POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

Spiking LSTM on Loihi 2: A Neuromorphic Reinterpretation of Recurrent Networks

Supervisors

Prof. Yulia SANDAMIRSKAYA

Prof. Gianvito URGESE

Dr. Vittorio FRA

Dr. Walter GALLEGO GOMEZ

Candidate

Aurora GRUBER

December 2025

Abstract

In recent years, the presence of artificial intelligence (AI) has become increasingly pervasive, with artificial neural networks (ANNs) being applied across a growing number of domains. These models, while powerful, are also becoming larger and more computationally demanding. In addition, an interest has emerged in understanding and emulating the remarkable efficiency and compactness of the human brain. This has led to the rise of neuromorphic computing, which aims to design lightweight and energy-efficient systems inspired by biological neurons, through spiking neural networks (SNNs). By combining ideas from neuroscience and machine learning, neuromorphic computing offers a way to reinterpret traditional AI models in a more biologically grounded way. Among the various approaches to bridging the gap between classical and brain-inspired computation, revisiting well-established ANN architectures within a neuromorphic framework is a viable direction.

This work focuses on translating the Long Short-Term Memory (LSTM) network, a widely used recurrent architecture known for its ability to capture long-term dependencies and handle sequential data, into a spiking form, where operations are rephrased in terms of neuron populations and synapses. The goal of this reinterpretation is to make the architecture compatible with neuromorphic hardware, enabling efficient execution on brain-inspired systems.
Several spiking LSTM (sLSTM) variants were explored, and the final design replaces conventional activation functions with the spiking dynamic of Leaky Integrate-and-Fire (LIF) neurons, leveraging membrane potential values for the computation of the internal states. The architecture was implemented on two frameworks: snnTorch, used for training and hyperparameter optimization via the Neural Network Intelligence (NNI) framework and NxKernel, Intel's proprietary framework for deployment on the Loihi 2 neuromorphic board.
The transition between the two frameworks required the implementation of custom neurons in microcode to reproduce the behavior observed in snnTorch. Additional challenges arose from the fixed-point arithmetic used in Loihi 2's synapses and neuron models, which demanded careful quantization and scaling strategies. Moreover,

Loihi 2's pipelined execution introduced differences in layer synchronization compared to the software-based simulation, requiring further adaptations to preserve consistent network dynamics across frameworks. Using the profiling tools available in the NxKernel framework, the model performance on the Loihi 2 hardware was evaluated, taking into account the effects of network sparsity and also multiple partitioning and mapping configurations obtained through a heuristic optimization algorithm.
For initial experiments, a Human Activity Recognition (HAR) task was used, employing a spike-encoded dataset with six input channels and seven output classes. Once a robust pipeline for training, weight transfer and hardware deployment was established, the architecture was further tested on the Spiking Heidelberg Digits (SHD) dataset, which involves classifying spoken digits. Without any preprocessing, the model was trained on snnTorch, achieving competitive test accuracies up to 90%. When deployed on Loihi 2, it exhibited a slight accuracy drop, yet the overall results highlight the potential of this neuromorphic reinterpretation of LSTM networks.

Table of Contents

List of Tables

List of Figures

Acronyms

AI

Artificial Intelligence

LSTM

Long Short Term Memory

GRU

Gated Recurrent Unit

sLSTM

spiking Long Short Term Memory

LIF

Leaky Integrate and Fire

ANN

Artificial Neural Network

RNN

Recurrennt Neuronal Network

CNN

Convolutional Neuronal Network

SNN

Spiking Neuronal Network

HAR

Human Activity Recognition

WISDM	Wireless Sensor Data Minin
SHD	Spiking Heidelberg Digits
NNI	Neural Network Intelligence
BPTT	Backpropagation Through Time
LR	Learning Rate

Chapter 1

Introduction

Throughout history, the idea of creating technology with brain-like abilities has been both fascinating and a source of innovation. The human brain performs remarkable feats thanks to its fundamental units, the neurons, which exchange information through discrete action potentials, or “spikes”. Neurons are interconnected through synapses, which enable the transfer of these signals [1].

Neuromorphic computing is a brain-inspired field driven by the goal of mimicking the nervous system by emulating the structures, processes and computational capabilities of biological neurons and synapses. In this context, neuromorphic hardware has been developed, along with the corresponding software frameworks needed to support it. Neuromorphic hardware operates under a spiking paradigm [2]: each computational unit, analogous to a biological neuron, becomes active only when it receives or emits information in the form of electrical impulses. As a result, these systems process information through sparse, event-driven signals, leveraging the brain’s inherently massively parallel, event-driven, and analog-inspired computation principles [3]. Spiking neural networks (SNNs) have emerged as a biologically inspired alternative to conventional artificial neural networks (ANNs) [4]. SNNs use discrete spikes as their basic units of computation, capturing key temporal and event-driven aspects of neural processing, making them well suited for deployment on neuromorphic hardware.

A growing number of neuromorphic platforms has been developed to support this computational paradigm, including Intel’s Loihi family of processors, which provides a flexible environment for SNN research. Loihi 2 [5], in particular, is optimized for SNN workloads, combining programmable neuron models, on-chip learning mechanisms and fully asynchronous spike-based communication.
Software frameworks such as snnTorch provide flexible simulation environments that allow researchers to design, train and evaluate spiking neural networks before deploying them to hardware. While frameworks like NxKernel are more hardware oriented, exposing Loihi 2 primitives and neuron-level. Bridging the gap between

these software tools and the constraints of neuromorphic processors remains a key challenge in the field.

Due to the capabilities and flexibility of SNNs, numerous attempts have been made to reinterpret conventional neural architectures within a spiking framework, demonstrating that replacing continuous activations with event-based computations is feasible and can reduce the computational cost while preserving the core functionality of the original models.
The core contribution of this Thesis lies in this context: the proposal of a spiking version of the LSTM, one of the most successful recurrent neural network architectures, which keeps information about the input data in both a short-term and a long-term memory units. In the proposed version the network retains the functional principles of the original LSTM, including short-term and long-term memory mechanisms, while fully embracing event-driven spiking computation. Each gate is realized with dedicated neuron populations, enabling the network to process temporal information efficiently on neuromorphic hardware.

During the initial phase of this work, the sLSTM architecture was implemented and evaluated in the snnTorch framework. Subsequently, it was porting to NxKernel, after a careful restructuring of its operations, as all computations on Loihi 2 must be expressed as neuron-level state updates and event-driven transitions, a constraint absent in snnTorch. This process also involved designing custom neuron models, made possible by Loihi 2's microcoded neurons. A key contribution of this work is the development of a robust pipeline that allows the trained sLSTM models in snnTorch to be accurately translated into NxKernel, ensuring that weights, thresholds and neuron dynamics can be deployed on Loihi 2 with minimal performance loss, aligning the behavior between the software simulation and hardware execution.

The Spiking Heidelberg Digits (SHD) dataset was selected as the final benchmark to evaluate the proposed architecture.
Experimental results demonstrate that the spiking LSTM achieves high performance, with a test accuracy of 91.92% on snnTorch and 85.34% on NxKernel. While these results place the model comparably on the current leaderboard, it is the only top-performing architecture that has been successfully deployed on any neuromorphic hardware, thus holding the leading position in terms of actual on-chip accuracy.
Beyond raw accuracy, a detailed analysis of the model was conducted, including hardware profiling and studies on network partitioning. These investigations reveal how neuron distribution and network partitioning impact runtime, synaptic reads, and overall efficiency.

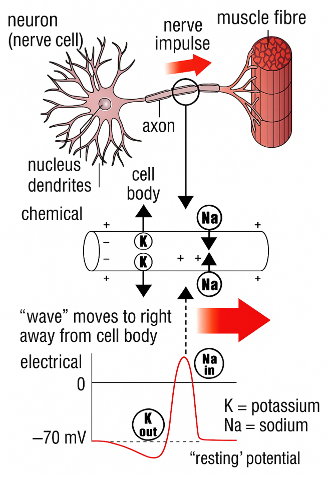
Chapter 2

Background

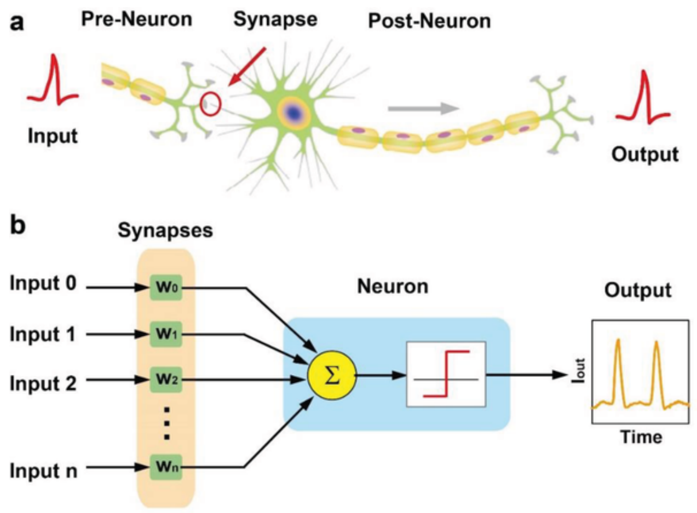
2.1 Neuromorphic Computing

Neuromorphic computing represents a revolutionary paradigm in computer science that fundamentally emulates the operational principles of the human brain [6, 7, 8, 9]. This approach involves the design of both hardware and software systems structured to mimic the neural and synaptic functions (Figure 2.1b) essential for information processing in biological brains. A central goal in neuromorphic research is to achieve the human brain's remarkable combination of adaptability, learning capability from unstructured data, and exceptional energy efficiency [10, 11, 12].

At its core, the brain's functionality relies on neurons (Figure 2.1a), its fundamental units. These nerve cells act as vital messengers, transmitting information across various brain regions and throughout the body. When a neuron fires i.e. it emits a spike and becomes active, it releases a cascade of chemical and electrical signals. These signals traverse intricate networks of connection points known as synapses, enabling communication between neurons [15]. Given this direct inspiration, neuromorphic computing draws extensively from principles in biology and neuroscience.
The replication of these intricate neurological and biological mechanisms in neuromorphic systems is primarily achieved through spiking neural networks (SNNs). Unlike traditional artificial neural networks (ANNs), SNNs are specifically designed to emulate the discrete, event-driven communication of biological neurons (Figure 2.2).
In an SNN, artificial neurons accumulate data through time, via their membrane potential and process them. Each neuron possesses an internal charge, representing the membrane potential, as well as delay and threshold values, much like their biological counterparts. Synapses establish connections between these artificial neurons, each with its own delay and weight parameters. These diverse values –



(a) Structure, chemical processes and electrical activity of the neuron. (Image from [13])



(b) Mapping the principles of neuronal functioning into a neuromorphic system. (Image from [14])

Figure 2.1: Comparison between a biological neuron and its neuromorphic counterpart.

including neuron charges, delays for both neurons and synapses, neuron thresholds and synaptic weights – are all programmable within neuromorphic computing architectures [16, 17].

Beyond this general structure, the specific mathematical model employed for artificial neurons significantly influences an SNN's behavior and computational capabilities. Various neuron models exist, each offering different levels of biological realism and computational complexity. Common examples include the Hodgkin-Huxley model, Izhikevich model and the Leaky Integrate-and-Fire (LIF) model.

The LIF model [20] is particularly prevalent in neuromorphic hardware and simulations due to its balance of computational efficiency and biological plausibility. In the LIF model (Figure 2.3), a neuron accumulates input currents from its connected synapses over time. This accumulation causes its membrane potential $V_m(t)$ to rise. The "leaky" aspect signifies that if the neuron does not receive sufficient input, the membrane potential gradually decays (represented by the decay term dV_m) back towards a toward a zero state, mimicking the passive ion channels in biological neurons [20]. Once the membrane potential reaches a predefined threshold, V_{th}, the neuron "fires", generating an output pulse that is then transmitted to downstream neurons. After spiking, the neuron's potential is typically returned to a predefined reset level, V_{reset}. The dynamics of the membrane potential $V_m(t)$ for a LIF neuron are typically described by the following differential equation:

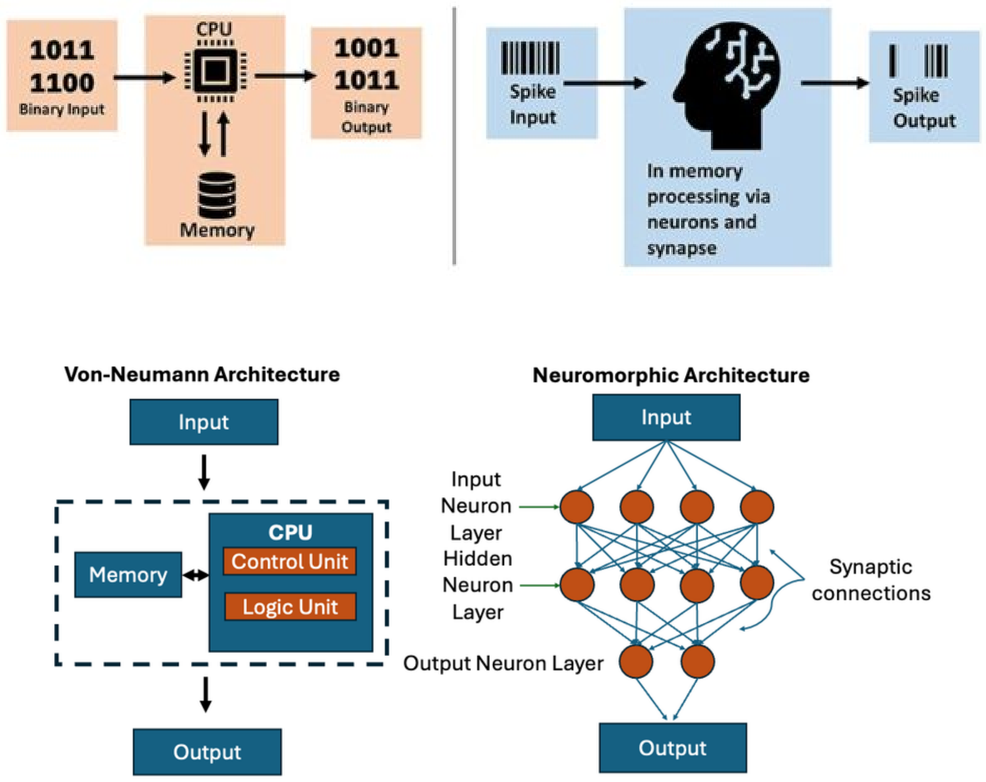


Figure 2.2: Comparison of ANN and SNN, highlighting differences in input representation and architecture. (Images from [18] and[19])

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

where:

- $V_m(t)$ is the membrane potential at time t.
- V_{rest} is the resting potential of the neuron.
- $\tau_m = R_m C_m$ is the membrane time constant, representing how quickly the potential changes.
- R_m is the membrane resistance.
- C_m is the membrane capacitance.
- $I(t)$ is the total input current flowing into the neuron at time t.

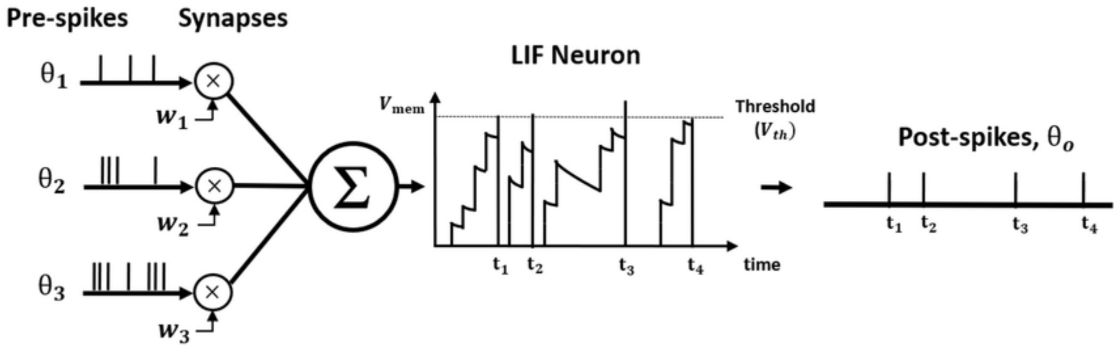


Figure 2.3: Schematic representation of the Leaky Integrate-and-Fire (LIF) neuron and its firing behavior. (Image from [21])

When $V_m(t)$ reaches the threshold potential V_{th}, the neuron fires a spike, and its membrane potential is then reset:

$$V_m(t) \to V_{\text{reset}} \quad \text{if} \quad V_m(t) \geq V_{\text{th}}$$

These mathematical formulations capture the integrate-and-fire behavior, providing a practical foundation for scalable neuromorphic implementations.

2.1.1 Benefits and Challenges of Neuromorphic Computing

Neuromorphic systems present an attractive vision for future computation, offering several significant advantages [22]:

Adaptability and Dynamic Learning Drawing inspiration from biological brains, neuromorphic computing systems are highly flexible and adaptive in solving diverse and complex problems. To do so, neuromorphic systems depend on continuous, real-time learning, allowing them to adapt fluidly to new inputs, stimuli, or environmental shifts. Such inherent flexibility allows them to identify challenges and address them in the most human-like way.

Energy Efficiency Neuromorphic systems are event-based: neurons and synapses are processed only in response to spikes from other neurons. Unlike traditional architectures that are continuously computed, in neuromophic networks power is consumed only by its active parts, while the rest remains idle, leading to substantial energy savings. These characteristics make neuromorphic systems particularly well suited for edge computing and low-power, real-time applications.

High Performance and Parallel Processing Conventional von Neumann computers suffer from a "bottleneck" caused by separate processing (CPU) and memory units and the constant transfer of data between them. In stark contrast, neuromorphic computing systems integrate both data storage and processing directly within individual "neurons". By tightly coupling memory and compute, they greatly reduce latency and achieve faster computation through minimized data transfers. Moreover, the asynchronous operation of Spiking Neural Networks (SNNs) allows individual neurons to perform distinct operations simultaneously. Theoretically, a neuromorphic device can execute as many tasks concurrently as it has neurons, showcasing immense parallel processing capabilities and enabling rapid function completion.

Efficient Information Encoding and Real-time Dynamics Neuromorphic architectures excel in sparse, distributed information encoding through spikes or events that inherently carry temporal information. This contrasts with dense, continuous data representation in traditional systems. Coupled with their ability to utilize dynamics across several timescales, neuromorphic systems are particularly well-suited for real-time learning and processing of complex, time-dependent data streams.

However, it is crucial to understand that these features do not offer a "magical" universal solution, neuromorphic computing remains an emerging field and like any nascent technology, it faces several significant challenges:

Decreased Accuracy The process of converting trained deep neural networks into spiking neural networks can sometimes lead to a reduction in accuracy. Furthermore, memory resistors, often used in neuromorphic hardware, can exhibit variations in their cycle-to-cycle and device characteristics, which may impact overall accuracy. In addition, hardware limitations such as restricted synaptic weight precision can further contribute to performance degradation.

Lack of Benchmarks and Standards As a relatively new technology, neuromorphic computing still lacks of standardized architectures, hardware and software frameworks. Although recent progress has introduced several benchmark datasets [23, 24, 25, 26, 27] and evaluation protocols[28] for spiking and neuromorphic systems, the field still lacks unified standards. Variations in architectures, coding schemes and learning paradigms make it challenging to establish consistent performance metrics or compare results across different hardware platforms and implementations.

Limited Accessibility and Software Ecosystem The software and hardware ecosystem for neuromorphic computing is still in development phase. While progress in recent years has improved accessibility, many frameworks, development tools and hardwares remain in active research or early-access stages. This limited availability can hinder experimentation, portability and reproducibility.

2.1.2 Neuromorphic Hardware

Neuromorphic hardware refers to physical computing systems that emulate the structure and dynamics of biological neural networks. These architectures are typically event-driven and operate using spiking communication, offering high energy efficiency and parallelism. Notable large-scale neuromorphic hardwares include the SpiNNaker [29, 30] and BrainScaleS [31, 32] projects. Other emerging platforms, including Xylo [33], SIMON and Speck [34], NeuroGrid [35] and the DYNAP [36, 37] family of chips, offer analog or mixed-signal approaches for low-power, real-time spiking neural computation. Among existing implementations, Intel's Loihi stands out as a digital neuromorphic chip designed for scalability and on-chip learning. Its architecture supports networks of spiking neurons interconnected through programmable synapses, enabling experiments in low-power and real-time computation. While several other neuromorphic platforms exist (such as analog and mixed-signal approaches), this Thesis focuses on the Loihi family of chips, which provides a flexible environment for spiking neural network research.

Loihi 2: Intel's Neuromorphic Research Chip

To provide functional systems for implementing SNNs, Intel Labs introduced Loihi 2 [5], a neuromorphic research chip designed specifically for large-scale spiking computation. This architecture is optimized for SNN workloads, combining programmable neuron models, on-chip learning mechanisms and fully asynchronous spike-based communication. The chip is engineered for high silicon density and fast circuit execution, enabling the deployment of complex spiking networks with unprecedented computational efficiency.
In addition to its architectural flexibility, Loihi 2 operates at extremely low power, typically consuming well below one watt [5], in stark contrast to the tens or even hundreds of watts required by conventional CPU- and GPU-based systems. Despite this small power envelope, the chip achieves state-of-the-art response times to incoming data streams and supports continuous on-chip learning and adaptation. This combination of low latency, energy efficiency and online adaptability positions Loihi 2 among the most versatile neuromorphic platforms available today.
Its rich feature set, including programmable neuron models, configurable plasticity rules and high-resolution spike messaging, enables a broad range of use cases,

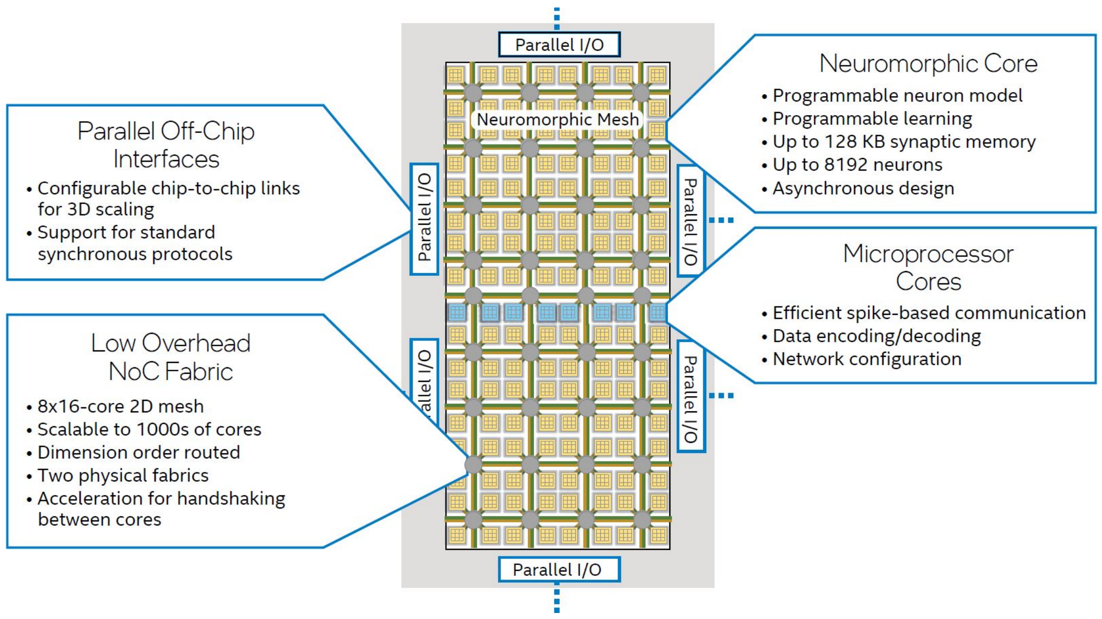


Figure 2.4: Loihi 2 architecture overview. (Image from [5])

though pushing these capabilities often demands increasing computational scale. As workloads grow, achieving larger network sizes and more demanding behaviors typically requires scaling across multiple Loihi 2 chips, leveraging the chip's multi-chip communication fabric to support distributed neuromorphic systems. Another key aspect of the chip is its built-in profiling capability, i.e., the ability to measure activity, runtime, energy consumption and other insightful characteristics of the network deployed on the hardware, making it a strong target for research projects. The Loihi 2 neuromorphic chip (Figure 2.4) integrates approximately 130,000 neurons, each capable of communicating with thousands of others through programmable synapses. Its 128 neuron cores include embedded learning engines, enabling programmatic control of on-chip resources and supporting complex spiking computations.

Synapse and Neuron Model Loihi 2 supports generalized event-based messaging in which spikes can optionally carry integer-valued payloads, allowing each spike to encode additional information while preserving the sparse and time-coded communication typical of spiking neural networks. These payloads, programmable up to 32 bits, are specified directly through the neuron's microcode and are used to modulate downstream synaptic weights with minimal energy or performance overhead. Neuron dynamics are defined by a programmable microcode pipeline within each core, where each neuron executes a compact sequence of instructions that implement its update rules. Because this microcode is fully configurable, neurons

Table 2.1: Highlights of the Loihi 2 Instruction Set from [5]

OP CODES	DESCRIPTION
RMW, RDC read-modify-write, read-and-clear	Access neural state variables in the neuron's local memory space.
MOV, SEL move, move if 'c' flag	Copy neuron variables and parameters between registers and the neuron's local memory space.
AND, OR, SHL and, or, shift left	Bitwise operations.
ADD, NEG, MIN add, negate, minimum	Basic arithmetic operations.
MUL_SHR multiply shift right	Fixed precision multiplication.
LT, GE, EQ less than, greater or equal, equals	Compare and write result to 'c' flag.
SKP_C, JMP_C skip ops, jump to program address based on 'c' flag	Branching to navigate program.
SPIKE, PROBE spike, send probe data	Generate spike or send probe data to processor.

can be customized to support a wide range of behaviors, models and computation patterns. The instruction set, summarized in table 2.1, includes arithmetic and bitwise operations, comparisons, conditional branching, memory access and dedicated instructions for generating and probing spikes. Synaptic connections themselves also support programmable delays, enabling fine-grained temporal control over spike propagation and network dynamics. This flexible programming model enables the implementation of a wide range of neuron behaviors and interaction patterns, from simple integrate-and-fire mechanisms to more elaborate, algorithmically defined update rules, without compromising computational efficiency.

Learning Capabilities The on-chip learning framework accommodates a wide range of synaptic update rules. Localized modulatory signals can be applied directly at the level of individual synapses, enabling the implementation of many neuro-inspired learning algorithms, including approximations of the *error backpropagation* algorithm. This design supports continuous adaptation to incoming data streams and allows networks to adjust to changing conditions in real time.

Resource and Memory Optimization Each neuromorphic core provides its own memory slot, which can be flexibly partitioned, adapting its storage resources to the requirements of a given network, balancing neuron state, synaptic data and internal buffers. Synaptic connectivity can be encoded in multiple formats, enabling efficient representation of both sparse and dense connectivity patterns. A local spike-broadcast mechanism reduces bandwidth usage across inter-core and inter-chip communication pathways, supporting large-scale networks while maintaining low memory overhead.

Speed and Interface Capabilities The chip executes neuron updates, synaptic operations and spike-processing logic through an asynchronous event-driven architecture designed for low latency and high throughput. Standardized interfaces—including Ethernet, GPIO, SPI and asynchronous event-based protocols—facilitate integration with digital systems, neuromorphic sensors and multichip configurations. This versatility enables the construction of scalable neuromorphic applications spanning embedded, edge and cloud-connected environments.

Processing modes Loihi 2 supports balancing throughput and latency depending on application requirements, enabling a range of processing modes spanning from pipelined to fall-through operation [38]. In the pipelined mode, a new input is injected at every time step. This enables all neuronal layers to operate concurrently in a pipeline fashion, resulting in maximal throughput. However, enforcing a fixed progression of time steps increases the latency experienced by individual inputs.
Conversely, the fall-through mode introduces new inputs only after the previous one has propagated through the entire network. At any given time step, only a single neuronal layer is active, reducing internal traffic and allowing the network to advance as quickly as possible. This minimizes per-sample latency, although overall throughput is limited by the time required for each input to traverse the architecture.
The effective operating point can be tuned by adjusting the rate at which new inputs are provided, enabling a continuum between strictly pipelined and strictly fall-through operation. This flexibility allows Loihi 2 to accommodate workloads ranging from high-throughput streaming applications to latency-critical real-time processing.

Loihi 2 performance bottlenecks As shown in [39], Loihi 2 like systems are characterized by three main execution bottlenecks, in order of relevance:

1. Synaptic memory reads in each neuron core, to fetch the synpatic weights

2. Dendrite updates in each neuron core, to execute each neuron's dynamics and update its state

3. Traffic congestion in the Network on Chip, to route spikes from source to destination neuron core

Given that in Loihi 2, at each timestep there is a barrier synchronization between all neuron cores, the memory and dendrite bottlenecks become effectively those of the slowest neuron core. For this reason, correct network partitioning, that attempts to balance the memory and execution load across all neuron cores, is essential in this type of parallel and pipelined architectures.

Traffic congestion on the other hand is reduced by an appropriate mapping strategy, that attempts to balance the traffic load across the routers in the network on chip, and to reduce the distance between pair of neuron cores that need to communicate the most.

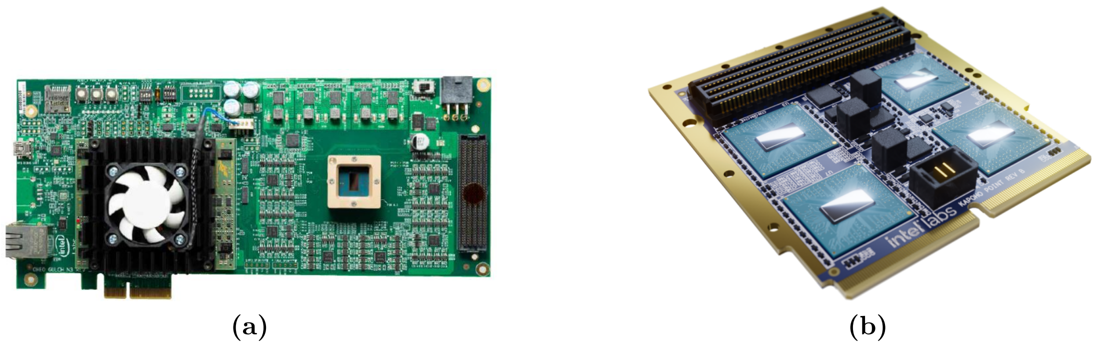

(a) **(b)**

Figure 2.5: Two of the available Loihi 2 hardware: Oheo Gulch on the left and Kapoho Point on the left.

Loihi 2 Hardware Availability Intel provides Loihi 2–based neuromorphic systems to the research community primarily through the Neuromorphic Research Cloud, which enables remote access to shared computational resources. Among the available platforms is *Oheo Gulch* (Figure 2.5a), a single-chip board designed for laboratory evaluation and detailed characterization. The chip is mounted on a socket and instrumented for low-level debugging, while exposing Loihi 2 through a standard Ethernet interface, allowing researchers to run experiments and collect detailed measurements on the chip's behavior. A second platform, *Kapoho Point* (Figure 2.5b), is a compact and stackable system that integrates eight Loihi 2 chips within an approximately 4×4-inch form factor. It offers Ethernet connectivity and is designed to support dense, scalable multi-chip configurations for more demanding neuromorphic workloads.

2.1.3 Neuromorphic Software Ecosystems

In the software domain, the development of training and learning algorithms for neuromorphic computing involves a blend of both machine learning and non-machine learning methodologies. Different categories of software frameworks have emerged to support these efforts. Some, such as *snnTorch*, *SpikingJelly* and *GeNN*, are primarily designed to emulate the behavior of SNNs on conventional hardware like GPUs. These tools enable rapid prototyping, experimentation and validation of spiking architectures in a familiar deep learning environment before hardware deployment. Other frameworks, such as *Lava*, *NxKernel* and *PyNN*, focus on bridging the gap between software and neuromorphic hardware. They provide an interface layer that allows SNN models to be mapped and deployed directly onto neuromorphic chips, supporting hardware-specific execution and optimization. Together, these software ecosystems play a crucial role in accelerating research and facilitating the transition from conceptual models to functional neuromorphic systems. For this thesis, three of the cited frameworks were used: *snnTorch* for training, *Lava* to better understand hardware functionality and *NxKernel* for hardware deployment.

snnTorch

snnTorch is an open-source Python library designed to extend PyTorch for the simulation and training of spiking neural networks (SNNs) [40, 41]. It integrates with PyTorch's computational graph, allowing spiking neurons to be treated as recurrent units and enabling gradient-based optimization through surrogate gradient methods. *snnTorch* supports several options for the surrogate function, such as sigmoid or arctangent, allowing flexible approximation of the non-differentiable spike function during training. This design allows users already familiar with deep learning frameworks to experiment with biologically inspired spiking models without changing their development workflow.

The framework introduces differentiable spiking neurons that can be trained through backpropagation despite the non-differentiable nature of spike events. Common neuron models include the first-order LIF (*snn.Leaky*) and the second-order LIF (`snn.Synaptic`), which accounts for synaptic conductance. Each model exposes a set of configurable parameters, such as membrane potential, threshold voltage, time constants, reset mechanism and optional reset delay. Many of these parameters can be made learnable, allowing fine-grained control over neuron dynamics, including firing rate, leakiness and refractory behavior. This flexibility enables the customization of network responses to suit different tasks or temporal patterns. In snnTorch, synaptic connections are implemented as `nn.Linear` layers in PyTorch, allowing efficient computation and seamless integration into the framework.

snnTorch also offers a set of loss functions and monitoring tools tailored for SNNs, enabling optimization based on membrane potentials, spike counts or spike timing. It is particularly suited for rapid prototyping, providing a bridge between conventional deep learning methods and spiking computation. As a simulation-based framework, it focuses on software implementations and does not directly map to neuromorphic hardware, though it serves as an effective tool for developing and testing models before hardware deployment, since it runs on GPU, making these steps faster.

Lava

Lava [42, 5] is Intel's open-source software framework for neuromorphic computing, developed to facilitate the design, simulation and deployment of neuro-inspired applications on both conventional and neuromorphic hardware, specifically on Loihi 2. Its architecture is platform-agnostic, modular and composable, allowing developers to integrate algorithmic contributions from multiple groups and build hierarchical abstractions to make neuromorphic programming accessible. Lava promotes progress in the field by providing a, professionally developed software foundation.
At its core, Lava introduces the concept of *processes*, which are stateful components that communicate asynchronously via event-based message passing. Processes encapsulate internal state, input/output ports and behavioral models and are organized hierarchically to enable scalable, parallel execution. Messages can carry payloads ranging from single-bit spikes to buffered packets of arbitrary size.
The low-level *Magma* interface maps high-level process abstractions to hardware-specific primitives, handling compilation, execution and profiling of neural networks. Magma supports cross-platform simulation, allowing models to be prototyped on conventional processors before deployment on neuromorphic hardware. It also provides profiling tools to estimate performance and energy consumption across targeted platforms.
Lava supports offline training through tools such as SLAYER, enabling event-driven neural networks to be trained via backpropagation and integrated with other processes. Furthermore, it is fully extensible, offering potential interfaces to third-party frameworks including ROS, YARP, TensorFlow, PyTorch and Nengo, allowing applications to span heterogeneous systems. In addition to the open-source components of the Lava framework, Intel has made the proprietary part, which provides support for Loihi 2, available to members of the INRC.

NxKernel

NxKernel is Intel's proprietary software stack, serving as an intermediate-level neuromorphic programming interface within the Lava ecosystem, available to members of the Intel Neuromorphic Research Community. It is specifically designed for Loihi 2 neuromorphic hardware and allows for more fined control of specific features of the hardware.

2.2 Recurrent neural networks (RNNs)

RNNs [43, 44] have been widely used for pattern recognition tasks involving temporal data. Unlike feedforward networks, which propagate information in a single direction, RNNs maintain a hidden state that is updated at each time step, allowing the network to retain information about previous inputs. This feedback mechanism enables RNNs to capture dependencies across time and effectively model sequential data.
However, classical RNNs suffer from difficulties in learning long-term dependencies due to vanishing and exploding gradients, which limits their ability to capture complex patterns in long sequences [45]. To address these issues, gated variants such as Long Short-Term Memory (LSTM) [46] networks and Gated Recurrent Units (GRUs) [47, 48] been introduced, incorporating mechanisms to control the flow of information across time steps. RNNs and their variants have found broad applications in natural language processing [49, 50], speech recognition [51] and time-series prediction [52]. Their structure and memory capabilities have also inspired the development of spiking recurrent networks, which aim to combine the benefits of RNNs with the efficiency and event-driven nature of neuromorphic systems.

2.2.1 Long Short-Term Memory

One of the most successful recurrent architecture is the LSTM [46] which, as the name suggests, keeps information about the input data in both a short-term memory unit, the *hidden state*, as well as a long-term memory unit, the *cell state*, to keep relevant information about past iterations. This allows the network to retain relevant information across long sequences, effectively addressing the vanishing and exploding gradient problems that typically hinder standard RNNs. It does so by making use of three different gates (Figure 2.6): the forget gate f_t which controls the longevity of information inside the cell state c_{t-1} by determining what to keep and what to discard from the previous timestep in varying proportions, the input gate i_t that, along with an assisting layer g_t, regulates the information entering the unit and, finally, the output gate that makes use of the previous hidden state h_{t-1} and the newly calculated cell state c_t to produce a new hidden state h_t. At each iteration, both memory units are updated to produce, in the end, a final output which is subsequently utilized to classify the input received. The process of

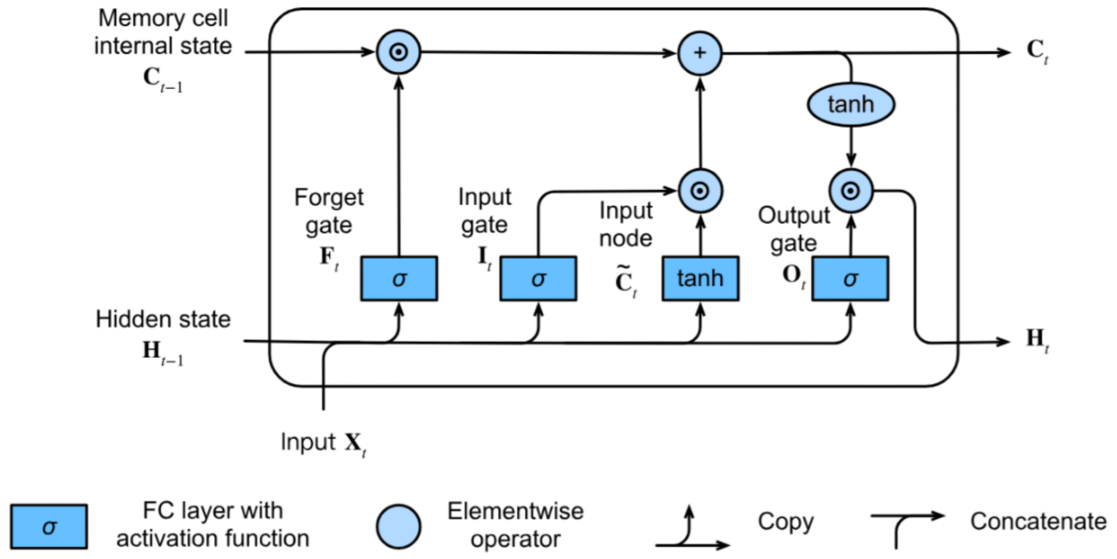


Figure 2.6: Architecture of an LSTM unit illustrating its internal structure: input, forget and output gates and the transformations that update the cell state and the hidden state.

a standard LSTM can be represented by the following equation:

$$\mathbf{F}_t = \sigma(\mathbf{W}_{hf} \cdot \mathbf{H}_{t-1} + \mathbf{W}_{if} \cdot \mathbf{X}_t), \tag{2.1}$$

$$\mathbf{I}_t = \sigma(\mathbf{W}_{hi} \cdot \mathbf{H}_{t-1} + \mathbf{W}_{ii} \cdot \mathbf{X}_t), \tag{2.2}$$

$$\tilde{\boldsymbol{C}}_t = \tanh(\mathbf{W}_{h\tilde{c}} \cdot \mathbf{H}_{t-1} + \mathbf{W}_{i\tilde{c}} \cdot \mathbf{X}_t), \tag{2.3}$$

$$\mathbf{O}_t = \sigma(\mathbf{W}_{ho} \cdot \mathbf{H}_{t-1} + \mathbf{W}_{io} \cdot {}_t), \tag{2.4}$$

$$\mathbf{C}_t = \mathbf{F}_t \cdot \mathbf{C}_{t-1} + \mathbf{I}_t \cdot \tilde{\boldsymbol{C}}_t, \tag{2.5}$$

$$\mathbf{H}_t = \mathbf{O}_t \cdot \tanh(\mathbf{C}_t). \tag{2.6}$$

Overall, the versatility of LSTMs in modeling complex temporal dependencies makes them suitable for a wide range of sequential data tasks. This strong capability to capture both short- and long-term patterns motivates their reinterpretation in spiking neural networks, where similar temporal dynamics can be exploited in a more energy-efficient, event-driven framework.

2.3 Neural Network Intelligence (NNI)

Neural Network Intelligence (NNI) [53] is an open–source toolkit developed by Microsoft to support automated hyperparameter optimisation and neural architecture search. Its design enables the definition of complex search spaces and the evaluation of candidate configurations through an external training script.

NNI operates by orchestrating a sequence of experiments in which different combinations of hyperparameters are selected according to a chosen optimisation strategy. Once a configuration is proposed, NNI launches the user–defined training procedure, collects the performance metric produced by the model and updates its internal tuner to refine the search. This process allows an efficient exploration of both continuous and categorical parameters, and supports a wide range of optimisation methods, including random sampling, Tree–structured Parzen Estimators (TPE), Bayesian optimisation, evolutionary algorithms, simulated annealing and more advanced multi–fidelity approaches.

A practical strength of NNI is the level of control it offers on the evaluation objective. The user can specify which metric the optimiser should target — such as validation accuracy, validation loss, test accuracy or any custom score — and NNI will guide the search accordingly. Once the search space and the reporting instructions are defined, the platform automatically manages the entire optimisation loop: proposing new hyperparameter combinations, executing the corresponding training runs and updating the tuner based on the obtained results. This makes NNI particularly suitable for analysing spiking architectures, where the behaviour of the model can be highly sensitive to parameters such as decay constants, thresholds or learning rates. In this work, NNI with the simulated annealing optimisation method was employed to explore the sensitivity of the proposed Spiking LSTM models and to identify stable and well-performing hyperparameter configurations.

2.4 Datasets Overview

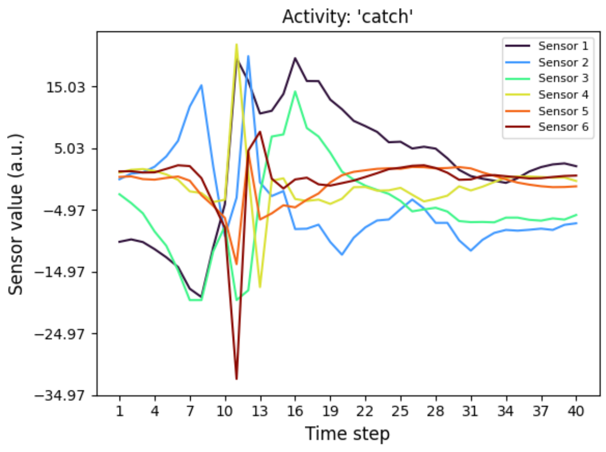
2.4.1 Human Activity Recognition (HAR)

A widely known classification problem is Human Activity Recognition (HAR) [23], which consists of a series of time-dependent signals, typically collected by smart devices containing sensors that perform body monitoring. In recent years, such personal and non-invasive devices have become more and more common in our everyday life, increasing the availability of data, spanning a variety of human life aspects such as healthcare, sports and surveillance. As a result, this task's popularity has increased in the research field, inspiring many works revolving around Convolutional Neural Networks (CNN) and RNN.

In 2019, the Wireless Sensor Data Mining (WISDM) Lab at Fordham University released the WISDM Smartphone and Smartwatch Activity and Biometrics Dataset [24], designed to capture human motion through the accelerometer and gyroscope of both devices. The dataset contains data collected from 51 subjects performing 18 different activities (Figure 2.7a), ranging from ambulatory movements such as walking, jogging and climbing stairs, to hand-centric tasks like brushing teeth, writing or eating.

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 ball
13	catch tennis ball
14	dribbling basketball
15	writing
16	clapping
17	folding clothes

(a) Complete list of activities in the WISDM dataset



(b) Random sample from WISDM dataset: six sensor channels (accelerometer x, y, z; gyroscope x, y, z)

Figure 2.7: Overview of the WISDM dataset. The table lists all recorded activities, while the plot shows an example of the time-series data collected from one subject.

Each activity was recorded for three minutes with a sampling frequency of 20 Hz, yielding over 15 million samples. Data were gathered simultaneously from the smartphone placed in the participant's pocket and the smartwatch worn on the dominant hand, ensuring synchronized multimodal acquisition. Every record includes six sensor readings (Figure 2.7b) — three accelerations (x, y, z axes) and three angular velocities — tagged with both the activity label and the subject identifier, allowing the dataset to be used not only for activity classification but also for biometric identification.

It offers a wide range of activities, with balanced distribution of the 15,630,26 total samples between classes, ranging from a contribution of 5.3% to 5.8% each. The dataset is often processed using a sliding window approach to segment the continuous time series into shorter fixed-length sequences suitable for machine learning models. A typical configuration employs 2-second windows (40 timesteps), each containing the six sensor channels as input features. This representation facilitates the training of temporal models such as recurrent and/or spiking neural

Number of subjects	51
Number of activities	18
Minutes collected per activity	3
Sensor polling rate	20Hz
Smartphone used	Google Nexus 5/5x or Samsung Galaxy S5
Smartwatch used	LG G Watch
Number raw measurements	15,630,426

Table 2.2: Summary information of the WISDM dataset. [54]

networks, which are capable of capturing temporal dependencies and dynamic state evolution across timesteps.
Thanks to its time-dependent nature, the WISDM dataset represents a suitable benchmark for evaluating models capable of learning temporal patterns, where temporal coding and dynamic state representation play a crucial role. A summary of the dataset key features can be find in Table 2.2.

2.4.2 Spiking Heidelberg Digits (SHD)

The Spiking Heidelberg Digits (SHD) [55] dataset is an audio-based classification benchmark specifically designed for event-driven neural computation. It consists of 10,420 recordings of spoken digits from 0 to 9, pronounced in both English and German by twelve different speakers, two of whom appear exclusively in the test set. The dataset is split into 8,332 training samples and 2,088 test samples, with no dedicated validation subset.

Each audio recording is converted into a spiking representation through *Lauscher*, a biologically inspired artificial cochlea model. This transformation produces 700 input channels, each representing a distinct frequency band and associated auditory neuron population. The resulting spike trains capture fine-grained temporal and spectral dynamics, closely mimicking the early auditory processing observed in the biological cochlea.
Unlike conventional frame-based audio datasets, SHD provides asynchronous spike events rather than continuous-valued spectrograms, making it particularly suitable for testing models that process temporally precise information, as spiking neurons naturally respond to discrete events. Each spike is typically binned with a temporal resolution of 1 ms, allowing networks to capture rapid temporal dependencies. This property allows evaluating a network's ability to extract and encode temporal dependencies from sparse, event-based data, an ability that is especially relevant for spiking recurrent architectures.
The duration of individual audio snippets varies, generally ranging from 1 to 2 seconds, reflecting natural speaker variability. Furthermore, the inclusion of

recordings in both English and German introduces phonetic variability, providing an additional challenge for models to generalize across speakers and languages.

The current state-of-the-art on the SHD dataset is summarized in Table 2.3. The upper section reports results from SNNs, while the lower section includes the non-spiking LSTMs for comparison with conventional ANNs. The dataset is commonly used as a benchmark for evaluating event-driven architectures and for comparing spiking networks to traditional ANN solutions in audio classification tasks [55].

Table 2.3: Comparison of published results on the SHD dataset.

	Publication	**Accuracy (%)**	**Network**
1	Sun et al. (2025) [56]	96.26±0.08	Parameter-free attention for delay SNNs
2	Baronig et al. (2024) [57]	95.8±0.6	RSNN with adaptive LIF neurons and symplectic-Euler discretization
3	Hammouamri et al. (2023) [58]	95.1±0.3	Fully connected SNN with learned delays
4	Bittar and Garner (2022) [59]	94.6	RSNN with adaptation
5	Nowotny et al. (2025) [60]	93.5±0.7	RSNN with delay line input and augmentations
6	Mészáros et al. (2025) [61]	93.2	RSNN with delay learning
7	Sun et al. (2023) [62]	92.45	Feed-forward SNN with adaptive axonal delays
8	Yu et al. (2022) [63]	92.4	Feed-forward SNN with spatio-temporal filters and attention
9	Yao et al. (2021) [64]	91.1	RSNN with temporal attention
10	D'Agostino et al. (2023) [65]	90.1 / 87.6	Feed-forward SNN with random dendritic delays (sim/hardware)
11	Yin et al. (2020) [66]	84.4	RSNN with adaptation
12	Rossbroich et al. (2022) [67]	83.5±1.5	Recurrent convolutional SNN with fluctuation-driven init
13	Cramer et al. (2020) [68]	83.2±1.3	RSNN with data augmentation and noise injection
14	Perez-Nieves et al. (2021) [69]	82.7±0.8	RSNN with heterogeneous time constants
15	Cramer et al. (2020) [70]	71.4±1.9	RSNN
16	Cramer et al. (2020) [71]	48.1±1.6	Feed-forward SNN (single hidden layer)
17	Schöne et al. (2024) [72]	95.9±0.9	Event-based linear state space model
18	Cramer et al. (2020) [73]	85.7±1.3	LSTM

Chapter 3

From ANN to SNN

3.1 Reinterpretation of the LSTM

Numerous attempts have been made to reinterpret conventional neural architectures within a spiking framework [74, 75, 76, 77]. These works show that replacing continuous activations with event-based computations can substantially reduce the computational cost while preserving the core functionality of the original models. Several studies, such as [78] and [79], demonstrate that spiking implementations can achieve comparable performance to their non-spiking counterparts, often with considerably lower energy consumption.

The transformation of LSTM networks into spiking counterparts has been explored through a variety of approaches in recent years. Early work, such as [80], focused on training recurrent spiking networks by leveraging firing-rate dynamics to reproduce complex temporal patterns. Although not directly aimed at converting LSTMs, this line of research demonstrated that recurrent SNNs can implement structured temporal computations, offering a foundation for more specialized architectures. A more direct neuromorphic reinterpretation was proposed in [81], which introduced a framework for converting the LSTM cell into a fully spiking unit. In their design, spikes are used as inputs and the classical activation functions in the gates are replaced by spiking nonlinearities, while preserving the functional roles of input, forget and output gates. The authors also developed a dedicated backpropagation method that enables the network to reach performance levels comparable to conventional LSTMs, while ensuring that the internal cell state remains bounded within the spiking regime.

Progressing from these prior contributions, the core idea of this Thesis is a neuromorphic reinterpretation of the LSTM is to express its computational structure in terms of neuron populations and synaptic interactions, while preserving the original model's functional principles. In the proposed spiking LSTM (sLSTM), each gate is maintained conceptually, but reimplemented using spiking neuron populations that replicate their roles within the network. Depending on the specific approach, traditional activation functions may either be retained or replaced by spiking dynamics, leading to models that vary in their level of biological plausibility and hardware compatibility.

3.1.1 Basic Implementation

The first architecture presented in this section is the result of converting the fundamental units of Long Short-Term Memory into populations of neurons, the architecture is displayed in Figure 3.1. Similar to traditional LSTMs, the central idea stands in the cell state, denoted as C_t (where t stands for *time*), which serves as a channel and manager of information flow between units (long-term memory), while the hidden state h_t captures the output and feeds it back into the network at the next timestep. This is achieved through the collaboration of the various gates and layers typical of this kind of model: the forget gate F_t, the input gate I_t, the candidate gate $\tilde{C}_t$ and, ultimately, the output gate O_t. Since both the hidden state and the input are represented as spikes entering the gate, a neuron population was introduced before the output stage. This population, H_t, takes as input the state calculated by the cell and passed through the tanh function (Hin_t), which represents the combined information from previous time steps. The neuron population then processes this input and generates spikes corresponding to the hidden state, denoted as $H_{t,spk}$, thus ensuring that the dynamics of the LSTM are preserved in a spiking neural network framework. More specifically, given a set of spiking inputs at $\{x_{1,spk},\ x_{2,spk},\ ...,\ x_{T,spk}\}$, where T is the total number of timesteps, the gates and states, at time t, are characterized as follows:

$$\mathbf{f}_t = \sigma(f((\mathbf{W}_{hf} \cdot \mathbf{h}_{t-1,spk} + \mathbf{W}_{if} \cdot \mathbf{x}_{1,t,spk})_{spk})), \tag{3.1}$$

$$\mathbf{i}_t = \sigma(f((\mathbf{W}_{hi} \cdot \mathbf{h}_{t-1,spk} + \mathbf{W}_{ii} \cdot \mathbf{x}_{2,t,spk})_{spk})), \tag{3.2}$$

$$\tilde{\boldsymbol{c}}_t = \tanh(f((\mathbf{W}_{h\tilde{c}} \cdot \mathbf{h}_{t-1,spk} + \mathbf{W}_{i\tilde{c}} \cdot \mathbf{x}_{3,t,spk})_{spk})), \tag{3.3}$$

$$\mathbf{o}_t = \sigma(f((\mathbf{W}_{ho} \cdot \mathbf{h}_{t-1,spk} + \mathbf{W}_{io} \cdot \mathbf{x}_{4,t,spk})_{spk})), \tag{3.4}$$

$$\mathbf{C}_t = \mathbf{I}_t \cdot \mathbf{C}_{t-1} + \mathbf{I}_t \cdot \tilde{\boldsymbol{C}}_t, \tag{3.5}$$

$$\mathbf{Hin}_t = \mathbf{O}_t \cdot \tanh(\mathbf{C}_t). \tag{3.6}$$

where the notation $_{spk}$ is used to indicate the spiking value and the $f(\cdot)$ corresponds to a mean function calculated as $\frac{n_{spikes}}{n_{pop}}$ (the *pop* represents the number of neuron of the population, as explained later, and depends on the gate).
In the above equations, all the activation functions from the original definitions of LSTM are retained. These functions receive the average of the spikes on their respective populations as input. This design choice ensures that the input of the activation functions varies in a range between 0 and 1, in order to allow the function

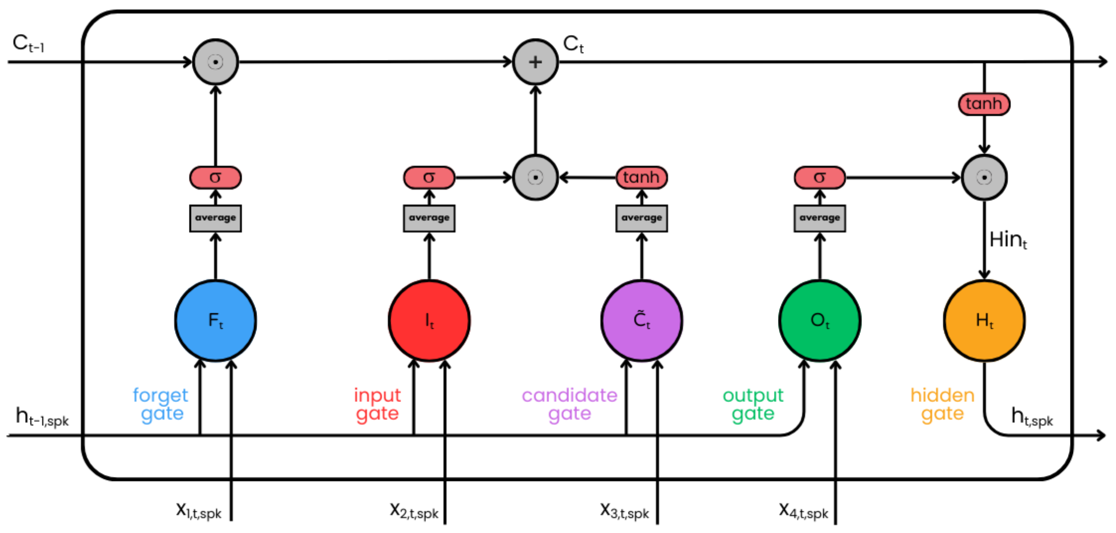


Figure 3.1: SLSTM basic implementation architecture. Each gate is converted into a population of LIF neurons with its own population size and specific values for *beta* and *threshold*

to produce a more varying output, rather than sticking to the same two values that it would have otherwise produced by receiving only spikes and non-spikes.
The last key aspect is the use of population coding (*pop*) within each cell gate. In a standard LSTM, the output dimension of each gate corresponds to the hidden size. To replicate this behavior in a spiking context, each single hidden dimension is represented by a number of neurons equal to the specific population size of the gate (i.e. *population size* = *gate population* * *hidden size*). The mean is then computed over the population size. Without this technique, averaging the outputs would result in a single value for the entire vector, thereby losing the granularity of information.

3.1.2 Fully Spiking implementation

The basic structure of this implementation resembles the previous one. However, the primary objective here is to design a fully spiking LSTM cell architecture. Unlike the previous approach, which still relied on some traditional activation functions, this version aims to leverage the inherent capabilities of spiking neural networks. This was achieved by replacing all activation functions through populations of neurons and their interaction.
A first attempt consisted in replacing all activation functions with simple LIF neuron populations. While this approach aligned with the goal of leveraging the intrinsic dynamics of spiking neurons, it showed limited potential: in particular,

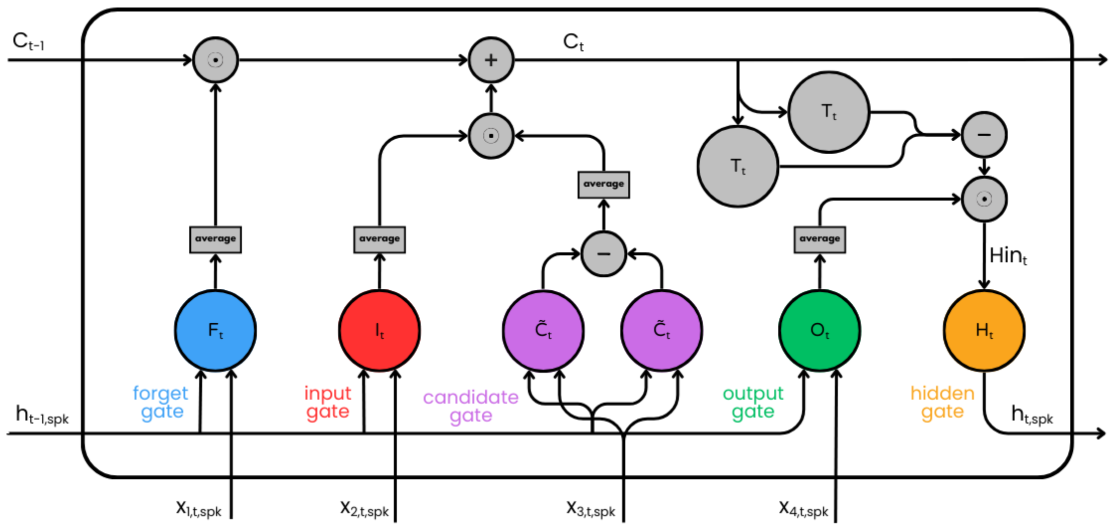


Figure 3.2: SLSTM Fully Spiking implementation architecture. The activation functions are replaced by neuron populations.

it failed to reproduce the expressive range of the tanh function, as the resulting activations could not take on negative values. These limitations motivated the development of an additional, more refined version described in the following.
The sigmoid function, typically used in forget and input gates, is approximated by the average firing rate (number of spikes divided by the population size) within a population of neurons. This substitution worked in practice because both the sigmoid function and the average firing rate produce output values between 0 and 1, representing a "percentage" of activation. The *tanh* function plays a crucial role in the LSTM architecture, allowing selective addition or removal of information from the hidden state due to its output range from -1 to 1. To mimic this behaviour with only spikes the solution draws inspiration from the nature of the human brain and nervous system. Two distinct populations of neurons were implemented, one excitatory and the other inhibitory. By calculating the difference between the outputs and averaging by the population size, it was possible to generate a range of output values equivalent to the *tanh*, ensuring the network's proper functioning.

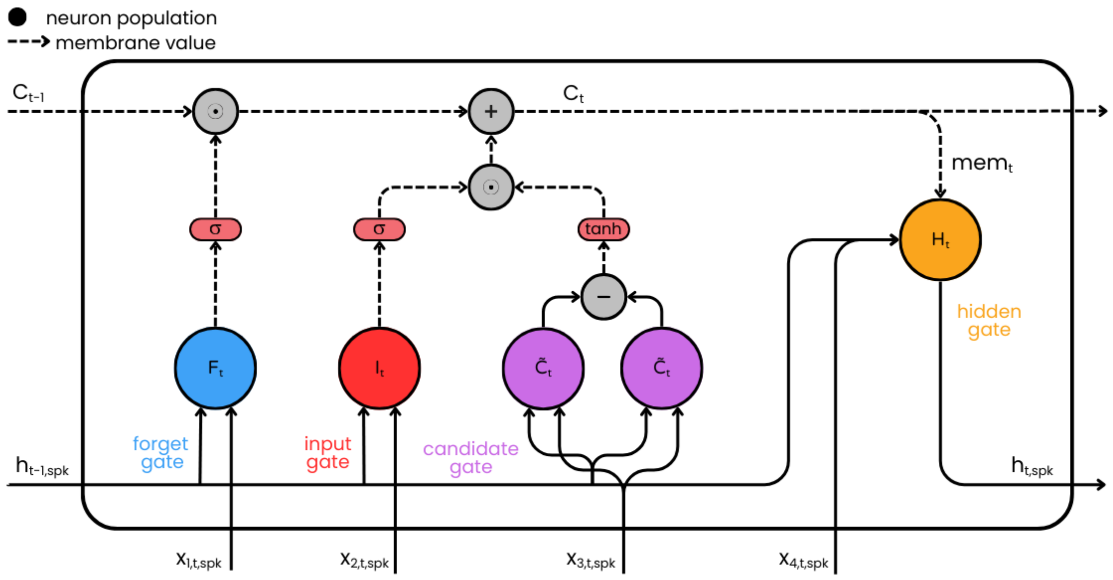


Figure 3.3: SLSTM Membrane-Based Implementation architecture. The activation exploits the membrane potential of the gates neuron populations.

The conversion of the equation is straightforward:

$$\begin{aligned}
\mathbf{f}_t &= f((\mathbf{W}_{hf} \cdot \mathbf{h}_{t-1,spk} + \mathbf{W}_{if} \cdot \mathbf{x}_{1,t,spk})_{spk}), && (3.7)\\
\mathbf{i}_t &= f((\mathbf{W}_{hi} \cdot \mathbf{h}_{t-1,spk} + \mathbf{W}_{ii} \cdot \mathbf{x}_{2,t,spk})_{spk}), && (3.8)\\
\tilde{\mathbf{c}}_t &= f\Big((\mathbf{W}^{+}_{h\tilde{c}} \cdot \mathbf{h}_{t-1,spk} + \mathbf{W}^{+}_{i\tilde{c}} \cdot \mathbf{x}_{3,t,spk})_{spk} && (3.9)\\
&\quad - (\mathbf{W}^{-}_{h\tilde{c}} \cdot \mathbf{h}_{t-1,spk} + \mathbf{W}^{-}_{i\tilde{c}} \cdot \mathbf{x}_{3,t,spk})_{spk}\Big), && (3.10)\\
\mathbf{o}_t &= f((\mathbf{W}_{ho} \cdot \mathbf{h}_{t-1,spk} + \mathbf{W}_{io} \cdot \mathbf{x}_{4,t,spk})_{spk}), && (3.11)\\
\mathbf{C}_t &= \mathbf{f}_t \cdot \mathbf{C}_{t-1} + \mathbf{i}_t \cdot \tilde{\mathbf{c}}_t, && (3.12)\\
\mathbf{Hin}_t &= \mathbf{o}_t \cdot f((\mathbf{W}^{+}_{C} \cdot \mathbf{C}_t - \mathbf{W}^{-}_{C} \cdot \mathbf{C}_t)_{spk}). && (3.13)
\end{aligned}$$

where $f(\cdot)$ is again $\frac{n_{spikes}}{n_{pop}}$, as in the basic implementation. The substitute of the *tanh* function can be seen as a small architecture composed by two distinct linear layers with two different populations of Leaky neurons and a final operation in which the average is computed on the subtraction between the spikes, as displayed in the general architecture Figure 3.2.

3.1.3 Membrane-Based Implementation

The fundamental distinction between RNN and SNN lies in the notion of time. In LSTMs, time is encoded within the memory cell state, which retains information about the cell's history. In contrast, in SNNs both the input and output of spiking neurons are temporally encoded using sparse spiking events occurring over a specific timeframe. These events are captured and stored through the membrane potential.

For this membrane-based implementation, the architecture was changed by leveraging the similarity between the LSTM memory cell state and the membrane potential in spiking neurons. The whole LSTM cell was conceptualized as a single neuron, where the memory state represents its membrane potential (as shown in Figure 3.5)). This membrane potential is internally updated following the classical operations of the gates in an LSTM architecture. It is then used as the membrane potential for the hidden gate, which is responsible for outputting the spikes used for the classification task. To align with this reasoning, the encoding block was replaced with a single fully connected layer followed by a single population layer, ensuring that the same input is fed to all gates.

The logic behind the implementation is similar to the LSTM, where the membrane potential is modified through gates:

- **Forget gate**: Determines the percentage of the current membrane potential to retain. Given the input and the hidden state, it processes them through a population and returns the sigmoid value of its *membrane potential.*
- **Input gate**: Determines the percentage of the candidate membrane potential to retain. Same operations of the forget gate.
- **Candidate gate**: Proposes an amount of potential to add to the actual value of the membrane. Given the input and the hidden state, it processes them through a population and returns the hyperbolic tangent value of its *membrane potential.*
- **Output gate**: It is responsible of the computation of the current output that goes into the hidden gate. Therefore, it does not involve a population of neurons.

The formulas regulating this process are as follows:

$$\mathbf{f}_t = \sigma((\mathbf{W}_{hf} \cdot \mathbf{h}_{t-1,spk} + \mathbf{W}_{if} \cdot \mathbf{x}_{t,spk})_{mem}), \tag{3.14}$$
$$\mathbf{i}_t = \sigma((\mathbf{W}_{hi} \cdot \mathbf{h}_{t-1,spk} + \mathbf{W}_{ii} \cdot \mathbf{x}_{t,spk})_{mem}), \tag{3.15}$$
$$\tilde{\boldsymbol{c}}_t = \tanh((\mathbf{W}_{h\tilde{c}} \cdot \mathbf{h}_{t-1,spk} + \mathbf{W}_{i\tilde{c}} \cdot \mathbf{x}_{t,spk})_{mem}), \tag{3.16}$$
$$\mathbf{o}_t = \mathbf{W}_{ho} \cdot \mathbf{h}_{t-1,spk} + \mathbf{W}_{io} \cdot \mathbf{x}_{t,spk}, \tag{3.17}$$
$$\mathbf{mem}_t = \mathbf{f}_t \cdot \mathbf{mem}_{t-1} + \mathbf{i}_t \cdot \tilde{\boldsymbol{c}}_t, \tag{3.18}$$
$$\mathbf{h}_t = spk(\mathbf{o}_t, \mathbf{mem}_t). \tag{3.19}$$

Here, the notation $_{mem}$ is used to indicate the membrane value, and the spiking function $spk(o_t, mem_t)$ indicates whether an output spike occurs based on the membrane potential, mem_t, and the output gate's weighted sum.

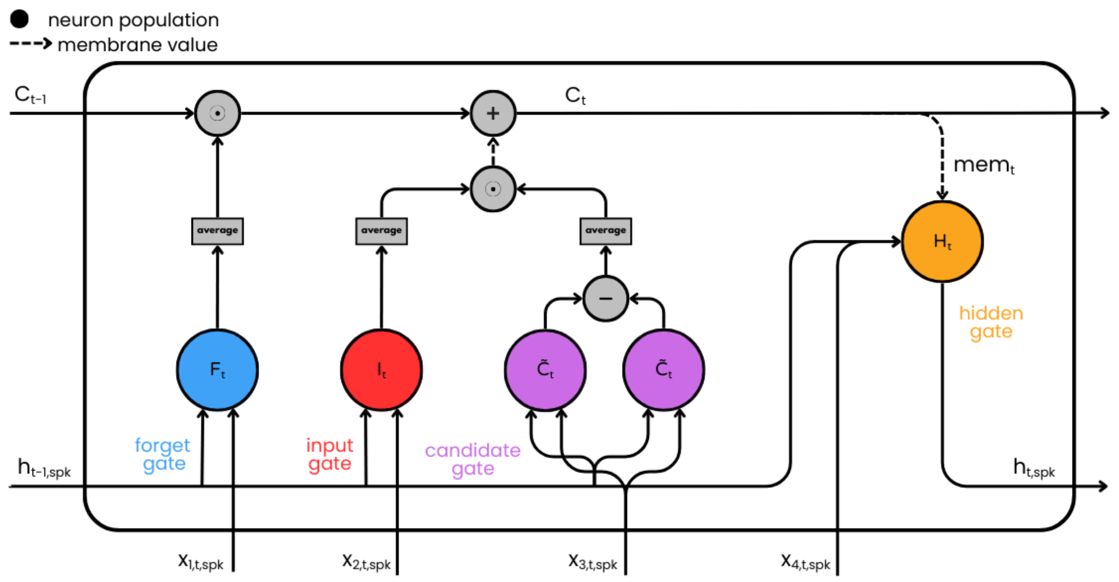


Figure 3.4: SLSTM Spiking implementation architecture. The activation functions are replaced by neuron populations.

In contrast to traditional LSTM implementations, the output gate in this approach skips activation functions and directly passes the weighted input and hidden state to an output neuron. This neuron receives the membrane potential value calculated by the rest of the architecture, maintaining the model's dynamic behavior and complexity.

In summary, the neuron receives the input x_t and the previous hidden state $spk(h_{t-1})$, encoded as spikes, as parameters and outputs both the updated hidden state $spk(h_t)$ and the neuron's internal state mem_t. This neuron can be seen as a population itself as it emulates the behavior of multiple neurons.

3.1.4 Spiking membrane

An additional variant of the *Membrane-Based Implementation* was explored to leverage the advantages of membrane dynamics while preserving a more neuromorphic structure. This architecture draws inspiration from the *Fully Spiking Implementation*, aiming to combine the strengths of both approaches.

In this modified architecture (Figure 3.4), the transformation of the sigmoid and tanh activation functions is retained, similar to the fully spiking version. Specifically, the sigmoid function is approximated by the mean of the output spikes, while the tanh function is replaced by the difference in the means of two populations of output spikes.

In this version, the cell state (C_t) is interpreted as the membrane potential (mem_t)

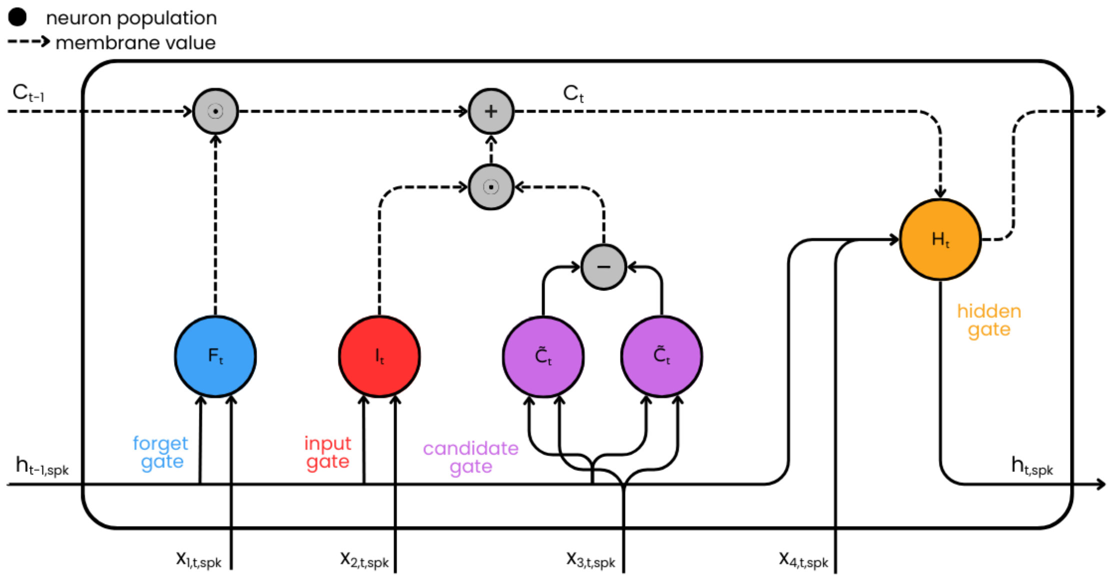


Figure 3.5: SLSTM Final implementation architecture. This architecture is the one deployed on hardware and used for all the experiments.

of the hidden neuron, following the same concept as in the original membrane-based implementation. This membrane potential is then assigned to the hidden population, which uses it as its internal state. The output gate provides the input current to this population and, based on the interaction between this input and the membrane potential, the hidden population generates its spiking output. The underlying logic of the model can be described with the following formulas:

$$\mathbf{f}_t = f((\mathbf{W}_{hf} \cdot \mathbf{h}_{t-1,spk} + \mathbf{W}_{if} \cdot \mathbf{x}_{t,spk})_{spk}) \tag{3.20}$$

$$\mathbf{i}_t = f((\mathbf{W}_{hi} \cdot \mathbf{h}_{t-1,spk} + \mathbf{W}_{ii} \cdot \mathbf{x}_{t,spk})_{spk}) \tag{3.21}$$

$$\tilde{\mathbf{c}}_t = f\Big((\mathbf{W}^{+}_{h\tilde{c}} \cdot \mathbf{h}_{t-1,spk} + \mathbf{W}^{+}_{i\tilde{c}} \cdot \mathbf{x}_{t,spk})_{spk} \tag{3.22}$$

$$- (\mathbf{W}^{-}_{hg} \cdot \mathbf{h}_{t-1,spk} + \mathbf{W}^{-}_{ig} \cdot \mathbf{x}_{t,spk})_{spk}\Big) \tag{3.23}$$

$$\mathbf{o}_t = \mathbf{W}_{ho} \cdot \mathbf{h}_{t-1,spk} + \mathbf{W}_{io} \cdot \mathbf{x}_{t,spk} \tag{3.24}$$

$$\mathbf{mem}_t = \mathbf{f}_t \cdot \mathbf{mem}_{t-1} + \mathbf{i}_t \cdot \tilde{\mathbf{c}}_t \tag{3.25}$$

$$\mathbf{h}_t = spk((\mathbf{W}_{syn} \cdot \mathbf{o}_t), \mathbf{mem}_t) \tag{3.26}$$

3.1.5 Final Implementation of the Spiking LSTM

This final architecture brings the spiking membrane approach one step closer to the Membrane-Based Implementation, while addressing the practical limitations observed in the previous variants. Two aspects of the previous models were reconsidered: the use of population averages, which is not a natural or efficient operation in a spiking context, and the large number of neuron populations required to approximate each activation function. Both elements made the architecture heavy, difficult to scale and far from the constraints of neuromorphic hardware.

For this reason, the design integrates ideas from both previous attempts. As in the second version, the membrane potential of the gates is used directly as the signal driving the computation. At the same time, the activation functions are replaced following the strategy introduced in the spiking-membrane variant, without relying on population-level approximations. The dynamics are described as following:

$$\mathbf{f}_t = (\mathbf{W}_{hf} \cdot \mathbf{h}_{t-1,spk} + \mathbf{W}_{if} \cdot \mathbf{x}_{t,spk})_{mem}, \tag{3.27}$$
$$\mathbf{i}_t = (\mathbf{W}_{hi} \cdot \mathbf{h}_{t-1,spk} + \mathbf{W}_{ii} \cdot \mathbf{x}_{t,spk})_{mem}, \tag{3.28}$$
$$\tilde{\boldsymbol{c}}_t = (\mathbf{W}^{+}_{h\tilde{c}} \cdot \mathbf{h}_{t-1,spk} + \mathbf{W}^{+}_{i\tilde{c}} \cdot \mathbf{x}_{3,t,spk})_{mem} \tag{3.29}$$
$$- (\mathbf{W}^{-}_{h\tilde{c}} \cdot \mathbf{h}_{t-1,spk} + \mathbf{W}^{-}_{i\tilde{c}} \cdot \mathbf{x}_{3,t,spk})_{mem}, \tag{3.30}$$
$$\mathbf{o}_t = \mathbf{W}_{ho} \cdot \mathbf{h}_{t-1,spk} + \mathbf{W}_{io} \cdot \mathbf{x}_{t,spk}, \tag{3.31}$$
$$\mathbf{mem}_t = \mathbf{f}_t \cdot \mathbf{mem}_{t-1} + \mathbf{i}_t \cdot \mathbf{mem}_t, \tag{3.32}$$
$$\mathbf{h}_t = spk(\mathbf{o}_t, \mathbf{mem}_t). \tag{3.33}$$

The result is a more compact and hardware-oriented architecture in which each gate contributes through its membrane dynamics, preserving the functional flow of a classical LSTM while avoiding operations that are unnatural for spiking systems. This design retains the core computational principles of the original model and, at the same time, produces a structure that can be efficiently deployed on neuromorphic hardware.

3.2 Familiarization with NxKernel and Loihi 2 Neuron Mechanics

During the initial phase of this Thesis, substantial effort was devoted to developing a practical understanding of NxKernel and of the computational primitives exposed by the Loihi 2 architecture. The official tutorials and the example code provided by Intel played a crucial role in clarifying how the platform models neuronal dynamics and low-level neural computation.

Through progressive experimentation, it became possible to identify how Loihi 2 structures a neuron into its fundamental components such as constant registers, dendritic accumulator units and the memory elements responsible for storing intermediate states. This exploration also shed light on the micro-operations

executed at each simulation tick, including how inputs are integrated, how thresholds and resets are applied and how spike events are generated or suppressed depending on the chosen neuron configuration.
A deeper understanding was also gained regarding the connectivity model. In Loihi 2, synaptic communication is tightly coupled with the hardware's routing constraints and the process of defining weight matrices, fan-in conditions and addressable synaptic channels revealed both the flexibility and the boundaries imposed by the architecture. Working with various connectivity and weight configurations clarified the practical boundaries of Loihi 2's synaptic memory model, in particular, how the available bit-depth for weights, the supported data formats and the memory allocated per core determine both the number of incoming synapses that can be instantiated and the level of numerical precision they can carry.
These explorations collectively helped outline the expressive power of Loihi 2 as a neuromorphic substrate: while the hardware imposes structural constraints on neuron models and connectivity, it also allows precise low-level control over computation, enabling the construction of custom dynamical systems that go beyond standard LIF behavior. This familiarity with NxKernel ultimately provided the foundation necessary to implement the Spiking LSTM architecture, as designed for the purposes of this thesis.

3.3 snnTorch Implementation

The architecture described in Section 3.1.5 was initially implemented and tested in the snnTorch environment. Each gate of the LSTM (input, forget, output, candidate and the hidden gate) was built using `snn.Leaky` neurons connected through standard `nn.Linear` layers.

The `Leaky` neuron provides several configurable parameters. In this Thesis, all gates were created with the following neuronal settings:

- `beta`: the membrane decay factor, controlling how much of the previous membrane potential is retained at each timestep;
- `threshold`: the firing threshold;
- `learn_beta=True` and `learn_threshold=True`: enabling gradient-based adaptation of both decay and threshold;
- `reset_delay=False`: ensuring that the reset is applied immediately after a spike.

These settings offer a flexible spiking substrate where both the temporal dynamics (`beta`) and the excitability of the neuron (`threshold`) can adapt during training.

While these properties were common across all gates, the *reset mechanism* differed depending on the computational role of each gate. The default reset, `zero`, was used for the standard LIF neurons, for the hidden-state neuron and also for the forget and input gates. These gates must represent a fraction of information to retain or discard, the standard zero-reset is appropriate: it ensures that each spike reflects an instantaneous contribution, the more excited the neuron is the more it needs to discard, culminating in a spike whose zero-reset clears the membrane completely, ensuring that no residual current persists across timesteps. The `subtract` reset, that enables a progressive accumulation of current as it does not fully clear the membrane value when spiking, allowing the membrane potential to approximate a continuous-valued signal. This property makes it suitable for the candidate gate, whose role is to emulate the continuous tanh activation.

The model was trained on the HAR dataset, using a subset of seven classes, as in [82]. Training was performed with Backpropagation Through Time (BPTT), using `SF.ce_count_loss()` as the loss function and the Adam optimizer. The count-based cross-entropy loss provided by snnTorch was chosen, since it is particularly well suited for spiking architectures, as it aligns the training objective with the discrete nature of spike outputs: instead of relying on instantaneous membrane values, it evaluates class evidence through spike counts accumulated over the full sequence, making the gradient signal more stable and better matched to the model's computation.

3.4 Translation from snnTorch to NxKernel

Compared to the snnTorch implementation, the translation to NxKernel required a conceptual restructuring of the LSTM operations. In snnTorch, gate interactions and nonlinearities are computed across multiple layers and modules, whereas on Loihi 2 these operations must be reformulated in terms of neuron-level state transitions and event-driven updates. This mismatch required reorganizing the computation flow of the LSTM cell so that each operation could be executed as part of a neuron's internal update cycle. The final final result is illustrated in Figure 3.6.

In this process, two types of neurons were adapted in NxKernel. First, the Leaky neuron model was extended: *mem_lif* is a Leaky capable of outputting its membrane potential directly. Instead of producing only a binary spike, the neuron attaches its membrane value as a payload, making it available to downstream populations. Of this neuron two variants were made implementing both the zero and the subtractive-reset behavior compatible with the candidate gate. Second, a fully custom neuron type was defined to implement the LSTM cell itself, the `lstm_neuron`. This neuron integrates the core operations of the LSTM—gate

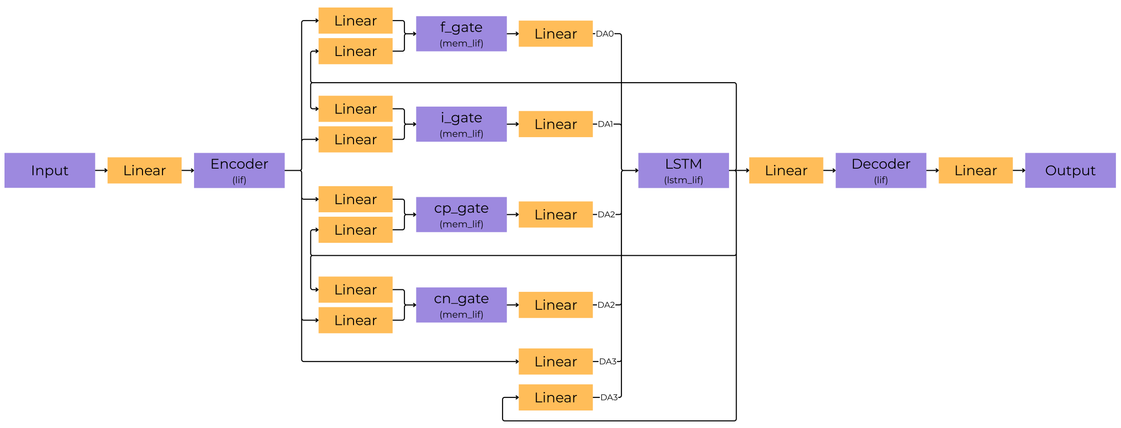


Figure 3.6: Diagram of the Spiking LSTM architecture implemented in NxKernel, showing the organization of the neuronal populations.

computation (see Algorithm 1), cell update and hidden-state emission. It receives as input the three gate signals, together with both the recurrent and external inputs. Because of this high degree of specialization, its behavior was implemented explicitly rather than derived from existing neuron models.

Algorithm 1 High-level description of the neuron update implemented in microcode

```
 1: ▷ Gate-based memory update
 2: Read the forget gate value $f$ and compute its contribution to the previous
    memory state: $m_f = f \cdot m$.
 3: Read the input gate value $i$.
 4: Read the candidate input $g$ and compute its contribution: $m_i = i \cdot g$.
 5: Update the internal memory: $m \leftarrow m_f + m_i$.
 6: ▷ Leaky integration and spike generation
 7: Read the incoming input current and apply the required scaling.
 8: Integrate the current into the membrane potential using standard LIF dynamics
    (leak + input + bias).
 9: if membrane potential exceeds threshold then
10:     Emit spike and reset membrane potential.
11: end if
```

3.5 Hardware-aware software simulation

To verify the correctness of the custom architecture and the dedicated neuron models, the first natural step was to perform a strict one-to-one comparison with snnTorch, both at the level of individual gates and for the entire LSTM module. This

process required identifying the appropriate scaling factors for weights, thresholds and the leakage parameter β, so that the behavior of each component would remain consistent across the two frameworks. Once each gate was validated in isolation, attention shifted to the full architecture and its dynamics over time.
During this phase, a fundamental discrepancy emerged between the execution model of Loihi 2 and the synchronous processing assumed in snnTorch: by default, Loihi 2 processes data in a pipelined fashion. At every timestep, the input is immediately forwarded to the next layer, allowing the hardware to process multiple timesteps in parallel. While this parallelism increases throughput, it introduces a latency between the computation of the future hidden state and its availability as input for the next timestep. Concretely, the hidden state produced at time t does not return to the gate populations within the same timestep, but only after the LSTM layer has completed its internal processing pipeline. This produces a misalignment between the input and hidden trajectories that cannot simply be corrected by inserting fixed delays. Two conceptual solutions were possible:

- Redefining the LSTM dynamics and consequently modifying its implementation in snnTorch to embrace Loihi's pipelined timing model; or
- Adapting the execution schedule on Loihi 2 to preserve the standard LSTM logic and match the synchronous behavior assumed during training.

To maintain architectural fidelity and avoid redefining the recurrent dynamics, the second strategy was chosen. The solution consisted in an *alternated execution scheme*, where even and odd layers, grouped as shown in Figure 3.7, are executed on alternating timesteps. This gives the hidden state sufficient time to be processed by the LSTM neuron population and ensures that its updated value is available as input exactly when needed. With this mechanism in place, the temporal alignment between Loihi 2 and snnTorch became consistent, enabling meaningful comparison between the two. Another source of discrepancy originated from a specific behavior in snnTorch related to `reset_delay`. Even when `reset_delay` is not enabled, if the membrane potential exceeds the threshold before the neuron update step, snnTorch resets the membrane immediately. In our LSTM implementation, however, the membrane potential fed to the recurrent neuron is the pre-computed input from the gates. This value can easily exceed the threshold depending on the gating operations, unintentionally triggering an early reset. Since this behavior is not present in the Loihi 2 implementation and is not desirable for the recurrent dynamics, the neuron's microcode was edited accordingly to ensure the correct functional correspondence. After resolving these two sources of mismatch (the pipeline-induced timing shift and the unintended membrane reset) the outputs of Loihi 2 and snnTorch aligned closely across all tested sequences, as depicted in Figure 3.8.

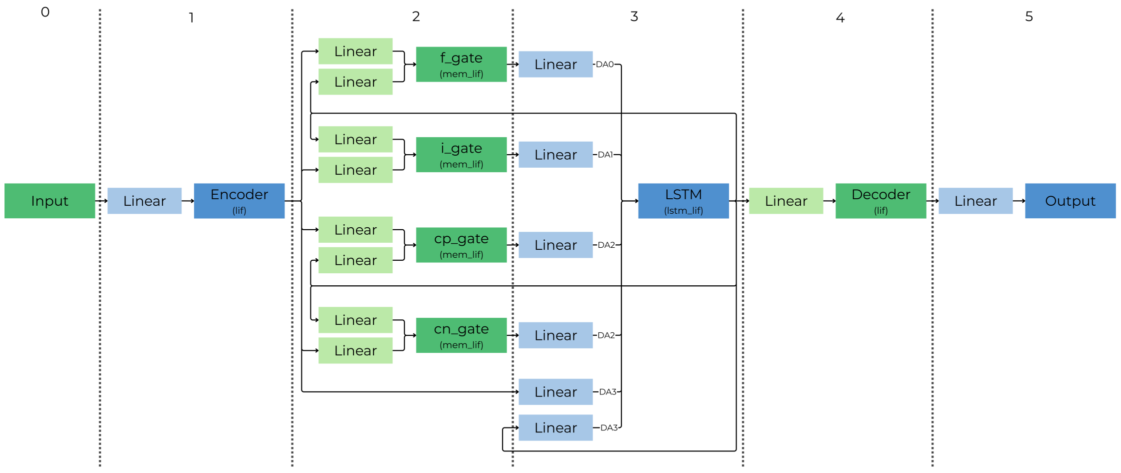
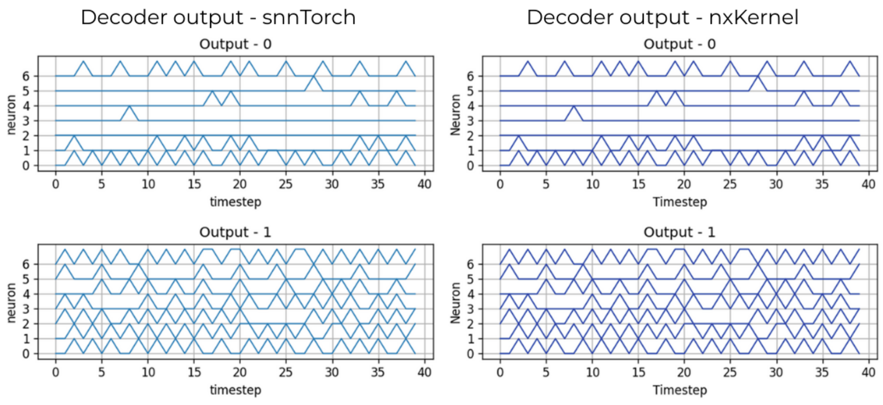


Figure 3.7: Schematic representation of the alternated execution strategy. Even- and odd-indexed layers are highlighted with distinct colors to indicate their staggered update schedule.

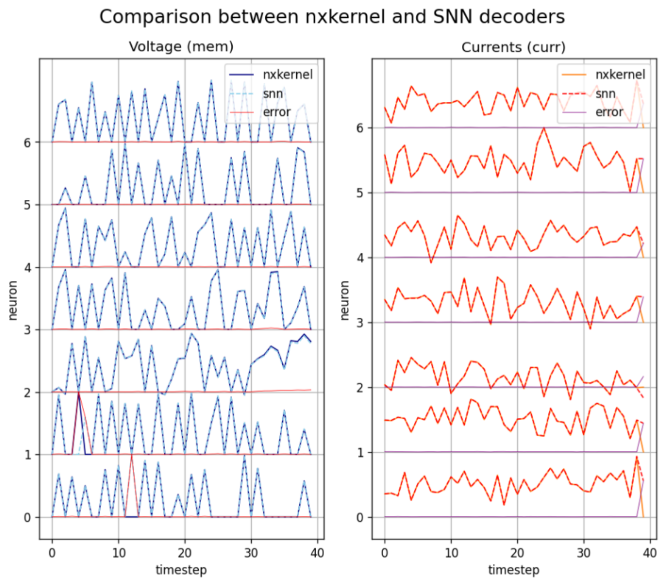
Because NxKernel does not provide training capabilities, training was carried out entirely in snnTorch, which provided a flexible and faithful software environment. The learned weights, β values and thresholds were then transferred to NxKernel for hardware execution.

3.5.1 Quantization

Since *snnTorch* was selected as the training framework, it was necessary to design a robust conversion pipeline to transfer trained parameters to *NxKernel* and obtain comparable performance on Loihi 2 hardware. Quantization plays a crucial role in this transition, since the two frameworks use different numerical representations — snnTorch relies on floating-point arithmetic, whereas NxKernel operates entirely with integers. Weights, thresholds and decay factors must be represented with limited precision while avoiding overflow on hardware.
For the leakage parameter β, the conversion is straightforward: in the microcode, β is encoded using 12 bits. Since decay values lie between 0 and 1, the βs produced by snnTorch are multiplied by 2^{12} and rounded to the nearest integer.
The quantization of weights and thresholds is less immediate. The threshold depends heavily on the quantization applied to the weights, as well as on an additional scaling factor introduced by an internal input shift performed by the neuron model. Thus, identifying an appropriate representation for the synaptic weights becomes the central step. The goal was to strike a balance between minimizing information loss from quantization, limiting the risk of overflow during



(a)



(b)

Figure 3.8: Validation of the custom NxKernel neuron model against its snnTorch reference. (a) Comparison of the output spike trains produced by the two implementations. (b) Comparison of the internal membrane potential and input current, confirming the numerical alignment of the two models.

computation and respecting the bit-width constraints of Loihi's synaptic memory. After several iterations, the following approach was adopted. The weights trained in snnTorch were scaled using the largest power of two (strictly less than 2^8) that allowed the weight with maximum absolute value in each matrix to be represented within the integer range from -127 to 127. After understanding how to handle parameter shifting between the two frameworks, a full pipeline for training and

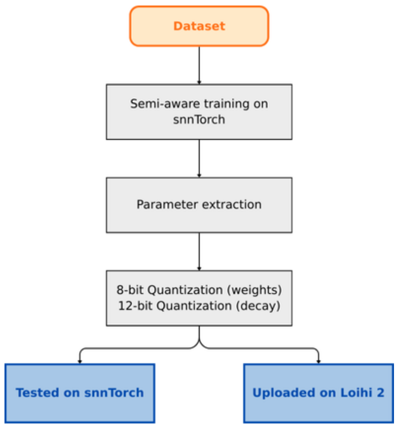

Dataset
Semi-aware training on snnTorch
Parameter extraction
8-bit Quantization (weights)
12-bit Quantization (decay)
Tested on snnTorch
Uploaded on Loihi 2

Figure 3.9: Overview of the pipeline used to transition from a snnTorch-trained SNN model to its implementation on Loihi 2 hardware.

testing was implemented (Figure 3.9). To anticipate hardware constraints, a *semi-quantized* training strategy was adopted: during training, only the membrane potentials were quantized at each timestep, allowing the model to adapt to the reduced numerical resolution typical of Loihi 2 while keeping the weights in floating point. After training, the final model was fully quantized by discretizing all parameters—weights, betas and thresholds—according to the bit-widths required by NxKernel.
This fully quantized version was then reloaded and evaluated directly in snnTorch to estimate how much performance degradation could be attributed purely to discretization.

3.6 Hardware Deployment

The board used in this thesis is an Oheo Gulch single-chip Loihi 2 (Figure 3.10) To deploy the Spiking LSTM directly on physical Loihi 2 hardware, setup process was carried out to enable communication between the local development machine and the Loihi 2 DevKit. The board was connected via Ethernet and assigned a static IP address, allowing direct access without routing through external network infrastructure. By configuring the host machine's network interface to operate on the same subnet, the device became reachable through SSH, enabling interaction with its runtime environment.

On the Loihi 2 side, the appropriate version of nxCore, the low-level runtime layer of Intel's Loihi neuromorphic platform, was verified or transferred when necessary, ensuring compatibility with the NxKernel installation used on the local system. The development environment was then prepared by installing NxKernel

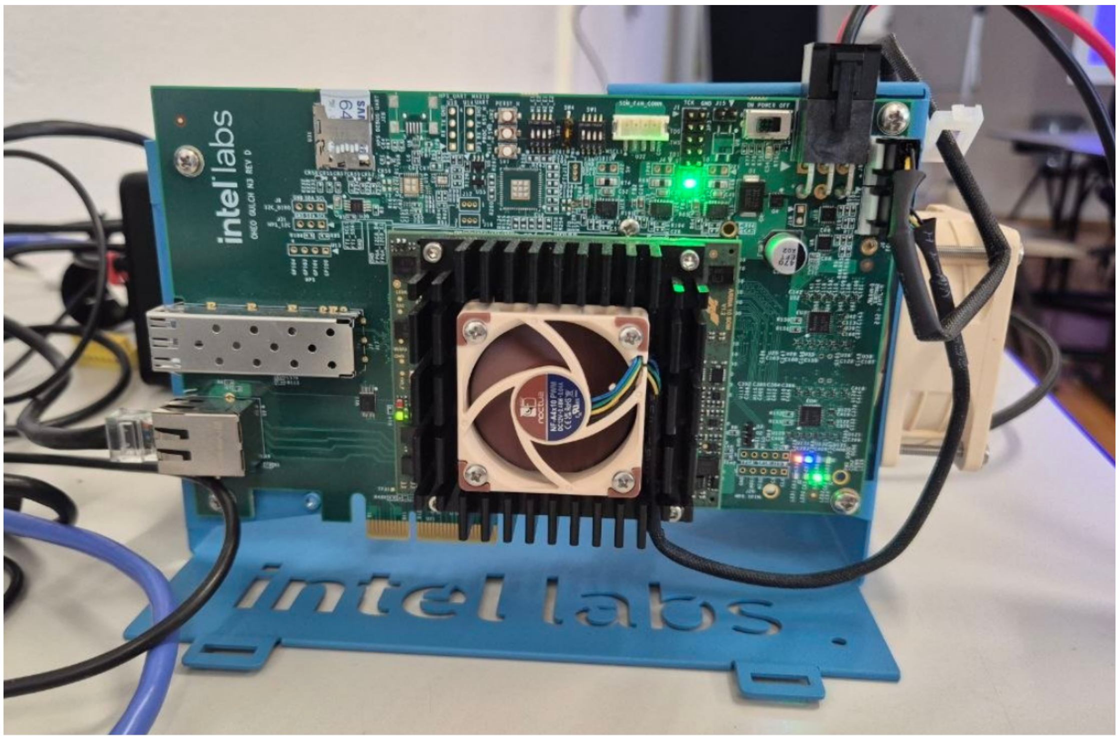


Figure 3.10: The physical Loihi 2 system used in the neuromorphic laboratory at ZHAW Wädenswil.

together with all required system dependencies, compilers and libraries. Proper environment variables were configured so that NxKernel could locate the Loihi 2 driver and communicate with the hardware over the network.

Once both environments were aligned, a set of minimal diagnostic tests confirmed that the board could be initialized, run a simple program and shut down correctly. Successful execution of these tests validated the connection and ensured that the DevKit and local environment were fully synchronized.

This setup provided the operational foundation needed to execute the Spiking LSTM experiments directly on physical neuromorphic hardware, rather than relying on remote servers or simulation-based environments.

3.7 Testing on the Final Dataset

The Spiking Heidelberg Digits (SHD) dataset was selected as the final benchmark to evaluate the proposed architecture. As already discussed in the *Background* section, SHD is a neuromorphic dataset composed of spike trains derived from spoken digits, characterised by a strong temporal structure and long-range dependencies across

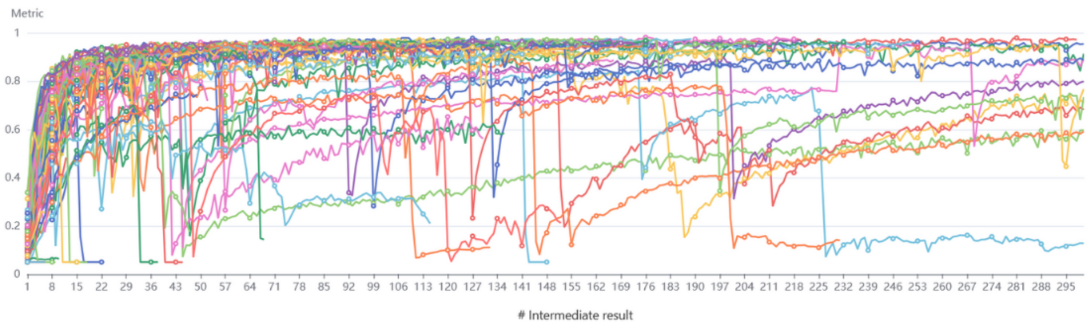


Figure 3.11: Accuracy curves for all NNI-generated configurations. Each line represents a different hyperparameter trial evaluated during the optimization process.

timesteps. These properties make it particularly well suited for assessing the ability of the Spiking LSTM to retain and process information over time, offering a more meaningful test than static or weakly temporal datasets.

To better understand the typical preprocessing strategies applied to SHD, the published work reported in the official leaderboard 2.3 were examined. Several approaches were identified: reducing the number of input channels, compressing temporal resolution by binning spikes into larger time windows, or trimming sequences to shorten the effective duration of each sample. These transformations generally aim to reduce sequence length and computational load, particularly for architectures that do not need fine-grained temporal information.

Among the available options, the preprocessing strategy adopted in this Thesis preserves the complete structure of the dataset. All input channels and all original timesteps were retained, avoiding any form of temporal compression. Samples were clipped to a maximum duration and shorter sequences were padded so that all inputs shared the same sequence length. This choice ensures that the model receives the full temporal dynamics of the dataset while still allowing batch-wise processing. The same preprocessing strategy is used in [83], and it was intentionally adopted here as well so that the resulting performance can be fairly compared to prior work that evaluates recurrent spiking architectures under equivalent input conditions.

This setup provides a reliable and temporally rich benchmark to assess the model's capacity for sequence processing, memory retention and temporal discrimination.

Table 3.1: Search space for hyperparameter optimization.

Hyperparameter	**Type**	**Values / Range**
hidden_size	choice	32, 64, 128, 256
enc_pop	choice	64, 128, 256, 512
beta_enc	quniform	0.05 – 1 (step 0.05)
beta_forget	quniform	0.05 – 1 (step 0.05)
beta_input	quniform	0.05 – 1 (step 0.05)
beta_gate_p	quniform	0.05 – 1 (step 0.05)
beta_gate_n	quniform	0.05 – 1 (step 0.05)
beta_hidden	quniform	0.05 – 1 (step 0.05)
beta_out	quniform	0.05 – 1 (step 0.05)
thr_enc	quniform	0.05 – 1 (step 0.05)
thr_forget	quniform	0.05 – 1 (step 0.05)
thr_input	quniform	0.05 – 1 (step 0.05)
thr_gate_p	quniform	0.05 – 1 (step 0.05)
thr_gate_n	quniform	0.05 – 1 (step 0.05)
thr_hidden	quniform	0.05 – 1 (step 0.05)
thr_out	quniform	0.05 – 1 (step 0.05)
learning_rate	choice	0.0001, 0.0002, 0.0005, 0.001, 0.002, 0.005, 0.01
batch_size	choice	64, 128, 256

3.8 Training and Test

The training of the spiking LSTM models was conducted entirely within the snnTorch framework, leveraging its ability to efficiently handle event-based data and simulate spiking dynamics. Hyperparameter optimization was performed using NNI to identify configurations that maximize performance while maintaining stability across runs. Since the SHD dataset does not include a dedicated validation set, a portion of the training data was reserved for the validation subset, which has the same size as the test set (around 30% of the training set). To further reduce the risk of overfitting during hyperparameter optimization, the training data was split into 10 different train/validation folds, allowing NNI to evaluate each configuration on a different data partitions.

The hyperparameters explored, including learning rate, membrane time constants and weight initialization scales, are summarized in Table 3.1. All the decay parameters are, as expected, between 0 and 1, since these are their nominal values. Thresholds were also constrained between 0 and 1, given that the input consists of sparse spikes. Common values were chosen for the batch size and learning rate. The hidden size affects the entire architecture, as it determines the dimensionality of all populations within the sLSTM cell, i.e., the gates and LSTM neurons, while the encoding population is responsible for extracting features from the input. Models were trained for up to 300 epochs, with early stopping enabled to terminate runs that showed minimal improvement in loss or accuracy. This setup ensured that the selected hyperparameters generalized well across different folds and prevented overfitting to specific subsets of the data. The resulting configurations were then used to train the final spiking LSTM models before deployment on NxKernel for hardware validation.

Following training and hyperparameter optimization in snnTorch, the learned weights, thresholds and neuron parameters were transferred to the Loihi 2 hardware for evaluation. The network was executed using the alternated mode previously described, which balances throughput and latency by sequentially updating even and odd layers at different timesteps.

Chapter 4

Results

This section presents the results obtained by evaluating the spiking LSTM on the SHD dataset. Models were first trained in *snnTorch* and subsequently deployed on the Loihi 2 hardware, leveraging the alternated execution mode and carefully quantized parameters to ensure consistency between software simulations and neuromorphic implementation.

The results are presented to provide a comprehensive evaluation of the proposed spiking LSTM. It begins by assessing the overall classification accuracy on the SHD test set, offering a first indication of the model's ability to capture temporal dependencies. Building on this, a sensitivity analysis examines how variations in key hyperparameters affect performance and stability, highlighting the most influential factors. Given the importance of transferring models from snnTorch to Loihi 2, then the impact of parameter quantization on accuracy and reliability is investigated. This is followed by an analysis of partitioning and profiling, which sheds light on computational efficiency and hardware resource utilization on Loihi 2. Finally, the proposed model is compared both to state-of-the-art spiking networks and to a reference recurrent architecture implemented on Loihi 2 [83], providing a nuanced view of its generalization capabilities across temporal sequences and neuromorphic platforms.

4.1 Accuracy

The proposed spiking LSTM achieves a maximum test accuracy of 91.917% on the SHD dataset. This result is obtained by the configuration whose training trajectory reaches 99.88% training accuracy and 97.57% validation accuracy at epoch 124 (out of a maximum of 300). Although this run represents the best-performing model, the overall distribution of results across the hyperparameter search is considerably broad (Figure 4.1): while some configurations fall near 40%, a large portion

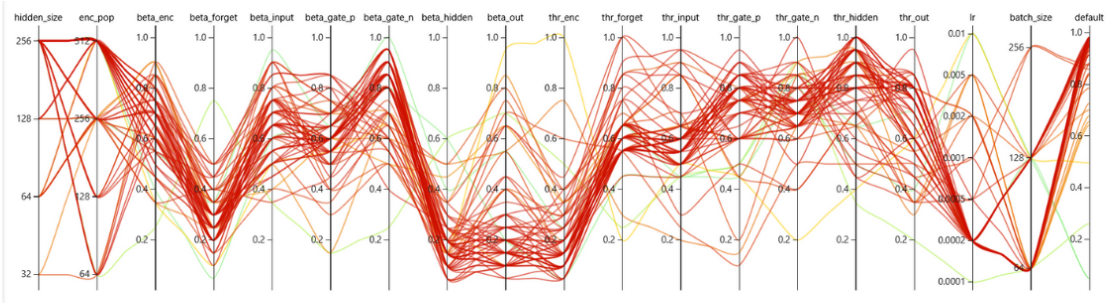


Figure 4.1: Visualization of NNI-explored hyperparameter configurations. Darker red indicates higher accuracy, showing which combinations of parameters led to better model performance. The last column refers to the final accuracy of the trial.

of the search space yields accuracies between 70% and 90%, indicating that the architecture is capable of learning the task under a wide range of parameter settings, but remains sensitive to specific choices that shape its temporal dynamics.

A noticeable discrepancy emerges between validation and test accuracy. This gap is expected, as the SHD test set includes recordings from two speakers who do not appear in either the training or validation sets. Consequently, the test evaluation measures not only temporal classification performance but also the model's ability to generalize to previously unseen speaker characteristics. The best model preserves high performance despite this distribution shift, suggesting that the spiking LSTM effectively captures structure in the temporal patterns rather than overfitting to speaker-specific characteristics.

These results provide a baseline for interpreting the following analyses. The sensitivity study investigates which hyperparameters most strongly influence this variability, the quantization section examines how performance translates to Loihi-compatible weight formats and the final comparisons contextualize the achieved accuracy relative to both spiking and recurrent implementations in the literature and on neuromorphic hardware.

4.2 Sensitivity Analysis

The explored hyperparameter values are reported in Figure 4.1 and were previously summarized in Table 3.1. The plotted lines are color-coded from green to dark red, with darker red indicating higher accuracy. Several patterns can be observed from this analysis. The decay factors (betas) of the encoder, input and positive gate show optimal performance when set to medium-high values, roughly between 0.5 and 0.8, suggesting that moderate retention of past information is beneficial in these components. In contrast, the forget and decoder betas achieve better results at

Table 4.1: Comparison of accuracies across snnTorch, quantized snnTorch and Loihi 2 (NxKernel) executions.

snnTorch	**snnTorch (quantized)**	**Loihi (NxKernel)**
91.917%	90.7686%	85.3357%
88.9134%	88.1184%	86.2191%
87.3233%	87.0583%	85.9982%

medium-low values (around 0.2–0.5), promoting a controlled forgetting mechanism. The hidden beta is consistently low (0.1–0.3), indicating a tendency to attenuate the contribution of the gates to the neuron's membrane potential, effectively mitigating potential over-accumulation of signal. Conversely, the negative gate beta is high (0.5–0.9), favoring stronger retention of inhibitory or negative contributions.

In general, higher beta values allow neurons to retain information from previous timesteps for longer, whereas lower beta values accelerate forgetting, highlighting the importance of temporal tuning for network stability. Since thresholds show relatively little variation and occupy a narrow range around 0.5–1, the differences in network dynamics across configurations are primarily driven by the beta parameters, which determine how information propagates and decays through the layers. Particularly interesting is the divergence between the positive and negative gates. Although they were initially designed to complement each other and approximate a *tanh*-like operation — where one might expect their ranges to mirror each other —analysis reveals a dominance of the negative gate, resulting in outputs that are skewed toward negative values after the subtraction of the two gates. This emergent asymmetry suggests that the network leverages the negative gate more strongly for fine-tuning the sign and amplitude of the hidden state.

Regarding other hyperparameters, the encoder population size shows little effect on final performance, implying that even smaller populations can adequately represent input features. On the other hand, larger hidden sizes, particularly 256, consistently lead to better accuracy, with 128 and 64 still performing acceptably but less robustly. Among batch sizes, 64 provides the most stable performance, with 128 occasionally yielding comparable results. The learning rate of 0.0002 emerges as the preferred choice, balancing convergence speed and stability. Lower accuracies are instead strongly associated with suboptimal learning rate choices, particularly when the learning rate is too high, which often leads to unstable training dynamics or insufficient convergence.
Overall, these observations highlight the network's sensitivity to certain hyperparameters, while also indicating a relatively broad region of configurations that achieve high accuracy.

4.3 Quantization Impact

Quantization plays a crucial role in the transition from snnTorch to Loihi 2, as weights, thresholds and decay factors must be represented with limited precision while avoiding overflow on hardware. Across all experiments, accuracy between the two frameworks remained remarkably stable: deviations from the original floating-point model were minimal, and in some cases the quantized model even performed slightly better. This suggests that the learned representations are robust to moderate quantization noise.
When the same quantized parameters were transferred to NxKernel and executed on Loihi 2, accuracy decreased more noticeably, although still within a modest 2–5% range depending on the model (Table 4.1). Despite using the same quantization rules, small discrepancies arise from hardware-level effects not captured during snnTorch-side evaluation. For instance, snnTorch does not simulate potential integer overflows or saturation in fixed-point arithmetic and interactions involving bit-shifts, spike payloads and accumulation dynamics may behave subtly differently on hardware. These effects likely explain the slight performance drop observed on Loihi 2.
An additional observation is that the models showing the smallest accuracy drop on Loihi 2 are also those whose learned parameters naturally fit the exponent used during hardware quantization. In contrast, the model with the larger gap required a higher exponent during training, but was quantized with a smaller one due to hardware constraints. This effectively reduced the magnitude of its weights, which may help explain why it was more sensitive to the transition to hardware. The results of these three steps for three representative, well-performing models are reported in Table 4.1.
Overall, these findings indicate that *semi-quantization-aware* training and post-training quantization are generally sufficient to preserve performance in software, while hardware execution introduces additional, but limited, constraints. Making these hardware-specific effects explicit in the training or simulation pipeline may further reduce the remaining 2–5% gap.

4.4 Hardware Execution Analysis

In this section, an analysis of the execution and computational behavior of the proposed spiking LSTM on Loihi 2 is presented. The study begins with an investigation of how to partition the network across hardware cores, selecting an intermediate-sized model that balances computational load and runtime. Following this, the focus is on models with test accuracy above 85%, ensuring that only well-performing configurations are considered. For each of the 20 classes in the SHD

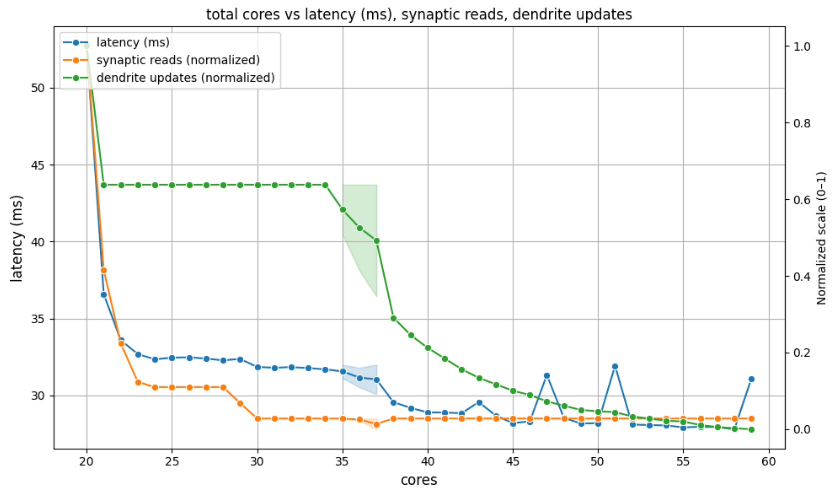


Figure 4.2: Effect of core partitioning on total runtime.

dataset, 10 representative samples were selected and the resulting metrics were averaged per class to provide a robust, dataset-level characterization of runtime performance, synaptic and dendritic activity, and hardware utilization. To obtain an accurate measurement of network execution time that excludes input-transfer delays, the selected samples were preloaded into a set of neuron cores on the Loihi 2 chip.

4.4.1 Impact of partitioning

A key aspect of deploying neuromorphic architectures on hardware is how the network is organized and partitioned across physical cores and chips. Partitioning and mapping directly impact total runtime and inter-core communication overhead. In [39], besides identifying the three main performance bottlenecks of Loihi 2 like systems (synaptic reads, dendrite updates, traffic congestion), the authors proposed a heuristic method to find an appropriate partitioning and mapping configuration for the target network, which we apply to our network. The method consists of iteratively adding more neuron cores, trying to identify the layer that needs it the most, by considering the performance bottlenecks. In short, the minimal partitioning configuration is found (i.e., the minimum number of neuron cores required to fit the network). The network is executed with a sample input, and the neuron core with maximum number of synaptic memory reads is identified, as it is a probable candidate to be the bottlenecking neuron core. A new neuron core is added to the layer that contains the bottlenecking core, as this will reduce the number of neurons in each of the cores of this layer, leading to a reduction in the synaptic memory reads, and possibly an improvement in runtime. This process is repeated iteratively until no improvement in runtime is detected. Then the same approach is applied using the dendrite updates metric. To tackle the traffic congestion bottleneck, the authors found that ordered mapping configurations are slower than random or strided approaches, so we use a random mapping for our network.

As shown in Figure 4.2, increasing the number of cores following the heuristic method from [39] reduces runtime, decreasing from over 52 ms to approximately 27 ms. Beyond a certain point, however, adding more cores no longer provides substantial gains, resulting in a plateau where runtime stabilizes due to communication overhead and hardware constraints. As additional cores are employed for the core that has the most connections and weights, the maximum number of reads per core decreases, eventually reaching a plateau. Once the plateau is reached, adding more cores translates in little to no further improvement. At this point targeting the cores with the most dendrite updates allows to further decrease the runtime. Since both the number of synaptic reads and the dendrite updates are directly correlated with runtime: higher values generally result in longer processing times. This explains why runtime initially decreases as cores are added. This behavior highlights the importance of identifying an optimal partitioning scheme that balances core utilization and inter-core communication to minimize total execution time. The best partitioning found is reported in table 4.2. As expected, the layer requiring more cores is the LSTM, as the neuron models in it are relatively complex, requiring around 4 times more instructions than neurons in the other layers.

Table 4.2: Best partitioning found by the heuristic approach

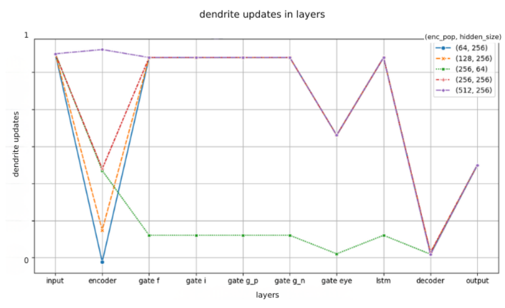
Layer	Encoder	f gate	i gate	cp gate	cn gate	LSTM	Decoder
Cores	4	4	4	4	4	8	1

4.4.2 Dendrite Updates and Synaptic Reads

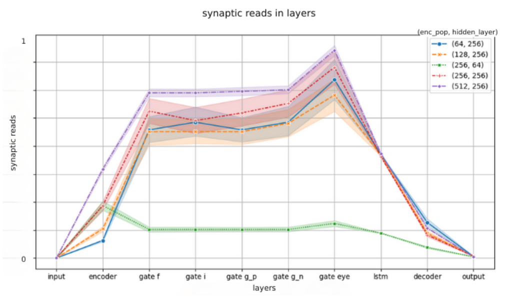
The analysis of dendritic and synaptic activity provides insight into how active each neuron group is during execution. These metrics reflect the number of updates occurring in each dendritic and synaptic compartment, which can be interpreted as a proxy for the computational load or engagement of each group. In Figure 4.3, it is evident that these values are closely tied to the layer dimensions rather than on other parameters.

Regarding dendrite updates (Figure 4.3a), the values strictly depend on the model dimension. This behavior occurs because, on Loihi, each neuron is always executed regardless of the input, allowing the membrane potential to decay even when the input is zero. For example, in the encoder layer, four distinct points correspond to the four possible sizes assigned to this layer, while for the other gates only two points appear, corresponding to hidden sizes of 256 and 64. As expected, smaller layers exhibit lower dendritic activity due to fewer neurons being updated. On the other hand, synaptic reads (Figure 4.3b) are also largely influenced by model dimension, but their exact values additionally depend on factors such as the synaptic weights, which explains why the plots do not perfectly overlap.

Figure 4.4 highlights the relationship between dendritic updates, synaptic reads,

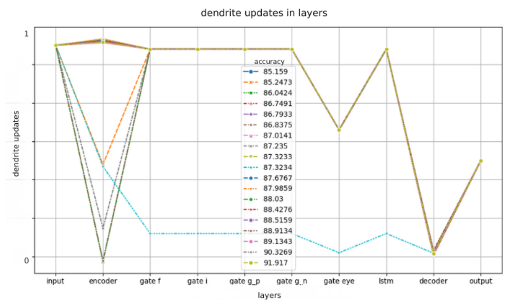


(a) Maximum dendritic updates per layer as a function of layer size.

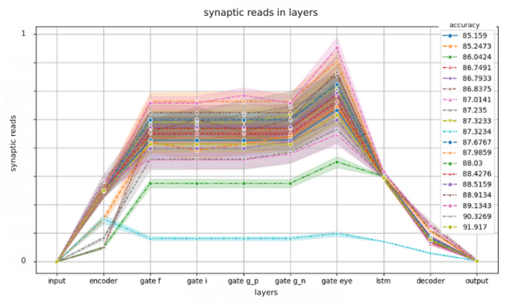


(b) Maximum synaptic reads per layer as a function of layer size.

Figure 4.3: For each layer size, the maximum synaptic reads and dendritic updates are plotted.



(a) Maximum dendritic updates per layer plotted highlighting model accuracy.



(b) Maximum synaptic reads per layer plotted, highlighting model accuracy.

Figure 4.4: For each model accuracy, the maximum synaptic reads and dendritic updates are plotted.

and model accuracy. Here, it becomes clear that there is little correlation: models with higher or lower accuracy do not systematically produce more or fewer dendritic updates or synaptic reads. This suggests that while dendritic activity and synaptic reads scale with network size, it is not directly predictive of performance, emphasizing that other factors, such as parameter tuning and network dynamics, play a more critical role in determining accuracy.

4.4.3 Accuracy and Architectural Characteristics

In the following plots, the models are analyzed in terms of their accuracy to investigate possible correlations with various architectural characteristics, such as runtime, dendritic accumulation and synaptic reads. Each model is further distinguished according to the total number of neurons in the network, allowing us to evaluate how network size influences overall performance.

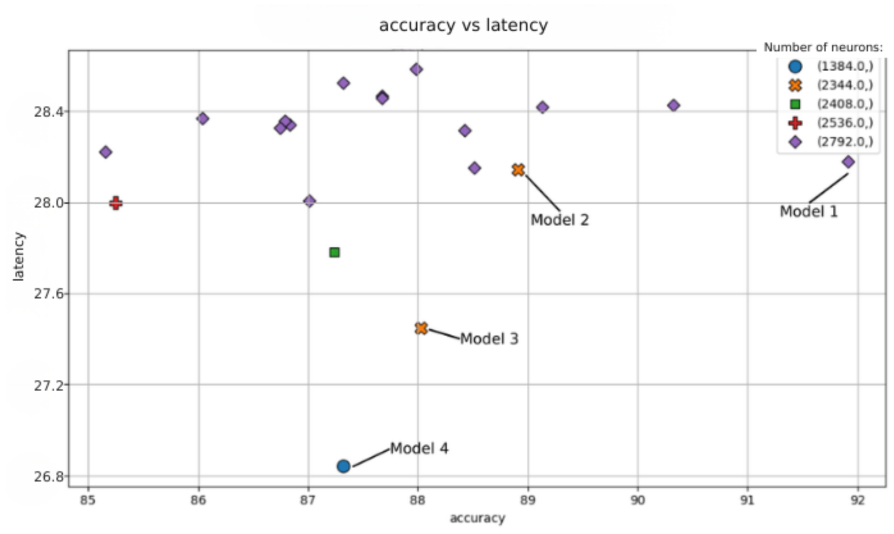


Figure 4.5: Accuracy versus runtime for all models, highlighting differences in performance across various network sizes.

The first analysis focuses on the relationship between accuracy and runtime, which is plotted in Figure 4.5. As expected, model size influences runtime: larger models generally appear in the upper part of the plot, corresponding to longer execution times, while the smallest model is located in the lower part, with faster runtime. This trend is not absolute: some smaller models are slightly slower and some larger models are faster than expected, indicating that runtime depends on multiple factors beyond network size alone.
Models achieving the highest accuracy typically have larger architectures and higher runtime, suggesting that increased model capacity can enhance performance. The inverse relationship does not hold: not all large models reach high accuracy, highlighting the role of other parameters in determining model performance. Interestingly, the smallest and fastest model still reaches high accuracy, demonstrating

Table 4.3: Hyperparameters and performance of selected spiking LSTM models.

Parameter	Model Name			
	model 1	model 2	model 3	model 4
test_accuracy	91.9170	88.9134	88.0300	87.3233
hidden_size	256	256	256	64
enc_pop	512	64	64	256
beta_enc	0.698153	0.900548	0.755746	0.733532
beta_forget	0.393573	0.296765	0.231026	0.548271
beta_input	0.590215	0.794695	0.807695	0.955342
beta_gate_p	0.639633	0.535150	0.712546	0.759258
beta_gate_n	0.982469	0.968170	0.986815	0.702799
beta_hidden	0.208221	0.343436	0.276347	0.082683
beta_out	0.460626	0.718267	0.409980	0.574586
thr_enc	0.279894	0.263220	0.241975	0.396425
thr_forget	0.759035	0.668511	0.603405	0.939939
thr_input	0.591368	0.541040	0.564978	0.987820
thr_gate_p	0.886300	0.590016	0.825161	0.599854
thr_gate_n	0.743576	0.612336	0.509402	1.005941
thr_hidden	0.833048	0.908163	1.028403	0.517627
thr_out	0.727680	0.711243	0.496498	0.436543

that compact architectures can achieve competitive results despite lower computational cost. In this plot some Pareto point con be identified, points that represent optimal trade-off between accuracy and runtime. Importantly, no point is strictly better than another; the choice depends on the specific application requirements. Details about these points are reported in Table 4.3.

In the next plot (Figure 4.6), the accuracy was analyzed as a function of average dendritic updates per timestep, computed across all classes. Models with the same total number of neurons exhibit identical average dendritic updates, which, as previously noted, depends solely on model size rather than the dataset. This plot further confirms that accuracy is not strictly determined by model size and, consequently, by the number of dendritic updates. The final plot examines the relationship between model accuracy and the average number of synaptic memory reads, computed over all 20 classes for each model. As previously observed, synaptic memory reads correlate strongly with runtime and, consequently, with model size. This explains why the trend in Figure 4.7 closely mirrors the one discussed earlier for time-per-timestep (Figure 4.5). Similar to the previous analyses, accuracy does not show a definitive relationship with synaptic activity or model size. Instead, it becomes evident that model dimensions primarily drive the hardware metrics

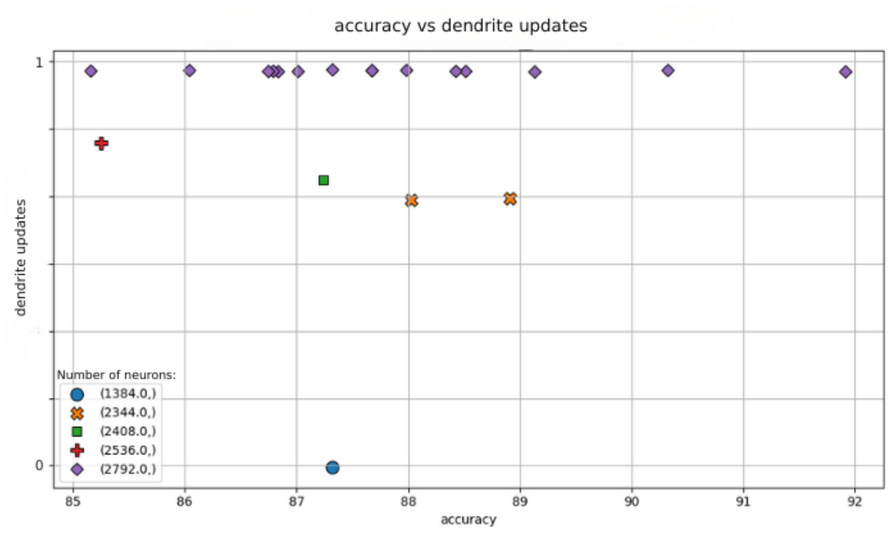


Figure 4.6: Accuracy versus average dendritic updates per timestep for all models, showing that models with the same size share identical dendritic activity.

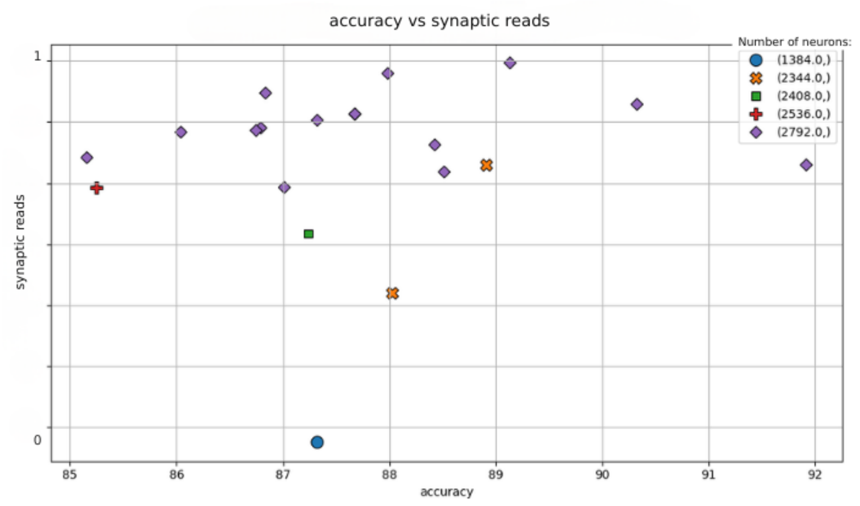


Figure 4.7: Accuracy versus average synaptic reads per timestep for all models.

examined so far.
These observations emphasize that accuracy alone is only one of several factors to consider when evaluating neuromorphic models. In some scenarios, accepting a slight reduction in accuracy may be advantageous if it leads to substantial improvements in efficiency, runtime or hardware resource usage.

4.5 SHD leaderboard

As discussed in Section 4.4.3, four Pareto-optimal models were identified, each representing a different trade-off between accuracy and runtime. For the comparison with existing works, however, the focus is on the model achieving the highest accuracy, as this choice enables a more direct evaluation of the architecture's performances.

As reported in Table 2.3, the proposed architecture achieves competitive accuracy on *snnTorch* simulations, ranking immediately after the work of Yu et al. (2022) [63] in 9th position. Importantly, among the models achieving similar or higher accuracy, this is the only one that has been successfully deployed on Loihi 2 or on any other neuromorphic hardware, demonstrating real on-chip performance. In this sense, the proposed model occupies the first position in terms of actual, on-chip accuracy, confirming its practical effectiveness while remaining fully compatible with neuromorphic hardware constraints.

As previously discussed, accuracy is not the only metric to consider when evaluating neuromorphic models. To provide a more complete comparison, also other factors are evaluated, comparing the architecture against another network deployed on Loihi 2: the RSNN proposed by Shoesmith et al. [83]. A summary of this comparison is reported in Table 4.4.

From the table, it is evident that the RSNN achieves a significantly lower latency (between 2.3 and 2.5 ms), whereas the LSTM-based model reaches latencies on the order of 28 ms. This difference is naturally attributable to the structural characteristics of the networks: the RSNN consists of only two layers, while the LSTM includes eight layers with dense interconnections. Moreover, the RSNN is composed solely of LIF neurons, whereas the LSTM incorporates several custom and more complex neuron models.

Regarding accuracy, even though the RSNN values are estimated, the two architectures remain broadly comparable with the spiking LSTM of this Thesis slightly outperforming, highlighting the trade-offs between structural complexity, computational cost and performance.

Model	Hidden size	Latency [ms]	Accuracy
RSNN	256	2.33	$\approx 82\%*$
	512	2.37	$\approx 85\%*$
	1024	2.56	$\approx 90\%*$
LSTM	256	28.16	91.92%
	256	28.14	88.91%
	256	27.44	88.03%
	64	26.84	87.32%

Accuracies marked with * are estimated from Figure 4 in the paper, as the authors did not explicitly report them.

Table 4.4: Latency and accuracy of both RSNN and sLSTM for comparison.

Chapter 5

Conclusion

This Thesis aimed to explore a spiking variant of the widely used LSTM architecture. In doing so, it investigated various design choices for a spiking LSTM, trying to preserve the fundamental properties of the original model while adapting it to the neuromorphic domain. The final architecture was successfully deployed on Loihi 2, demonstrating the feasibility of translating complex recurrent spiking networks to physical neuromorphic hardware.

To achieve this goal, the model was first implemented in snnTorch, where it was trained using hyperparameter optimization through NNI. A custom pipeline was then developed to translate the model into NxKernel, ensuring that all network parameters, including weights, decay factors, and thresholds, could be loaded onto Loihi 2 with minimal performance loss. This pipeline was essential to bridge the differences between a flexible software environment and the constraints of neuromorphic hardware.

A comprehensive analysis of the model showed that it is highly sensitive to hyperparameters: decay factors, thresholds, and neuron populations directly influence accuracy, stability and the network's ability to retain information over time. Hardware profiling and partitioning studies confirmed that how neurons and synapses are distributed across cores strongly affects runtime, inter-core communication, synaptic reads, and dendritic updates. These results provide practical guidance for designing spiking networks on neuromorphic hardware, highlighting how layer sizes and partitioning can be chosen to balance performance, efficiency, and temporal processing capabilities.

Importantly, the final spiking LSTM architecture achieved an accuracy of 91.917% on snnTorch and 85.336% on NxKernel on the SHD benchmark. While placing comparably on the current leaderboard, it is the only model among top-performing approaches to be successfully deployed on Loihi 2, thereby occupying the first position in terms of actual on-chip accuracy. Its performance is comparable to other approaches, such as RSNNs, demonstrating that the model effectively balances high accuracy with full compatibility with neuromorphic hardware constraints.

Overall, this work demonstrates that a careful co-design of network architecture, training strategies, and hardware deployment can significantly reduce the gap between software simulation and neuromorphic execution.

Future research could focus on refining the translation pipeline between snnTorch and NxKernel, enabling even closer alignment between software and hardware performance. Additionally, a deeper exploration of neuron design, reset mechanisms, and gate dynamics of the LSTM may improve both the efficiency and effectiveness of the architecture. Finally, optimizing the balance between dendritic updates and synaptic reads, in particular regarding the gates, could yield faster and more resource-efficient networks, paving the way for more scalable and real-time neuromorphic applications.

Bibliography

[1] Kaushik Roy, Angshuman Jaiswal, and Amogh Agrawal. «Towards spike-based machine intelligence with neuromorphic computing». In: *Nature* 575.7784 (2019), pp. 607–617. DOI: `10.1038/s41586-019-1677-2` (cit. on p. 1).

[2] Benedikt Jung, Maximilian Kalcher, Merlin Marinova, Piper Powell, and Esma Sakallı. «Neuromorphic Computing – An Overview». In: *arXiv preprint* (2025). eprint: `arXiv:2510.06721v1` (cit. on p. 1).

[3] Kwabena Boahen. «A Neuromorph's Prospectus». In: *Computing in Science & Engineering* 19.2 (2017), pp. 14–28. DOI: `10.1109/MCSE.2017.33` (cit. on p. 1).

[4] Wolfgang Maass. «Networks of spiking neurons: the third generation of neural network models». In: *Neural Networks* 10.9 (1997), pp. 1659–1671. DOI: `10.1016/S0893-6080(97)00011-7` (cit. on p. 1).

[5] Intel Labs. *Taking Neuromorphic Computing to the Next Level with Loihi 2: Technology Brief.* Tech. rep. Technology Brief. Accessed: 2025-11-24. Intel Labs, 2021. URL: `https://download.intel.com/newsroom/2021/new-technologies/neuromorphic-computing-loihi-2-brief.pdf` (cit. on pp. 1, 8–10, 14).

[6] Seung Ju Kim, Sang Bum Kim, and Ho Won Jang. «Competing memristors for brain-inspired computing». In: *iScience* 24.1 (2021). Accessed: 2025-11-25, p. 101889. DOI: `10.1016/j.isci.2020.101889`. URL: `https://www.sciencedirect.com/science/article/pii/S2589004220310865` (cit. on p. 3).

[7] Mohamadreza Zolfagharinejad, Unai Alegre-Ibarra, Tao Chen, Sachin Kinge, and Wilfred G. van der Wiel. «Brain-inspired computing systems: a systematic literature review». In: *The European Physical Journal B* 97 (2024). Accessed: 2025-11-25, p. 70. DOI: `10.1140/epjb/s10051-024-00703-6`. URL: `https://link.springer.com/article/10.1140/epjb/s10051-024-00703-6` (cit. on p. 3).

[8] Charlotte Frenkel, David Bol, and Giacomo Indiveri. «Bottom-up and top-down approaches for the design of neuromorphic processing systems: Tradeoffs and synergies between natural and artificial intelligence». In: *arXiv preprint arXiv:2106.01288* (2021). Accessed: 2025-11-25. DOI: 10.48550/arXiv.2106.01288. URL: https://arxiv.org/abs/2106.01288 (cit. on p. 3).

[9] Wiktoria Agata Pawlak and Newton Howard. «Neuromorphic algorithms for brain implants: a review». In: *Frontiers in Neuroscience* 19 (2025). PMCID: PMC12021827, Accessed: 2025-11-25. DOI: 10.3389/fnins.2025.1570104. URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC12021827/ (cit. on p. 3).

[10] Shuming Liu. «Energy-Efficient Neuromorphic Chips for Real-Time Robotic Control: A Review». In: *Theoretical and Natural Science* 134 (2025). Accessed: 2025-11-25, pp. 73–78. URL: https://direct.ewa.pub/proceedings/tns/article/view/26488 (cit. on p. 3).

[11] Lyes Khacef, Philipp Klein, Matteo Cartiglia, Arianna Rubino, Giacomo Indiveri, and Elisabetta Chicca. «Spike-based local synaptic plasticity: A survey of computational models and neuromorphic circuits». In: *arXiv preprint arXiv:2209.15536* (2022). Accessed: 2025-11-25. DOI: 10.48550/arXiv.2209.15536. URL: https://arxiv.org/abs/2209.15536 (cit. on p. 3).

[12] Xiangjing Wang, Yixin Zhu, Zili Zhou, Xin Chen, and Xiaojun Jia. «Memristor-Based Spiking Neuromorphic Systems Toward Brain-Inspired Perception and Computing». In: *Nanomaterials* 15.14 (2025). Accessed: 2025-11-25, p. 1130. DOI: 10.3390/nano15141130. URL: https://www.mdpi.com/2079-4991/15/14/1130 (cit. on p. 3).

[13] *Neuronal Activation*. https://www.ch.ic.ac.uk/local/projects/quek/chnact.htm. Image accessed: 2025-11-26 (cit. on p. 4).

[14] Qilin Hua, Huaqiang Wu, Bin Gao, Qingtian Zhang, Wei Wu, Yujia Li, Xiaohu Wang, Weiguo Hu, and He Qian. «Low-Voltage Oscillatory Neurons for Memristor-Based Neuromorphic Systems». In: *Global Challenges* 3.11 (2019). PMC free article, accessed via PubMed Central, p. 1900015. DOI: 10.1002/gch2.201900015. URL: https://doi.org/10.1002/gch2.201900015 (cit. on p. 4).

[15] The University of Queensland Queensland Brain Institute. *Action potentials and synapses*. https://qbi.uq.edu.au/brain-basics/brain/brain-physiology/action-potentials-and-synapses. Accessed: 2025-11-25 (cit. on p. 3).

[16] Intel Labs. *Neuromorphic Computing – Next Generation of AI*. https://www.intel.la/content/www/xl/es/research/neuromorphic-computing.html. Accessed: 2025-11-24 (cit. on p. 4).

[17] Human Brain Project. *Neuromorphic Computing.* `https://www.humanbrainproject.eu/en/science-development/focus-areas/neuromorphic-computing/`. Accessed: 2025-11-24 (cit. on p. 4).

[18] Eldar Sido. *Developing Neuromorphic Devices for TinyML. Electronic Design*, "Engineering Essentials" section, Nov. 30 2022. Accessed: 2025-11-24. URL: `https://digital.electronicdesign.com/electronicdesign/spring_2023/MobilePagedArticle.action?articleId=1940453` (cit. on p. 5).

[19] Douglas Z. Plummer, Emily D'Alessandro, Aidan Burrowes, Joshua Fleischer, Alexander M. Heard, and Yingying Wu. «2D Spintronics for Neuromorphic Computing with Scalability and Energy Efficiency». In: *Journal of Low Power Electronics and Applications* 15.2 (2025). Accessed: 2025-11-24, p. 16. DOI: `10.3390/jlpea15020016`. URL: `https://www.mdpi.com/2079-9268/15/2/16` (cit. on p. 5).

[20] A. N. Burkitt. «A Review of the Integrate-and-Fire Neuron Model: I. Homogeneous Synaptic Input». In: *Biological Cybernetics* 95.1 (2006), pp. 1–19. DOI: `10.1007/s00422-006-0068-6` (cit. on p. 4).

[21] Chankyu Lee, Syed Shakib Sarwar, Priyadarshini Panda, Gopalakrishnan Srinivasan, and Kaushik Roy. «Enabling Spike-Based Backpropagation for Training Deep Neural Network Architectures». In: *Frontiers in Neuroscience* 14 (2020). Accessed: 2025-11-24, p. 119. DOI: `10.3389/fnins.2020.00119`. URL: `https://www.frontiersin.org/articles/10.3389/fnins.2020.00119/full` (cit. on p. 6).

[22] Rina Diane Caballar and Cole Stryker. *What Is Neuromorphic Computing?* `https://www.ibm.com/think/topics/neuromorphic-computing`. IBM Think, accessed: 2025-11-24. 2025 (cit. on p. 6).

[23] Davide Anguita, Alessandro Ghio, Luca Oneto, Xavier Parra, and Jorge Luis Reyes-Ortiz. «A Public Domain Dataset for Human Activity Recognition using Smartphones». In: *21st European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning (ESANN 2013), Bruges, Belgium, 24–26 April 2013.* Accessed: 2025-11-25. 2013, pp. 437–442. URL: `https://www.esann.org/sites/default/files/proceedings/legacy/es2013-84.pdf` (cit. on pp. 7, 17).

[24] Gary Weiss. *WISDM Smartphone and Smartwatch Activity and Biometrics Dataset.* UCI Machine Learning Repository. DOI: https://doi.org/10.24432/C5HK59. 2019 (cit. on pp. 7, 17).

[25] G. Orchard, G. Cohen, A. Jayawant, and N. Thakor. «Converting Static Image Datasets to Spiking Neuromorphic Datasets Using Saccades». In: *Frontiers in Neuroscience* 9 (2015), p. 437. DOI: `10.3389/fnins.2015.00437` (cit. on p. 7).

[26] Hongmin Li, Hanchao Liu, Xiangyang Ji, Guoqi Li, and Luping Shi. «CIFAR10-DVS: An Event-Stream Dataset for Object Classification». In: *Frontiers in Neuroscience* 11 (2017), p. 309. DOI: 10.3389/fnins.2017.00309 (cit. on p. 7).

[27] A. Amir, B. Taba, D. Berg, T. Melano, D. McKinstry, A. Di Nolfo, R. D. Boahen, C. A. Chou, and T. E. Carlson. «A Low Power, Fully Event-Based Gesture Recognition System». In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR).* 2017, pp. 7388–7397. DOI: 10.1109/CVPR.2017.780 (cit. on p. 7).

[28] Qian Liu, Garibaldi Pineda-García, Evangelos Stromatias, Teresa Serrano-Gotarredona, and Steve B. Furber. «Benchmarking Spike-Based Visual Recognition: A Dataset and Evaluation». In: *Frontiers in Neuroscience* 10 (2016), p. 496. DOI: 10.3389/fnins.2016.00496 (cit. on p. 7).

[29] Steve B. Furber, Francesco Galluppi, Steve Temple, and Luis A. Plana. «The SpiNNaker Project». In: *Proceedings of the IEEE* 102.5 (2014), pp. 652–665. DOI: 10.1109/JPROC.2014.2304638 (cit. on p. 8).

[30] Christian Mayr, Sebastian Höppner, and Steve B. Furber. «SpiNNaker 2: A 10 Million Core Processor System for Brain Simulation and Machine Learning». In: *arXiv preprint* (2019). DOI: 10.48550/arXiv.1911.02385. eprint: arXiv:1911.02385 (cit. on p. 8).

[31] Eric Müller et al. «The Operating System of the Neuromorphic BrainScaleS-1 System». In: *arXiv preprint* (2020). DOI: 10.48550/arXiv.2003.13749. eprint: arXiv:2003.13749 (cit. on p. 8).

[32] Christian Pehle et al. «The BrainScaleS-2 accelerated neuromorphic system with hybrid plasticity». In: *arXiv preprint* (2022). DOI: 10.48550/arXiv.2201.11063. eprint: arXiv:2201.11063 (cit. on p. 8).

[33] SynSense. *Xylo: Overview of the Xylo™ Family Neuromorphic Devices.* Rockpool documentation. Accessed: 2025-11-26. URL: https://rockpool.ai/devices/xylo-overview.html (cit. on p. 8).

[34] Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers. *The SIMON and SPECK Families of Lightweight Block Ciphers.* Cryptology ePrint Archive, Paper 2013/404. 2013. URL: https://eprint.iacr.org/2013/404 (cit. on p. 8).

[35] Ben Varkey Benjamin et al. «Neurogrid: A Mixed-Analog-Digital Multichip System for Large-Scale Neural Simulations». In: *Proceedings of the IEEE* 102.5 (2014), pp. 699–716. DOI: 10.1109/JPROC.2014.2313565 (cit. on p. 8).

[36] D. Casanueva-Morato et al. «A bio-inspired hardware implementation of an analog spike . . . » In: *Neurocomputing* (2025). DOI: 10.1016/j.neucom.2025.S0925231225025640 (cit. on p. 8).

[37] Ole Richter, Chenxi Wu, Adrian M. Whatley, German Köstinger, Carsten Nielsen, Ning Qiao, and Giacomo Indiveri. «DYNAP-SE2: a scalable multi-core dynamic neuromorphic asynchronous spiking neural network processor». In: *arXiv preprint* (2023). DOI: `10.48550/arXiv.2310.00564`. eprint: `arXiv:2310.00564` (cit. on p. 8).

[38] Alessandro Pierro, Steven Abreu, Jonathan Timcheck, Philipp Stratmann, Andreas Wild, and Sumit Bam Shrestha. «Accelerating Linear Recurrent Neural Networks for the Edge with Unstructured Sparsity». In: *arXiv preprint arXiv:2502.01330* (2025). Accessed: 2025-11-25. URL: `https://arxiv.org/pdf/2502.01330v1` (cit. on p. 11).

[39] Jason Yik et al. *Modeling and Optimizing Performance Bottlenecks for Neuromorphic Accelerators.* 2025. arXiv: `2511.21549 [cs.AR]`. URL: `https://arxiv.org/abs/2511.21549` (cit. on pp. 11, 46, 47).

[40] Jason K. Eshraghian, Max Ward, Emre Neftci, Xinxin Wang, Gregor Lenz, Girish Dwivedi, Mohammed Bennamoun, Doo Seok Jeong, and Wei D. Lu. «Training Spiking Neural Networks Using Lessons From Deep Learning». In: *Proceedings of the IEEE* 111.9 (Sept. 2023). Accessed: 2025-11-24. DOI: `10.1109/JPROC.2023.3309605`. URL: `https://ieeexplore.ieee.org/document/10183703` (cit. on p. 13).

[41] Open Neuromorphic. *snnTorch.* https://open-neuromorphic.org/neuromorphic-computing/s
Accessed: 2025-11-24. 2025 (cit. on p. 13).

[42] Intel Labs. *Lava Software Framework — An open-source framework for neuromorphic computing.* `https://lava-nc.org/index.html`. Accessed: 2025-11-24. 2024 (cit. on p. 14).

[43] GeeksforGeeks. *Introduction to Recurrent Neural Network.* GeeksforGeeks online tutorial. Last updated 07 Oct, 2025. 2025. URL: `https://www.geeksforgeeks.org/introduction-to-recurrent-neural-network/` (cit. on p. 15).

[44] Robin M. Schmidt. «Recurrent Neural Networks (RNNs): A gentle Introduction and Overview». In: *arXiv preprint* (2019). DOI: `10.48550/arXiv.1912.05911`. eprint: `arXiv:1912.05911` (cit. on p. 15).

[45] Sepp Hochreiter. «The Vanishing Gradient Problem During Learning Recurrent Neural Nets and Problem Solutions». In: *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems* 6.2 (1998), pp. 107–116. DOI: `10.1142/S0218488598000094` (cit. on p. 15).

[46] Sepp Hochreiter and Jürgen Schmidhuber. «Long Short-Term Memory». In: *Neural Computation* 9.8 (1997), pp. 1735–1780. DOI: `10.1162/neco.1997.9.8.1735` (cit. on p. 15).

[47] Kyunghyun et al. Cho. «Learning phrase representations using RNN encoder-decoder for statistical machine translation». In: *arXiv preprint arXiv:1406.1078* (2014) (cit. on p. 15).

[48] Kyunghyun Cho, Bart van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio. «On the Properties of Neural Machine Translation: Encoder–Decoder Approaches». In: *Proceedings of SSST-8, Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation (SSST 2014)*. 2014, pp. 103–111. DOI: 10.3115/v1/W14-4012. URL: https://aclanthology.org/W14-4012/ (cit. on p. 15).

[49] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. «Neural machine translation by jointly learning to align and translate». In: *arXiv preprint arXiv:1409.0473* (2014) (cit. on p. 15).

[50] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. «Sequence to sequence learning with neural networks». In: *arXiv preprint arXiv:1409.3215* (2014) (cit. on p. 15).

[51] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. «Speech recognition with deep recurrent neural networks». In: *ICASSP 2013 - 2013 IEEE International Conference on Acoustics, Speech and Signal Processing*. IEEE, 2013, pp. 6645–6649 (cit. on p. 15).

[52] John T. Connor, David Martin, and Les Atlas. «Recurrent neural networks and robust time series prediction». In: *IEEE Transactions on Neural Networks* 5.2 (1994), pp. 240–254. DOI: 10.1109/72.279181 (cit. on p. 15).

[53] Microsoft. *Neural Network Intelligence (NNI)*. Version stable. 2023. URL: https://nni.readthedocs.io/en/stable/ (cit. on p. 16).

[54] Benedetto Leto. «LIF-based Legendre Memory Unit: neuromorphic redesign of a recurrent architecture and its application to human activity recognition». Tesi di laurea magistrale, Webthesis PoliTO, identificatore 32998. MA thesis. Politecnico di Torino, 2024. URL: https://webthesis.biblio.polito.it/secure/32998/1/tesi.pdf (cit. on p. 19).

[55] Benjamin Cramer, Yannik Stradmann, Johannes Schemmel, and Friedemann Zenke. «The Heidelberg spiking datasets for the systematic evaluation of spiking neural networks». In: *arXiv preprint arXiv:1910.07407* (2019). Accessed: 2025-11-25. DOI: 10.48550/arXiv.1910.07407. URL: https://arxiv.org/pdf/1910.07407 (cit. on pp. 19, 20).

[56] Y. Sun, X. Li, Z. Ma, et al. «Parameter-free Attention for Delay Spiking Neural Networks». In: *Neural Networks* 184 (2025), p. 107154. DOI: 10.1016/j.neunet.2025.107154 (cit. on p. 21).

[57] M. Baronig, A. Wendel, and K. Meier. *Recurrent Spiking Neural Networks with Adaptive LIF Neurons and Symplectic-Euler Discretization*. 2024. arXiv: 2408.07517 (cit. on p. 21).

[58] A. Hammouamri, J. Wu, and F. Zenke. *Fully Connected Spiking Neural Networks with Learned Delays*. 2023. arXiv: 2306.17670 (cit. on p. 21).

[59] A. Bittar and P. N. Garner. «Recurrent Spiking Neural Networks with Adaptation for Speech Recognition». In: *Frontiers in Neuroscience* 16 (2022), p. 865897. DOI: 10.3389/fnins.2022.865897 (cit. on p. 21).

[60] T. Nowotny et al. «Recurrent Spiking Neural Networks with Delay Line Input and Data Augmentation». In: *Neuromorphic Computing and Engineering* (2025). DOI: 10.1088/2634-4386/ada852 (cit. on p. 21).

[61] T. Mészáros, T. Kiss, and G. Cserey. *Recurrent Spiking Neural Networks with Delay Learning*. 2025. arXiv: 2501.07331 (cit. on p. 21).

[62] Y. Sun, Z. Ma, and X. Li. «Feed-forward Spiking Neural Networks with Adaptive Axonal Delays». In: *IEEE International Joint Conference on Neural Networks (IJCNN)*. 2023. DOI: 10.1109/IJCNN54540.2023.10094768 (cit. on p. 21).

[63] Q. Yu, B. Yin, Y. Liu, et al. «Spatio-Temporal Attention for Spiking Neural Networks». In: *Frontiers in Neuroscience* 16 (2022), p. 1079357. DOI: 10.3389/fnins.2022.1079357 (cit. on pp. 21, 52).

[64] Y. Yao, Y. Li, and J. Wu. *Temporal Attention Mechanisms for Recurrent Spiking Neural Networks*. 2021. arXiv: 2107.11711 (cit. on p. 21).

[65] L. D'Agostino, F. Conti, et al. *Feed-forward Spiking Neural Networks with Random Dendritic Delays: Simulation and Hardware Implementation on RRAM*. 2023. arXiv: 2312.08960 (cit. on p. 21).

[66] B. Yin, F. Corradi, and S. M. Bohté. *Effective and Efficient Recurrent Spiking Neural Networks through Adaptation and Learning of Neuron Dynamics*. 2020. arXiv: 2005.11633 (cit. on p. 21).

[67] J. Rossbroich, A. Kugele, T. Pfeil, et al. «Recurrent Convolutional Spiking Neural Networks with Fluctuation-driven Initialization». In: *Neuromorphic Computing and Engineering* 2 (2022), p. 034016. DOI: 10.1088/2634-4386/ac97bb (cit. on p. 21).

[68] B. Cramer, Y. Stradmann, J. Schemmel, and F. Zenke. «The Heidelberg Spiking Datasets and Their Applications: Data Augmentation and Noise Injection in Spiking Neural Networks». In: *IEEE Transactions on Neural Networks and Learning Systems* (2020). DOI: 10.1109/TNNLS.2020.3044364 (cit. on p. 21).

[69] N. Perez-Nieves and D. F. M. Goodman. *Heterogeneous Time Constants in Spiking Neural Networks.* 2021. bioRxiv: `2020.12.18.423468` (cit. on p. 21).

[70] B. Cramer, Y. Stradmann, J. Schemmel, and F. Zenke. «Recurrent Spiking Neural Networks for Audio Processing». In: *IEEE Transactions on Neural Networks and Learning Systems* (2020). DOI: `10.1109/TNNLS.2020.3044364` (cit. on p. 21).

[71] B. Cramer, Y. Stradmann, J. Schemmel, and F. Zenke. «Feed-forward Spiking Neural Networks for Audio Processing». In: *IEEE Transactions on Neural Networks and Learning Systems* (2020). DOI: `10.1109/TNNLS.2020.3044364` (cit. on p. 21).

[72] M. Schöne, T. Zhang, and C. Posch. *Event-based Linear State Space Model.* 2024. arXiv: `2404.18508` (cit. on p. 21).

[73] Benjamin Cramer, Yannik Stradmann, Johannes Schemmel, and Friedemann Zenke. «The Heidelberg spiking data sets for the systematic evaluation of spiking neural networks». In: *IEEE Transactions on Neural Networks and Learning Systems* 33.7 (2022), pp. 2744–2757. DOI: `10.1109/TNNLS.2020.3044364` (cit. on p. 21).

[74] Benjamin Rueckauer, Igor Lungu, Yulia Sandamirskaya, and Robert J. Vogels. «Conversion of continuous-valued deep networks to efficient event-driven networks for image classification». In: *Frontiers in Neuroscience* 11 (2017), p. 682. DOI: `10.3389/fnins.2017.00682` (cit. on p. 22).

[75] Abhronil Sengupta, Yansong Ye, Rui Wang, Chao Liu, and Kaushik Roy. «Going deeper in spiking neural networks: VGG and residual architectures». In: *Frontiers in Neuroscience* 13 (2019), p. 95. DOI: `10.3389/fnins.2019.00095` (cit. on p. 22).

[76] Yujie Cao, Yu Chen, and Deepak Khosla. «Spiking deep convolutional neural networks for energy-efficient object recognition». In: *International Journal of Computer Vision* 113 (2015), pp. 54–66. DOI: `10.1007/s11263-014-0772-3` (cit. on p. 22).

[77] Peter U. Diehl, Daniel Neil, Jakob Binas, Matthew Cook, Shih-Chii Liu, and Michael Pfeiffer. «Fast-classifying, high-accuracy spiking deep networks through weight and threshold balancing». In: *2015 International Joint Conference on Neural Networks (IJCNN).* IEEE, 2015, pp. 1–8. DOI: `10.1109/IJCNN.2015.7280719` (cit. on p. 22).

[78] Peter Blouw and Chris Eliasmith. «Event-Driven Signal Processing with Neuromorphic Computing Systems». In: *ICASSP 2020 - 2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP).* 2020, pp. 8534–8538. DOI: `10.1109/ICASSP40776.2020.9053043` (cit. on p. 22).

[79] Vittorio Fra, Evelina Forno, Riccardo Pignari, Terrence C Stewart, Enrico Macii, and Gianvito Urgese. «Human activity recognition: suitability of a neuromorphic approach for on-edge AIoT applications». In: *Neuromorphic Computing and Engineering* 2.1 (Feb. 2022), p. 014006. DOI: `10.1088/2634-4386/ac4c38`. URL: `https://dx.doi.org/10.1088/2634-4386/ac4c38` (cit. on p. 22).

[80] Brian DePasquale, Mark M. Churchland, and L. F. Abbott. *Using Firing-Rate Dynamics to Train Recurrent Networks of Spiking Model Neurons*. 2016. arXiv: `1601.07620 [q-bio.NC]`. URL: `https://arxiv.org/abs/1601.07620` (cit. on p. 22).

[81] Ali Lotfi-Rezaabad and Sriram Vishwanath. «Long Short-Term Memory Spiking Networks and Their Applications». In: *CoRR* abs/2007.04779 (2020). arXiv: `2007.04779`. URL: `https://arxiv.org/abs/2007.04779` (cit. on p. 22).

[82] Vittorio Fra, Evelina Forno, Riccardo Pignari, Terrence C. Stewart, Enrico Macii, and Gianvito Urgese. «Human activity recognition: suitability of a neuromorphic approach for on-edge AIoT applications». In: *Neuromorphic Computing and Engineering* 2.1 (2022), p. 014006. DOI: `10.1088/2634-4386/ac4c38` (cit. on p. 32).

[83] Thomas Shoesmith, James C. Knight, Balázs Mészáros, Jonathan Timcheck, and Thomas Nowotny. *Eventprop training for efficient neuromorphic applications*. 2025. arXiv: `2503.04341` (cit. on pp. 39, 42, 52).