 6 embedded x86 processor that facilitates efficient spike-based input/output processing, network configuration, and data handling, which simplifies integration with other systems. The x86 processor offloads tasks from the neuromorphic cores, optimizing performance and enabling faster data encoding, decoding, and transfer.

- **3D Multi-Chip Scaling**: Loihi 2 introduces a redesigned asynchronous communication fabric with support for 3D multi-chip scaling, allowing up to 16,384 interconnected chips for high scalability. This feature expands the potential of Loihi-based systems to tackle large-scale, distributed neuromorphic applications.

- **Improved Connectivity and Standard Interfaces**: Loihi 2 features standardized interfaces, including 10G Ethernet, GPIO, and SPI, which facilitate integration with conventional computing systems and sensors for real-world deployments.

2.3.2 Performance Improvements

Loihi 2's improvements over Loihi 1 include significant advancements in computational speed, precision, and energy efficiency. These improvements stem from architectural changes such as asynchronous pipelining and enhanced circuit design, which reduce latency and enable faster more efficient neuromorphic processing.

With up to 10x faster spike generation, 5x improved synaptic operations, and 2x faster neuron updates, Loihi 2 delivers substantial speedups over Loihi 1. The introduction of graded spikes and improved learning flexibility make Loihi 2 more adaptable to a broader range of applications, including complex tasks that require precise, adaptive control.

2.3.3 Research Applications and Results

Research conducted by the Intel Neuromorphic Research Community (Intel NRC) [32] has showcased Loihi's capabilities across various fields:

Feature	Loihi 1	Loihi 2
Process Technology	14nm	4nm
Die Size	60 mm²	31 mm²
Max Neuron Count	128,000	1,000,000
Max Synapse Count	128 million	120 million
Neuron Model Programmability	Limited	Fully programmable
Spike Precision	Binary spikes	Graded spikes (32-bit)
Learning Architecture	Two-factor, fixed rules	Three-factor, adaptive rules
Onboard x86 Processor	3	6
Networking Interface	Proprietary	Standard (10G Ethernet, GPIO, SPI)
minimum Time per Timestep	~-	200 ns

Table 2.3: Comparison of Intel Loihi 1 and Loihi 2 Specifications [13]

- **Adaptive Control in Robotics**: Loihi 2 has been employed in robotic arm control [33] and drone [34] systems, achieving fast, adaptive responses with really low power consumption, much lower than what traditional processors require.
- **Real-Time Sensory Processing**: In sensory perception tasks, such as visual-tactile integration [35] and olfactory processing [36], Loihi demonstrated the ability to process multi-sensory data in real time, which is valuable for applications in human-computer interaction.
- **Optimization and Scientific Computing**: Loihi has been applied in challenging optimization problems, such as solving QUBO problems [37]. In these tasks, Loihi-based solutions showed significant energy savings and processing speed improvements, often achieving results an order of magnitude faster and more efficient than conventional approaches.
- **Low Power Efficiency**: Across applications, Loihi 2 has shown an energy efficiency several orders of magnitude higher than traditional systems. This capability makes Loihi 2 well-suited for use in edge devices and embedded systems where power availability is limited.

2.3.4 Available Loihi 2 Hardware Systems

Intel has released two hardware systems to facilitate research and deployment on Loihi 2:

- **Oheo Gulch**: A single-chip system that offers remote access via an Arria 10 FPGA over Ethernet, designed for laboratory research and testing.
- **Kapoho Point**: This compact, stackable system consists of eight Loihi 2 chips with 10G Ethernet interfaces, making it ideal for embedded applications and scalable edge deployments.

Intel's Loihi 2 combines enhanced programmability, high computational efficiency, and adaptability, paving the way for low-power solutions in real-world environments. These advances enable researchers and developers to deploy neuromorphic computing in innovative applications, pushing the boundaries of artificial intelligence.

2.4 Lava Framework

The **Lava** framework [38] is an open-source software platform developed by Intel to facilitate the development of neuro-inspired applications. By leveraging neuromorphic computing principles, Lava enables efficient, real-time processing, and learning with minimal power consumption, which is particularly valuable in embedded systems, robotics, and AI applications. Lava provides a high-level API for designing, simulating, and executing neural network models on both conventional hardware (CPUs, GPUs) and neuromorphic platforms, including Intel's Loihi 2.

2.4.1 Core Structure of Lava

At the heart of Lava is the **Process** concept, which represents the fundamental computational unit in a neuromorphic system. A Process defines the structure, including input/output ports and internal state variables (Vars), but not the implementation details. Processes are designed to interact asynchronously through ports, enabling parallel, event-driven communication between units in a distributed network.

Processes in Lava are defined using `AbstractProcess`, which serves as a blueprint by specifying the process's ports and internal variables without detailing its behavior. This structural definition allows a process to be agnostic of specific hardware platforms, promoting portability and modularity.

2.4.2 Process Models

To implement the actual behavior of a Process, Lava uses **Process Models**. Process Models define the specific computations and logic for a Process, making it executable on different platforms. Each Process can have multiple Process Models, allowing it to run on various resources (CPU, GPU, Loihi 2) or in different languages (e.g., Python, C).

Lava supports two primary types of Process Models:

- **LeafProcessModel**: Implements the behavior of a Process directly, detailing the specific logic needed to execute its intended function.

- **SubProcessModel**: Allows for hierarchical composition by defining a Process's behavior using other, smaller Processes. This enables complex, reusable designs, where larger processes are built from smaller, interlinked sub-processes.

For example, a LeafProcessModel for a Leaky Integrate-and-Fire (LIF) neuron might include computations for updating the membrane potential and emitting spikes when the potential crosses a threshold. By contrast, a SubProcessModel could represent a network of LIF neurons, implementing higher-level neural dynamics through hierarchical connections.

Below is a simple example of a LeafProcessModel in Python for a LIF neuron:

```
from lava.magma.core.model.py.model import PyLoihiProcessModel
from lava.magma.core.decorator import implements, requires, tag
from lava.proc.lif.process import LIF

@implements(proc=LIF)
@requires(CPU)
@tag('floating_pt')
class LIFNeuronModel(PyLoihiProcessModel):
    def run_spk(self):
        # Update and spike logic here
```

2.4.3 Execution of Processes

Processes in Lava are executed by defining both a **Run Condition** and a **Run Configuration**. The `RunCondition` specifies the duration of execution, such as a fixed number of steps or continuous execution until paused or stopped. The `RunConfiguration` sets the target hardware platform, allowing the same Process to execute on different devices with minimal code changes.

For instance, running a LIF neuron Process for 42 steps on a CPU simulation can be achieved as follows:

```
from lava.proc.lif.process import LIF
from lava.magma.core.run_conditions import RunSteps
from lava.magma.core.run_configs import Loihi1SimCfg

lif = LIF(shape=(1,))
lif.run(condition=RunSteps(num_steps=42), run_cfg=Loihi1SimCfg())
```

Lava also supports controlling the lifecycle of a Process through commands like `pause()` and `stop()`, allowing developers to examine and manipulate the internal states mid-execution. This control enables debugging, state inspection, and adjustments during long-running simulations.

2.4.4 Inter-Process Communication

Lava processes communicate asynchronously using **Ports**. Each Process can have multiple `InPort` and `OutPort` objects, which facilitate data exchange and enable the creation of large-scale, distributed networks. Ports support flexible, non-blocking communication where processes can interact without sequential dependencies, supporting parallelism across the network.

Processes can connect dynamically, allowing for complex network configurations. For instance, an output port from one Process can connect to multiple input ports on other Processes, creating broadcast capabilities. This setup is critical for distributed neuromorphic networks, where data needs to flow efficiently across interconnected nodes.

2.4.5 Lava-DL: Deep Learning Extension

Lava-DL [39] is an extension of the Lava framework that adds support for deep learning on neuromorphic hardware. Built around the SLAYER 2.0, an improved version of SLAYER [2] (Spiking Layer Error Reassignment in Time) algorithm, Lava-DL provides tools for training SNNs, supporting various neuron models, synaptic dynamics, and easy-to-use exporting policies to deploy networks on Lava.

Lava-DL integrates seamlessly with PyTorch, a popular deep-learning library, allowing developers to create hybrid models that combine traditional ANNs and SNNs. This flexibility expands the potential of neuromorphic systems, making Lava-DL a powerful toolkit for applications requiring adaptive, event-driven processing in real-time.

Additional tools in Lava-DL include:

- **Event Input/Output Management**: Utilities for handling event-driven data, essential for applications such as sensory processing and real-time decision-making.
- **Visualization and Logging**: Tools for visualizing neuron activity and logging training metrics, which facilitate model development and debugging.
- **Support for Various Learning Rules**: Includes gradient-based and local learning rules, enabling versatile training options on Loihi 2 and other neuromorphic hardware.

By providing these tools, Lava-DL allows developers to design sophisticated, adaptive models optimized for low-power, high-efficiency neuromorphic platforms.

Chapter 3

Materials and methods

This section outlines the materials and methods used in this thesis, providing a foundation for understanding how the various components and ideas were integrated into a cohesive and functional pipeline, as shown in Figure 3.1. The work began with an in-depth analysis of the dataset, focusing on identifying the best subset for optimizing model performance. This involved employing two distinct methods to evaluate separability scores, which provided critical insights for tailoring the dataset splits.

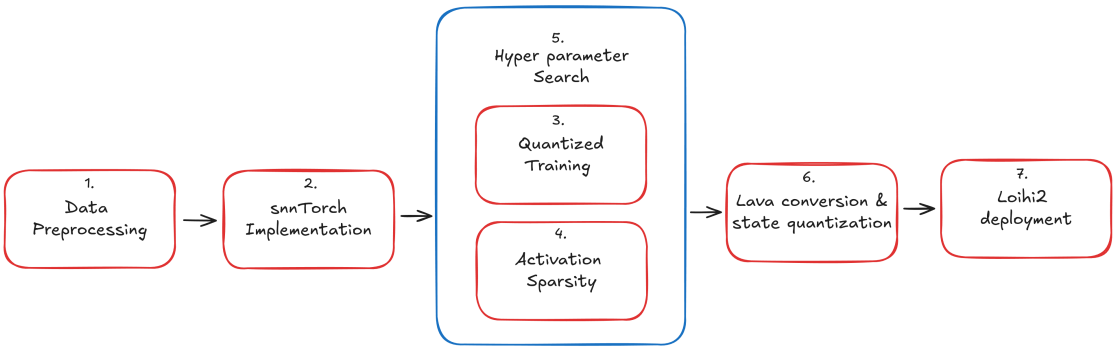


Figure 3.1: 1. Data preprocessing using KDE and KLD. 2. Building the network. 3. Implementation of quantized aware training and 4. Definition of activation sparsity loss. 5. Iterative training using NNI. 6. lava porting and state quantization. 7. Loihi2 deployment and power consumption metrics.

Subsequently, the thesis delves into the architecture of the network itself, offering a detailed explanation of each component and the reasoning behind the design choices. Each network section is thoroughly analyzed, discussing the origin of the design concepts, the rationale for modifications during development, and how these changes contributed to the final behavior.

The section also addresses the challenges introduced during the porting process, examining how these issues were anticipated and managed throughout training.

Particular emphasis is placed on aligning the functionalities of *snnTorch* and *Lava*, including strategies for preserving consistency across frameworks. Additionally, the limitations and implications of quantization are explored in detail, highlighting how fixed-point conversion affects network behavior and performance.

By providing a structured overview of the methodologies and decisions, this section aims to offer a comprehensive understanding of the steps taken to build, train, and deploy the spiking neural network in a robust and effective manner.

3.1 Data pre-processing

To facilitate the attainment of the thesis objective and facilitate comparison with previous research, it was necessary to divide the data set into seven distinct subgroups. In the WISDM dataset, we have data from gyroscope and accelerometer coming from smartphone and smartwatch, that recorded the same activity for each person at the same moment. The data obtained from the smartphone was not utilized in this study due to the presence of certain issues during the data collection process. The smartphone was placed in the subject's pockets without first verifying its orientation and position. This results in more significant complications for the classification process, as it entails the interchange of axes that can potentially impede the training procedure. Only smartwatch data was used for this work, being recorded in a better and more reliable way, due to the position on the wrist that leads to better data characterization for hand-oriented tasks, also counting on a better IMU present in the smartwatch itself. A detailed examination of the KDE for the hand-oriented activity reveals a significant degree of overlap among the constituent classes. This observation underscores the need for a more nuanced approach to classification, particularly in regard to identifying a subset of non-overlapping, classless labels. Doing so will enhance the efficacy of our pipeline in achieving its desired outcomes. KDE is a non-parametric method to estimate the probability density function (PDF) of a random variable based on a finite data sample. Unlike parametric methods, KDE does not assume that the data follows a specific distribution (e.g., normal distribution). KDE alone is insufficient for characterizing the available activity and identifying the optimal subset. As illustrated in Figure 3.2, the majority of classes exhibit substantial overlap, rendering them difficult to differentiate.

KLD Class Separability Metric was used instead to score and select the subsets.

3.1.1 Kullback-Leibler Divergence as Separability Metric

KLD [40], a measure of how one probability distribution diverges from another, can be used to assess class separability in tasks like HAR. In this approach, the KLD is

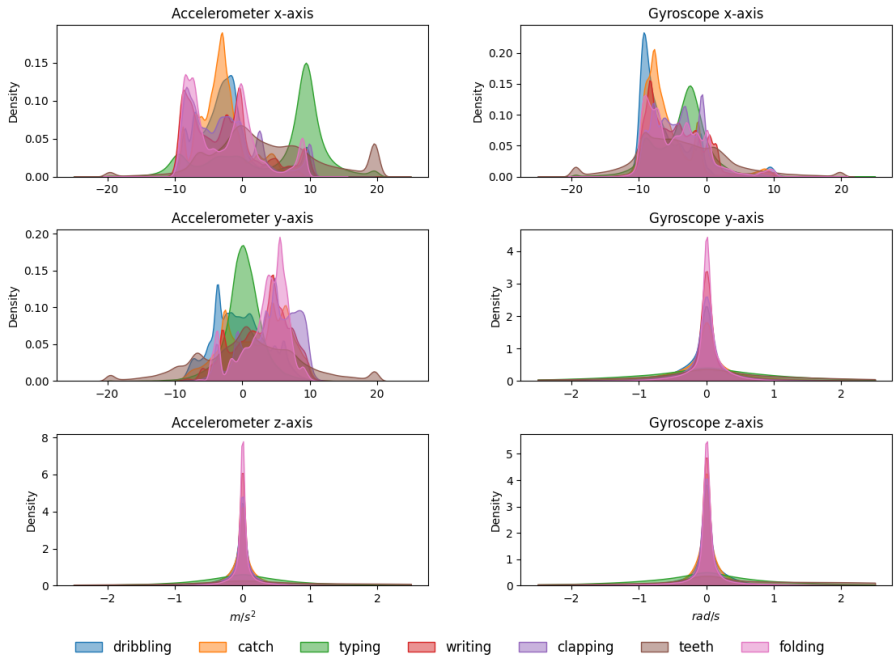


Figure 3.2: Visualization of KDE metrics over WISDM dataset

calculated over the KDE of each class, allowing for a smooth, non-parametric representation of the feature distributions. A visual example can be seen in Figure 3.3

Definition For two probability distributions P and Q over the same variable space X, the KLD Equation 3.1

$$D_{\mathrm{KL}}(P \parallel Q) = \int_{-\infty}^{\infty} p(x) \log \frac{p(x)}{q(x)}\, dx. \tag{3.1}$$

In our case, P and Q represent the KDEs of two different classes. This metric reflects the average inefficiency in encoding samples from P using Q, with a divergence of zero only if P and Q are identical.

Application in Temporal Signal Separability In HAR, temporal signals are divided into segments, and features are extracted from these segments to represent different activities. Each activity's distribution is estimated using KDE, which provides a smoothed, continuous approximation of the feature distribution without assuming a parametric form. Calculating the KLD between the KDEs of different classes (e.g., P_A and P_B for classes A and B) gives a direct measure of class separability. A high KLD $D_{\mathrm{KL}}(P_A \parallel P_B)$ indicates minimal overlap between classes A and B, implying strong separability in the feature space.

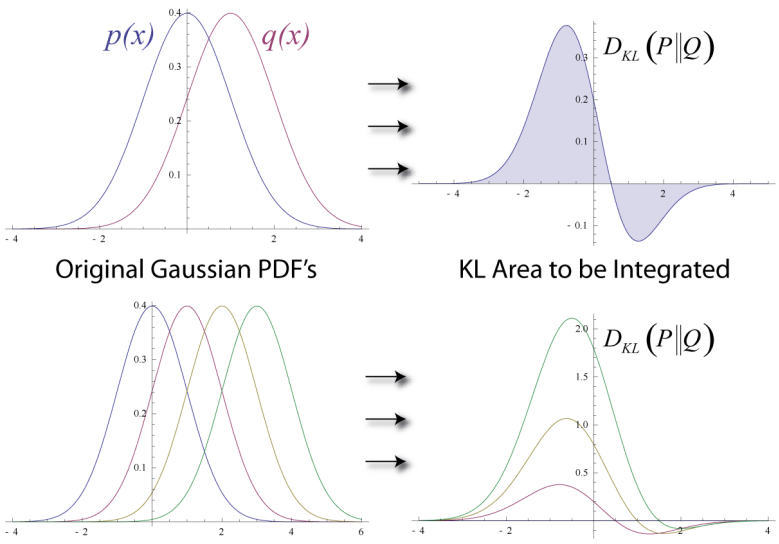


Figure 3.3: Visualization of KDL score over different pairs of functions

Considerations and Advantages Using KDE in conjunction with KLD provides several advantages for temporal signal analysis:

1. **Smooth Distribution Estimation** KDE offers a flexible way to estimate class distributions, capturing subtleties in the data without relying on rigid parametric models. This is particularly useful in HAR, where distributions may vary widely across activities.

2. **Comprehensive Comparison**: By applying KLD over KDEs, we achieve a detailed comparison of distribution shapes, enabling the detection of even subtle distributional differences between classes.

3. **Bandwidth Sensitivity**: While KDE depends on bandwidth selection, careful optimization ensures accurate distribution representation, making KLD a reliable measure of separability when KDE is properly configured.

The combination of KDE and KLD offers a powerful, non-parametric approach to quantifying class separability in HAR, balancing flexibility in distribution estimation with the robust, comprehensive comparison provided by KLD. This approach allows for a nuanced assessment of class distinctions in temporally varying signals.

3.1.2 Separability score

Kullback-Leibler Divergence-based Class Separability Metric The class separability metric based on KLD is computed as follows:

1. **Efficient KDE Computation**: To optimize performance, the algorithm first computes the Kernel Density Estimates (KDEs) for each class across all data dimensions. These KDEs are stored in a map or dictionary for efficient retrieval in subsequent calculations.

2. **Creation of a Class Distance Matrix**: The algorithm then constructs a matrix representing the distances between classes by calculating the KLD between the KDEs of each pair of classes, taking all data dimensions into account. Given the asymmetry of KLD (i.e., $D_{\mathrm{KL}}(P \parallel Q) \neq D_{\mathrm{KL}}(Q \parallel P)$), the divergence is calculated in both directions for each class pair. The two values are then summed to produce a symmetric distance measure, ensuring that the distance from class A to class B is the same as from class B to class A. A matrix result can be seen in Figure 3.4.

3. **Generation of Class Combinations**: After selecting a value for the variable n, which specifies the number of classes to include in each analysis set, the system generates all possible combinations of class labels of size n. This step allows for analysis of separability across different groupings of classes.

4. **Calculation of Scores**: For each combination of classes, the algorithm calculates two scores:

   - the total separability score calculated in Equation 3.2, summing the distance values between every pair of classes within the combination. Higher is better.

   $$S_{\text{total}} = \sum_{i<j} D_{\mathrm{KL}}(P_{c_i} \parallel P_{c_j}) + D_{\mathrm{KL}}(P_{c_j} \parallel P_{c_i}) \tag{3.2}$$

   - the MSE score in Equation 3.3 calculated between the uniform distribution and the normalized score values for each pair. Lower is better.

   $$S_{\text{MSE}} = \frac{1}{N} \sum_{j<i} \left(\tilde{S}_{i,j} - \frac{1}{N} \right)^2 \tag{3.3}$$

   Where N is the number of class pairs in the combination, and $\frac{1}{N}$ represents the ideal value in a uniform distribution. While $\tilde{S}_{i,j}$ is the score of the pair i j normalized with respect to the current combination set.

5. **Ranking the score**: to rank the class combination, should be minimized the weighted difference between the Total score and MSE in Equation 3.4. The subtraction is needed because we want to maximize the first metric and minimize the second one.

   $$S_{\text{rank}} = w_{\text{total}} \cdot S_{\text{total}} - w_{\text{MSE}} \cdot S_{\text{MSE}} \tag{3.4}$$

This approach enables an efficient and comprehensive assessment of class separability by leveraging KLD over KDEs.

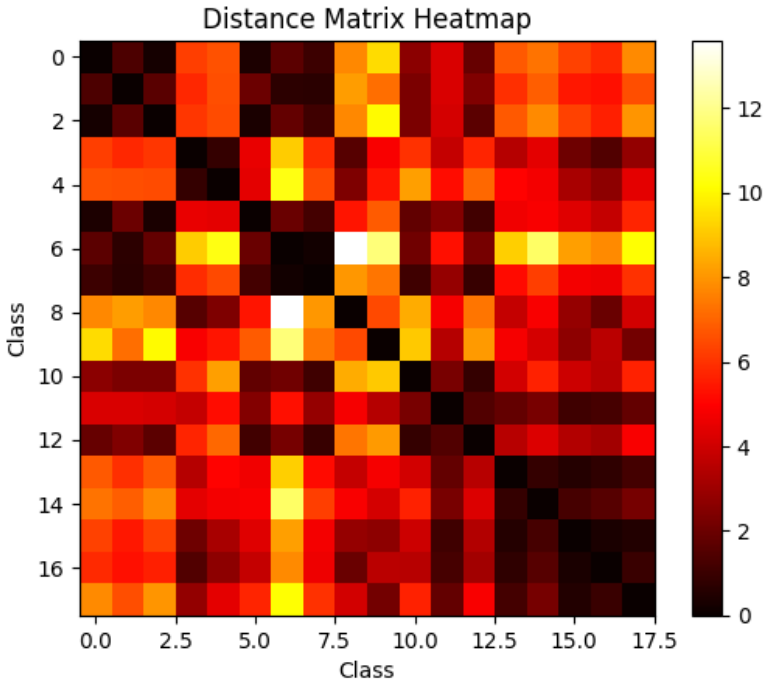


Figure 3.4: Distance matrix calculated with the described process

3.2 Spiking Network Definition

This paragraph will present a detailed account of the network structure and quantization, elucidating the underlying concepts and the actual algorithmic and code implementation. The selected approach employs the use of *snnTorch* and *Brevitas*. 3.5 depicts the network in its entirety, offering a visual representation that can be referenced while following the step-by-step description.

3.2.1 Encoding Layer

Given the spiking nature of the network and the constraints of the Loihi 2 chip as the deployment platform, there are several critical requirements that must be addressed during the design of the input encoding layer:

1. **R1** The input data must be converted into the spiking domain, as SNNs process information exclusively through discrete spike events.

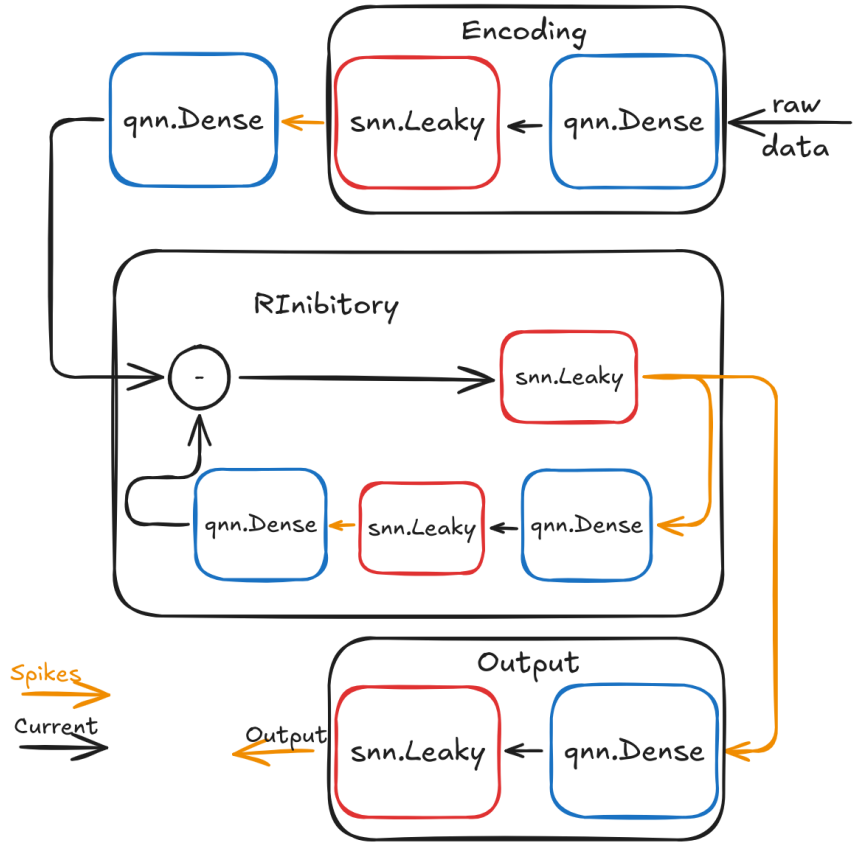


Figure 3.5: the full network designed for this work. From the top, we can recognize: *Encoding layer*, responsible for the spiking encoding of the encoded data. *RInibitory populations*: responsible for the feature extraction from incoming data, using two neuron populations. *Output population*: responsible for the output Rate coding.

2. **R2** Loihi 2 imposes strict hardware limitations, allowing computations only in fixed-point arithmetic. Consequently, all input data must be converted to meet this requirement.

3. **R3** Achieving a task-agnostic encoding pipeline is highly desirable to ensure the portability of the network across different applications without the need for extensive retuning.

Traditional static encoding methods, such as Sigma-Delta encoding, frequency-based encoding, or time encoding, were evaluated. While these methods successfully convert input signals into spike-based representations, they fail to satisfy all the outlined requirements. Specifically, these methods are highly task-dependent, requiring significant manual tuning to adapt to different use cases as can be seen in [20]. This lack of flexibility makes them unsuitable for scenarios where portability

across tasks and domains is a priority.

To address these challenges, an *encoding layer* based on a *LIF population* is proposed as the optimal solution. This method meets all three requirements:

- **Task-Agnostic Encoding (R3)**: Unlike static methods, the LIF population is trainable alongside the rest of the network. This enables the encoding layer to automatically adapt to different tasks, optimizing its performance in a task-specific manner without requiring manual adjustments. The trainability ensures that the encoding layer can generalize across a wide range of applications while maintaining high efficiency.
- **Direct Spike-Based Encoding (R1)**: The LIF population directly converts continuous-valued input signals into discrete spike trains. This ensures seamless integration with the SNN architecture, preserving the event-driven nature of the network and aligning with the requirements of spiking computation.
- **Compatibility with Loihi 2 (R2)**: Although Loihi 2 only supports fixed-point arithmetic for computation, the platform is typically coupled with a conventional CPU for data preprocessing and I/O operations. The encoding layer, implemented as a population of LIF neurons, can leverage this CPU for input data conversion prior to deployment on Loihi 2, ensuring compatibility with the chip's constraints.

Structure of the Encoding Layer The encoding layer consists of two main components:

- A **trainable connection**: This layer learns to project the input data into a form that is optimal for the LIF population to process. The weights of this connection are adjusted during training to ensure the encoding is tailored to the specific task.
- A **LIF population**: This group of spiking neurons processes the pre-projected input and generates spike trains based on the dynamics of the LIF model. These neurons integrate input over time, leak their membrane potential at a predefined rate, and generate spikes whenever the potential crosses a threshold. This behavior ensures temporal representation of the input data in a spiking format.

The use of a trainable encoding layer allows the network to adapt its input representation dynamically, making it robust and generalizable across diverse applications. This flexibility is particularly advantageous in neuromorphic systems, where diverse tasks and datasets often require encoding methods tailored to specific constraints.

Additionally, by offloading the preprocessing and encoding tasks to a conventional CPU, the encoding layer ensures that the constraints of Loihi 2's fixed-point arithmetic are fully respected. This hybrid approach—combining the computational efficiency of the neuromorphic chip with the flexibility of CPU-based preprocessing—provides a practical and scalable solution for deploying SNNs across real-world tasks.

3.2.2 Recurrent Block

In HAR using (SNNs), temporal signals from activities like walking and running often exhibit periodic patterns due to repetitive body movements, while others are less periodic and produce higher signal magnitudes. To manage such diverse signal characteristics, traditional ANNs replace simple feed-forward layers with recurrent structures, such as Recurrent Neural Networks (RNNs), Gated Recurrent Units (GRUs), and Long Short-Term Memory (LSTM) networks. These recurrent layers are capable of retaining and processing past information along with current inputs, refining predictions, and improving classification accuracy for temporal tasks as can be seen in [41].

Previous efforts to classify HAR signals with feedforward SNNs have produced results that are generally inferior to those of state-of-the-art recurrent neural networks [30]. This discrepancy in performance is largely due to the inherent nature of spiking activations in SNNs. In an SNN, a neuron's membrane potential increases as it receives input, potentially reaching a threshold that triggers a spike. After each spike, the neuron's potential resets to zero, causing it to "forget" prior inputs. While biologically inspired, this reset behavior limits the neuron's ability to retain information over time, which is essential in tasks requiring temporal context, such as HAR.

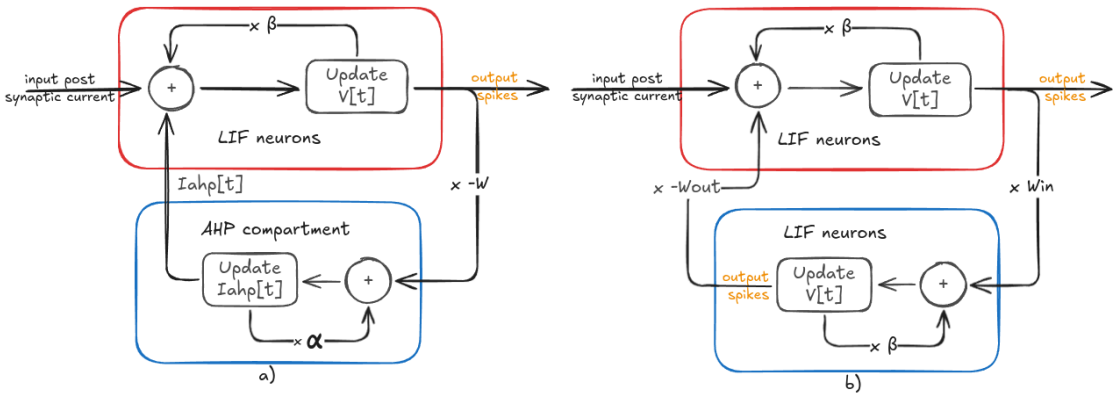


Figure 3.6: a) Original AHP Compartment compared to b) this thesis implementation

To overcome this limitation, this work focuses on developing a recurrent spiking neural network architecture that can effectively process temporal information in HAR. The architecture is inspired by the After-Hyperpolarizing Compartment (AHPC) component introduced in the work of *Philipp Plank et al.* [42]. The AHPC implements a mechanism known in neuroscience as "spike-frequency adaptation" [43]. Specifically, after-hyperpolarizing (AHP) currents slow the readiness of biological neurons to fire again immediately after recent activity, reducing their firing frequency. This adaptation preserves temporal information by enabling neurons to "remember" recent spikes over short periods.

The incorporation of the AHPC component within a recurrent SNN design is particularly useful for HAR, as it allows the network to handle both periodic and non-periodic signal patterns by retaining relevant temporal information across time steps. This biologically inspired mechanism addresses the limitations of traditional feedforward SNNs, equipping the network to perform better in classification tasks involving dynamic signals. These concepts explored in this paper are quite valuable and proved their validity, but was not easy to import this behavior on the neuromorphic chip, given the necessity to extensively test and debug microcode over the actual Loihi2 chip.

Two Implementations of Inhibitory Populations The Figure 3.6 illustrates two different implementations of inhibitory populations within a SNN architecture: (a) the AHPC model, and (b) a simpler inhibitory model, which is used in this work.

1. **AHPC Model (a):** In this implementation, the inhibitory mechanism is modeled through an additional component known as the After-Hyperpolarizing (AHP) compartment. The main excitatory neurons are modeled as (LIF) neurons (shown in the red box), which receive an input post-synaptic current.

   The process is as follows:

   - The LIF population, receives the input, updates the membrane potential, and eventually generates a spike. If the threshold is not passed, the current voltage is multiplied by the decay factor β and added in the next time step.
   - The spike is passed to the following computational block and to the AHP compartment. Here spikes are multiplied by a negative weight, updating the After-Hyperpolarizing current *Iahp*. The current at time t is added at $t+1$ multiplied by decay factor α.
   - The *Iahp*, being a negative value, is then added to the input of the LIF, inhibiting the membrane potential of the neuron.

The AHP compartment effectively implements *"spike-frequency adaptation"*, reducing the likelihood of frequent firing and enabling the neuron to retain information about recent spikes. Output spikes are generated from the LIF neurons based on their updated membrane potential, with an inhibitory feedback term W applied to the output to modulate future spikes.

2. **Simpler Inhibitory Model (b):** In this simpler model, the inhibitory effect is achieved through a direct feedback loop to the main LIF neuron population (in red), using another LIF neuron (in blue).

   The process in this model is as follows:

   - The LIF population, receives the input, updates the membrane potential, and eventually generates a spike. If the threshold is not passed, the current voltage is multiplied by the decay factor β and added in the next time step.
   - Instead of an adaptive *Iahp*, this model implements a simpler feedback mechanism where the output spikes are fed back into the inhibitory LIF through a weight term W_{in} and W_{out}, modulating the input current and membrane potential directly.

   This approach is computationally simpler than the AHPC model, as it lucks the current compartment and allows seamless porting to Lava. However, it provides a basic level of inhibition by adjusting the neuron's input based on recent spikes.

snnTorch implementation The actual implementation of the network was done exploiting the already existing spiking layer defined in *snntorch.snn*. The selected one was *snn.RLeaky*. As shown in Figure 3.7 it implements a neuron population that feeds back the input on itself, passing across a fully connected layer, that can be both a dense connection or a sparse one-to-one connection.

That was modified accordingly to better mimic the needed behavior represented in Figure 3.6.b. The backward branch was expanded to host the three blocks shown in Figure 3.7. One dense block represents the input weight of the following inhibitory population, then an output weight that inhibits the incoming currents.

This method gave us a lot of flexibility and the possibility of updating the structure down the road.

3.2.3 Output Loss Function and Rate Coding

The output of the network is processed through the output block, located at the bottom of Figure 3.5. This block consists of a *qnn.Dense* layer followed by a *(LIF) population*. The *qnn.Dense* layer projects the continuous values generated by the

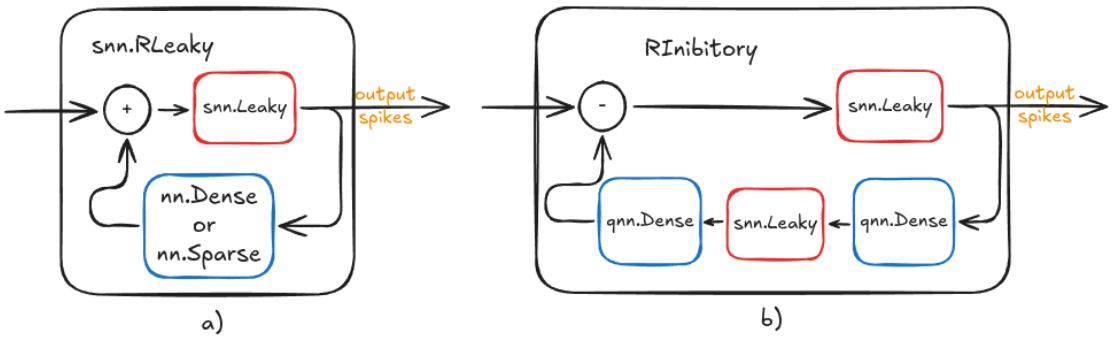


Figure 3.7: a) visualization of *snn.RLeaky* block b) implementation of Rinibitory that Inherit from *snn.RLeaky*

network into a lower-dimensional space suitable for the task at hand. These outputs are then passed to the LIF population, which converts them into spike trains by integrating the input over time and firing when the membrane potential exceeds a threshold. This structure ensures that the final output of the network is encoded in the spiking domain, making it compatible with the rest of the network and the neuromorphic hardware.

To interpret the output spike trains, the network employs *rate coding*, where the firing rate of each output neuron over a specific time window represents the predicted output. The firing rate of an output neuron i is defined in equation 3.5

$$r_i = \frac{\sum_{t=1}^{T} S_i(t)}{T}, \tag{3.5}$$

where $S_i(t)$ is the spike output of neuron i at timestep t , and T is the total number of timesteps.

Cross-Entropy Spike Count Loss, provided by the snnTorch library, which proved to be significantly more effective for our tasks. The Cross-Entropy Spike Count Loss is defined in equation 3.6

$$L_{\text{CE}} = -\sum_{i=1}^{N} y_i \log\left(\frac{r_i}{\sum_{j=1}^{N} r_j}\right), \tag{3.6}$$

where y_i is the one-hot encoded target for class i, r_i is the predicted firing rate for neuron i, and N is the total number of output neurons. This loss treats the firing rates as logits and calculates the probability distribution across classes, penalizing incorrect classifications while encouraging the network to optimize the likelihood of the correct class.

It allowed the network to converge reliably and efficiently by leveraging the probabilistic interpretation of spike rates, which better aligned with the classification objectives of our system. By adopting this loss, we ensured that the network

could learn meaningful patterns while maintaining compatibility with the spiking framework. Additionally, the use of *snnTorch*'s prebuilt loss functions simplified integration, making the training process efficient and deployment-ready for neuromorphic platforms like Loihi 2.

3.2.4 Quantization in Brevitas

Quantization is a critical technique for reducing the precision of numerical representations in neural network models to enable efficient deployment on hardware with limited resources. *Brevitas* [44] serves as a versatile framework for implementing reduced precision data paths during training. It provides a modular and extensible platform that caters to both researchers developing novel quantization-aware training techniques and practitioners applying established methods to their models.

One of *Brevitas*'s key strengths lies in its ability to support a super-set of quantization schemes, unified under a single API. This abstraction simplifies the process of adopting and experimenting with various quantization approaches across different frameworks and compilers. Furthermore, for specific combinations of layers and quantization types, *Brevitas* facilitates inference acceleration by enabling seamless export to *ONNX Runtime*, or *PyTorch's* native quantized operators.

Brevitas has demonstrated its efficacy, mainly These deployments targe custom accelerators, with particular success on *Xilinx FPGAs* [45]. The framework predominantly focuses on affine quantization, emphasizing uniform quantization schemes. However, it does not currently provide out-of-the-box support for non-uniform quantization.

By bridging the gap between research innovation and practical application, *Brevitas* stands as a powerful tool for advancing quantization techniques and optimizing neural networks for resource-constrained environments.

Brevitas provides a powerful framework for implementing quantized neural networks, with key components such as *qnn.QuantLinear* and the *brevitas.quant* type *Int8WeightPerTensorFixedPoint*.

The *qnn.QuantLinear* module serves as a replacement for PyTorch's *nn.Linear*, providing support for quantization. This layer allows weights to be quantized during both training and inference, thereby closely mimicking hardware behavior. The quantization type for the weights can be specified using the *weight_quant* parameter, and an optional bias term can also be included.

The *Int8WeightPerTensorFixedPoint* type in Brevitas represents weights using 8-bit signed integers with per-tensor affine quantization. This quantization method applies a single scale and zero-point to all weights in a tensor, simplifying implementation while maintaining compatibility with hardware accelerators. The process involves linear scaling of weights using a quantization scale and zero-point, enabling efficient computation and reduced memory usage.

An example usage in combination with *qnn.QuantLinear* is:

```
from brevitas.nn import QuantLinear
from brevitas.quant import Int8WeightPerTensorFixedPoint

quant_linear = QuantLinear(
in_features=256,
out_features=128,
weight_quant=Int8WeightPerTensorFixedPoint,
bias=False
)

import torch
x = torch.randn(1, 256)
output = quant_linear(x)
```

One notable feature of *qnn.QuantLinear* is its ability to share the same *weight_quant* object across multiple quantized layers. This capability ensures that different layers can operate with a consistent quantization range, which is particularly advantageous when deploying models on hardware systems that require uniform quantization parameters across layers.

For example, a quantized linear layer can be defined as:

```
from brevitas.nn import QuantLinear
from brevitas.quant import Int8WeightPerTensorFixedPoint

Use the shared weight quantizer in multiple layers

quant_linear1 = QuantLinear(
in_features=128,
out_features=64,
weight_quant=Int8WeightPerTensorFixedPoint,
bias=True
)

quant_linear2 = QuantLinear(
in_features=64,
out_features=32,
weight_quant=quant_linear1.weight_quant,
bias=True
)
```

By leveraging these components, Brevitas provides a seamless platform for creating and optimizing quantized neural networks, enabling both research and practical deployment on hardware-constrained environments.

3.2.5 Cosine Annealing Learning Rate and its Role in Quantized Networks

Cosine Annealing Learning Rate scheduling has emerged as a powerful technique to improve the training of Quantized Spiking Neural Networks (QSNNs) and is being explored by *Jason K. E. et al.* in [46]. In quantized networks, the loss landscape is often plagued by flat regions, local minima, or sharp transitions caused by non-differentiable operations like weight quantization and hard thresholding. Such conditions make convergence to an optimal solution challenging. Cosine annealing addresses this by periodically modulating the learning rate, helping the network escape suboptimal solutions, and exploring uncharted regions of the solution space.

The cosine annealing schedule modulates the learning rate η_t over training iterations t using the equation 3.7:

$$\eta_t = \frac{1}{2}\eta\left(1 + \cos\left(\frac{\pi t}{T}\right)\right), \tag{3.7}$$

where η is the initial learning rate and T is the period of the schedule. This periodic reset introduces learning rate peaks at regular intervals, effectively providing the optimizer with momentum to jump out of flat regions or local minima.

In the context of quantized networks, cosine annealing offers several benefits. Firstly, by periodically increasing the learning rate, the network is less likely to get stuck in suboptimal barriers or flat regions of the loss landscape. This is particularly crucial for quantized models, where the reduced resolution of parameters exacerbates the risk of such occurrences. Secondly, the periodic resets allow the model to revisit earlier solution paths with updated conditions, potentially improving convergence to a more optimal state.

Empirical evaluations on datasets like MNIST, FashionMNIST, and DVS128 Gesture in [46] demonstrate that cosine annealing not only reduces the performance degradation caused by quantization but also achieves more consistent results across trials compared to alternative schedules like step decay or loss-dependent LR adjustments. The robustness of this schedule lies in its simplicity and ability to explore a wider solution space without introducing complex hyperparameter dependencies. Moreover, the schedule is well-suited for QSNNs, as it complements the inherently noisy gradient feedback caused by quantization and spiking approximations, helping to mitigate their adverse effects on training.

The ability of cosine annealing to balance exploration and exploitation in the optimization process makes it a particularly effective tool for training quantized networks, enhancing their robustness and performance in both high-precision and fixed-precision settings.

The cosine annealing schedule was defined in the code in this way:

```
import torch
from torch import lr_scheduler.CosineAnnealingLR as CosineAnnealingLR
```

```
optimizer = torch.optim.Adam(net.parameters(),
                             lr=params['lr'],
                             betas=(0.9, 0.999))

scheduler = CosineAnnealingLR(optimizer,
                              T_max=4690,
                              eta_min=0,
                              last_epoch=-1)
```

3.2.6 Sparsity Enforcing

One of the greatest advantages of SNNs running on neuromorphic hardware, such as Intel's Loihi 2, is their high computational efficiency combined with extremely low power consumption. This efficiency arises from the asynchronous, event-driven nature of SNNs, where only discrete spikes are exchanged over time, significantly reducing energy requirements compared to traditional, continuously-activated neural networks.

However, achieving these power savings in practice requires strategies to enforce sparsity in the network's activity. Sparse spiking activity ensures that neurons only fire when necessary, thereby conserving energy. In neuromorphic hardware, the power consumption is directly related to spiking frequency [47]: the lower the average firing rate across the network, the less energy is consumed. Consequently, by controlling and limiting the frequency of spikes, we can minimize power usage without sacrificing computational capacity.

To enforce sparsity, two strategies can be implemented:

- **Dynamic Threshold** [48]: By setting dynamic firing thresholds, This model adapts thresholds based on the neuron's energy and temporal dynamics neurons are less likely to spike, thereby reducing spiking frequency across the network. This adjustment can lead to substantial power savings.
- **Activity Regularization**[47]: Regularizing the spiking activity during training, such as by penalizing high firing rates in the loss function, encourages the network to learn sparse representations that naturally translate to energy efficiency during inference. This was the selected strategy applied to our neuron network.

Activity regularization was the selected method to enforce sparsity and consequently reduce the network power consumption. There are multiple loss types that aim to regularize the neuron's activity, here there are some examples:

- **Firing Rate Loss** penalizes deviations from a target firing rate, encouraging neurons to maintain a desired activity level. This loss function is formulated

as equation 3.8.

$$\text{Firing Rate Loss} = \lambda \cdot \sum_{i=1}^{N} |\text{firing rate}_i - \text{target rate}| \tag{3.8}$$

where λ is a regularization parameter, N is the total number of neurons, and firing rate_i represents the average firing rate of neuron i. This method is simple to implement and highly effective in promoting sparse activity across the network. However, if the target firing rate is set too strictly, it can overly constrain neuron flexibility, potentially reducing the network's responsiveness to input.

- **Spike Count Loss** controls the overall spike activity by adding a penalty based on the total spike count over a specified time window. This method is particularly effective for maintaining a global constraint on activity, formulated as equation 3.9.

$$\text{Spike Count Loss} = \lambda \cdot \sum_{i=1}^{N} (\text{spike count}_i - \text{target count})^2, \tag{3.9}$$

where spike count_i denotes the total spikes generated by neuron i within the time window. Spike count loss helps in significantly reduce energy usage, making it advantageous for power-efficient applications. However, this approach may not enforce sparsity uniformly across all neurons, resulting in potential imbalances in neuron activity.

- **L1 Regularization on Membrane Potential** [47] penalizes high membrane potential values, indirectly limiting spiking frequency by making it more challenging for neurons to reach their firing threshold. The loss function is expressed as equation 3.10.

$$\text{L1 Membrane Loss} = \lambda \cdot \sum_{i=1}^{N} |\text{membrane potential}_i|, \tag{3.10}$$

where $\text{membrane potential}_i$ represents the potential of neuron i. By enforcing a constraint on membrane potential magnitude, L1 regularization promotes sparsity without directly impacting spike counts. However, it may reduce network expressiveness if neurons become too inhibited, limiting their ability to react to input changes.

- **L2 Regularization on Membrane Potential** [47], similar to L1 regularization, applies a constraint on membrane potential but uses the squared value for a smoother penalization, as can be seen in equation 3.11.

$$\text{L2 Membrane Loss} = \lambda \cdot \sum_{i=1}^{N} (\text{membrane potential}_i)^2, \tag{3.11}$$

L2 regularization provides a softer constraint, allowing more flexibility in neuron activation compared to L1.

- **Target Firing Rate Loss** *Kinjal Patel et al.* in [49] describe a loss that enforces a desired average firing rate for each neuron by penalizing deviations from a predefined target rate range. Instead of enforcing a fixed target firing rate, this regularization method allows neurons to fire within a specified range, defined by a minimum $F_{\min}$ and maximum $F_{\max}$ firing rate. The loss function is formulated to penalize neurons that fall outside this range. Using a rank-based statistical approach, this method regularizes the p-th percentile of firing rates, making it more robust to outliers and variations across samples. The firing rate regularization loss is defined as equation 3.12

$$L_{FR} = \frac{1}{L_j} \sum_{i=1}^{L_j} \left(\left[F_{\min} - R_{i,p}^{j} \right]^{+} + \left[R_{i,p}^{j} - F_{\max} \right]^{+} \right)^2, \tag{3.12}$$

  where L_j is the number of neurons in layer j, $R_{i,p}^{j}$ represents the p-th percentile firing rate of neuron i in layer j across a batch of inputs, and $[\cdot]^{+}$ denotes the ramp function, which is zero if the expression inside is negative and the expression itself if positive. This function imposes a squared penalty if the neuron's firing rate falls below $F_{\min}$ or exceeds $F_{\max}$, encouraging firing rates within the acceptable range.

  In practice, setting $p = 99\%$ for the percentile value allows the regularization to be robust to outliers, making it applicable across various inputs.

Most of the losses listed above, force the network with hard constraints, limiting the population dynamics. The only one that mitigates this behavior is the **Target Firing Rate Loss** because it effectively balances neuron activity, enhancing network stability and reducing energy consumption by preventing excessive or too low spiking. The main idea behind this loss is that if we think of time as our encoding range, forcing the neurons to the same fixed frequency it actually forces the encoding in the same range, killing all the dynamics inside the network and losing details in the codification of the signals. This loss contrary to all the other ones gives the networks less strict bandwidth borders, leaving space for neuron frequency characterization other than trying to regularize the overall spiking activity.

Input: Spike count arrays for each layer, *spike_count_array*, with dimensions $[time, batch, channels]$
Output: Regularization loss, *loss*
Initialize $loss \leftarrow 0$;
foreach *layer i in spike_count_array* **do**
  Initialize $layer\_loss \leftarrow 0$;
  Compute the firing frequencies for each channel
  $frequency\_matrix \leftarrow \frac{\sum \text{spike\_count\_array}[i]}{\text{time\_window}}$;
  $frequency\_matrix \leftarrow \text{sort}(frequency\_matrix)$;
  $R_{\text{pth}} \leftarrow \text{frequency\_matrix}[p_{th} \cdot \text{len}(frequency\_matrix)]$;
  foreach *channel j in spike_count_array*$[i]$ **do**
    $layer\_loss \leftarrow layer\_loss + (\max(0, R_{\text{pth},j} - max\_hz) + \max(0, min\_hz - R_{\text{pth},j}))^2$;
  end
  $loss \leftarrow loss + \frac{layer\_loss}{\text{batch\_size}}$;
  return *loss*
end

Algorithm 1: Firing Rate Regularization Loss Calculation

3.3 Lava and Loihi 2 Porting

The design choices made during the development of the network were heavily influenced by the runtime environment of the Loihi 2 chip. Each decision was validated to ensure compatibility with Intel's *Lava* framework, to simplify the porting process between *snnTorch* and *Lava*. This iterative process required careful evaluation of the computational blocks used in the network, focusing on two critical questions for each component:

- Does the selected computational block exist in *Lava*?
  - If not, the block was discarded for now, awaiting potential future implementation in *Lava*.
  - If yes, does it function as intended in *snnTorch*, with matching parameters and behavior?
- Does the network maintain equivalent behavior between *snnTorch* and *Lava*?

To answer these questions, extensive experimentation was conducted to evaluate both functionality and compatibility. This led to the development of the current network architecture, optimized for both frameworks.

3.3.1 Network Definition in Lava

Each computational block available in *snnTorch* was assessed for both functionality and availability within Lava. The building blocks for the final network were carefully selected to ensure compatibility while maintaining simplicity and versatility. As a result, the network primarily relies on CUrrent BAse Leaky Integrate-and-Fire (CUBA LIF) and **Dense connections**, which offer both flexibility and compatibility with Lava.

CUBA LIF Neurons The chosen neuron model, the CUBA LIF neuron, extends the standard LIF model by introducing a second-order filter. This adds a second decay parameter to the neuron current in addition to the membrane potential decay. While this model is highly versatile, a few constraints were identified during its implementation in Lava.

First, the Lava framework enforces a hard reset of the neuron membrane potential to zero after a spike. In contrast, *snnTorch* offers flexibility by subtracting the threshold from the current membrane potential (default behavior) or resetting it to zero. The Lava behavior was adopted for consistency across platforms.

Second, Lava emits spikes simultaneously with the membrane reset, whereas *snnTorch* provides an option to emit spikes either at the reset timestep or at the subsequent timestep. This difference required adjustments to ensure that the network behavior in *snnTorch* was consistent with Lava's constraints.

Dense Connections Dense connections are essential in *Lava* for connecting different neuron populations. While these connections pose no issues during simulation, the Loihi 2 hardware configuration raises an error if a neuron population is directly connected without a dense layer. Consequently, the *snnTorch* network was adapted to include dense connections where necessary, such as in the inhibitory population block shown in Figure 3.7.b.

Although these dense layers could be initialized as diagonal matrices to mimic identity connections, their weights were made trainable to enable the network to learn meaningful transformations and improve performance.

One notable constraint in *Lava* is the fixed delay of one timestep for all connections. This is a design choice to prevent deadlocks, as all input reads in *Lava* connections are blocking. While this delay does not affect feedforward networks, it introduces temporal desynchronization in recurrent connections. In *snnTorch*, recurrent connections compute feedback at timestep t, based on the forward spike at t, and feed it back at $t+1$. In Lava, due to the delay, the feedback computed at t is introduced at $t+2$.

This delay was initially perceived as a potential issue, particularly for networks with strict temporal dependencies. However, extensive simulations were conducted to assess the impact of this delay on network performance. These simulations, using

the floating-point execution in *Lava*, demonstrated that the network was robust to the introduced delays, showing negligible accuracy loss even at the decimal level. These results validated the feasibility of proceeding with the Lava implementation.

Data Loader A custom data loader block was implemented to feed input data into the network while adhering to *Lava*'s constraints. The design mimicked the functionality of *PyTorch*'s *DataLoader*, ensuring compatibility and seamless integration with the rest of the framework. This custom block handles data preprocessing and conversion, allowing the network to receive properly formatted inputs for spiking computation.

Output Manager To handle the spiking output of the network, an output manager block was introduced. This block processes the spike-based outputs to compute the final predicted labels and aligns them with the ground truth. This alignment facilitates accurate calculation of performance metrics and statistical evaluation outside the Lava runtime environment. The output manager ensures that predictions are correctly interpreted for downstream analysis and performance reporting.

3.3.2 Neuron Internal Variable Quantization

To successfully port the network to neuromorphic hardware, an intermediate step is required to quantize the internal state variables of neurons, passing through the Lava framework. Direct training with quantized variables is not feasible in the current workflow, necessitating this intermediate step for compatibility. Lava provides a seamless conversion pipeline within its simulator to facilitate this process.

The conversion process is highly transparent and user-friendly. At a high level, the API requires only the trained network with floating-point values for both weights and internal neuron variables. After running the network through the conversion tool, it outputs all the converted variables in a fixed-point format, correctly prepared for hardware deployment.

Internally, the conversion pipeline operates by simulating the network and recording the execution ranges of each variable during floating-point operation. These ranges are then discretized and used to determine the appropriate quantized representation for each variable, interpolating values as necessary. This ensures that the quantized variables closely approximate the behavior of the original floating-point network within the constraints of fixed-point arithmetic.

However, this quantization process imposes limitations on the operational range of the network. As variables are discretized, saturation can occur at the extremes of the range, a behavior inherent to the fixed-point nature of computations on neuromorphic hardware. While this introduces some constraints, it is an unavoidable aspect of hardware execution. Careful validation and tuning during the quantization phase can minimize the impact of these limitations on overall network performance.

3.3.3 Hardware Execution

Once the neuron variables and weights were quantized using the *Lava* framework, the network was configured for execution on the Loihi 2 hardware. Since the major issues were addressed during the simulation and quantization steps, the final deployment required minimal additional adjustments.

As per the system design, data loading and the execution of the first encoding block were handled by the CPU. The spiking components, including the recurrent inhibitory block and the output layer, were deployed directly onto the Loihi 2 chip. This hybrid approach leveraged the strengths of both computational domains: the flexibility of the CPU for preprocessing and the efficiency of the neuromorphic chip for spike-based processing.

To allow communication between the CPU and the Loihi 2 chip, specialized I/O blocks were integrated, as required by the Lava framework. These blocks ensure efficient and seamless data transfer between the two processing units, enabling synchronized operation of the hybrid system.

Performance evaluation was carried out using the benchmarking APIs provided by the *Lava* framework [38]. These tools allowed for detailed analysis of metrics such as power consumption, latency, and throughput. The results validated the network's performance and its efficient execution on the Loihi 2 chip, demonstrating that the hybrid configuration successfully met the design objectives.

3.4 Hyperparameter Search

Training SNNs is inherently challenging due to its event-driven and temporal dynamics, which make traditional optimization techniques less effective. Optimizing these networks for execution on neuromorphic hardware, such as Loihi 2, introduces further complexity, as the hardware imposes constraints on numerical precision, memory, and compute operations. HPO is crucial to determining the optimal configuration of trainable and structural parameters, providing the network with the best starting point for its training process.

The sheer number of hyperparameters, including learning rates, decay constants, spike thresholds, and weight initializations, makes it impractical to evaluate all possible combinations within their respective operational ranges manually. This complexity is further compounded by the interactions between hyperparameters, where a suboptimal choice in one parameter can severely affect the performance of the entire network. Table 3.1 illustrates the number of hyperparameters in each search space that must be selected for the network, highlighting the impracticality of manual tuning. Without a systematic and efficient tool to guide the selection of hyperparameters, finding an optimal solution would be nearly impossible.

Parameter	Type	Search Space
batch_size	choice	{128, 256, 512, 1024}
slope	quniform	[Lower: 5, Upper: 50, Step: 5]
lr	choice	{0.01, 0.005, 0.002, 0.001, 0.0005, 0.0002, 0.0001, 0.00005, 0.00002, 0.00001}
net_hidden_1	quniform	[Lower: 20, Upper: 200, Step: 20]
net_hidden_2	quniform	[Lower: 100, Upper: 600, Step: 50]
vth_in	quniform	[Lower: 0.1, Upper: 2, Step: 0.1]
vth_recurrent	quniform	[Lower: 0.1, Upper: 2, Step: 0.1]
vth_out	quniform	[Lower: 0.1, Upper: 2, Step: 0.1]
vth_back	quniform	[Lower: 0.1, Upper: 2, Step: 0.1]
beta_in	quniform	[Lower: 0.1, Upper: 1, Step: 0.1]
beta_recurrent	quniform	[Lower: 0.1, Upper: 1, Step: 0.1]
beta_back	quniform	[Lower: 0.1, Upper: 1, Step: 0.1]
beta_out	quniform	[Lower: 0.1, Upper: 1, Step: 0.1]
drop_recurrent	quniform	[Lower: 0.1, Upper: 0.5, Step: 0.05]
drop_back	quniform	[Lower: 0.1, Upper: 0.5, Step: 0.05]
drop_out	quniform	[Lower: 0.1, Upper: 0.5, Step: 0.05]

Table 3.1: Parameter Search Spaces

3.4.1 Neural Network Intelligence (NNI)

To address the challenges of hyperparameter optimization, we leveraged a tool called NNI. This tool provides a robust and extensible framework for automating hyperparameter search and optimization, enabling a systematic exploration of the parameter space while reducing the time and effort required for manual tuning.

Hyperparameter Optimization (HPO): The goal of HPO is to identify the combination of hyperparameters that results in the best performance for a given task. This is achieved by defining an objective function—such as validation accuracy, loss minimization, or energy efficiency—and systematically evaluating different parameter configurations to maximize or minimize this objective. HPO is especially critical for SNNs, as their training dynamics are sensitive to parameters such as:

- **Learning rate**, which controls the step size in weight updates.
- **Spike threshold**, which determines when a neuron fires.
- **Decay constants** for membrane potentials in LIF neurons.
- **batch size** affects memory, training time and generalization.
- **neuron population sizes** affects how many neurons will be assigned per population.

- **slope** affects the surrogate gradient slope.
- **dropout** the probability of each neuron to be shut down for each epoch step.

Traditional grid search or random search methods are often computationally expensive and fail to explore the parameter space efficiently, particularly for high-dimensional hyperparameter configurations.

NNI Framework for HPO: NNI streamlines this process by providing a flexible and scalable environment for hyperparameter optimization. Users define a search space for each hyperparameter and specify the optimization algorithm to guide the search. NNI supports a wide range of optimization methods, including:

- **Grid Search:** Systematically evaluates all combinations in the search space.
- **Random Search:** Randomly sample configurations, offering simplicity and baseline performance.
- **Bayesian Optimization:** Constructs a probabilistic model of the objective function to guide exploration, focusing on regions likely to yield high performance.
- **Tree-structured Parzen Estimator (TPE):** A model-based method that estimates distributions of promising hyperparameter configurations.
- **Evolutionary Algorithms:** Simulates natural selection processes to evolve hyperparameter configurations.
- **Anneal:** A simple annealing-based heuristic that begins by sampling randomly from the entire search space and gradually focuses on regions closer to the best solutions observed. It leverages smoothness in the response surface, but its annealing rate is fixed and not adaptive. It is lightweight and effective for simpler search tasks.

Application to SNNs: For the training of SNNs, NNI allowed us to define a search space encompassing key hyperparameters such as learning rate, spike thresholds, and decay constants. By automating the search process, NNI enabled efficient exploration of this space, identifying configurations that maximized the network's accuracy and robustness while maintaining compatibility with the hardware constraints of Loihi 2.

During the search, NNI efficiently distributed trials across available computational resources, ensuring rapid evaluation of candidate configurations. Among the various optimization algorithms provided by NNI, we selected **Anneal** as our optimizer due to its demonstrated ability in previous works to converge to optimal solutions faster than alternative methods. This made it particularly suitable for scenarios with constrained computational resources or time-sensitive tasks.

The use of NNI, combined with the Anneal optimizer, significantly improved the training process, reducing the time and computational resources required for hyperparameter tuning. The optimized configurations discovered through this process led to better convergence, higher accuracy, and improved energy efficiency compared to manually selected hyperparameters. These results highlight the importance of automated HPO tools in developing high-performance SNNs for neuromorphic applications.

Chapter 4

Results

This chapter presents the results obtained from the proposed approach, evaluating each step of the process in Figure 3.1 and assessing their alignment with the expected outcomes. The evaluation begins with an analysis of the data preprocessing pipeline, specifically focusing on how the dataset was split into subsections and the performance metrics achieved for each split.

Next, the HPO process is examined, highlighting the parameter configurations that produced the best results and analyzing the accuracy losses incurred during each porting stage: from *snnTorch* to *Lava* and from floating-point to fixed-point arithmetic.

The performance of each model is then evaluated on the test set, providing a comprehensive view of their effectiveness. Special attention is given to the impact of sparsity-enforcing mechanisms, quantifying the reduction in overall network activity compared to a baseline network without sparsity loss, as measured using the NeuroBench framework [27].

Finally, the chapter concludes with a detailed analysis of power metrics, evaluating the energy efficiency of the proposed network on the Loihi 2 chip, and showcasing the practical benefits of this neuromorphic hardware implementation.

4.1 KDL metrics evaluation

Following the execution of the pipeline described in Section 3.1, the complete distance matrix is presented for analysis in Figure 4.1. This matrix quantifies the relationships between data subsets, providing a clear view of their similarities and differences. It validates the effectiveness of the preprocessing steps, ensuring the dataset splits are appropriately structured for training and evaluation. This distance matrix serves as a foundation for assessing the balance and representativeness of the prepared data.

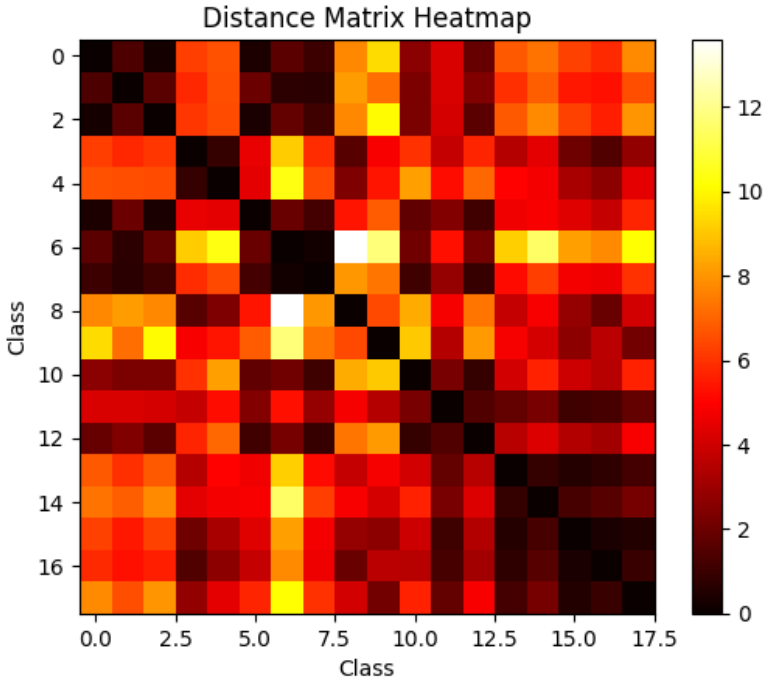


Figure 4.1: Distance matrix calculated with the described process

High score split: This split achieved the highest sum of pairwise distances, indicating the greatest overall class separability among the evaluated subsets. Combinations of selected classes are reported in Table 4.1.

Subset Index	Dataset Index	Activity
0	0	walking
1	2	stairs
2	6	brushing teeth
3	8	eating chips
4	9	eating pasta
5	14	dribbling basketball
6	17	folding clothes

Table 4.1: Activity for High score split

As can be seen in Figure 4.2, most of the contribution that increases the sum score is given by the class in position 2. It has the best separability pair score with all the classes in this split, meanwhile, the are two big clusters of similar classes at the start and the end of the label indexes. This will probably result in a less-than-optimal class separation. This is also represented by the higher than excerpted

MSE score, that underlies this characteristic of this subset.

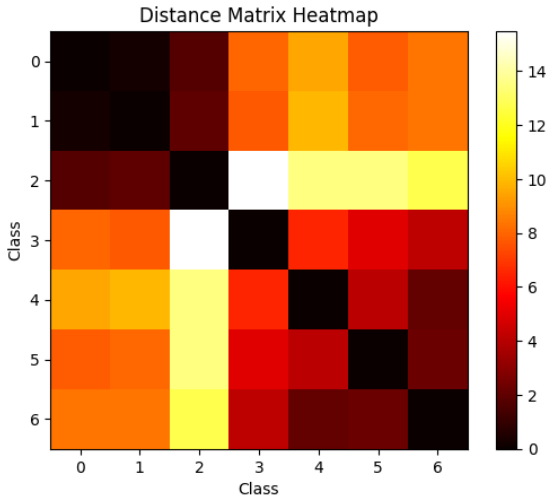


Figure 4.2: **Sum Score:** 150.30 **MSE Score:** $2.380 \cdot 10^{-4}$

Balanced score split: This split represents the optimal balance between a high sum of pairwise distances and the MSE score. The result was achieved by weighting the MSE score with a coefficient of 2, compared to a coefficient of 1 for the sum score, emphasizing the need for more evenly distributed pairwise separability across the dataset. The subset of the classes in this split are listed in Table 4.2.

Subset index	Dataset Index	Activity
0	0	walking
1	1	jogging
2	4	standing
3	8	eating_chips
4	9	eating_pasta
5	10	drinking
6	14	dribbling_basketball

Table 4.2: Activity for balanced score split

In Figure 4.3 can be seen that the sum Class score is pretty high, but is distributed, as expected, more evenly across all the pairs compared to the previous split. This is shown also by the low MSE Score, having as lowest MSE score $1.287 \cdot 10^{-4}$.

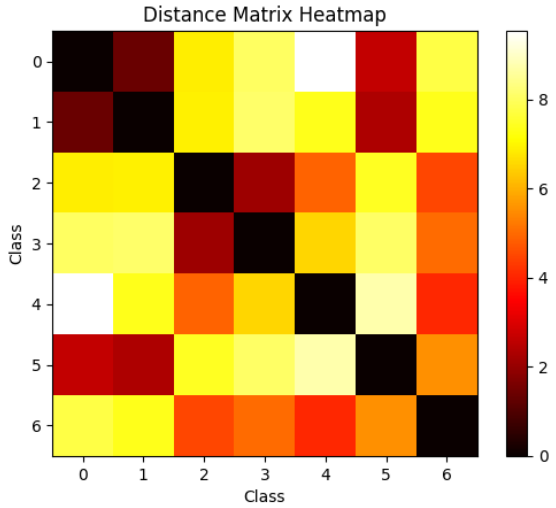


Figure 4.3: **Sum Score:** 125.05 **MSE Score:** $1.436 \cdot 10^{-4}$

Worst split: This split exhibits the lowest sum of pairwise distances, representing the worst-case scenario in terms of class separability. The class list can be seen in Table 4.3

Subset Index	Dataset Index	Activity
0	0	walking
1	1	jogging
2	2	stairs
3	5	typing
4	6	brushing teeth
5	7	eating soup
6	12	kicking soccer

Table 4.3: Activity for worst score split

In Figure 4.4, the class separability score is notably low, making it challenging to distinguish between classes effectively. For this subset, relying on the MSE score does not provide meaningful insights, as it fails to capture the nuances required for accurate evaluation under these conditions.

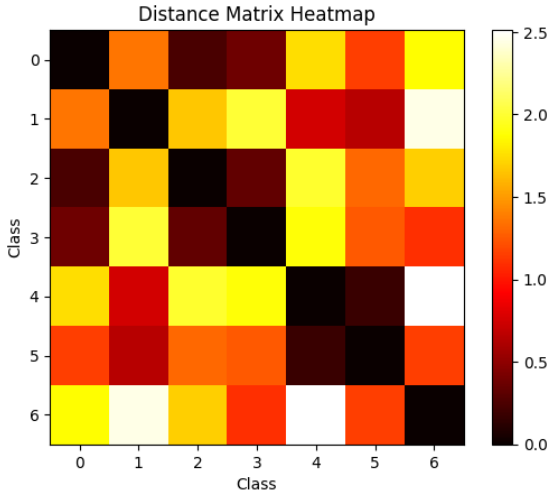


Figure 4.4: **Sum Score:** 27.67 **MSE Score:** $2.011 \cdot 10^{-4}$

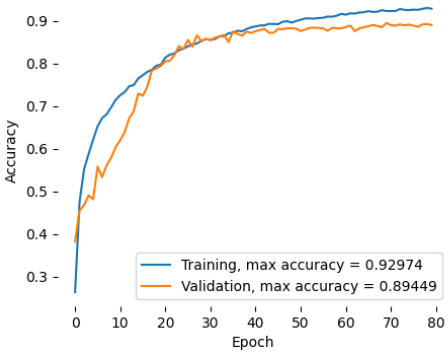
4.2 Training results

This section presents the training results of the proposed SNN in Figure 3.5 and evaluates the effectiveness of the HPO process conducted using the NNI framework. The primary goal of this process was to identify the optimal configuration of hyperparameters to maximize the network's performance while ensuring compatibility with the Loihi 2 hardware constraints. The evaluation begins with an analysis of the hyperparameter search results, focusing on the configurations selected by the Anneal optimizer and their impact on the network's accuracy and convergence during training. The validation set is used to assess the performance of each configuration, ensuring that the chosen parameters contribute to robust and efficient network behavior. Finally, the section concludes by presenting the performance of the optimized networks on the test set, highlighting the generalization capabilities and overall effectiveness of the proposed approach. This comprehensive evaluation underscores the significance of systematic HPO in achieving high-performing SNNs while meeting the requirements of neuromorphic hardware deployment.

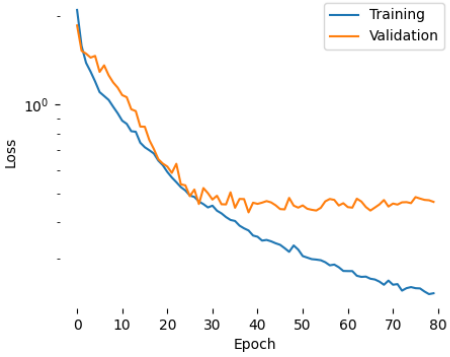
4.2.1 Best results analysis

This section discusses the results of the best experiments conducted for each network configuration. The evaluation includes an analysis of accuracy scores for both training and validation, the loss trends, and the corresponding confusion matrices. The discussion begins with the worst-performing results, derived from the subsets with extreme similarity between classes and high sum scores. It then transitions to an analysis of the best-performing results, achieved with the balanced score subset.

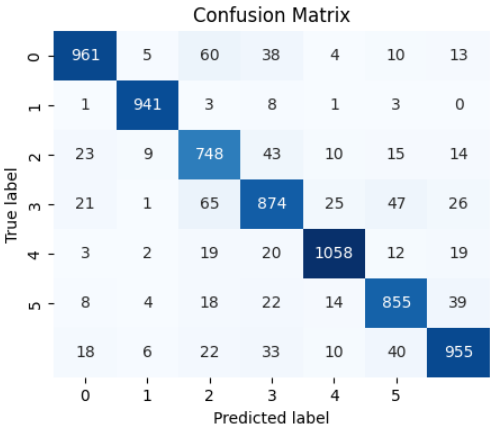
worst split This split was used as a control to evaluate the network's performance on a challenging subset of data, all the meaningful results are shown in Figure 4.5. Achieving satisfactory results on this split provides a reliable indication of the network's overall capabilities, particularly in handling low-separability classes. It serves as a potential lower bound for the accuracy, offering insight into the robustness of the network structure. The network achieved a score of 89.94%, coming remarkably close to the critical threshold of 90%, further validating the effectiveness of the proposed design and process.



(a) Accuracy



(b) Loss



(c) Confusion Matrix

Parameter	Value
batch_size	512
slope	5
lr	0.001
net_hidden_1	140
net_hidden_2	450
vth_in	1.3
vth_recurrent	0.4
vth_out	0.4
vth_back	0.8
beta_in	0.7
beta_recurrent	0.4
beta_back	0.3
beta_out	0.3
drop_recurrent	0.35
drop_back	0.35
drop_out	0.25
Score	**89.449**

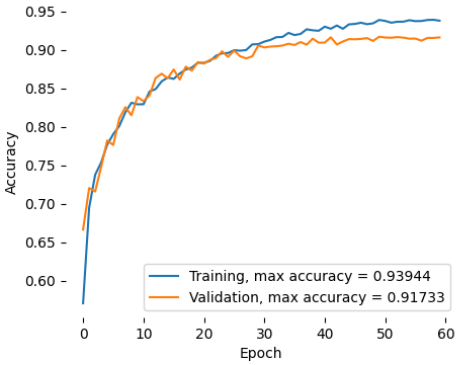
(d) HPO Parameters

Figure 4.5: Evaluation of the worst split.(a) Subfigures show Accuracy , (b) Loss , (c)Confusion Matrix , (d) the parameters chosen by the HPO process .

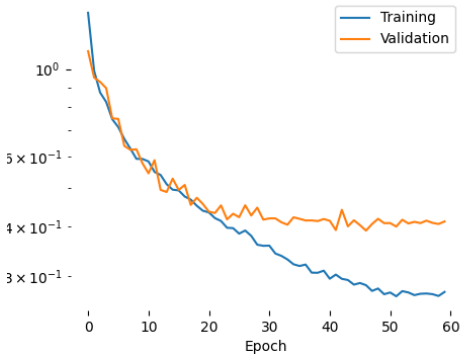
high score split This split was not expected to yield good results due to the poor separability of score clusters, which inherently penalized the network's performance. Despite these challenges, the network configuration achieved good enough results, reaching an accuracy of 91.73% for this experiment. The confusion matrix in Figure 4.6.c highlights that the most frequently misclassified instances belong to the least separable classes, underscoring the impact of low cluster separability on the network's classification accuracy. These observations validate the network's robustness under suboptimal conditions while identifying areas for potential improvement. The accuracy of this split, which is close to the performance of the worst split, definitely stresses the previous sub-optimal evaluation of the sum score as a metric for best separable splits.

balanced split The balanced split represented the most promising candidate for achieving optimal performance, as it was carefully crafted using additional selection policies to enhance class separability and overall representativeness. Compared to other splits, which faced challenges such as poor separability or imbalanced distributions, the result in Figure 4.7 for the balanced split provided a more consistent foundation for training and evaluation reaching the final validation accuracy of 96.508%. Experiments validated these assumptions, with the network achieving the highest accuracy observed across all subsets. This result underscores the effectiveness of the balanced split in leveraging the network's full potential, highlighting its ability to handle class distributions more effectively than splits with inherent cluster separability issues. The superior performance on this subset confirms the robustness of the proposed network structure and optimization process.

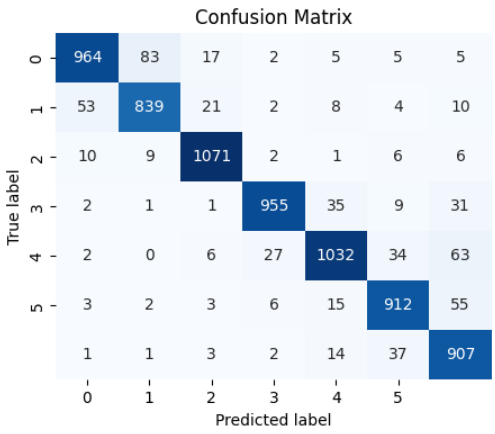
Experiments validated these assumptions, with the network achieving the highest accuracy observed across all subsets. This result underscores the effectiveness of the balanced split in leveraging the network's full potential, highlighting its ability to handle class distributions more effectively than splits with inherent cluster separability issues. The superior performance on this subset confirms the robustness of the proposed network structure and optimization process.



(a) Accuracy



(b) Loss

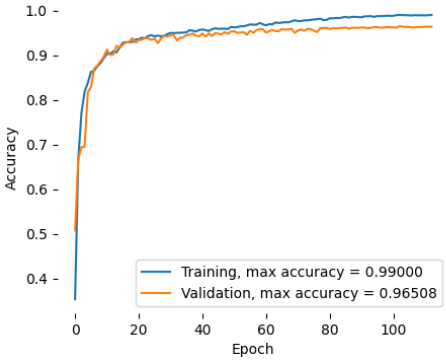


(c) Confusion Matrix

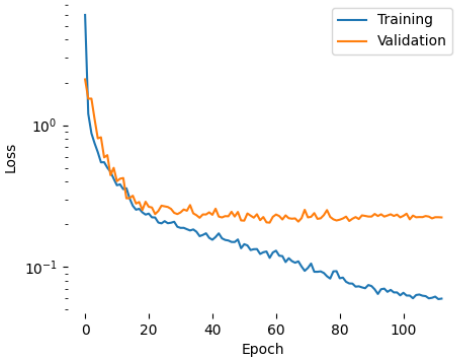
Parameter	Value
batch_size	256
slope	5
lr	0.0005
net_hidden_1	140
net_hidden_2	550
vth_in	1.7
vth_recurrent	1.0
vth_out	0.9
vth_back	0.4
beta_in	0.4
beta_recurrent	0.8
beta_back	0.6
beta_out	0.8
drop_recurrent	0.15
drop_back	0.25
drop_out	0.45
Score	**91.733**

(d) HPO Parameters

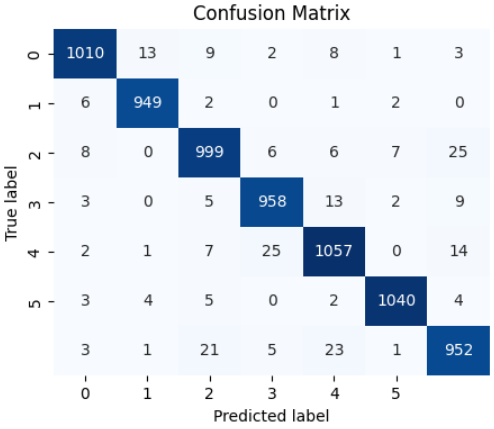
Figure 4.6: Evaluation of the High Score split. Subfigures show Accuracy (a), Loss (b), Confusion Matrix (c), and the parameters chosen by the HPO process (d).



(a) Accuracy



(b) Loss



(c) Confusion Matrix

Parameter	Value
batch_size	512
slope	5
lr	0.005
net_hidden_1	200
net_hidden_2	500
vth_in	1.5
vth_recurrent	0.5
vth_out	0.3
vth_back	1.4
beta_in	0.4
beta_recurrent	0.6
beta_back	0.6
beta_out	0.3
drop_recurrent	0.4
drop_back	0.2
drop_out	0.2
Score	**96.508**

(d) HPO Parameters

Figure 4.7: Evaluation of the balanced score split. Subfigures show Accuracy (a), Loss (b), Confusion Matrix (c), and the parameters chosen by the HPO process (d).

4.3 Activation sparsity results

Activation sparsity is a fundamental feature of SNNs that contributes significantly to their energy efficiency and computational performance. By minimizing the number of active neurons during execution, sparsity reduces the overall computational cost and memory usage while preserving network functionality. In this study, sparsity loss was selectively applied to the network to encourage this behavior. The application of sparsity loss, however, requires careful consideration, particularly for neuron populations involved in feedback mechanisms, as excessive constraints can disrupt the learning process.

The following analysis focuses on the evaluation of activation sparsity and other associated metrics, as calculated using the Neurobench framework. The impact of sparsity loss is assessed by comparing multiple versions of the network, each trained under different conditions, to understand how sparsity constraints influence performance and efficiency.

Sparsity loss was applied to all neuron populations, except for the backward population. This decision was made because the backward population's behavior is directly influenced by the forward population. Adding sparsity constraints to this feedback loop could unnecessarily complicate the learning process, potentially leading to convergence issues during training.

Neurobench provides a range of metrics to analyze the performance and efficiency of SNNs. The following metrics were calculated:

- **footprint**: This metric measures the memory footprint, in bytes, required to represent the model. It accounts for factors such as quantization, parameter storage, and buffering requirements. (Note: It is unclear whether this metric includes quantized weights, which may impact the result).

- **parameter_count**: This represents the total number of parameters in the network, as returned by the `nn.Module.parameters()` method. It provides a direct measure of the model's complexity.

- **activation_sparsity**: This metric calculates the average sparsity of neuron activations during execution. It is averaged over all neurons in all layers, across all timesteps and tested samples. A value of 0 indicates no sparsity (all neurons are always active), while a value of 1 indicates complete sparsity (all neurons are inactive).

- **membrane_updates**: This metric captures the average number of updates made to neuron membrane potentials during execution. It is computed across all neurons, timesteps, and tested samples. This metric is specifically tailored for SNNs implemented with `snnTorch`.

These metrics were evaluated on three different networks to provide a comprehensive comparison (hyperparameters and accuracy results can be seen in Table 4.4):

- **Balanced split**: This model represents the optimized network obtained from the HPO process, incorporating the activation sparsity loss.
- **Balanced split no loss**: This network shares the same hyperparameters as the **Balanced score split**, but the activation sparsity loss was omitted to evaluate the impact of sparsity constraints on network performance.
- **Balanced no loss optimal**: This network was directly optimized without any sparsity constraints applied, representing an idealized case for comparison.

Parameter	**Balanced**	**Balanced no loss**	**Balanced optimal no loss**
batch_size	512	512	256
slope	5	5	5
lr	0.005	0.005	0.002
net_hidden_1	**200**	**200**	**120**
net_hidden_2	**500**	**500**	**250**
vth_in	1.5	1.5	0.7
vth_recurrent	0.5	0.5	1.9
vth_out	0.3	0.3	1.5
vth_back	1.4	1.4	0.9
beta_in	0.4	0.4	0.6
beta_recurrent	0.6	0.6	0.7
beta_back	0.6	0.6	0.7
beta_out	0.3	0.3	0.4
drop_recurrent	0.4	0.4	0.25
drop_back	0.2	0.2	0.4
drop_out	0.2	0.2	0.25
Score	**96.508**	**96.508**	**87.834**

Table 4.4: HPO for each network analyzed. In **bold the main parameters that change model size.**

By comparing these networks using the metrics provided by Neurobench, we can assess the impact of sparsity loss and other design choices on memory efficiency, computational cost, and overall network performance.

Metric	Balanced score split	Balanced score split no loss	Balanced score split no loss optimal
footprint (KB)	590.52	590.52	153.78
parameter_count	604710	604710	157480
activation_sparsity	**0.963**	0.744	0.863
membrane_updates	7779.61	6807.251	3575.9675
accuracy	**96.508**	95.358	87.834

Table 4.5: Comparison of metrics across different configurations.

As can be seen from the results in Table 4.5, comparing networks with the same hyperparameters, the network trained with the activation sparsity loss achieves significantly better accuracy and a superior activation sparsity score. The activation sparsity loss proves to be highly effective, both enhancing the overall performance of the model and optimizing the spiking behavior of the neurons. Reducing unnecessary activity, not only improves the model's efficiency but also boosts its classification accuracy.

On the other hand, the network optimized without the loss demonstrates a lower memory footprint compared to the others. Despite this, it manages to achieve performance levels close to that of the best-performing network trained with sparsity loss, albeit without the same level of activation sparsity.

Overall, these results show that the activation sparsity loss is highly beneficial, effectively reducing activity while improving accuracy. This dual optimization improves the encoding efficiency of spikes, making the network more effective and efficient in handling sparse neural activity.

4.4 Networks conversion results

To achieve deployment on the Loihi 2 hardware, two intermediate steps are necessary:

- Transitioning the network from *snnTorch* to *Lava*.
- Converting the network in Lava from floating-point to fixed-point arithmetic.

Significant effort was dedicated to ensuring that the network behaved consistently across both frameworks, despite differences such as the timestep execution delay between *snnTorch* and *Lava*. This section evaluates the robustness of the network with respect to these fundamental principles, first by examining its behavior during the transition from *snnTorch* to *Lava* and then by assessing the impact of state quantization. The analysis focuses on understanding how the reduced precision and dynamics introduced by fixed-point arithmetic affect the network's overall performance.

4.4.1 From snnTorch to Lava

After obtaining the best results from the *snnTorch* framework, the network was ported to *Lava* for further validation. This validation process involved analyzing accuracy, raster plots, and state dynamics to assess the robustness of the network and the reliability of the porting process, given the inherent execution discrepancies between the two frameworks.

Table 4.6 shows that the accuracy loss during this transition was marginal, with the largest drop observed in the Balanced Score split, where accuracy decreased by only 0.582%. These results highlight the robustness of the network and suggest that the differences in dynamics between *snnTorch* and *Lava* do not critically impact performance for this application.

The minimal accuracy loss observed during this stage was highly encouraging, providing confidence in the pipeline's ability to maintain performance across the entire porting process.

Split name	**snnTorch accuracy**	**Lava accuracy**	**accuracy loss**
High score split	91.733	91.609	0.124
Balanced score split	96.508	95.926	**0,582**
Worst score split	89.449	88.875	0.574

Table 4.6: comparison of validation score for the same network on the two different frameworks using floating point values

Now, we analyze the state plots and raster plots of each population, comparing the dynamic behavior between *snnTorch* and *Lava* for the same class signal (Figure 4.8). The network under consideration is the one trained on the balanced score split, as it experienced the largest accuracy loss during this step.

From the graphs, it is evident that *Lava* introduces a time delay for each block due to its inherent processing characteristics. This delay causes the input signal to propagate through the network at staggered time steps compared to *snnTorch*. For instance, the final step of the input signal is processed in *Lava* at timestep 43, whereas in *snnTorch* it completes at the same timestep, 43. This subtle difference highlights the processing discrepancy introduced by *Lava*'s execution model.

In the subsequent analysis, we will take a detailed look at each population individually, evaluating how this delay affects their state dynamics and spike raster patterns. This comparison provides deeper insights into the robustness of the network under different execution environments.

Encoding layer For the encoding layer graphs in Figure 4.9, the state plots and spike rasters show a high degree of similarity between *snnTorch* and *Lava*. The primary observable difference is the expected time delay introduced by *Lava*, which is not a significant issue at this initial processing stage. This similarity

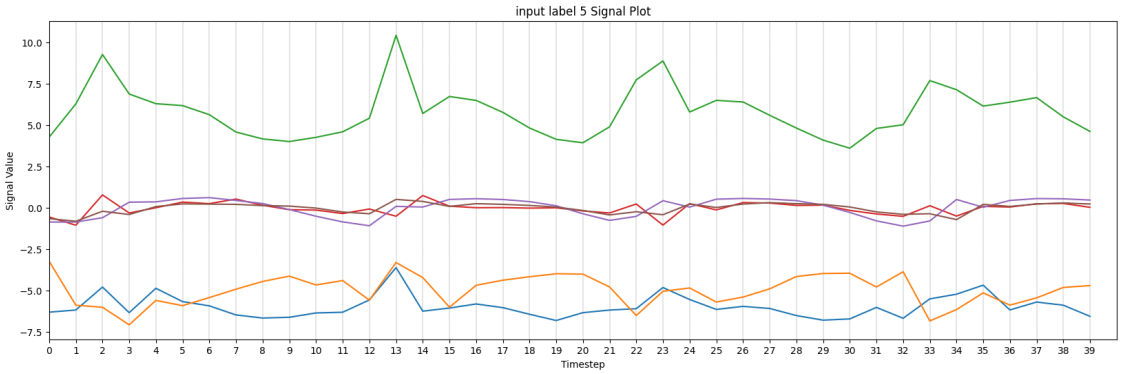


Figure 4.8: Visualization of signal from class 5 (Drinking) of balanced split. It will be the input for the state and raster visualization.

indicates that the encoding layer's functionality and dynamics are preserved across frameworks, ensuring consistent behavior during the input signal transformation.

Forward layer In Figure 4.10, the first noticeable activity discrepancy between the two implementations can be observed, particularly in the raster plots. While the state representations exhibit minimal differences and maintain the same operational range, slight variations in the actual signals are evident. These differences are not drastic but indicate the influence of the time delay introduced by *Lava*, which consequently impacts the inhibitory action of the backward population.

This effect is further highlighted in the raster plots, where the *Lava* implementation shows a slightly higher spike density compared to *snnTorch*. This increased activity aligns with the expected delay in inhibitory feedback, subtly altering the dynamics of the forward population.

Backward layer Similar observations can be made for the backward population in Figure 4.11, but with a greater degree of difference. Both the state dynamics and operational ranges show notable discrepancies between *Lava* and *snnTorch*.

The differences in spiking activity are clearly visible in the raster plots: the backward population in *snnTorch* exhibits fewer spikes compared to *Lava*. This indicates a need for increased inhibition in the *Lava* implementation, as the introduced delay allows the forward population to accumulate more charge in the neuron states before inhibition takes effect. These dynamics underscore the impact of timing discrepancies on the inhibitory feedback mechanism and highlight the importance of accounting for these effects during the porting process.

Output layer Despite the observed differences in the dynamics of the forward and backward populations, the robustness of the network can still be assessed through the behavior of the output population, as shown in Figure 4.12. The

output dynamics largely follow the same patterns across both *snnTorch* and *Lava*, demonstrating the network's resilience to the introduced discrepancies.

The class label prediction remains correct, with firing patterns for the correct class exhibiting only slight differences while still being decisive for accurate class selection. Additionally, the raster plots for both frameworks show that the same incorrect class labels are minimally stimulated, highlighting consistent overall behavior despite minor variations in the spike patterns. This reinforces the conclusion that the network's functionality and classification capability are preserved across the two implementations.

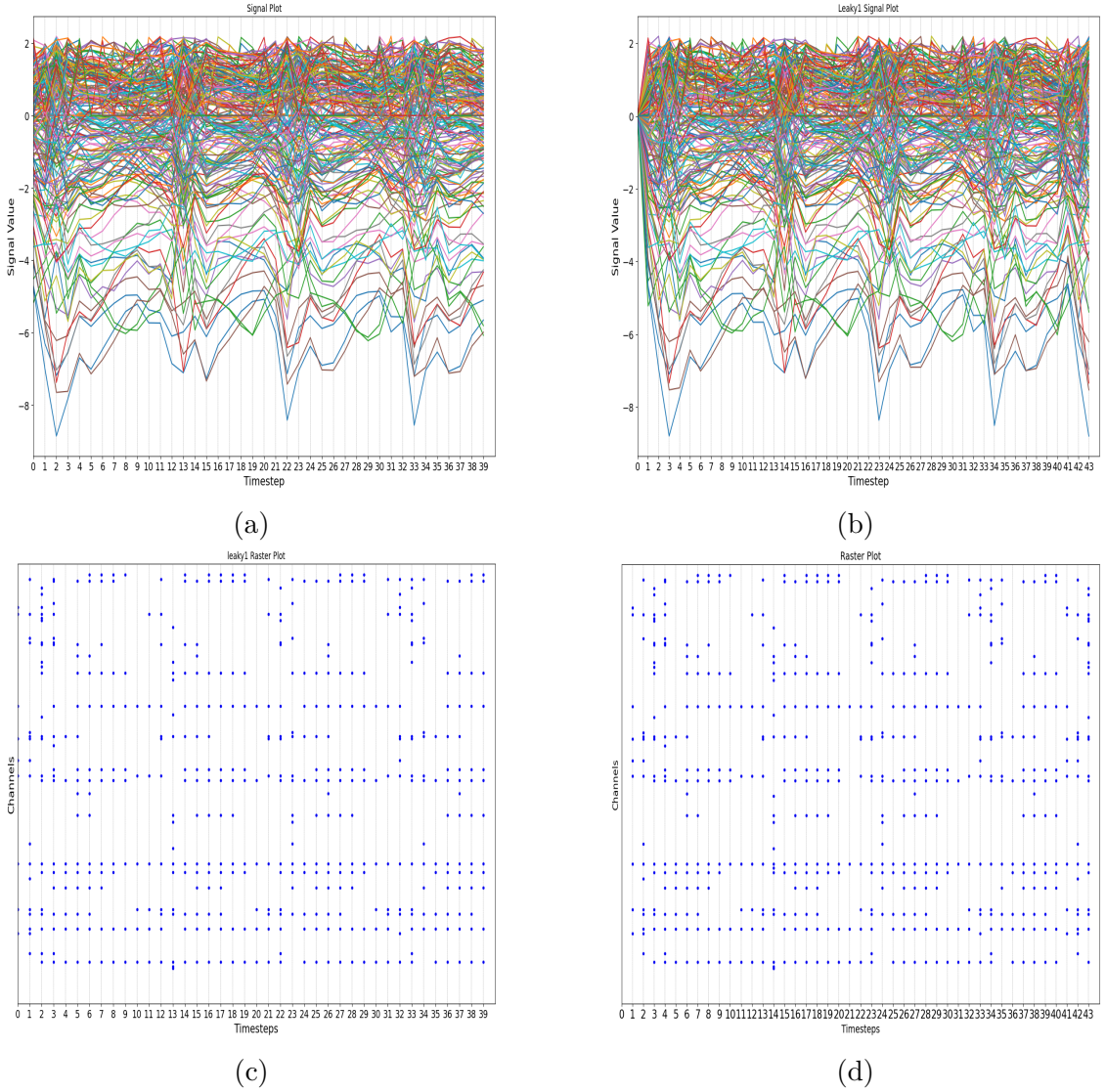


Figure 4.9: Comparison between **encoding population** state and spikes of ((a) and (c)) snnTorch and ((d) and (b)) Lava

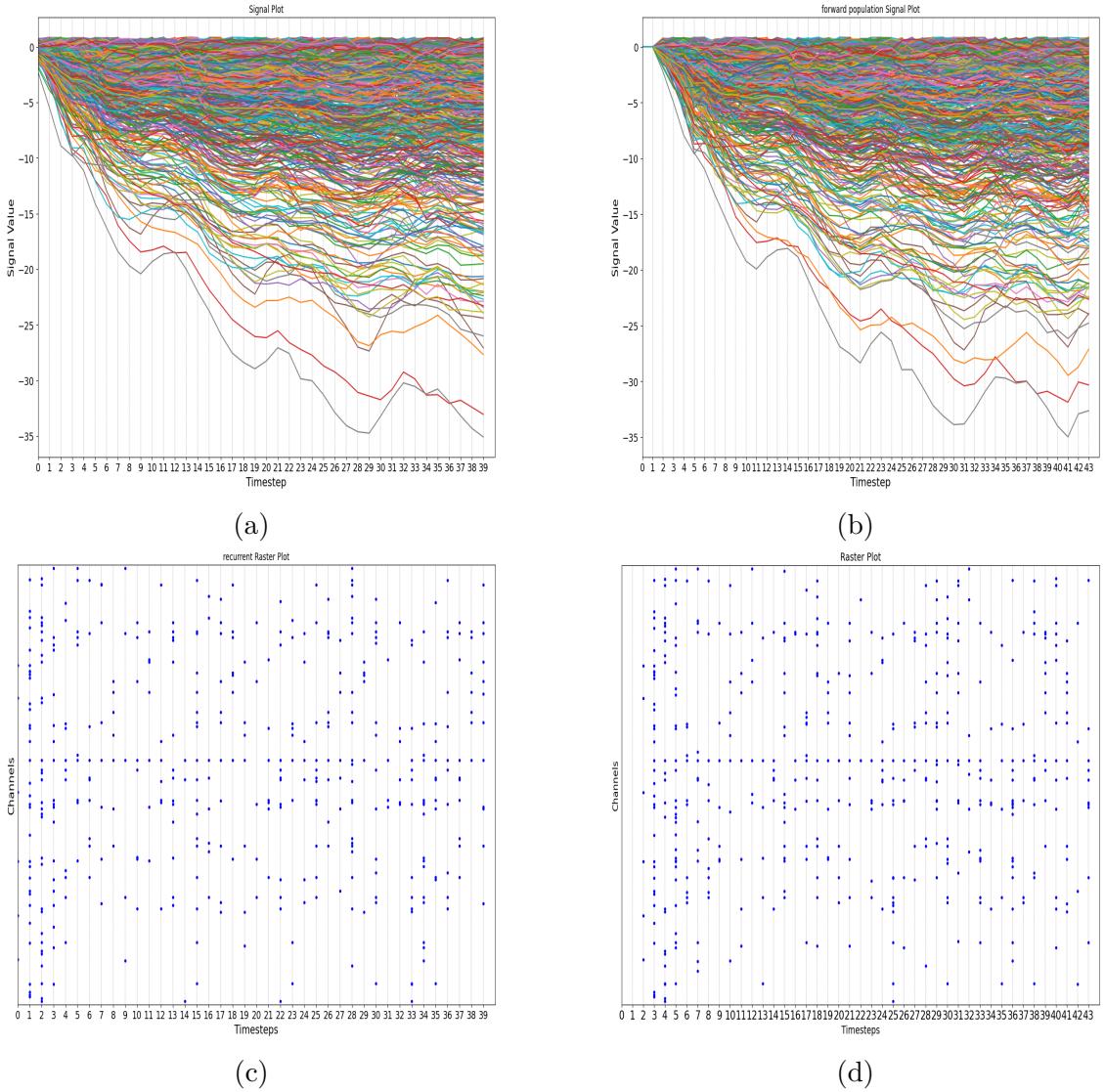


Figure 4.10: Comparison between **forward population** state and spikes of ((a) and (c)) snnTorch and ((d) and (b)) Lava

4.4.2 From Floating Point to Fixed Point in Lava

We now turn to the results obtained from the conversion between floating-point and fixed-point representations. Contrary to what might be expected, this step poses the greatest risk to network performance due to the operational range constraints imposed by the quantization process. As shown in Table 4.7, the conversion introduces varying degrees of accuracy loss across different splits.

The balanced split exhibited the smallest accuracy drop, losing only 1.080%,

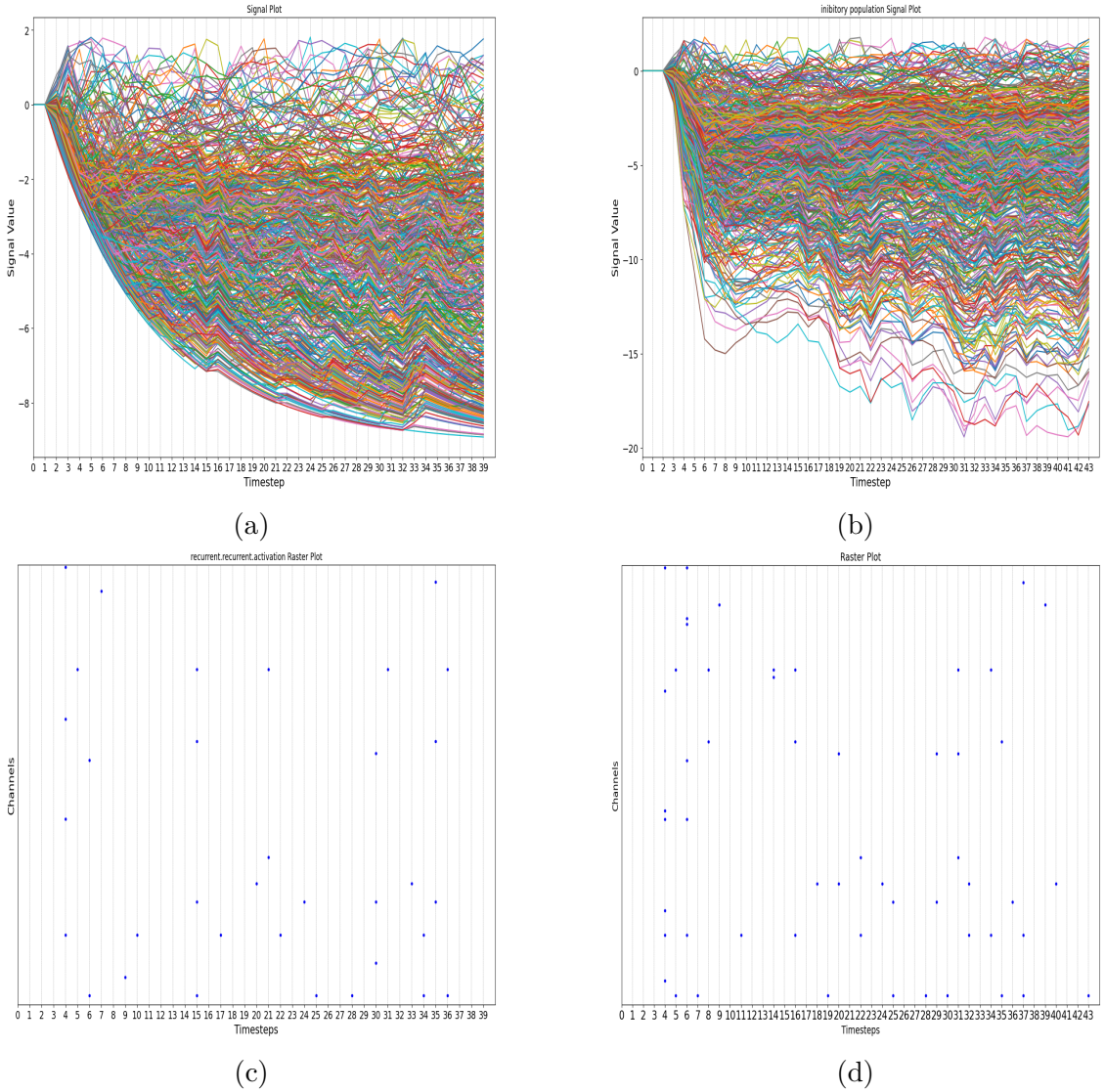


Figure 4.11: Comparison between **backward population** state and spikes of ((a) and (c)) snnTorch and ((d) and (b)) Lava

further reinforcing its robustness. The worst split experienced a slightly greater loss in accuracy. However, the most surprising result came from the High Score split, which showed the largest drop in accuracy at 9.338%, despite its otherwise favorable characteristics.

We will first analyze the conversion process for the balanced split to understand its resilience to quantization. Following this, we will take a closer look at the High Score split to identify the underlying causes of its significant accuracy loss during this step.

We will skip the first **encoding layer** for the fixed point visualization, given

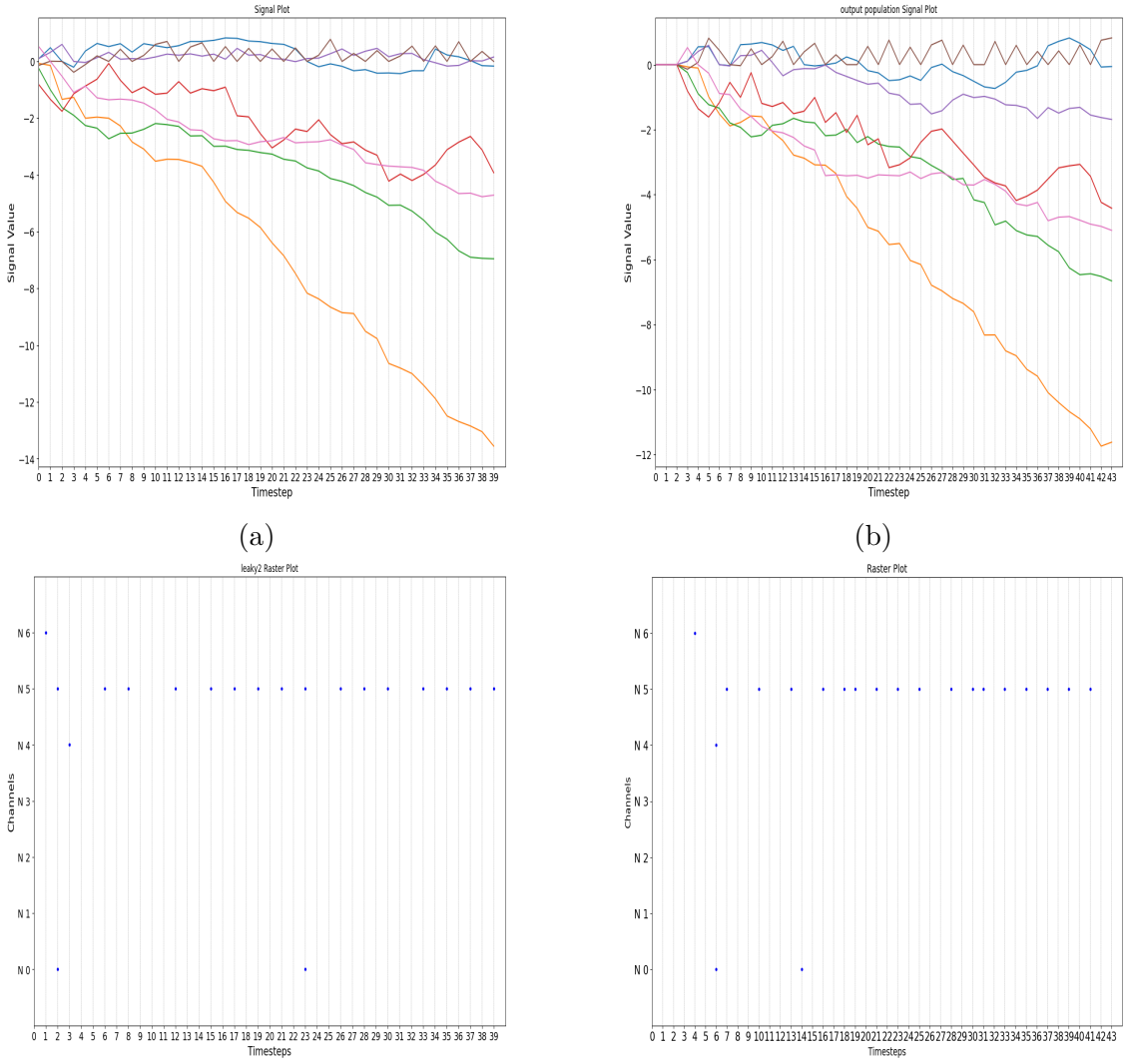


Figure 4.12: Comparison between **output population** state and spikes of ((a) and (c)) snnTorch and ((d) and (b)) Lava

that it will not be converted in this step.

Split name	**Lava fp accuracy**	**Lava fixed accuracy**	**accuracy loss**
High score split	91.609	82.271	**9.338**
Balanced score split	95.926	94.846	**1.080**
Worst score split	88.875	84.593	4.282

Table 4.7: comparison of validation score for the same network in Lava using floating point and fixed point arithmeticians

Forward layer In Figure 4.13, the primary effects of quantization are evident: range limitations and signal saturation at the lower boundary. This fundamental change in state dynamics leads to noticeable differences. While the spiking patterns remain generally similar to the original floating-point behavior, the saturation at the bottom limit reduces spiking activity, lowering the later information encoding that can lead to the registered accuracy loss. As a result, neurons are less likely to spike compared to the original floating-point behavior.

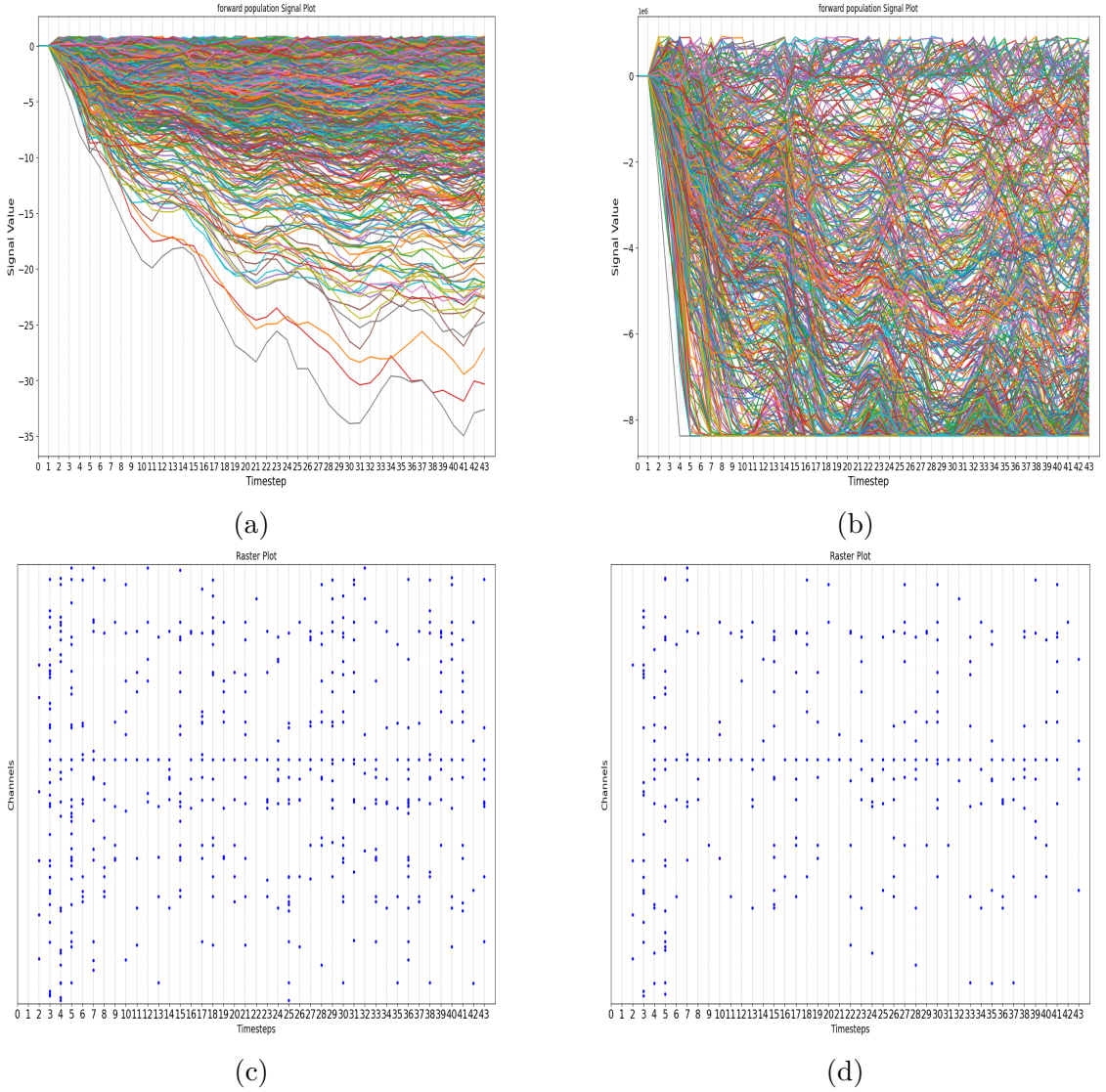


Figure 4.13: Comparison between **forward population** state and spikes of Lava ((a) and (c)) Floating-pt and ((d) and (b)) Lava Fixed-pt

Backward layer Similar observations can be made for the backward population, as shown in Figure 4.14. Given the decrease in spiking activity of the forward population, one might expect a corresponding decrease in the backward feedback because there is less need to help regulate and keep the forward population "under control." However, this adjustment is evident in a meaningful and visible way, as the backward population consistently lowered the spiking activity. This lack of significant feedback response highlights the saturation effects on the network dynamics.

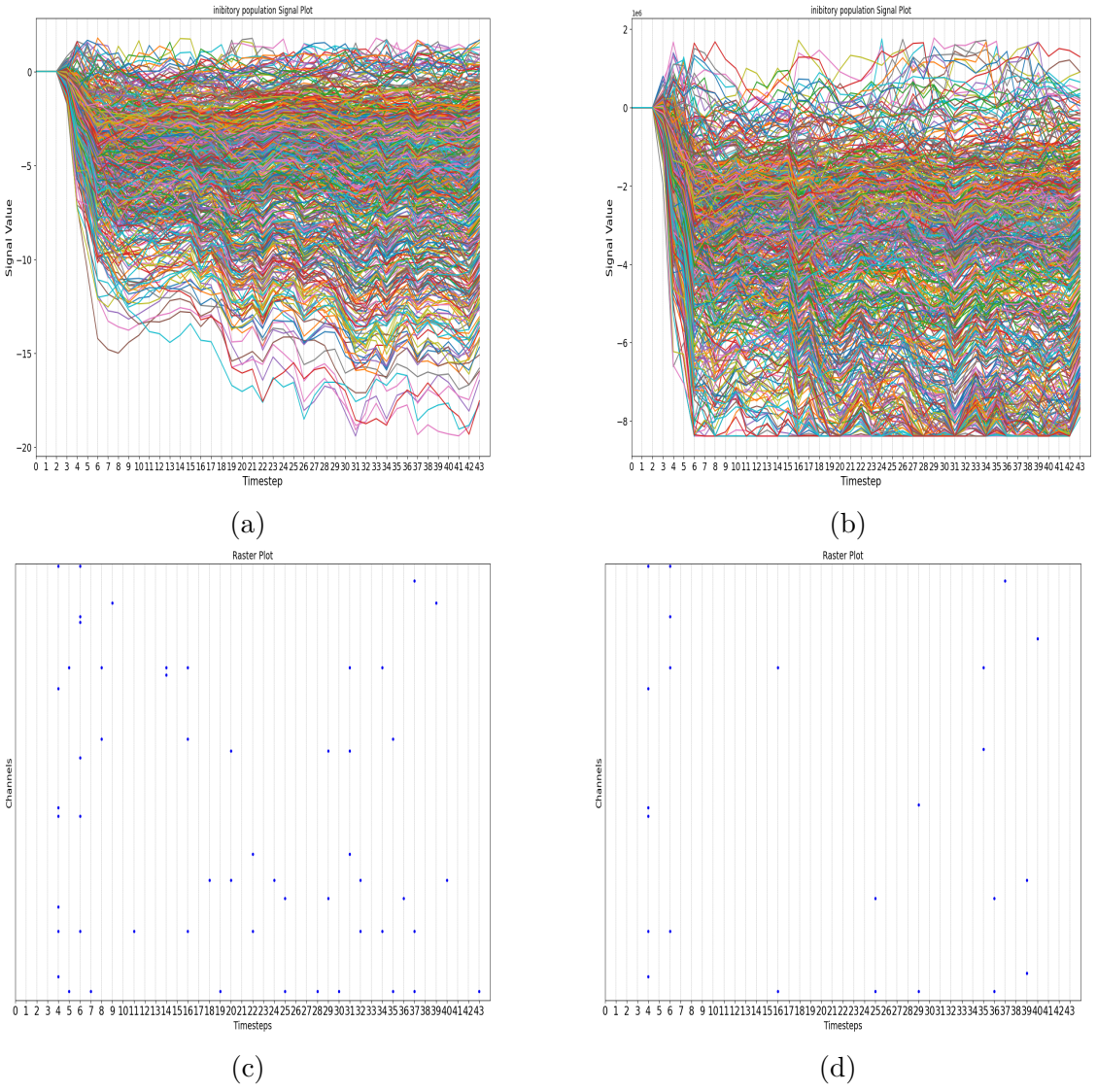


Figure 4.14: Comparison between **backward population** state and spikes of Lava ((a) and (c)) Floating-pt and ((d) and (b)) Lava Fixed-pt

Output layer Examining the output population in Figure 4.15, it is evident that the network dynamics are preserved, with spike patterns closely resembling those of the floating-point implementation. In this population, the inhibitory effects are more pronounced. The spike pattern for the correct label shows reduced activity, reflecting the modulation introduced by quantization. Additionally, the outputs for incorrect labels remain silent throughout the entire time window, further highlighting the effects of the quantization limitations.

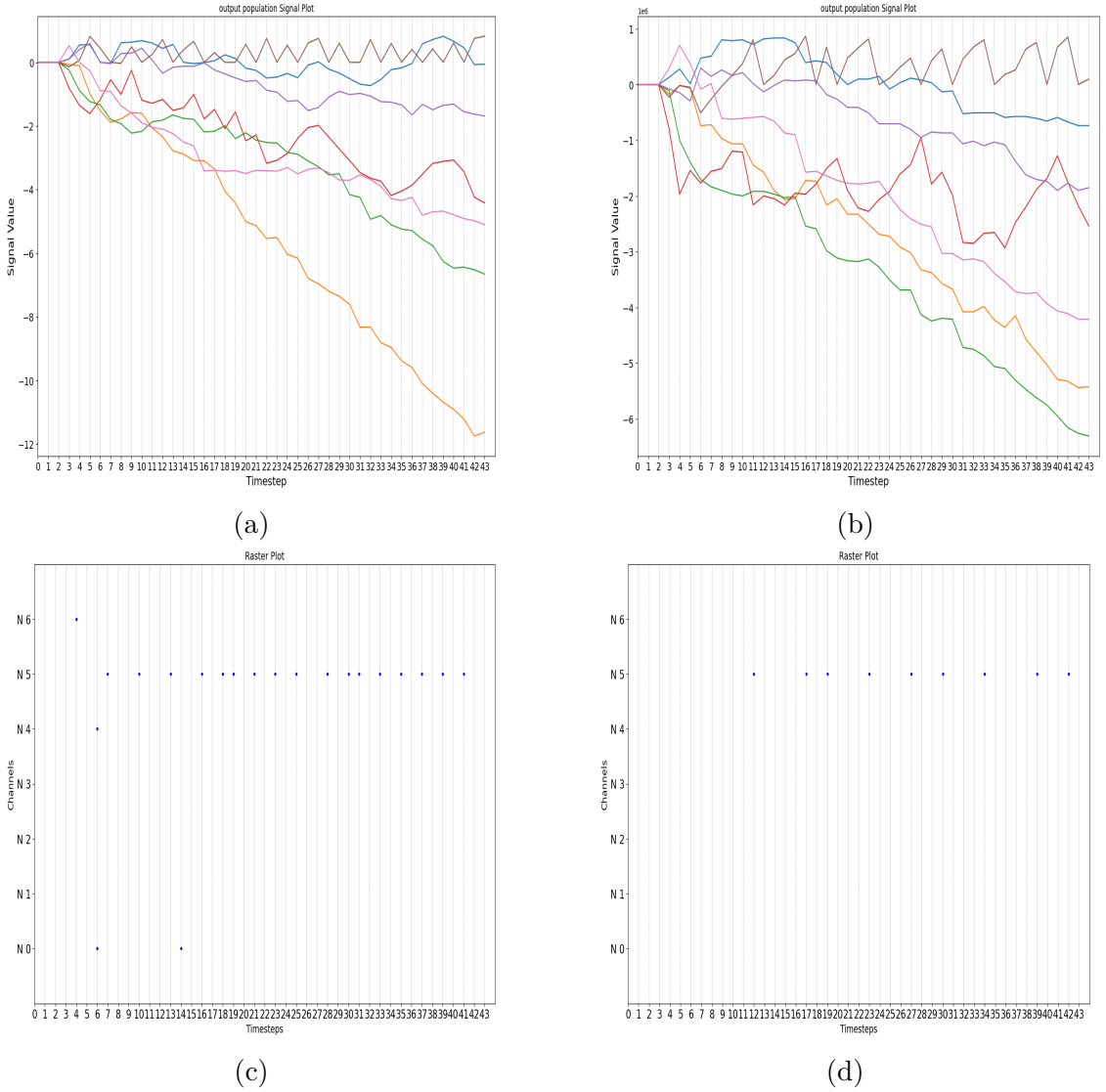
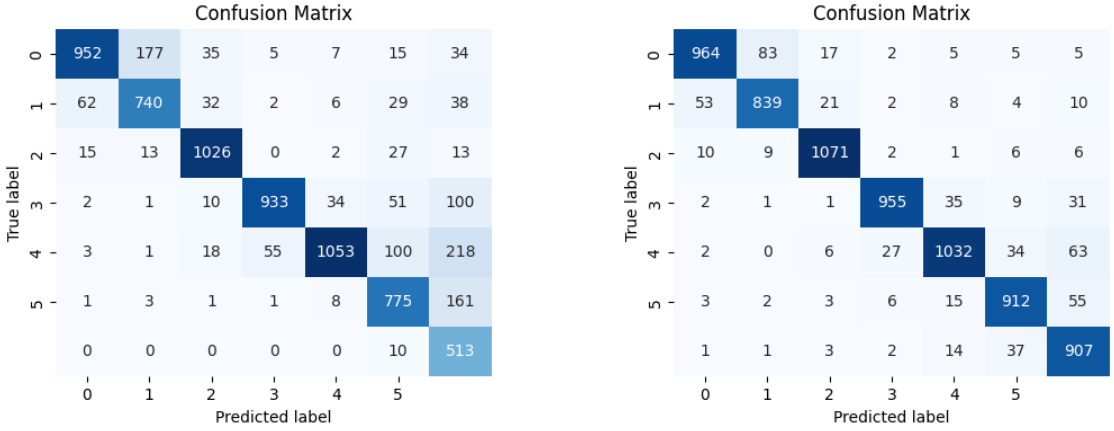


Figure 4.15: Comparison between **output population** state and spikes of Lava ((a) and (c)) Floating-pt ((d) and (b)) Lava Fixed-pt

investigate accuracy losses Given the relatively low accuracy loss observed for the conversion of the other two splits, the significant accuracy drop highlighted in Figure 4.7 for the High Score split warranted further investigation to understand the underlying reasons for this discrepancy. Such analysis was essential to identify the specific challenges posed by the conversion process and to determine whether factors like low separability or dynamic range clipping during quantization contributed to this performance degradation. The first idea was to take a look at the confusion matrix for this model on *Lava*, to see if the losses are distributed over all classes or are concentrated in some specific ones.



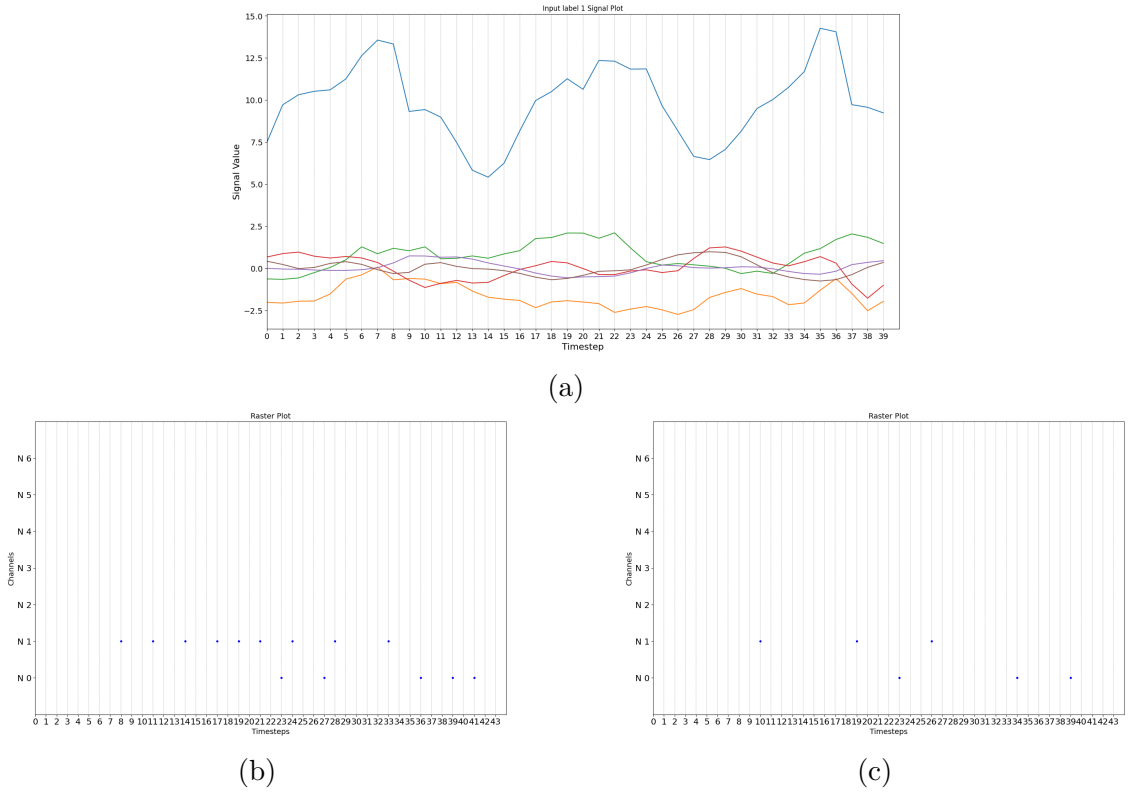
(a) (b)

Figure 4.16: Comparison of Confusion Matrices: (a) Fixed High score split (b) Original snnTorch results for High score split.

In Figure 4.16, it can be observed that the lower classification scores are primarily associated with classes 1, 5, and 6. Referring back to Figure 4.1, it becomes evident that these classes have the lowest separability between the actual label and the prediction. This suggests that their inherent difficulty in classification is further exacerbated by the range limitations imposed by quantization during the final dynamics conversion.

The hypothesis is that the quantization process may have truncated some of the meaningful information necessary for distinguishing these classes, leading to reduced performance. To gain a deeper understanding of this behavior, examining the corresponding raster plots can provide additional insights into the underlying dynamics.

As shown in Figure 4.17 and Figure 4.18, the inhibition patterns observed in previous analyses reveal that the output layer struggles to produce the correct classification. Due to the inherently low separability scores of these classes, the output layer exhibits difficulty in distinguishing the correct label from the incorrect ones, resulting in a marginal spike count difference.



(a)

(b)

(c)

Figure 4.17: (a)Class signals for label 1 (b) raster plot of *output layer floating point* network (c) raster plot of *output layer fixed point* network both in Lava

This already narrow spike margin is further penalized by the inhibition mechanism results from quantization. In the best-case scenario, as seen in Figure 4.17, the spike counts for both the correct and incorrect labels are nearly identical, reducing the network's confidence in the classification. In the worst case, depicted in Figure 4.18, the correct label is inhibited more than the incorrect one, ultimately leading the network to misclassify the input.

This behavior suggests that critical information encoded in the neurons' dynamics has been lost during processing, likely due to the range constraints introduced by quantization. This loss of dynamic information exacerbates the challenges faced by the network in distinguishing between classes with low initial separability, further hindering its ability to make accurate predictions.

Test results evaluation In this paragraph, the performance of the network on the test set will be evaluated, providing final considerations regarding the porting process based on the obtained results. This step was not performed on the actual chip for two primary reasons: the limited availability of chip runtime was insufficient to process the entire test set, and the computational accuracy of the

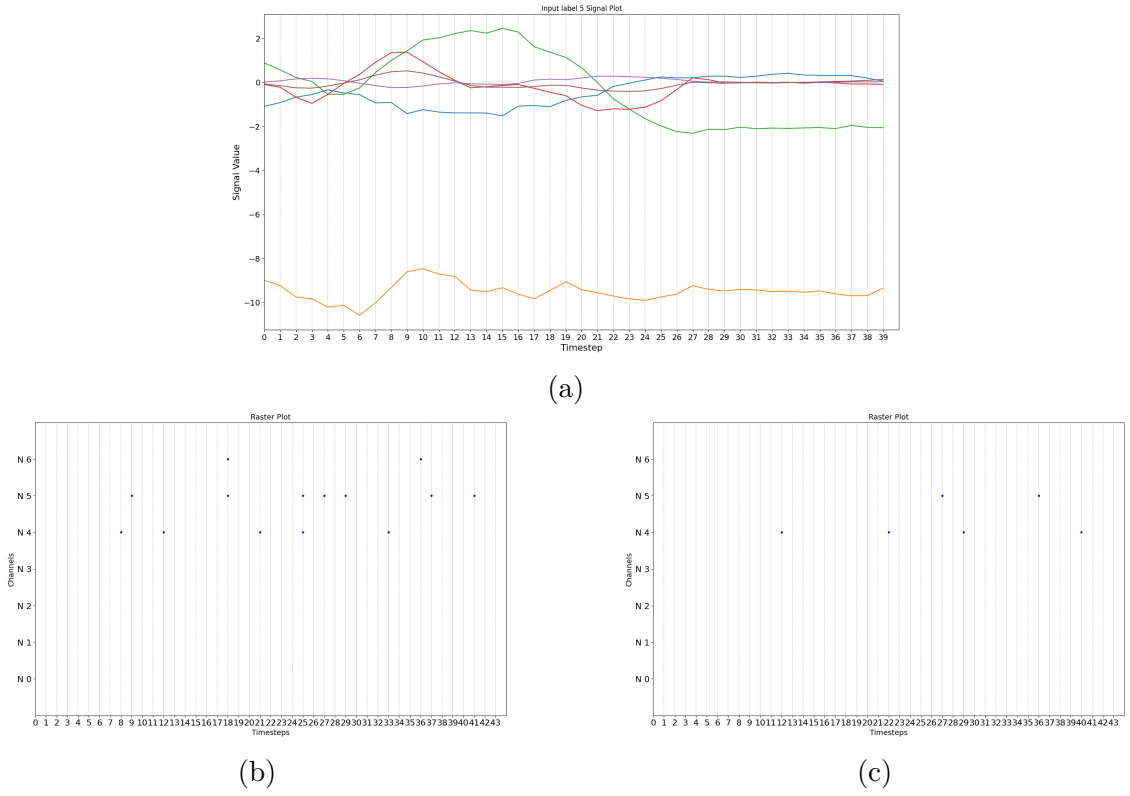


Figure 4.18: (a) Class signals for label 5, (b) raster plot of *output layer floating point* network (c) raster plot of *output layer fixed point* network both in Lava

fixed-point simulator is known to seamlessly replicate the chip's behavior. Therefore, the evaluation was carried out using the fixed-point simulator.

As shown in Table 4.8, the results demonstrate the network's ability to maintain its performance even after all the conversion steps and adaptations required to port it from *snnTorch* to *Lava* in fixed-point format.

Split name	**Lava fixed accuracy**
High score split	81.8347
Balanced score split	**95.0653**
Worst score split	84.1214

Table 4.8: Lava fixed test results for al three dataset splits

This is particularly evident for the split where all the efforts were focused: the *Balanced split*. Its uniform diversity among the classes facilitates seamless classification performance, making it robust even to the quantization step. The *Worst split*, despite its challenging separability, also performed above expectations, further validating the effectiveness of the network design.

However, the unexpected results from the *High score split* opened new opportunities for investigation. These results provided deeper insights into how quantization influences the network's behavior, highlighting critical areas for future improvement and optimization.

4.4.3 Lava fine-tuning using local learning rule

To ensure a seamless porting of the network and maintain its performance despite the observed drop, it was hypothesized that fine-tuning the network could help achieve comparable or even superior results to those obtained with *snnTorch*. For this purpose, the local learning rules provided by *Lava* were utilized.

Among these, local learning rules such as STDP and Reward-modulated Spike-Timing-Dependent Plasticity (RSTDP) were selected to update the synaptic weights to a potentially better state. These learning rules were already implemented within *Lava*, and existing examples were leveraged to guide the integration into the final steps of the pipeline.

The fine-tuning process involved applying these learning rules exclusively to the synapses of the output layer. Specifically, the synapses connecting between the forward branch of the *RInibitory block* (pre-synaptic neurons) to the *Output population* (post-synaptic neurons) were updated. This targeted approach allowed for a direct evaluation of the validity of the updates, as the classification accuracy could be used as a metric for penalizing or rewarding synaptic changes.

RSTDP, which incorporates a reward function, was particularly suited for this task, as it directly aligns synaptic updates with performance improvements. However, implementing these learning rules was not without challenges. While the provided examples served as a foundation, they lacked clarity, and when integrated into the pipeline, the resulting updates did not yield any significant improvements over the already achieved results. This outcome suggests either limitations in the application of these learning rules within the specific network context or the need for further refinement of the implementation process.

4.5 Loihi2 power consumption

As already explained, having developed all the previous pipelines significantly simplified the deployment process on the neuromorphic chip, requiring only a few additional steps to achieve the desired results. This step is critical, as it represents the final stage of transitioning the network from simulation to real-world hardware, where the true advantages of neuromorphic computing—such as energy efficiency and low latency—can be fully realized.

We opted not to run the entire validation procedure directly on the chip. Instead, thanks to extensive evaluation comparisons, we confirmed that the fixed-point simulator accurately mimics the chip's computational behavior. This allowed us to focus on measuring the power consumption of a single classification task, which is one of the most significant metrics for assessing the performance of neuromorphic systems. The decision to use the simulator was also necessitated by time constraints on the availability of the chip itself, emphasizing the importance of efficient pipeline development to minimize deployment bottlenecks.

Measurement	**Loihi 2**
Power (W): Static	0.212560
Power (W): Dynamic	0.037387
Power (W): Total	0.249947
Latency (μs)	10000
Energy (μJ): Total	900280.00
Energy (μJ): Dynamic	63761.00
Energy (μJ): Static	836519.00

Table 4.9: Performance Benchmarking Comparison Template (Transposed)

The proposed model occupies 21 neuromorphic cores. A single sample classification takes 10000 μs on the neuromorphic cores and consumes 900280.00 μJ of energy, 63761.00 μJ of which is dynamic energy. These results highlight the network's ability to maintain computational efficiency while adhering to the constraints of the neuromorphic hardware. All measurements were obtained using Lava on Loihi 2, version 0.6.0, on the board *oheogulch*. Table 4.9 provides a detailed breakdown of the energy consumption.

The deployment step not only validates the compatibility of the network with the hardware but also demonstrates its practical applicability for low-power, high-speed applications. This is a crucial milestone in leveraging the full potential of the Loihi 2 platform and neuromorphic computing in general.

4.6 Conclusion

This thesis presented a comprehensive approach for developing and deploying SNNs on neuromorphic hardware, focusing on creating a pipeline that balances flexibility, efficiency, and generalization. By leveraging existing tools such as *snnTorch* other than the predefined framework, *Lava*, addressing challenges across data preprocessing, training, and hardware deployment, the work demonstrated how to successfully adapt custom SNNs for fixed-point neuromorphic platforms.

The importance of data preprocessing was highlighted early in the pipeline, where different dataset splits were created to evaluate the network under various conditions, from best-case scenarios to challenging worst-case separability. The balanced split proved to be the most effective configuration, achieving the highest accuracy of 96.508% and showcasing its ability to manage class diversity and separability effectively. This outcome validated the importance of a well-designed split, showing how the careful selection of training subsets can greatly influence the final network performance.

The activation sparsity loss was another critical contribution to the pipeline, as demonstrated by the comparison between two models: one trained with sparsity loss and another without it. This comparison showed that the inclusion of sparsity loss not only improved classification accuracy but also optimized the spiking behavior of the neurons. The network trained with sparsity loss achieved higher activation sparsity and better accuracy compared to its counterpart. For example, the balanced split with sparsity loss achieved an activation sparsity score of *0.963* and an accuracy improvement of *8.674%* compared to the same network trained without sparsity loss.

This dual benefit underscores the importance of sparsity loss in reducing unnecessary activity while enhancing the encoding efficiency of spikes. By promoting efficient neuron activations, sparsity loss effectively balances computational efficiency with improved network performance, making it a valuable tool in the development of spiking neural networks.

Interestingly, even the worst-case split, with its inherently poor separability, performed above expectations. Achieving an accuracy of *88.875%*, this result demonstrated the robustness of the proposed network structure and optimization process. Although the high-score split faced unexpected challenges due to quantization effects, the analysis provided valuable insights into how fixed-point representation influences network behavior, offering new directions for future improvements.

The hardware deployment step further validated the pipeline's robustness. The transition from *snnTorch* to *Lava* in fixed-point format, while not without challenges, retained high computational fidelity. Using the fixed-point simulator to replicate chip behavior, it was shown that a single classification task required 10000 μs and consumed 900280.00 μJ, with 63761.00 μJ attributed to dynamic energy. The memory footprint of the final deployed model was *590.52* Kilo Bytes, distributed across *21* neuromorphic cores. These results reinforce the feasibility of deploying such networks on real-world hardware while maintaining the expected efficiency and performance.

In conclusion, this work demonstrated the viability of transferring SNNs to a new computational framework, achieving high performance even under challenging conditions. The proposed pipeline offers a scalable, adaptable, and efficient approach to deploying SNNs on neuromorphic hardware, paving the way for future

research into improving both network design and hardware compatibility. The insights gained from this study provide a strong foundation for further exploration into the development of energy-efficient and flexible neuromorphic computing systems, also expanding this pipeline compatibility with other frameworks such as Neuromorphic Intermediate Representation (NIR).

Acknowledgements

I would like to dedicate these lines to all the people who have accompanied me on this journey and made it possible to reach this important milestone.

First and foremost, I would like to thank my supervisor, Gianvito Urgese, for giving me the opportunity to explore this fascinating field of research related to neuromorphic computing, and for his unwavering support and valuable advice throughout this journey.

I would also like to express my sincere gratitude to Vittorio Fra, whose patience and expertise were crucial in helping me refine the concepts of this thesis. His deep knowledge of the field has been essential in shaping the quality of this research. I would also like to thank the members of the EDA research group at the Politecnico di Torino who, even with a small contribution, helped me to complete this work.
We acknowledge a contribution from the Italian National Recovery and Resilience Plan (NRRP), M4C2, funded by the European Union – NextGenerationEU (Project IR0000011, CUP B51E22000150006, "EBRAINS-Italy").

Grazie a mio padre che, anche se non si scompone mai troppo, mi dimostra sempre un affetto smisurato ed e' sempre pronto ad aiutarmi e consigliarmi, anche quando lo disturbo nei momenti piu' inopportune.

Grazie a mia madre che con i suoi modi gentili mi supporta quando nota delle piccole indecisioni, riesce sempre a tirarmi su il morale, consolandomi anche nei momenti peggiori, che mi chiama sempre per chiedermi anche stupidaggini, soltanto per sentire come sto.

Grazie a mi nonno che mi ha insegnato a usare il trapano e la penna, grazie a mia nonna che mi ha accudito e protetto quando nessuno era in casa, grazie ai miei zii, sempre disposti a darmi qualcosa da fare per non farimi annoiare.

Grazie a tutta la mia famiglia sparsa un po per tutta l'Italia, che mi fa sentire sempre amato e in quei pochi momenti all'anno in cui ci si riunisce, e' capace di farmi tornare bambino.

Grazie a mia sorella, la persona di cui vorrei dire troppe cose belle, in lei trovo sempre un confronto e un supporto indescrivibile, anche se ormai siamo distanti, non siamo mai stati cosi vicini, sono davvero entusiasta di avere te come sorella, di poche cose sono estremamente orgoglioso, la prima di queste e' il fantastico rapporto che abbiamo costruito.

Grazie ai miei amici che mi sopportano dalle scuole superiori, con cui abbiamo affrontato tutti insieme il trasferimento all'universita' e che anche in modo strampalato, continuiamo a volerci bene.

Grazie ai miei nuovi amici incontrati a Torino, con cui abbiamo condiviso studio, divertimento e bagna cauda, collezionato un numero indescrivibile di nuove esperienze vissute insieme.

Grazie a Elisa, Elisabetta e Christian, con cui abbiamo costruito un'amicizia speciale, fatta di semplici uscite, viaggi e momenti di sincero affetto.

Grazie ai nuovi coinquilini che mi hanno accolto nella nuova casa, grazie ai vecchi coinquilini, con cui abbiamo condiviso piccoli momenti di vita qotidiana, litigi e piccole confessioni, feste assurde e pranzi abbondanti.

Grazie ai vecchi membri di RoboTO, con cui abbiamo iniziato questo pazzo viaggio, pieno di difficolta' e sfide, che ci ha permesso di crescere ed imparare ad una velocita' impressionante, che ci ha permesso di costruire il gruppo di amici piu' bizzarro che c'e'.

Grazie ai nuovi membri, che con il loro entusiasmo, dedizione, passione e disturbi dell'attenzione, sono disposti a portare avanti la piccola realta' che abbiamo creato con tanta fatica.

Ringrazio tutti coloro che sono presenti, anche soltanto con il cuore in questa giornata, se siete qui, oggi, riconoscete il valore della nostra amicizia e avete contribuito a farmi arrivare a questo importantissimo traguardo.

Bibliography

[1] Wolfgang Maass. «Networks of spiking neurons: The third generation of neural network models». In: *Neural Networks* 10.9 (1997), pp. 1659–1671. ISSN: 0893-6080. DOI: https://doi.org/10.1016/S0893-6080(97)00011-7. URL: https://www.sciencedirect.com/science/article/pii/S0893608097000117.

[2] Sumit Bam Shrestha and Garrick Orchard. «SLAYER: Spike Layer Error Reassignment in Time». In: *Advances in Neural Information Processing Systems 31*. Ed. by S. Bengio et al. Curran Associates, Inc., 2018, pp. 1419–1428. URL: http://papers.nips.cc/paper/7415-slayer-spike-layer-error-reassignment-in-time.pdf.

[3] Sander M. Bohte, Joost N. Kok, and Han La Poutré. «Error-backpropagation in temporally encoded networks of spiking neurons». In: *Neurocomputing* (2002). DOI: https://doi.org/10.1016/S0925-2312(01)00658-0. URL: https://www.sciencedirect.com/science/article/pii/S0925231201006580.

[4] Bodo Rueckauer et al. «Conversion of Continuous-Valued Deep Networks to Efficient Event-Driven Networks for Image Classification». In: *Frontiers in Neuroscience* 11 (2017). ISSN: 1662-453X. DOI: 10.3389/fnins.2017.00682. URL: https://www.frontiersin.org/journals/neuroscience/articles/10.3389/fnins.2017.00682.

[5] J. Sjöström and W. Gerstner. «Spike-timing dependent plasticity». In: *Scholarpedia* (2010). DOI: 10.4249/scholarpedia.1362.

[6] Mike Davies et al. «Advancing Neuromorphic Computing With Loihi: A Survey of Results and Outlook». In: *Proceedings of the IEEE* (2021). DOI: 10.1109/JPROC.2021.3067593.

[7] Filipp Akopyan et al. «TrueNorth: Design and Tool Flow of a 65 mW 1 Million Neuron Programmable Neurosynaptic Chip». In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2015). DOI: 10.1109/TCAD.2015.2474396.

[8] 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.

[9] SynSense. *Xylo*. 2023. URL: https://www.synsense.ai/products/xylo/.

[10] Mike Davies et al. «Loihi: A neuromorphic manycore processor with on-chip learning». In: *Ieee Micro* 38.1 (2018), pp. 82–99.

[11] Garrick Orchard et al. «Efficient neuromorphic signal processing with loihi 2». In: *2021 IEEE Workshop on Signal Processing Systems (SiPS)*. IEEE. 2021, pp. 254–259.

[12] Sumit Bam Shrestha et al. «Efficient video and audio processing with Loihi 2». In: *ICASSP 2024-2024 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE. 2024, pp. 13481–13485.

[13] Intel. *Taking Neuromorphic Computing to the Next Level with Loihi 2*. 2021. URL: https://download.intel.com/newsroom/2021/new-technologies/neuromorphic-computing-loihi-2-brief.pdf.

[14] 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.

[15] Rodney Douglas, Misha Mahowald, and Carver Mead. «Neuromorphic analogue VLSI». In: *Annual review of neuroscience* 18.1 (1995), pp. 255–281.

[16] Alan L Hodgkin and Andrew F Huxley. «A quantitative description of membrane current and its application to conduction and excitation in nerve». In: *The Journal of physiology* 117.4 (1952), p. 500.

[17] Mike Davies et al. «Loihi: A Neuromorphic Manycore Processor with On-Chip Learning». In: *IEEE Micro* (2018). DOI: 10.1109/MM.2018.112130359.

[18] Jason K Eshraghian et al. «Training spiking neural networks using lessons from deep learning». In: *Proceedings of the IEEE* (2023).

[19] Nicolas Brunel and Mark van Rossum. «Quantitative investigations of electrical nerve excitation treated as polarization: Louis Lapicque 1907 · Translated by:» in: *Biological Cybernetics* (2007). DOI: 10.1007/s00422-007-0189-6.

[20] Evelina Forno et al. «Spike encoding techniques for IoT time-varying signals benchmarked on a neuromorphic classification task». In: *Frontiers in Neuroscience* (2022).

[21] Peter U. Diehl and Matthew Cook. «Unsupervised learning of digit recognition using spike-timing-dependent plasticity». In: *Frontiers in Computational Neuroscience* 9 (2015), p. 99. DOI: 10.3389/fncom.2015.00099.

[22] Junfeng Shao et al. «A Spiking Neural Network for Phoneme Recognition Using Real Acoustic Feature Vectors». In: *International Journal of Computational Intelligence Systems* (2012). DOI: 10.1080/18756891.2012.733224.

[23] Shih-Chii Liu and Tobi Delbruck. «Temporal Coding in Sensing and Decision Making». In: *IEEE Transactions on Neural Networks* 21.5 (2010), pp. 758–770. DOI: 10.1109/TNN.2010.2040793.

[24] Chunlin Yu et al. «A Tactile Sensing System for an Anthropomorphic Artificial Hand Based on Spiking Neural Networks». In: *Sensors* 19.3 (2019), p. 683. DOI: 10.3390/s19030683.

[25] Patrick Lichtsteiner, Christoph Posch, and Tobi Delbruck. «A 128 x 128 120 dB 15 µs Latency Asynchronous Temporal Contrast Vision Sensor». In: *IEEE Journal of Solid-State Circuits* 43.2 (2008), pp. 566–576. DOI: 10.1109/JSSC.2007.914337.

[26] Nikita Skatchkovsky, Garrick Orchard, and Jonathan Tapson. «Detection of Dynamic Events in Real-World Sensor Data Using Spiking Neural Networks». In: *IEEE Transactions on Neural Networks and Learning Systems* 31.2 (2020), pp. 393–405. DOI: 10.1109/TNNLS.2019.2899508.

[27] Jason Yik et al. *NeuroBench: A Framework for Benchmarking Neuromorphic Computing Algorithms and Systems*. 2024. URL: https://arxiv.org/abs/2304.04640.

[28] Francesco Fioranelli, Julien Le Kernec, and Syed Aziz Shah. «Radar for Health Care: Recognizing Human Activities and Monitoring Vital Signs». In: *IEEE Potentials* (2019). DOI: 10.1109/MPOT.2019.2906977.

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

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

[31] Irvine Univercity of California. *UCI Machine Learning Repository*. URL: https://archive.ics.uci.edu/.

[32] Intel. *Intel Neuromorphic Research Community*. 2021. URL: https://intel-ncl.atlassian.net/wiki/spaces/INRC/overview.

[33] Michael Ehrlich et al. «Adaptive control of a wheelchair mounted robotic arm with neuromorphically integrated velocity readings and online-learning». In: *Frontiers in Neuroscience* (2022). DOI: 10.3389/fnins.2022.1007736.

[34] Federico Paredes-Vallés et al. *Fully neuromorphic vision and control for autonomous drone flight*. 2023. URL: https://arxiv.org/abs/2303.08778.

[35] Tasbolat Taunyazov et al. *Event-Driven Visual-Tactile Sensing and Learning for Robots*. 2020. arXiv: 2009.07083 [cs.RO]. URL: https://arxiv.org/abs/2009.07083.

[36] Nabil Imam and Thomas Cleland. «Rapid online learning and robust recall in a neuromorphic olfactory circuit». In: *Nature Machine Intelligence* 2 (Mar. 2020), pp. 181–191. DOI: 10.1038/s42256-020-0159-4.

[37] Alessandro Pierro et al. *Solving QUBO on the Loihi 2 Neuromorphic Processor*. 2024. arXiv: 2408.03076 [cs.NE]. URL: https://arxiv.org/abs/2408.03076.

[38] Intel. *Lava Software Framework*. 2021. URL: https://lava-nc.org/.

[39] Intel. *Lava Deep Learning*. 2021. URL: https://lava-nc.org/lava-lib-dl/index.html.

[40] Wikipedia. *Kullback–Leibler divergence*. 2021. URL: https://en.wikipedia.org/wiki/Kullback%E2%80%93Leibler_divergence.

[41] Farhad Mortezapour Shiri et al. *A Comprehensive Overview and Comparative Analysis on Deep Learning Models: CNN, RNN, LSTM, GRU*. 2024. URL: https://arxiv.org/abs/2305.17473.

[42] Philipp Plank et al. *A Long Short-Term Memory for AI Applications in Spike-based Neuromorphic Hardware*. 2021. arXiv: 2107.03992 [cs.NE]. URL: https://arxiv.org/abs/2107.03992.

[43] B. Gutkin and F. Zeldenrust. «Spike frequency adaptation». In: *Scholarpedia* (2014). DOI: 10.4249/scholarpedia.30643.

[44] Alessandro Pappalardo. *Xilinx/brevitas*. 2023. DOI: 10.5281/zenodo.3333552. URL: https://doi.org/10.5281/zenodo.3333552.

[45] Quentin Ducasse et al. «Benchmarking quantized neural networks on FPGAs with FINN». In: *arXiv preprint arXiv:2102.01341* (2021).

[46] Jason K. Eshraghian et al. *Navigating Local Minima in Quantized Spiking Neural Networks*. 2022. URL: https://arxiv.org/abs/2202.07221.

[47] Simon Narduzzi et al. «Optimizing the consumption of spiking neural networks with activity regularization». In: *ICASSP 2022-2022 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE. 2022, pp. 61–65.

[48] Jianchuan Ding et al. «Biologically inspired dynamic thresholds for spiking neural networks». In: *Advances in Neural Information Processing Systems* 35 (2022), pp. 6090–6103.

[49] Kinjal Patel et al. *A Spiking Neural Network for Image Segmentation*. 2021. URL: https://arxiv.org/abs/2106.08921.