



ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

DOTTORATO DI RICERCA IN
INGEGNERIA ELETTRONICA, TELECOMUNICAZIONI E
TECNOLOGIE DELL'INFORMAZIONE

Ciclo 36

Settore Concorsuale: 09/H1 - SISTEMI DI ELABORAZIONE DELLE INFORMAZIONI

Settore Scientifico Disciplinare: ING-INF/05 - SISTEMI DI ELABORAZIONE DELLE INFORMAZIONI

DIRECTLY TRAINING SPIKING NEURAL NETWORKS FOR CYBER-
PHYSICAL SYSTEMS: FROM SUPERVISED TO REINFORCEMENT
LEARNING

Presentata da: Luca Zanatta

Coordinatore Dottorato

Aldo Romani

Supervisore

Andrea Acquaviva

Co-supervisor

Andrea Bartolini

Luca Benini

Esame finale anno 2024

ALMA MATER STUDIORUM - UNIVERSITY OF BOLOGNA

**Directly Training Spiking Neural
Networks for Cyber-Physical Systems:
From Supervised to Reinforcement
Learning**

by

Luca Zanatta

A thesis submitted for the degree of
Doctor of Philosophy

in the
Faculty of Engineering
Department of Electrical, Electronic and Information Engineering “G. Marconi”
(DEI)

February 2024

Acknowledgements

Il mio percorso da dottorato è stato probabilmente come la media di tutti gli altri dottorati: molto vivace, pieno di sfide, imprevisti e, ogni tanto, qualche soddisfazione. Di certo mi ha insegnato molto da diversi punti di vista ed è stato motivo di una crescita personale e professionale non indifferente. Nel corso di questi 3 anni, sono state diverse le figure che mi hanno aiutato, sostenuto e insegnato tanto. Ringrazio il mio supervisor Andrea Acquaviva, per la fiducia che mi ha dato e per aver sempre creduto nelle mie capacità. Tutti i progetti, le riunioni e gli eventi in cui ci siamo confrontati sono stati fonte di riflessione per il mio percorso e sono certo che terrò bene in mente le cose che ho imparato grazie a lui. Il mio co-supervisor Andrea Bartolini ha avuto un importante ruolo in questi anni, soprattutto per avermi insegnato ad affrontare ed elaborare le criticità trovate lungo il percorso. Durante i 6 mesi a Copenhagen, ho avuto modo di lavorare con la prof.ssa Silvia Tolu. La ringrazio per l'accoglienza mostrata e per il lavoro che siamo riusciti a fare tra i diversi progetti svolti. Ai miei colleghi Martin, Alberto ed Emanuele riservo un ringraziamento speciale per tutte le camminate e chiacchierate fatte in questi anni. Ringrazio i miei cari amici Momo, Nico e Roberto. Ci conosciamo ormai da anni e abbiamo passato molti momenti insieme, dallo studio, ai videogames, ai supporti reciproci fino al famoso evento "Birrone per l'ubriacone". Ringrazio di cuore i miei genitori che mi hanno sempre sostenuto, ascoltato ed aiutato. Sono fortunato ad avere dei genitori con cui ho potuto condividere anche questa parte della mia vita, trovando sempre un appoggio sicuro e un conforto costante. I miei fratelli Fabio, Mauro e Mattia sono stati fonte di enorme distrazione tra corsi di Python e partite a LOL. Inutile dire che il la spensieratezza e il divertimento che mi hanno dato sono impareggiabili. Ringrazio mia sorella Yvonne, mio cognato Matteo e il mio piccolo-grande nipotino Marco per tutti i momenti divertenti trascorsi insieme. Infine, Irene che è sempre stata presente in ogni momento di questo dottorato. A lei rivolgo un grazie speciale per esserci sempre stata.

Acronyms

ACC Accuracy. xi, 43

AI Artificial Intelligence. 6, 35

ALIF Adaptive Leaky Integrate-and-Fire. 30, 32, 42, 45, 46

ANN Artificial Neural Network. vi, 1–3, 6, 7, 29, 33, 34, 36, 54, 56

AUC Area Under the Curve. 66–68

BP BackPropagation. 11

BPTT BackproPagation Through Time. vi, 2, 36, 44

BSA Ben’s Spiker Algorithm. 33

CNN Convolutional Neural Network. vi, 3, 4, 9, 10, 59, 67, 69–71

CPS Cyber-Physical System. vi, 2

D2QN Double DQN. 25, 56

DFT Discrete Fourier Transform. 39

DL Deep Learning. 1, 6, 7

DVS Dynamic Vision Sensor. vi, viii, x, 3, 54–58, 62, 63

eLSNN embedded LSNN. vi, ix–xi, 3, 46–52

FFT Fast Fourier Transform. 40, 47, 51, 52

IF Integrate-and-Fire. 30

LIF Leaky Integrate-and-Fire. 30–33, 42, 55, 56, 58

-
- LSNN** Long Short-Term SNN. vi, ix, xi, 2, 3, 36, 39–42, 44–47, 49, 52, 53
- LSTM** Long Short-Term Memory. 36, 44
- LTD** Long-Term Depression. 33
- LTP** Long-Term Potentiation. 33
- MC** Monte Carlo. 22, 23
- MCC** Matthews Correlation Coefficient. ix, xi, 43, 46–48
- MDP** Markov Decision Process. 16–18
- MEMS** Micro-ElectroMechanical System. vi, 2, 3, 35, 36, 52
- MLP** Multi-Layer Perceptron. ix, 8, 9, 12
- MSE** Mean Squared Error. 22
- ODE** Ordinary Differential Equation. 1, 29
- POMDP** Partially Observable Markov Decision Process. 18
- PPO** Proximal Policy Optimization. 25, 27, 28
- PSD** Power Spectrum Density. 38
- RL** Reinforcement Learning. vi, 1–3, 7, 16, 17, 21, 22, 54, 56, 62, 63, 68, 69
- RNN** Recurrent Neural Network. ix, 11
- ROC** Rank-Order Coding. 33, 40
- SGD** Stochastic Gradient Descend. 13
- SHM** Structural Health Monitoring. vi, xi, 1, 2, 35, 36, 49, 50, 52
- SL** Supervised Learning. 7
- SNE** Sparse Neural Engine. 55–57, 59, 61, 62, 69, 70
- SNN** Spiking Neural Network. vi, ix, xi, 1–4, 29, 33–36, 38, 39, 41, 43, 44, 47, 52, 54–61, 68–71
- SOD** Send On Delta. 32
- SPI** Serial Peripheral Interface. x, xi, 3, 49–52

STBP Spatio-Temporal BackPropagation. vi, 3, 59

STDP Spike-Timing-Dependent Plasticity. 33

TCN Temporal Convolutional Network. 36, 44

TD Temporal Difference. 23

TTFS Time To First Spike. 32

UAV Unmanned Aerial Vehicle. vi, 3

UL Unsupervised Learning. 7

Abstract

Spiking Neural Networks (SNNs) are a type of Artificial Neural Networks (ANNs) inspired by the structure and function of biological nervous systems. Unlike traditional neural networks, SNNs use discrete spiking signals similar to neuron communication in the human brain. This characteristic makes them ideal for real-time processing and energy-efficient applications in Cyber-Physical Systems (CPSs). The event-driven behavior of SNNs provides efficient information processing and decision-making capabilities within CPS domains.

Advancements in Structural Health Monitoring (SHM) are driven by the convergence of IoT and machine learning. Recent studies have shown that low-cost MEMS accelerometers can effectively monitor vibrations, with neural networks analyzing data streams. In this research, I propose utilizing SNNs to detect early damage in motorway bridges. Particularly, Long Short-Term SNNs (LSNNs) show promise but involve complex learning processes. This study examines the feasibility of using LSNNs for SHM and compares their accuracy in determining structural health to other ANN models and training algorithms. The findings suggest that SNNs can effectively identify structural damage with comparable levels of accuracy to ANNs trained using Backpropagation Through Time and e-prop methods. Furthermore, I conducted a thorough analysis of an optimized embedded LSNN that utilized spike-based and current-based input encoding. The former implementation demonstrated a 54% reduction in execution time compared to naive versions. However, it should be noted that spike-based encoding necessitated larger input vectors, leading to longer pre-processing and sensor access times. Ultimately, transitioning this coding approach to the sensor level offers great potential for creating a more energy-efficient monitoring system.

Moreover, I investigate the application of SNNs in an obstacle avoidance task for a Unmanned Aerial Vehicle (UAV) using a RL algorithm. The implementation involves training the SNN directly and utilizing a DVS as an event-based input source. To perform RL, I employ an adapted Spatio-Temporal BackPropagation (STBP) algorithm. I evaluate the performance of the SNN by comparing it to a Convolutional Neural Network (CNN) performing the same task. Additionally, I develop and train embedded implementations of SNNs to measure latency and throughput in real-world deployments. In addition, to evaluating obstacle avoidance capabilities, a comparison between SNNs and CNNs reveals that SNNs exhibit superior energy efficiency. Specifically, SNNs demonstrate a 6x decrease in energy consumption compared to CNNs.

Contents

Acknowledgements	ii
Abstract	vi
List of Figures	ix
List of Tables	xi
1 Introduction	1
1.1 Contributions	2
1.1.1 Spiking Neural Networks for Structural Health Monitoring	2
1.1.2 Spiking Neural Networks for Obstacle Avoidance	3
1.2 List of Publications	4
2 Deep Learning	6
2.1 Basics	7
2.2 Architectures	8
2.2.1 Multi Layer Perceptron	8
2.2.2 Convolutional Neural Networks	9
2.2.3 Recurrent Neural Networks	11
2.3 Backpropagation	11
2.4 Optimizers	12
2.4.1 Stochastic Gradient Descend	13
2.4.2 Adam	13
2.5 Learning Paradigms	14
2.5.1 Supervised Learning	14
2.5.1.1 Regression	14
2.5.1.2 Classification	15
2.5.2 Reinforcement Learning	15
2.5.2.1 Basics	16
2.5.2.2 Learnable Functions	19
2.5.2.3 Double Deep Q-Network	22
2.5.2.4 Proximal Policy Optimization	25
3 Spiking Neural Networks	29

3.1	Neuron Models	30
3.1.1	Integrate-and-Fire Model	30
3.1.2	Leaky Integrate-and-Fire Model	31
3.1.3	Adaptive Leaky Integrate-and-Fire Model	32
3.2	Input Encoding	32
3.3	Training Methods	33
4	Supervised Learning with SNNs	35
4.1	Dataset	37
4.2	Preprocessing	38
4.3	LSNN	41
4.4	Results	44
4.4.1	Comparison among LSNNs, TCN, and LSTM	44
4.4.2	Comparison between current-driven inputs and event-driven	45
4.4.2.1	Accuracy vs. N and t_{inf} hyperparameters	46
4.4.3	Execution time vs. activity-factors	48
4.4.4	Event-driven vs. Current-driven	50
4.5	Conclusion	52
5	Reinforcement Learning with SNNs	54
5.1	Dynamic Vision Sensor	54
5.2	Neuromorphic Platforms	55
5.3	SNNs with D2QN	56
5.3.1	Spiking Neural Network and DVS Input Model	57
5.3.2	The Adaptation of STBP for Q-function Estimation	59
5.3.3	Spiking Neural Network on SNE Embedded Neuromorphic Accelerator	61
5.4	Training and Evaluation Framework	62
5.4.1	Simulation Environment and DVS Model	62
5.4.2	Training and Evaluation Setup	62
5.5	Results	66
5.5.1	Neural Networks Performance	66
5.5.2	Spiking Neural Network Footprint	68
5.6	Conclusions	70
6	Conclusion	72
	Bibliography	74

List of Figures

2.1	Two layers MLP with 4 input features, 5 hidden neurons, and 3 output neurons.	8
2.2	Example of RNN. On the left, the RNN is folded, while on the right, it is unfolded.	11
2.3	The reinforcement training loop involves the agent taking an action a_t in response to which the environment transitions and provides information about the new state s_t as well as the reward received, denoted by r_t	16
4.1	Monitoring system installation.	37
4.2	Mean natural frequency shift before and after scheduled maintenance. . .	37
4.3	First data preprocessing and encoding pipelines.	38
4.4	Second data preprocessing and encoding pipelines.	40
4.5	SNN's architecture. The blue ensemble represents the forward part, while the green one is the backward part. The former is used in the mono-directional LSNN while, in the bi-directional, the network has both components.	41
4.6	The graph on the right shows how accuracy changes when increasing the exposure time on T50-W05 and T50-W10 datasets. On the left, we can see a distribution of accuracy based on the number of spectral features and dataset type.	44
4.7	The x-axis represents the number of Inference Ticks, while the y-axis shows the MCC. Each data point depicts a different LSNN configuration with its unique set of hyperparameters. The percentage indicated corresponds to the proportion of SNNs plotted out of the total number of configurations explored, which was 1728.	47
4.8	A) Cycle count for the event-driven eLSNN in different configurations. It includes variations in input numbers and inference ticks, computed on minimal, median, and maximal spike activities on the input layer (x). B) Cycle count for the current-driven eLSNN with variations in spike activities on the recurrent/hidden layer (z), computed on minimal, median, and maximal settings. Different colours represent different numbers of input configurations for each eLSNN network. All networks have 10 inference ticks.	48
4.9	A) A breakdown of the cycle count for the best configurations of event-driven (event) and current-driven (current) eLSNN, computed on median activity conditions. B) The distribution of spikes in the input for different configurations of the event-driven eLSNN, with corresponding input numbers and inference ticks.	49

4.10	The current consumption patterns of the best eLSNN networks performing SHM applications are shown in the waveforms. Two sets of inputs were used: current inputs (top) and event inputs (bottom). The waveforms were obtained by reducing the MCU clock to 16 MHz and setting the SPI clock to 2 MHz, allowing for a clearer depiction of different stages in the application.	49
5.1	Comparison between an RGB image and a DVS image.	54
5.2	The setup process involves converting RGB frames into DVS frames, which are then inputted into the network. The network generates a Q-function as output, which is utilized to determine the appropriate action for the drone to execute.	56
5.3	The arrangement of obstacles in the three lanes. The first row depicts the representation from the top, while the second row shows the representation from the left side.	63
5.4	Performance of the networks in three lanes using Normalized AUC, which was computed using bins of 10 meters.	66
5.5	Convolutional layers are indicated as “Cn”, Linear layers are indicated as “Ln”, and “IN” indicates the network input activity from the event camera.	68

List of Tables

4.1	SNN Experiment Results. ACC is the accuracy, MCC is the Matthews Correlation Coefficient, T represents the length in seconds of the samples, and W is the number of sub-segments.	43
4.2	LSNN parameters explored	45
4.3	The power analysis of the SHM application featuring the best LSNNs tested. The SPI master transfers data at a clock speed of 8 MHz. The SPI stage includes three components: i) transferring data via SPI, ii) reconfiguring the system-on-chip's clock to 168 MHz after waking up from SLEEP mode, and iii) converting sensor data from raw integers to floating-point format.	49
4.4	The power analysis of the SHM application featuring the best eLSNNs tested. The SPI master transfers data at a clock speed of 8 MHz. The SPI stage includes three components: i) transferring data via SPI, ii) reconfiguring the system-on-chip's clock to 168 MHz after waking up from SLEEP mode, and iii) converting sensor data from raw integers to floating-point format.	50
5.1	Three instances of neuron output values calculated using different approaches. In these illustrations, I utilize a duration of three ticks ($N = 3$).	59
5.2	Comparison between the features of a fully customized neuron and the neuron in SNE.	60
5.3	Networks architecture. HR-LIF stands for Hard-reset LIF and NS LIF for Non-spiking LIF	61
5.4	Simulation parameters value and description.	65
5.5	Networks performance.	66
5.6	Number of the total operation and inference energy for the proposed approach and the baseline CNN executed on a reference state-of-the-art hardware accelerator [1]. Estimates are based on the highest energy per inference reported by the author.	68

Chapter 1

Introduction

Spiking Neural Networks (SNNs), which are the third generation of Artificial Neural Networks (ANNs), have gained significant attention in recent years due to their resemblance to neural structures found in mammalian brains. This sets them apart from earlier generations of ANNs [2].

The fundamental distinction between traditional neural networks, classified as second-generation ANNs, and SNNs lies in their central computational component - the neuron. In conventional ANNs, neurons are characterized by activation functions that calculate their outputs based on input values. Conversely, SNNs utilize spiking neurons that are distinguished by a collection of Ordinary Differential Equations (ODEs). These ODEs describe the spiking neuron as a dynamic system, where the membrane potential accumulates over time as a concealed variable. When this membrane potential crosses a predetermined threshold in an upward trajectory, the neuron generates spikes known as events. These spikes enable asynchronous communication among neurons, contrasting with the synchronous interactions seen in traditional ANNs. This paradigm shift opens up new possibilities for neural network architecture.

SNNs offer numerous benefits due to their inherent characteristics, including low latency, fast inference speed, and energy efficiency. Additionally, the temporal aspect of SNN neurons makes them highly effective in domains that rely on time series data such as Natural Language Processing, Structural Health Monitoring (SHM), and Reinforcement Learning (RL).

This thesis consists of several chapters that cover various aspects of Deep Learning (DL), ANNs, and SNNs. In the first chapter, I provide background information on DL and ANNs. The second chapter delves into the fundamentals of SNNs. Moving forward, in the third chapter, I present my work utilizing SNNs for SHM data analysis.

Additionally, in the fourth chapter, we explore my research combining SNNs with RL. Finally, concluding remarks are discussed in the last chapter.

1.1 Contributions

In this thesis, I will discuss two distinct aspects of SNNs in the Cyber-Physical System (CPS) field: SHM and Obstacle Avoidance. The focus of structural health monitoring is on classifying time series data, while obstacle avoidance deals with dynamic environments.

1.1.1 Spiking Neural Networks for Structural Health Monitoring

The field of SHM is rapidly advancing, combining the potential of the Internet of Things and state-of-the-art machine learning technologies. Recent research has demonstrated the effectiveness of low-cost MEMS accelerometers in monitoring building vibrations, with neural networks used for analyzing the generated data streams. In this work, I propose a novel approach to SHM using SNNs applied to MEMS data for early detection of infrastructural damages in a motorway bridge. SNNs are inspired by the human brain and offer potential benefits such as compactness and energy efficiency compared to traditional network models. Specifically, Long Short-Term SNNs (LSNNs) have shown great promise in analyzing data streams but require an involved learning process.

This research investigates the viability of LSNNs for SHM and compares their accuracy with other ANN models and training models. The results indicate that SNNs can effectively determine if a structure is healthy or damaged, achieving similar levels of accuracy as ANNs. To achieve this, Backpropagation Through Time (BPTT) and e-prop were utilized as training algorithms. Additionally, it was observed that the inference times meet the real-time demands of SHM.

In addition, this research introduces an innovative method for detecting damage to infrastructure through the use of edge devices. By employing a novel algorithm inspired by the human brain, the solution utilizes LSNNs which are known for their energy efficiency and compactness. The goal is to accurately identify structural damage by analyzing data from affordable Micro-ElectroMechanical System (MEMS) accelerometers directly at the sensor node. One key aspect is optimizing how MEMS data is encoded in order to enhance SNN performance on low-power microcontrollers. The performance and energy consumption of LSNN on a hardware prototype sensor node, which utilized

an STM32 embedded microcontroller and a digital MEMS accelerometer, were analyzed. An environment called Hardware-in-the-Loop was used to evaluate the system using data generated by virtual sensors connected to the physical microcontroller via an SPI interface. This allowed for the evaluation of the system using real viaduct data streams. The study also explored different on-sensor encoding techniques that simulated a bio-inspired sensor capable of generating events instead of accelerations.

The findings revealed that the optimized embedded LSNN (eLSNN), utilizing spike-based input encoding, demonstrated a 54% reduction in execution time compared to a naive LSNN algorithm implementation found in current literature. The optimized eLSNN required approximately 47 kCycles, which is comparable to the data transfer cost from the Serial Peripheral Interface (SPI) interface. However, the spike-based encoding technique necessitated larger input vectors to achieve the same classification accuracy, resulting in longer pre-processing and sensor access times. Overall, event-based encoding techniques slightly increased execution times (1.49x) while maintaining similar energy consumption levels. Transitioning this coding approach to the sensor level has the potential to eliminate this limitation and create a more energy-efficient monitoring system as a whole.

1.1.2 Spiking Neural Networks for Obstacle Avoidance

Recent research has highlighted the potential of SNNs to provide energy-efficient and fast inference capabilities on neuromorphic hardware. These studies have demonstrated that SNNs can achieve competitive performance in comparison to traditional ANNs for various classification tasks.

In this research study, I present a novel approach to address the obstacle avoidance task for an Unmanned Aerial Vehicle (UAV) using a RL algorithm implemented with SNNs. My implementation leverages the use of a Dynamic Vision Sensor (DVS) as an event-based input source. Unlike previous approaches that utilize hybrid SNNs without direct training, my method involves training the SNN directly. To enable this, I propose an adaptation of the Spatio-Temporal BackPropagation (STBP) algorithm, specifically tailored for RL purposes. The key focus of my investigation is on comparing the performance of the SNN with a state-of-the-art Convolutional Neural Network (CNN) designed for solving the same obstacle avoidance task. For evaluation, I created a realistic training pipeline based on the AirSim framework.

In order to achieve accurate measurements of latency and throughput in embedded deployments, I have developed and trained three different versions of embedded SNNs. These implementations showcase the latest advancements in neuromorphic technology.

Apart from assessing obstacle avoidance performance, I also conducted a comparison between SNN and CNN algorithms. The findings indicate that the SNN algorithm surpasses the CNN in terms of energy efficiency, demonstrating a $6\times$ reduction in energy consumption. Additionally, I investigated various hardware implementations of SNN and analyzed aspects like energy consumption and spiking activity to gain further insights.

1.2 List of Publications

- Francesco Barchi, **Luca Zanatta**, Emanuele Parisi, Alessio Burrello, Davide Brunelli, Andrea Bartolini, and Andrea Acquaviva. *Spiking Neural Network-Based Near-Sensor Computing for Damage Detection in Structural Health Monitoring*. In: Future Internet 13.8 (2021), p. 219.
- **Luca Zanatta**, Francesco Barchi, Alessio Burrello, Andrea Bartolini, Davide Brunelli, and Andrea Acquaviva. *Damage Detection in Structural Health Monitoring with Spiking Neural Networks*. In: 2021 IEEE International Workshop on Metrology for Industry 4.0 & IoT (MetroInd4.0&IoT). IEEE. 2021, pp. 105–110.
- **Luca Zanatta**, Francesco Barchi, Andrea Bartolini, and Andrea Acquaviva. *Artificial versus spiking neural networks for reinforcement learning in UAV obstacle avoidance*. In: Proceedings of the 19th ACM International Conference on Computing Frontiers. 2022, pp. 199–200.
- Alberto Musa, **Luca Zanatta**, Francesco Barchi, Bartolini Andrea, and Acquaviva Andrea. *A Method for Accelerated Simulations of Reinforcement Learning Tasks of UAVs in AirSim*. In: SIMUL 22 (2022).
- **Luca Zanatta**, Alfio Di Mauro, Francesco Barchi, Andrea Bartolini, Luca Benini, and Andrea Acquaviva. *Directly-trained Spiking Neural Networks for Deep Reinforcement Learning: Energy efficient implementation of event-based obstacle avoidance on a neuromorphic accelerator*. In: Neurocomputing (2023), p. 126885.
- Udayanga K. N. G. W. Gamage, **Luca Zanatta**, Matteo Fumagalli, Cesar Cadena, and Silvia Tolu. *Event-Based Classification of Defects in Civil Infrastructures with Artificial and Spiking Neural Networks*. In: Advances in Computational Intelligence. Ed. by Ignacio Rojas, Gonzalo Joya, and Andreu Catala. Cham: Springer Nature Switzerland, 2023, pp. 629–640.
- Amirhossein Moallemi, **Luca Zanatta**, Alessio Burrello, Mattia Salvaro, Monica Longo, Paola Darò, Francesco Barchi, Davide Brunelli, Luca Benini, and Andrea

Acquaviva. *Unsupervised Vehicle Classification Using a Structural Health Monitoring System*. In: 14th International Workshop on Structural Health Monitoring (IWSHM). 2023.

Chapter 2

Deep Learning

DL is a branch of Artificial Intelligence (AI) in which we teach a computer how to learn and to do “intelligent” things.

Initially, the approach was to stand in front of a whiteboard and brainstorm on how to create a code that could handle all the possible cases. Then write the code, which can involve hundreds of lines, to achieve the task or solve a problem. After a good amount of tests and trials, you were finally able to get a working solution 100% of the time. However, as problems and tasks became more complex, this traditional approach of manually coding solutions proved to be insufficient.

The emergence of deep learning has revolutionized the field by enabling machines to learn and make decisions without explicit programming. For instance, in weather forecasting, deep learning algorithms can utilize satellite image history to predict future weather conditions. Likewise, in healthcare, these algorithms can analyze various data sources to understand a person’s clinical situation. Additionally, deep learning plays a crucial role in developing self-driving cars that rely on real-time decision-making based on sensory inputs. Deep learning algorithms excel at automatically identifying patterns and extracting features from large datasets for making predictions or classifications. To achieve this, we construct a comprehensive structure for the deep learning model and expose it to thousands of data samples so that the model can effectively learn key features from the input data. These techniques are commonly referred to as data-driven approaches.

Deep learning algorithms, known as ANNs, draw inspiration from the structure of the brain. Within these networks, computational units are referred to as neurons and the connections between them are called synapses. Each synapse is characterized by a weight that represents its importance or strength.

2.1 Basics

DL algorithms are built upon the concept of ANNs, which are computational systems inspired by the structure and function of the human brain. With DL, we can solve different types of tasks depending on the input of the system and on the available information. The three primary machine learning paradigms are Supervised Learning (SL), Unsupervised Learning (UL), and RL.

Among all three paradigms, supervised learning is widely recognized as the most common and renowned paradigm. In SL, a dataset with complete knowledge and labels is available for analysis. For instance, if we have a picture of a dog and we are aware that it represents a dog. SL can be effectively utilized for classification and regression tasks. UL, on the other hand, involves datasets without associated labels or prior knowledge. UL is often employed in clustering techniques to group data based on their features as well as in dimensionality reduction approaches where data representations are condensed into fewer dimensions. RL stands out as another significant paradigm wherein an agent learns through interaction with its environment to maximize cumulative rewards. Unlike in other paradigms, here the collected data results from various interactions conducted by the agent within the environment; however, precise information about actions taken becomes less certain while suggestions about possible actions emerge instead.

DL relies on certain key components, regardless of the task or learning paradigm being used. These components include data, architectures (or models), objective functions, and optimization algorithms. Data plays a critical role in the success of DL algorithms. Each example or data point consists of a set of features (or covariance) that serve as inputs for predicting outcomes. Architectures or models in DL refer to the structure of neural networks designed for specific tasks. This structure facilitates the transformation from input data to output data, such as converting an image into a probability value indicating its classification within a particular class. Additionally, the objective function establishes the desired outcome that the model aims for during training. Typically, we define functions where a lower value indicates better performance; in this case, we refer to it as a loss function. Lastly, optimization functions are used to update the parameters of the DL model and minimize its objective function. Modern optimization functions utilize the gradient descent method wherein parameters are adjusted iteratively using gradients calculated from their respective objective functions with respect to model parameters.

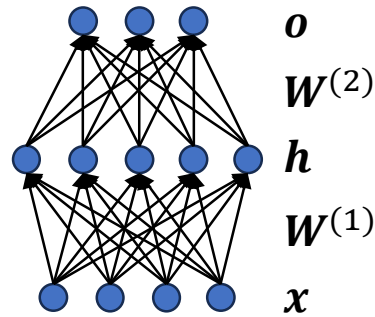


Figure 2.1: Two layers MLP with 4 input features, 5 hidden neurons, and 3 output neurons.

2.2 Architectures

Deep learning utilizes neural network architectures or models that encompass the arrangement of layers, number of neurons, and connections to perform various tasks such as image recognition and natural language processing. These architectures serve as blueprints for the behavior and functionality of the neural networks, enabling them to efficiently carry out specific machine learning tasks.

2.2.1 Multi Layer Perceptron

Multi-Layer Perceptrons (MLPs) are the most basic form of deep networks. They are comprised of multiple layers of neurons that are fully interconnected with the layer below, receiving input, and influencing the layer above. In Fig. 2.1, an example MLP is shown with four inputs, three outputs, and five hidden layers. This particular MLP has two computation layers - one in the hidden layer and one in the output layer. The forward step also referred to as inference, is executed by passing the input through each layer in a neural network. This process involves calculating the affine transformation of the input $\mathbf{x} \in \mathbb{R}^d$ (in this example is just one sample) using the parameters of the first layer $\mathbf{W}^{(1)} \in \mathbb{R}^{h \times d}$ and $\mathbf{b}^{(1)} \in \mathbb{R}^h$, denoted as $\mathbf{z} \in \mathbb{R}^h$:

$$\mathbf{z} = \mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)} \quad (2.1)$$

Next, an activation function ϕ is applied to \mathbf{z} to obtain output \mathbf{h} :

$$\mathbf{h} = \phi(\mathbf{z}) \quad (2.2)$$

Finally, $\mathbf{z} \in \mathbb{R}^q$, which represents the output of the neural network, is obtained by performing another affine transformation on $\mathbf{h} \in \mathbb{R}^h$ using parameters $\mathbf{W}^{(2)} \in \mathbb{R}^{q \times h}$ and

$\mathbf{b}^{(2)} \in \mathbb{R}^q$:

$$\mathbf{o} = \mathbf{W}^{(2)}\mathbf{h} + \mathbf{b}^{(2)} \quad (2.3)$$

The weights, denoted as \mathbf{W} , and biases, referred to as \mathbf{b} , are considered parameters in the network. These parameters undergo adjustments during the training process to minimize the disparity between the predicted output of the network and the actual output.

The affine transformation of an affine transformation is still an affine transformation:

$$\mathbf{o} = \mathbf{W}^{(2)}(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}) + \mathbf{b}^{(2)} = \mathbf{W}^{(2)}\mathbf{W}^{(1)}\mathbf{x} + \mathbf{W}^{(2)}\mathbf{b}^{(1)} + \mathbf{b}^{(2)} = \mathbf{W}\mathbf{x} + \mathbf{b} \quad (2.4)$$

However, relying solely on deep neural networks without incorporating non-linear activation functions renders them ineffective in capturing complex relationships. This limitation led to the introduction of non-linear activations, such as Eq. 2.2, which significantly enhance the capabilities of neural networks in modeling intricacies and complexities within data. Well-know activation functions are:

- ReLU [3]: $ReLU(x) = \max(x, 0)$
- sigmoid: $sigmoid(x) = \frac{1}{1+e^{-x}}$
- tanh [4]: $tanh(x) = \frac{1-e^{-2x}}{1+e^{-2x}}$

Neural networks with a single hidden layer have been recognized as universal approximators due to their ability to learn any function. However, it is important to note that this does not necessarily make them the optimal choice for all problem-solving scenarios. In certain cases involving infinite-dimensional spaces, kernel methods have shown to be more effective, as highlighted by the studies conducted in [5, 6]. Furthermore, deeper networks can often lead to more efficient approximations of functions compared to wider ones, as discussed in [7].

2.2.2 Convolutional Neural Networks

CNNs [8] are a specialized type of neural network that leverages the spatial relationships in input data to discern patterns and features. This characteristic proves especially beneficial when tackling tasks like image classification and object detection, where the arrangement of pixels holds significance within the two-dimensional tensor representation [9]. Furthermore, CNNs possess additional advantages: they exhibit a reduced parameter count compared to MLP, and their architecture lends itself well to

efficient parallelization on GPUs [10]. As a result, CNNs have gradually found application in various one-dimensional domains such as audio analysis [11], text processing [12], and time series analysis [13].

To ensure the algorithm’s ability to accurately classify images of dogs, even when they are slightly rotated, scaled, or shifted, CNNs employ a moving path (kernel) that traverses along the input axes in order to detect objects such as dogs. This spatial invariance enables CNNs to respond consistently and identify the object regardless of its position within the image. The principle behind this is called translation invariance.

Furthermore, CNN architectures emphasize different layers capturing specific information from an image based on their depths. The earlier layers focus on extracting details from local regions of the image using locality principles while deeper layers aim at identifying more complex and abstract features globally.

CNNs employ a mathematical convolution operation on the input data. In continuous one-dimensional space, this operation is represented as:

$$(f * g)(x) = \int f(z)g(x - z)dz \quad (2.5)$$

where the output value measures the overlap between two functions - one flipped and shifted by x . However, in discrete spaces like images with two-dimensional tensors, we need to modify the integral into a sum:

$$(f * g)(i) = \sum_a f(a)g(i - a) \quad (2.6)$$

Therefore, we apply convolution along both axes for images:

$$(f * g)(i, j) = \sum_a \sum_b f(a, b)g(i - a, j - b) \quad (2.7)$$

Images are not only characterized by their 2D tensor structure but also by their color. In order to capture the full complexity of images, we aim to have a three-dimensional representation. Instead of having just one hidden representation for each spatial position, our goal is to have a vector of hidden representations associated with each spatial position. These hidden representations can be visualized as multiple two-dimensional grids stacked together, often referred to as “channels” or “feature maps”. Each channel provides a spatially organized collection of extracted features for the subsequent layers. As we move closer to the input layer, certain channels may specialize in edge detection while others focus on recognizing textures. This hierarchical organization enables CNNs to learn and remember intricate details [9].

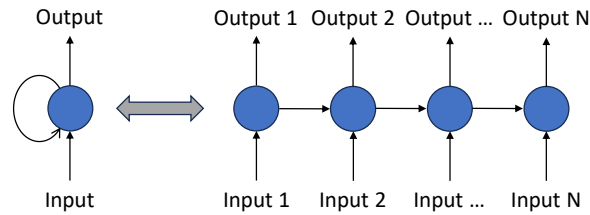


Figure 2.2: Example of RNN. On the left, the RNN is folded, while on the right, it is unfolded.

2.2.3 Recurrent Neural Networks

Recurrent Neural Networks (RNNs) are a class of deep learning models that aim to capture sequential dynamics using recurrent connections. These connections, which can be visualized as cycles within the network's nodes, enable RNNs to extend their predictions over multiple time steps or sequence steps. Unlike regular connections that transmit activations from one layer to the next simultaneously within the same time step, recurrent connections allow information to flow between adjacent time steps dynamically. This means that when examining the unfolded representation shown in Fig. 2.2, RNNs can be thought of as feedforward neural networks with shared parameters across these time steps for both conventional and recurrent layers.

RNNs have demonstrated their effectiveness in various tasks such as handwriting recognition [14], machine translation [15], and the recognition of medical diagnoses [16]. However, in recent times, RNNs have seen a significant shift in dominance, conceding ground to Transformer models [17]. Nevertheless, it is important to acknowledge that RNNs originally gained prominence as the preferred models for tackling intricate sequential structures in deep learning and continue to be fundamental models for sequential modeling today.

2.3 Backpropagation

BackPropagation (BP) is a method for calculating the gradient of a neural network's parameters. It addresses the spatial credit assignment problem, which involves determining which parameter should be adjusted to minimize the objective function given an output. Backpropagation applies the chain rule from the output to the input of each layer in order to calculate gradients for each layer's parameters with respect to the loss function. This process allows for efficient optimization and training of deep learning models.

The initial step in applying the backpropagation algorithm involves performing a forward pass. During this step, the inputs are fed into the neural network to compute

the objective function. To illustrate, we will consider a two-layer MLP without biases. Thus, the forward step can be described as follows:

$$\mathbf{z} = \mathbf{W}^{(1)}\mathbf{x} \quad (2.8)$$

where the inputs of the neural network $\mathbf{x} \in \mathbb{R}^d$ are multiplied by the first weight matrix $\mathbf{W}^{(1)} \in \mathbb{R}^{h \times d}$. The resulting vector, $\mathbf{z} \in \mathbb{R}^h$, is then passed through an activation function, denoted as ϕ , applied element-wise:

$$\mathbf{h} = \phi(\mathbf{z}) \quad (2.9)$$

$\mathbf{h} \in \mathbb{R}^h$ which is the output of the previous layer, will be multiplied by the weights of the second layer $\mathbf{W}^{(2)} \in \mathbb{R}^{q \times h}$ to compute a final output vector denoted as:

$$\mathbf{o} = \mathbf{W}^{(2)}\mathbf{h} \quad (2.10)$$

In order to assess how well our model performs on a single sample, we calculate the loss term L using a specific loss function l , which takes into account both our predicted outputs and corresponding labels:

$$L = l(\mathbf{o}; y) \quad (2.11)$$

The backpropagation algorithm enables us to efficiently compute gradients of the network's parameters with respect to L - that is, it computes how changes in each parameter affect L :

$$\frac{\partial L}{\partial \mathbf{W}^{(2)}} = \frac{\partial L}{\partial \mathbf{o}} \frac{\partial \mathbf{o}}{\partial \mathbf{W}^{(2)}} = \frac{\partial L}{\partial \mathbf{o}} \mathbf{h}^T \quad (2.12)$$

$$\frac{\partial L}{\partial \mathbf{h}} = \frac{\partial L}{\partial \mathbf{o}} \frac{\partial \mathbf{o}}{\partial \mathbf{h}} = \mathbf{W}^{(2)T} \frac{\partial L}{\partial \mathbf{o}} \quad (2.13)$$

$$\frac{\partial L}{\partial \mathbf{z}} = \frac{\partial L}{\partial \mathbf{h}} \frac{\partial \mathbf{h}}{\partial \mathbf{z}} = \frac{\partial L}{\partial \mathbf{h}} \odot \phi'(\mathbf{z}) \quad (2.14)$$

where \odot is an element-wise multiplication. Finally:

$$\frac{\partial L}{\partial \mathbf{W}^{(1)}} = \frac{\partial L}{\partial \mathbf{z}} \frac{\partial \mathbf{z}}{\partial \mathbf{x}} = \frac{\partial L}{\partial \mathbf{z}} \mathbf{x}^T \quad (2.15)$$

2.4 Optimizers

Optimizers play a significant role in training deep learning models. These algorithms are responsible for adjusting the model's parameters (weights and biases) during training to minimize the loss function and enhance the performance of the model on its given task.

In deep learning, optimization is typically carried out through an iterative process where the optimizer updates the parameters based on gradients obtained from calculating changes in loss with respect to these parameters. Two well-known optimizer algorithms that have gained popularity are the Stochastic Gradient Descend (SGD) and the Adam algorithm [9].

2.4.1 Stochastic Gradient Descend

The widely adopted optimization algorithm in deep learning is SGD. Particularly suited for handling large-scale datasets, it updates the model's parameters using mini-batches of training data [9]. At each learning step t , given the parameters θ_t , their gradient $\nabla f(\theta_t)$, and the learning rate η , the update rule for these parameters can be described as follows:

$$\theta_{t+1} = \theta_t - \eta \nabla f(\theta_t) \quad (2.16)$$

The authors in [18] propose a method to compute the update by introducing a momentum $\mu \in [0, 1]$:

$$v_{t+1} = \mu v_t - \eta \nabla f(\theta_t) \quad (2.17)$$

$$\theta_{t+1} = \theta_t + v_{t+1} \quad (2.18)$$

This additional term acts as a smoothing factor, similar to taking the mean, which helps alleviate high-frequency variations in gradients.

2.4.2 Adam

Adam, introduced [19], is a widely embraced learning algorithm in the field of deep learning. It combines features from SGD with momentum [18], Adagrad [20] optimizer, and RMSProp [21] optimizer to create a powerful optimization algorithm. Despite its popularity for its robustness and effectiveness, Adam faces challenges. For instance, research conducted in [22] revealed instances where Adam could exhibit divergence due to inadequate variance control. To address these issues, the authors of [23] proposed Yogi as a solution for improving upon the limitations associated with Adam's performance.

2.5 Learning Paradigms

The term “learning paradigm” refers to a core methodology or framework through which a machine learning system assimilates knowledge and generates predictions. These paradigms establish the manner in which algorithms acquire insights from data and create generalizations for accurate predictions and informed decisions.

2.5.1 Supervised Learning

Supervised learning refers to a category of tasks where there is a dataset consisting of both input features and corresponding labels. The goal is to build a model that can predict the labels based on the input features. Each example/sample in this context consists of paired features and labels. The term “supervision” comes into play as experts or supervisors provide the model with labelled examples to help determine optimal parameters. From a probabilistic perspective, we aim to estimate the conditional probability of a label given input features.

Supervised learning is a fundamental paradigm in machine learning that serves as the basis for many successful applications across various industries. This approach enables us to estimate the probability of an unknown outcome based on specific available data, making it applicable to tasks such as predicting cancer or non-cancer from an input image, translating between languages, and forecasting stock values using historical data.

The learning process typically begins with the collection of a large dataset consisting of examples, for which we have knowledge of their features. From this dataset, a random subset is selected to obtain the ground truth labels for each example. This selected subset, along with their corresponding labels, forms the training set. Next, this training dataset is fed into a supervised learning algorithm, an algorithm that takes in a dataset and outputs another function, known as the learned model. The learned model captures patterns and relationships between inputs and outputs based on the provided data. Finally, with the help of the learned model, predictions can be made for new inputs that were not included in the original training set, usually part of the validation or test set. By using the outputs generated by the model as predictions for these inputs’ respective labels or outcomes.

2.5.1.1 Regression

A regression task is characterized by the nature of the target variable. For example, consider a scenario where you are in the market for a new home and want to estimate

its fair market value based on various attributes. In this case, the dataset would contain historical home listings and their corresponding sales prices as labels. When the labels can have arbitrary numerical values, it is referred to as a “regression” problem. The main objective is to build a model that accurately predicts label values. Thus, regression problems aim to answer questions such as “how many?” or “how much?”.

2.5.1.2 Classification

In classification tasks, the primary objective is for our model to analyze features, such as pixel values in an image, and then make predictions concerning which category or class the example belongs to from a discrete set of options. For instance, when recognizing handwritten digits, there are typically ten classes corresponding to the digits 0 through 9. Binary classification occurs when there are only two classes involved; for example, our dataset could consist of images of animals with labels assigned as either “cat” or “dog”.

In contrast to regression, where the objective is to predict a numerical value, classification involves building a classifier that assigns predicted class labels. For instance, in an animal classification scenario with classes {cat, dog}, a classifier may analyze an image and generate a probability indicating whether it depicts a cat or not. This probability can be interpreted as the confidence level of the classifier in its prediction (e.g., 0.9 means 90% certainty). The magnitude of this probability reflects the model’s uncertainty regarding the assigned class label.

When faced with a situation involving more than two potential classes, the problem is referred to as “multiclass classification”. This type of problem often arises in tasks such as recognizing handwritten characters, where the classes can range from 0 to 9 and include alphabets. In contrast to regression problems that aim to minimize squared error loss, classification problems typically employ a different loss function known as “cross-entropy”. The cross-entropy loss measures the difference between predicted class probabilities and true class labels, making it ideal for training classifiers to accurately assign classes.

2.5.2 Reinforcement Learning

Reinforcement learning is centred on the concept of optimizing a numerical reward signal by learning how to take action. Unlike other forms of learning, where specific actions are provided, reinforcement learning requires the learner to explore and discover which actions yield the highest rewards through trial and error. This paradigm becomes

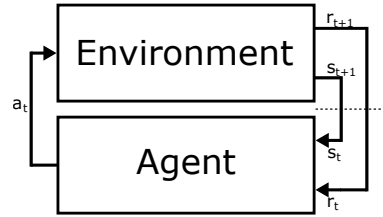


Figure 2.3: The reinforcement training loop involves the agent taking an action a_t in response to which the environment transitions and provides information about the new state s_t as well as the reward received, denoted by r_t .

more complex because not only these actions do impact immediate rewards but they can also influence future situations and subsequent rewards. Trial-and-error exploration and delayed reward outcomes are two defining features that distinguish reinforcement learning from other approaches in machine learning.

To formally define the problem of reinforcement learning, we rely on principles from dynamic systems theory. Specifically, we frame it as the optimal control of incompletely known Markov Decision Process (MDP). The core concept here is to capture the fundamental aspects of the real challenge that a learning agent faces while interacting over time with its environment in order to achieve a specific goal [24].

A learning agent must possess the capability to perceive and understand the state of its environment to some degree. It should also be able to take actions that have an impact on the state. Additionally, having one or more goals relating to the state of the environment is crucial for effective decision-making [24].

MDPs encapsulate these three essential aspects (sensation, action, and goal) in their simplest forms without oversimplifying any of them. Methods well-suited to tackle such problems are considered reinforcement learning methods [24].

Reinforcement learning differs from supervised learning in several key ways. Firstly, RL agents do not have access to a labelled dataset or an expert for guidance, which means they lack an oracle. Secondly, the feedback received by RL agents is often sparse, meaning that the neural network does not receive a reward at every timestep. Finally, instead of relying on pre-existing labelled data, RL agents generate their own data through interaction with the environment during training [25].

2.5.2.1 Basics

Reinforcement learning problems are commonly represented as a system consisting of an agent and an environment. The generic RL loop, depicted in Fig. 2.3, illustrates the iterative nature of RL. Agents, such as drones, cars, robots, humans, or neural

networks interact with their environments and produce actions that induce changes within them [25]. Simulators play a crucial role in RL by emulating agents and their respective environments. As presented in Figure Fig. 2.3, the agent selects actions (a_t) based on its policy function (π), which determines its decision-making process. These chosen actions modify the environment through specified transition functions. During this interactive process, feedback is received from the environment to evaluate the quality of selected actions via rewards (r_t) and updates regarding the new state reached by the agent (s_t). Hence, a reinforcement learning system operates as a feedback control loop where an agent interacts with an environment in order to maximize its objective. This interaction is represented by signals denoted as (s_t, a_t, r_t) . These signals combine to form what is referred to as an “experience” within this framework. This control loop can continue indefinitely or come to an end either when a terminal state is reached or after reaching the maximum specified time step, denoted as $t = T$. The duration from $t = 0$ until the termination of the environment is referred to as an “episode.” During each episode, a trajectory is formed, consisting of a sequence of experiences represented by $\tau = (s_0, a_0, r_0), (s_1, a_1, r_1), \dots$. For an agent to effectively learn a good policy in this framework, it typically requires numerous episodes ranging from several hundred to millions depending on the complexity of the problem being addressed [25].

The environment evolves based on the action taken, as determined by a transition function formulated using a MDP. This mathematical framework is used to model sequential decision-making. To better understand why transition functions are represented as MDPs, let’s examine a general formulation. The state at time $t + 1$ is dependent on the previous states and actions:

$$s_{t+1} \sim P(s_{t+1} | (s_0, a_0), (s_1, a_1), \dots, (s_t, a_t)) \quad (2.19)$$

where at time step t , the next state s_{t+1} follows a probability distribution P conditioned on the entire history. This implies that the likelihood of transitioning from state s_t to s_{t+1} is influenced by all preceding states and actions throughout the episode. Designing a transition function in this manner can be demanding, particularly for lengthy episodes. Such a transition function would need to account for numerous combinations of past effects occurring at any given point. Furthermore, this formulation adds complexity to an agent’s policy, and its action-generating function, since it must consider the complete history of states and actions when determining how to act [25].

To ensure the practicality of the transition function in the environment, we incorporate it into a MDP by assuming that the transition to the next state s_{t+1} is solely dependent on the previous state s_t and action a_t . This assumption, referred to as the

Markov property, simplifies our new transition function:

$$s_{t+1} \sim P(s_{t+1}|s_t, a_t) \tag{2.20}$$

where the next state, denoted as s_{t+1} , is probabilistically determined by a distribution P that depends on the current state s_t and action a_t . This simplified transition function adheres to the Markov property, which means that the current state and action at time step t provide sufficient information to completely determine the probability of transitioning to the next state at time step $t + 1$. Despite its simplicity, this formulation remains powerful and versatile. Many processes, including games, robotic control, and planning, can be expressed using this form [25].

To this point, our focus has been on simplifying the concept of states. However, it is important to acknowledge that there are two types of states within the context of reinforcement learning:

- **Observed State (s_t):** This state is generated by the environment and directly observed by the agent. It serves as the foundational information for decision-making and action-taking.
- **Internal State of Environment (s_t^{int}):** This represents the internal state specific to an environment, which may contain additional information or details about its current conditions. The environment utilizes this internal state during transitions to subsequent states.

In a traditional MDP, it is assumed that the observed state equals the internal state of the environment ($s_t = s_t^{int}$). This assumption allows for simplified learning since the agent's information about the current state aligns with what the environment utilizes to transition to the next state. However, it is important to note that the equivalence does not always hold true. In certain cases, there may be a discrepancy between these two states. This concept gives rise to what is known as a Partially Observable Markov Decision Process (POMDP). In a POMDP, the state s_t that is visible to the agent only provides partial information about the actual state of the environment ($s_t \neq s_t^{int}$). This introduces an added layer of complexity as agents now have to make decisions based on incomplete or partially observable information. Despite being more challenging, this representation better reflects many real-world scenarios and adds realism to decision-making processes in various domains [25].

Deep reinforcement learning algorithms can be categorized as on-policy or off-policy, and this distinction affects how training iterations use data. An algorithm is classified as on-policy if it learns directly from the policy being used. This means that during

training iterations with different versions of the policy (e.g., $\pi_1, \pi_2, \pi_3, \dots$), only the current policy is used to generate training data. As a result, any data generated by previous policies becomes irrelevant and must be discarded. On-policy methods are conceptually simple but tend to require a large amount of training data for meaningful improvements due to their sample efficiency [26]. In contrast, an algorithm is considered off-policy if it does not require the exclusive use of data generated by the current policy. Instead, it can utilize data collected from various policies during training. As a result, off-policy methods are generally more sample-efficient as they can leverage a wider range of collected data over time. However, implementing this approach may require significantly more memory to store and reuse the accumulated data throughout training [26].

2.5.2.2 Learnable Functions

To formalize a reinforcement learning problem, it is essential to define the objective that the agent seeks to maximize. We can start by defining the return $R(\tau)$, using a trajectory from an episode, $\tau = (s_0, a_0, r_0), \dots, (s_T, a_T, r_T)$:

$$R(\tau) = r_0 + \gamma r_1 + \gamma^2 r_2 + \dots + \gamma^T r_T = \sum_{t=0}^T \gamma^t r_t \quad (2.21)$$

where the return is the discounted sum of rewards in a trajectory. The discount factor γ falls within and represents how much weight is given to future rewards compared to immediate ones [25].

The objective $J(\tau)$ is expressed as the expectation of the returns over multiple trajectories:

$$J(\tau) = \mathbb{E}_{\tau \sim \pi}[R(\tau)] = \mathbb{E}_{\tau \sim \pi} \left[\sum_{t=0}^T \gamma^t r_t \right] \quad (2.22)$$

The return $R(\tau)$ includes the sum of discounted rewards $\gamma^t r_t$ over all time steps $t = 0, \dots, T$. Conversely, $J(\tau)$ represents the average return across multiple episodes. This expectation takes into account the inherent stochasticity in actions and the environment, recognizing that the return may vary in repeated runs [25]. Maximizing the objective function $J(\tau)$ is equivalent to maximizing the return. The choice of the discount factor γ , which belongs in the range, plays a crucial role in determining how future rewards are valued:

- A smaller γ makes the agent more “shortsighted”, giving less importance to rewards in distant time steps. If $\gamma = 0$, only the initial reward r_0 is considered:

$$R(\tau)_{\gamma=0} = \sum_{t=0}^T \gamma^t r_t = r_0 \quad (2.23)$$

- A larger γ makes the agent more “farsighted”, assigning greater significance to rewards in distant future time steps. When $\gamma = 1$, all rewards from every time step receive equal weight:

$$R(\tau)_{\gamma=1} = \sum_{t=0}^T \gamma^t r_t = \sum_{t=0}^T r_t \quad (2.24)$$

For problems with an infinite time horizon, it is necessary to set $\gamma < 1$ to ensure that there exists an upper bound for the objective function. In finite-time horizon problems, selecting an appropriate value for γ becomes critical as it impacts problem-solving complexity. Different values of γ lead to different trade-offs between immediate and future rewards, shaping both agent behaviour and the learning process accordingly [25].

In reinforcement learning, there are three primary functions that an agent could learn: the policy, the value function and the environment model [25].

The policy (π) plays a fundamental role in this control loop as it generates the actions needed to navigate and interact with the environment. The policy is a function that maps states to actions. An action sampled from a policy is denoted as $a \sim \pi(s)$. The policy can be stochastic, meaning it may probabilistically output different actions for the same state, which can be represented as $\pi(a|s)$ to denote the probability of an action a given a state s . It defines how an agent chooses actions in the environment to maximize its objective [25].

The value functions provide valuable insights into the objective and aid the agent in evaluating the desirability of states and available actions based on their expected future returns. There are two distinct types of value functions:

$$V^\pi(s) = \mathbb{E}_{s_0=s, \tau \sim \pi} \left[\sum_{t=0}^T \gamma^t r_t \right] \quad (2.25)$$

$$Q^\pi(s, a) = \mathbb{E}_{s_0=s, a_0=a, \tau \sim \pi} \left[\sum_{t=0}^T \gamma^t r_t \right] \quad (2.26)$$

The V-function (Eq. 2.25) assesses the quality of a state by considering its expected future returns when adhering to a given policy π from the current state s . On the other hand, the Q-function evaluates the anticipated future returns associated with taking a specific action a within state s under policy π [25].

Finally, the transition function, denoted as $P(s_0|s, a)$, plays a crucial role in providing information about the environment. When an agent acquires knowledge of this function through learning, it gains the ability to anticipate and predict the subsequent state s_1 that will occur after taking action a in state s . By utilizing this acquired knowledge of the transition function, the agent can simulate or “envision” how its actions will influence and shape future states without requiring direct interaction with the environment. This predictive capability empowers the agent to effectively plan ahead and make well-informed decisions based on its anticipation of how the environment is likely to respond to different choices it makes [25].

Depending on the learnable function, the RL algorithm can be value-based, policy-based, model-based, or a combination of them.

The V-function and the Q-function are closely related, with the former being an expectation of the latter over Q-values for all possible actions in a given state under a specific policy:

$$V^\pi(s) = \mathbb{E}_{a \sim \pi(s)}[Q^\pi(s, a)] \quad (2.27)$$

However, determining which function is better suited for learning poses an important question [25].

The V-function has a significant drawback as it does not take into account the value of individual actions. This means that knowing the value of a state alone does not provide information on which action to choose in order to transition to a more favorable state. To overcome this limitation, one possible approach is to compute all potential future V-values for each possible action, referred to as $V^\pi(s')$ ¹. However, this method can be computationally intensive and may not be feasible in situations where resetting environmental states back to previous steps is impractical. For example, consider calculating the V-value during a complex chess game; such calculations would require substantial computational resources and time constraints [25].

A potential solution to this challenge can be found in the Q-function. Unlike the V-function, the Q-function offers agents a more direct mechanism for decision-making. By calculating the Q-value for each available action a in a given state s , agents are able to select the action with the highest corresponding Q-value. In an optimal scenario,

¹To make the notation more compact we substitute $t + 1$ with $'$, e.g. s_{t+1} will be s' .

$Q^*(s, a)$ represents the optimal expected value obtained from taking action a in state s . This signifies that by understanding and utilizing this optimal Q-value, agents can pursue actions that result in maximum rewards and ultimately guide them toward an optimal policy [25].

It is important to note that learning the Q-function in deep reinforcement learning is a more computationally complex and data-intensive task compared to learning the V-function. In order to obtain a reliable estimate for $V^\pi(s)$, it is necessary for the training data to adequately cover the state space. However, in order to achieve an accurate estimate for $Q^\pi(s, a)$, it is required that the training data covers all possible state-action pairs (s, a) . Since the combined state-action space can be significantly larger than just the state space itself, gathering enough data becomes challenging when trying to learn a robust estimation of Q-function [24]. As a result, RL algorithms often approximate $Q^\pi(s, a)$ instead of obtaining its exact value in practice by using learned value functions for action selection purposes.

2.5.2.3 Double Deep Q-Network

The process of learning a neural network to estimate the policy, known as a value network, follows a specific workflow. Initially, trajectories τ_i are generated and for each state-action pair (s, a) , we make predictions on \hat{Q} -values as our estimations. These predicted \hat{Q} -values serve as our initial beliefs. We then utilize these trajectories to create target Q_{tar} -values. The primary goal is to minimize the discrepancy between our predictions and these target values using regression loss methods such as Mean Squared Error (MSE). This iterative process is repeated multiple times, resembling supervised learning workflows where every prediction corresponds with predetermined targets. However, in reinforcement learning scenarios like this one, it is necessary that we devise means of generating these target values for each trajectory since they are not given beforehand [25].

Given N trajectories $\tau_i, i \in 1, \dots, N$ starting in state s with the agent taking action a , we can calculate an estimate of $Q_{tar}^\pi(s, a)$ using Monte Carlo (MC) sampling. This estimate is obtained by averaging the returns from all trajectories:

$$Q_{tar:MC}^\pi(s, a) = \frac{1}{N} \sum_{i=1}^N R(\tau_i) \quad (2.28)$$

If we have access to an entire trajectory τ_i , it becomes possible to compute the actual Q-value achieved by the agent for each (s, a) pair within that trajectory. This

calculation aligns with the definition of the Q -function presented in Eq. 2.26 where future cumulative discounted rewards are expected values starting from a single instance and equal to cumulative discounted reward from the current time step till the end of the episode. Essentially, Eq. 2.28 shows the MC estimate for $Q^\pi(s, a)$ can be performed even with $N = 1$, i.e., using just one trajectory [25].

The consequence of this process is that each (s, a) pair in the dataset is assigned a target Q_{tar} -value, effectively labelling the dataset. However, one drawback of using MC sampling is the need to wait for episodes to complete before any data from those episodes can be utilized for learning. Eq. 2.28 emphasizes this requirement as it relies on having access to rewards throughout the trajectory starting from (s, a) . Given that episodes can have multiple time steps T , this leads to delays in the training process. Consequently, an alternative strategy called Temporal Difference (TD) learning has been motivated as an option for learning Q-values [25].

TD learning introduces a fundamental insight: Q -values for the current time step can be defined recursively in terms of Q -values for the next time step:

$$Q^\pi(s, a) = \mathbb{E}_{s' \sim p(s'|s, a), r \sim \mathcal{R}(s, a, s')} [r + \gamma \mathbb{E}_{a' \sim \pi(s')} [Q^\pi(s', a')]] \quad (2.29)$$

where $\mathcal{R}(s, a, s')$ is the reward function of the environment. Eq. 2.29, referred to as the Bellman equation, holds exactly when the Q -function aligns with the policy π . It not only describes the recursive nature of Q -values but also provides a methodology for learning them. Similar to MC sampling, TD learning involves deriving target values, $Q_{tar}^\pi(s, a)$, for each (s, a) pair using the temporal difference concept. Assuming we have a neural network, denoted as Q_θ , to represent the Q -function, we estimate the right-hand of Eq. 2.29 using Q_θ . In each training iteration, we update our Q -value predictions $\hat{Q}^\pi(s_t, a_t)$ to make them closer to the target values $Q_{tar}^\pi(s_t, a_t)$. This approach is effective because the action-value function $Q_{tar}^\pi(s_t, a_t)$ takes into account information from one time step ahead compared to $\hat{Q}^\pi(s_t, a_t)$, which only considers the immediate reward r from the next state s' . As a result, $Q_{tar}^\pi(s_t, a_t)$ provides slightly more insight into how the trajectory will unfold in the long run. This methodology operates under the assumption that knowledge about maximizing cumulative rewards gradually becomes apparent as the trajectory progresses and is not initially available at the beginning of an episode - an inherent characteristic of reinforcement learning problems [25].

However, building $Q_{tar}^\pi(s_t, a_t)$ using Eq. 2.29 presents two challenges related to the presence of two expectations within the equation. Firstly, the outer expectation $\mathbb{E}_{s' \sim p(s'|s, a), r \sim \mathcal{R}(s, a, s')} [\dots]$ deals with the next state and reward. When considering a collection of trajectories $\tau_1, \tau_2, \dots, \tau_N$, where each τ_i consists of set of tuples (s, a, r, s') ,

it is important to note that only one instance of the next state s' is available for each (s, a, r, s') tuple due to selecting an action. In deterministic environments, focusing solely on the actual observed next state suffices; however, stochastic environments bring forth challenges as taking action a in state s can lead to multiple potential future states while only one is observed at each step. This issue can be resolved by focusing on the example that corresponds to the observed next state, even though it may introduce variance in the Q -value estimate, especially in stochastic environments. To simplify the estimation process, a single example can be used to estimate the distribution over the next states s' and rewards r [25]. This leads to a reexpression of the Bellman equation:

$$Q^\pi(s, a) = r + \gamma \mathbb{E}_{a' \sim \pi(s')} [Q^\pi(s', a')] \quad (2.30)$$

Another challenge is related to the inner expectation $\mathbb{E}_{a' \sim \pi(s')} [\dots]$ in Eq. 2.29 which involves actions in the next state s' . Since Q -value estimates for all possible actions in the next state s' are available through current Q -function estimates, determining the required probability distribution over actions for this expectation becomes challenging. To address this issue, two approaches are commonly used: SARSA [27] and DQN [28] algorithms.

In the case of DQN, the solution is to choose the action with the maximum Q -value:

$$Q^\pi(s, a) \approx r + \gamma \max_{a'_i} Q^\pi(s', a'_i) = Q_{tar}^\pi(s, a) \quad (2.31)$$

This approach assumes an implicit policy where the action selected with certainty (probability 1) is the one with the highest Q -value. The equation remains valid because in Q -learning, being greedy towards Q -values constitutes an implicit policy. This policy is defined as optimal policy [24] and, thanks to it, the DQN is considered an off-policy algorithm since the data used in training can be collected from different policies [25].

The DQN algorithm can be improved through two key adjustments. The first improvement involves the use of a target network for computing $Q_{tar}^\pi(s, a)$. In the original DQN algorithm, $Q_{tar}^\pi(s, a)$ continuously changes as it depends on $\hat{Q}^\pi(s, a)$. During training, the parameters θ of the Q -network are adjusted to minimize the difference between predicted and target values. However, this can be challenging when there are frequent changes in $Q_{tar}^\pi(s, a)$ during training steps [29].

To address this issue, a target network with parameters ϕ is introduced. The target network acts as a secondary neural network that represents a delayed copy of the primary Q -network Q^{π_θ} used for calculating predictions. By employing this separate target

network $Q^{\pi\phi}$, we can calculate more stable values for $Q_{tar}^{\pi}(s, a)$ using an updated Bellman equation:

$$Q_{tar}^{\pi\phi}(s, a) = r + \gamma \max_{a'} Q^{\pi\phi}(s', a') \quad (2.32)$$

Periodically, the parameters ϕ are updated to match the current values of θ . This update process is referred to as a replacement update.

The second improvement in the DQN framework is called Double DQN (D2QN) [30, 31]. This modification aims to address the issue of overestimating Q -values. In $Q^{\pi}(s', a')$, there are several factors that can lead to inaccuracies in the Q -values: imperfections in neural network function approximation, limited exploration by agents, and environmental noise affecting observations. As a result, it is expected that some errors will exist in $Q^{\pi}(s', a')$, leading to an overestimation of Q -values. Moreover, when there are more actions available in a given state s' , there is an increased likelihood of overestimation occurring. The original DQN has been observed to overestimate $Q^{\pi}(s, a)$ for frequently visited pairs (s, a) , which could be problematic if agent exploration is not evenly distributed across all possible actions. In these cases, the distorted ranking caused by overestimated Q -values may result in suboptimal action selections.

The Double DQN algorithm utilizes two separate Q -function estimates based on different sets of experiences. To select the action with the highest Q -values, one estimate is used while another estimate is used to calculate the Q -value for determining the target value. Using a second trained Q -function helps reduce positive bias and improves accuracy in estimating Q -values. Hence, implementing D2QN requires two distinct neural networks: ϕ and θ . The network θ selects actions, while ϕ computes the Q -value for (s', a') using this equation:

$$Q_{tar:D2QN}^{\pi}(s, a) = r + \gamma Q^{\pi\phi}(s', \operatorname{argmax}_{a'} Q^{\pi\theta}(s', a')) \quad (2.33)$$

By employing this strategy, overestimation issues are minimized, leading to more accurate action selection within the framework of DQN algorithm.

2.5.2.4 Proximal Policy Optimization

Before delving into Proximal Policy Optimization (PPO) [32], we need to introduce a few more concepts: i) the advantage function and ii) the performance collapse.

The advantage function $A^{\pi}(s_t, a_t)$ is a learnable function that measures the relative superiority or inferiority of a specific action compared to the average action chosen by

the policy in a given state. It is defined as:

$$A^\pi(s_t, a_t) = Q^\pi(s_t, a_t) - V^\pi(s_t) \quad (2.34)$$

This concept offers several valuable properties. Firstly, when considering the expected value of $A^\pi(s_t, a_t)$, it equals zero. This implies that if all available actions are approximately equally advantageous in a given state, then the advantage function will be close to zero for all actions. As a result, during training using A^π as guidance for updating policy probabilities through reinforcement signals will maintain relatively stable probabilities for these actions over time. In contrast, other methods rely on absolute state or state-action values which may not be zero under similar circumstances and thus lead to more significant adjustments in action probabilities [25].

Another challenging situation arises when the selected action performs worse than the average action, even though the expected return is still positive. This means that $Q^\pi(s_t, a_t) > V^\pi(s_t)$ but $A^\pi(s_t, a_t) < 0$. Ideally, in such cases, we would want to reduce the likelihood of selecting this action since there are better alternatives available. The A^π method aligns more closely with this intuitive behaviour as it discourages choosing that particular action. Conversely, using $Q^\pi(s_t, a_t)$ may inadvertently encourage the selection of the suboptimal action, which contradicts our expectations [25].

The advantage function also provides a comparative perspective. It evaluates the state-action pair $Q^\pi(s_t, a_t)$, determining whether taking action will result in a better or worse outcome compared to the value of the state $V^\pi(s_t)$. Importantly, it avoids penalizing an action solely based on the current unfavourable state of the policy. Likewise, it does not give excessive credit to an action for being in a favourable state. This approach is beneficial because an action can only influence future trajectories and not how the policy arrived at its current state. Therefore, evaluating actions based on their impact on future outcomes is more rational [33].

In the training process of reinforcement learning, there are situations wherein the agent's performance declines and it becomes necessary to restart the training from scratch. This occurrence is a result of how the optimization process develops.

In the field of reinforcement learning, a sequence of policies denoted as $\pi_1, \pi_2, \dots, \pi_n$ is explored within the policy space Π . Each policy can be expressed as π^θ where θ represents the parameters existing in parameter space ϕ . The aim is to find the optimal policy by searching for an appropriate set of parameters represented by $\phi \in \Phi$. To regulate the magnitude of parameter updates during this search process, a learning rate α is used:

$$\Delta\theta = \alpha \nabla_\theta J(\pi^\theta) \quad (2.35)$$

The challenge arises because the mapping between the policy space and parameter space is not always direct, and the distances in both spaces may not align. Let's consider two pairs of parameters, (θ_1, θ_2) and (θ_2, θ_3) , where they have equal distances in the parameter space: $d_\theta(\theta_1, \theta_2) = d_\theta(\theta_2, \theta_3)$. However, their corresponding policies $(\pi_{\theta_1}, \pi_{\theta_2})$ and $(\pi_{\theta_2}, \pi_{\theta_3})$ do not necessarily have the same distance:

$$d_\theta(\theta_1, \theta_2) = d_\theta(\theta_2, \theta_3) \not\Rightarrow d_\pi(\pi_{\theta_1}, \pi_{\theta_2}) = d_\pi(\pi_{\theta_2}, \pi_{\theta_3}) \quad (2.36)$$

This misalignment creates a challenge in selecting the appropriate step size α for updating parameters. Determining how small or large of a step in the parameter space corresponds to changes in the policy space is not straightforward. If α is too small, training will be time-consuming and the policy may become trapped in suboptimal solutions. On the other hand, if α is too large, the movement in the policy space might exceed regions that contain good policies, causing a decline in performance. This leads to an updated policy that performs significantly worse and generates inadequate trajectory data for subsequent updates, further degrading the overall policy quality [25].

To overcome this challenge, an optimal algorithm should dynamically modify the step size of the parameters based on the current position of the policy in both policy and parameter spaces. This adjustment ensures that the new policy remains within a certain range, known as the trust region, around the old policy. The boundaries of this neighbourhood are defined as trust region constraints to prevent significant degradation in performance [25].

PPO comprises a set of algorithms designed to address the trust region constraints problem. There are two variations within this algorithm family: i) PPO with adaptive KL penalty, and ii) PPO with clipped surrogate objective [25].

The first variation, PPO with adaptive KL penalty, introduces a KL penalty and utilizes the following KL surrogate objective:

$$J^{KL PEN}(\theta) = \max_{\theta} \mathbb{E}_t[r_t(\theta)A_t - \beta KL(\pi_\theta(a_t|s_t) || \pi_{\theta_{old}}(a_t|s_t))] \quad (2.37)$$

Here, β serves as an adaptive coefficient that governs the extent of the imposed KL penalty; higher values permit greater differences between π and π_{old} . The term $r(\theta)A_t$ denotes the surrogate gradient which is defined as:

$$J^{CLI}(\theta) = \mathbb{E}_t \left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}} A_t^{\theta_{old}} \right] = \mathbb{E}_t[r_t(\theta)A_t] \quad (2.38)$$

The second variation in the PPO family is referred to as PPO with clipped surrogate objective:

$$J^{CLIP}(\theta) = \mathbb{E}_t[\min_{\theta} (r_t(\theta)A_t, \text{clip}(r_t(\theta), 1 - \epsilon, 1 + \epsilon)A_t)] \quad (2.39)$$

In this case, ϵ establishes constraints for policy updates.

Chapter 3

Spiking Neural Networks

SNNs are often seen as the third iteration of ANNs [2]. This classification is primarily due to their resemblance to the functioning of mammalian brains, distinguishing them from earlier generations of ANNs.

The key distinction between traditional neural networks, which belong to the second generation of ANNs, and SNNs lies in their fundamental computational unit - the neuron. In conventional ANNs, neurons are defined by activation functions that determine their output based on input values. In contrast, SNNs utilize spiking neurons characterized by a set of ODEs. An ODE system is utilized to represent the spiking neuron as a dynamic system, where a hidden state called the membrane potential accumulates over time. Once this membrane potential surpasses a predefined threshold on the upward trend, the neuron emits an event referred to as a spike. These spikes function as the primary means for neurons to communicate with one another, and a sequence of spikes is known as a spike train. The existence of spikes in SNNs allows for asynchronous communication among neurons, which diverges from the synchronous nature observed in standard ANNs.

SNNs possess several advantageous characteristics due to their complex structure. They have the capability to replicate key characteristics found within mammalian brains such as low latency, rapid inference speed, and energy efficiency. These qualities make SNNs highly promising for various applications, specifically those that require brain-inspired computing capabilities.

3.1 Neuron Models

Neuroscience presents various models of neurons, each with its own level of complexity and biological plausibility. The Hodgkin-Huxley model [34] stands out for accurately describing neural behaviour in terms of ion channels and currents. However, due to its complexity, alternative neuron models have been proposed as more practical options. These include the Integrate-and-Fire model, the Leaky Integrate-and-Fire model, and the Adaptive Leaky Integrate-and-Fire model [35]. By choosing these models, researchers strike a balance between preserving biological accuracy and managing computational demands.

3.1.1 Integrate-and-Fire Model

The Integrate-and-Fire (IF) neuron model, initially introduced [36], was one of the earliest computational representations of neurons. This concept emerged during a period when researchers lacked direct measurement techniques for observing the electrical and chemical processes occurring within live neurons. Instead, it offers a simplified depiction: the neuron’s membrane can be likened to a capacitor that gradually accumulates and stores electrical charge over time [37].

The name “Integrate-and-Fire” highlights two key behaviours exhibited by this model: 1. **Current Integration:** The neuron progressively integrates incoming electrical current across time, similar to how a capacitor accrues charge. 2. **Firing:** When the voltage across the neuron’s membrane surpasses a specific threshold value, it generates an action potential denoted also as fire or spike. In this particular model, the voltage of the neuron’s membrane changes over time as it integrates incoming currents. This integration process is determined by the following equation:

$$C \frac{dv(t)}{dt} = I(t) \quad (3.1)$$

where C is the membrane capacitance, $v(t)$ is the membrane voltage at a given time, and $I(t)$ is the input current of the neuron. Essentially, the neuron accumulates and combines incoming currents over time.

To determine when a spike occurs in the neuron, a threshold voltage called v_{th} is introduced. When the membrane voltage exceeds this threshold, triggering mechanisms cause an action potential or spike to be generated. This behaviour aligns with neurophysiological principles where reaching a certain threshold initiates firing activities. Following each spike event, there is a reset procedure implemented. The membranes’ voltages are restored to their resting state v_{rest} . This resetting mechanism agrees with

observed physiological phenomena where various ionic currents involving potassium ions actively contribute to gradually bringing back neurons' potential difference towards their resting states.

IF models often include an absolute refractory period (t_{ref}) to account for the physiological process that prevents neurons from firing immediately after a spike. During this refractory period, the neuron's voltage remains at the resting potential v_{rest} :

3.1.2 Leaky Integrate-and-Fire Model

The Leaky Integrate-and-Fire (LIF) model [36, 38, 39] is a significant advancement in understanding the physiological aspects of neuron membranes. Unlike ideal capacitors, actual neuron membranes exhibit slow leakage of current over time. This gradual leakage allows the membrane voltage to gradually return to its resting potential, incorporating a “forgetting” mechanism into the LIF model.

The dynamics of the LIF model can be described using the following equation:

$$C \frac{dv(t)}{dt} = -(v(t) - v_{rest}) + I(t) \quad (3.2)$$

where C represents membrane capacitance while R is the membrane resistance. The term $v(t)$ refers to membrane voltage and v_{rest} denotes resting potential. Additionally, $I(t)$ signifies the input current received by the neuron. The inclusion of the resistance term R accounts for current leakage from the cell's membrane. When no external inputs are present, the membrane voltage asymptotically returns back to its resting potential:

$$\tau \frac{dv(t)}{dt} = -v(t) + I(t) \quad (3.3)$$

which can be rewritten as:

$$v(t+1) = v(t)e^{-\frac{dt}{\tau}} + I(t) \quad (3.4)$$

Similar to the IF model, the LIF model also features a resetting procedure following a spike. After firing, the membrane voltage is reset to the resting potential, maintaining consistency with the IF model's physiological principles. It can be hard reset:

$$v(t+1) = \begin{cases} v_{rest} & \text{if } z(t) = 1 \\ v(t+1) & \text{if } z(t) = 0 \end{cases} \quad (3.5)$$

or soft reset:

$$v(t+1) = \begin{cases} v(t) - v_{thr} & \text{if } z(t) = 1 \\ v(t+1) & \text{if } z(t) = 0 \end{cases} \quad (3.6)$$

3.1.3 Adaptive Leaky Integrate-and-Fire Model

The Adaptive Leaky Integrate-and-Fire (ALIF) [40] neuron model represents a modified version of the traditional Leaky Integrate-and-Fire neuron model, incorporating adaptivity in the dynamics of membrane potential. This variation aims to capture specific aspects of neuronal behaviour that are not accounted for in the conventional LIF model.

One notable distinction between the ALIF and LIF models is the inclusion of an adaptation mechanism in the former. This adaptive mechanism enables dynamic changes to occur in the firing threshold of the neuron based on its recent spiking activity:

$$a^t = \rho a^{t-1} + z^{t-1} \quad (3.7)$$

$$A^t = v_{th} + \beta a^t \quad (3.8)$$

Where a^t represents an adaptive component, A^t is the adaptive threshold, β denotes a multiplicative term, ρ signifies the decay factor, and z^{t-1} corresponds to the output spike.

3.2 Input Encoding

Different encoding strategies are utilized to transform continuous input into spiking input, depending on the nature of the input. One such method is the Send On Delta (SOD) [41] spike encoding technique. This approach detects noteworthy amplitude fluctuations, whether positive or negative and encodes them as spikes. The process involves examining each sample of the input signal and generating spikes whenever significant changes in amplitude occur. To determine significance, the current signal value is compared with the previous spike's amplitude using a defined threshold. Positive and negative variations are stored separately for future reference.

The Time To First Spike (TTFS) encoding method, widely used for image encoding, encodes each pixel based on its intensity [42, 43]. High-intensity pixels generate spikes that arrive earlier, while low-intensity pixels produce spikes that arrive later in time.

The LIF neuron model is a biologically plausible approach for encoding real-valued signals into spike trains. In this method, the input signal is fed to a LIF neuron corresponding to a specific frequency channel index. Spikes are generated when the membrane potential of the neuron reaches a pre-defined threshold [44].

The Ben's Spiker Algorithm (BSA) is a widely recognized time series encoding method that falls under the category of stimulus estimation methods [45–47]. It utilizes linear filtering to estimate the stimulus of a biological neuron based on a sequence of spikes. BSA involves convolving the input signal with a finite impulse response filter and generating spikes when there is a significant deviation between the filtered signal and the input signal, surpassing a predetermined threshold value. However, determining optimal filter parameters and threshold values poses as an intricate challenge often resolved through grid search techniques by employing the Signal-Noise Ratio as an error metric [48].

The authors of [49] examine various biological encoding techniques. These techniques involve using the timing of spikes in multiple neurons to encode information. The effectiveness of signal encoding depends on the number of neurons utilized. One method involves storing information in the delay between a stimulus and the first spike of a neuron, with all other neurons being inhibited by this first spike resulting in single-spike encoding. Another approach called latency code encodes information based on time intervals between spikes across different neurons. Rank-Order Coding (ROC) is also discussed where information is encoded based on the specific order of spike occurrences. In this method, each neuron can only fire once per sample.

3.3 Training Methods

SNNs can be trained using a variety of learning algorithms, including STDP-based, gradient-based, and conversion from ANN to SNN.

Spike-Timing-Dependent Plasticity (STDP) is an essential biological process that adjusts the strengths of connections between neurons. In one common form of biological STDP, if a presynaptic neuron fires just before a postsynaptic neuron (around 10 ms), their connection strength is increased in what is known as Long-Term Potentiation (LTP). Conversely, if the presynaptic neuron fires after the postsynaptic neuron, the connection weakens through Long-Term Depression (LTD). The change in synaptic weight Δw can be calculated using the following rule:

$$\Delta w = \begin{cases} Ae^{-\frac{|t_{pre}-t_{post}|}{\tau}} & t_{pre} - t_{post} \leq 0, A > 0 \\ Be^{-\frac{|t_{pre}-t_{post}|}{\tau}} & t_{pre} - t_{post} > 0, B < 0 \end{cases} \quad (3.9)$$

where w represents the synaptic weight, A and B are the learning rates, and τ is the time constant (e.g., 15 ms) for the learning window. This rule models both long-term potentiation and long-term depression, with adjustments based on the time difference between pre- and postsynaptic spikes. SNNs often utilize modified versions of this rule rather than an exact biological implementation.

The second category of learning algorithms focuses on updating network parameters through the process of backpropagation (Sec. 2.3). However, this approach encounters difficulty due to the non-differentiable nature of spiking neuron activation functions. In order to overcome this challenge, an approximation of the derivative is employed using surrogate functions. Popular choices for surrogate functions $h(v)$ include equations such as:

$$h(v) = \frac{1}{a} \text{sign}\left(|v - v_{thr}| < \frac{a}{2}\right) \quad (3.10)$$

$$h(v) = \left(\frac{\sqrt{a}}{2} - \frac{a}{4}|v - v_{thr}|\right) \text{sign}\left(\frac{2}{\sqrt{a}} - |v - v_{thr}|\right) \quad (3.11)$$

$$h(v) = \frac{1}{a} \frac{e^{\frac{v_{thr}-v}{a}}}{\left(1 + e^{\frac{v_{thr}-v}{a}}\right)^2} \quad (3.12)$$

$$h(v) = \frac{1}{\sqrt{2\pi a}} e^{-\frac{(v-v_{thr})^2}{2a}} \quad (3.13)$$

where a represents the peak width of the derivative. The four functions mentioned, listed from top to bottom, correspond to the derivative of the rectangular function, the polynomial function, the sigmoid function, and the Gaussian function, as described in the reference [50]. Well-known algorithms that adopt this training paradigm are STBP [50], SpikeProp [51], Slayer [52], and e-prop [40].

The third approach involves converting an artificial neural network into a spiking neural network. In this method, the ANN is first trained and then transformed into an SNN. However, there are certain drawbacks to this approach. For instance, it does not fully utilize the temporal characteristics during training and may result in a small drop in accuracy. One popular framework that implements this training procedure is NengoDL [53].

Chapter 4

Supervised Learning with SNNs

Continuous monitoring of physical processes using distributed sensors is a crucial field, especially in SHM. SHM aims to ensure the safety of structures such as buildings and bridges by detecting structural changes caused by damage. AI with IoT and edge computing presents an opportunity to address these challenges effectively. I can potentially optimize data transmission overhead and improve response times by implementing AI detection algorithms directly on IoT sensor nodes. With the advent of low-cost, low-power, and increasingly accurate MEMS accelerometers, their application in distributed SHM is gaining popularity [54].

Furthermore, there have been proposed techniques to compress data from MEMS sensors located on different nodes before transmitting it to cloud storage and analytic facilities, aiming for signal compression at the edge [55]. Additionally, modal estimation methods directly on the sensor have been suggested to identify significant peaks in the acquired signal spectrum from MEMS data [56].

In order to enhance the efficacy of distributed structural health monitoring detection, recent proposals have focused on optimizing machine learning algorithms for edge computing. This involves utilizing code and libraries that are specifically developed for low-power microcontrollers [57, 58], which enables the implementation of these algorithms directly on the sensor or at an edge node.

An example illustrating this approach is found in hazard monitoring applications that utilize event-triggered single-channel micro-seismic sensors combined with advanced signal processing techniques. In such cases, a Convolutional Neural Network can be implemented on a low-power microcontroller as described in [59].

SNNs have gained considerable interest in the research community across various domains, including SHM. This enthusiasm stems from their brain-inspired, event-based

nature which holds promise for reducing energy requirements compared to traditional ANNs [60–64]. While ANNs have been successfully applied in SHM applications [54, 65–68], there is an increasing focus on SNNs due to their potential for more efficient information processing through sparse computations. For example, in [69], feedforward SNNs were employed for cost-effective inspection of damaged buildings using MEMS sensors.

However, the existing state of SHM applications lacks a comprehensive integration of a data processing pipeline that directly applies SNNs on the sensor node. SNNs have shown efficacy in handling time-series data from sensors, particularly when recurrent neural networks are involved [70]. Therefore, rather than using simpler feed-forward architectures, I have opted to explore recurrent SNNs for SHM data processing. Specifically, my focus is on an advanced implementation known as LSNN, which was introduced and documented in [71]. This choice is motivated by its appealing signal processing capabilities and learning potential within the context of structural health monitoring.

Furthermore, the analysis of input signal encoding is being explored due to its significant impact on subsequent computations and energy consumption. While SNNs have been successfully applied with event-based inputs such as pixel variations from Dynamic Vision Sensor cameras [72], they are also effective in processing continuous data streams as demonstrated in speech recognition applications [71]. However, determining the optimal encoding method for anomaly detection tasks in the field of SHM has not yet been exhaustively studied.

This study involves the implementation and evaluation of a low-power Spiking Neural Network on a sensor node with a commercial microcontroller and MEMS accelerometer. In this chapter, I present a comprehensive analysis comparing LSNN to Temporal Convolutional Network (TCN) and LSTM models, where LSNN is trained using both BPTT and e-prop methods. Both these methods are used to directly train the SNNs, in order to enhance the capabilities of the SNNs in the training phase. Furthermore, an investigation is conducted to examine the impact of input coding on performance metrics such as processing cycles and energy consumption in the microcontroller. To validate these comparisons, real data collected from a highway viaduct is utilized. The results demonstrate that LSNN has the ability to accurately detect structural variations associated with deterioration.

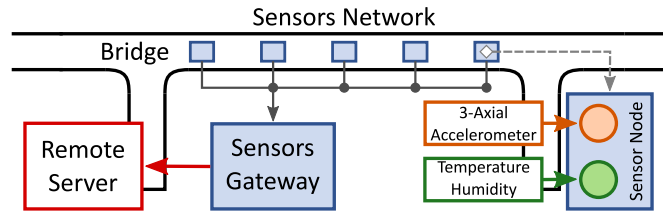


Figure 4.1: Monitoring system installation.

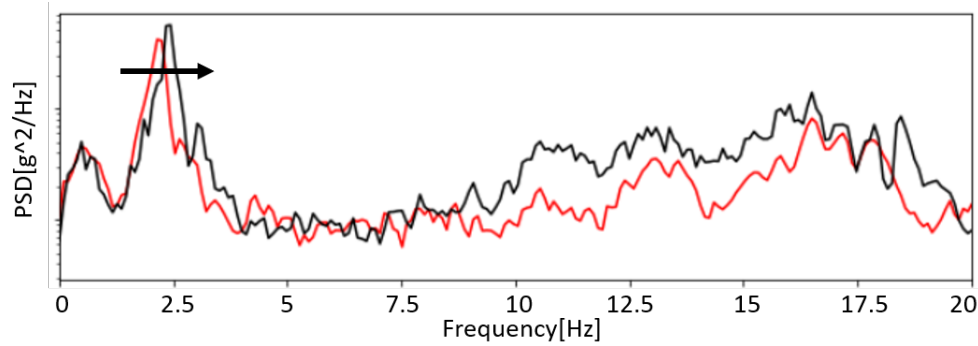


Figure 4.2: Mean natural frequency shift before and after scheduled maintenance.

4.1 Dataset

The object of my investigation is a viaduct on the A32 Torino-Bardonecchia highway, specifically, the section supported by two pairs of concrete pillars. My research focuses on this particular section and its sensors that have been installed for data analysis in preparation for necessary maintenance work to enhance the structural integrity of the viaduct.

The data acquisition system described in Fig. 4.1 consists of five sensor nodes, each featuring an STM32F4 microcontroller. These nodes capture acceleration and temperature data along three axes. The collected data is then transmitted to the cloud via a Raspberry Pi3 gateway with 4G connectivity. The interconnection between these nodes occurs through CAN-BUS communication with the gateway. It is important to emphasize that the data acquisition system solely collects the raw data without conducting any local signal processing at the edge. In this configuration, all data analysis occurs in the cloud. The accelerometers sample data at a rate of 25.6 kHz to prevent aliasing and then subsample it to achieve a final output frequency (f_s) of 100 Hz.

The cloud-based component comprises of a data-ingestion function responsible for receiving and storing incoming sensor readings from the gateway. Additionally, there are regularly scheduled analysis tasks implemented to monitor the structural health status of the bridge [73].

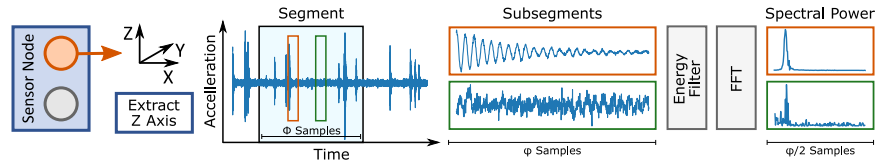


Figure 4.3: First data preprocessing and encoding pipelines.

The microcontroller unit utilized in this study is an ARM 32-bit Cortex-M4 chip running at a frequency of 168 MHz. It has 192 kB of SRAM and 1 MB of Flash memory, making it a popular choice for edge applications due to its energy-efficient design. Notably, the MCU incorporates a floating-point unit and a digital signal-processing library.

For data transmission and processing, the system utilizes a Raspberry Pi 3 module B as the gateway device [74]. This gateway has a Broadcom BCM2837 System-on-a-Chip. The SoC comprises four cores based on the Cortex-A53 architecture operating at 1.2 GHz with support for x64 instructions. Additionally, it includes 1 GB of DDR2 RAM. The Ubuntu operating system runs on the Raspberry Pi device and facilitates communication tasks using widely-supported Python interfaces such as MQTT brokers [75].

In terms of cloud infrastructure, there are separate storage resources along with computing nodes hosted on IBM’s cloud service platform.

The viaduct underwent a technical intervention to strengthen its structure, resulting in changes in the bridge’s natural frequencies. Fig. 4.2 visually shows how the PSD of the vibrations was altered before and after the intervention.

I utilize this unique dataset as a representation of an aged viaduct compared to a healthy one. The signals obtained after the intervention are considered normal data from a healthy bridge, while the data collected before maintenance is seen as anomalies since they were recorded on a damaged and aged structure. Although these data do not cover the entire history of the viaduct, they provide valuable insight into vibration patterns during two distinct structural phases of the bridge.

4.2 Preprocessing

In the studies that follow, I utilize two distinct preprocessing procedures and three encoding techniques. One of these techniques is event-based and results in spiking output, while the other two are current-based and produce scalar outputs that are fed into the SNNs.

One of the preprocessing approaches with the current coding strategy follows a series of steps shown in Fig. 4.3:

- The trace from each sensor was divided into segments lasting for T seconds, with each segment containing ϕ samples ($\phi = T * f_s$). These segments were then split into W sub-segments $W = w^0, \dots, w^{W-1}$, where each sub-segment contains φ samples ($\varphi \leq \phi$).
- The signal power for each sub-segment w was calculated using the formula:

$$P_w = \frac{1}{\varphi} \sum_{t=0}^{\varphi-1} w_t^2 \quad (4.1)$$

- If the signal power of at least one sub-segment exceeded a predetermined threshold P_{th} , the entire segment was considered a valid dataset sample; otherwise, it was discarded.
- The Discrete Fourier Transform (DFT) was used to calculate the spectral power P for each sub-segment. It was calculated using the following equation:

$$P = \frac{|X|^2}{\varphi}, \quad X = \text{DFT}(w) \quad (4.2)$$

Since the DFT is an even function, only half of φ , denoted as $s = \varphi/2$, were considered as input for the SNN.

- To eliminate high frequencies associated with vehicle approach frequencies that could affect classifier performance, a low-pass filter with a cutoff frequency (f_c) of 7 Hz was applied to the spectral power during filtering.

During dataset construction, I had flexibility in selecting parameters T and W , which influenced the input features represented by

$$s = \frac{\varphi}{2} = \frac{\phi}{2W} = \frac{Tf_s}{2W} \quad (4.3)$$

The number of samples in training and validation sets varied depending on segment length; longer segments resulted in fewer dataset samples. The output of this algorithm will be the input of the LSNN.

The second algorithm for preprocessing, depicted in Fig. 4.4, provides flexibility in how its output can be used. It can either serve as the current input for LSNN or be transformed into an event-based signal that acts as the input for LSNN. Below are the step-by-step details:

- After collecting data from the sensor, only the z-axis values are retained due to their higher sensitivity to vehicle passages. These z-axis data points are sampled at a rate of 100 Hz and then downsampled to 12.5 Hz.

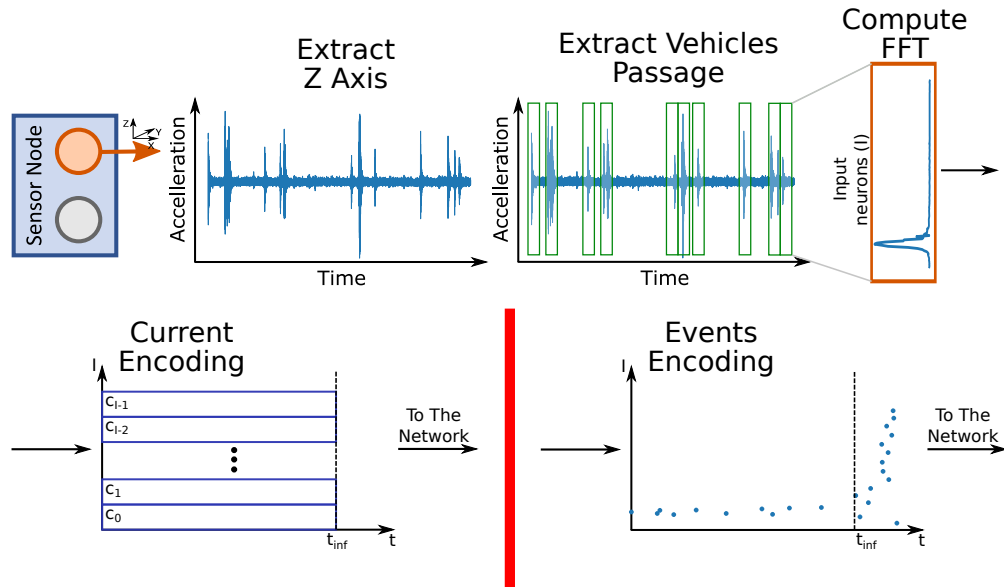


Figure 4.4: Second data preprocessing and encoding pipelines.

- Only the windows corresponding to vehicle passages are extracted by applying signal thresholding methods.
- The power spectra are calculated using the Fast Fourier Transform (FFT) on these windows. Two options are considered for the number of input neurons: 50 or 150. Practical considerations were taken into account to make this choice, as a larger number of inputs would result in an excessively large input weight matrix (W^{In}) for a relatively small network, while a smaller number of inputs may not capture sufficient signal features. Each window representing a vehicle’s data contains either 100 or 300 coefficients, corresponding to an 8 or 24-second time interval.
- These FFT coefficients serve as the input for the encoding stage. In Current-Driven encoding, each coefficient is assigned to an LSNN neuron for a fixed duration called t_{inf} .
- In the event-driven encoding approach, I employ the ROC algorithm. This algorithm encodes each coefficient as a spike time interval. Higher coefficients result in shorter “time-to-spike” intervals, meaning that coefficients with higher energy levels - which are more crucial for damage detection - will generate spikes earlier. Conversely, lower coefficients will produce spikes later on. It should be noted that t_{inf} denotes the inference time for each sample and any neuron that fails to fire within $t < t_{inf}$ becomes unable to do so thereafter. The ROC encoding algorithm used is outlined in Alg. 1.

Algorithm 1 Compute Time-to-Event**Require:** A signal S of I coefficients and a inference time t_{inf} **Ensure:** A Time-to-Event T array of I spiking intervals

```

 $m = \min(S)$ 
 $M = \max(S)$ 
for  $i \leftarrow 0$  to  $I - 1$  do
   $t = (S[i] - m)/(M - m)$ 
   $t_{check} = \text{round}(1/t)$ 
  if  $t_{check} \leq t_{inf}$  then
     $T[i] = t_{check}$ 
  end if
end for

```

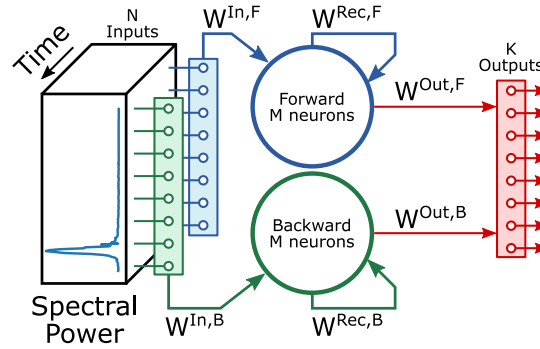


Figure 4.5: SNN's architecture. The blue ensemble represents the forward part, while the green one is the backward part. The former is used in the mono-directional LSNN while, in the bi-directional, the network has both components.

4.3 LSNN

Fig. 4.5 illustrates the network utilized in this investigation. It comprises the following elements:

- Input Layer: Consisting of N sources.
- Two Recurrent Ensembles: Each ensemble includes M neurons. The connection between the input layer and ensembles follows an all-to-all pattern, resulting in two connection matrices, namely $W^{In,x} \in \mathbb{R}^{M,N}$.
- Recurrent Connections: Both ensembles are connected to each other with a recurrent all-to-all configuration, employing two matrices denoted as $W^{Rec,x} \in \mathbb{R}^{M,M}$.
- Output Layer: Comprised of K neurons. The ensembles are fully interconnected with this output layer through two additional connection matrices: $W^{Out,x} \in \mathbb{R}^{K,N}$.

To enable bi-directionality in the LSNN, two recurrent ensembles are utilized. The blue ensemble (F) is responsible for processing forward samples and can be used in both

mono- and bi-directional LSNN configurations. On the other hand, the green ensemble (B) is exclusively employed when a bi-directional LSNN is desired and samples are fed in reverse order.

In this work, I utilized the ALIF model (Sec. 3.1.2) due to its ability to incorporate recent spiking activity, which is an improvement over standard LIF neurons. ALIF neurons have two hidden states: the membrane potential and the adaptive threshold.

The equation governing the membrane potential (v_j^t) of an ALIF neuron is as follows:

$$v_j^t = \overbrace{e^{-\frac{\delta t}{\tau_m}} v_j^{t-1}}^{\alpha} + \overbrace{\sum_{i \neq j} W_{ji}^{Rec} z_i^{t-1}}^{Lif \rightarrow Lif} + \overbrace{\sum_n W_{jn}^{In} x_n^t}^{In \rightarrow Lif} - \overbrace{v_{th} z_j^{t-1}}^{Reset} \quad (4.4)$$

$$z_j^t = \begin{cases} 1 & \text{if } v_j^t \geq v_{th} \text{ and } r_j^t \neq 1 \\ 0 & \text{otherwise} \end{cases} \quad (4.5)$$

Here, τ_m represents the membrane decay constant, x denotes network inputs, z_i characterizes the pre-synaptic activity of recurrent neurons, and z_j signifies the activity of a particular neuron for which the membrane voltage is computed.

The adaptive threshold (A_j^t) is described by:

$$a_j^t = \overbrace{e^{-\frac{\delta t}{\tau_a}} a_j^{t-1}}^{\rho} + z_j^{t-1} \quad (4.6)$$

$$A_j^t = v_{th} + \xi a_j^t$$

$$z_j^t = \begin{cases} 1 & \text{if } v_j^t \geq A_j^t \text{ and } r_j^t \neq 1 \\ 0 & \text{otherwise} \end{cases} \quad (4.7)$$

Here, τ_a represents the threshold decay constant, and ξ is a constant fixed at 0.07. The network's output is determined by the post-synaptic activity of the K output neurons:

$$y_k^t = \beta y_k^{t-1} + \sum_j W_{kj}^{Out} z_j^t + b_k \quad (4.8)$$

The parameter τ_o signifies the decay constant of the membrane potential of the neurons.

Table 4.1: SNN Experiment Results. ACC is the accuracy, MCC is the Matthews Correlation Coefficient, T represents the length in seconds of the samples, and W is the number of sub-segments.

Dataset	Network ¹	Exposure Time ²					
		ACC			MCC		
		1	5	10	1	5	10
T10-W05	TCN	92.1	-	-	0.84	-	-
	LSTM	92.0	-	-	0.83	-	-
	LSTM*	92.7	-	-	0.85	-	-
	LSNN - BPTT	66.4	82.3	88.0	0.29	0.66	0.75
	LSNN - eP.R	54.7	83.2	86.4	0.36	0.67	0.73
	LSNN* - BPTT	60.9	87.0	87.9	0.41	0.73	0.74
	LSNN* - eP.R	69.0	85.7	86.3	0.33	0.70	0.72
T10-W10	TCN	90.8	-	-	0.81	-	-
	LSTM	90.7	-	-	0.81	-	-
	LSTM*	91.3	-	-	0.82	-	-
	LSNN - BPTT	68.8	84.8	83.2	0.46	0.70	0.67
	LSNN - eP.R	73.9	84.9	83.9	0.56	0.70	0.69
	LSNN* - BPTT	77.8	84.6	85.4	0.61	0.69	0.70
	LSNN* - eP.R	75.6	81.6	82.9	0.57	0.64	0.65
T50-W05	TCN	95.0	-	-	0.90	-	-
	LSTM	95.1	-	-	0.90	-	-
	LSTM*	95.1	-	-	0.90	-	-
	LSNN - BPTT	71.4	93.6	94.4	0.50	0.86	0.88
	LSNN - eP.R	80.1	92.1	93.8	0.56	0.84	0.87
	LSNN* - BPTT	78.7	93.0	95.6	0.64	0.85	0.90
	LSNN* - eP.R	74.6	93.3	94.5	0.59	0.86	0.88
T50-W10	TCN	95.6	-	-	0.91	-	-
	LSTM	94.9	-	-	0.90	-	-
	LSTM*	95.1	-	-	0.90	-	-
	LSNN - BPTT	89.0	94.5	94.1	0.78	0.88	0.88
	LSNN - eP.R	88.2	93.9	93.4	0.76	0.87	0.86
	LSNN* - BPTT	87.2	91.9	92.2	0.75	0.83	0.86
	LSNN* - eP.R	84.5	92.6	93.4	0.71	0.84	0.86

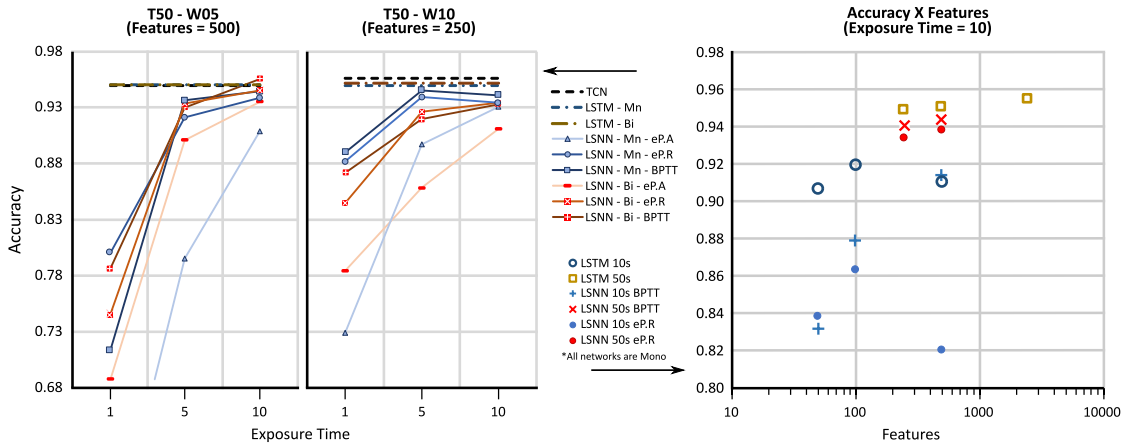


Figure 4.6: The graph on the right shows how accuracy changes when increasing the exposure time on T50-W05 and T50-W10 datasets. On the left, we can see a distribution of accuracy based on the number of spectral features and dataset type.

4.4 Results

4.4.1 Comparison among LSNNs, TCN, and LSTM

Table 4.1 presents the results obtained by various LSNN models and compares them to state-of-the-art neural networks such as TCN and LSTM. The LSNN models were trained using different techniques: e-prop random, adaptive e-prop, and BPTT. However, it should be noted that the LSNN model trained with adaptive e-prop did not achieve comparable results to the other methods and will not be discussed further.

The table also indicates that LSNN-eP.R has slightly lower accuracy compared to LSNN- BPTT, suggesting that e-prop is an approximation of BPTT. This decrease in accuracy aligns with the findings presented in [71].

Fig. 4.6.left, the impact of exposure time (t_{exp}) on network performance can be observed. Generally, increasing t_{exp} results in improved accuracy. This is because SNNs require more time to observe an input in order to transition from a transient state and adapt to new patterns. However, this trend does not hold for LSNN-Mn-eP.R and LSNN-Mn- BPTT networks as they only experience a slight decrease in accuracy at $W = 10$, which can be attributed to the increase in time depth from 50 to 100 integration steps.

Fig. 4.6.right illustrates the relationship between accuracy and the number of input features (s). The highest accuracy is achieved within a range of 250 to 500 features. Once beyond this range, the network’s performance declines due to difficulties in training

¹Models with “*” use a bi-directional network variant

²Used in SNN models only

Parameter	Description	Explored Values
N	Input neurons	50, 150
τ_m	Recurrent layer membrane potential decay	20, 30
τ_o	Output layer membrane potential decay	3, 10, 30
v_{thc}	Spike threshold coefficient	0.01, 0.03
β_c	Adaptive threshold coefficient	1.7, 1.8
τ_{ac}	Adaptive threshold decay	0.5, 1.0
t_{inf}	Inference ticks	5, 10, 20
reg_c	Loss regularization coefficient	1, 100, 300
reg_r	Firing rate regularization coefficient	0.01, 0.001

Table 4.2: LSNN parameters explored

the large input weight matrix W^{In} . Conversely, when $s < 250$, there is insufficient information available for effective network operation. Surprisingly, increasing subsample numbers does not consistently improve accuracy; specifically, LSNN with only 250 input features and $W = 10$ exhibited worse performance compared to LSNN with 500 features and $W = 5$.

4.4.2 Comparison between current-driven inputs and event-driven

The LSNN network architecture, as depicted in Figure Fig. 4.5, is a mono-directional neural network comprising 20 ALIF neurons and two output neurons used for classifying the bridge into its respective classes. The study investigates different hyperparameters classified into two groups: network parameters and training parameters. A summary of these parameters can be found in Table 4.2. Various network parameters were investigated during the study. These include:

- Input Number (N): Two values, 50 and 150, were considered for this parameter. N refers to the number of input neurons in the initial layer, and selecting an appropriate value is crucial to achieving a balance between network complexity and its ability to extract meaningful features from the input signal.
- Membrane Time Constant (τ_m): Two values, 20 and 30, were explored for this parameter. It determines how quickly the membrane potential decays in the recurrent layer. A higher value indicates that it takes longer for a neuron to return to its resting potential.
- Output Membrane Time Constant (τ_o): Three values - 3, 10, and 30 - were evaluated as options for this parameter. These are used to determine how rapidly the membrane potential decays in the output neurons.

- Output Membrane Time Constant (τ_o): Three values - 3, 10, and 30 - were evaluated as options for this parameter. These are used to determine how rapidly the membrane potential decays in the output neurons.
- The spike threshold coefficient (v_{thc}): Two values, namely 0.01 and 0.03, to determine the neurons' spike threshold.
- Adaptive Threshold Increase Constant (β_c): Two different values, 1.7 and 1.8, were investigated. This constant is utilized in calculating the increment of the adaptive threshold in ALIF neurons.
- Adaptive Threshold Decay Constant (τ_{ac}): Two different values, 0.5 and 1, were investigated. This constant governs how quickly the adaptive threshold decays over time.

The training loss is comprised of two components (Eq. 4.10): the classification loss (L_p) calculated using cross-entropy and the weighted firing-rate loss (L_{fr}). L_{fr} is computed as the discrepancy between the firing rate activity of the network and the firing rate regularization coefficient reg_r (Eq. 4.9). This thorough exploration of hyperparameters aids in comprehending their influence on both computational complexity and internal activity within LSNN.

$$L_{fr} = \frac{1}{2} \sum_j \left(\frac{1}{nT} \sum_{t=1}^{nT} z_j^t - reg_r \right)^2 \quad (4.9)$$

$$L = L_p + L_{fr} * reg_c \quad (4.10)$$

Several training parameters were investigated, including the inference time (t_{inf}) which was tested with values of 5, 10, and 20. The regularization coefficient (reg_c) was explored using values of 1, 100, and 300 to compute the weighted firing-rate loss during network training. Additionally, firing rate regularization coefficient (reg_r) was examined at rates of both 0.01 and 0.001 to determine the desired firing rate in the weighted firing-rate loss calculation (Table 4.2).

4.4.2.1 Accuracy vs. N and t_{inf} hyperparameters

In this study, an exhaustive search was conducted to evaluate the impact of different hyperparameters and eLSNN variations on network accuracy. The Matthews Correlation Coefficient (MCC) was used as a metric to measure accuracy on the test set,

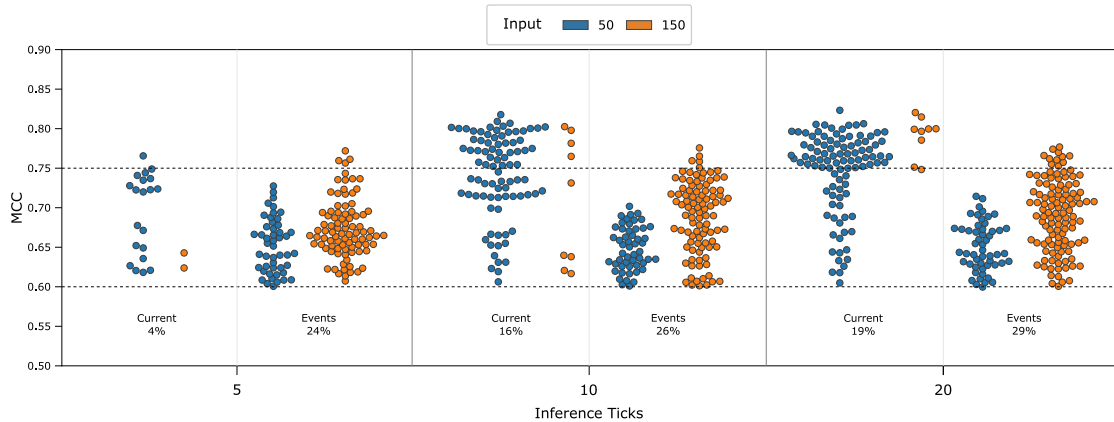


Figure 4.7: The x-axis represents the number of Inference Ticks, while the y-axis shows the MCC. Each data point depicts a different LSNN configuration with its unique set of hyperparameters. The percentage indicated corresponds to the proportion of SNNs plotted out of the total number of configurations explored, which was 1728.

which consisted of samples related to vehicle passages on a bridge section before and after maintenance. Figure 4.7 illustrates the relationship between network accuracy and various configurations. The plot only includes configurations that resulted in eLSNNs with MCC values equaling or exceeding 0.6, which corresponds to an accuracy of 0.77 or higher. A horizontal line at $MCC = 0.75$ represents “good” configurations with accuracies above approximately 0.88.

The x-axis of the figure displays three sets of plots, representing different values for inference ticks (t_{inf}). Each set includes a plot for current-driven eLSNNs and another one for event-driven eLSNNs. Within each plot, there are configurations of eLSNN that achieve an MCC value higher than 0.6. The percentage of these configurations out of the total evaluated (1728 for each type) is shown at the bottom of each plot. Configurations with an input number equal to 50 are depicted by blue bins, while those with an input number equal to 150 are represented by orange bins. In terms of structural health monitoring, the input number (N) corresponds to twice the magnitude of spectral components obtained from the FFT, which is equivalent to double the number of acceleration samples necessary for sensor data in conducting eLSNN inference.

The combination of these parameters (t_{inf} and N) constitutes the first-order hyperparameters that impact eLSNN inference execution time and energy consumption, as discussed in the next section. For current-driven eLSNNs, the largest number of acceptable configurations is achieved with an input number (N) of 50, which also corresponds to lower computational complexity. In contrast, event-driven eLSNNs require an input number (N) of 150 to achieve acceptable configurations. Additionally, for current-driven eLSNNs, accuracy improves with larger inference ticks, having acceptable configurations with $N = 50$ and $t_{inf} = 5$, as well as several with $N = 50$ and $t_{inf} = 10, 20$. However,

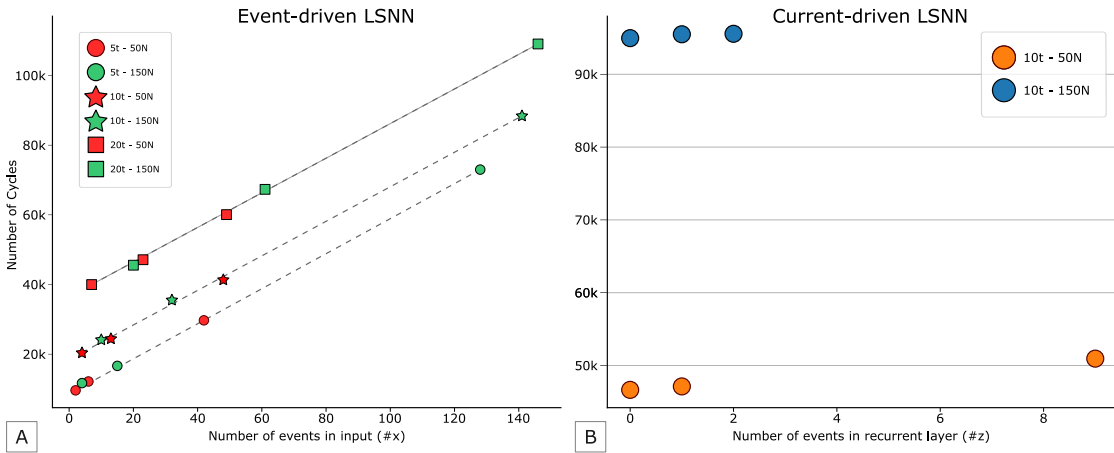


Figure 4.8: A) Cycle count for the event-driven eLSNN in different configurations. It includes variations in input numbers and inference ticks, computed on minimal, median, and maximal spike activities on the input layer (x). B) Cycle count for the current-driven eLSNN with variations in spike activities on the recurrent/hidden layer (z), computed on minimal, median, and maximal settings. Different colours represent different numbers of input configurations for each eLSNN network. All networks have 10 inference ticks.

this is not the case for event-driven eLSNNs, as their MCC does not significantly improve with increasing t_{inf} .

4.4.3 Execution time vs. activity-factors

The distribution of non-zero elements in the input layer for event-driven eLSNNs is depicted in Figure 4.9.B. This distribution depends on two hyperparameters, namely inference ticks (t_{inf}) and number of inputs (N). Each point along the y-axis represents a different configuration of these hyperparameters. The plot reveals that the number of events, spikes, or non-zero elements in the input layer (x) is more significant and exhibits greater variability compared to the recurrent/hidden layer (Figure 4.8.B). Additionally, both the average and standard deviation of input events/spikes/non-zero elements increase as I have higher values for input numbers and inference ticks.

The execution time of event-driven eLSNN for different configurations of the hyperparameters is shown in Figure 4.8.A. The plot demonstrates that both inference ticks and the number of events (x) have an impact on the execution time. There is a linear increase in execution time with higher values of these parameters. Notably, unlike current-driven eLSNNs, the input number (N) does not directly affect the execution time for event-driven eLSNNs but indirectly through its relationship with x .

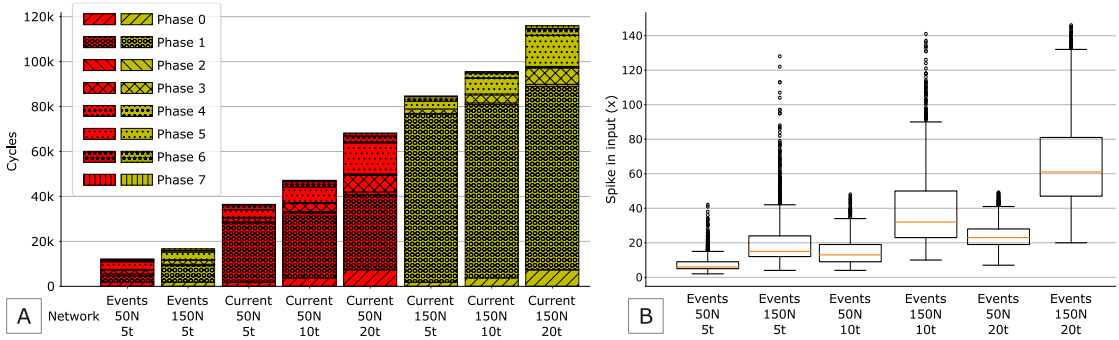


Figure 4.9: A) A breakdown of the cycle count for the best configurations of event-driven (event) and current-driven (current) eLSNN, computed on median activity conditions. B) The distribution of spikes in the input for different configurations of the event-driven eLSNN, with corresponding input numbers and inference ticks.

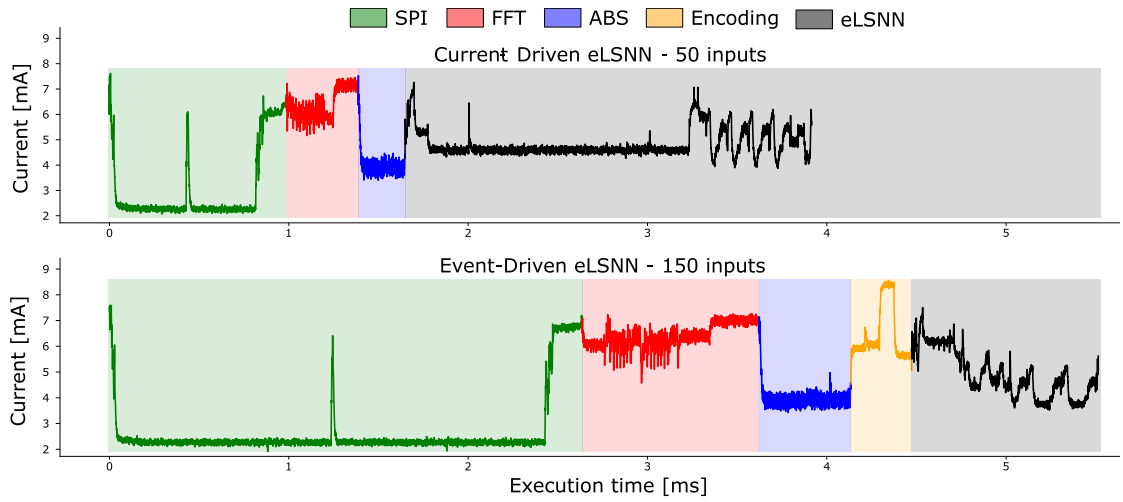


Figure 4.10: The current consumption patterns of the best eLSNN networks performing SHM applications are shown in the waveforms. Two sets of inputs were used: current inputs (top) and event inputs (bottom). The waveforms were obtained by reducing the MCU clock to 16 MHz and setting the SPI clock to 2 MHz, allowing for a clearer depiction of different stages in the application.

Stage	Current-Driven				
	Time [μ s]	Cycles [#]	Current [mA]	Power [mW]	Energy [μ J]
SPI	542	-	5.01	16.53	8.96
FFT	41	6971	38.76	127.91	5.24
ABS	26	4284	36.58	120.71	3.14
Encoding	-	-	-	-	-
eLSNN	215	36036	40.26	132.86	28.56
Total	824	47291	-	-	45.90
Mean	-	-	16.88	55.70	-

Table 4.3: The power analysis of the SHM application featuring the best LSNNs tested. The SPI master transfers data at a clock speed of 8 MHz. The SPI stage includes three components: i) transferring data via SPI, ii) reconfiguring the system-on-chip's clock to 168 MHz after waking up from SLEEP mode, and iii) converting sensor data from raw integers to floating-point format.

Stage	Event-Driven				
	Time [μ s]	Cycles [#]	Current [mA]	Power [mW]	Energy [μ J]
SPI	949	-	4.33	14.29	13.56
FFT	101	16968	38.17	125.96	12.72
ABS	50	8316	35.97	118.70	5.94
Encoding	32	5459	42.65	140.75	4.50
eLSNN	99	16631	38.20	126.06	12.48
Total	1231	47374	-	-	49.20
Mean	-	-	12.11	39.97	-

Table 4.4: The power analysis of the SHM application featuring the best eLSNNs tested. The SPI master transfers data at a clock speed of 8 MHz. The SPI stage includes three components: i) transferring data via SPI, ii) reconfiguring the system-on-chip’s clock to 168 MHz after waking up from SLEEP mode, and iii) converting sensor data from raw integers to floating-point format.

4.4.4 Event-driven vs. Current-driven

Fig. 4.9.A presents a detailed analysis of the total number of cycles needed by the median inference time for current-driven and event-driven eLSNNs. The breakdown showcases the different computational phases involved in executing an eLSNN for various configurations, with a focus on N and t_{inf} hyperparameters and using the median sample data considering both x and z .

The chart displays all the $N - t_{inf}$ hyperparameter combinations for current-driven eLSNNs, while only configurations with inference ticks equal to 5 are shown for event-driven eLSNNs, as they are considered the most energy-efficient networks.

In comparing the different networks, it is observed that the two highest-performing eLSNN configurations, with $MCC \geq 0.75$, are $N = 150$ and $t_{inf} = 5$ for event-driven eLSNNs and $N = 50$ and $t_{inf} = 5$ for current-driven eLSNNs. Notably, the event-driven eLSNN requires less than half of the cycles compared to the current-driven eLSNN. This indicates that the event-driven variant is significantly more efficient (over 50%) in comparison to its current-based counterpart.

It should be noted that this finding does not take into consideration the preprocessing of the input sample, which has a significant impact on the input number and is especially relevant for the chosen event-driven eLSNN configuration. Interestingly, even though the event-driven eLSNN configured with $N = 50$ and $t_{inf} = 5$ is the most efficient in terms of execution time, it falls slightly below the MCC threshold at 0.72.

In Figure 4.9.A, different patterns represent the number of cycles required for various computational phases (Eq. 4.4, 4.5, 4.6, 4.7, and 4.8), including the following:

- Phase 0: Compute αv (Eq. 4.4)
- Phase 1: Compute $W^{In}x$ (Eq. 4.4)
- Phase 2: Compute $W^{Rec}z$ (Eq. 4.4)
- Phase 3: Compute ξa^t (Eq. 4.6)
- Phase 4: Compute $\rho a + z$ (Eq. 4.6)
- Phase 5: Compute z (Eq. 4.5)
- Phase 6: Compute y (Eq. 4.8)
- Phase 7: Prepare for the next iteration

Phase 1 is the most time-consuming step in current-driven eLSNNs as it involves a full matrix-vector multiplication. Phases 0 and 3 each require N multiplications, and their execution times become more noticeable with increasing t_{inf} . This breakdown provides insight into how the computational load is distributed across different phases and contributes to the overall execution time.

Fig. 4.10 depicts the current consumption of the MCU during various processing steps, including reading sensor values via SPI interface, preprocessing the sample (FFT + ABS), and computing the eLSNN kernel. The left plot represents the current-driven eLSNN in its optimal configuration, while the right plot illustrates the event-driven eLSNN in its optimal configuration.

On the one hand, the event-driven eLSNN kernel requires fewer cycles than the current-driven eLSNN. However, it also requires three times as many sensor readings to compute an inference for a given input sample. This increase in sensor readings adds complexity to the computational process and results in higher cycle counts needed for reading data from SPI, performing FFT calculations, and computing absolute values. Furthermore, compared to the current-driven eLSNN, the event-driven approach involves an additional preprocessing step that includes coding spectral components into spikes/events.

The network with the current input necessitates a matrix-vector multiplication for each input tick $t_{inp} = t_{inf}/t_{exp}$. In this particular case, there is only one input tick and a sole matrix-vector multiplication occurs. However, even with just one computation, it significantly increases the number of cycles required. Fig. 4.10.A clearly shows this phase lasting from approximately milliseconds 2 to 3, whereas it is absent in Fig. 4.10.B.

This analysis provides insight into the impact of processing steps on both current-driven and event-driven eLSNNs' current consumption. It highlights the trade-offs

between computational efficiency, sensor readings, preprocessing steps, and energy consumption for optimal performance.

Table 4.3 and Table 4.4 summarize the observed effects. Although the event-driven eLSNN kernel requires fewer cycles (54%) compared to the current-driven eLSNN network, the total computation time is 1.51 times longer for the event-driven eLSNN due to a longer SPI transfer time. However, even with this additional execution time, it still meets the real-time requirements of the SHM application

The energy consumption of both event-driven and current-driven eLSNN flavours is comparable, ranging from 46-49 μJ . This shows that although the event-driven eLSNN requires extra execution time, the lower power cost of the SPI transfer (with DMA) compensates for it in terms of overall energy consumption between the two flavours.

Further research could focus on investigating temporal coding methods and implementing coding directly on the MEMS sensor in the time domain. By using on-sensor coding techniques, it may be possible to eliminate the requirement for FFT and decrease the data volume that needs to be read from the MEMS sensors. This would result in reduced energy consumption within the sensor node.

4.5 Conclusion

This study explores the potential of LSNNs for structural health monitoring applications and compares their effectiveness to ANNs. Through experiments, it was observed that LSNNs can effectively analyze input sequences and perform classification tasks. The versatility and robustness of LSNNs were demonstrated by testing them on different versions of the dataset with varying spectral features and signal lengths. The best-performing LSNN model, trained using e-prop, achieved a classification accuracy of $MCC = 0.88$ in distinguishing between damaged and healthy conditions of a bridge. These results highlight not only the capabilities of SNNs in SHM but also indicate the possibility of energy-efficient hardware platforms powered by neuromorphic accelerators.

Additionally, I introduced an enhanced version of the LSNN specifically designed for SHM applications on microcontroller-based platforms. I investigated the operational behaviour of spiking neural networks, with a particular focus on evaluating the benefits and trade-offs associated with event-driven inputs.

In my investigation, I focused on examining different methods for encoding input data and assessing their impact on system performance and energy usage. My results reveal the trade-offs between the operational requirements of eLSNNs and the overhead

associated with transferring data, providing guidance for selecting optimal configurations that balance energy efficiency and performance. Through empirical analysis of real-world applications, I demonstrate that LSNNs are a viable approach in structural health monitoring, achieving accuracy $MCC \geq 0.75$ while minimizing energy consumption and computational overhead, particularly when compared to the expenses incurred from transferring data.

Chapter 5

Reinforcement Learning with SNNs

RL tasks naturally involve a temporal aspect and are known for their complexity. In recent years, SNNs have emerged as a promising tool for solving RL problems efficiently [40]. The appeal of SNNs in this context stems from two key factors. Firstly, the inherent time dimensionality and dynamic nature of SNNs enable them to effectively retain information within the membrane potential. Secondly, empirical findings indicate that SNNs outperform traditional ANNs in terms of expressiveness [2].

In the past, reinforcement learning tasks were mainly performed on powerful computational platforms. However, there is now a trend towards running complex cognitive tasks on resource-limited platforms such as nano- and pico-sized vehicles. This shift emphasizes the attractiveness of using spiking neural networks due to their energy efficiency, which makes them well-suited for these emerging applications [76].

5.1 Dynamic Vision Sensor

DVS is a neuromorphic sensor that mimics the functioning of animal eyes. Unlike traditional cameras, the DVS only generates events when there is a change in log

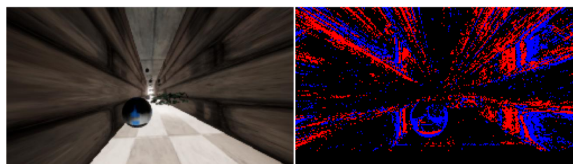


Figure 5.1: Comparison between an RGB image and a DVS image.

luminance, disregarding static background information and reducing redundant data transmission. The output of the DVS consists of asynchronous and sparse events, each containing pixel coordinates, timestamps, and polarity values $\{-1, 1\}$.

In my study, a drone equipped with a DVS camera is utilized as an input source for the agent. The agent then employs a neural network to acquire knowledge about its surroundings and make decisions regarding navigation and obstacle avoidance. An illustrative example of simulated DVS output alongside a comparison with an RGB image is presented in Figure Fig. 5.1. In the right-hand side DVS image, events are depicted by red pixels (positive events), blue pixels (negative events), and black pixels indicating background activity (no events). The left-sided image represents the RGB version of the scene for reference purposes.

5.2 Neuromorphic Platforms

Neuromorphic platforms can be categorized into two main types: analogue and mixed-signal, and digital SNN accelerators. Platforms falling into the former category are typically more efficient, offering a smaller neuron area footprint for the same neuron model [77, 78]. However, their neuron functionality often relies on technology-specific implementations, operating individual transistors in their sub-threshold regime. Consequently, significant engineering efforts are required for porting to different technology nodes.

On the other hand, digital neuromorphic platforms implement less complex neuron models, typically based on LIF or its derivatives [79–82]. These platforms use equation-solver data paths composed of digital elementary adders and multipliers. Notably, digital neuromorphic platforms find applications beyond neuromorphic simulation, enabling fast integration into digital SoCs [83] and technology porting [84].

Across various state-of-the-art neuromorphic hardware, common characteristics induce approximations in SNN models. Both Loihi [79] and Spinnaker2 [85] support biases, while SNE omits them to avoid slowing down simulations and increasing energy consumption. Access to membrane potential at runtime enhances network expressiveness, with Spinnaker2 allowing it [85], Loihi providing access only at the simulation’s end [79], and Sparse Neural Engine (SNE) lacking this capability.

SNE, with high quantization weights (4 bits) and significantly lower energy consumption, embodies a worst-case scenario, imposing multiple approximations on the neuron model. Hence, the targeted deployment platform for the quantized spiking neural network in this study is an implementation of SNE [84]. SNE is a fully digital, non-Von

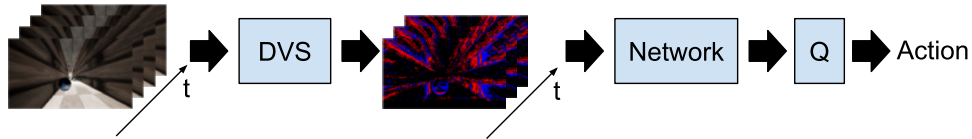


Figure 5.2: The setup process involves converting RGB frames into DVS frames, which are then inputted into the network. The network generates a Q-function as output, which is utilized to determine the appropriate action for the drone to execute.

Neumann data-flow architecture with DMA capabilities, implementing a programmable number of digital LIF neurons. Integration into a conventional SoC involves connecting it to two dedicated memory ports and one configuration advanced peripheral bus (APB).

SNE utilizes an explicit coordinate list (COO) representation encoded on 32 bits to address and consume single events (input feature maps) linearly stored in the main system memory. The architecture features a dedicated local memory to store up to $256 \times 3 \times 3$ 4-bit-quantized convolutional kernels. The firing threshold and LIF exponential decay time constant are held by dedicated configuration registers and can be programmed at runtime. In this work, reference is made to an SNE configuration with 8 parallel computing engines, each simulating 1024 configurable LIF output neurons.

5.3 SNNs with D2QN

The main purpose of this research is to evaluate the effectiveness of utilizing an SNN in an RL algorithm, in particular D2QN, specifically for tasks involving obstacle avoidance. To accomplish this, a comparative analysis is conducted between the SNN-based approach and an ANN-based version previously discussed in [86]. The core components utilized for this comparison are outlined in Fig. 5.2.

My assessment starts by feeding RGB frames as input to a DVS simulator. This simulator converts the frames into DVS images. These DVS images are then used as input for neural networks that model the Q-function. The result from these networks is represented as a Q-array, which determines the action taken by the agent based on selecting the maximum value within it.

After comparing the approaches of SNN and ANN, I move forward to create a framework for simulating the behaviour of SNE [84]. In this framework, I incorporate both membrane potentials and weights quantization to align with the limitations imposed by the hardware target implementation.

In addition, I engage in developing a novel architecture for SNN that is specifically designed to function smoothly within the SNE framework. My approach involves utilizing different training algorithms and adapting various SNN topologies to effectively address the limitations imposed by both the hardware implementation and the specific reinforcement learning task.

5.3.1 Spiking Neural Network and DVS Input Model

My research explores the utilization of Spiking Neural Networks in reinforcement learning tasks, which necessitates considering various aspects related to network architecture and input representation for SNNs employed in classification tasks.

In classification tasks, SNNs operate over multiple computational time steps referred to as “ticks” (N). During these ticks, the same input data is repeatedly presented to the network, and its spike activity is measured for class assignments. This iterative process allows the network to adapt and reach an optimal state for performance. Recent studies have emphasized that tuning the duration of each tick is a crucial hyperparameter with a potential impact on network performance [87].

Two common methods are used to keep the input constant for a specific period. First, in the case of event-based input like frames from a DVS camera, the input sequence is recorded and replayed multiple times [88]. This guarantees that the network processes the same data repeatedly for a desired duration. Secondly, when a RGB or grayscale frame is provided as input, it remains constant for a specified period of time. During this duration, the encoding technique transforms the frame into spike trains. This transformation can be achieved through different methods: i) Mapping pixel values to the probability of neuronal firing to create spike trains [89]. ii) Utilizing pixel values as a source of electrical current for neurons in the input layer which affects their firing rates [40].

To ensure the persistence of input information, these strategies are employed to allow the spiking neural network to reach a stable state and achieve accurate classification. Selecting an encoding strategy can have implications for the network’s performance and computational efficiency.

My research focuses on reinforcement learning using event-based inputs, which are characterized by temporal variations and encoding as spike events. These inputs are obtained through a simulator for a DVS camera. In the context of reinforcement learning, the network’s objective is to observe and interact with the environment in order to estimate a Q-function. Unlike classification tasks, there is no predefined sequence to

classify. Therefore, it is not feasible to record and repeat scenes as the network must process input events in real-time as they occur. Moreover, the network operates within time constraints due to the agent’s need for prompt decision-making and navigation in its environment. To accommodate these factors, I choose to directly utilize the spikes produced by the DVS camera as input for the network. This strategy not only reduces latency but also guarantees real-time processing of input information by the network.

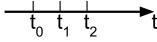
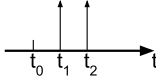
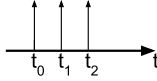
The network receives a series of DVS (differential) images that are combined together. The total number of ticks in the network (N) matches the number of provided DVS frames. The SNN analyzes all the combined frames and generates an estimated output for the Q-function. To prevent excessive delay, it is recommended to keep the number of stacked frames relatively low as a design consideration. My experimental findings suggest that my approach can achieve satisfactory accuracy with just three stacked frames, proving its feasibility.

To adapt the event-based input and time dynamics of SNNs for the reinforcement learning task, I incorporate certain considerations in my approach. First, to account for the fact that SNNs can only handle positive spikes and the inherent temporal nature of SNNs, I convert negative DVS events into positive events. These converted events are then directed to a separate input channel alongside the positive events. As a result, two input channels are required: one for positive events and another for converted negative events. Second, I present stacked frames as a temporal sequence of samples to the SNN in order to detect environmental status. The number of time steps (“ticks”) is equal to the number of stacked frames S , where S represents 3 in my specific setup configuration ($N = S = 3$). During this process, each individual frame is presented to the network for a duration equivalent to 1 tick.

The neural network used for the reinforcement learning task includes two types of neurons: i) LIF neurons in the hidden layer, and ii) non-spiking LIF neurons in the output layer. Both types of neurons have a bias term. This particular network configuration is considered my baseline SNN, referred to as SCNN.

Modern neural network architectures, which are commonly used in advanced applications, often consist of billions of parameters that are represented as single-precision floating-point values for numerical stability during training. However, this choice comes at the cost of increased memory usage [90]. These neural networks can have substantial memory footprints, typically requiring several tens of gigabytes. This poses a challenge when deploying these networks on embedded computing platforms with limited available memory.

Table 5.1: Three instances of neuron output values calculated using different approaches. In these illustrations, I utilize a duration of three ticks ($N = 3$).

		Case 1	Case 2	Case 3
Activity				
Output	Spikes	$z_0 = 0; z_1 = 0;$ $z_2 = 0$	$z_0 = 0; z_1 = 1;$ $z_2 = 1$	$z_0 = 1; z_1 = 1;$ $z_2 = 1$
	i: $1/N * \sum_i z_i$	0	0.67	1
	ii: $\sum_i 2^i * z_i$	0	6	7
	iii: Eq. 3.4 & Eq. 3.5	$v(t) < v_{th}$	v_{th}	v_{th}
	iv: Eq. 3.4	$v(t) \in (-\infty, \infty)$	$v(t) \in (-\infty, \infty)$	$v(t) \in (-\infty, \infty)$

In order to evaluate the practicality of implementing an SNN-based agent on actual neuromorphic hardware, I have made adjustments to the SNN model to conform with the limitations and capabilities of these hardware platforms. Furthermore, I have employed quantization methods to minimize memory demands and ensure suitability for embedded devices [91].

In this context, I am investigating quantization techniques to decrease the precision of network parameters and internal representations in order to ensure compatibility with the targeted SNE neuromorphic accelerator. Traditional approaches for quantization involve transitioning from 32-bit or 64-bit floating-point representations to smaller 16-bit floating-point representations. However, I am exploring more aggressive quantization strategies that aim for significantly reduced bitwidth parameters, such as using 4-bit quantized network weight parameters and 8-bit quantized internal membrane potential representation [92]. This level of quantization is specifically tailored to meet the requirements of the SNE neuromorphic accelerator.

It is noteworthy to mention that in SNNs, the intermediate feature maps, which are the outputs generated by neural network layers, are intrinsically quantized with 1-bit activations. Unlike continuous-valued activations in conventional CNNs, the activation function of SNN layers generates binary events that indicate when a neuron’s membrane potential surpasses a threshold at a specific time point.

5.3.2 The Adaptation of STBP for Q-function Estimation

In this study, I focus on utilizing the SNN to estimate a Q-function for reinforcement learning. The accurate representation of the Q-function is crucial in navigating complex environments using photorealistic simulations. To achieve high expressiveness, I adopted the STBP training algorithm proposed in [50], which is specifically designed for training

Table 5.2: Comparison between the features of a fully customized neuron and the neuron in SNE.

Features	LIF Neuron	SNE LIF Neuron
Bias	✓	✗
No-spiking neuron	✓	✗
Type of reset	Hard & Soft	Hard
Access to $v(t)$	✓	✓
Access to $z(t)$	✓	✓

SNNs in classification tasks. This algorithm relies on a pseudo-derivative (Eq. 3.10) and uses average output activity as a key component.

Table 5.1 provides three examples for each of the output strategies discussed below. These examples are computed using a specific number of time steps (N), which in this case is set to 3. The first row represents the plot of the output activity, while the second row shows the output activity translated into spike events.

- **Mean Output Activity:** This strategy calculates the average output activity, resulting in limited expressiveness since the output values are constrained within. The possible levels for outputs (n_{ol}) are directly determined by the number of time steps (N) in the network: $n_{ol} = N + 1$.
- **Time to Spike:** To increase expressiveness, I can consider “time to spike” from my last layer. Using this approach expands my range with $n_{ol} = 2^N$ possible levels for outputs.
- In order to enhance the expressiveness of the output, one approach is to utilize the membrane potential $v(t)$ of spiking neurons in the output layer. This allows for a wider range of values within the interval $v(t) \in (\infty, v_{thr}]$, resulting in an effectively infinite number of output levels ($n_{ol} = \infty$). It should be noted that this implementation cannot exceed values greater than v_{thr} .
- To overcome the limitation mentioned above, an alternative option is to employ non-spiking LIF neurons as the output. This enables a broader span of values for the output membrane potential within $v(t) \in (-\infty, \infty)$, and therefore leads to an effectively infinite number of output levels ($n_{ol} = \infty$).

Table 5.3: Networks architecture. HR-LIF stands for Hard-reset LIF and NS LIF for Non-spiking LIF

	CNN	SCNN	eSCNN	qeSCNN	pqeSCNN
Input channels	3	2	2	2	2
CONV1	(16, 8, 4)	(16, 8, 4)	(16, 8, 4)	(16, 8, 4)	(16, 8, 4)
CONV2	(32, 8, 4)	(32, 8, 4)	(32, 8, 4)	(32, 8, 4)	(32, 8, 4)
FC1	512	512	512	512	512
FC2	5	5	5	5	5
Bias	✓	✓	✗	✗	✗
Hidden neurons	ReLU	HR LIF	HR LIF	HR LIF	HR LIF
Output neurons	Linear	NS LIF	NS LIF	NS LIF	NS LIF
Quantization	✗	✗	✗	Full quantized	No last layer

5.3.3 Spiking Neural Network on SNE Embedded Neuromorphic Accelerator

To meet the requirements of the SNE accelerator, modifications are made to the neuron model, and weight quantization is applied in adapting the SNN model. The discrepancies between the software neuron model and SNE implementation are summarized in Table 5.2, with a notable distinction being the exclusion of bias from the neuron model to align with that of SNE. This adapted network is denoted as eSCNN.

When utilizing SCNNs on embedded neuromorphic hardware such as SNE, it is necessary to encode the network parameters such as weights and membrane potential using a bit-width that the hardware can accommodate. Specifically, on SNE, weights are encoded as 4-bit signed integers, while the LIF membrane potential, which serves as an internal neuron state variable, is represented with 8 bits.

Quantlib [91], a software tool that facilitates the quantization of weights to meet specific numerical precision requirements, has been enhanced to support the implementation of the LIF function as an activation function on SNE hardware. This updated version allows for the deployment of fully quantized SNE networks, which consist of non-spiking neurons in the last layer and spiking neurons in all other layers. The resulting network is referred to as qeSCNN. Quantizing both the weights and membrane potential according to hardware constraints is essential for ensuring efficient operation of the SNN on the SNE platform.

Another variation of the SNN implementation was utilized in addition to qeSCNN. In this version, the final layer operates on the general-purpose microcontroller of the SNE platform. This modification allows for increased numerical precision: i.e. n_{ol} value is not limited to 8 bit. The purpose of this adjustment is to account for any possible reduction in expressiveness within the SNN caused by limited time steps used

in the decision layer. This network is referred to as pqeSCNN. Further details regarding compared networks can be found in Table 5.3.

In order to assess the accuracy of inference on the SNE hardware platform, I simulated the behaviour of LIF neurons implemented on SNE. This simulation involved replicating the 8-bit membrane potential decay performed on real hardware using a lookup table, as well as simulating how the neuron’s membrane potential is updated when a spike occurs and how it resets once the threshold is exceeded, returning to its resting value. These simulations were conducted by extending the base neural network classes defined in the Quantlib framework [91].

The synaptic connectivity supported by SNE includes both convolutional and fully connected types.

5.4 Training and Evaluation Framework

In this section, I provide an overview of the simulator utilized for training the drone on obstacle avoidance. This includes details about the environment characteristics, RL parameters, and DVS configuration.

5.4.1 Simulation Environment and DVS Model

To enable the reinforcement learning agent’s training for obstacle avoidance, a simulator environment is utilized. For this purpose, AirSim [93] is selected as the simulator of choice. Built on Unreal Engine 4 [94], AirSim provides realistic physics simulations and rendering capabilities. Its photo-realistic nature enables neural networks to be trained in a simulated setting while still maintaining acceptable performance when deployed in real-world scenarios [95].

To integrate the DVS into the simulation, I employ a DVS simulator called v2e [96]. This tool accurately models the dynamics and noise associated with DVS. By integrating v2e into AirSim, I can process the images generated within the simulator environment using this tool. The processed DVS data is then utilized as input for training an RL agent in a realistic and dynamic event-based sensor environment.

5.4.2 Training and Evaluation Setup

To conduct experiments and evaluation of reinforcement learning models for obstacle avoidance, a simulated environment is utilized in this study. The setup closely follows

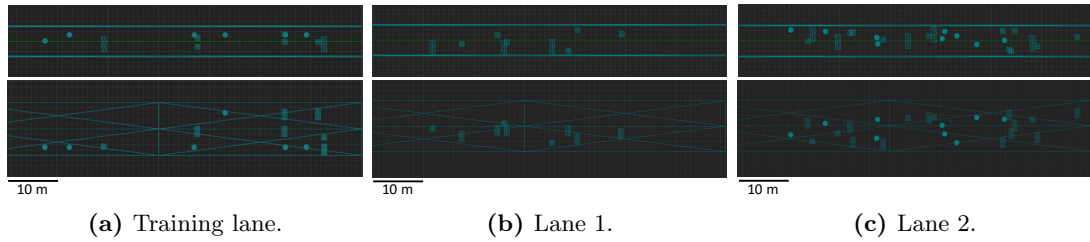


Figure 5.3: The arrangement of obstacles in the three lanes. The first row depicts the representation from the top, while the second row shows the representation from the left side.

the framework presented in [86], with three distinct lanes implemented within Unreal Engine 4. These lanes span 70 meters along the y-axis and feature diverse obstacles that mimic real-world challenges faced by drones or agents. Outlined details of the lanes include:

- Fig. 5.3(a): Training lane will serve as the designated area for training my RL model. This lane has been designed with 16 obstacles to provide a suitable level of challenge for effective model training. Furthermore, I will also utilize this lane for evaluating the performance of the trained model and assessing its effectiveness.
- Fig. 5.3(b): Lane 1 features a simpler setup with only 9 obstacles. This configuration allows for a more direct assessment of the model’s fundamental capabilities.
- Fig. 5.3(c): Lane 2 is densely packed with 25 obstacles. This setting aims to test the model’s performance in complex scenarios.

In this setup, the interaction between the environment and the agent involves two primary types of data: observations and rewards.

The input data for the agent consists of three pre-processed frames obtained from a DVS. These DVS frames were generated using v2e, a tool that simulates the functionality of real DVS cameras [96]. The v2e toolbox generates frames that are expressed as a series of tuples, wherein each tuple comprises the following details:

- Timestamp: The exact time when the event (alteration in log-luminance) happened.
- X-coordinate: The horizontal location where the event was detected on the pixel.
- Y-coordinate: The vertical position where the event was identified on the pixel.
- Polarity: This value can either be 1 or -1, denoting an increase or decrease in log-luminance respectively.

The agent uses rewards as a form of feedback to guide the learning process. In this particular environment, the rewards are calculated using a predefined formula:

$$R(s, a) = \gamma_y \Delta y - \gamma_c C - \gamma_a A \quad (5.1)$$

where, R denotes the reward assigned to a particular state s and action a , while γ_y , γ_c , and γ_a are regularization factors that assign different weights to specific components of the reward. Δy represents the vertical distance between the current position of the agent and its desired goal. C acts as an indicator if there has been any collision with obstacles along the way. Lastly, A signifies the action chosen by the agent in response to its environment.

To determine the value of action A , the following process is followed:

- If the action corresponds to maintaining the current course (index 4), then A is assigned a value of 0.
- For all other actions, A is assigned a value of 1.

The C flag signifies if there has been a collision or if the agent has surpassed the maximum allowable number of actions (M). The episode concludes when either the agent reaches its goal or when $C = 1$.

The agent conveys its decision through actions, and this environment is defined by discrete action options. These options include:

- 0: steer left to avoid obstacles
- 1: steer right to avoid obstacles
- 2: move downwards to avoid obstacles
- 3: move upwards to avoid obstacles
- 4: maintain the current course.

All the parameters used in the simulation are in Table 5.4.

Table 5.4: Simulation parameters value and description.

Parameters	Values	Description	
DVS	Positive Threshold	0.2	Nominal threshold of triggering positive event in log intensity
	Negative Threshold	0.2	Nominal threshold of triggering negative event in log intensity
	Sigma Threshold	0.03	Standard deviation of threshold in log intensity
	Cutoff Frequency [Hz]	300	3 dB cutoff frequency of DVS photoreceptor
	Leak Rate Frequency [Hz]	0.01	Leak event rate per pixel
	Shot Noise Rate Frequency [Hz]	0.001	Shot noise rate
Reward	γ_y	1	Weight for the distance reward
	γ_a	-0.1	Weight for the action reward
	γ_c	-10	Weight for the collision reward
	M	250	Maximum amount of actions that the drone can take
Simulation	γ	0.99	Discount factor
	Staked Frames	3	Number of staked frames feed to the neural network
	# Train Games	5000	Number of training games
	# Test Games	100	Number of testing games per lane

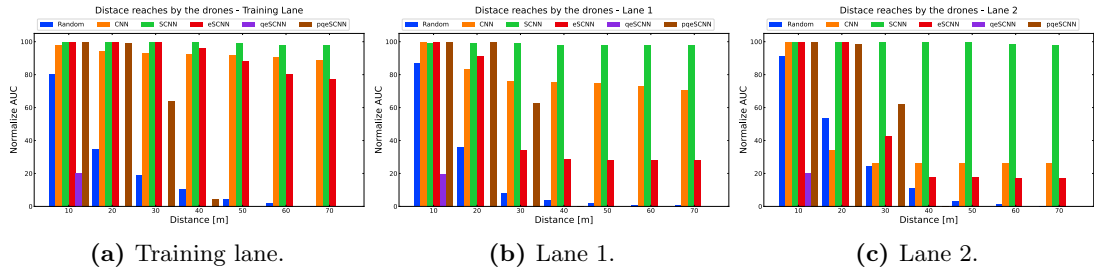


Figure 5.4: Performance of the networks in three lanes using Normalized AUC, which was computed using bins of 10 meters.

Table 5.5: Networks performance.

	Normalized AUC			Drones reaching the goal		
	Training lane	Lane 1	Lane 2	Training lane	Lane 1	Lane 2
Random	21.70	19.83	26.63	0%	1%	0%
CNN	92.79	78.94	37.74	89%	70%	26%
SCNN	99.34	98.51	99.5	98%	98%	98%
eSCNN	91.67	48.30	44.64	77%	28%	17%
qeSCNN	2.86	2.83	2.86	0%	0%	0%
pqeSCNN	38.20	37.61	37.23	0%	0%	0%

5.5 Results

5.5.1 Neural Networks Performance

The performance evaluation methodology explained in this section concentrates on two primary metrics: the Normalized Area Under the Curve and the overall count of drones that successfully reach their assigned objectives.

The Area Under the Curve (AUC) is a well-known measure used to evaluate classifier effectiveness. However, for this research, I have redefined the AUC to quantify the cumulative distance achieved by a specified number of drones. In this case, the AUC is calculated using the formula:

$$AUC = \sum_{rd} d \quad (5.2)$$

where rd represents the distance that d drones have successfully navigated. Essentially, this revised definition of AUC enables us to assess overall drone performance across different distances.

To address variations in bin sizes or segments of the environment, it is important to normalize the AUC by dividing it by the width of these bins. This ensures that performance evaluation remains consistent regardless of environmental setups or bin widths.

The success rate of drones in completing their tasks can be measured by the total number of drones that reach their goals. This metric indicates the proportion of deployed drones that successfully navigate and arrive at their designated destinations.

Fig. 5.4 illustrates the performance of the network in 10-meter bins across the three lanes, while Table 5.5 provides metrics such as Normalized AUC values and the percentage of drones that reach the 70-meter goal. In order to establish a baseline, I compared it against a random agent’s performance represented by the blue bars. Amongst the neural networks evaluated, it is observed that SCNN (green bars) outperforms others consistently reaching toward end locations for all three lanes with an accuracy rate of 98%. On the contrary, CNN (orange bars) achieves this milestone approximately at rates of 89%, 70%, and merely only around 17% respectively for Training lane, Lane 1, and Lane 2 thus depicting inferior temporal task handling capabilities highlighting SCNN’s generalization performances compared to artificial neural networks.

The architecture of the eSCNN (depicted by red bars) is similar to that of the SCNN, but it does not include biases. It achieves success rates of only 77%, 28%, and 17% in Training lane, Lane 1, and Lane 2, respectively. This difference highlights the importance of biases in enhancing the network’s ability to achieve its goals effectively. Although the performance of the eSCNN is generally lower compared to that of the CNN, both networks exhibit similar behaviours in terms of goal attainment in Training lane and Lane 2. However, a significant decrease can be observed for the third bin in Lane 1 which indicates difficulties faced by eSCNN when navigating through a series of obstacles due to lack of expressiveness caused by insufficient biases. This visual analysis aligns with numerical data presented in Table 5.5.

Both the qeSCNN (purple bars) and the pqeSCNN (brown bars) networks have undergone post-training quantization. However, they differ in terms of which layer is quantized. The performance of the network with a quantized last layer (qeSCNN) is noticeably worse compared to other models. It does not even reach the second bin in Fig. 5.4. On the other hand, the unquantized last layer of the pqeSCNN executed on an 8-core RISC-V-based cluster performs better, reaching up to the fourth bin in Training lane and the third bin in both Lane 1 and Lane 2.

The decreased efficiency of the qeSCNN can be attributed to the limited expressiveness of its final layer. With only 256 levels available to represent the Q-function, it faces challenges in handling complex tasks. On the other hand, the pqeSCNN takes advantage of a wider dynamic range, achieving similar accuracy as the SCNN and eSCNN in bins one and two (Fig. 5.4). However, performance begins to decline starting from bin three due to approximations introduced by post-training quantization.

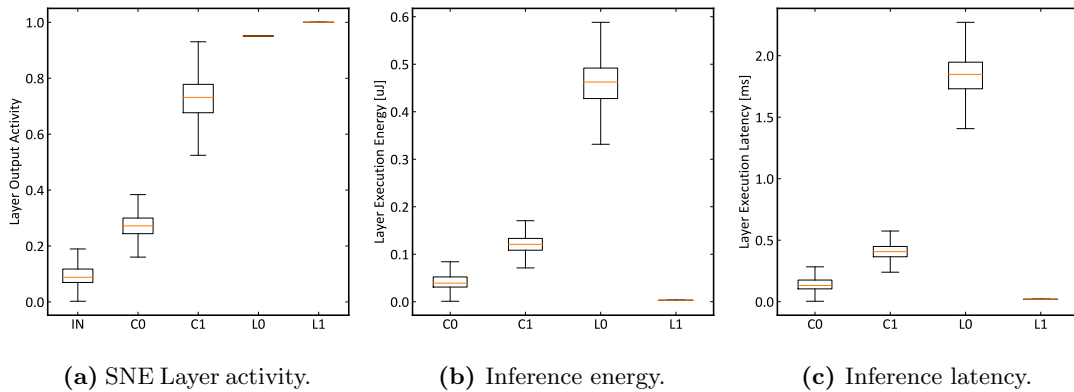


Figure 5.5: Convolutional layers are indicated as “Cn”, Linear layers are indicated as “Ln”, and “IN” indicates the network input activity from the event camera.

Table 5.6: Number of the total operation and inference energy for the proposed approach and the baseline CNN executed on a reference state-of-the-art hardware accelerator [1]. Estimates are based on the highest energy per inference reported by the author.

Network	Number of Operations	Inference Energy
CNN	23.2 MOP	$24.11 \mu J^a$ ($3.6 \mu J^b$)
qeSCNN	2.9 MOP	$0.62 \mu J$
pqeSCNN	2.9 MOP	$0.66 \mu J$

^a Energy per inference estimate based on the number of MAC executed on the reference hardware accelerator [1]

^b Energy per inference estimate on the reference hardware accelerator with energy cost scaled to the SNE technology.

Despite the fact that the random agent reached the sixth bin and the pqeSCNN only reached the fourth, it was found that the latter achieved a higher Normalized AUC as shown in Table 5.5. This is because of consistently poor performance by the random agent, while the pqeSCNN displayed declining performance starting from the third bin onwards (Fig. 5.4).

5.5.2 Spiking Neural Network Footprint

Energy and latency estimates for the quantized SNN running on digital embedded neuromorphic platforms are presented in this section. These estimates have been calculated based on the spike activity in the input and hidden layers of the network. The input activity estimates were derived from the complete dataset used to train the network in the RL task proposed in [84].

In Fig. 5.5, I can see three important metrics that are derived from the spike activity of the quantized SNN. In Fig. 5.5.a, it is evident that there is an increase in activity as

I go deeper into the network. This outcome was anticipated because, during training, there was no penalty for having high spike activity. As a result, redundant synaptic connections may have increased in the deeper layers [97]. Since the accelerator aims to achieve energy-to-activity proportionality, this spike activity has a direct impact on inference energy consumption and latency. Both these metrics scale accordingly with activity levels.

In Fig. 5.5.b, the energy consumption of the SNE platform is shown for each network layer. The convolutional layers have fewer synaptic operations due to their sparsity in the early layers. As a result, on average, these layers require less energy compared to the linear ones.

The energy consumption during inference is significantly impacted by the network's activity. The average total energy required for a complete inference is $0.62 \mu\text{J}/\text{inf}$, with variations depending on the input stream of events from the event camera within the field of view. For the dataset used, estimated energy consumption ranges from a minimum of $0.41 \mu\text{J}/\text{inf}$ to a maximum of $0.85 \mu\text{J}/\text{inf}$. It should be noted that reducing spike activity in hidden layers was not specifically trained during the RL training phase, as shown in Fig. 5.5.a. This variability in energy consumption during inference can be attributed to this lack of explicit training focus

Fig. 5.5.c illustrates the estimated inference latency for the suggested SNN running on the SNE platform. On average, the inference latency is 2.4 ms , with a minimum of 1.7 ms , and a maximum of 3.14 ms .

These findings indicate that the navigation and obstacle avoidance task can be successfully performed at a frame rate similar to or higher than the reported performance of state-of-the-art neural networks deployed on resource-constrained platforms in small drones, such as Dronet. The deployment of Dronet on an embedded System-on-Chip with a RISC-V cluster has been optimized to achieve an energy consumption of 15 mJ per inference.

In comparison to this approach, the combination of the proposed reinforcement learning strategy and efficient deployment on specialized neuromorphic hardware can result in a significant improvement in energy efficiency by more than three orders of magnitude.

To conduct an energy analysis between the CNN and SNN models, both workloads were broken down into fundamental operations, such as multiplication or addition. The findings are outlined in Table 5.6.

CNN operations are typically represented by MACs (multiply and accumulate) for computing output pixel values. On the other hand, SNN involves additions during the sparse weight accumulation on membrane potential and multiplications for implementing the exponential decay of membrane potential.

The results presented in Table 5.6 clearly illustrate the superior energy efficiency of executing SNN inference on dedicated hardware compared to CNN. The sparse nature of SNN allows for significantly fewer operations, resulting in lower energy consumption. Even when considering the energy consumed by a reference CNN accelerator [1] and scaling those numbers to the SNE technology, the energy cost remains substantially lower for SNE, approximately six times less.

5.6 Conclusions

This study aimed to tackle the issue of training Spiking Neural Networks for effective obstacle avoidance tasks, particularly on advanced neuromorphic hardware called SNE. The following are the main contributions and discoveries made in this research:

- I have created a robust and realistic training pipeline for networks involved in obstacle avoidance tasks. To achieve this, I utilized Unreal Engine 4 as an environment simulator, AirSim as a UAV simulator, and v2e to convert RGB frames from Unreal Engine 4 into Dynamic Vision Sensor frames.
- To overcome the obstacle avoidance task, I opted to train a Spiking Neural Network from scratch instead of converting an existing Artificial Neural Network. My approach involved adapting the Spatio-Temporal BackPropagation SNN training method to evaluate the membrane potential of the output neurons for action assessment at each step.
- To simulate the behaviour of SNNs in real neuromorphic hardware, a QuantLab plugin was developed for quantizing weights and membrane potential.
- A study was conducted to compare the performance of different SNN architectures, including SCNN, eSCNN, qeSCNN, and pqeSCNN, with a CNN trained in Reinforcement Learning. The results showed that SCNN had superior performance compared to the other architectures. It achieved a success rate of 98% in reaching the end of all three lanes, while CNN only achieved up to 89% success in its trained lane. Additionally, it was found that biases played a significant role in network performance as seen by a drop in performance for eCNN due to their absence.

-
- The energy consumption and latency of the SNN were assessed as if it was executed on specialized hardware. The evaluation showed that, on average, a complete inference required $9.73 \mu\text{J}/\text{inf}$ with an average latency of $33.35 \mu\text{s}$. It was observed that spike activity strongly influenced both energy and latency, although in this study there was no specific training to minimize activity levels.
 - I compared the energy consumption of both embedded SNNs (qeSCNN and pqeSCNN) to that of the CNN. The findings revealed that even without explicitly training for minimal spike activity, both SNNs utilized approximately 6 times less energy in comparison to the CNN.

Chapter 6

Conclusion

Within the field of Cyber-Physical Systems, this thesis explored two distinct applications of Spiking Neural Networks: Structural Health Monitoring and Obstacle Avoidance. The former focuses on classifying time series data for the purpose of identifying structural integrity, while the latter involves navigating dynamic environments to avoid obstacles.

Structural Health Monitoring (SHM): My research focused on the application of Long Short-Term SNNs in Structural Health Monitoring, comparing their performance to that of Artificial Neural Networks. The LSNN models proved to be highly effective in analyzing input sequences and successfully performing classification tasks. In fact, the best-performing LSNN achieved an MCC score greater than 0.88 when distinguishing between damaged and healthy bridge conditions. These results highlight the potential of SNNs in SHM applications and their ability to contribute to energy-efficient neuromorphic accelerators. Additionally, I developed an enhanced variant of LSNN specifically designed for microcontroller-based platforms, studying the impact of event-driven inputs along with energy efficiency considerations. My analysis revealed important trade-offs between energy consumption and data transfer costs, providing valuable insights into optimal configurations for real-world implementations.

Obstacle Avoidance with SNNs on Neuromorphic Hardware: I have developed a robust training pipeline for obstacle avoidance using Unreal Engine 4, AirSim, and v2e to simulate obstacle scenarios. Through reinforcement learning, I trained an SNN from scratch by introducing an adapted Spatio-Temporal BackPropagation method. I compared four different SNN architectures: SCNN, eSCNN, qeSCNN, and pqeSCNN, along with a CNN trained in Reinforcement Learning. The results showed that the performance of SCNN was superior to the other architectures with a success rate of 98% in reaching the end of all three lanes. This highlights the importance of biases since their

absence in eSCNN led to decreased performance. Furthermore, the evaluation of energy consumption and latency of SNNs when deployed on neuromorphic hardware revealed the potential for energy-efficient and low-latency neuromorphic inference.

Combined Insights: In summary, the research findings indicate that SNNs demonstrate versatility and promise in diverse applications such as SHM and decision-making tasks. Their energy efficiency, low latency, and high performance make them attractive options for deployment on specialized hardware platforms.

Bibliography

- [1] Yu-Hsin Chen, Tien-Ju Yang, Joel Emer, and Vivienne Sze. Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 9(2):292–308, 2019. doi: 10.1109/JETCAS.2019.2910232.
- [2] Wolfgang Maass. Networks of spiking neurons: the third generation of neural network models. *Neural networks*, 10(9):1659–1671, 1997.
- [3] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010.
- [4] Barry L Kalman and Stan C Kwasny. Why tanh: choosing a sigmoidal function. In *[Proceedings 1992] IJCNN International Joint Conference on Neural Networks*, volume 4, pages 578–581. IEEE, 1992.
- [5] George Kimeldorf and Grace Wahba. Some results on tchebycheffian spline functions. *Journal of mathematical analysis and applications*, 33(1):82–95, 1971.
- [6] Bernhard Schölkopf, Ralf Herbrich, and Alex J Smola. A generalized representer theorem. In *International conference on computational learning theory*, pages 416–426. Springer, 2001.
- [7] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.
- [8] Yann LeCun, LD Jackel, Leon Bottou, A Brunot, Corinna Cortes, JS Denker, Harris Drucker, and I Guyon. Ua m uller, e. s ackinger, p. simard, and v. vapnik. comparison of learning algorithms for handwritten digit recognition. In *Proceedings ICANN*, volume 95, pages 53–60, 1995.
- [9] Aston Zhang, Zachary C. Lipton, Mu Li, and Alexander J. Smola. *Dive into Deep Learning*. Cambridge University Press, 2023. <https://D2L.ai>.

-
- [10] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.
- [11] Ossama Abdel-Hamid, Abdel-rahman Mohamed, Hui Jiang, Li Deng, Gerald Penn, and Dong Yu. Convolutional neural networks for speech recognition. *IEEE/ACM Transactions on audio, speech, and language processing*, 22(10):1533–1545, 2014.
- [12] Nal Kalchbrenner, Edward Grefenstette, and Phil Blunsom. A convolutional neural network for modelling sentences. *arXiv preprint arXiv:1404.2188*, 2014.
- [13] Yann LeCun, Yoshua Bengio, et al. Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks*, 3361(10):1995, 1995.
- [14] Alex Graves, Marcus Liwicki, Santiago Fernández, Roman Bertolami, Horst Bunke, and Jürgen Schmidhuber. A novel connectionist system for unconstrained handwriting recognition. *IEEE transactions on pattern analysis and machine intelligence*, 31(5):855–868, 2008.
- [15] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. *Advances in neural information processing systems*, 27, 2014.
- [16] Zachary C Lipton, David C Kale, Charles Elkan, and Randall Wetzell. Learning to diagnose with lstm recurrent neural networks. *arXiv preprint arXiv:1511.03677*, 2015.
- [17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.
- [18] Boris T Polyak. Some methods of speeding up the convergence of iteration methods. *Ussr computational mathematics and mathematical physics*, 4(5):1–17, 1964.
- [19] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.
- [20] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, 12(7), 2011.
- [21] Tijmen Tieleman, Geoffrey Hinton, et al. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 4(2):26–31, 2012.

-
- [22] Sashank J Reddi, Satyen Kale, and Sanjiv Kumar. On the convergence of adam and beyond. *arXiv preprint arXiv:1904.09237*, 2019.
- [23] Manzil Zaheer, Sashank Reddi, Devendra Sachan, Satyen Kale, and Sanjiv Kumar. Adaptive methods for nonconvex optimization. *Advances in neural information processing systems*, 31, 2018.
- [24] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.
- [25] Laura Graesser and Wah Loon Keng. *Foundations of deep reinforcement learning*. Addison-Wesley Professional, 2019.
- [26] Maxim Lapan. *Deep Reinforcement Learning Hands-On: Apply modern RL methods to practical problems of chatbots, robotics, discrete optimization, web automation, and more*. Packt Publishing Ltd, 2020.
- [27] Gavin A Rummery and Mahesan Niranjan. *On-line Q-learning using connectionist systems*, volume 37. University of Cambridge, Department of Engineering Cambridge, UK, 1994.
- [28] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.
- [29] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.
- [30] Hado Hasselt. Double q-learning. *Advances in neural information processing systems*, 23, 2010.
- [31] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30, 2016.
- [32] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.
- [33] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*, 2015.

- [34] Alan L Hodgkin and Andrew F Huxley. A quantitative description of membrane current and its application to conduction and excitation in nerve. *The Journal of physiology*, 117(4):500, 1952.
- [35] Eugene M Izhikevich. Which model to use for cortical spiking neurons? *IEEE transactions on neural networks*, 15(5):1063–1070, 2004.
- [36] Louis Édouard Lapicque. Louis lapicque. *J. physiol*, 9:620–635, 1907.
- [37] Larry F Abbott. Lapicque’s introduction of the integrate-and-fire model neuron (1907). *Brain research bulletin*, 50(5-6):303–304, 1999.
- [38] Bruce W Knight. Dynamics of encoding in a population of neurons. *The Journal of general physiology*, 59(6):734–766, 1972.
- [39] Christof Koch. *Biophysics of computation: information processing in single neurons*. Oxford university press, 2004.
- [40] Guillaume Bellec, Franz Scherr, Anand Subramoney, Elias Hajek, Darjan Salaj, Robert Legenstein, and Wolfgang Maass. A solution to the learning dilemma for recurrent networks of spiking neurons. *Nature communications*, 11(1):3625, 2020.
- [41] Marek Miskowicz. Send-on-delta concept: An event-based data reporting strategy. *sensors*, 6(1):49–63, 2006.
- [42] Xiaochuan Guo, Xin Qi, and John G Harris. A time-to-first-spike cmos image sensor. *IEEE Sensors Journal*, 7(8):1165–1175, 2007.
- [43] Wenzhe Guo, Mohammed E Fouda, Ahmed M Eltawil, and Khaled Nabil Salama. Neural coding in spiking neural networks: A comparative study for robust neuromorphic systems. *Frontiers in Neuroscience*, 15:638474, 2021.
- [44] Sidi Yaya Arnaud Yarga, Jean Rouat, and Sean Wood. Efficient spike encoding algorithms for neuromorphic speech recognition. In *Proceedings of the International Conference on Neuromorphic Systems 2022*, pages 1–8, 2022.
- [45] Nuttapod Nuntalid, Kshitij Dhoble, and Nikola Kasabov. Eeg classification with bsa spike encoding algorithm and evolving probabilistic spiking neural network. In *Neural Information Processing: 18th International Conference, ICONIP 2011, Shanghai, China, November 13-17, 2011, Proceedings, Part I 18*, pages 451–460. Springer, 2011.
- [46] Benjamin Schrauwen and Jan Van Campenhout. Bsa, a fast and accurate spike train encoding scheme. In *Proceedings of the International Joint Conference on Neural Networks, 2003.*, volume 4, pages 2825–2830. IEEE, 2003.

- [47] Neelava Sengupta, Nathan Scott, and Nikola Kasabov. Framework for knowledge driven optimisation based data encoding for brain data modelling using spiking neural network architecture. In *Proceedings of the fifth international conference on fuzzy and neuro computing (fancco-2015)*, pages 109–118. Springer, 2015.
- [48] Balint Petro, Nikola Kasabov, and Rita M Kiss. Selection and optimization of temporal spike encoding methods for spiking neural networks. *IEEE transactions on neural networks and learning systems*, 31(2):358–370, 2019.
- [49] Filip Ponulak and Andrzej Kasinski. Introduction to spiking neural networks: Information processing, learning and applications. *Acta neurobiologiae experimentalis*, 71(4):409–433, 2011.
- [50] Yujie Wu, Lei Deng, Guoqi Li, Jun Zhu, and Luping Shi. Spatio-temporal back-propagation for training high-performance spiking neural networks. *Frontiers in neuroscience*, 12:331, 2018.
- [51] Sander M Bohte, Joost N Kok, and Johannes A La Poutré. Spikeprop: backpropagation for networks of spiking neurons. In *ESANN*, volume 48, pages 419–424. Bruges, 2000.
- [52] Sumit B Shrestha and Garrick Orchard. Slayer: Spike layer error reassignment in time. *Advances in neural information processing systems*, 31, 2018.
- [53] Daniel Rasmussen. NengoDL: Combining deep learning and neuromorphic modelling methods. *arXiv*, 1805.11144:1–22, 2018. URL <http://arxiv.org/abs/1805.11144>.
- [54] Kay Smarsly, Kosmas Dragos, and Jens Wiggenbrock. Machine learning techniques for structural health monitoring. In *Proceedings of the 8th European workshop on structural health monitoring (EWSHM 2016)*, Bilbao, Spain, pages 5–8, 2016.
- [55] A. Burrello, A. Marchioni, D. Brunelli, et al. Embedded streaming principal components analysis for network load reduction in structural health monitoring. *IEEE Internet of Things Journal*, 8(6):4433–4447, 2021. doi: 10.1109/jiot.2020.3027102.
- [56] Christos Riziotis, Nicola Testoni, Cristiano Aguzzi, et al. A sensor network with embedded data processing and data-to-cloud capabilities for vibration-based real-time shm. *Journal of Sensors*, 2018:2107679, 2018. doi: 10.1155/2018/2107679. URL <https://doi.org/10.1155/2018/2107679>.
- [57] TinyML Foundation. Tinyml. <https://www.tinymml.org>, 2021.
- [58] ST Microelectronics. Stm32 solutions for artificial neural networks. https://www.st.com/content/st_com/en/ecosystems/stm32-ann.html, 2021.

- [59] Matthias Meyer, Timo Ferei-Campagna, Akos Pasztor, et al. Event-triggered natural hazard monitoring with convolutional neural networks on the edge. In *Proceedings of the 18th International Conference on Information Processing in Sensor Networks*, Ipsn '19, page 73–84, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362849. doi: 10.1145/3302506.3310390. URL <https://doi.org/10.1145/3302506.3310390>.
- [60] Wolfgang Maass. Networks of spiking neurons: The third generation of neural network models. *Neural Networks*, 10(9):1659–1671, 1997. ISSN 0893-6080. doi: [https://doi.org/10.1016/S0893-6080\(97\)00011-7](https://doi.org/10.1016/S0893-6080(97)00011-7). URL <https://www.sciencedirect.com/science/article/pii/S0893608097000117>.
- [61] Giacomo Indiveri and Timothy Horiuchi. Frontiers in neuromorphic engineering. *Frontiers in Neuroscience*, 5:118, 2011. ISSN 1662-453x. doi: 10.3389/fnins.2011.00118. URL <https://www.frontiersin.org/article/10.3389/fnins.2011.00118>.
- [62] Kaushik Roy, Akhilesh Jaiswal, and Priyadarshini Panda. Towards spike-based machine intelligence with neuromorphic computing. *Nature*, 575(7784):607–617, 2019. doi: 10.1038/s41586-019-1677-2. URL <https://doi.org/10.1038/s41586-019-1677-2>.
- [63] Lei Deng, Yujie Wu, Xing Hu, et al. Rethinking the performance comparison between snns and anns. *Neural Networks*, 121:294–307, 2020. ISSN 0893-6080. doi: <https://doi.org/10.1016/j.neunet.2019.09.005>. URL <https://www.sciencedirect.com/science/article/pii/S0893608019302667>.
- [64] S. Lamba and R. Lamba. Spiking neural networks vs convolutional neural networks for supervised learning. In *2019 International Conference on Computing, Communication, and Intelligent Systems (ICCCIS)*, pages 15–19, 2019.
- [65] A. C. Neves, I. González, J. Leander, et al. Structural health monitoring of bridges: a model-free ann-based approach to damage detection. *Journal of Civil Structural Health Monitoring*, 7(5):689–702, 2017. doi: 10.1007/s13349-017-0252-5. URL <https://doi.org/10.1007/s13349-017-0252-5>.
- [66] Panagiotis Seventekidis, Dimitrios Giagopoulos, Alexandros Arailopoulos, et al. Structural health monitoring using deep learning with optimal finite element model generated data. *Mechanical Systems and Signal Processing*, 145, 2020.
- [67] Onur Avci, Osama Abdeljaber, Serkan Kiranyaz, and Daniel Inman. Structural damage detection in real time: implementation of 1d convolutional neural networks

- for shm applications. In *Structural Health Monitoring & Damage Detection, Volume 7*, pages 49–54. Springer, 2017.
- [68] Liman Yang, Chenyao Fu, Yunhua Li, et al. Survey and study on intelligent monitoring and health management for large civil structure. *International Journal of Intelligent Robotics and Applications*, 3(3):239–254, 2019. doi: 10.1007/s41315-019-00079-2. URL <https://doi.org/10.1007/s41315-019-00079-2>.
- [69] Lili Pang, Junxiu Liu, Jim Harkin, George Martin, Malachy McElholm, Aqib Javed, and Liam McDaid. Case study—spiking neural network hardware system for structural health monitoring. *Sensors*, 20(18):5126, 2020.
- [70] J. Madrenas, M. Zapata, D. Fernández, et al. Towards efficient and adaptive cyber physical spiking neural integrated systems. In *2020 27th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, pages 1–4, 2020. doi: 10.1109/icecs49266.2020.9294982.
- [71] Guillaume Bellec, Franz Scherr, Anand Subramoney, et al. A solution to the learning dilemma for recurrent networks of spiking neurons. *Nature Communications*, 11:1–15, 2020.
- [72] Evangelos Stomatias, Miguel Soto, Teresa Serrano-Gotarredona, et al. An event-driven classifier for spiking neural networks fed with synthetic or dynamic vision sensor data. *Frontiers in Neuroscience*, 11:350, 2017. ISSN 1662-453x. doi: 10.3389/fnins.2017.00350. URL <https://www.frontiersin.org/article/10.3389/fnins.2017.00350>.
- [73] Alessio Burrello, Alex Marchioni, Davide Brunelli, et al. Embedded streaming principal components analysis for network load reduction in structural health monitoring. *IEEE Internet of Things Journal*, 2020.
- [74] Raspberry Pi (Trading) Ltd. Raspberry Pi Compute Module 3+ datasheet. https://www.raspberrypi.org/documentation/hardware/computemodule/datasheets/rpi_DATA_CM3plus_1p0.pdf, 2019.
- [75] HiveMQ. HiveMQ documentation v4.5. <https://www.hivemq.com/docs/hivemq/4.5/user-guide/introduction.html>, 2019.
- [76] Abraham Bachrach. Skydio autonomy engine: Enabling the next generation of autonomous flight. In *2021 IEEE Hot Chips 33 Symposium (HCS)*, 2021. doi: 10.1109/HCS52781.2021.9567400.
- [77] Arianna Rubino, Can Livanelioglu, Ning Qiao, Melika Payvand, and Giacomo Indiveri. Ultra-low-power fdsoi neural circuits for extreme-edge neuromorphic intelligence, 2020.

- [78] Saber Moradi, Ning Qiao, Fabio Stefanini, and Giacomo Indiveri. A scalable multicore architecture with heterogeneous memory structures for dynamic neuromorphic asynchronous processors (dynaps). *IEEE transactions on biomedical circuits and systems*, 12(1):106–122, 2017.
- [79] Mike Davies, Narayan Srinivasa, Tsung-Han Lin, Gautham Chinya, Yongqiang Cao, Sri Harsha Choday, Georgios Dimou, Prasad Joshi, Nabil Imam, Shweta Jain, et al. Loihi: A neuromorphic manycore processor with on-chip learning. *IEEE Micro*, 38(1):82–99, 2018.
- [80] M. V. DeBole, B. Taba, A. Amir, F. Akopyan, A. Andreopoulos, W. P. Risk, J. Kusnitz, C. Ortega Otero, T. K. Nayak, R. Appuswamy, P. J. Carlson, A. S. Cassidy, P. Datta, S. K. Esser, G. J. Garreau, K. L. Holland, S. Lekuch, M. Mastro, J. McKinstry, C. di Nolfo, B. Paulovicks, J. Sawada, K. Schleupen, B. G. Shaw, J. L. Klamo, M. D. Flickner, J. V. Arthur, and D. S. Modha. Truenorth: Accelerating from zero to 64 million neurons in 10 years. *Computer*, 52(05):20–29, may 2019. ISSN 1558-0814. doi: 10.1109/MC.2019.2903009.
- [81] Charlotte Frenkel, Jean-Didier Legat, and David Bol. A 28-nm convolutional neuromorphic processor enabling online learning with spike-based retinas. In *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5, 2020. doi: 10.1109/ISCAS45731.2020.9180440.
- [82] B. V. Benjamin, P. Gao, E. McQuinn, S. Choudhary, A. R. Chandrasekaran, J. Bussat, R. Alvarez-Icaza, J. V. Arthur, P. A. Merolla, and K. Boahen. Neurogrid: A mixed-analog-digital multichip system for large-scale neural simulations. *Proceedings of the IEEE*, 102(5):699–716, 2014.
- [83] Francesco Barchi, Gianvito Urgese, Enrico Macii, and Andrea Acquaviva. An efficient mpi implementation for multi-core neuromorphic platforms. In *2017 New Generation of CAS (NGCAS)*, pages 273–276. IEEE, 2017.
- [84] Alfio Di Mauro, Arpan Suravi Prasad, Zhikai Huang, Matteo Spallanzani, Francesco Conti, and Luca Benini. SNE: an energy-proportional digital accelerator for sparse event-based convolutions. 2022. doi: 10.3929/ethz-b-000543342. Design, Automation and Test in Europe Conference (DATE 2022).
- [85] Christian Mayr, Sebastian Hoepfner, and Steve Furber. Spinnaker 2: A 10 million core processor system for brain simulation and machine learning. *arXiv preprint arXiv:1911.02385*, 2019.

- [86] Nikolaus Salvatore, Sami Mian, Collin Abidi, and Alan D George. A neuro-inspired approach to intelligent collision avoidance and navigation. In *2020 AIAA/IEEE 39th Digital Avionics Systems Conference (DASC)*, pages 1–9. IEEE, 2020.
- [87] Sungmin Hwang, Jeesoo Chang, Min-Hye Oh, Kyung Kyu Min, Taejin Jang, Kyungchul Park, Junsu Yu, Jong-Ho Lee, and Byung-Gook Park. Low-latency spiking neural networks using pre-charged membrane potential and delayed evaluation. *Frontiers in Neuroscience*, 15:629000, 2021.
- [88] Arnon Amir, Brian Taba, David Berg, Timothy Melano, Jeffrey McKinstry, Carmelo Di Nolfo, Tapan Nayak, Alexander Andreopoulos, Guillaume Garreau, Marcela Mendoza, et al. A low power, fully event-based gesture recognition system. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 7243–7252, 2017.
- [89] Mazdak Fatahi, Mahmood Ahmadi, Mahyar Shahsavari, Arash Ahmadi, and Philippe Devienne. evt_mnist: A spike based version of traditional mnist. *arXiv preprint arXiv:1604.06751*, 2016.
- [90] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. *Commun. ACM*, 60(6):84–90, May 2017. ISSN 0001-0782. doi: 10.1145/3065386.
- [91] Matteo Spallanzani, Gian Paolo Leonardi, and Luca Benini. Training quantised neural networks with ste variants: the additive noise annealing algorithm. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 470–479, 2022.
- [92] Angelo Garofalo, Manuele Rusci, Francesco Conti, Davide Rossi, and Luca Benini. PULP-NN: Accelerating Quantized Neural Networks on Parallel Ultra-Low-Power RISC-V Processors. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 378(2164):20190155, 2020. doi: 10.1098/rsta.2019.0155.
- [93] Shital Shah, Debadeepta Dey, Chris Lovett, and Ashish Kapoor. Airsim: High-fidelity visual and physical simulation for autonomous vehicles. In *Field and Service Robotics*, 2017. URL <https://arxiv.org/abs/1705.05065>.
- [94] Epic Games. Unreal engine. URL <https://www.unrealengine.com>.
- [95] Chia Yu Ho, Shau Yin Tseng, Chin Feng Lai, Ming Shi Wang, and Ching Ju Chen. A parameter sharing method for reinforcement learning model between airsim and uavs. In *2018 1st International Cognitive Cities Conference (IC3)*, pages 20–23. IEEE, 2018.

- [96] Yuhuang Hu, Shih-Chii Liu, and Tobi Delbruck. v2e: From video frames to realistic dvs events. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 1312–1321, 2021.
- [97] Torsten Hoefer, Dan Alistarh, Tal Ben-Nun, Nikoli Dryden, and Alexandra Peste. Sparsity in deep learning: Pruning and growth for efficient inference and training in neural networks. *CoRR*, abs/2102.00554, 2021. URL <https://arxiv.org/abs/2102.00554>.
- [98] Amirhossein Moallemi, Alessio Burrello, Davide Brunelli, et al. Model-based vs. data-driven approaches for anomaly detection in structural health monitoring: a case study. In *2021 IEEE International Instrumentation and Measurement Technology Conference (I2MTC)*, pages 1–6. Ieee, 2021.
- [99] Nicola Testoni, Federica Zonzini, Alessandro Marzani, et al. A tilt sensor node embedding a data-fusion algorithm for vibration-based shm. *Electronics*, 8(1), 2019. ISSN 2079-9292. doi: 10.3390/electronics8010045. URL <https://www.mdpi.com/2079-9292/8/1/45>.
- [100] Omitted for blind-review. 2020.
- [101] Maral Amir and Tony Givargis. Priority neuron: A resource-aware neural network for cyber-physical systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Pp:1–1, 07 2018. doi: 10.1109/tcad.2018.2857319.
- [102] Jonathan Goh, Sridhar Adepu, Yi Xiang Marcus Tan, et al. Anomaly detection in cyber physical systems using recurrent neural networks. pages 140–145, 01 2017. doi: 10.1109/hase.2017.36.
- [103] S. Yoginath, V. Tansakul, S. Chinthavali, et al. On the effectiveness of recurrent neural networks for live modeling of cyber-physical systems. In *2019 IEEE International Conference on Industrial Internet (ICII)*, pages 309–317, 2019. doi: 10.1109/icii.2019.00062.
- [104] Francesco Barchi, Luca Zanatta, Emanuele Parisi, et al. An automatic battery recharge and condition monitoring system for autonomous drones. In *2021 IEEE International Workshop on Metrology for Industry 4.0 IoT*, 2021.
- [105] C Arcadius Tokognon, Bin Gao, Gui Yun Tian, et al. Structural health monitoring framework based on internet of things: A survey. *IEEE Internet of Things Journal*, 4(3):619–635, 2017.

-
- [106] Guillaume Bellec, Franz Scherr, Anand Subramoney, et al. A solution to the learning dilemma for recurrent networks of spiking neurons. *Nature Communications*, 11(1):3625, 2020. doi: 10.1038/s41467-020-17236-y. URL <https://doi.org/10.1038/s41467-020-17236-y>.
- [107] National Instruments. Pxi systems. <https://www.ni.com/it-it/shop/pxi.html>, 2021.
- [108] M. V. DeBole, B. Taba, A. Amir, et al. Truenorth: Accelerating from zero to 64 million neurons in 10 years. *Computer*, 52(05):20–29, may 2019. ISSN 1558-0814. doi: 10.1109/mc.2019.2903009.
- [109] Johannes Schemmel, Johannes Fierens, and Karlheinz Meier. Wafer-scale integration of analog neural networks. In *2008 IEEE International Joint Conference on Neural Networks (IEEE World Congress on Computational Intelligence)*, pages 431–438. Ieee, 2008.
- [110] Joongheon Kim, Giuseppe Caire, and Andreas F Molisch. Quality-aware streaming and scheduling for device-to-device video delivery. *IEEE/ACM Transactions on Networking*, 24(4):2319–2331, 2015.
- [111] Francesco Barchi, Gianvito Urgese, Enrico Macii, et al. An efficient mpi implementation for multi-core neuromorphic platforms. In *2017 New Generation of CAS (NGCAS)*, pages 273–276. Ieee, 2017.