

# ENERGY-EFFICIENCY-OPTIMIZED CONVOLUTIONAL SPIKING NEURAL NETWORKS FOR PATIENT-SPECIFIC SEIZURE DETECTION

ABDUL MUNEEB

A THESIS SUBMITTED TO  
THE FACULTY OF GRADUATE STUDIES  
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS  
FOR THE DEGREE OF  
MASTER OF APPLIED SCIENCE

GRADUATE PROGRAM IN  
ELECTRICAL ENGINEERING AND COMPUTER SCIENCE  
YORK UNIVERSITY  
TORONTO, ONTARIO

DECEMBER 2024

© Abdul Muneeb, 2024

## ABSTRACT

This thesis presents the design, development, hardware implementation, and optimization of neuromorphic spiking convolutional neural networks (SNNs) for patient-specific seizure detection. Through leveraging inherent advantages of SNN in energy-efficient event-driven computing, we aim to achieve a low-power small-latency high-accuracy processing unit to perform seizure detection on multi-channel implantable brain-machine interfaces.

We first present the design and validation of spiking convolutional neural networks (SCNN) with dedicated energy-based and phase-based extracted features for cross-patient seizure detection. Evaluation on publicly available datasets demonstrates average sensitivities and specificities of 83.02% and 86.31%, respectively, with a false positive rate of 0.69/hr for cross-patient seizure detection. The SCNN, incorporating a 1-bit Integer-Net, maintains sensitivity comparable to a 32-bit CNN while improving memory efficiency by  $27\times$  and energy efficiency by 98.6%, achieving  $1.28\mu\text{J}/\text{classification}$ .

Furthermore, this thesis details the design and optimization of both regular and spiking CNNs for patient-specific seizure detection, achieving high performance metrics on microcontroller (STM32H747) and neuromorphic processor simulator (Intel Loihi). The work highlights the significance of customizing parameters such as quantization resolution and encoding schemes to fully harness SNN potentials. A custom-developed SCNN, optimized from scratch and implemented using HDL, demonstrates substantial improvements in sensitivity (92.2%) and specificity (97.3%), alongside significant gains in memory and computational efficiency.

This thesis contributes to the field of neurostimulation by providing an energy-efficient, high-performance approach to seizure detection, with potential implications for future research and



development in adaptive brain interfaces and neuromorphic computing systems.

## **ACKNOWLEDGEMENTS**

I would like to express my special thanks to Dr. Hossein Kassiri, my Masters supervisor, for his non-stopping and timeless guidance and support to complete this project. He introduced me to this project and generously shared his comprehensive knowledge with me. He continuously encouraged me throughout the project to remove any obstacle. I would like to thank him for patiently guiding me through all aspects of my academic and non-academic life.

I would also like to thank my defense committee and supervisor committee, Dr. Amirali Amirsoleimani, and Dr. Kohitij Kar for reading my thesis and giving me valuable feedback to improve this thesis.

I would like to thank my colleagues and friends Mansour Taghadosi, Alireza Dabbaghian, Tayebbeh Yousefi, Mina Sayedi, Rojin Salahi, Nauman Bashir, and Hydar Isa at York University.

Finally, I want to express my deepest gratitude to my beloved parents, my elder brother Abdul Munim and my siblings for always providing me with continuous encouragement and motivation throughout the years and through the process of researching and writing this thesis. Your belief in me has been a constant source of strength, and I am forever thankful.

# Table of Contents

|                                                             |            |
|-------------------------------------------------------------|------------|
| <b>Abstract</b>                                             | <b>ii</b>  |
| <b>Acknowledgements</b>                                     | <b>iv</b>  |
| <b>List of Figures</b>                                      | <b>xi</b>  |
| <b>List of Tables</b>                                       | <b>xvi</b> |
| <b>1 Introduction</b>                                       | <b>1</b>   |
| 1.1 Objective . . . . .                                     | 1          |
| 1.2 Performance metrics . . . . .                           | 7          |
| 1.2.1 Algorithm performance metrics . . . . .               | 7          |
| 1.2.2 Hardware implementation performance metrics . . . . . | 9          |
| 1.2.3 Datasets . . . . .                                    | 10         |
| 1.3 Importance of individualization . . . . .               | 11         |

|       |                                                                      |    |
|-------|----------------------------------------------------------------------|----|
| 1.3.1 | Need for a data-driven approach . . . . .                            | 11 |
| 1.3.2 | Channel selection . . . . .                                          | 14 |
| 1.3.3 | Feature Extraction . . . . .                                         | 15 |
| 1.3.4 | Feature Selection . . . . .                                          | 17 |
| 1.4   | Patient-specific classification . . . . .                            | 18 |
| 1.4.1 | Classification algorithms and their on-chip implementation . . . . . | 18 |
| 1.4.2 | Support Vector Machine (SVM) . . . . .                               | 19 |
| 1.4.3 | Decision Trees . . . . .                                             | 22 |
| 1.4.4 | Deep Neural Networks (DNN) . . . . .                                 | 24 |
| 1.5   | Fundamentals of SNNs . . . . .                                       | 27 |
| 1.5.1 | Building blocks . . . . .                                            | 27 |
| 1.5.2 | Principle of Operation . . . . .                                     | 28 |
| 1.5.3 | Temporal dynamics . . . . .                                          | 30 |
| 1.5.4 | Advantages for implantable neural signal processing . . . . .        | 30 |
| 1.6   | Thesis objectives . . . . .                                          | 32 |
| 1.6.1 | Context and relevance . . . . .                                      | 33 |
| 1.6.2 | Scope of the research . . . . .                                      | 33 |
| 1.7   | Thesis organization . . . . .                                        | 34 |
| 1.8   | Thesis contributions . . . . .                                       | 35 |

|          |                                                                                                          |           |
|----------|----------------------------------------------------------------------------------------------------------|-----------|
| <b>2</b> | <b>A 9.5ms-Latency 6.2<math>\mu</math>J/Inference Spiking CNN for Patient-Specific Seizure Detection</b> | <b>36</b> |
| 2.1      | Input layer: raw EEG vs extracted features . . . . .                                                     | 37        |
| 2.2      | SCNN Development: Organic vs Shadow Training . . . . .                                                   | 39        |
| 2.3      | CNN Architecture Design . . . . .                                                                        | 40        |
| 2.3.1    | Neural architecture search . . . . .                                                                     | 40        |
| 2.3.2    | Input and preprocessing . . . . .                                                                        | 41        |
| 2.3.3    | CNN layers . . . . .                                                                                     | 42        |
| 2.4      | Train/test data division . . . . .                                                                       | 42        |
| 2.5      | CNN to SCNN conversion . . . . .                                                                         | 43        |
| 2.6      | CNN/SCNN Evaluation . . . . .                                                                            | 45        |
| 2.7      | Hardware implementation . . . . .                                                                        | 48        |
| <b>3</b> | <b>Customized Development and Hardware Optimization of a Fully Spiking SNN</b>                           | <b>54</b> |
| 3.1      | Introduction . . . . .                                                                                   | 54        |
| 3.2      | Motivation for a customizable SNN shadow training . . . . .                                              | 56        |
| 3.3      | Model development and evaluation . . . . .                                                               | 61        |
| 3.3.1    | CNN design and conversion methodology . . . . .                                                          | 62        |
| 3.3.2    | Temporal encoding and resolution . . . . .                                                               | 64        |
| 3.3.3    | Neuron model . . . . .                                                                                   | 65        |

|          |                                                          |           |
|----------|----------------------------------------------------------|-----------|
| 3.3.4    | Parameters quantization . . . . .                        | 66        |
| 3.4      | Model testing . . . . .                                  | 67        |
| <b>4</b> | <b>Digital back-end hardware implementation of FSCNN</b> | <b>70</b> |
| 4.1      | Introduction . . . . .                                   | 70        |
| 4.2      | Block-level Implementation . . . . .                     | 71        |
| 4.2.1    | Input register . . . . .                                 | 71        |
| 4.2.2    | Spiking convolution layer . . . . .                      | 73        |
| 4.2.3    | Neuron model . . . . .                                   | 73        |
| 4.2.4    | ReLU function . . . . .                                  | 75        |
| 4.2.5    | Max pooling layer . . . . .                              | 76        |
| 4.2.6    | System Integration . . . . .                             | 76        |
| 4.2.7    | Hardware Implementation Evaluation . . . . .             | 79        |
| 4.3      | Simulation results . . . . .                             | 80        |
| 4.4      | Synthesis results . . . . .                              | 83        |
| <b>5</b> | <b>Conclusion and future works</b>                       | <b>85</b> |
| 5.1      | Conclusion . . . . .                                     | 85        |
| 5.2      | Future directions . . . . .                              | 87        |
|          | <b>References</b>                                        | <b>89</b> |

|                                                                    |            |
|--------------------------------------------------------------------|------------|
| <b>APPENDICES</b>                                                  | <b>101</b> |
| <b>A CNN model development in TensorFlow Keras</b>                 | <b>102</b> |
| A.1 Data loader and CNN model . . . . .                            | 102        |
| A.2 Python Jupyter notebook code for CNN . . . . .                 | 102        |
| <b>B Customized Partial Spiking CNN model development in Numpy</b> | <b>114</b> |
| B.1 Partial Spiking CNN model . . . . .                            | 114        |
| B.2 Python Jupyter notebook code for PSCNN . . . . .               | 114        |
| <b>C Customized Fully Spiking CNN model development in Numpy</b>   | <b>140</b> |
| C.1 Fully Spiking CNN model . . . . .                              | 140        |
| C.2 Python Jupyter notebook code for FSCNN . . . . .               | 140        |
| <b>D Hardware implementation of FSCNN in Verilog</b>               | <b>162</b> |
| D.1 Verilog HDL implementation . . . . .                           | 162        |
| D.2 Verilog code for FSCNN . . . . .                               | 162        |
| D.3 Classifier . . . . .                                           | 163        |
| D.4 Data input register . . . . .                                  | 167        |
| D.5 Spiking convolution layer . . . . .                            | 171        |
| D.6 Max pooling layer . . . . .                                    | 176        |

|     |                                        |     |
|-----|----------------------------------------|-----|
| D.7 | Add block for sign additions . . . . . | 183 |
| D.8 | Add block for sign adder . . . . .     | 184 |
| D.9 | 4-bit adder . . . . .                  | 185 |



# List of Figures

|     |                                                                                                                                                                                              |    |
|-----|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----|
| 1.1 | (a) Widespread and (b) Localized seizure activity in brain. . . . .                                                                                                                          | 2  |
| 1.2 | Block diagram of a generic (a) open-loop and (b) closed-loop neurostimulator system. . . . .                                                                                                 | 3  |
| 1.3 | Block diagram of a closed-loop neurostimulator system monitoring and classifying EEG signal for epilepsy seizure detection. . . . .                                                          | 5  |
| 1.4 | Examples of dual-channel EEG recordings showing ictal, preictal, and interictal periods, highlighting the significant differences in seizure manifestations across patients. . . . .         | 6  |
| 1.5 | Summary of commonly used parameters for seizure detection algorithm performance evaluation. . . . .                                                                                          | 8  |
| 1.6 | Four main stages of signal processing conducted on multi-channel implantable closed-loop neurostimulators for seizure detection, and some of the more common options for each stage. . . . . | 12 |
| 1.7 | Examples of diverse seizure manifestation in EEG recordings for different patients. Data from [22] . . . . .                                                                                 | 13 |

|      |                                                                                                                                                                                                                                                                                                                                                                                                                            |    |
|------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----|
| 1.8  | Examples of different seizure-like artifacts that could appear on EEG recordings.<br>Data from [22]                                                                                                                                                                                                                                                                                                                        | 13 |
| 1.9  | (a) Feature extraction conducted on a subset of channels identified as critical for a specific patient during offline training. (b) Comparison showing better differentiation between seizure and normal states using patientspecific channels versus all channels or a fixed subset [23]. (c) Optimization of spectral bands for phase-related and energy-related feature extraction tailored to individual patients [24] | 16 |
| 1.10 | Example of thresholding as a simple, efficient, and straightforward, yet not patient-adaptive, classifier used for seizure detection [34].                                                                                                                                                                                                                                                                                 | 19 |
| 1.11 | Conceptual illustration of popular ML-based classifiers used for seizure detection, including (a) linear, (b) nonlinear, and (c) coarse-fine SVMs, random forest classifiers with (d) bagging, and (e) boosting techniques, and deep neural networks (f) with and (g) without dedicated feature extraction                                                                                                                 | 21 |
| 1.12 | Qualitative comparison of most common classifiers used for seizure detection based on the most important performance metrics.                                                                                                                                                                                                                                                                                              | 25 |
| 1.13 | (a) Simplified biological model of a neuron and its major sections and their associated functions. (b) Operation principle of an LIF neuron model and (c) sample input and outputs.                                                                                                                                                                                                                                        | 28 |
| 1.14 | (a) ANN neuron model (b) Multiplier-less SNN neuron model.                                                                                                                                                                                                                                                                                                                                                                 | 31 |
| 1.15 | Event driven sparse SNN.                                                                                                                                                                                                                                                                                                                                                                                                   | 32 |

|      |                                                                                                                                                                                             |    |
|------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----|
| 2.1  | Top-level block diagram of the CNN-based seizure detection algorithm with feature extraction and image creation blocks. . . . .                                                             | 38 |
| 2.2  | The developed CNN architecture utilizing 1D time- and spatial convolutions, with all kernels and layer dimension shown. . . . .                                                             | 40 |
| 2.3  | Data division for training, cross validation, and testing of the presented SCNN. .                                                                                                          | 43 |
| 2.4  | A simplified block diagram of the employed shadow training, showing steps taken for CNN-to-SCNN conversion and spike encoding. . . . .                                                      | 44 |
| 2.5  | Correlation between the CNN activations and SCNN spike rates for each layer and across 32 Samples (1-batch). . . . .                                                                        | 46 |
| 2.6  | Definition of event-based and vector-based sensitivity in seizure detection. . . .                                                                                                          | 47 |
| 2.7  | Classification performance of the developed CNN and SCNN in terms of (a) sensitivity and (b) specificity. . . . .                                                                           | 47 |
| 2.8  | Subject-wise seizure detection performance of the developed SCNN. . . . .                                                                                                                   | 48 |
| 2.9  | Hardware Implementation Pipeline for (a) CNN on Micro-controller and (b) Spiking CNN on Loihi. . . . .                                                                                      | 49 |
| 2.10 | Error rate versus the simulation time: SCNN vs CNN. . . . .                                                                                                                                 | 50 |
| 2.11 | SCNN's sensitivity, specificity, energy consumption, and Inference Time as a function of time- steps, showing the optimal overall performance of the selected number of time-steps. . . . . | 51 |

|     |                                                                                                                                                                                                                                                                                                                                           |    |
|-----|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----|
| 3.1 | (a) A partially spiking neural network (PSNN) compared to (b) and fully spiking neural network (FSNN) realized through pre-SNN encoding. (c) An FSNN interfaced with asynchronous recording circuit. . . . .                                                                                                                              | 58 |
| 3.2 | High-level visualization of the shadow training process for SNN development and the critical parameters that need to be decided to ensure a hardware-optimized implementation. . . . .                                                                                                                                                    | 60 |
| 3.3 | Top-level block diagram of the system architecture, illustrating the characteristics of the input signals and encoded spiking data. The diagram also details the architecture of the spiking convolutional neural network, which utilizes 1D temporal and spatial kernels. All kernel and layer dimensions are shown for clarity. . . . . | 62 |
| 3.4 | Seizure detection sensitivity, specificity, and computational cost as a function of the SNN's temporal resolution. . . . .                                                                                                                                                                                                                | 65 |
| 3.5 | Seizure detection sensitivity, SNN model size, and power consumption as a function of quantization resolution used for weights and bias values. . . . .                                                                                                                                                                                   | 67 |
| 3.6 | Number of Post synaptic spikes at the output of each layer of the network in different situations. . . . .                                                                                                                                                                                                                                | 68 |
| 3.7 | Subject-wise seizure detection performance of the custom-developed SCNN. . .                                                                                                                                                                                                                                                              | 69 |
| 4.1 | Block level illustration of processing in SCNN layers. . . . .                                                                                                                                                                                                                                                                            | 71 |
| 4.2 | Input to a system that pass from the serializer and then stored into the input shift register. . . . .                                                                                                                                                                                                                                    | 72 |

|      |                                                                                       |    |
|------|---------------------------------------------------------------------------------------|----|
| 4.3  | Spiking convolution layer composed of select multiplexers and sign adder block.       | 74 |
| 4.4  | Digital blocks of integrate and fire neuron model. . . . .                            | 75 |
| 4.5  | Implementation of the ReLU function on hardware . . . . .                             | 76 |
| 4.6  | Hardware implementation of the spiking max-pooling. . . . .                           | 77 |
| 4.7  | Integration of the fundamental layers in hardware. . . . .                            | 78 |
| 4.8  | System Integration of the spiking hardware for the proposed SCNN architecture.        | 78 |
| 4.9  | System Integration of the spiking hardware . . . . .                                  | 79 |
| 4.10 | Simulation results for data input register. . . . .                                   | 80 |
| 4.11 | Simulation results for seizure activity detection at first spiking convolution layer. | 81 |
| 4.12 | Simulation results for max pooling layer of SCNN. . . . .                             | 82 |
| 4.13 | Simulation results for the input of second convolution layer. . . . .                 | 82 |

# List of Tables

|     |                                                                                    |    |
|-----|------------------------------------------------------------------------------------|----|
| 2.1 | CNN vs SCNN Performance Comparison . . . . .                                       | 52 |
| 2.2 | Comparison with the state-of-the-art patient-specific seizure detection algorithms | 52 |
| 3.1 | Constraints of SNN Development Approaches/Examples . . . . .                       | 61 |
| 4.1 | SCNN Parameters and Synthesis Results . . . . .                                    | 83 |
| 4.2 | Comparison with the state-of-the-art patient-specific seizure detection algorithms | 84 |

# Chapter 1

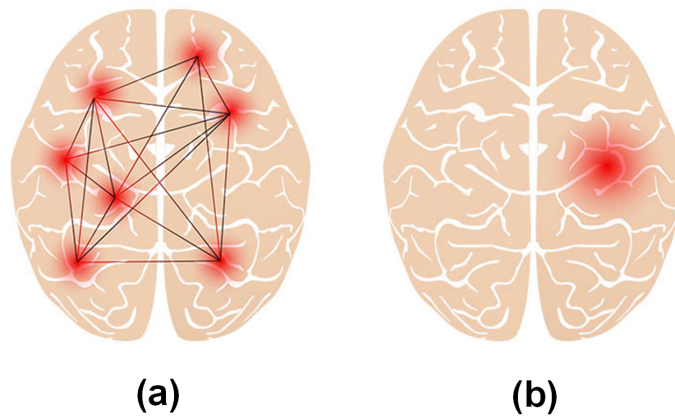
## Introduction

### 1.1 Objective

Epilepsy is a chronic neurological disorder characterized by recurrent and unpredictable seizures, affecting about one in every hundred individuals [1]-[3], making it the fourth most common neurological disorder globally [4]. It is typically characterized by an enduring pre-disposition to generate epileptic seizures and by the neurobiological, psychological, and social consequences [5].

Seizures, the primary symptom of epilepsy, are defined as periodic and unpredictable bursts of sudden uncontrolled electrical activity between neurons that cause temporary abnormalities in muscle movements, behaviors, sensations or states of awareness [6]. They can last from a few seconds to several minutes and may occur sporadically or several times a day. The types of seizures include convulsive seizures, which are noticeable due to visible muscle jerking, and

absence seizures, which have no apparent symptoms [7]. Epilepsy is also classified into focal and generalized types, based on whether the brain involvement is localized or widespread as shown in Figure 1.1.



**Figure 1.1: (a) Widespread and (b) Localized seizure activity in brain.**

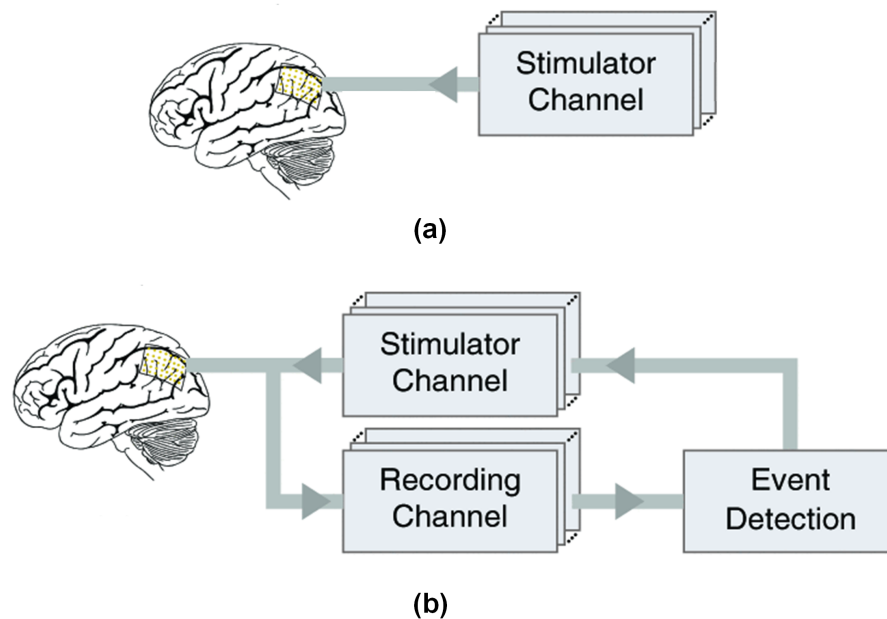
Although nearly 70% of patients find relief through pharmacological treatments, the rest continue to suffer from symptoms such as anxiety, depression, physical injury, and an increased risk of sudden death. For these patients, particularly those who cannot undergo resection surgery, neurostimulation offers a viable alternative [8]. These stimulation devices represent a significant advancement in the management of epilepsy, offering hope for patients who have not found relief through medication alone. Neurostimulators work by delivering electrical impulses to specific areas of the brain cells in order to modulate neural activity [10].

Stimulation effectiveness and safety heavily depend on its timing, which has been shown to be best if done before or shortly after the seizure onset [11]. Open-loop stimulation shown in Figure 1.2(a), which delivers stimulation at preset intervals without considering the timing of



seizures, is generally unsafe and inefficient. This kind of blind stimulation can lead to excessive periodic stimulations, causing short battery life and, more importantly, cell damage due to overheating. In contrast, closed-loop stimulation shown in Figure 1.2(b) provides a better alternative by triggering stimulation only upon the detection of a seizure event. This approach not only enhances efficacy by ensuring timely intervention but also avoids unnecessary stimulations, thereby preserving battery life and minimizing the risk of cell damage.

Closed-loop stimulation, operates by continuously monitoring brain activity and delivering stimulation precisely when a seizure is detected. This method's advantage lies in its ability to adapt to the dynamic nature of seizures, providing intervention exactly when needed, rather than relying on a fixed schedule. This precise timing is crucial for both the effectiveness of the treatment and the safety of the patient, as it minimizes the risks associated with overstimulation.

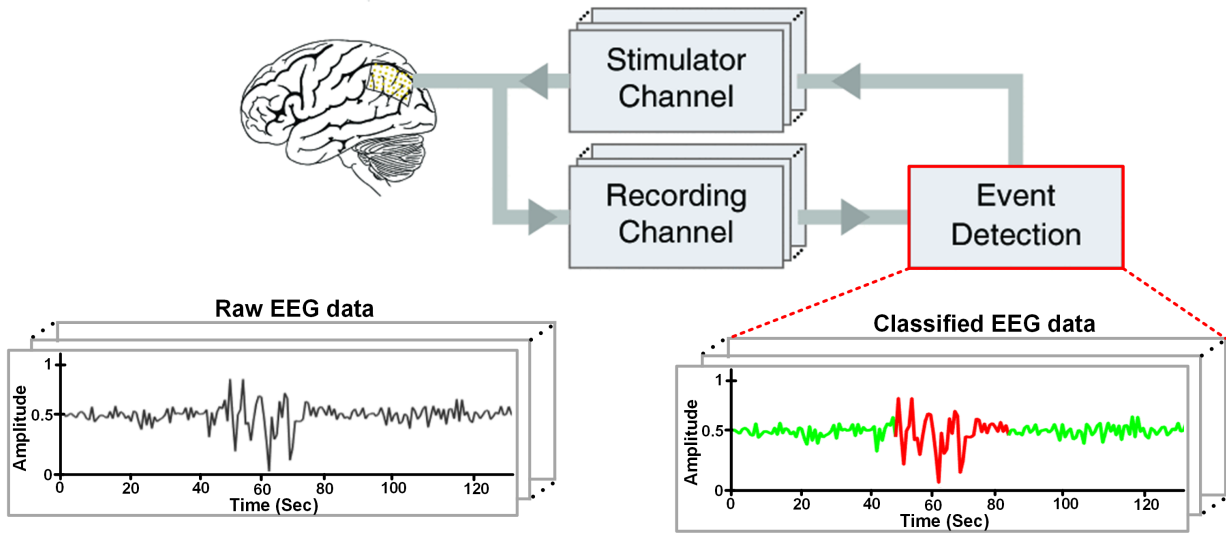


**Figure 1.2: Block diagram of a generic (a) open-loop and (b) closed-loop neurostimulator system.**

To record the electrical activity of the brain, different variations of Electroencephalography (EEG) are used clinically, including the noninvasive scalp EEG and invasive intracranial EEG (iEEG). Scalp EEG involves placing electrodes on the scalp to detect and measure the electrical impulses generated by neurons. These impulses are then amplified and recorded, providing a visual representation of brain wave patterns. The dynamic nature of seizures necessitates a recording method with millisecond-level temporal resolution, making EEG essential. EEG can capture rapid changes in brain activity, revealing distinctive patterns during seizure events. These patterns include preictal periods (preceding seizures), ictal episodes (from onset to end of a seizure), and interictal epochs (intervals between seizures). Moreover, multi-channel EEG is essential because seizures often originate from large neuronal networks across extensive brain areas. This comprehensive monitoring allows for accurate detection of seizure events. This understanding has driven the research community to develop implantable and wearable closed-loop stimulators capable of long-term, automated EEG monitoring and responsive stimulation, as shown in Figure 1.3.

These devices typically integrate numerous sensing and stimulation channels, alongside a processing unit responsible for analyzing the recorded data for seizure event detection and triggering responsive stimulation upon detection. Such systems, represent significant advancements in epilepsy treatment technology. Key studies and developments in this field include [12]-[15], showcasing the progress and potential of closed-loop neurostimulation in improving the lives of epilepsy patients.

Given the variability and complexity of seizures can present a wide variety of symptoms, ranging from convulsive episodes with muscle contractions to subtle changes in behavior or consciousness. This diversity in seizure types complicates the development of effective detection

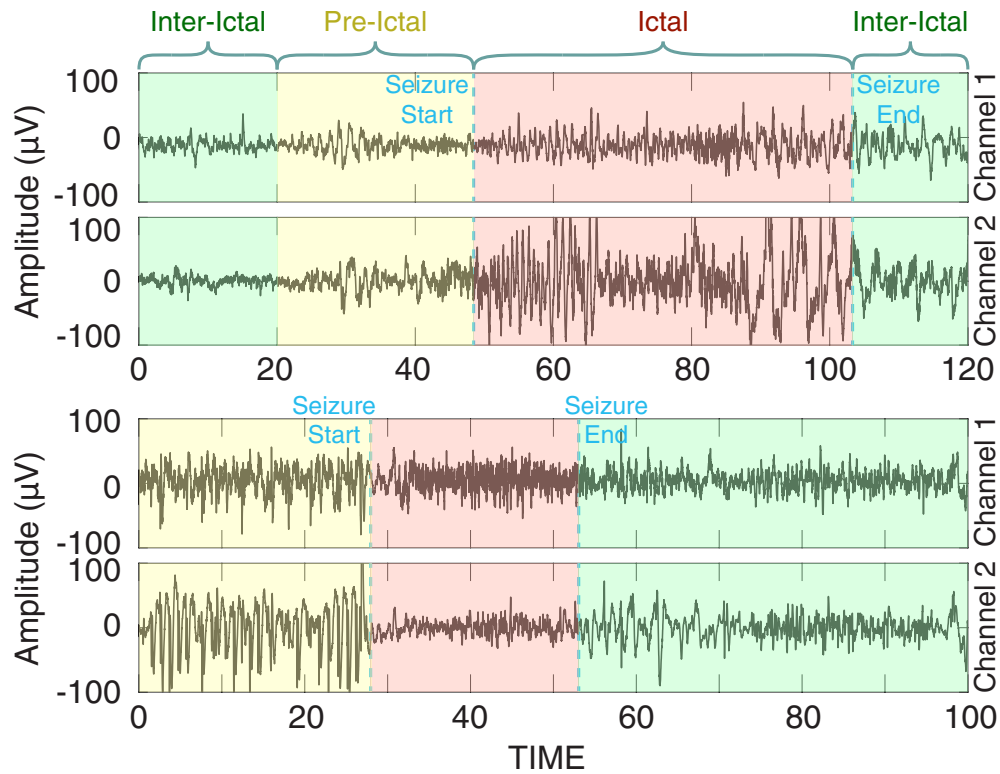


**Figure 1.3: Block diagram of a closed-loop neurostimulator system monitoring and classifying EEG signal for epilepsy seizure detection.**

methods, as each type of seizure has unique characteristics and manifestations. For instance, generalized tonic-clonic seizures involve a sudden loss of consciousness and violent muscle contractions, making them relatively easier to identify through physical and EEG monitoring. In contrast, focal seizures, which originate in a specific area of the brain, can manifest with much more subtle signs, such as slight twitching, sensory changes, or brief lapses in awareness. Absence seizures, primarily seen in children, are characterized by a brief and sudden lapse of consciousness, which might go unnoticed without careful observation or EEG analysis.

Moreover, the diversity of seizure patterns from subject to subject and even within the same subject over time adds further complexity. As shown in Figure 1.4, seizure manifestations can be significantly different, underscoring the complexity of seizure detection, a pivotal challenge in epilepsy research. Each individual with epilepsy may exhibit a unique combination of seizure types, frequencies, and triggers. This variability necessitates highly personalized detection ap-

proaches. The challenge in detecting seizures lies in the need for systems to accurately and reliably identify diverse manifestations in real time EEG. This requires development of advanced algorithms capable of distinguishing between normal brain activity and the subtle, often variable signs of a seizure. To address these challenges, sophisticated seizure detection algorithms are required that leverage machine learning and artificial intelligence. These algorithms can be trained on large datasets of EEG recordings to recognize patterns associated with different types of seizures.



**Figure 1.4: Examples of dual-channel EEG recordings showing ictal, preictal, and interictal periods, highlighting the significant differences in seizure manifestations across patients.**

In recent years, the utilization of machine learning (ML) techniques in seizure detection has shown remarkable potential in enhancing the accuracy and efficiency of epilepsy manage-

ment. ML-based algorithms have significantly advanced the field by enabling the development of patient-specific, and cross-patient approaches that leverage the distinctive characteristics of electroencephalography (EEG) signals.

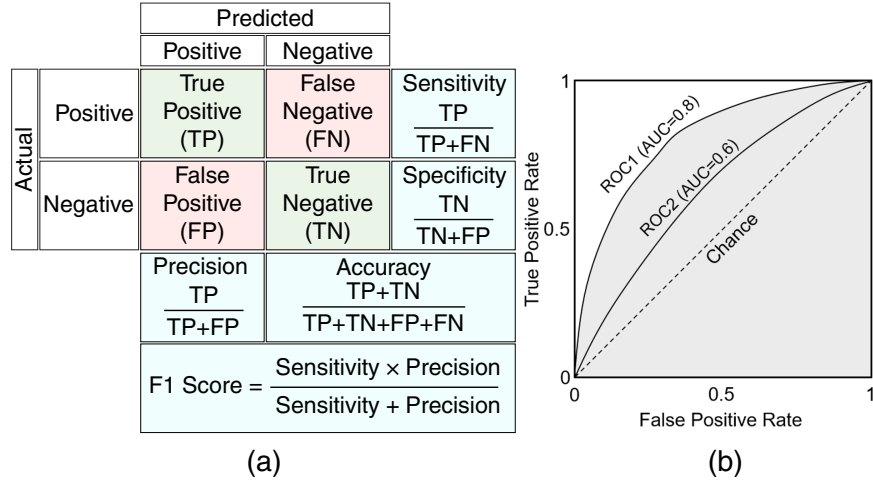
## **1.2 Performance metrics**

Assessing the performance of seizure detection algorithms involves a range of metrics, encompassing both algorithm and hardware considerations.

### **1.2.1 Algorithm performance metrics**

Figure 1.5(a) shows a summary of commonly-used parameters for algorithm performance evaluation, where TP (True Positive) indicates correctly identified seizure events, TN (True Negative) represents correctly identified non-seizure events, FP (False Positive) refers to non-seizure events incorrectly identified as seizures, and FN (False Negative) signifies seizure events that are missed and incorrectly identified as nonseizures.

The most common metrics, as shown in Figure 1.5 are sensitivity, specificity, precision, accuracy, and F1 score. In the context of epilepsy monitoring and responsive stimulation, a FN is considered worse than a FP due to more serious consequences of missing a seizure than side effects of an unnecessary stimulation, so the aim is generally to achieve a 100% sensitivity while maximizing specificity. A point to note about specificity is that Considering the significant class imbalance between seizure and non-seizure events/epochs, even for an algorithm that performs



**Figure 1.5: Summary of commonly used parameters for seizure detection algorithm performance evaluation.**

poorly and has a considerable number of FPs, TN is typically orders of magnitude higher than FP, leading to specificity remaining a relatively high value, which could be misleading.

As such, False Positive Rate (FPR), which is another way to represent the performance from this perspective, is more popular among recent reports as it provides a more accurate picture of reality than specificity. It is defined in several different ways, which include, but not limited to, the number of FPs per hour, or the proportion of FPs among all actual non-seizure instances. We will further discuss how FPR is also not a perfect metric and our suggested substitute in Section. Figure 1.5(b) shows area under the receiver operating characteristic curve (AUC-ROC), which plots the true positive rate (sensitivity) against the false positive rate (here, defined as 1-specificity) at various threshold settings. AUC-ROC provides an aggregate measure of a model's ability to discriminate between positive and negative instances across all possible threshold values. It is preferred when comparing models or when the threshold for decision-making is not predetermined, as it provides a comprehensive evaluation of the model's performance across all

possible operating points.

In addition to the above, latency plays a critical role in responsive stimulation systems, emphasizing the importance of timely intervention during seizure events. Latency could be due to the required data for processing or due to the algorithm and its hardware implementation. Data processing latency (or input-related latency) is the inherent delay caused by the need to process a specific duration of EEG signals (e.g., 2 seconds) before detection can occur, irrespective of the actual computation speed. Algorithmic latency refers to the time required by the algorithm to process input data and generate an output, which is influenced by the computational complexity of the algorithm, as well as hardware factors such as hardware architecture (e.g., conventional von Neumann or neuromorphic), clock frequency, and memory access times.

### **1.2.2 Hardware implementation performance metrics**

Developing seizure detection algorithms without considering their hardware implementation limits their practical relevance and diminishes their significance. Among hardware-related metrics, energy efficiency is perhaps the most crucial one, considering that the algorithm should be implemented on an implantable device that operates with stringent power constraints. Minimizing energy consumption per classification extends the operational lifespan of the device and reduces the thermal footprint, which is essential for patient safety and comfort. Computational resource utilization is the next metric, which focuses on the efficiency of the algorithm's hardware implementation. Techniques such as circuit sharing, pipelining, parallel processing, and optimized logic design can significantly reduce the required circuit area and the number of computational logic elements.

Scalability is another critical metric, ensuring that the algorithm maintains or improves efficiency and accuracy as the number of EEG channels increases. The key challenge is to ensure that resource and power consumption scale as slowly as possible, ideally sub-linearly relative to the channel count. This allows the system to handle larger datasets and more electrodes, thus improving performance without disproportionate increases in cost. Finally, in case of using a data-driven algorithm (discussed in next Section), model size also plays a role, as compact models reduce the memory footprint, as well as energy required for memory access, enhancing real-time processing capabilities within the constraints of implantable devices.

### **1.2.3 Datasets**

The development of patient-specific seizure detection typically begins with software-based algorithm development in Python or MATLAB. Initial testing on pre-recorded labeled datasets allows for performance evaluation and refinement. The algorithm is then implemented on a programmable hardware platform (e.g., an FPGA) for extensive testing, including validation on the same datasets and/or animal models of epilepsy, to ensure robustness before ASIC implementation. This highlights the importance of access to diverse and reliable datasets for algorithm development and evaluation. Table II summarizes some of the most popular publicly available datasets used by the community, highlighting the significant class imbalance between seizure and non-seizure data, which poses challenges for training and testing.

For example, the CHB-MIT dataset has only about 3 hours of seizure data, which is less than 0.5% of the total 940 hours of recordings, assuming a generous average seizure duration of 60 seconds. Variability in recording quality, signal type, pre-processing methods, and epilepsy



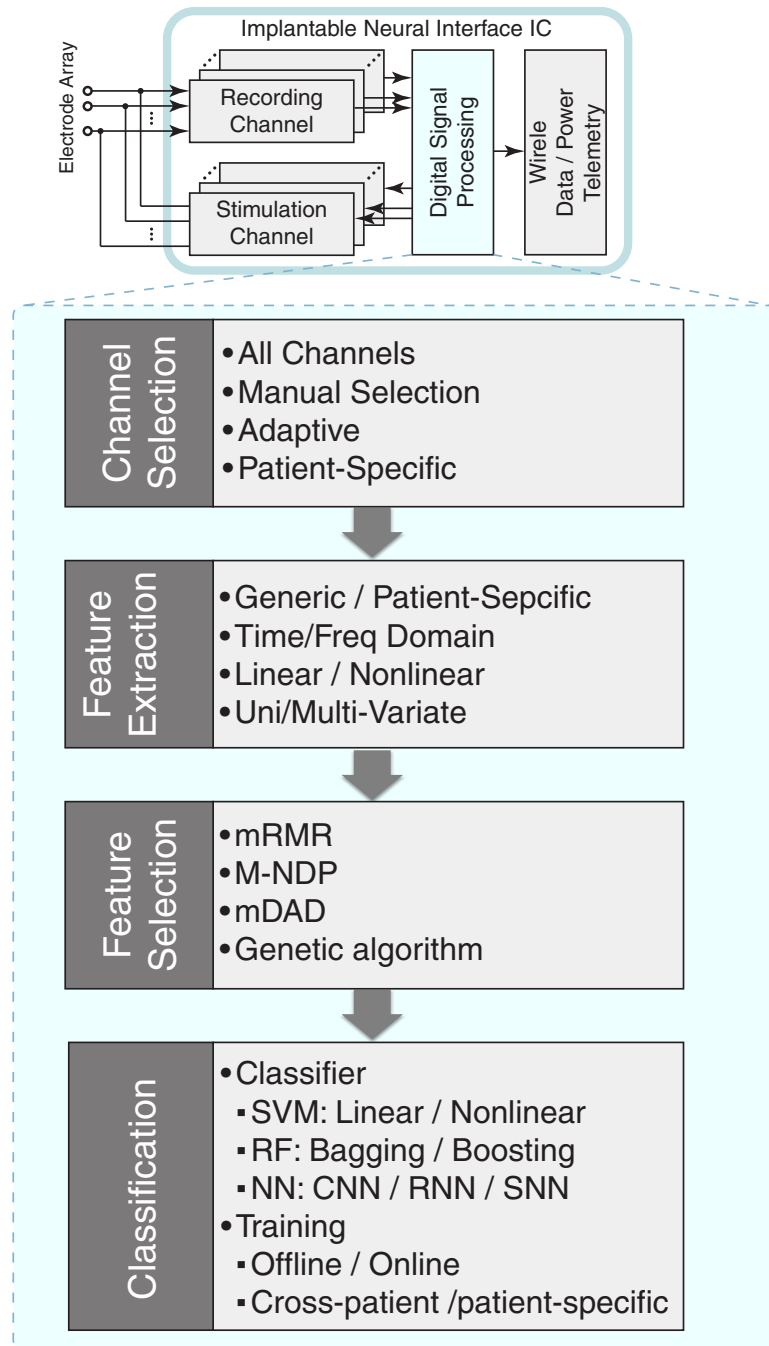
type further complicates comparisons and generalization across studies. Other notable datasets include the TUH EEG Seizure Corpus (642 subjects, scalp) [16], Melbourne-Neurovista (12 subjects, iEEG) [17], Kaggle Upenn and Mayo Clinic (12 subjects, iEEG) [18], Kaggle American Epilepsy Society (7 subjects, iEEG) [19], Helsinki University Hospital (79 subjects, scalp) [20], and Siena Scalp EEG (14 subjects, scalp) [21].

## **1.3 Importance of individualization**

### **1.3.1 Need for a data-driven approach**

Seizure detection algorithms generally involve four main processing stages: channel selection, feature extraction, feature selection, and classification, as summarized in Figure 1.6. While we always need a classifier, the other stages may or may not be present in different implementations. Each of these stages offers a variety of options, which will be discussed extensively over next sections. Despite this diversity, a common critical factor among all stages is the need for individualization. This need arises from the significant variations in EEG manifestations of seizures across different channels, different patients, and over time. For example, as shown in Figure 1.7, EEG activity during seizures may exhibit flattening or oscillations in specific frequency bands across different electrodes and with different delays after the seizure onset. Additionally, as shown in Figure 1.8, artifacts such as chewing, eye blinking, or sleep spindles can further complicate signal analysis by creating patterns that have temporal or spectral similarities to seizure patterns.

Given the variability and complexity of the problem, relying on a fixed approach for all these



**Figure 1.6: Four main stages of signal processing conducted on multi-channel implantable closed-loop neurostimulators for seizure detection, and some of the more common options for each stage.**

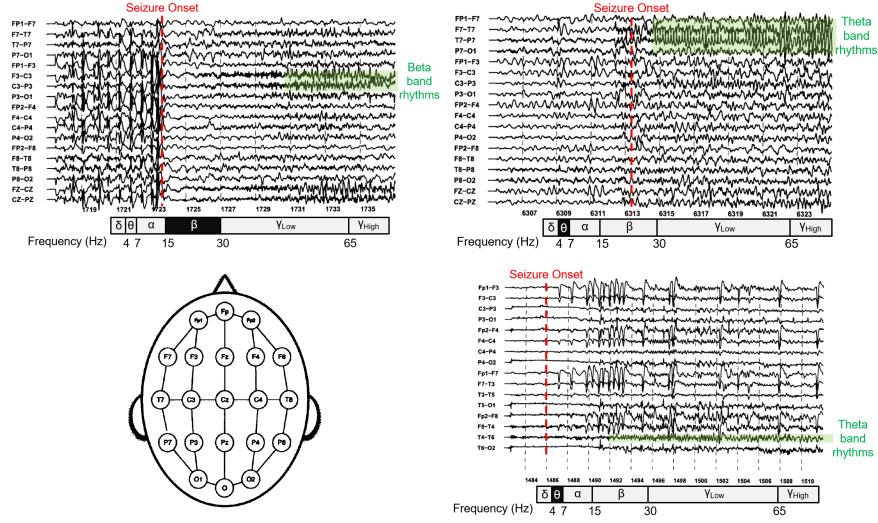


Figure 1.7: Examples of diverse seizure manifestation in EEG recordings for different patients.

Data from [22]

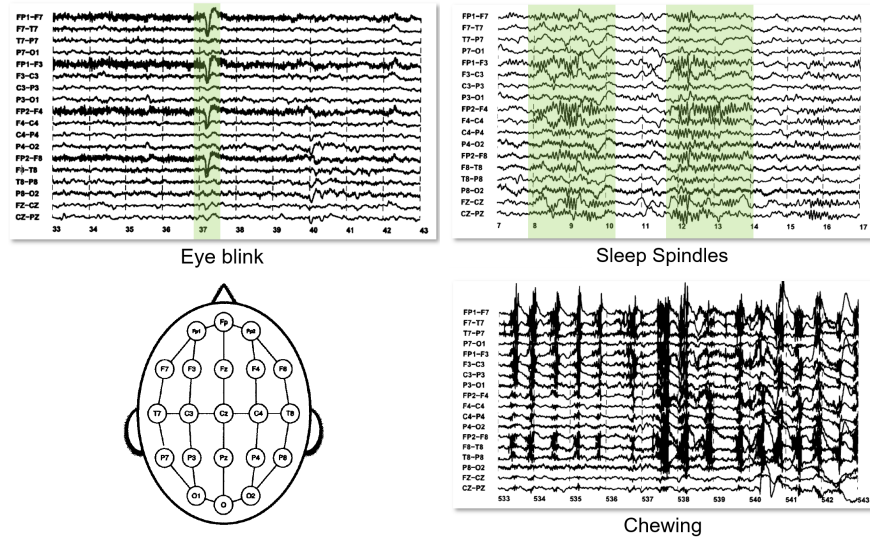


Figure 1.8: Examples of different seizure-like artifacts that could appear on EEG recordings. Data from [22]

four stages does not yield the best performance. Instead, a more adaptive strategy is required. This includes (a) using data from a subset of recording channels that are chosen in a patient-specific way and/or in a time-adaptive fashion, (b) extracting a set of patient-specific and/or channel-specific and/or time-adaptive features from the raw data, (c) employing a time-adaptive approach for feature redundancy reduction, and (d) using a data-driven machine learning (ML)-based algorithm for classification.

### **1.3.2 Channel selection**

Channel selection in EEG-based seizure detection is crucial for both accuracy and efficiency. Different strategies are employed based on the specific requirements and constraints of the application. Using all channels can provide comprehensive spatial information but increases computational complexity and power consumption, which is not ideal for resourceconstrained implantable devices. Moreover, not all channels contribute useful information for seizure detection; some may introduce noise or irrelevant features, potentially reducing overall detection accuracy.

A fixed subset of channels, typically chosen based on prior knowledge of common seizure focal points, can reduce complexity but might not be optimal for all patients. As demonstrated in [23], feature extraction conducted only on a subset of channels identified as most critical for a specific patient during offline training could be an alternative solution (Figure 1.9(a)). In [23], it is shown that for each patient, data from a specific set of channels provide better differentiation between seizure and normal states than using all channels or a fixed subset (Figure 1.9(b)). In [24], it is shown that the optimal number of spectral bands for phase-related and energy-related

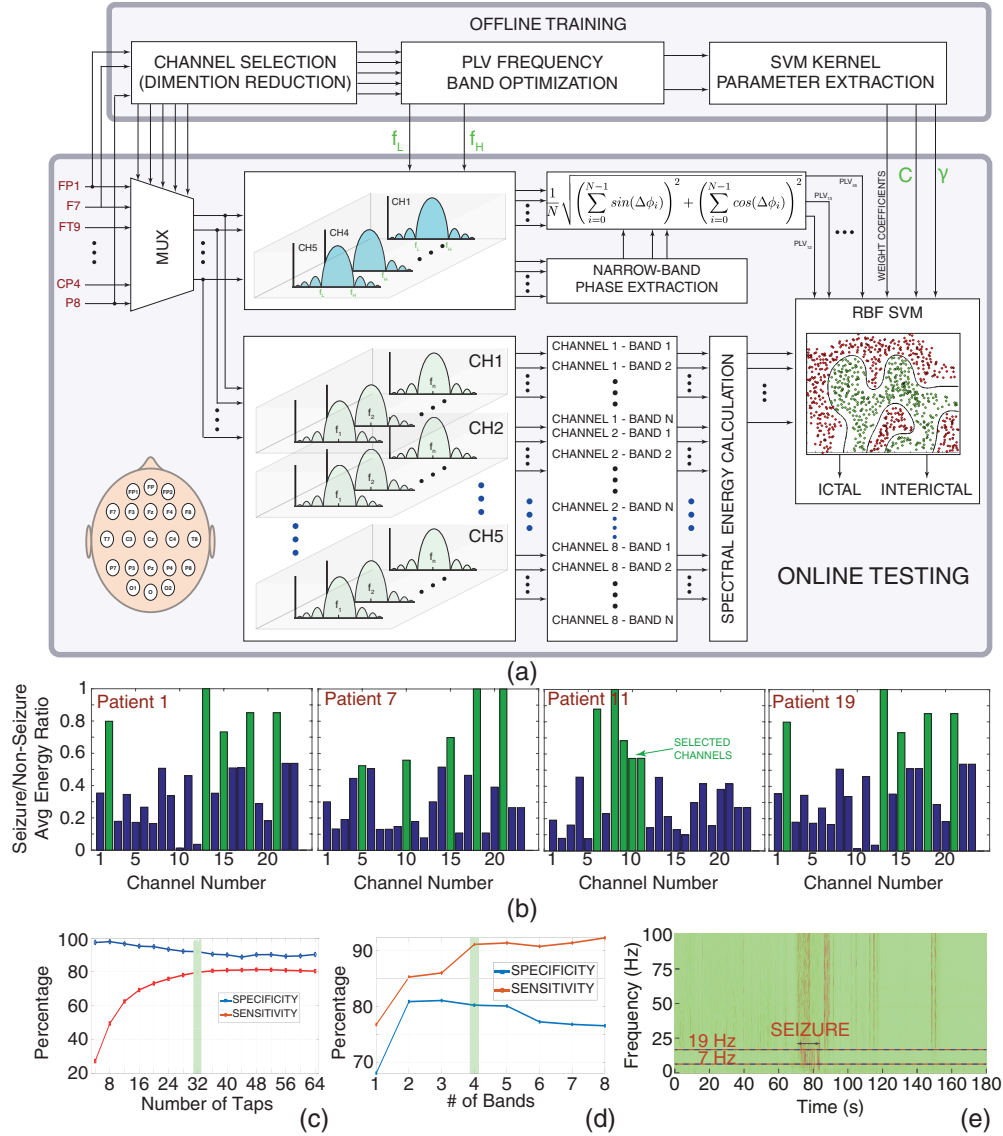
feature extraction varies across patients and can be optimized in a patient-specific manner (Figure 1.9(c-e)).

Based on the above, an individualized adaptive approach, where a subset of channels is selected based on patient-specific seizure patterns, offers improved detection accuracy by focusing on the most relevant regions for each patient. Furthermore, adaptive channel selection, which dynamically adjusts the subset of channels based on real-time analysis, can potentially offer the best performance by continuously optimizing the channels used for detection as seizure characteristics evolve over time. This approach, while computationally more demanding, can provide a balance between accuracy and resource efficiency, making it a promising area of research for developing advanced, patient-tailored seizure detection systems.

### **1.3.3 Feature Extraction**

In general, feature extraction aims to distill relevant information from input signals. Features can be univariate or multivariate (i.e., using data from a single or multiple channels), linear or nonlinear, and may encompass both time and frequency domains. In the literature, a diverse range of features has been explored to capture various aspects of EEG signals. Multivariate features offer significant advantages over univariate ones, particularly in the context of seizure detection, as they can capture the spatial network connectivity of seizures and the importance of synchrony across different brain regions. This spatial information is crucial for accurately identifying seizure activity, which often involves complex interactions between multiple areas of the brain.

Moreover, employing features from different domains and those that are orthogonal to each



**Figure 1.9: (a) Feature extraction conducted on a subset of channels identified as critical for a specific patient during offline training. (b) Comparison showing better differentiation between seizure and normal states using patientspecific channels versus all channels or a fixed subset [23]. (c) Optimization of spectral bands for phase-related and energy-related feature extraction tailored to individual patients [24]**

other enhances the robustness and comprehensiveness of the detection system. Features derived from the time domain can provide insights into signal amplitude and temporal dynamics, while frequency domain features reveal information about the signal's spectral composition. Mixed-domain features combine aspects of both, offering a more holistic view. Using a combination of diverse features, rather than relying on a single type, can improve detection accuracy and reliability by capturing a wider array of signal characteristics, of course at a higher computational cost.

### **1.3.4 Feature Selection**

As mentioned, in most seizure detection systems, a combination of multiple features is used. The challenge lies in determining which features are most informative for distinguishing between seizure and non-seizure states. With numerous available features, selecting the most discriminative and non-redundant ones is essential for building robust classification models. Feature selection methods aim to identify the optimal subset of features that enhance detection accuracy and efficiency.

A few of these methods were listed in Figure 1.6. Minimum Redundancy Maximum Relevance (mRMR) balances relevance to the target class (seizures) with the reduction of redundancy, optimizing overall information gain by prioritizing unique and informative features [25]–[27]. Minimum Normalized Difference of Percentiles (M-NDP) selects features with significantly different percentile distributions between seizure and non-seizure classes, making them more effective for distinguishing between states [28]. Maximum Difference of Amplitude Distribution Histograms (mDAD) emphasizes features with distinct histogram distributions, capturing signifi-

cant value range differences between seizure and non-seizure states [29]. The Genetic Algorithm, inspired by natural selection, evolves candidate feature subsets based on classification performance, iteratively refining them through reproduction and elimination of less effective features, thus uncovering optimal combinations [30].

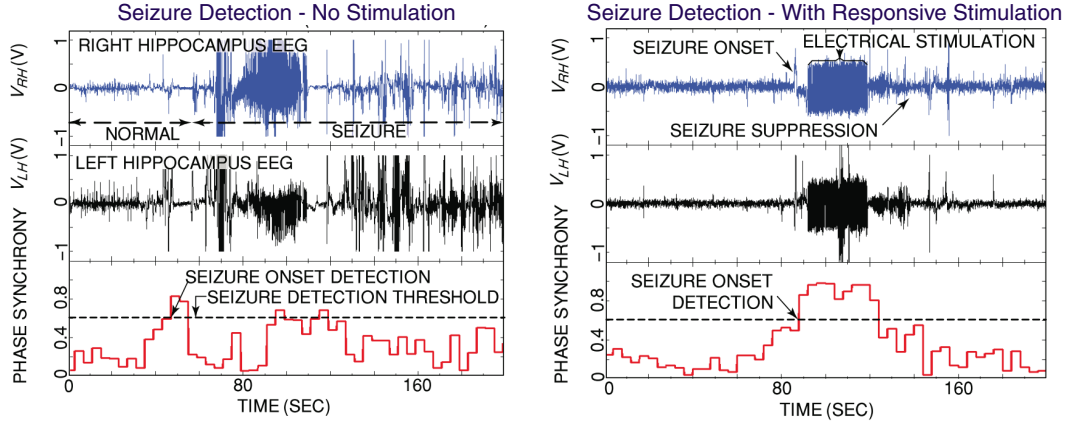
Each of these methods has its strengths and limitations. mRMR is effective in balancing relevance and redundancy, making it suitable for datasets with many features. M-NDP and mDAD focus on distribution differences, providing intuitive and direct ways to select discriminative features. The Genetic Algorithm offers a flexible and adaptive approach, capable of exploring a vast search space but may require significant computational resources.

## **1.4 Patient-specific classification**

### **1.4.1 Classification algorithms and their on-chip implementation**

Early reports on seizure detection algorithms/systems employed thresholding as the classifier [31]–[33]. It is the classifier used in the previously-discussed Neuropace’s RNS, the only FDA-approved closed-loop implantable stimulator for epilepsy. Figure 1.10 shows an example of successful thresholding applied to extracted phase-based features (PLV) from a pair of channels, with and without subsequent responsive stimulation for seizure suppression [34]. While straightforward and efficient, it lacks adaptability to the dynamic nature of seizure activity over time and across different patients, often resulting in high false positive or false negative rates. Motivated by this limitation, machine learning (ML) algorithms and their energy-efficient hardware implementations have been extensively explored over the past decade to achieve patient-specific





**Figure 1.10: Example of thresholding as a simple, efficient, and straightforward, yet not patient-adaptive, classifier used for seizure detection [34].**

classifiers [35]–[42], [43].

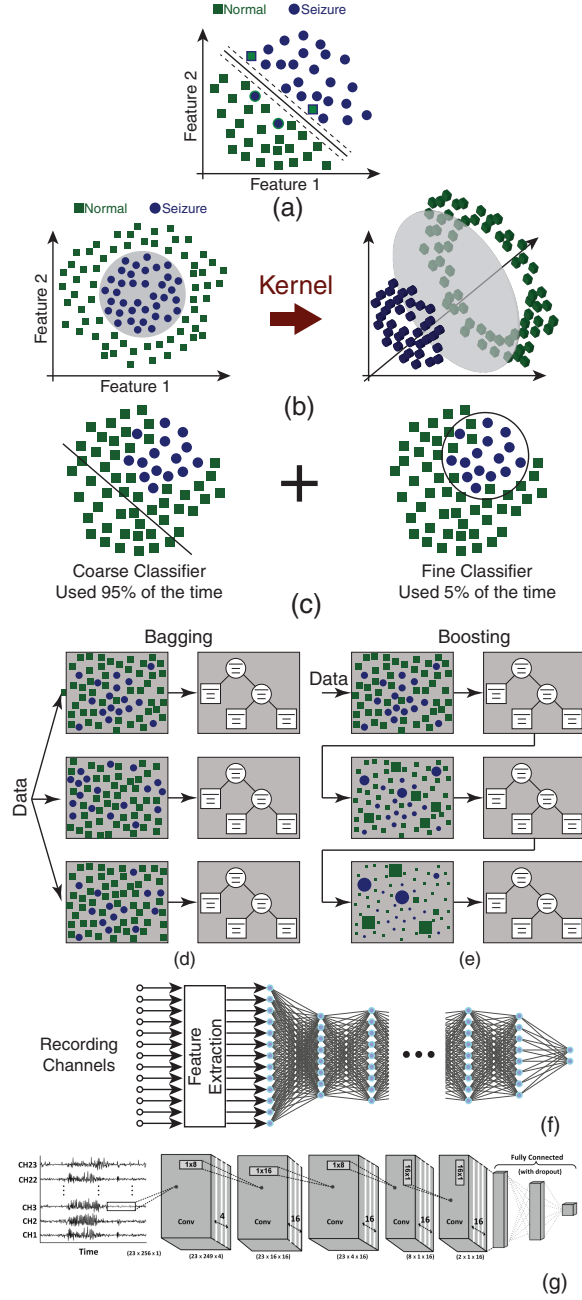
When comparing classifiers, the metrics discussed in Section 1.2.2 are used to evaluate their performance from both algorithmic and hardware perspectives. Additionally, robustness to noise and generalizability are crucial aspects in selecting classifiers. The former ensures that the classifier maintains high performance despite the artifacts and signal variability inherent in EEG data, which is essential for real-world efficacy. The latter refers to the classifier’s ability to adapt and perform well across different seizure patterns and conditions, ensuring reliability in varying clinical scenarios. In the following sections, we will discuss three popular categories of such algorithms: support vector machines (SVM), decision trees (DT), and deep neural networks (DNN), and highlight recent example implementations for each.

### 1.4.2 Support Vector Machine (SVM)

As a popular ML-based classifier known for their effectiveness in binary classification, SVMs operate by finding an optimal hyperplane that maximizes the margin between classes

in a high-dimensional feature space. While earlier works utilized linear SVMs shown in Figure 1.11(a) [35], [43], more recent efforts have shifted towards employing kernel methods to enable the use of non-linear SVMs for seizure detection (Figure 1.11(b)), resulting in improved accuracy, of course, at a significantly higher computational cost. To put this into perspective, in [38], it is shown that for ECG-based cardiac arrhythmia detection, a polynomial kernel requires more than  $20,000\times$  energy per classification than a linear kernel, and a radian basis function (RBF) kernel requires more than  $2\times$  the polynomial one. A similar pattern exist for memory requirements of SVMs with such kernels, i.e., RBF requiring  $>82\times$  and  $>2\times$  memory than linear and polynomial kernels.

This has motivated investigating various system, circuit and algorithmic/arithmetic-level techniques to improve the energy and resource efficiency of nonlinear SVMs' hardware implementation. For example, in [44], authors employ a natural log operator to linearize the RBF kernel, which results in replacing many of the multiplication operations with addition, effectively reducing the computational cost of the classification. In [45] authors employ a two-level coarse-fine SVM (conceptually depicted in Figure 1.11(c)), in which the coarse level applies thresholding on averaged temporal differences, yielding high sensitivity but low specificity. Meanwhile, the fine classifier implemented on a separate DSP IC that gets activated only if the coarse classifiers output indicate an upcoming seizure, computes maximum modulus discrete wavelet transform and kernel density estimation (KDE), and feeds them to a least squares SVM for high-selectivity seizure detection. By doing so, only 266 classification are conducted per hour, resulting in a significant power reduction without sacrificing accuracy. In [46], authors use an auto-encoder to perform spatial filtering and principal component extraction, which significantly enhances resource utilization efficiency by dimensionality reduction. Additionally, an exponentially de-



**Figure 1.11: Conceptual illustration of popular ML-based classifiers used for seizure detection, including (a) linear, (b) nonlinear, and (c) coarse-fine SVMs, random forest classifiers with (d) bagging, and (e) boosting techniques, and deep neural networks (f) with and (g) without dedicated feature extraction**

caying memory (EDM) SVM is used for classification that enables more efficient hardware implementation by circumventing the limitations of windowing methods, as it represents complex temporal relationships without the need for accumulating and concatenating fixed windows. By employing continuous sampling and recursive windowing with efficient EDM updates, the SVM can process multiple timescales simultaneously, optimizing memory usage and minimizing detection latency for capturing subtle seizure onset biomarkers.

As a final note on SVMs, we must mention that a major issue with SVMs is that they struggle with imbalanced datasets common in epilepsy research, where seizure instances are significantly fewer than non-seizure instances. In such cases, techniques such as data resampling, where the dataset is balanced by either oversampling the minority class or undersampling the majority class, or cost-sensitive SVM, which assigns different misclassification costs to different classes to address the class imbalance, are often employed [48]–[50].

### **1.4.3 Decision Trees**

Decision trees offer a straightforward approach to classification by recursively partitioning feature space based on simple yes-no questions until reaching a decision. To achieve a competitive performance, researchers often employ ensembles of decision trees, leveraging the collective wisdom of multiple trees working in parallel. Two common ensemble methods are bagging and boosting (Figure 1.11(d-e)). In bagging, multiple decision trees are trained on randomly selected subsets of the data, and their decisions are aggregated through voting. This approach reduces the risk of over-fitting and enhances robustness by incorporating diverse perspectives from individual trees. However, one drawback of bagging is that it can sometimes lead to less interpretability

of the overall model compared to a single decision tree. Boosting on the other hand, iteratively trains decision trees in a sequential manner, with each subsequent tree focusing on correcting the mistakes of its predecessors. By adjusting parameters to prioritize misclassified instances, boosting can progressively improve classification accuracy. However, boosting may increase the risk of overfitting to the training data.

In [51], authors employed gradient-boosted decision trees for seizure detection. They implemented asynchronous tree operation, allowing each decision tree to work independently and in parallel, enhancing efficiency and reducing latency by enabling dynamic decision-making based on completed computations. Additionally, they utilized sequential feature extraction, extracting features step by step to optimize computational resources, enhance energy efficiency, and prioritize relevant features for accurate classification without exceeding resource constraints. These strategies collectively contributed to the achievement of significant improvement in energy and area efficiency. In [52], authors present an adaptivelyboosted decision tree ensemble classifier architecture featuring a 1024-tree tile array for enhanced temporal boosted classification, addressing long and short-term temporal dependencies. It also utilizes a bit-serial approach, dynamically adjusting bit-precision to compensate for quantization errors at longterm timescales, thus reducing area overhead and enhancing accuracy. In [53], authors employ a gradient-boosted machine learning model for patient-specific feature extraction. This method trains decision tree ensembles for each patient's data, extracting relevant features tailored to their physiological signals, which enhances classification accuracy by focusing on informative features. They also optimize hardware for energy efficiency using shallow trees with limited features to reduce overfitting and computational complexity. Additionally, asynchronous architecture is employed to enable parallel processing of input data, improving performance by averaging decisions from multiple trees

while minimizing detection latency.

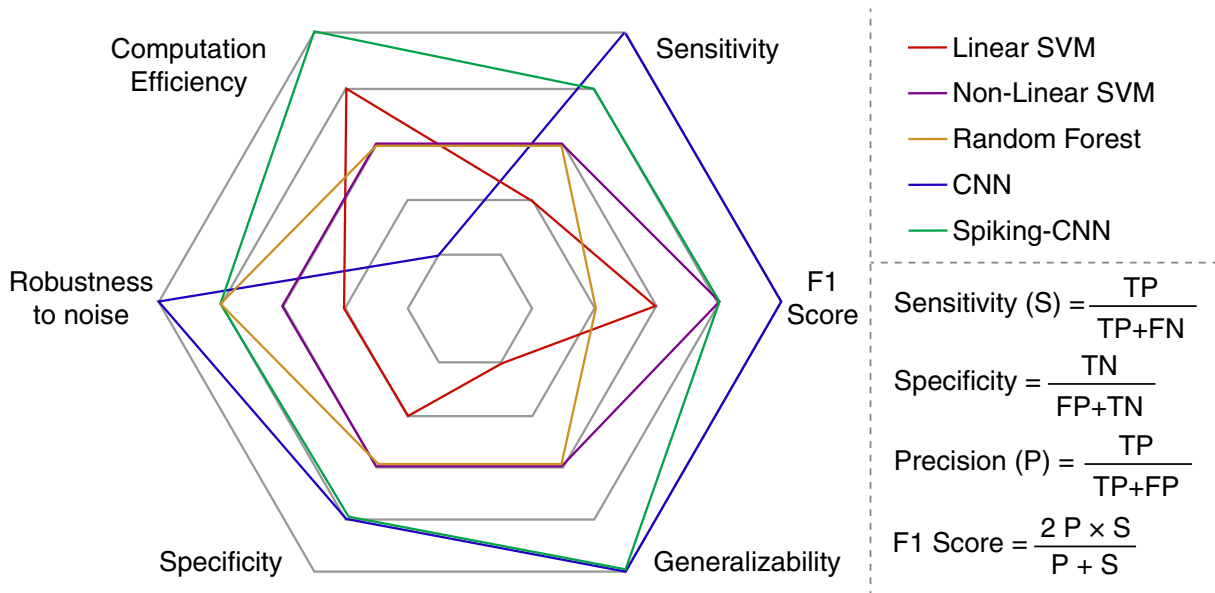
#### **1.4.4 Deep Neural Networks (DNN)**

Due to their higher computational resource requirements, DNNs initially gained less popularity for implantable on-chip seizure detection compared to other ML algorithms. However, their unique advantage of processing raw neural recordings, instead of using pre-determined extracted features, has become increasingly attractive. As shown in Figure 1.11(f-g), DNNs can take both predetermined features and the raw signals as their input. The former has the advantage of using knowledge of the field, i.e., extracting features that we know are of high importance, while the latter allows the training process to decide which features are best for each patient and each channel, resulting in a highly-individualized solution that has further enhanced DNN's appeal for seizure detection. Emerging techniques for improving hardware and energy efficiency have also contributed to the growing popularity of DNNs in recent years.

Among DNN architectures, convolutional neural networks (CNNs) are widely favored for seizure detection due to their ability to apply spatial filters to input data, capturing spatial patterns inherent in EEG recordings (Figure 1.11(g)). As discussed earlier, seizures often involve synchronous activity across different brain regions, emphasizing the importance of spatial feature extraction. However, CNNs are among the most computationally expensive DNNs. Addressing their higher computational cost (i.e., power consumption and memory requirements) requires innovative solutions to optimize resource utilization. Techniques such as model compression, quantization, and hardware acceleration aim to achieve this. A recent approach involves leveraging neuromorphic computing, particularly spiking neural networks (SNNs) [54]. SNNs con-

duct computation using asynchronous temporally-coded spikes, which have key advantages of (a) eliminating numerous resource-intensive multi-bit multiplications, and (b) adapting computation load to input activity levels, ideal for information-sparse signals like EEG/iEEG.

Figure 1.12 provides an overview of some popular classifiers utilized in these implementations, aiming to illustrate the qualitative strengths and weaknesses of each method.



**Figure 1.12: Qualitative comparison of most common classifiers used for seizure detection based on the most important performance metrics.**

It is important to note that this is a high-level comparison based on typical characteristics of these classifiers, and selecting the most suitable classifier for a seizure detection application depends on several factors, including dataset size, performance metric priorities, and available hardware resources. Fairly comparing classifiers can be challenging due to significant variations in the size and type of datasets, performance metric definitions, and signal pre-processing

techniques employed across different studies.

Among the choices for seizure detection, support vector machines (SVMs) and decision trees have been the most popular. SVMs, known for their computational efficiency, can capture complex relationships through the use of kernel functions. However, their performance may be limited when dealing with large and high dimensional datasets. Decision trees offer a more interpretable model but may struggle with complex nonlinear dependencies present in EEG signals. On the other end of the spectrum, the CNN classifier demonstrates superior performance across multiple parameters but incurs high computational costs. Notably, CNNs exhibit high generalizability, enabling their application in patient-independent seizure detection scenarios [55]-[67].

To address the need for computationally efficient alternatives to CNNs, the emergence of neuromorphic computing has recently led to the development of spiking convolutional neural networks (SCNNs) [68]-[72]. SCNNs, with their asynchronous operation and suitability for information-sparse signals like EEG, offer an attractive solution for EEG-based seizure detection. They have shown great potential for impressive energy efficiency, especially when implemented on dedicated neuromorphic hardware [73]-[75]. Motivated by these advantages, this work focuses on the design, development, optimization, and hardware implementation of an SCNN for patient-specific seizure detection. Section 1.5 includes a brief discussion on the fundamentals of SNNs and next chapter explain various design choices throughout the development, conversion, and hardware implementation processes, and presents simulation and measurement results showcasing the performance of the proposed approach in terms of seizure detection accuracy and latency, as well as energy efficiency.



## 1.5 Fundamentals of SNNs

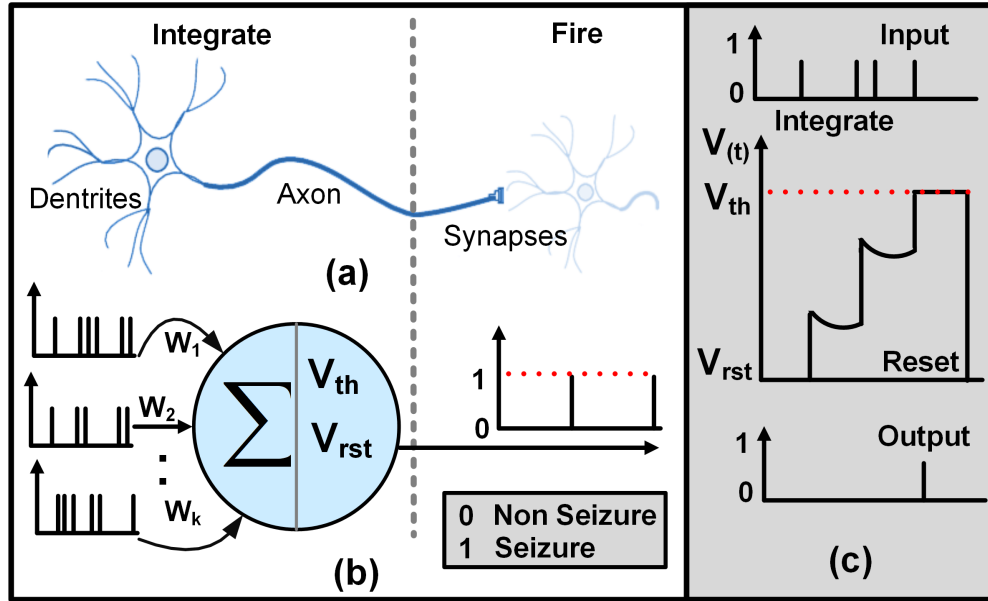
Spiking Neural Networks (SNNs) are a class of artificial neural networks that closely mimic the behavior of biological neural networks. While conventional DNNs process information in a continuous manner using layers of neurons that compute weighted sums and apply activation functions, resembling traditional digital computations, SNNs mimic the brain more closely by using neurons that communicate through discrete spikes or events. Instead of continuously updating neuron states, SNNs update only when spikes occur, making them event-driven and more energy-efficient. This spiking mechanism allows SNNs to capture temporal dynamics and process information in a time-sensitive manner, whereas DNNs typically handle static data with fixed layers and weights.

### 1.5.1 Building blocks

The fundamental building blocks of SNNs are spiking neurons and synaptic connections. These components work together to process and transmit information in a manner like their biological counterparts

Spiking neurons are the fundamental computational units in SNNs. They receive input signals from other neurons through synaptic connections and generate output spikes based on their internal state and activation function. The internal state of a spiking neuron is typically modeled as a membrane potential, which integrates the incoming signals over time. When the membrane potential crosses a specific threshold, the neuron fires a spike, and its membrane potential is reset.

Synaptic connections are the links that enable communication between spiking neurons.



**Figure 1.13: (a) Simplified biological model of a neuron and its major sections and their associated functions. (b) Operation principle of an LIF neuron model and (c) sample input and outputs.**

These connections have weights that determine the strength of the signal transmitted from one neuron to another. Synaptic weights can be excitatory, increasing the membrane potential of the receiving neuron, or inhibitory, decreasing the membrane potential.

## 1.5.2 Principle of Operation

Each neuron in an SNN receives input signals as spikes. A common model used is the Leaky Integrate-and-Fire (LIF) neuron, which integrates incoming spikes and generates an output spike when the membrane potential

The membrane potential  $V(t)$  of a neuron is updated based on incoming spikes and a decay over time. The equation governing this is:

$$\tau_m \frac{dV(t)}{dt} = -V(t) + RI(t) \quad (1.1)$$

where  $\tau$  is the membrane time constant,  $R$  is the membrane resistance, and  $I(t)$  is the input current from incoming spikes.

**Spike Generation:** When  $V(t)$  reaches the threshold  $V_{th}$ , the neuron fires a spike and resets its membrane potential:

$$V(t) = 0 \quad \text{if} \quad V(t) \geq V_{th} \quad (1.2)$$

During inference, the process of spike transmission and neuron activation happens layer by layer. The input neurons receive stimuli (e.g., from sensory data) and convert these into spike trains. These spike trains are then fed into the next layer of neurons. Each hidden layer neuron integrates incoming spikes from the previous layer, updates its membrane potential according to the LIF model, and generates output spikes if the potential crosses the threshold. This can be described as:

$$V_i(t) = V_i(t-1) + \sum_j w_{ij} S_j(t) \quad (1.3)$$

where  $w_{ij}$  are the synaptic weights from neuron  $j$  to neuron  $i$ , and  $S_j(t)$  indicates whether neuron  $j$  has fired a spike at time  $t$ .

The final layer of neurons accumulates spikes and provides the network's output based on the spiking activity. The output can be interpreted in various ways, such as the rate of spikes over a period or the timing of individual spikes.

### **1.5.3 Temporal dynamics**

One of the key features of SNNs is their ability to capture and process temporal dynamics. Unlike traditional artificial neural networks, which operate on static inputs, SNNs can process and learn from time-varying signals. This temporal aspect makes SNNs well-suited for applications involving sequential data, such as speech recognition, natural language processing, and time-series forecasting.

### **1.5.4 Advantages for implantable neural signal processing**

SNNs offer several advantages over traditional artificial neural networks, including improved biological plausibility, potential for energy-efficient hardware implementations, and the ability to process temporal and spatial information effectively. Specifically, for neural signal processing, SNNs have the following advantages:

#### **Non-binary multiplier-less computation**

SNNs perform only accumulation (ACC) operations instead of the more computationally expensive multiply-and-accumulate (MAC) operations required by ANNs. This makes SNNs more efficient in terms of computational complexity. The lack of multiplication operations in SNNs simplifies the hardware design and reduces power consumption. Figure 1.14 shows the difference between conventional continuous-time ANN neuron model and multiplier-less spiking neuron.

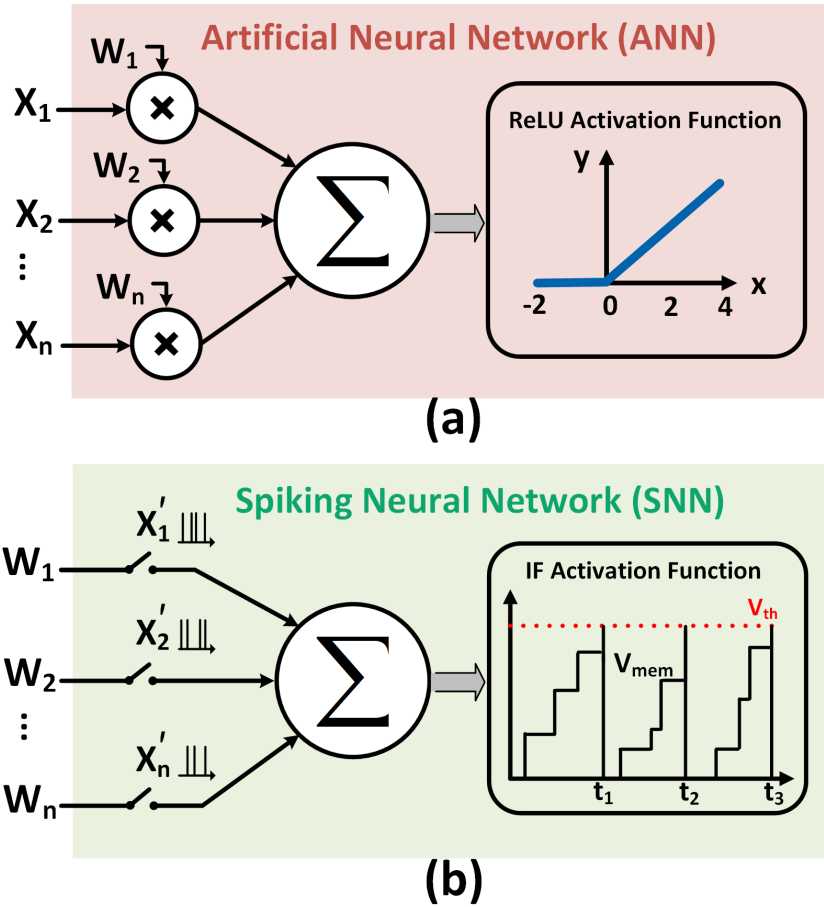
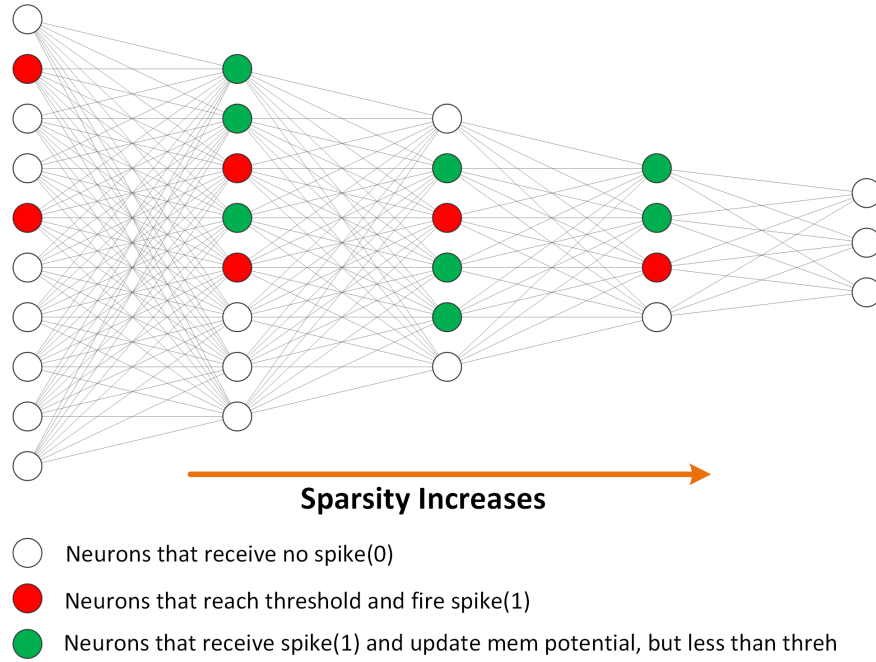


Figure 1.14: (a) ANN neuron model (b) Multiplier-less SNN neuron model.

### Asynchronous (Event driven operation)

SNNs operate in an asynchronous, event-driven manner, where neurons only update their state and communicate when they receive or generate a spike event as shown in Figure 1.15. Here the red circles This sparse activity leads to significant power savings compared to the synchronous, clock-driven operation of ANNs. The event-driven nature of SNNs allows for efficient exploitation of the inherent information sparsity in neural activity, reducing unnecessary computations and memory accesses.



**Figure 1.15: Event driven sparse SNN.**

## 1.6 Thesis objectives

The primary objective of this thesis is to develop Spiking Neural Network (SNN)-based neuromorphic processors tailored for EEG-based seizure detection, aiming to achieve optimal algorithm performance and hardware efficiency. To accomplish this, the following specific objectives have been identified:

1. Design and implement various SNN architectures suitable for processing EEG signals. These models will be trained using a comprehensive dataset to accurately detect epileptic seizures.
2. Enhance the performance of the developed SNN models through optimization techniques, ensuring high accuracy and low latency in seizure detection.

3. Develop and deploy these optimized SNN models onto hardware platforms. This involves rigorous testing and validation to ensure the hardware efficiency and reliability of the SNN processors in real-world scenarios.

### **1.6.1 Context and relevance**

Achieving these objectives will significantly contribute to the broader field of neuromorphic computing and epilepsy research. The development of efficient SNN-based processors for seizure detection could revolutionize the treatment and monitoring of epilepsy, providing a more accurate and timely response to seizures. This research holds the potential to enhance the quality of life for epilepsy patients through improved detection strategies.

### **1.6.2 Scope of the research**

This research will focus on the design, training, optimization, and hardware implementation of SNN model for EEG-based seizure detection. It will cover the following aspects:

- Detailed development of multiple SNN architectures.
- Comprehensive training and optimization of these models.
- Deployment and testing on neuromorphic hardware platforms.

## 1.7 Thesis organization

Chapter 2 details the design, optimization, hardware implementation (both generic and neuromorphic), and performance validation of regular and spiking convolutional neural networks (CNN and SCNN) for patient-specific seizure detection. It includes a comprehensive performance evaluation using the CHB-MIT dataset and compares the performance of conventional and neuromorphic hardware in implementing the seizure detection algorithm.

Chapter 3 presents a customized approach in development of a fully-spiking neural networks (SNNs) tailored for energy efficient data-driven signal processing in implantable brain neural interfaces. The importance of a customized design lies in its ability to optimize hardware and energy efficiency while maintaining high classification performance. Our approach allows for the customization of key parameters, including quantization resolution of weights and biases, encoding scheme, encoder placement, temporal resolution, neuron type, and internal parameters such as threshold value, resetting process, and refractory period. We demonstrated the efficacy of this customization in improving hardware and energy efficiency through model development, software-based training and testing.

Chapter 4 focuses on the hardware implementation of an energy-efficient spiking neural network for implantable neural interfaces. It begins with the Input Register, managing data from the LC-ADC, followed by the Spiking Convolution Layer, the core processing unit inspired by neural networks. The Neuron Model performs weighted summations and thresholding, while the Max Pooling Layer reduces spatial dimensions for computational efficiency. These components are integrated in the System Integration stage, forming a cohesive framework. The chapter concludes with Simulation results and Synthesis results for ASIC design, showcasing the system's



performance and hardware optimization.

Chapter 5 concludes the thesis and discusses possible future directions for this research work.

## 1.8 Thesis contributions

1. **A. Muneeb**, and H. Kassiri, “Energy-Efficient Spiking-CNN-Based Cross-Patient Seizure Detection,” *IEEE International Symposium on Circuits and Systems (ISCAS)*, Monterey, CA, USA, 2023, pp. 1-5.
2. **A. Muneeb**, S. Mehrotra and H. Kassiri, “A 9.5ms-Latency 6.2 $\mu$ J/Inference Spiking CNN for Patient-Specific Seizure Detection,” *IEEE Biomedical Circuits and Systems Conference (BioCAS)*, Toronto, ON, Canada, 2023, pp. 1-5.
3. **A. Muneeb**, and H. Kassiri, “Customized Development and Hardware Optimization of a Fully-Spiking SNN for EEG-Based Seizure Detection,” *IEEE Biomedical Circuits and Systems Conference (BioCAS)*, Xi’an, China, 2024, pp. 1-5.
4. H. Kassiri, **A. Muneeb**, R. Salahi, and A. Dabbaghian, “Closed-Loop Implantable Neurostimulators for Individualized Treatment of Intractable Epilepsy: A Review of Recent Developments, Ongoing Challenges, and Future Opportunities,” *IEEE Transactions on Biomedical Circuits and Systems (TBioCAS)*, 2024.

## Chapter 2

# A 9.5ms-Latency 6.2 $\mu$ J/Inference Spiking CNN for Patient-Specific Seizure Detection

In this chapter, we will present design, optimization, hardware implementation (both generic and neuromorphic), and performance validation of regular and spiking convolutional neural networks (CNN and SCNN) for patient-specific seizure detection. The primary objective is to develop highly accurate and efficient neural network model, tailored for this critical application, leveraging the unique characteristics of both conventional and spiking neural architectures. In addition, the chapter provides a performance summary along with computational resources comparison of conventional and spiking neural networks.

The chapter also discusses various design choices throughout the development, conversion, and hardware implementation processes, and presents simulation and measurement results showcasing the performance of the proposed approach in terms of seizure detection accuracy and

latency, as well as energy efficiency.

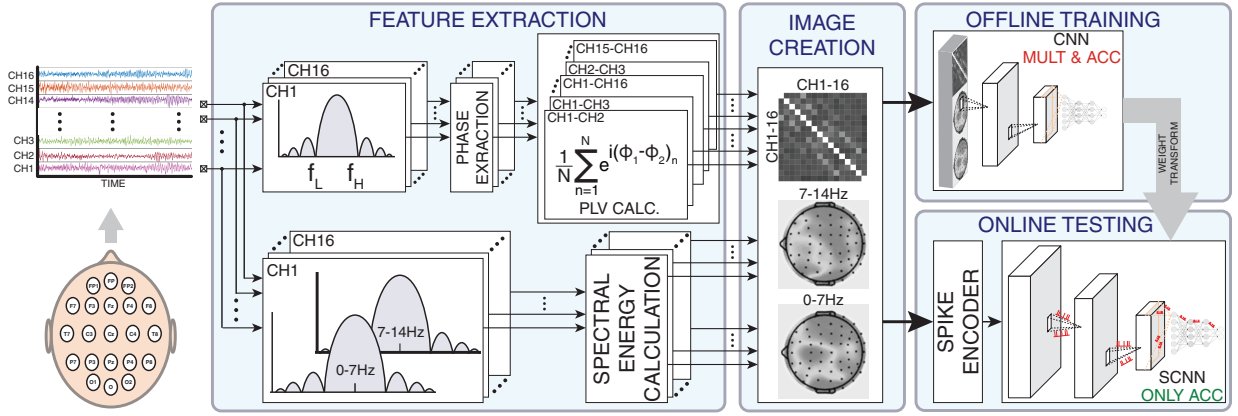
## 2.1 Input layer: raw EEG vs extracted features

The input to a Convolutional Neural Network (CNN) for EEG data analysis can be either raw signals or extracted features. Both approaches have their own advantages and disadvantages. Using raw signals as input allows the CNN to learn and extract relevant features automatically during the training process, potentially tailoring the feature extraction to be patient-specific and channel-specific. However, this might require a more complex CNN, which can increase computational resources. On the other hand, feature extraction involves preprocessing steps to derive features from the raw signals, often selected based on the designer’s knowledge of the field.

We have previously developed CNNs for seizure detection that take features as inputs, as described in our [55] paper. Figure 2.1 shows the architecture of this approach, focusing on feature extraction and image creation blocks. In this setup, spectral band energy (SBE) and phase locking value (PLV) are extracted from the EEG signals. SBE represents the spectral energy distribution, which is crucial for identifying abnormal energy patterns in different frequency bands. PLV measures phase synchrony between EEG channels, which is important for detecting inter-channel synchronization anomalies associated with seizures.

Features were extracted from 16 channels across two frequency bands (0-7 Hz and 7-14 Hz finite impulse response (FIR) band-pass filters), resulting in 32 SBE values as well as 120 PLVs. The features were mapped into three distinct 2D images, where pixel brightness represents the magnitude of the specific feature. For the spectral band energies (SBEs), each pixel corresponds

to the energy extracted from the EEG signal recorded at the electrode closest to that pixel. The first two images display the SBEs from 16-channel EEG recordings in the 0-7Hz and 7-14Hz frequency bands, respectively. The third image represents a phase locking value (PLV) matrix, where each pixel (i,j) indicates the PLV value between channels i and j. The three images were then combined (concatenated) and the result was fed to the CNN as shown in Figure 2.1. The CNN architecture is designed to process these images and perform seizure detection with high accuracy for both patient-specific and cross-patient scenarios.



**Figure 2.1: Top-level block diagram of the CNN-based seizure detection algorithm with feature extraction and image creation blocks.**

While the described system led to reasonable performance reported in [56] it also made problems with dedicated static feature extraction evident. Dedicated feature extraction can introduce computational overhead and complexity due to the additional preprocessing steps. More importantly, handcrafted features might not be optimal for every patient, requiring domain-specific knowledge and extensive trial-and-error.

To address these issues, we explore using raw EEG signals as inputs to the CNN. This approach eliminates the need for explicit feature extraction and allows the CNN to learn features

that are specific to individual patients and channels. While this can potentially reduce computational overhead, it may necessitate a more complex CNN architecture. Ultimately, the choice between using raw signals or extracted features depends on the specific application, the complexity of the task, and the available computational resources.

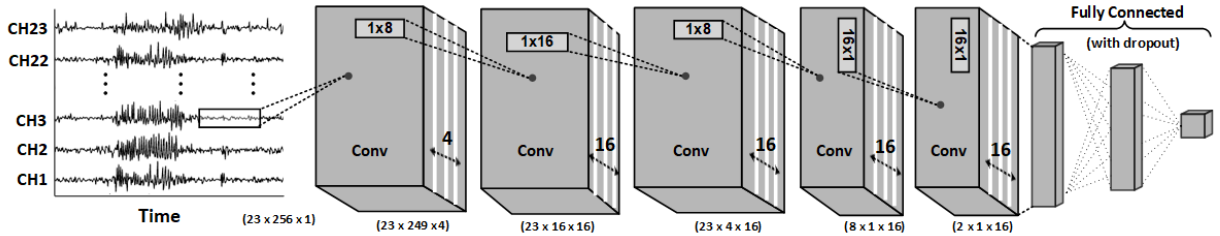
## **2.2 SCNN Development: Organic vs Shadow Training**

At the beginning of this chapter, we mentioned the presentation of a Spiking Convolutional Neural Network (SCNN). However, before diving into the CNN design and architecture, it is essential to clarify our approach. Currently, there are two options available for developing SNNs, each with its own set of advantages and challenges. Organic SNN training, which directly trains the network using spike-based learning rules, can capture the temporal dynamics of neural activity more naturally, leading to potentially more biologically plausible models. However, it can be challenging due to the complex and non-differentiable nature of spike-timing-dependent plasticity [57]-[59].

In contrast, shadow training, which involves training a DNN and then converting the model to a spiking one, leverages the well-established training algorithms of DNNs. This approach generally results in higher accuracy and more stable performance, as it benefits from the robust optimization techniques available for DNNs before conversion to an SNN [60]-[62]. Given the advantages of shadow training, such as reduced computational costs and leveraging pre-trained models, we have chosen this approach for developing our SCNN.

## 2.3 CNN Architecture Design

Figure 2.2 shows the top-level architecture of the developed convolutional neural network (CNN) for epileptic seizure detection from scalp electroencephalogram (EEG) signals. The architecture design, including the number of convolutional layers, kernel sizes, and other hyperparameters, was determined through neural architecture search (NAS).



**Figure 2.2: The developed CNN architecture utilizing 1D time- and spatial convolutions, with all kernels and layer dimension shown.**

### 2.3.1 Neural architecture search

Neural Architecture Search (NAS) is a technique that automatically generates and evaluates numerous candidate neural network architectures, assesses their performance on a given task, and iteratively refines the search process to find an optimal design. For our CNN, we employed strategies such as evolutionary algorithms and gradient-based optimization. Evolutionary algorithms, inspired by biological evolution, evolve a population of candidate architectures over generations, selecting the fittest individuals for mutation and recombination. Gradient-based optimization methods, on the other hand, leverage gradient descent to optimize the architecture parameters, treating the neural network design itself as a computational graph.

The search was driven by performance metrics like accuracy, sensitivity, and specificity, evaluated through training, cross-validation, and testing phases on a labeled scalp EEG dataset from epilepsy patients. We fed the NAS process with raw EEG data from each patient in the dataset, with 23 channels per patient. The outcome of our search led to the architecture shown in Figure 2.2, which we finalized based on its superior performance metrics compared to other candidates.

### **2.3.2 Input and preprocessing**

The raw EEG data from open-source CHB-MIT dataset is fed to the CNN model. The CHB-MIT EEG database is a comprehensive dataset widely used for research in epilepsy and seizure detection, collected at the Children’s Hospital Boston, this dataset includes EEG recordings from 24 subjects with intractable seizures. Each identified by a unique code (chb01, chb02, etc.).

The dataset includes EEG recordings from up to 23 scalp electrodes following the International 10-20 system. Subjects in the dataset range in age from 1.5 to 22 years, with a median age of around 11 years, and include both male and female participants. The dataset comprises approximately 916 hours of EEG recordings, with each recording session ranging from 0.5 to 4 hours in length. It contains a total of 198 seizures, with each patient’s seizure count varying significantly. EEG signals are sampled at a frequency of 256 Hz, providing high-resolution data suitable for detailed analysis.

For the input compatibility of CNN model a 2D image matrices of size  $23 \times 256$  (i.e., # of channels  $\times$  # of samples per second), data is fed to system representing one second of signals from 23 channels sampled at 256 Hz. Unlike prior work that extracts handcrafted features, we

chose to feed raw signals directly to the classifier. As mentioned previously, this data-driven approach allows the CNN to automatically identify the most critical features in a patient-specific manner, enabling it to capture unique patterns and characteristics present in individual EEG signals.

### **2.3.3 CNN layers**

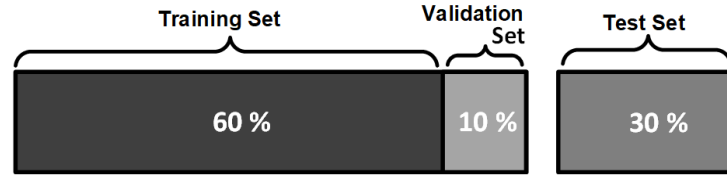
The CNN architecture consists of five convolutional layers with varying sizes, as shown in Figure 2.2. EEG signals differ from typical image datasets, where information can be mixed using 2D kernels. EEG signals have distinct time and frequency domain features. To capture both types of information from the input signal, we employed 1D kernels in the first three layers, using horizontal kernels to extract temporal features within channels. The last two layers use vertical kernels to capture spatial features across channels. This design effectively captures both short-term and long-range spatio-temporal dependencies. Max pooling layers follow, which downsample the feature maps and reduce their dimensionality. Finally, fully connected layers handle high-level reasoning and classification output. The depth and configuration of the convolutional layers allow the CNN to systematically extract higher-level spatial-temporal features that are robust, discriminative, and well-suited for the epileptic seizure detection task.

## **2.4 Train/test data division**

For training, we used 60% of data, and kept 10% for cross validation and 30% for testing as shown in Figure 2.3, to evaluate the patient specific seizure detection algorithm accuracy



of the dataset. To mitigate the challenge of an extremely unbalanced dataset, we employed a hybrid sampling strategy that combines random over-sampling of the minority class (i.e., ictal) with random under-sampling of the majority class (i.e., interictal). This significantly reduced the interictal-ictal ratio by almost two orders of magnitude (i.e., from 300 to 7), which led to improving the robustness of the classifier. During CNN training, the data was normalized between 0 and 1 to enhance the model’s learning capabilities by promoting stability, convergence, and improved performance in CNNs. The EEG data was also truncated by removing outliers that exceed three standard deviations from the mean, a practice referred to as three-sigma limits. This ensures a consistent scale for all inputs, preventing larger inputs from overshadowing smaller ones and avoiding biased learning outcomes. Additionally, effective operation of activation functions is ensured, improving the network’s ability to learn and generalize from the data.



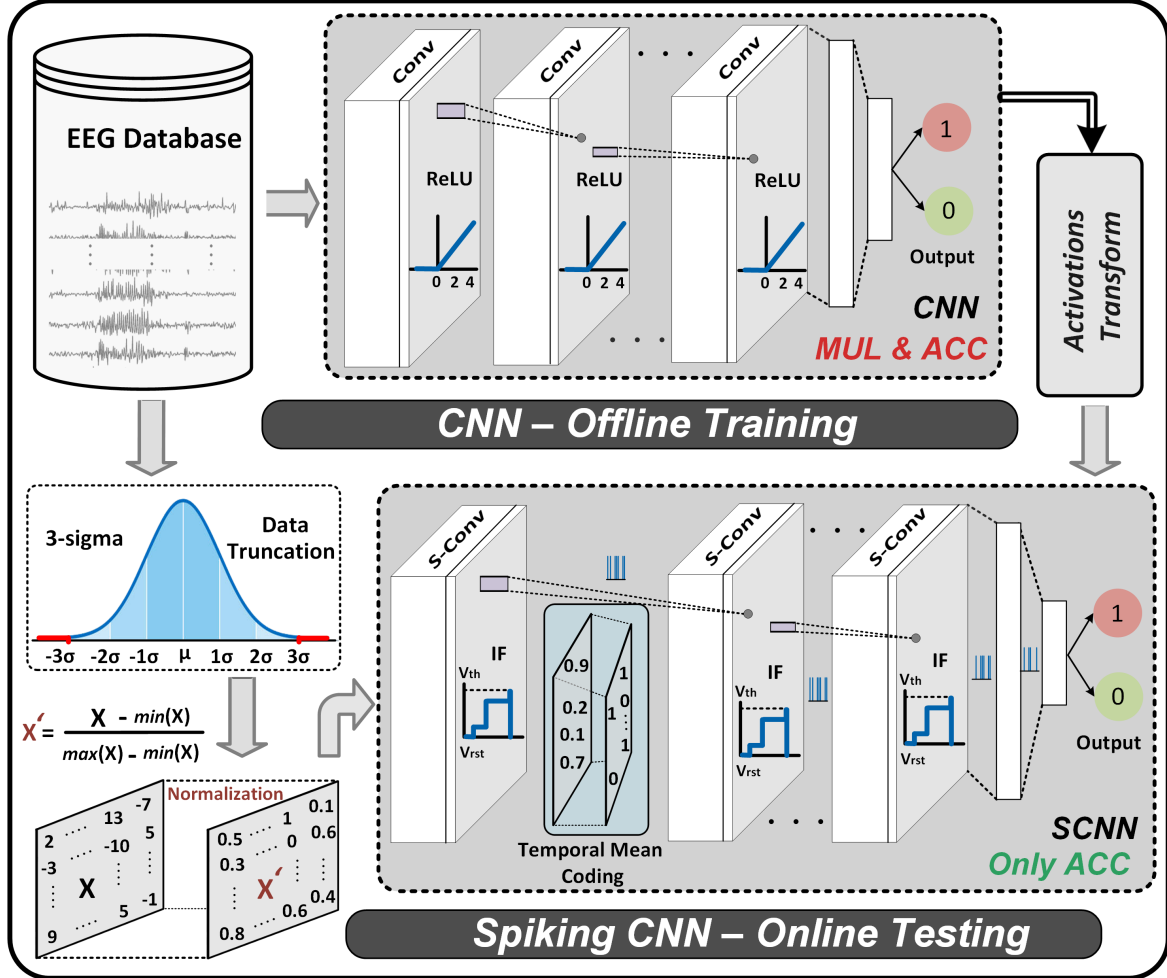
**Figure 2.3:** Data division for training, cross validation, and testing of the presented SCNN.

## 2.5 CNN to SCNN conversion

As shown in Figure 2.4, the spiking CNN is developed by converting the trained CNN model (i.e., shadow training). While the conversion process may introduce a slight performance degradation, it offers the advantage of leveraging well established learning and training techniques to enhance overall classification accuracy. For the spiking neuron model, we decided to use a

simplified integrate-and-fire model, which is widely employed in SNNs and has been successful in capturing the dynamics of neural firing.

The CNN weights were translated into the neuron model parameters such as neuron biases,



**Figure 2.4:** A simplified block diagram of the employed shadow training, showing steps taken for CNN-to-SCNN conversion and spike encoding.

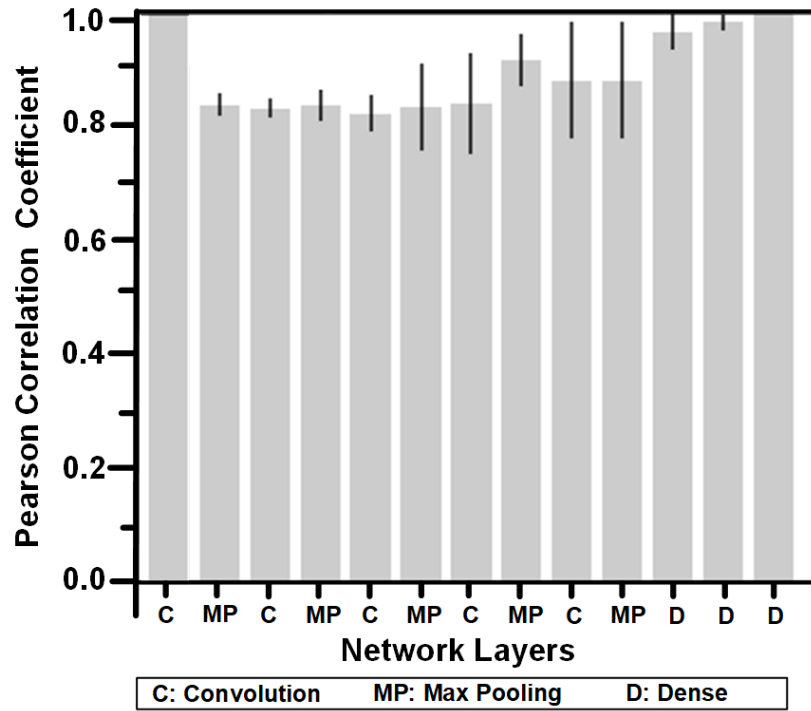
synaptic weights, or membrane time constants. We then fine-tuned them to adapt to the spiking neuron model and the temporal mean encoding scheme. Fine-tuning involved adjusting the SNN's parameters and optimizing them specifically for the spiking domain. We iteratively eval-

uated the SNN’s performance on a test dataset, measuring accuracy, sensitivity, and specificity.

For magnitude-to-spike-train conversion of the input data, we chose the temporal mean encoding scheme, in which data is represented by the mean firing rate of neurons over a specific time window. While other schemes (e.g., time to first spike (TTFS)) have shown better energy efficiency, we opted for temporal mean encoding due to its scalability and straightforward implementation pipeline for hardware platforms. The time-step for the temporal mean encoding was set as a trade off between energy efficiency and detection accuracy, as will be shown in the results section. It determines the granularity at which the input temporal data is divided and processed within the SNN. A shorter time-step allows for finer-grained temporal analysis at the expense of increased computational complexity. To evaluate the fidelity of the conversion process, we used Pearson correlation [76] between CNN activations values and transformed SCNN spike rates at the output of each layer, as shown in Figure 2.5. The high correlation at all layers indicates a close match between the CNN and SCNN, promising a successful conversion.

## 2.6 CNN/SCNN Evaluation

For performance evaluation both vector-based and event-based methods illustrated in Figure 2.6 and originally defined in [78] are used. Vector-based sensitivity means that each sample of the seizure event is detected properly, and event-based sensitivity means that the overall seizure episode is captured by the classification algorithm.



**Figure 2.5: Correlation between the CNN activations and SCNN spike rates for each layer and across 32 Samples (1-batch).**

The average “vector based” performance metrics for the CNN and SCNN are presented in Figure 2.7, which show an average sensitivity of 93.4% and 92.2% for CNN and SCNN, respectively, and an average specificity of 99.1% and 97.3% for CNN and SCNN, respectively for the open-source CHB-MIT dataset.

Similarly the patient-wise “vector based” performance metrics for the SCNN are presented in Figure 2.8, which shows sensitivity, specificity and accuracy of each patient.

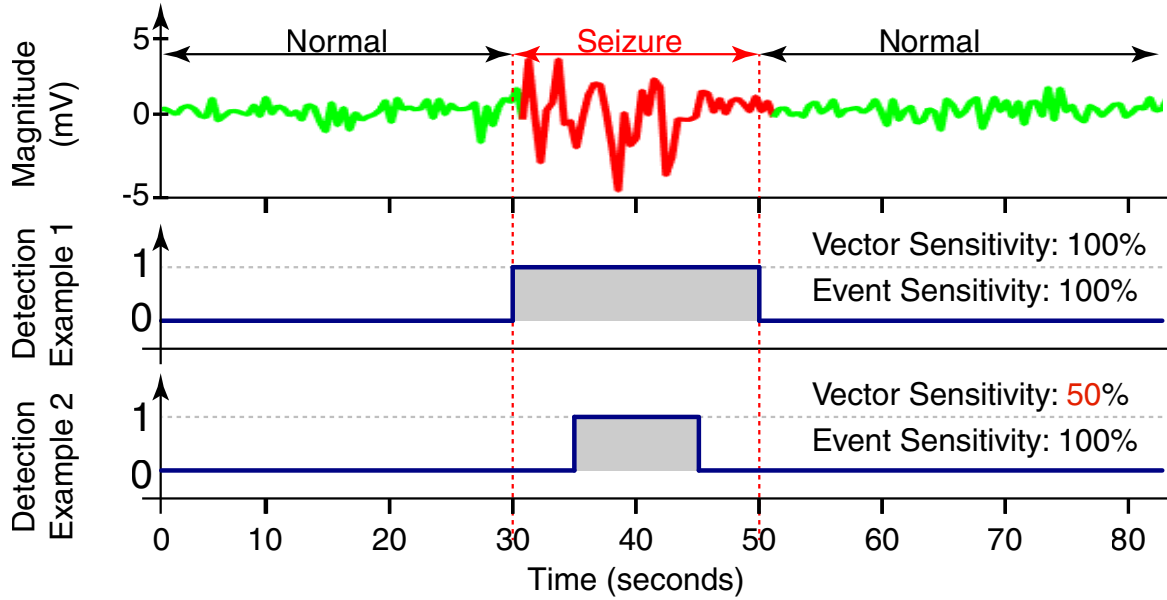


Figure 2.6: Definition of event-based and vector-based sensitivity in seizure detection.

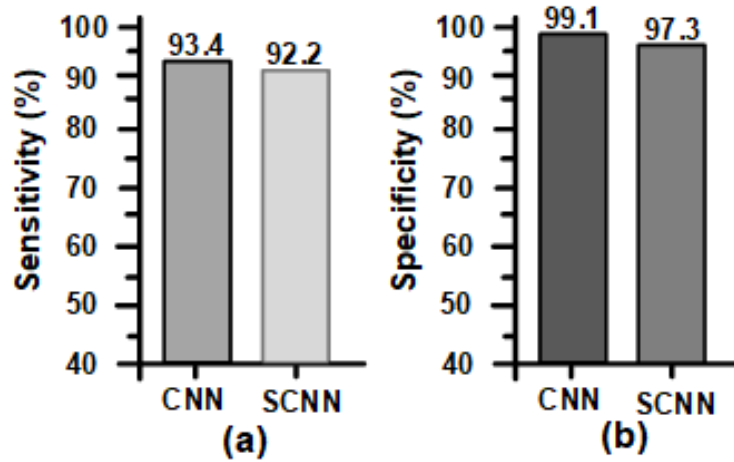


Figure 2.7: Classification performance of the developed CNN and SCNN in terms of (a) sensitivity and (b) specificity.

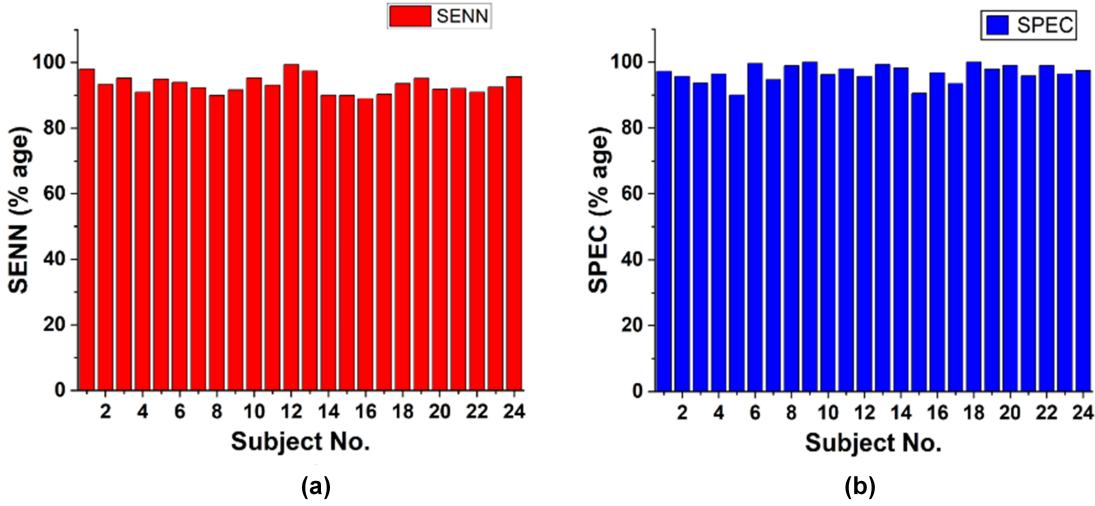


Figure 2.8: Subject-wise seizure detection performance of the developed SCNN.

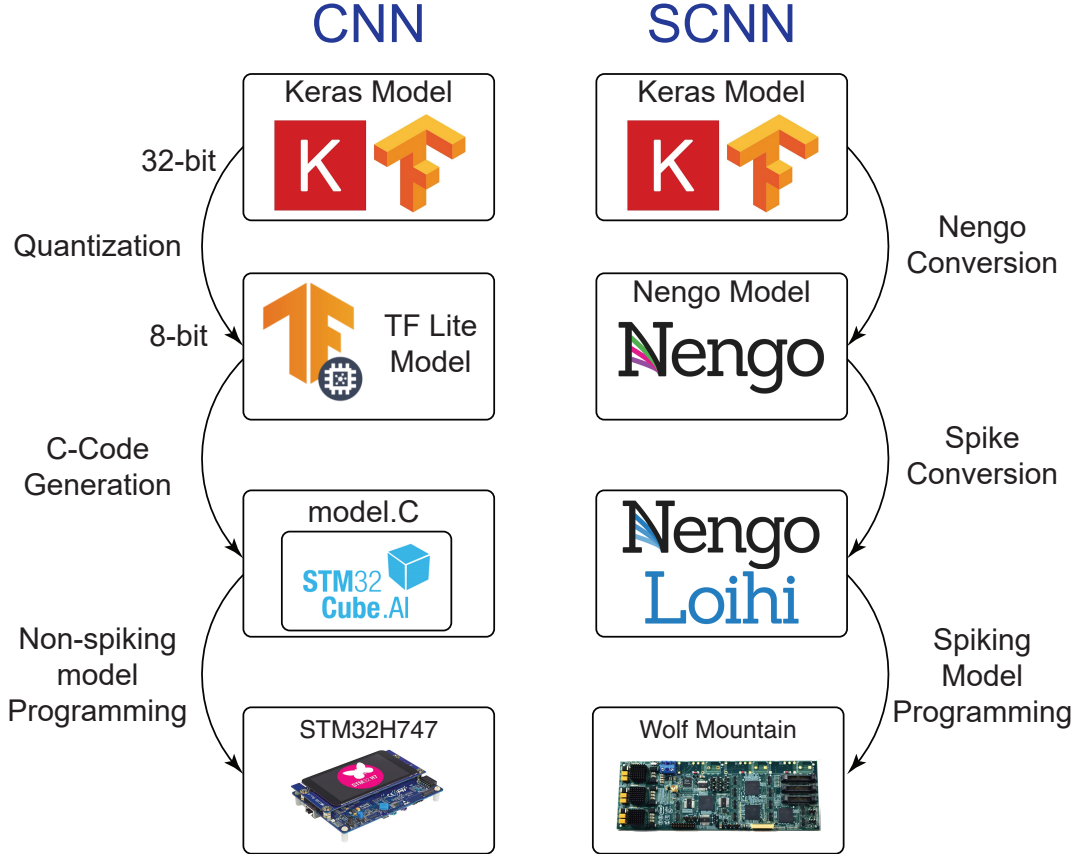
## 2.7 Hardware implementation

As discussed earlier, to be suitable for implementation on an energy constrained brain-implantable device, in addition to seizure detection accuracy, its energy efficiency and detection latency are of crucial importance. To evaluate computational and energy efficiency of the CNN model, it was programmed on a STM32H747 microcontroller [79], using the steps illustrated in Figure 2.9(left). This involved extracting the model weights from an HDF5 file and quantizing them from 32-bit floating-point values to 8-bit integers.

Once the model was quantized, further optimization and compression techniques that exploit redundancy and eliminate unnecessary parameters were applied. The compressed model was then fed into XCube-AI to generate C code suitable for the STM32 microcontroller. This code included all the necessary dependencies and libraries required for execution. The generated C code, along with the model and dependencies, was flashed onto the STM32H747I-DISCO evaluation board. With the model deployed on the microcontroller, pre-recorded labeled EEG data

were processed for seizure detection.

For the SCNN, the true energy efficiency cannot be exploited by standard conventional pro-



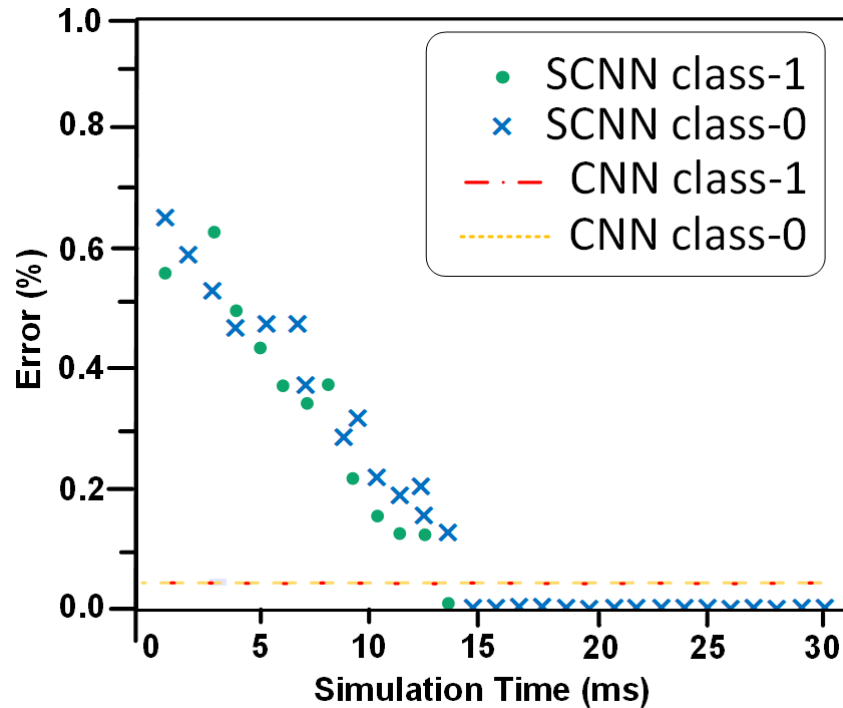
**Figure 2.9: Hardware Implementation Pipeline for (a) CNN on Micro-controller and (b) Spiking CNN on Loihi.**

cessors (e.g., CPU/GPU, microcontrollers, FPGAs, etc.) as they are inherently synchronous in nature. To deploy the SNN on an asynchronous, neuromorphic hardware to leverage its full capability, Nengo was used [80]. It allows conversion and deployment of non spiking models into spiking ones that can be deployed on actual neuromorphic hardware (such as Intel’s Loihi [81]) or simulated on a neuromorphic hardware simulator. Using this method (shown in

Figure 2.9(right)), we could estimate the energy efficiency and inference time of the SCNN in addition to its seizure detection accuracy.

As discussed, the choice of time-step in the spike encoding process affects both detection accuracy and latency. To strike an optimal trade-off between the two, as shown in Figure 2.10, simulation time (a direct function of the time-step for a fixed number of steps) was increased until the detection error of the SCNN reduced to below the CNN level.

Figure 2.11 shows how the SCNN's sensitivity and energy efficiency change as the time-step

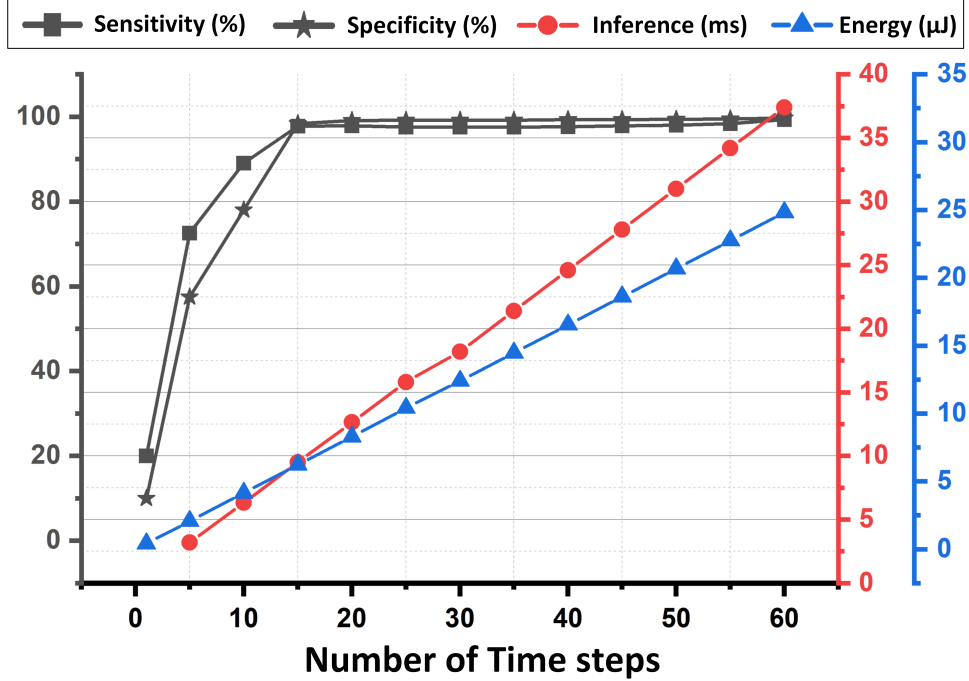


**Figure 2.10: Error rate versus the simulation time: SCNN vs CNN.**

(hence the inference time) is increased. As expected, the energy per inference increases almost linearly with the inference time. On the other hand, there is no significant increase in sensitivity beyond 15ms, suggesting that it is the optimal point.



SCNN's sensitivity, specificity, energy consumption, and Inference Time as a function of time-



**Figure 2.11: SCNN's sensitivity, specificity, energy consumption, and Inference Time as a function of time- steps, showing the optimal overall performance of the selected number of time-steps.**

steps, showing the optimal overall performance of the selected number of time-steps.

The spiking encoder's time step was accordingly set based on this optimal value for SCNN performance evaluation and comparison, as summarized in Table 2.1. The table shows a clear advantage of the SCNN in terms of product of energy efficiency and inference time. Given that the input data is fed to the algorithms every one second, the average power consumption for each case is estimated by multiplying  $\mu$ J/classification numbers with the inference time in seconds.

Table 2.2 compares the presented SCNN with the developed CNN as well as the state of the art, highlighting the excellent overall performance of the presented design.

Table 2.1: CNN vs SCNN Performance Comparison

|           | CNN                  |                   |                     | SCNN                 |                   |                     |
|-----------|----------------------|-------------------|---------------------|----------------------|-------------------|---------------------|
|           | Energy<br>( $\mu$ J) | Inf. Time<br>(ms) | Power<br>( $\mu$ W) | Energy<br>( $\mu$ J) | Inf. Time<br>(ms) | Power<br>( $\mu$ W) |
| CPU       | 6940                 | 15                | 104                 | -                    | -                 | -                   |
| STM32H747 | 17000                | 54.48             | 926                 | -                    | -                 | -                   |
| Loihi     | -                    | -                 | -                   | 6.22                 | 9.5               | 0.059               |

Table 2.2: Comparison with the state-of-the-art patient-specific seizure detection algorithms

| Parameter                   |        | JETCAS'18<br>[51] | TBioCAS'16<br>[44] | TBME'22<br>[63] | JSSC'22<br>[64] | This<br>Work |        |
|-----------------------------|--------|-------------------|--------------------|-----------------|-----------------|--------------|--------|
| Classifier                  |        | CNN               | NL-SVM             | CNN             | NL-SVM          | CNN          | SCNN   |
| HW Imp                      |        | ASIC              | ASIC               | N/A             | ASIC            | STM32H747    | Loihi  |
| Sensitivity<br>(%)          | Vector | N/R               | N/R                | N/R             | 97.8            | 93.4         | 92.2   |
|                             | Event  | 83.7              | 95.1               | 93.4            | 100             | 100          | 100    |
| Specificity/FPR             |        | 88.1%             | 99.1%              | 0.27/hr         | 99.5%           | 99.1%        | 97.3%  |
| Latency(s)                  |        | 1.79              | 2                  | N/R             | 0.74            | 0.054        | 0.0095 |
| Efficiency ( $\mu$ J/Class) |        | 0.04              | 1.83               | 2270            | 0.97            | 17000        | 6.2    |

N/R: Not Reported  
N/A: Not Applicable

In this chapter, we presented the design, optimization, hardware implementation, and performance evaluation of a CNN and SCNN for patient-specific seizure detection. Various design considerations for both networks were discussed and their design parameters (e.g. architecture and kernel size for CNN, and spike encoding and time step for SCNN) were optimized for the best overall performance. Our results from implementation of the SCNN on a neuromorphic asynchronous processor's simulator shows orders of magnitude energy efficiency improvement compared to the CNN, while achieving approximately the same seizure detection accuracy in

terms of sensitivity and specificity verified with CHB-MIT database. Our results for the SCNN also shows that it can achieve at least an order of magnitude ( $12.1\times$  better than the best prior art) improvement compared to the state-of-the-art in terms of Latency $\times$ Energy Efficiency, while yielding better or comparable seizure detection performance. The further details of custom implementation of such neuromorphic circuits discussed in Chapter 3.

## **Chapter 3**

# **Customized Development and Hardware Optimization of a Fully Spiking SNN**

### **3.1 Introduction**

In this chapter, we will present a fully customizable approach to the development of fully spiking neural networks (SNNs) tailored specifically for energy-efficient, data-driven signal processing in implantable brain neural interfaces. As pointed out in previous chapters, energy-efficient data-driven signal processing for implantable brain neural interfaces is of paramount importance, particularly in applications such as seizure detection, which require real-time analysis of multi-channels recordings while constrained by stringent implant size and battery life [47]. This has made the need for effective low-power implementation of such algorithms increasingly critical.

SNNs are highly regarded for their energy-efficient, asynchronous, event-driven processing of sparse neural signals, making them an attractive choice for these applications. However, to fully harness the potential of SNNs, certain critical design decisions must be meticulously optimized to ensure efficiency, latency, and accuracy. One pivotal factor is the spike encoding scheme, which directly influences how information is represented and processed within the network. The placement of spike encodings is even more important, as it will be analytically shown that it can significantly affect the computational cost of the entire network. Additionally, the quantization resolution plays a crucial role in balancing the trade-off between computational complexity and the precision of the processed signals. The number of time steps, or the temporal resolution, is another critical parameter that affects the network’s ability to capture dynamic features over time. Without careful optimization of these parameters, the performance of SNNs can be significantly compromised. Thus, a thorough understanding and fine-tuning of them is essential for maximizing the benefits of SNNs and achieving optimal performance in real-world applications.

However, current tools for SNN development, such as SNN Toolbox or SNN-Torch, fall short in enabling the necessary optimization of these critical parameters. These tools are primarily developed for image processing and vision applications, which limits their flexibility and applicability to time-series data. One significant limitation is that they do not allow for the described extensive customization required for SNN development. More importantly, they output SNNs with a non-spiking first layer, which undermines the inherent computational cost benefits of SNNs. The presence of a non-spiking first layer introduces inefficiencies and increases the overall energy consumption, negating one of the primary advantages of using SNNs. Given these constraints, it is clear that existing tools are inadequate for fully optimizing SNNs for specific

applications. Therefore, there is a pressing need for a custom development framework that allows for the bespoke design and fine-tuning of SNNs from scratch. Such a framework would enable researchers and developers to fully exploit the potential of SNNs by allowing comprehensive control over all critical parameters, ultimately leading to more efficient, accurate, and low-latency neural network models.

In this chapter, we introduce a fully customizable approach for developing spiking neural networks (SNNs), streamlining the model conversion process from Python to hardware description language (HDL)-based implementation, ready for FPGA and subsequent ASIC implementation. Our framework allows comprehensive customization of encoding schemes, weight and bias quantization resolution, and other critical parameters, optimizing both energy and area efficiency of the SNN’s hardware implementation.

We will explore various design choices throughout the development, conversion, and hardware implementation processes. Additionally, we will present simulation results that demonstrate the performance of our proposed approach, focusing on seizure detection accuracy, latency, and energy efficiency.

## **3.2 Motivation for a customizable SNN shadow training**

Considering the event-driven asynchronous nature of SNNs, it is reasonable to assume that the number of spikes decreases rapidly as the signals propagate through the network layers. Assuming an exponential decay in spike activity, the first layer is responsible for processing a significant majority of the operations. To estimate the

computational load of the first layer in an SNN with  $n$  layers, each containing  $N$  neurons per layer and an exponential spike decay factor  $r$  ( $0 < r < 1$ ), let  $T_i$  be the number of spikes per neuron in layer  $i$ , with  $T_{i+1} = r \cdot T_i$ . The total computations in the network,  $C_{\text{total}}$ , and the computations in the first layer,  $C_1$ , can be expressed as follows:

$$C_{\text{total}} = \sum_{i=1}^n N \times T_i = N \times T_1 (1 + r + r^2 + \dots + r^{n-1}) \quad (3.1)$$

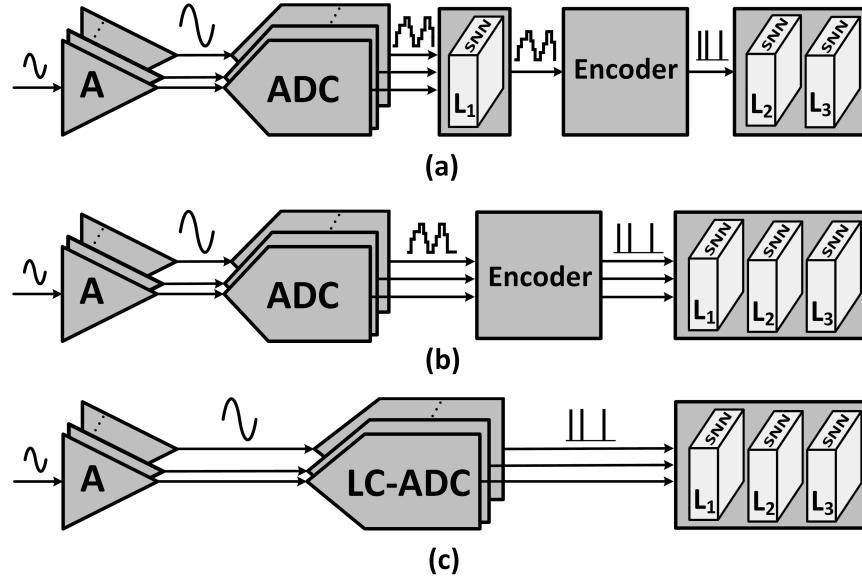
$$C_{\text{total}} = N \times T_1 \frac{1 - r^n}{1 - r} \quad (3.2)$$

The first layer quota is (FLQ):

$$\text{FLQ} = \frac{C_1}{C_{\text{total}}} = \frac{N \times T_1}{N \times T_1 \frac{1 - r^n}{1 - r}} = \frac{1 - r}{1 - r^n} \quad (3.3)$$

This estimation highlights the critical importance of optimizing the first layer for efficient spike processing, as it significantly impacts the overall performance and energy efficiency of the SNN. As illustrated in Figure 3.1, a fully spiking SNN, including the first layer, can be integrated with event-driven, asynchronous spiking front ends, which can unlock substantial power savings for implantable brain-computer interfaces (BCIs). Traditional analog front ends with amplifiers and ADCs consume considerable power, reducing the benefits of an energy-efficient neuromorphic processor.

In contrast, level crossing ADCs (LC-ADCs) operate asynchronously, responding to variations in the input signal. This results in event-driven power consumption and data output, pro-



**Figure 3.1: (a) A partially spiking neural network (PSNN) compared to (b) and fully spiking neural network (FSNN) realized through pre-SNN encoding. (c) An FSNN interfaced with asynchronous recording circuit.**

viding a more power-efficient solution compared to conventional front ends [66].

A PSNN is a hybrid neural network where only some of the layers or neurons operate using spike-based computation, while others function like traditional artificial neurons. This mixed approach can provide certain benefits, such as leveraging the advantages of spiking neurons for specific tasks (e.g., temporal coding and low-power operations) while maintaining the simplicity and high performance of traditional neural networks for other tasks. However, PSNNs may not fully exploit the potential benefits of spiking neural networks, such as reduced power consumption and increased biological plausibility.

Fully Spiking Neural Network (FSNN) realized through pre-SNN encoding. An FSNN, on the other hand, is entirely composed of spiking neurons across all layers. This network processes



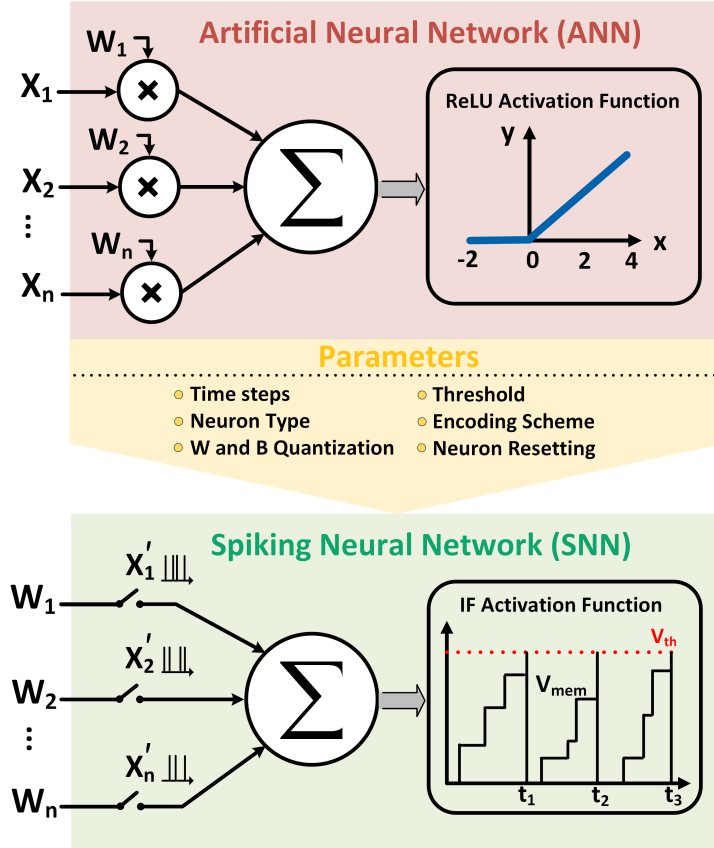
information using spikes, which are discrete events, mimicking the way biological neurons communicate. Pre-SNN encoding refers to the process of converting input data into spike trains before feeding them into the network. This encoding is crucial for ensuring that the entire network operates using spike-based computation. FSNNs can achieve greater energy efficiency and are well-suited for applications that require low-power consumption, such as implantable medical devices.

A FSNN interfaced with an asynchronous recording circuit represents a highly efficient and biologically inspired system. These circuits record neural signals in an event-driven manner, activating only when there is a significant change or event in the input signal, rather than continuously sampling at regular intervals. This leads to substantial power savings, as the circuit remains inactive during periods of no significant signal change. The asynchronous nature of these circuits aligns perfectly with the event-driven processing of FSNNs. By only recording and processing spikes, when necessary, the system significantly reduces power consumption, which is critical for battery-powered or energy-harvesting applications like implantable brain-computer interfaces (BCIs).

The integration of an asynchronous recording circuit with an FSNN ensures that both the input acquisition and processing stages operate in a low-power, event-driven manner. This synergy maximizes the energy efficiency of the entire system. Such a setup is particularly advantageous in medical devices, where energy efficiency is crucial. For instance, in implantable BCIs used for seizure detection or other neural monitoring applications, the combination of asynchronous recording circuits and FSNNs can lead to longer operational life and reduced need for battery replacements.

Various toolboxes and libraries are publicly available for the development and training of SNNs. For example, SNN Toolbox [62] is a versatile tool designed to facilitate the conversion of trained artificial neural networks (ANNs) to SNNs, as shown in Figure 3.2. It supports various deep learning frameworks, enabling seamless integration and conversion.

Another example is SNN Torch [59], a library built on top of PyTorch, providing a compre-



**Figure 3.2: High-level visualization of the shadow training process for SNN development and the critical parameters that need to be decided to ensure a hardware-optimized implementation.**

hensive framework for creating, training, and evaluating SNNs. Despite their advantages, these tools have critical constraints (listed in Table I) that hinder the optimization of SNNs in terms of computational and energy efficiency.

Table 3.1: Constraints of SNN Development Approaches/Examples

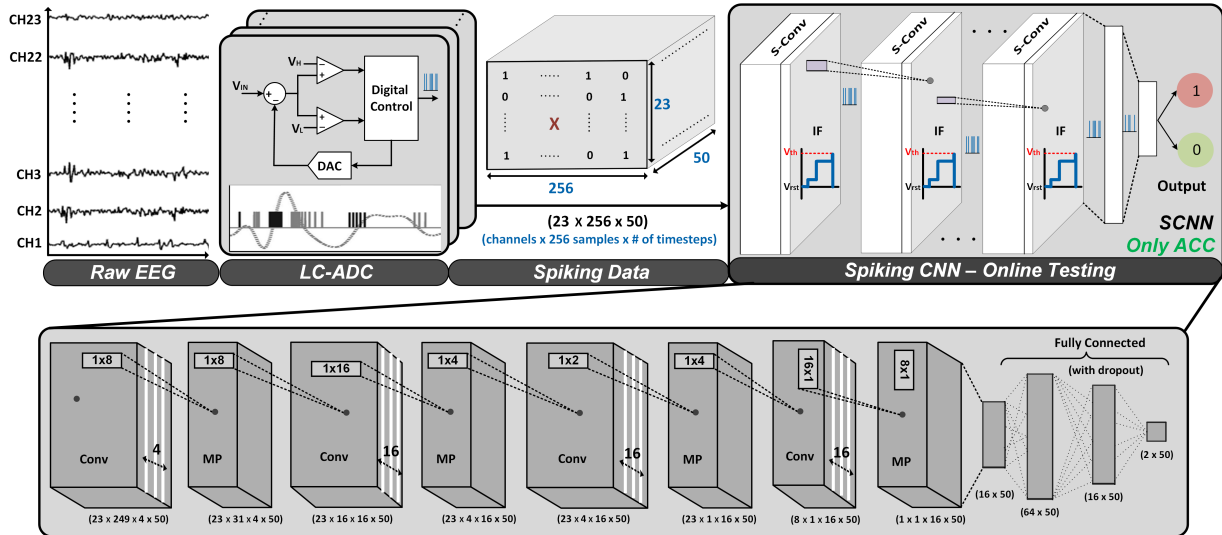
| Implementation                 | Hardware | Fully Spiking | Quantization | Encoder before first layer |
|--------------------------------|----------|---------------|--------------|----------------------------|
| <b>Spiking-CNN [83]</b>        | ×        | ✓             | ×            | ✓                          |
| <b>Spiking-CNN [61]</b>        | ✓        | ×             | ×            | ✓                          |
| <b>SNN Toolbox [62]</b>        | ×        | ×             | ×            | ×                          |
| <b>SNN Torch [59]</b>          | ×        | ×             | ×            | ×                          |
| <b>Spiking Con-former [82]</b> | ×        | ×             | ×            | ×                          |

### 3.3 Model development and evaluation

The development and evaluation of the SCNN model involves several key steps to ensure optimal performance and efficiency. The model development phase includes designing the network architecture, customizing spiking neuron parameters, and implementing effective encoding schemes. After developing the model, extensive training and fine-tuning are conducted using relevant datasets to achieve high accuracy and robustness. For evaluation, the SCNN model undergoes rigorous testing to assess its classification performance, latency, and energy efficiency. Simulation results are analyzed to compare the SCNN’s performance with traditional neural networks and other SNN models. Metrics such as accuracy, spike rate, and power consumption are measured to ensure the SCNN meets the desired criteria for practical applications, particularly in energy-constrained environments like implantable brain-computer interfaces.

### 3.3.1 CNN design and conversion methodology

Figure 3.3 illustrates the top-level architecture of the proposed seizure detection system, where the spiking convolutional neural network (SCNN) is connected to an array of LC-ADCs. These LC-ADCs feed asynchronously temporally coded data into the network. For designing the SCNN architecture, including the number of convolutional layers and kernel sizes, we employed neural architecture search (NAS).



**Figure 3.3: Top-level block diagram of the system architecture, illustrating the characteristics of the input signals and encoded spiking data. The diagram also details the architecture of the spiking convolutional neural network, which utilizes 1D temporal and spatial kernels. All kernel and layer dimensions are shown for clarity.**

This approach involves automatically generating, evaluating, and refining numerous candidate architectures based on their performance. The performance of the proposed model was evaluated through training, cross-validation, and testing phases using a labeled scalp EEG dataset .

The final design consists of four convolutional layers: three temporal-convolution layers to extract time-domain features from different channels, followed by one spatial-convolution layer to extract cross-channel spatial features from the input data. To enhance training and mitigate overfitting, each convolutional layer is followed by batch normalization and max pooling. Finally, the architecture includes three fully connected layers.

After making the design decisions for conventional CNN, which includes number of layers, type of layers, and neuron type. Next, we converted the developed CNN into a spiking CNN using a custom-made process. Instead of relying on existing libraries for SNN inference, we implemented the entire system from scratch using only NumPy. Our approach was grounded in a theoretical understanding of SNN operations, including the internal mechanisms of spiking neurons, layer interactions, and overall network function.

First, we developed a single spiking neuron, defining its internal mechanism for asynchronous membrane potential updates, including integration and resetting. We then defined layers, kernels, and the convolution process in a fully customizable, parametric fashion, ultimately constructing the network with a flexible number of layers.

This approach allowed us to choose all aspects of the resulting SCNN, including network architecture (such as the number of layers, neurons per layer, and connectivity patterns like fully connected or convolutional), neuron models, encoding schemes, and parameters' quantization resolution. We also ensured visibility to all nodes of the network, enabling comprehensive monitoring and analysis of signals received and generated by each neuron, as well as their real-time membrane potential.

### 3.3.2 Temporal encoding and resolution

As depicted in Figure 3.4, the process of converting Convolutional Neural Networks (CNNs) to Spiking Convolutional Neural Networks (SCNNs) extends beyond simply determining weights and biases during the training phase. Key decisions include selecting appropriate encoding schemes and defining temporal resolutions for spike generation. We opted for Poisson rate encoding due to its straightforward implementation and intuitive process. Poisson rate encoding involves representing input data as spike rates, where higher rates indicate stronger input signals. This method is particularly compatible with Level Crossing Analog-to-Digital Converters (LC-ADCs), which are commonly used in asynchronous event-driven systems.

Additionally, we had to decide on the number of time-steps, which corresponds to the number of time-domain quantization levels. This decision involved balancing computational efficiency with the accuracy of seizure detection. By carefully selecting the number of time-steps, we aimed to achieve a suitable compromise that optimizes both the computational resources required and the precision of our detection system. Overall, these decisions, choosing Poisson rate encoding and determining the number of time-steps, were crucial steps in adapting the CNN architecture to function effectively as an SCNN, especially in the context of real-time applications such as seizure detection in neurological monitoring systems.

Figure 3.4 shows the performance evaluation and the trade-off between number of time steps and computational cost for the developed SNN architecture, suggesting 50 to be an optimal number, beyond which detection performance does not improve significantly

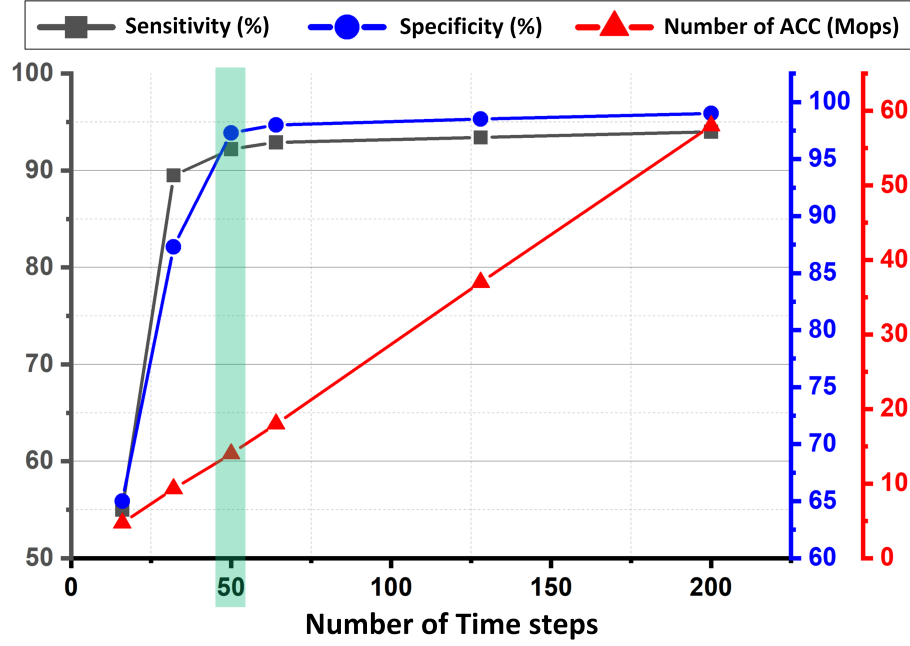


Figure 3.4: Seizure detection sensitivity, specificity, and computational cost as a function of the SNN's temporal resolution.

### 3.3.3 Neuron model

Integrate and fire (IF) neuron model is used, for which the discrete-time equation for updating the membrane potential is given by:

$$V(t+1) = V(t) + I(t) \quad (3.4)$$

where  $V(t)$  is the membrane potential at time  $t$  and  $I(t)$  is the input current at time  $t$ . If the updated membrane potential  $V(t+1)$  exceeds the firing threshold  $\theta$ , a spike is generated and the membrane potential is reset by subtracting the threshold:

$$\text{if } V(t+1) \geq \theta, \text{ then } V(t+1) = V(t+1) - \theta \quad (3.5)$$

To avoid immature spike generation instead of resetting to zero, we reset membrane potential through subtraction of the threshold value. To further improve energy and computational efficiency of the IF neuron model, a gating mechanism was embedded in it to prune zero-value post-synaptic spikes.

### **3.3.4 Parameters quantization**

While weights and biases are determined through training, the quantization resolution used for storing and performing computations with them is crucial for the network’s hardware implementation. Figure 3.5 illustrates the design trade-off involved in selecting the quantization resolution. Despite starting out as 32-bit floating-point numbers, lowering the resolution to 10-bit fixed-point values has been shown to minimally affect seizure detection performance. The range of weight and bias values, which are typically determined during training, does not necessitate the precision provided by higher resolutions. The resolution at which input signals are quantized aligns closely with the chosen 10-bit resolution for weights and biases. This coherence ensures that higher resolutions would not significantly enhance performance. Therefore, opting for higher resolutions in representing weights and biases would exceed practical needs, as the precision provided by 10-bit fixed-point values is adequate for effectively capturing and processing neural signals in our seizure detection application.



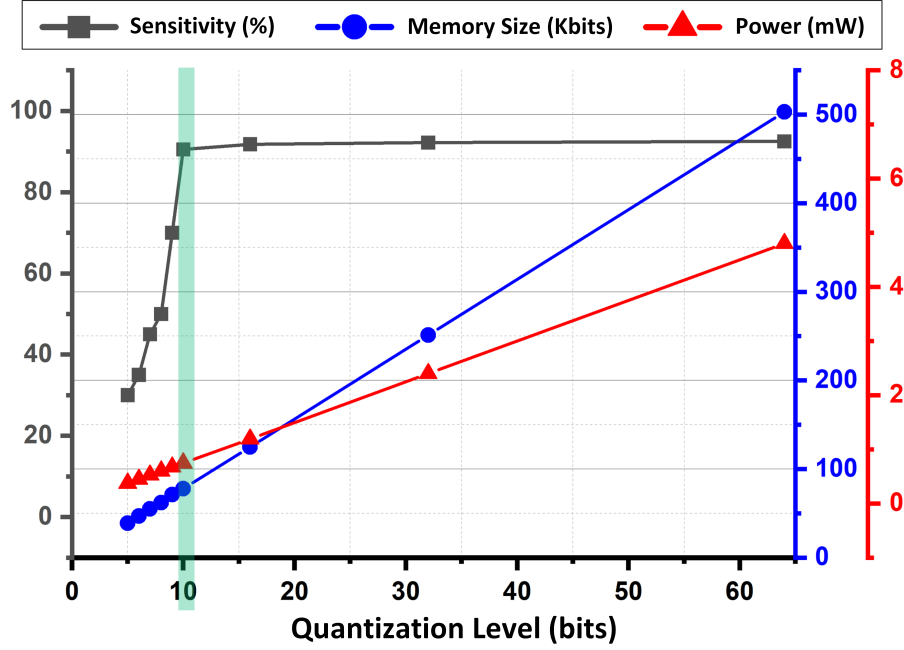
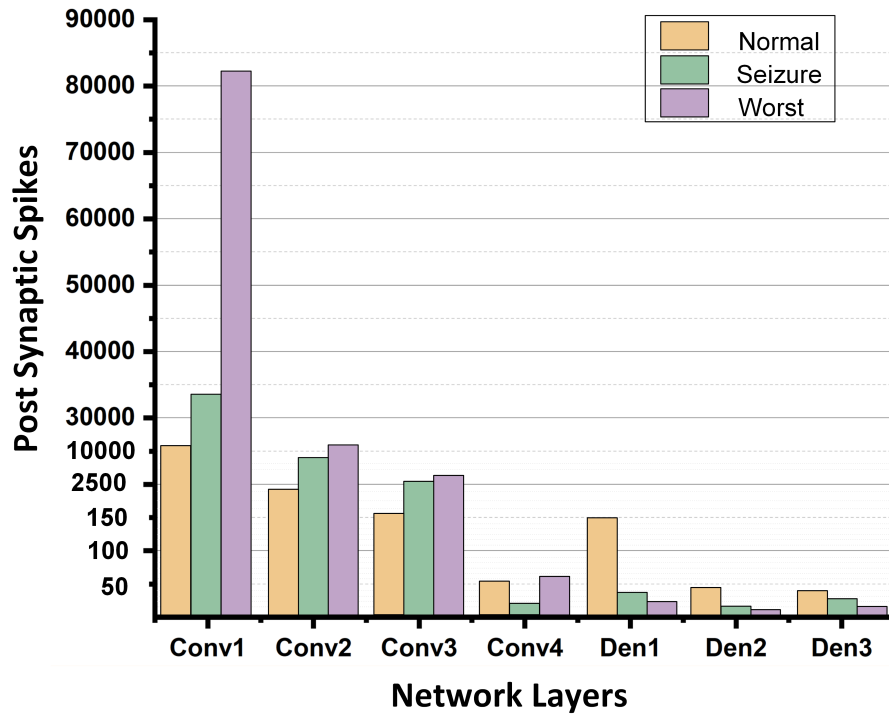


Figure 3.5: Seizure detection sensitivity, SNN model size, and power consumption as a function of quantization resolution used for weights and bias values.

### 3.4 Model testing

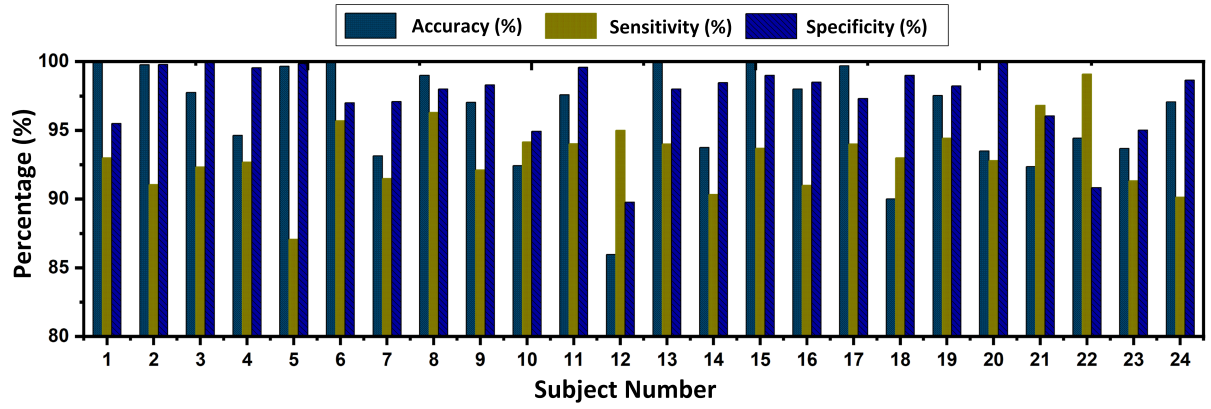
We utilized the open-source CHB-MIT dataset for both training and testing purposes. This dataset comprises scalp EEG recordings from 24 patients aged between 1.5 to 22 years, sampled at 256Hz [77]. Figure 3.6 illustrates the average spike counts per ten seconds of data at the output of each layer under normal conditions, during seizure episodes, and in the most active input dynamics scenario. The data validates our earlier discussion that spikes tend to become sparser as they propagate through the network, resulting in reduced computational demands in intermediate and deeper layers.

Moreover, the figure underscores the importance of employing a fully spiking network and implementing spike encoding before the first layer, especially considering its substantial compu-



**Figure 3.6: Number of Post synaptic spikes at the output of each layer of the network in different situations.**

tational workload compared to subsequent layers. This approach not only optimizes computational efficiency but also enhances the overall performance of the neural network in processing EEG data for seizure detection. For each patient in our study, we allocated 60% of the data for training, 10% for cross-validation, and 30% for testing purposes. Our primary focus at this stage was on evaluating the sensitivity and specificity of seizure detection, following the definitions provided in [61]. To assess the performance of our model, we employed vector-based metrics as outlined in previous chapter. Figure 3.7 displays subject-specific performance metrics for the SCNN, illustrating an average sensitivity of 92.2% and specificity of 97.3%. These metrics underscore the robustness and reliability of the SCNN in accurately detecting seizures across the



**Figure 3.7: Subject-wise seizure detection performance of the custom-developed SCNN.**

tested subjects.

In essence, these evaluations highlight the effectiveness of our approach in achieving high accuracy and reliability in seizure detection, crucial for its potential application in clinical settings.

## **Chapter 4**

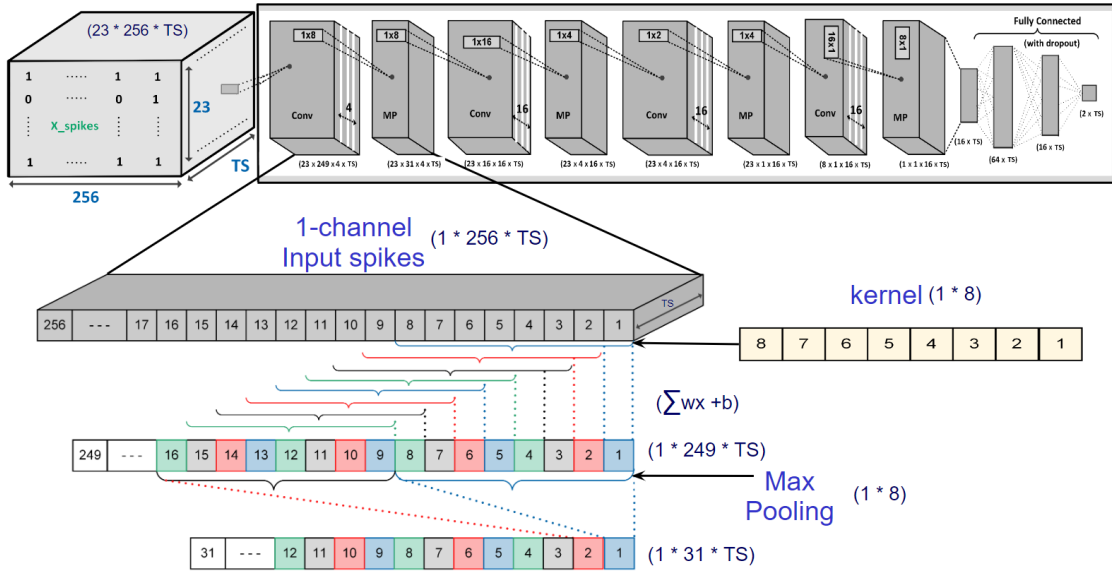
# **Digital back-end hardware implementation of FSCNN**

### **4.1 Introduction**

In the previous chapter, a behavioural system-level implementation of the proposed multi-channel, patient-specific fully spiking neural network was described. In this chapter, we will discuss the hardware implementation of the digital back-end, the physical implementation of the design in a standard CMOS sub-micron process, and the developed testbench and simulation results. The digital blocks in this system were first implemented in register transfer level (RTL) using Verilog hardware description language (HDL). Next, using Cadence Genus, they were synthesized and optimized for gate-level implementation. In the following sections, the implementation of each of these blocks is described and their simulation results are shown.

## 4.2 Block-level Implementation

Figure 4.1 depicts the top-level block diagram of the two layers of proposed spiking convolutional neural network. The hardware implementation details, simulation and synthesis results of each block are discussed in the next subsections.



**Figure 4.1: Block level illustration of processing in SCNN layers.**

### 4.2.1 Input register

Considering the integration of proposed SCNN with spiking front-end LC-ADC, an sterilizer is required to serialize the input data bits and store into the input shift register of size  $8 \times 50$  bits. A 14-bit counter is also placed there to keep the track of 1-sec input window. The 400 bits size of input register is required because the first convolution kernel size is  $1 \times 8$  and the time steps are 50 so once the system get this much of the data in an input register from LC-ADC then

it triggers the `data_valid` signal high and consider incoming data as a valid data to start spiking convolution operation. Figure 4.2 shows the hardware implementation of input shift register along with sterilizer block for a network.

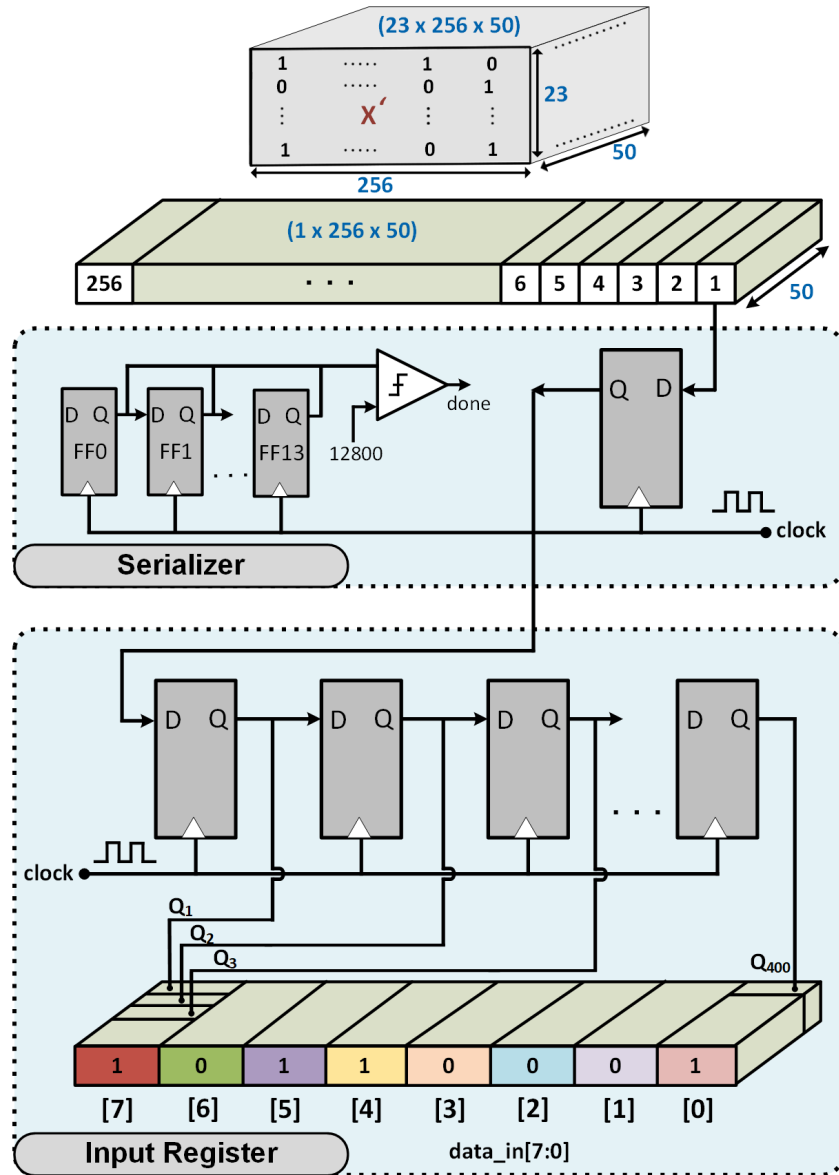


Figure 4.2: Input to a system that pass from the serializer and then stored into the input shift register.

### **4.2.2 Spiking convolution layer**

As compared to the conventional continuous-time convolution layer a spiking convolution layer is composed of select multiplexers and adders. The multiplexers are responsible to forward the corresponding weight value if select pin of multiplexer get 1 from the input layer of previous layer. But if there is 0 at the input of multiplexer then it will send 0 instead of 10-bit weight value and the next adder block will skip the addition operation and eventually save the extra 10-bit addition operation. This leads to the power saving by utilizing sparse information from EEG data. A series of series of multiplexes look at the input bit at each time step and send the corresponding weight to an accumulator of neuron model. Figure 4.3 shows more clearly the described functionality.

### **4.2.3 Neuron model**

An integrate and fire neuron model is implemented in hardware using accumulation unit before the comparator. The accumulation unit do summations of weights that are forwarded by the select unit of convolution layer. The accumulated output at each time step is compared with the defined threshold value. If accumulated membrane potential value is equal or more than the threshold value then send spike to next layer and subtract the threshold value form membrane potential. Threshold value is chosen 1000 in this case for the simplicity of hardware implementation in 10-bit integers values. Figure 4.4, shows the hardware implementation of IF neuron model with accumulator, comparator, and subtract blocks.

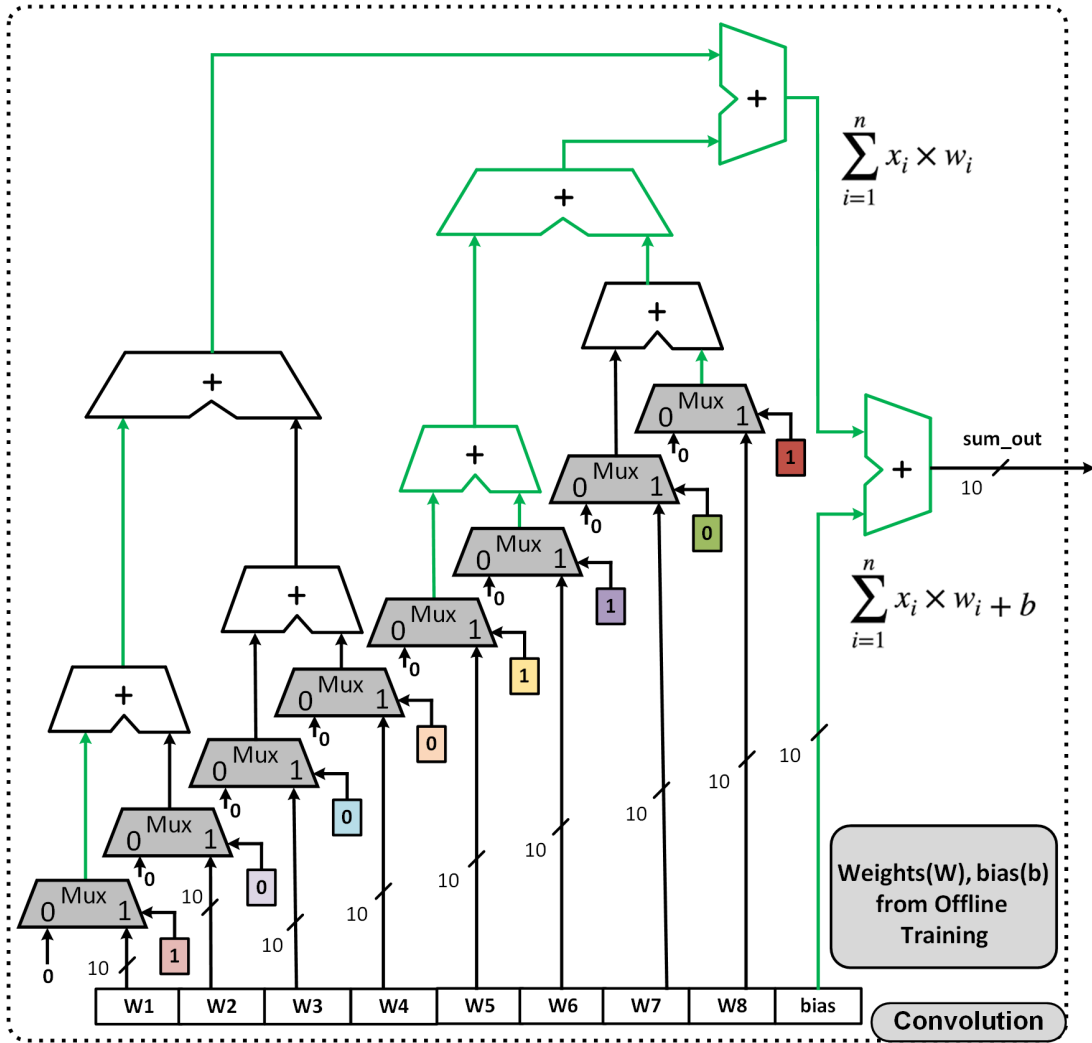
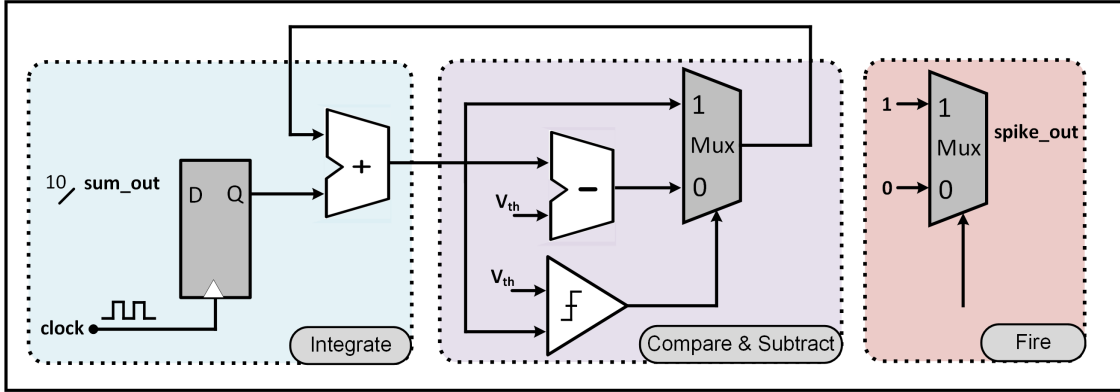


Figure 4.3: Spiking convolution layer composed of select multiplexers and sign adder block.





**Figure 4.4: Digital blocks of integrate and fire neuron model.**

#### 4.2.4 ReLU function

In spiking convolutional layers of neuromorphic processor, the Rectified Linear Unit (ReLU) activation function is employed to introduce non-linearity, which is essential for learning complex patterns in the input data. ReLU, defined as:

$$f(x) = \max(0, x) \quad (4.1)$$

It ensures that only positive signals are propagated forward, effectively mimicking the behavior of biological neurons that fire only when their membrane potential exceeds a certain threshold. This property of ReLU helps in maintaining sparsity in the activations, reducing computational complexity and enhancing the efficiency of the network. The introduction of non-linearity by ReLU in spiking convolutional layers allows for the representation of more intricate features, which is crucial for tasks such as EEG-based seizure detection in neuromorphic processors. The hardware implementation of ReLU function is done by using one multiplexer that checks the sign it at the select pin and send zero or same data(if positive) at the output. Figure 4.5 shows

the simplified hardware implementation of ReLU function.

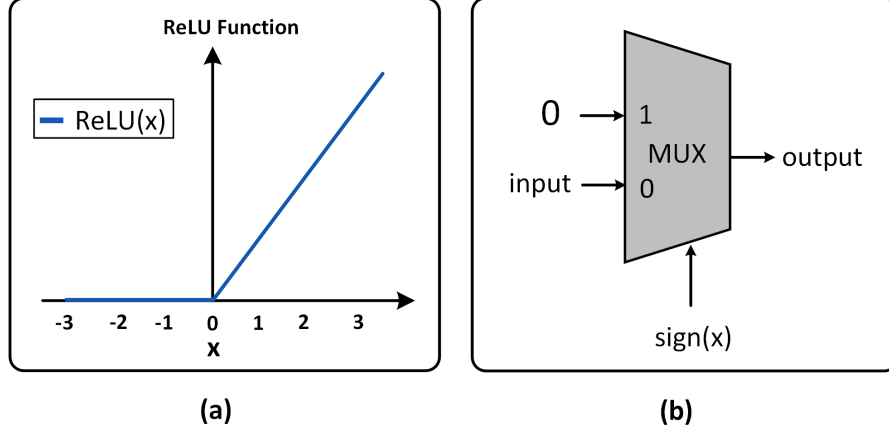


Figure 4.5: Implementation of the ReLU function on hardware

### 4.2.5 Max pooling layer

In proposed fully spiking convolutional neural network, all the convolution layers are followed by max-pooling layer. The basic functionality of max pooling is filter out the data with maximum information, means it will forward the neuron with maximum value from pooling window (i.e.  $1 \times 8$  pooling window size, means from the 8 neurons) to the next layer. In FSCNN it is implemented in a way to count number of spikes from each of the neuron and send the spikes (0,1) data of the neuron which has maximum spiking value. The block-level hardware implementation of spiking max-pooling layer is shown in Figure 4.6.

### 4.2.6 System Integration

Figure 4.7 shows the integration of spike convolution, spike max-pooling, and spiking integrate and fire neuron model. It also shows the extension of logic unit for multi-kernels and multi-

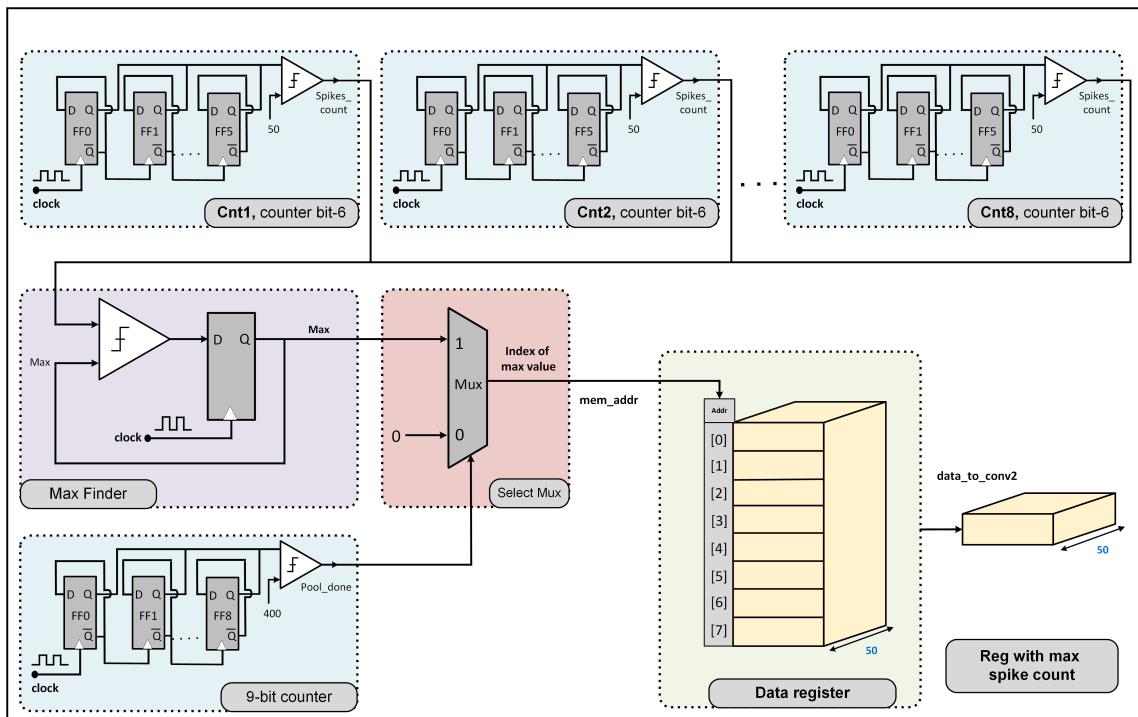


Figure 4.6: Hardware implementation of the spiking max-pooling.

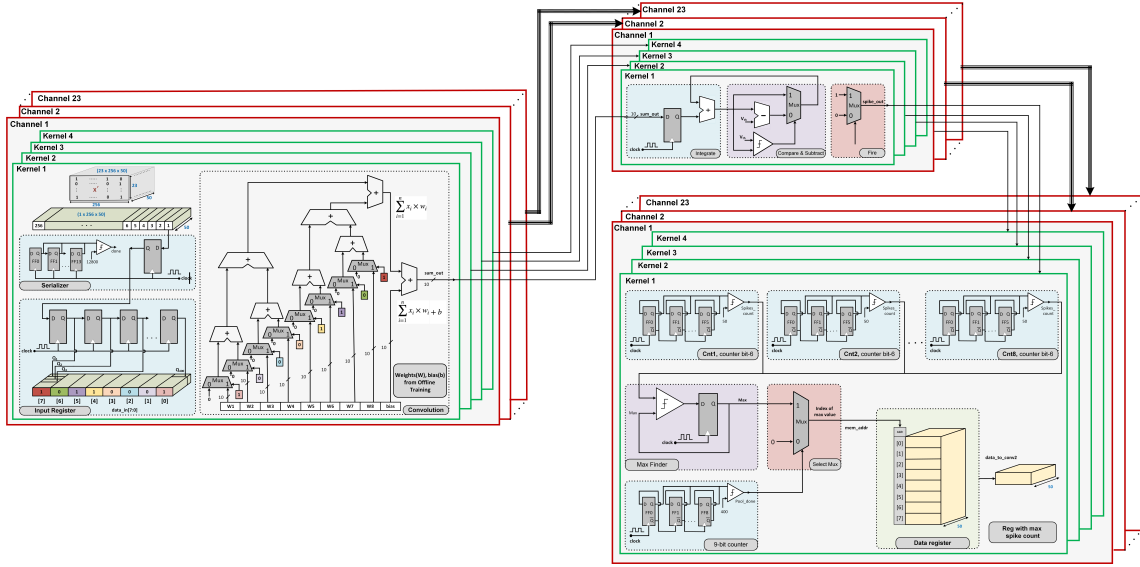


Figure 4.7: Integration of the fundamental layers in hardware.

channels system.

Figure 4.8 shows the overall system implementation, and integration of proposed SCNN architecture in hardware.

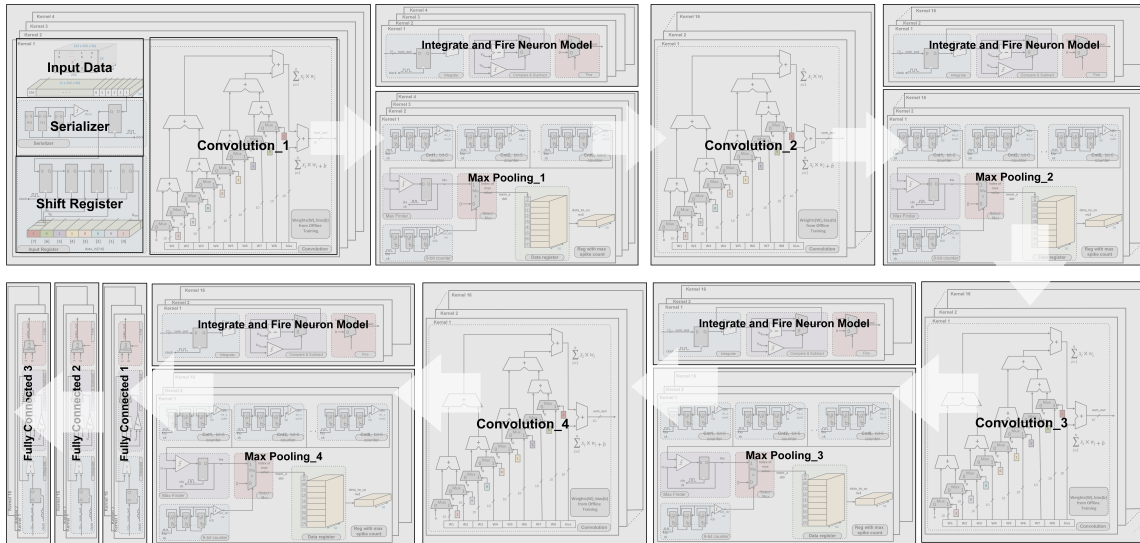
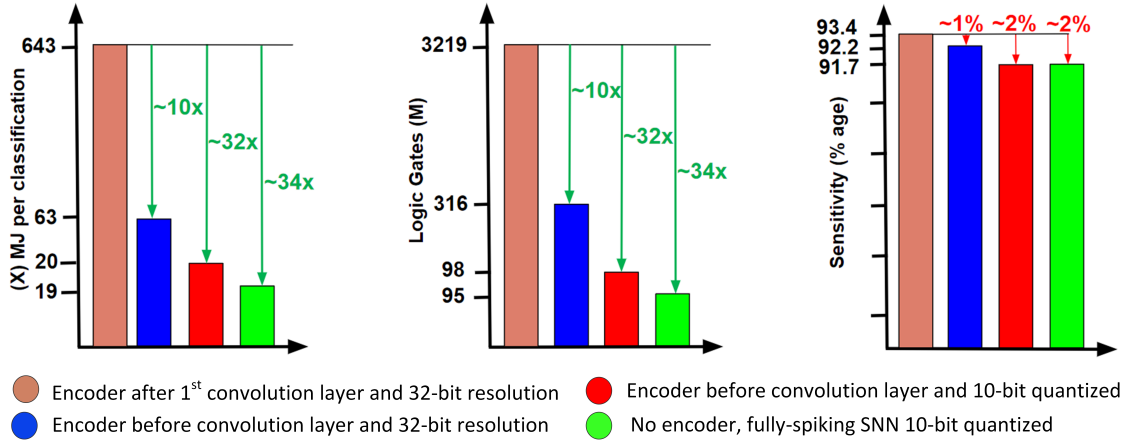


Figure 4.8: System Integration of the spiking hardware for the proposed SCNN architecture.

## 4.2.7 Hardware Implementation Evaluation

Figure 4.9 summarizes the comparative evaluation of four different hardware implementations in terms of energy efficiency, gate count, and sensitivity. The implementation where the encoder is placed after the first convolution layer with 32-bit resolution consumes the most energy and requires the largest number of logic gates (3,219 million), though it achieves the highest sensitivity (93.4%). Moving the encoder before the convolution layer with the same 32-bit resolution reduces energy consumption and logic gates by approximately 10 $\times$ , with only a slight drop in sensitivity (1%). Introducing 10-bit quantization in the encoder further reduces energy consumption and logic gates by 32 $\times$ , though sensitivity decreases marginally (2%). The fully spiking neural network (SNN) implementation with 10-bit quantization achieves the highest efficiency, with a 34 $\times$  reduction in both energy consumption and logic gates compared to the baseline. Despite these optimizations, the sensitivity remains comparable at 91.7%, demonstrating that spiking architectures and quantization techniques can significantly enhance resource efficiency with minimal performance trade-offs.



**Figure 4.9: System Integration of the spiking hardware**

### 4.3 Simulation results

The behavioral simulation of verilog HDL design is done using Xilinx, Vivado. The spiking input of a chb-02's seizure window is fed to the system by test bench and weights and biases of subject chb-02 are used to verify the functionality of a system. Figure 4.10 shows that once the value of reset signal is 0 and value of counter num\_neuron reaches to 8 (kernel size of 1st convolution layer is  $1 \times 8$ ) which means that the input data is ready for further processing, it inserts the data\_valid signal 1. Once the data\_valid signal is high, it triggers the input buffer register to start sending data to the 1st convolution layer as shown in Figure 4.10 at timesetp 406ns of simulation window.

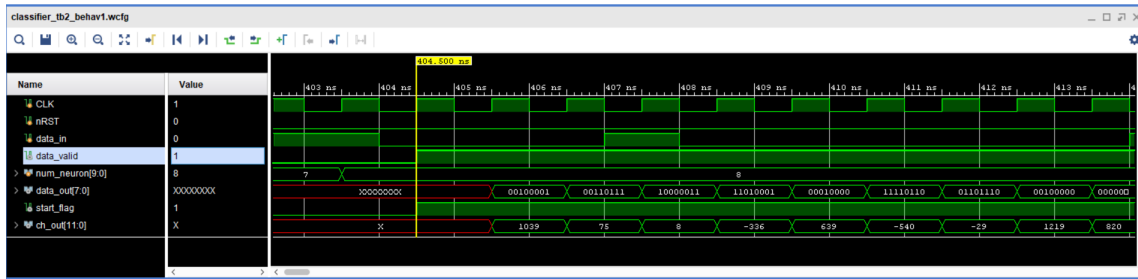
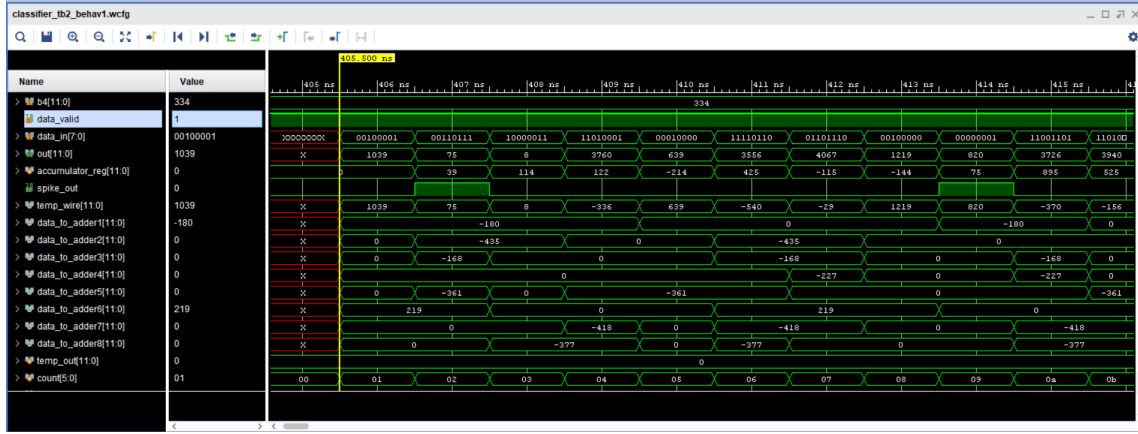


Figure 4.10: Simulation results for data input register.

Figure 4.11 shows the spiking convolution functionality of spiking convolution layer along with integrate and fire neuron. Here data\_in is the 8-bit input data going to the select block of the 1st convolution layer to see, if the input bit is 1 then corresponding weight value is forwarded to a summation block to add up together to make the convolution operation. 12-bit out signal shows the value of summation block of convolution layer. The summed output is forwarded to IF neuron model to get integrated into a membrane potential, an accumulator\_reg register shown in signal list. Once the accumulator register's value become more or equal to the threshold value (1000 in this case) it will generates the spike 1 named as spike\_out for the next layer

and subtracts the threshold value from the membrane potential.



**Figure 4.11: Simulation results for seizure activity detection at first spiking convolution layer.**

Figure 4.12 shows the functionality of spiking max pooling layer. `cnt1`, `cnt2`, ... `cnt8` signals are the 8 counters because the pooling window size of 1st Max-pooling layer is  $1 \times 8$  which means it has to choose maximum spiking neuron from 8 of neurons. So each of these counters count the number of spikes generated by the convolution layer and when 9-bit counter `counter_pool` reaches to value 400, which shows that all the respective input is processed for this window then it finds the max `cnt (x)` from above 8 counters and forward the spike data (0,1) of that neuron to 50-bit `data_to_conv2` register and insert the `pool_done_l1` signal 1. So that it indicates next layer to get activated for the forwarded data processing.

Figure 4.13 shows the data received into the subsequent convolution layer named as `data_to_conv2` and the stored into a buffer register named as `mem_conv2_in` in which once it reaches to the value that is needed by 2nd convolution layer's kernel to start, it will assert `data_valid_l2` signal to 1 and starts operation in 2nd convolution layer.

In this way the rest of subsequent convolution, max-pooling layers work and generate the final spike output to indicate the seizure or non-seizure activity detected from EEG signal.

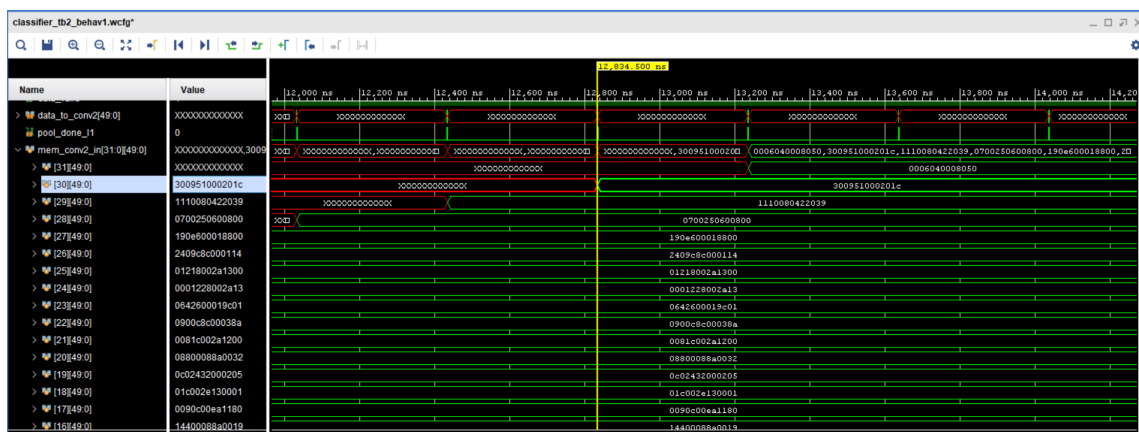
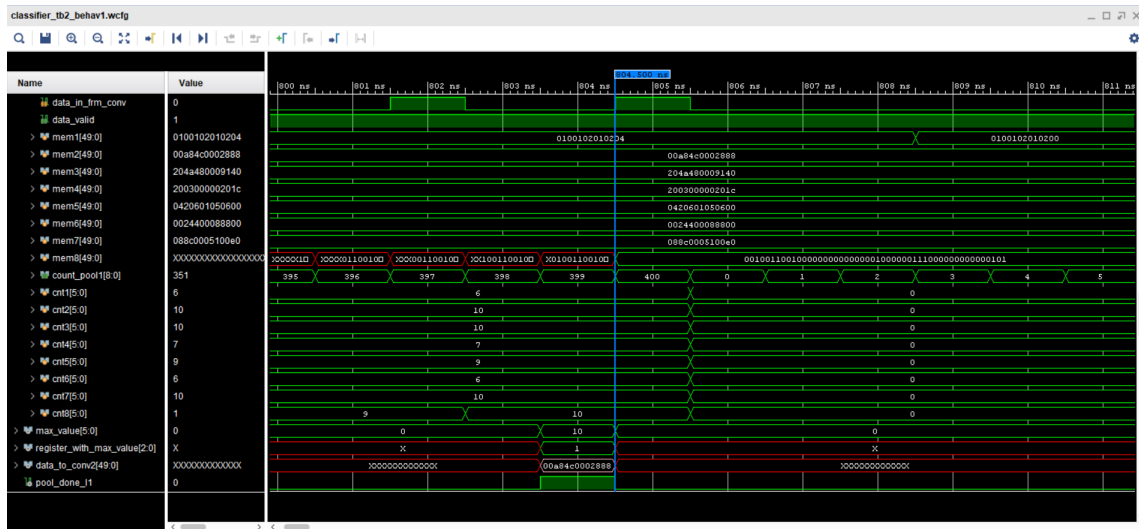




Table 4.1: SCNN Parameters and Synthesis Results

| Parameters   |                                 | Synthesis Report        |                                  |
|--------------|---------------------------------|-------------------------|----------------------------------|
| Memory       | 9.75KB                          | Process                 | Standard 0.13 $\mu\text{m}$ CMOS |
| Cell count   | 78.9K                           | Voltage                 | 1.8 V                            |
| Time Steps   | 50                              | Total Power             | 1.37mW                           |
| Neuron Model | Integrate and Fire              | Total Area              | 3.3mm <sup>2</sup>               |
| Thresholding | Thresh: 1, Reset by subtraction | Quantization Resolution | 10-bits                          |

## 4.4 Synthesis results

The developed Spiking Convolutional Neural Network (SCNN) is converted into a hardware description language, specifically Verilog Register-Transfer Level (RTL). This conversion allows the SCNN to be implemented on hardware, making it suitable for deployment on actual devices. The synthesized hardware was created using a standard 130nm CMOS (Complementary Metal-Oxide-Semiconductor) cell library.

The synthesis process was conducted using Cadence Genus, an advanced tool for logic synthesis that translates the high-level Verilog RTL design into a detailed gate-level netlist, optimized for performance, power, and area. The results of this synthesis process are summarized in Table 4.1. This table provides detailed metrics such as area, power consumption, and timing performance, giving a clear picture of the hardware efficiency and feasibility of the SCNN design.

Furthermore, Table 4.2 presents a comparative analysis between the fully spiking SCNN developed in this work and other existing SCNN and CNN architectures from the state of the art. This comparison highlights the superior performance of the presented design in terms of various parameters such as energy efficiency, computational cost, memory usage, and classification

Table 4.2: Comparison with the state-of-the-art patient-specific seizure detection algorithms

| Ref                |        | BSPC'23<br>[84] | ISCAS'24<br>[82]    | TBME'22<br>[85] | JSSC'22<br>[64] | This<br>Work [65]    |                                         |
|--------------------|--------|-----------------|---------------------|-----------------|-----------------|----------------------|-----------------------------------------|
| Classifier         |        | SNN+2DLSVM      | Spiking Conformer   | CNN             | GTCA-SVM        | CNN                  | SCNN                                    |
| HW Imp             |        | ASIC            | ASIC                | N/A             | ASIC            | STM32H747            | ASIC                                    |
| No. of Parameter   |        | 14.9K           | 9.9K                | N/A             | N/A             | 7.8K                 | 7.8K                                    |
| Model Size         |        | 59.6KB          | 39.6KB              | 45.2KB          | 70KB            | 31.2KB               | 9.75KB                                  |
| No. of Operations  |        | 6K ACC          | 0.32M ADD, 1.0K MUL | -               | N/A             | 0.65M ACC, 0.66M MUL | $1.5M^{\oplus}$ , $0.2M^{\diamond}$ ACC |
| Feature Extraction |        | Power Spectrum  | Raw EEG             | N/A             | Raw EEG         | Raw EEG              | Raw EEG                                 |
| Sensitivity<br>(%) | Vector | N/R             | N/R                 | N/R             | 97.8            | 93.4                 | 92.2                                    |
|                    | Event  | 88.4            | 94.9                | 93.4            | 100             | 100                  | 100                                     |
| Specificity/FPR    |        | 84.6%           | 99.3%               | 0.27/hr         | 99.5%           | 99.1%                | 97.3%                                   |
| Time steps         |        | 200             | 8                   | N/R             | N/R             | N/A                  | 50                                      |

N/R: Not Reported, N/A: Not Applicable

MUL: Multiplication, ACC: Accumulation  $\oplus$ : Avg. for Seizure,  $\diamond$ : Avg. for Non-seizure

accuracy. The excellent overall performance of the presented design demonstrates its potential advantages for practical applications in neuromorphic computing and energy-efficient seizure detection.

# Chapter 5

## Conclusion and future works

### 5.1 Conclusion

Design, optimization, hardware implementation, and performance evaluation of Convolutional Neural Networks (CNNs) and Spiking Convolutional Neural Networks (SCNNs) for patient-specific seizure detection and neurological disorder diagnostics were presented. A conclusive review and analysis for various design considerations and optimizations for crucial parameters such as architecture and kernel size for CNNs, and spike encoding and time steps for SCNNs were conducted to achieve a significant leap in performance and energy efficiency.

The design phase focused on creating a robust and efficient seizure detection system tailored to the unique requirements of EEG signal processing. For the CNN, a neural architecture search is employed to determine the most effective network configuration, resulting in a 7-layer structure with one-dimensional kernels. This setup proved highly capable of capturing spatio-temporal

features from the input data. For the SCNN, shadow training was used to convert the CNN model into a spiking version, preserving the high fidelity of the original model. This conversion process was optimized to minimize the loss in sensitivity and specificity.

The hardware implementation phase involved translating the developed SCNN into a hardware description language (Verilog RTL) and synthesizing it using a standard 130nm CMOS cell library. This step was critical in demonstrating the feasibility of the SCNN for real-world applications, particularly in resource-constrained environments. Using Cadence Genus for synthesis, we obtained detailed metrics that highlighted the SCNN's superior performance in terms of area, power consumption, and timing.

Performance evaluation using the CHB-MIT dataset revealed that the SCNN achieved an average sensitivity of 83.02% and specificity of 86.31%, with a false positive rate of 0.69/hr for cross-patient seizure detection. These results are comparable to traditional CNNs but with significantly enhanced energy efficiency. The SCNN demonstrated an order of magnitude ( $12.1\times$ ) improvement in Latency $\times$ Energy Efficiency compared to the best prior art, underscoring its potential for practical neuromorphic applications.

In addition to the SCNN, a fully customized approach is developed for fully-spiking neural networks (SNNs) aimed at energy-efficient data-driven signal processing in implantable brain neural interfaces. This approach allowed for comprehensive customization of key parameters, including the quantization resolution of weights and biases, encoding schemes, encoder placement, temporal resolution, neuron types, and internal parameters such as threshold value, resetting process, and refractory period.

The SNNs were synthesized using Verilog RTL on FPGA and subsequently for ASIC imple-

mentation. The synthesis reports provided valuable insights into the memory, computation, and energy requirements, highlighting the efficiency gains achieved through our customized design. Performance evaluation using the CHB-MIT dataset showed an average sensitivity of 92.2% and specificity of 97.3% for seizure detection, confirming the high classification performance of our SNNs while significantly enhancing hardware efficiency.

The key contributions of this work lie in the demonstration of significant advancements in neuromorphic computing for seizure detection. The SCNN's energy efficiency and performance metrics make it a viable candidate for deployment in wearable and implantable devices, where power consumption is a critical constraint. The customizable SNNs further extend these benefits by providing tailored solutions that optimize hardware and energy efficiency without compromising on detection accuracy. Future research could focus on further refining these models and exploring their applications in other areas of biomedical signal processing. Additionally, advancements in neuromorphic hardware platforms could be leveraged to push the boundaries of what is achievable with SCNNs and SNNs, potentially leading to even greater improvements in energy efficiency and computational performance.

## **5.2 Future directions**

In the short term, future work will focus on enhancing the accuracy and energy efficiency of the developed convolutional spiking neural network (SNN) for epilepsy seizure detection. This includes a thorough investigation of different encoding schemes, such as time-to-first-spike encoding, and the implementation of various neuron models, like different variations of the leaky integrate-and-fire model. Additionally, comprehensive system testing of the hardware implemen-

tation on an FPGA will be conducted using the entire dataset. This process will identify necessary design adjustments and tuning. Moreover, hardware implementation will be optimized through techniques like pipelining and resource sharing. The final step in the short-term future work involves implementing the design as an ASIC and integrating it with an array of stimulation and recording circuits. For recording, LC-ADCs will be used to provide spiking data to be processed by the SNN.

In the long term, the focus will shift towards testing the developed chip in animal models of epilepsy to assess its efficacy in a biological setting. Following successful animal testing, the goal is to advance to clinical testing phases. This will involve the development of a diagnostic medical device, potentially leading to its commercialization. Through these efforts, the ultimate aim is to create a reliable and effective tool for epilepsy seizure detection that can significantly improve patient outcomes and enhance the quality of life for those affected by epilepsy.

# References

- [1] P. O. Shafer, J. I. Sirven, “Facts about Seizures and Epilepsy” 2019.[online]. Available: <https://www.epilepsy.com/learn/about-epilepsybasics/facts-about-seizures-and-epilepsy>.
- [2] A. Dabbaghian, et al., “A 9.2-g Fully-Flexible Wireless Ambulatory EEG Monitoring and Diagnostics Headband With Analog Motion Artifact Detection and Compensation,” *IEEE Transactions on Biomedical Circuits and Systems*, vol. 13, no. 6, pp. 1141-1151, Dec. 2019.
- [3] R. S. Fisher et al., “The impact of epilepsy from the patient’s perspective, descriptions and subjective perceptions,” *Epilepsy research*, vol. 41, no. 1, pp. 39–51, 2000.
- [4] L. Modglin, A. Jacobson, “Epilepsy statistics” 2022. [online]. Available: <https://www.singlecare.com/blog/news/epilepsy-statistics>
- [5] R. Fisher et al., “Epileptic seizures and epilepsy: Definitions proposed by the International League Against Epilepsy (ILAE) and the International Bureau for Epilepsy (IBE),” *Epilepsia*, vol. 46, no. 4, pp. 470–472, 2005
- [6] J. Hopkins, Health System, “Type of Seizures,”[online]. Available: <https://www.hopkinsmedicine.org/health/conditions-anddiseases/epilepsy/types-of-seizures>.

- [7] P. Kwan and M. J. Brodie, "Definition of refractory epilepsy: defining the indefinable?," *The Lancet. Neurology*, vol. 9, no. 1, pp. 27-9, Jan 2010.
- [8] P. Vespa, "Continuous EEG monitoring for the detection of seizures in traumatic brain injury, infarction, and intracerebral hemorrhage: 'To detect and protect,'" *J. Clin. Neurophysiol.*, vol. 22, no. 2, pp. 99–106, 2005.
- [9] R. Nickel et al., "Quality of life issues and occupational performance of persons with epilepsy," *Arquivos de neuro-psiquiatria*, vol. 70, no. 2, pp. 140–144, 2012.
- [10] A. M. Lozano et al., "Deep brain stimulation: current challenges and future directions," *Nat. Rev. Neurol.*, vol. 15, no. 3, pp. 148-160, 2019.
- [11] T. L. Skarpaas and M. J. Morrell, "Intracranial stimulation therapy for epilepsy," *Neurotherapeutics*, vol. 6, pp. 238-243, 2009.
- [12] W. M. M. Schuepbach et al., "Neurostimulation for Parkinson's disease with early motor complications," *N. Engl. J. Med.*, vol. 368, no. 7, pp. 610-622, 2013.
- [13] H Kassiri, A Muneeb, R Salahi, A Dabbaghian, " Closed-Loop Implantable Neurostimulators for Individualized Treatment of Intractable Epilepsy: A Review of Recent Developments, Ongoing Challenges, and Future Opportunities," *IEEE Transactions on Biomedical Circuits and Systems*, 2024.
- [14] N. Bruggemann et al., "Short-and long-term outcome of chronic " pallidal neurostimulation in monogenic isolated dystonia," *Neurology*, vol. 84, no. 9, pp. 895-903, 2015.



- [15] S. Groppa et al., “Physiological and anatomical decomposition of subthalamic neurostimulation effects in essential tremor,” *Brain*, vol. 137, no. 1, pp. 109-121, 2014.
- [16] V. Shah et al., “The temple university hospital seizure detection corpus,” *Front. Neuroinform.*, vol. 12, p. 83, 2018.
- [17] P. Karoly et al., “Melbourne NeuroVista Seizure Prediction Trial,” *The University of Melbourne, Dataset*, 2018. doi: 10.26188/5b6a999fa2316.
- [18] S. N. Baldassano et al., “Crowdsourcing seizure detection: algorithm development and validation on human implanted device recordings,” *Brain*, vol. 140, pp. 1680–1691, 2017.
- [19] B. H. Brinkmann et al., “Crowdsourcing reproducible seizure forecasting in human and canine epilepsy,” *Brain*, vol. 139, pp. 1713–1722, 2016.
- [20] N. Stevenson et al., “A dataset of neonatal EEG recordings with seizure annotations,” *Sci. Data*, vol. 6, pp. 1–8, 2019.
- [21] P. Detti et al., “EEG synchronization analysis for seizure prediction: a study on data of noninvasive recordings,” *Processes*, vol. 8, p. 846, 2020.
- [22] J. Gutttag, “CHB-MIT Scalp EEG Database (version 1.0.0),” *PhysioNet*, 2010. doi: 10.13026/C2K01R.
- [23] M. R. Karimi et al., “A multi-feature nonlinear-SVM seizure detection algorithm with patient-specific channel selection and feature customization,” in 2020 *IEEE International Symposium on Circuits and Systems (ISCAS)*, 2020, pp. 1-5.

- [24] T. Zhan et al., “A resource-optimized VLSI implementation of a patient-specific seizure detection algorithm on a custom-made 2.2 cm<sup>2</sup> wireless device for ambulatory epilepsy diagnostics,” *IEEE Trans. Biomed. Circuits Syst.*, vol. 13, no. 6, pp. 1175-1185, 2019.
- [25] H. Peng et al., “Feature selection based on mutual information criteria of max-dependency, max-relevance, and minredundancy,” *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 27, no. 8, pp. 1226-1238, 2005.
- [26] D. Kaya, “The mRMR-CNN based influential support decision system approach to classify EEG signals,” *Meas.*, vol. 156, p. 107602, 2020
- [27] E. B. Assi et al., “A hybrid mRMR-genetic based selection method for the prediction of epileptic seizures,” in *Proc. 2015 IEEE Biomed. Circuits Syst. Conf. (BioCAS)*, 2015, pp. 1-4.
- [28] M. Bandarabadi et al., “Epileptic seizure prediction based on a bivariate spectral power methodology,” in *Proc. 2012 Annu. Int. Conf. IEEE Eng. Med. Biol. Soc.*, 2012, pp. 5943-5946.
- [29] M. Bandarabadi et al., “Epileptic seizure prediction using relative spectral power features,” *Clin. Neurophysiol.*, vol. 126, no. 2, pp. 237-248, 2015.
- [30] B. Direito et al., “Optimized feature subsets for epileptic seizure prediction studies,” in *Proc. 2011 Annu. Int. Conf. IEEE Eng. Med. Biol. Soc.*, 2011, pp. 1636-1639.
- [31] K. Abdelhalim et al., “Phase-synchronization early epileptic seizure detector VLSI architecture,” *IEEE Trans. Biomed. Circuits Syst.*, vol. 5, no. 5, pp. 430-438, 2011.

- [32] K. Abdelhalim et al., “64-channel UWB wireless neural vector analyzer SOC with a closed-loop phase synchrony-triggered neurostimulator,” *IEEE J. Solid-State Circuits*, vol. 48, no. 10, pp. 2494-2510, 2013.
- [33] R. Hopfengartner et al., “Automatic seizure detection in longterm scalp EEG using an adaptive thresholding technique: a validation study for clinical routine,” *Clin. Neurophysiol.*, vol. 125, no. 7, pp. 1346-1352, 2014.
- [34] H. Kassiri et al., “Inductively-powered direct-coupled 64- channel chopper-stabilized epilepsy-responsive neurostimulator with digital offset cancellation and tri-band radio,” in *Proc. ESSCIRC 2014-40th Eur. Solid-State Circuits Conf.*, 2014, pp. 95-98.
- [35] M. A. B. Altaf et al., “A 1.52  $\mu$ J/classification patient-specific seizure classification processor using Linear SVM,” in *Proc. 2013 IEEE Int. Symp. Circuits Syst. (ISCAS)*, 2013, pp. 849-852.
- [36] M. A. B. Altaf et al., “A 1.83  $\mu$ J/classification, 8-channel, patient-specific epileptic seizure classification SoC using a nonlinear support vector machine,” *IEEE Trans. Biomed. Circuits Syst.*, vol. 10, no. 1, pp. 49-60, 2015.
- [37] S.-A. Huang et al., “A 1.9-mW SVM processor with on-chip active learning for epileptic seizure control,” *IEEE J. Solid-State Circuits*, vol. 55, no. 2, pp. 452-464, 2019.
- [38] K. H. Lee et al., “A low-power processor with configurable embedded machine-learning accelerators for high-order and adaptive analysis of medical-sensor signals,” *IEEE J. Solid-State Circuits*, vol. 48, no. 7, pp. 1625-1637, 2013.

- [39] M. Zhang et al., “A patient-specific closed-loop epilepsy management SoC with one-shot learning and online tuning,” *IEEE J. Solid-State Circuits*, vol. 57, no. 4, pp. 1049-1060, 2022.
- [40] Y. Liu et al., “Automatic seizure detection using wavelet transform and SVM in long-term intracranial EEG,” *IEEE Trans. Neural Syst. Rehabil. Eng.*, vol. 20, no. 6, pp. 749-755, 2012.
- [41] P. W. Mirowski et al., “Comparing SVM and convolutional networks for epileptic seizure prediction from intracranial EEG,” in *Proc. 2008 IEEE Workshop Mach. Learn. Signal Process.*, 2008, pp. 244-249.
- [42] T. Zhang et al., “LMD based features for the automatic seizure detection of EEG signals using SVM,” *IEEE Trans. Neural Syst. Rehabil. Eng.*, vol. 25, no. 8, pp. 1100-1108, 2016.
- [43] J. Yoo et al., “An 8-channel scalable eeg acquisition soc with patient-specific seizure classification and recording processor,” *IEEE J. Solid-State Circuits*, vol. 48, pp. 214–228, Jan. 2013.
- [44] M. A. Bin Altaf and J. Yoo, “A 1.83  $\mu$ J/Classification, 8-Channel, Patient-Specific Epileptic Seizure Classification SoC Using a Non-Linear Support Vector Machine,” *IEEE Transactions on Biomedical Circuits and Systems*, vol. 10, no. 1, pp. 49-60, Feb. 2016.
- [45] Y. Wang et al., “26.3 a closed-loop neuromodulation chipset with 2-level classification achieving 1.5 v pp cm interference tolerance, 35db stimulation artifact rejection in 0.5 ms and 97.8% sensitivity seizure detection,” in *Proc. IEEE Int. Solid-State Circuits Conf.*, 2020, pp. 406-408.

- [46] G. O’Leary, D. M. Groppe, T. A. Valiante, N. Verma and R. Genov, “NURIP: Neural Interface Processor for Brain-State Classification and Programmable-Waveform Neurostimulation,” *IEEE J. Solid-State Circuits (JSSC)*, vol. 53, no. 11, pp. 3150-3162, Nov. 2018.
- [47] A. Muneeb, M. Ali and M. A. B. Altaf, “A 2.7 $\mu$ J/classification Machine-Learning based Approximate Computing Seizure Detection SoC,” *IEEE International Symposium on Circuits and Systems (ISCAS)*, May, 2022.
- [48] C. A. Teixeira et al., “Epileptic seizure predictors based on computational intelligence techniques: a comparative study with 278 patients,” *Comput. Methods Progr. Biomed.*, vol. 114, pp. 324-336, 2014.
- [49] M. Bandarabadi et al., “Epileptic seizure prediction using relative spectral power features,” *Clin. Neurophysiol.*, vol. 126, no. 2, pp. 237-248, 2015.
- [50] Y. Park et al., “Seizure prediction with spectral power of EEG using cost-sensitive support vector machines,” *Epilepsia*, vol. 52, pp. 1761-1770, 2011.
- [51] M. Shoaran et al., “Energy-efficient classification for resource constrained biomedical applications,” *IEEE J. Emerg. Sel. Topics Circuits Syst.*, vol. 8, no. 4, pp. 693-707, 2018.
- [52] G. O’Leary et al., “26.2 A Neuromorphic Multiplier-Less Bit-Serial Weight-Memory-Optimized 1024-Tree Brain-State Classifier and Neuromodulation SoC with an 8-Channel Noise-Shaping SAR ADC Array,” *IEEE Int. Solid-State Circuits Conf. (ISSCC)*, 2020, pp. 402-404.

- [53] U. Shin et al., "NeuralTree: A 256-channel 0.227- $\mu$ J/class versatile neural activity classification and closed-loop neuromodulation SoC," *IEEE J. Solid-State Circuits*, vol. 57, no. 11, pp. 3243-3257, 2022.
- [54] F. Tian et al., "A new neuromorphic computing approach for epileptic seizure prediction," in *Proc. 2021 IEEE Int. Symp. Circuits Syst. (ISCAS)*, 2021, pp. 1-5
- [55] A. Muneeb, and H. Kassiri, "Energy-Efficient Spiking-CNN-Based Cross-Patient Seizure Detection," *IEEE International Symposium on Circuits and Systems (ISCAS)*, 2022, pp. 1-5.
- [56] X. Liu et al., "An energy-efficient compressed sensing-based encryption scheme for wireless neural recording," *IEEE J. Emerg. Sel. Top. Circuits Syst.*, vol. 11, no. 2, pp. 405-414, 2021
- [57] C. Sun et al., "An Energy Efficient STDP-Based SNN Architecture With On-Chip Learning," *IEEE Transactions on Circuits and Systems*, vol. 69, no. 12, pp. 5147-5158, Dec. 2022.
- [58] Q. Chen, X. Dong, D. Ma and X. Zhu, "A Hardware Accelerator of the Convolutional Spike Neural Network Based on STDP Online Learning," *Conference of Science and Technology for Integrated Circuits (CSTIC)*, 2024.
- [59] J. K. Eshraghian et al., "Training Spiking Neural Networks Using Lessons From Deep Learning," *Proc. of the IEEE*, vol. 111, no. 9, pp. 1016-1054, Sept. 2023.
- [60] S. Hwang and J. Kung, "One-Spike SNN: Single-Spike Phase Coding With Base Manipulation for ANN-to-SNN Conversion Loss Minimization," *IEEE Transactions on Emerging Topics in Computing*, 2024.

- [61] A. Muneeb, S. Mehrotra and H. Kassiri, “A 9.5ms-Latency 6.2 $\mu$ J/Inference Spiking CNN for Patient-Specific Seizure Detection,” *IEEE Biomedical Circuits and Systems Conference (BioCAS)*, Oct. 2023.
- [62] B. Rueckauer, et al., “Conversion of Continuous-Valued Deep Networks to Efficient Event-Driven Networks for Image Classification,” *Frontiers in Neuroscience*, vol. 11, Dec. 2017.
- [63] S. Zhao, J. Yang and M. Sawan, “Energy-Efficient Neural Network for Epileptic Seizure Prediction,” *IEEE Transactions on Biomedical Engineering*, vol. 69, no. 1, pp. 401-411, Jan. 2022.
- [64] M. Zhang, L. Zhang, C. -W. Tsai and J. Yoo, “A Patient-Specific Closed-Loop Epilepsy Management SoC With One-Shot Learning and Online Tuning,” *IEEE J. Solid-State Circuits*, vol. 57, no. 4, pp. 1049-1060, April 2022.
- [65] A. Muneeb, H. Kassiri, “Customized Development and Hardware Optimization of a Fully-Spiking SNN for EEG-Based Seizure Detection,” *IEEE Biomedical Circuits and Systems Conference (BioCAS)*, 2024.
- [66] J. V. Assche and G. Gielen, “Analysis and Design of a 10.4-ENOB 0.92–5.38-  $\mu$  W Event-Driven Level-Crossing ADC With Adaptive Clocking for Time-Sparse Edge Applications,” *IEEE Journal of Solid-State Circuits*, Mar. 2024.
- [67] C. W. Tsai et al., “SciCNN: A 0-Shot-Retraining Patient-Independent Epilepsy-Tracking SoC,” *IEEE International Solid- State Circuits Conference (ISSCC)*, 2023.
- [68] Y. Yang et al., “Neuromorphic deep spiking neural networks for seizure detection,” *Neuromorphic Computing and Engineering*, vol. 3, no. 1, February 2023.

- [69] I. Kiral-Kornek et al., “Epileptic seizure prediction using big data and deep learning: Toward a mobile system,” *EBioMedicine*, vol. 27, pp. 103–111, 2018.
- [70] J. K. Eshraghian et al., “Training Spiking Neural Networks Using Lessons From Deep Learning,” *arXiv:2109.12894v5*, May 2023.
- [71] C. Frenkel and G. Indiveri, “ReckOn: A 28nm Sub-mm<sup>2</sup> Task-Agnostic Spiking Recurrent Neural Network Processor Enabling On-Chip Learning over Second-Long Timescales,” *IEEE International Solid- State Circuits Conference (ISSCC)*, 2022, pp. 1-3.
- [72] C. D. Schuman, et al., “Opportunities for neuromorphic computing algorithms and applications,” *Nat Comput Sci*, vol.2, pp.10–19, 2022.
- [73] S. Zhao, J. Yang and M. Sawan, “Energy-Efficient Neural Network for Epileptic Seizure Prediction,” *IEEE Transactions on Biomedical Engineering*, vol. 69, no. 1, pp. 401-411, Jan. 2022.
- [74] A. Balaji et al., “Mapping Spiking Neural Networks to Neuromorphic Hardware,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 1, pp. 76-86, Jan. 2020.
- [75] F. Tian, et al., “NeuroCARE: A generic neuromorphic edge computing framework for healthcare applications,” *Frontiers in Neuroscience*, vol.17, 2023.
- [76] D. Freedman, R. Pisani, and R. Purves, “Statistics (international student edition),” 4th edn. New York, 2007.



- [77] A. Shoeb, “Application of machine learning to epileptic seizure onset detection and treatment,” *Ph.D. dissertation, Massachusetts Inst. Technol., Cambridge, MA, USA*, Sep. 2009.
- [78] M. Zhang, L. Zhang, C. -W. Tsai and J. Yoo, “A Patient-Specific Closed-Loop Epilepsy Management SoC With One-Shot Learning and Online Tuning,” *IEEE J. Solid-State Circuits*, vol. 57, no. 4, pp. 1049-1060, April 2022.
- [79] St. microelectronics, STM32H747/757, Available at: <https://www.st.com/en/microcontrollers-microprocessors/stm32h747-757.html>.
- [80] T. Bekolay, et al., “Nengo: a Python tool for building large-scale functional brain models,” *Front. Neuroinform.*, vol. 7, no. 13, pp. 1049-1060, Jan 2014.
- [81] M. Daviesasa, et al., “Loihi: A Neuromorphic Manycore Processor with On-Chip Learning,” *IEEE Micro.*, pp. 1-10, Jan 2018.
- [82] Q. Chen, et al., “Epilepsy Seizure Detection and Prediction using an Approximate Spiking Convolutional Transformer,” *IEEE International Symposium on Circuits and Systems (ISCAS)*, 2024.
- [83] F. Tian, J. Yang, S. Zhao and M. Sawan, “A New Neuromorphic Computing Approach for Epileptic Seizure Prediction,” *IEEE International Symposium on Circuits and Systems (ISCAS)*, 2021.
- [84] H. Shan, et al., “Compact seizure detection based on spiking neural network and support vector machine for efficient neuromorphic implementation,” *Biomedical Signal Processing and Control (BSPC)*, vol. 86, p. 105268, 2023.

- [85] Shiqi Zhao , Jie Yang , and Mohamad Sawan, “Energy-Efficient Neural Network for Epileptic Seizure Prediction,” *IEEE Trans. on Biomedical Engineering*, vol. 69, no. 1, 2022.

# APPENDICES

# **Appendix A**

## **Python code for CNN in TensorFlow keras**

### **A.1 Data loader and CNN model**

The EEG data from CHB-MIT dataset is in the form of .edf files. So the provided coded in this section will read the input files of specified patient from the given directory and write image(.png) files of each second(as we are using 1 sec window here) and also save the corresponding labels according to given information is summary file that which timestamp is labeled as seizure or non-seizure.

### **A.2 Python Jupyter notebook code for CNN**

All the required libraries need to be installed on a system running given code and the required paths are need to be updated to read and write data.

```

1 #####      install the required libraries      #####
2 # pip install pyEDFlib
3 # pip install keras
4 # pip install tensorflow
5 # pip install tensorflow-gpu
6 # pip install scikit-learn
7 import os                                # check our work directory
8 import PIL                               # lib for Imaging tasks
9 import keras                             # lib for deep learning
10 import pyedflib                          # lib to read EDF files
11 import numpy as np                       # lib for algebra/numerical operations
12 import pandas as pd                      # lib for data manipulation
13 import tensorflow as tf                  # lib for deep learning
14 import matplotlib.pyplot as plt          # lib for plotting
15 import random                            # lib for random number generation
16
17 from PIL import Image                    # python Imaging Library
18 from scipy import signal                 # import signal
19 from numpy.random import seed            # set seed for reproducibility
20 from tensorflow.keras.utils import Sequence # import Sequence
21 from tensorflow.keras.models import Sequential # import Sequential
22 from tensorflow.keras.optimizers import Adam # import optimizers
23 from sklearn.model_selection import train_test_split # train/test split
24 from sklearn.metrics import accuracy_score # import performance
25 from sklearn.metrics import f1_score, roc_curve # import performance
26 from tensorflow.keras.layers import Dropout, Dense # import layers from
27 from tensorflow.keras.layers import Flatten, Conv # import layers from

```

```

28 from tensorflow.keras.layers import Conv2D, MaxPooling2D, GlobalMaxPooling2D,
    GlobalAveragePooling1D, GlobalAveragePooling2D, BatchNormalization,
    TimeDistributed, SimpleRNN, GRU
29
30 # To check GPU and CUDA configurations
31 !nvidia-smi
32
33 ##### Dataset Creation #####
34 ### (Reading .edf files and save them as .jpg or .npy) #####
35
36 # parent directory to datapath
37 %cd /local/data0/amuneeb/chb-mit/chb_02/chb02_edf/
38
39 patients = ["02"] # specify subject number
40 fs = 256 # sampling rate
41 pathDataSet = '/amuneeb/chb-mit/chb_02/chb02_edf/' # specify path
42
43 ### function to read the summary file of subject
44 def loadSummaryPatient(index):
45     f = open(pathDataSet+'/chb'+patients[index]+'-summary.txt', 'r')
46     parent = 'chb'+patients[index]+'/'
47     return f, parent
48
49 ### function to read edf files and save them in image format for CNN
50 def dataset_file_creation(secSt, secEn, name_F, parent):
51     file1 = pyedflib.EdfReader(pathDataSet+name_F)
52     n = file1.signals_in_file

```

```

53 print(n)
54 signal_labels = file1.getSignalLabels()
55 signal_headers = file1.getSignalHeaders()
56 rate = signal_headers[0]['sample_rate']
57 dur = file1.getFileDuration()
58 x = np.zeros((n, file1.getNSamples()[0]))
59 for i in range(n):
60     x[i,:] = file1.readSignal(i)
61     label = file1.getLabel(i)
62 file1.close()
63
64 path = '/local/data0/amuneeb/chb-mit/chb_02/np_'+ parent
65
66 if os.path.isdir(path) is not True:
67     os.makedirs(path)
68
69 picnum = int(dur*rate/256)
70 for i in range(picnum):
71     img = x[:,i*256:(i+1)*256]
72     if secSt <= i+1 <= secEn:                # window size is 1sec
73         filename = '_time_seizure_'+ str(i)
74         np.save(path + name_F.split('.')[0] + filename, img)
75
76 else:
77     filename = '_time_nonseizure_'+ str(i)
78     np.save(path + name_F.split('.')[0] + filename, img)
79

```

```

80 ### function to take input and create dataset
81 def create_dataset():
82     print("START Reading Dataset \n")
83     for indexPatient in range(0, len(patients)):
84         f, parent = loadSummaryPatient(indexPatient)
85         line=f.readline()
86
87         while (line):
88             data=line.split(':')
89             if (data[0]=="File Name"):
90                 name_F=data[1].strip()
91                 print(name_F)
92
93                 for i in range(3):
94                     line=f.readline()
95                     for j in range(0, int(line.split(': ')[1])):
96                         secSt=int(f.readline().split(': ')[1].split(' ')[0])
97                         secEn=int(f.readline().split(': ')[1].split(' ')[0])
98                         dataset_file_creation(secSt, secEn, name_F, parent)
99                     line=f.readline()
100                 f.close()
101     print("\n END Reading Dataset")
102
103 ### call above functions to create dataset
104 create_dataset()
105
106

```



```

107 #####    Load Dataset to CNN model    #####
108
109 ### datagenerator loop, to load train and test data
110 def generatePathList(patients, test_size, val_size):
111     parent_path = '/amuneeb/chb-mit/chb_02/np_chb02/'
112     pathList = []
113     for indexPatient in range(0, len(patients)):
114         sub_path = 'chb'+patients[indexPatient]+'/'
115         directory_name = parent_path
116         for filename in os.listdir(directory_name):
117             pathList.append(directory_name+filename)
118             L = len(pathList)
119             test_index = int(L*test_size)
120             val_index = int(L*val_size)
121             index = random.sample(range(L), L)
122             return index[:test_index], index[test_index:(test_index+val_index)],
123                 index[(test_index+val_index):], pathList
124
125 class DataGenerator(Sequence):
126
127     def __init__(self, index, pathList, parent_path = '/local/data0/amuneeb/
128         chb-mit/chb_02/np_chb02/', batch_size=32):
129         self.batch_size = batch_size
130         self.parent_path = parent_path
131         self.pathList = pathList
132         self.index = index
133         self.L = len(self.index)

```

```

132
133 def __len__(self):
134     return self.L - self.batch_size
135
136 def __getitem__(self, idx):
137     batch_indexes = self.index[idx:(idx+self.batch_size)]
138     data_path = [self.pathList[k] for k in batch_indexes]
139     return self._load_data(data_path)
140
141 def _load_data(self, data_path):
142     features = np.zeros((len(data_path), 23, 256))
143     labels = np.zeros((len(data_path)), dtype=int)
144     i = 0
145     for name in data_path:
146         if '_seizure_' in name:
147             features[i] = np.load(name)[0:23,:]
148             labels[i] = 1
149         elif '_nonseizure_' in name:
150             features[i] = np.load(name)[0:23,:]
151             labels[i] = 0
152         i = i+1
153     return np.expand_dims(np.array(features), axis=3), labels
154
155 ##### Defining CNN model #####
156 import tensorflow
157 model = Sequential()
158

```

```

159 model.add(Conv2D(4, (1, 8), input_shape=((23, 256, 1))))
160 model.add(MaxPooling2D(pool_size=(1, 8)))
161 model.add(BatchNormalization())
162 model.add(Activation('relu'))
163
164 model.add(Conv2D(16, (1, 16)))
165 model.add(MaxPooling2D(pool_size=(1, 4)))
166 model.add(BatchNormalization())
167 model.add(Activation('relu'))
168
169 model.add(Conv2D(16, (1, 8), padding='same'))
170 model.add(MaxPooling2D(pool_size=(1, 4)))
171 model.add(BatchNormalization())
172 model.add(Activation('relu'))
173
174 model.add(Conv2D(16, (16, 1)))
175 model.add(MaxPooling2D(pool_size=(4, 1)))
176 model.add(BatchNormalization())
177 model.add(Activation('relu'))
178
179 model.add(Conv2D(16, (16, 1), padding='same'))
180 model.add(MaxPooling2D(pool_size=(4, 1), padding='same'))
181 model.add(BatchNormalization())
182 model.add(Activation('relu'))
183
184 model.add(GlobalAveragePooling2D(data_format=None, keepdims=False))
185 model.add(Dropout(0.15))

```

```

186
187 model.add(Dense(120, activation='relu'))
188 model.add(Dense(84, activation='relu'))
189 model.add(Dense(1, activation='sigmoid'))
190
191 Adam = tensorflow.keras.optimizers.Adam(learning_rate=0.001, beta_1=0.9,
      beta_2=0.999, epsilon=1e-03, amsgrad=False, name='Adam')
192
193 model.compile(metrics=['ACC', keras.metrics.Recall(name='SENN')],
194 loss='binary_crossentropy', optimizer='Adam')
195
196 # print summary of defined model
197 model.summary()
198
199 # folder path
200 %cd /amuneeb/chb-mit/chb_02/
201 dir_path = r'img_chb02/'
202 count = 0
203 # Iterate directory
204 for path in os.listdir(dir_path):
205 # check if current path is a file
206 if os.path.isfile(os.path.join(dir_path, path)):
207 count += 1
208 print('File count:', count)
209
210 # train and test datasplit

```

```

211 test_id, val_id, train_id, path_list = generatePathList(patients, test_size
    =0.3, val_size=0.2)    # 30% test_size
212 train_data = DataGenerator(train_id, path_list)
213 test_data  = DataGenerator(test_id, path_list)
214 val_data   = DataGenerator(val_id, path_list)
215
216 # input shape must be (batch_size, height, width, channels) for cnn in
    tensorflow
217 print("check len of train_data--->", len(train_data))
218 print("check len of val_data---->", len(val_data))
219 print("check len of test_data---->", len(test_data))
220 print("check shape of data---->", (test_data[0][0].shape))
221
222 from tensorflow.keras.callbacks import EarlyStopping
223 early_stop = EarlyStopping(monitor='val_loss', patience=100)
224
225 # train model
226 history = model.fit(train_data, epochs= 20, verbose=1, callbacks=[early_stop
    ], steps_per_epoch=None,
227 validation_data=val_data, class_weight={0:1, 1:50}, workers=4,
    use_multiprocessing=True,
228 shuffle=False, initial_epoch=0)
229
230 # evaluate model
231 model.evaluate(test_data, batch_size=32)
232
233 # test model

```

```

234 y_pred = model.predict(test_data)
235 y_pred = np.round(y_pred).astype(int)
236 y_val = []
237 for i in range(len(test_data)):
238     list_val = list(test_data[i])
239     y_val = np.hstack((y_val, list_val[1]))
240
241 print('size', y_pred.shape, y_val.shape)
242 print('size', y_pred.size, y_val.size)
243
244 # confusion matrix for perofrmance evaluation of model
245 tn, fp, fn, tp = confusion_matrix(y_val, y_pred).ravel()
246 print('accuracy \t', (accuracy_score(y_val, y_pred)*100), '%')
247 sensitivity = int(tp)/int(tp+fn)
248 print('sensitivity \t', (sensitivity*100), '%')
249 specificity = int(tn)/int(fp+tn)
250 print('specificity \t', (specificity*100), '%')
251 print('F1-score \t', (f1_score(y_val, y_pred)*100), '%')
252
253 # plot model loss
254 plt.plot(history.history['loss'])
255 plt.plot(history.history['val_loss'])
256 plt.title('Model loss')
257 plt.ylabel('Loss')
258 plt.xlabel('Epoch')
259 plt.legend(['Train', 'Val'], loc='upper right')
260 plt.show()

```

```
261
262 # plot model accuracy
263 plt.plot(history.history['ACC'])
264 plt.plot(history.history['val_ACC'])
265 plt.title('Model accuracy')
266 plt.ylabel('Accuracy')
267 plt.xlabel('Epoch')
268 plt.legend(['Train', 'Val'], loc='lower right')
269 plt.show()
```

Listing A.1: Python Jupyter Notebook Code for CNN development in TensorFlow Keras

# Appendix B

## Python code Partial Spiking CNN

### B.1 Partial Spiking CNN model

For the better understanding of each and every block of SNNs a customized Partial spiking CNN model is developed using Numpy library in which the first layer of PSCNN is non-spiking(working in continuous 32-bit MAC operations) and rest of network is spiking.

### B.2 Python Jupyter notebook code for PSCNN

All the required libraries need to be installed on a system running given code and the required paths are need to be updated to read and write data.

```
1 ##### import all the required libraries #####
2 import os
```



```

3 import sys
4 import time
5 import matplotlib.pyplot as plt
6 import numpy as np
7 import configparser
8 import tensorflow as tf
9 import numpy as np
10 import sys
11 np.set_printoptions(threshold=sys.maxsize)
12 print(np.__version__)
13
14 # define configurations
15 sim_time = 18
16 thresh = 1
17 dt = 1
18
19 def flatten(input):
20     batch_size = input.shape[0]
21     in_height = input.shape[1]
22     in_width = input.shape[2]
23     in_channels = input.shape[3]
24     output = np.zeros((batch_size * in_height * in_width * in_channels))
25     idx = 0
26
27     for b in range(batch_size):
28         for h in range(in_height):
29             for w in range(in_width):

```

```

30         for c in range(in_channels):
31             output[idx] = input[b][h][w][c]
32             idx += 1
33     return output
34
35
36 def flatten_spike(input):
37     time_step = input.shape[0]
38     batch_size = input.shape[1]
39     in_height = input.shape[2]
40     in_width = input.shape[3]
41     in_channels = input.shape[4]
42     output = np.zeros((time_step * batch_size * in_height * in_width *
43                        in_channels))
44     idx = 0
45
46     for t in range(time_step):
47         for b in range(batch_size):
48             for h in range(in_height):
49                 for w in range(in_width):
50                     for c in range(in_channels):
51                         output[idx] = input[t][b][h][w][c]
52                         idx += 1
53     return output
54
55 def dense_layer_relu(input, weights, biases, num_neurons):

```

```

56 output = np.zeros(num_neurons)
57 mem     = np.zeros(num_neurons)
58 spikes  = np.zeros(num_neurons)
59 out     = np.zeros(num_neurons)
60
61 for i in range(num_neurons):
62     sum_val = 0.0
63     for j in range(input.shape[0]):
64         sum_val = sum_val + (input[j] * weights[j][i])
65     output[i] = sum_val + biases[i]
66     output[output < 0] = 0          ###ReLU
67     mem[i] = output[i]
68 # spiking
69 spikes[i] = np.where(output[i] >= thresh, dt, 0)
70 # membrane potential
71 out[i] = np.where(mem[i] >= thresh, dt, 0)
72 mem[i] = np.where(mem[i] >= thresh, mem[i] - thresh, mem[i])
73 out[i] = np.where(mem[i] >= thresh, out[i] + dt, out[i])
74 mem[i] = np.where(mem[i] >= thresh, mem[i] - thresh, mem[i])
75 return output, mem, spikes, out
76
77
78 def dense_layer(input, weights, biases, num_neurons):
79     output = np.zeros(num_neurons)
80     mem     = np.zeros(num_neurons)
81     spikes  = np.zeros(num_neurons)
82     out     = np.zeros(num_neurons)

```

```

83
84 for i in range(num_neurons):
85     sum_val = 0.0
86     for j in range(input.shape[0]):
87         sum_val = sum_val + (input[j] * weights[j][i])
88         output[i] = sum_val + biases[i]
89         mem[i] = output[i]
90     # spiking
91     spikes[i] = np.where(output[i] >= thresh, dt, 0)
92     # membrane potential
93     out[i] = np.where(mem[i] >= thresh, dt, 0)
94     mem[i] = np.where(mem[i] >= thresh, mem[i] - thresh, mem[i])
95     out[i] = np.where(mem[i] >= thresh, out[i] + dt, out[i])
96     mem[i] = np.where(mem[i] >= thresh, mem[i] - thresh, mem[i])
97     return output, mem, spikes, out
98
99
100 def conv2d(input, weights, biases, batch_size, input_height, input_width,
            input_channels, filter_height, filter_width, num_filters, padding, stride
            ):
101     if padding == -1:
102         # calculate new padded height and width
103         padded_height = (input_height - 1) * stride + filter_height
104         padded_width = (input_width - 1) * stride + filter_width
105
106         pad_top = (padded_height - input_height) // 2
107         pad_left = (padded_width - input_width) // 2

```

```

108
109     # create a new input array with padded dimensions
110     padded_input = np.zeros((batch_size, padded_height, padded_width,
111                               input_channels))
112
113     # copy the original input array to the padded input array
114     for b in range(batch_size):
115         for i in range(input_height):
116             for j in range(input_width):
117                 for k in range(input_channels):
118                     padded_input[b][i + pad_top][j + pad_left][k] = input[b][i][j][k]
119
120     # update input dimensions and padding value
121     input_height = padded_height
122     input_width = padded_width
123     padding = 0 # Set padding to 0 for the rest of the function
124
125     # update the input reference to use the padded input array
126     input = padded_input
127
128     output_height = (input_height - filter_height + 2 * padding) // stride + 1
129     output_width = (input_width - filter_width + 2 * padding) // stride + 1
130
131     # Initialize the output tensor
132     output = np.zeros((batch_size, output_height, output_width, num_filters)
133                       )

```

```

132 mem      = np.zeros((batch_size, output_height, output_width, num_filters)
    )
133 spikes   = np.zeros((batch_size, output_height, output_width, num_filters)
    )
134 out      = np.zeros((batch_size, output_height, output_width, num_filters)
    )
135
136 # perform convolution
137 for b in range(batch_size):
138     for i in range(output_height):
139         for j in range(output_width):
140             for k in range(num_filters):
141                 output[b][i][j][k] = 0.0
142                 for ii in range(filter_height):
143                     for jj in range(filter_width):
144                         for kk in range(input_channels):
145                             input_row = i * stride + ii - padding
146                             input_col = j * stride + jj - padding
147                             if 0 <= input_row < input_height and 0 <= input_col <
input_width:
148                                 output[b][i][j][k] += input[b][input_row][input_col][kk] *
weights[ii][jj][kk][k]
149                             output[b][i][j][k] += biases[k]
150                             mem[b][i][j][k] = output[b][i][j][k]
151 return output, mem, spikes, out
152
153 ##### change paths and specify data #####

```

```

154 cc = np.load ("/amuneeb/proj/chb-mit/snn_toolbox/
    simple_network_py_implementation/net_eeg_conv_imp/arr_0.npy")
155 print (cc.shape)
156 x = cc[47,:]
157 x = x.reshape((1, 23, 256, 1))
158 print(x.shape)
159
160 ### input _ layer1  ----- layer 1
161 input_l1 = x
162 out_l1 = input_l1
163 print("Shape of output:", out_l1.shape)
164
165 ### conv _ layer1  ----- layer 2
166 input_l2 = out_l1
167 filter_height_l2 = 1
168 filter_width_l2 = 8
169 num_filters_l2 = 4
170 padding_l2 = 0
171 stride_l2 = 1
172
173 batch_size_l2 = input_l2.shape[0]
174 in_height_l2 = input_l2.shape[1]
175 in_width_l2 = input_l2.shape[2]
176 in_channels_l2 = input_l2.shape[3]
177
178 weights_l2 = np.load("/amuneeb/proj/chb-mit/snn_toolbox/
    simple_network_py_implementation/net_eeg_conv_imp/w_conv_1.npy")

```

```

179 biases_l2 = np.array ([ 1.          , -0.21174717, -0.45101094,  0.8199355 ])
180
181 # Perform convolution with thresholding and spike generation
182 output_height_l2 = (in_height_l2 - filter_height_l2 + 2 * padding_l2) //
    stride_l2 + 1
183 output_width_l2  = (in_width_l2 - filter_width_l2 + 2 * padding_l2) //
    stride_l2 + 1
184 output_l2, mem_l2, spikes_l2, out_l2 = conv2d(input_l2, weights_l2,
    biases_l2, batch_size_l2, in_height_l2, in_width_l2,
185 in_channels_l2, filter_height_l2, filter_width_l2, num_filters_l2,
186 padding_l2, stride_l2)
187
188 a = output_l2                                ## sum(Wx+b)
189 z = thresh*a                                ## z = Vthr * a
190 v      = np.zeros((batch_size_l2, output_height_l2, output_width_l2,
    num_filters_l2))
191 v_prev  = np.zeros((batch_size_l2, output_height_l2, output_width_l2,
    num_filters_l2))
192 spike_list = []
193
194 for i in range (sim_time//dt):
195 v = (v_prev + z)                                ## integrate mem_potentail
196 spike_timestep = np.zeros((batch_size_l2, output_height_l2, 31,
    num_filters_l2))                                ## to store spikes for current timestep
197
198 for f in range (num_filters_l2):
199     for y in range(output_height_l2):

```



```

200     for x in range(0, 8*31, 8):
201         window = v[:,y,x:x+8,f]
202         m=int(x/8)
203         if (np.any(window-thresh >= 0)):    ## compare with thresh
204             spike_timestep[:, y, m, f] = 1          ## spike
205             v[:,y,x:x+8,f] = v[:,y,x:x+8,f] - thresh ## subtract mem
206             v[:,y,x:x+8,f][v[:,y,x:x+8,f] < 0] = 0    ## reLU
207         else:
208             spike_timestep[:, y, m, f] = 0
209
210
211     # print (spike_timestep.shape)
212     spike_list.append(spike_timestep)
213     v_prev = v.copy()
214     spike_l2 = np.array(spike_list)
215     print('spikes of layer_2', spike_l2.shape)
216     print('spikes of layer_2: \n', spike_l2)
217
218     #####   spiking network from this 2nd convolution layer   #####
219     ### conv _ Layer2  ----- layer 4
220     filter_height_l4 = 1
221     filter_width_l4 = 16
222     num_filters_l4 = 16
223     padding_l4 = 0
224     stride_l4 = 1
225
226     batch_size_l4 = 1

```

```

227 in_height_l4 = 23
228 in_width_l4 = 31
229 in_channels_l4 = 4
230
231 weights_l4 = np.load("/amuneeb/proj/chb-mit/snn_toolbox/
    simple_network_py_implementation/net_eeg_conv_imp/w_conv_2.npy")
232 biases_l4 = np.array ([-0.1198286 , -0.1224545 , -0.04240831,  0.12511705,
233 -0.09632934, -0.09495766,  0.01580807, -0.15692943,
234 -0.0649002 , -0.10462347, -0.02581768, -0.08982677,
235 -0.09719819, -0.09956122, -0.1272312 ,  0.00084822])
236
237 output_height_l4 = (in_height_l4 - filter_height_l4 + 2 * padding_l4) //
    stride_l4 + 1
238 output_width_l4  = (in_width_l4 - filter_width_l4 + 2 * padding_l4) //
    stride_l4 + 1
239
240 # perform convolution with thresholding and spike generation
241 v      = np.zeros((batch_size_l4, output_height_l4, output_width_l4,
    num_filters_l4))
242 v_prev = np.zeros((batch_size_l4, output_height_l4, output_width_l4,
    num_filters_l4))
243 spike_list = []
244 for i in range (sim_time//dt):
245     input_l4 = spike_l2[i, :, :, :, :]
246     output_l4, mem_l4, spike_l4, out_l4 = conv2d(input_l4, weights_l4,
    biases_l4, batch_size_l4, in_height_l4, in_width_l4, in_channels_l4,
    filter_height_l4, filter_width_l4, num_filters_l4, padding_l4, stride_l4)

```

```

247
248 a = output_l4                                # sum(Wx+b)
249 z = thresh*a                                  # z = Vthr * a
250 v = (v_prev + z)                              # integrate mem_potentail
251 spike_timestep = np.zeros((batch_size_l4, output_height_l4, 4,
    num_filters_l4))    ## to store spikes for current timestep
252
253 for f in range (num_filters_l4):
254     for y in range(output_height_l4):
255         for x in range(0, 4*4, 4):
256             window = v[:, y, x:x + 4, f]
257             m=int(x/4)
258             if (np.any(window-thresh >= 0)):      # compare with thresh
259                 spike_timestep[:, y, m, f] = 1    # spike
260                 v[:,y,x:x+4,f] = v[:,y,x:x+4,f] - thresh # subtract mem
261                 v[:,y,x:x+4,f][v[:,y,x:x+4,f] < 0] = 0    # reLU
262             else:
263                 spike_timestep[:, y, m, f] = 0
264 spike_list.append(spike_timestep)
265 v_prev = v.copy()
266 spike_l4 = np.array(spike_list)
267 print('spikes of layer_4', spike_l4.shape)
268 print('spikes of layer_4: \n', spike_l4)
269
270
271 ### conv _ layer3 ----- layer 6
272 filter_height_l6 = 1

```

```

273 filter_width_l6 = 8
274 num_filters_l6 = 16
275 padding_l6 = -1
276 stride_l6 = 1
277
278 batch_size_l6 = 1
279 in_height_l6 = 23
280 in_width_l6 = 4
281 in_channels_l6 = 16
282
283 weights_l6 = np.load("/amuneeb/proj/chb-mit/snn_toolbox/
    simple_network_py_implementation/net_eeg_conv_imp/w_conv_3.npy")
284 biases_l6 = np.array ([-0.04850903, -0.12406814, -0.08137448, -0.07195912,
285 0.12712198, -0.06715503, -0.04275383, -0.05668198,
286 -0.06211237, -0.04940676, -0.02966597, -0.0338957 ,
287 0.03929811, -0.0486182 , -0.03896089,  0.0025328 ])
288
289 if padding_l6 == -1:
290     # Calculate new padded height and width
291     padded_height = (in_height_l6 - 1) * stride_l6 + filter_height_l6
292     padded_width = (in_width_l6 - 1) * stride_l6 + filter_width_l6
293     pad_top = (padded_height - in_height_l6) // 2
294     pad_left = (padded_width - in_width_l6) // 2
295     # Create a new input array with padded dimensions
296     padded_input = np.zeros((batch_size_l6, padded_height, padded_width,
        in_channels_l6))
297     # Update input dimensions and padding value

```

```

298 in_height_l6_padded = padded_height
299 in_width_l6_padded  = padded_width
300 padding_l6    = 0 # Set padding to 0 for the rest of the function
301
302 output_height_l6 = (in_height_l6_padded - filter_height_l6 + 2 * padding_l6)
    // stride_l6 + 1
303 output_width_l6  = (in_width_l6_padded - filter_width_l6 + 2 * padding_l6)
    // stride_l6 + 1
304
305 # Perform convolution with thresholding and spike generation
306 v = np.zeros((batch_size_l6, output_height_l6, output_width_l6,
    num_filters_l6))
307 v_prev = np.zeros((batch_size_l6, output_height_l6, output_width_l6,
    num_filters_l6))
308 spike_list = []
309 padding_l6    = -1
310 for i in range (sim_time//dt):
311     input_l6 = spike_l4[i, :, :, :, :]
312     output_l6, mem_l6, spike_l6, out_l6 = conv2d(input_l6, weights_l6,
        biases_l6, batch_size_l6, in_height_l6, in_width_l6, in_channels_l6,
        filter_height_l6, filter_width_l6, num_filters_l6, padding_l6, stride_l6)
313
314     a = output_l6                                ## sum(Wx+b)
315     z = thresh*a                                ## z = Vthr * a
316
317     v = (v_prev + z)                            ## integrate mem_potential

```

```

318 spike_timestep = np.zeros((batch_size_l6, output_height_l6, 1,
    num_filters_l6))          ## to store spikes for current timestep
319
320 for f in range (num_filters_l6):
321     for y in range(output_height_l6):
322         for x in range(0, 1*4, 4):
323             window = v[:, y, x:x + 4, f]
324             m=int(x/4)
325             if (np.any(window-thresh >= 0)):          # compare with thresh
326                 spike_timestep[:, y, m, f] = 1          # spike
327                 v[:,y,x:x+4,f] = v[:,y,x:x+4,f] - thresh # subtract mem
328                 v[:,y,x:x+4,f][v[:,y,x:x+4,f] < 0] = 0    # reLU
329
330             else:
331                 spike_timestep[:, y, m, f] = 0
332 spike_list.append(spike_timestep)
333 v_prev = v.copy()
334
335 spike_l6 = np.array(spike_list)
336 print('spikes of layer_6', spike_l6.shape)
337 print('spikes of layer_6: \n', spike_l6)
338
339
340 ### conv _ layer4 ----- layer 8
341 filter_height_l8 = 16
342 filter_width_l8 = 1
343 num_filters_l8 = 16

```

```

344 padding_l8 = 0
345 stride_l8 = 1
346
347 batch_size_l8 = 1
348 in_height_l8 = 23
349 in_width_l8 = 1
350 in_channels_l8 = 16
351
352 weights_l8 = np.load("/amuneeb/proj/chb-mit/snn_toolbox/
    simple_network_py_implementation/net_eeg_conv_imp/w_conv_4.npy")
353 biases_l8 = np.array ([-0.07730611, -0.01295044, -0.13072164, -0.11829069,
354 -0.06862988, -0.06416431, -0.13668312,  0.1565233 ,
355 0.02352257, -0.03312271, -0.02246861,  0.1085486 ,
356 -0.01628919, -0.00659762, -0.07357125, -0.01929061])
357
358 output_height_l8 = (in_height_l8 - filter_height_l8 + 2 * padding_l8) //
    stride_l8 + 1
359 output_width_l8  = (in_width_l8 - filter_width_l8 + 2 * padding_l8) //
    stride_l8 + 1
360 # Perform convolution with thresholding and spike generation
361 v = np.zeros((batch_size_l8, output_height_l8, output_width_l8,
    num_filters_l8))
362 v_prev = np.zeros((batch_size_l8, output_height_l8, output_width_l8,
    num_filters_l8))
363 spike_list = []
364 for i in range (sim_time//dt):
365     input_l8 = spike_l6[i, :, :, :, :]

```

```

366 output_l8, mem_l8, spike_l8, out_l8 = conv2d(input_l8, weights_l8,
        biases_l8, batch_size_l8, in_height_l8, in_width_l8, in_channels_l8,
        filter_height_l8, filter_width_l8, num_filters_l8, padding_l8, stride_l8)
367
368 a = output_l8                                ## sum(Wx+b)
369 z = thresh*a                                ## z = Vthr * a
370 v = (v_prev + z)                            ## integrate mem_potential
371 spike_timestep = np.zeros((batch_size_l8, 2, output_width_l8,
        num_filters_l8))                        ## to store spikes for current timestep
372 for f in range (num_filters_l8):
373     for y in range(0, 2*4, 4):
374         window = v[:,y:y+4,:,f]
375         m=int(y/4)
376         if (np.any(window-thresh >= 0)):        # compare with thresh
377             spike_timestep[:, m, :, f] = 1      # spike
378             v[:,y:y+4,:,f] = v[:,y:y+4,:,f] - thresh # subtract mem
379             v[:,y:y+4,:,f][v[:,y:y+4,:,f] < 0] = 0 # reLU
380         else:
381             spike_timestep[:, m, :, f] = 0
382 spike_list.append(spike_timestep)
383 v_prev = v.copy()
384
385 spike_l8 = np.array(spike_list)
386 print('spikes of layer_8', spike_l8.shape)
387 print('spikes of layer_8: \n', spike_l8)
388
389

```



```

390 ### conv _ Layer5 ----- layer 10
391 filter_height_l10 = 16
392 filter_width_l10 = 1
393 num_filters_l10 = 16
394 padding_l10 = -1
395 stride_l10 = 1
396
397 batch_size_l10 = 1
398 in_height_l10 = 2
399 in_width_l10 = 1
400 in_channels_l10 = 16
401
402 weights_l10 = np.load("/amuneeb/proj/chb-mit/snn_toolbox/
    simple_network_py_implementation/net_eeg_conv_imp/w_conv_5.npy")
403 biases_l10 = np.array ([ 0.03151372, -0.04661058,  0.0132325 ,  0.00946984,
404 -0.11233933, -0.09899666, -0.00634665,  0.01023328,
405 0.09918611, -0.00097357,  0.0489762 ,  0.02648545,
406 0.03093834,  0.00122427, -0.01635609, -0.11589059])
407
408 output_height_l10 = (in_height_l10 - filter_height_l10 + 2 * padding_l10) //
    stride_l10 + 1
409 output_width_l10  = (in_width_l10 - filter_width_l10 + 2 * padding_l10) //
    stride_l10 + 1
410 if padding_l10 == -1:
411 # calculate new padded height and width
412 padded_height = (in_height_l10 - 1) * stride_l10 + filter_height_l10
413 padded_width = (in_width_l10 - 1) * stride_l10 + filter_width_l10

```

```

414 pad_top = (padded_height - in_height_l10) // 2
415 pad_left = (padded_width - in_width_l10) // 2
416 # Create a new input array with padded dimensions
417 padded_input = np.zeros((batch_size_l10, padded_height, padded_width,
    in_channels_l10))
418 #update input dimensions and padding value
419 in_height_l10_padded = padded_height
420 in_width_l10_padded = padded_width
421 padding_l10 = 0 # Set padding to 0 for the rest of the function
422 output_height_l10 = (in_height_l10_padded - filter_height_l10 + 2 *
    padding_l10) // stride_l10 + 1
423 output_width_l10 = (in_width_l10_padded - filter_width_l10 + 2 *
    padding_l10) // stride_l10 + 1
424 # perform convolution with thresholding and spike generation
425 v = np.zeros((batch_size_l10, output_height_l10, output_width_l10,
    num_filters_l10))
426 v_prev = np.zeros((batch_size_l10, output_height_l10, output_width_l10,
    num_filters_l10))
427 spike_list = []
428 padding_l10 = -1
429 for i in range (sim_time//dt):
430     input_l10 = spike_l8[i, :, :, :, :]
431     output_l10, mem_l10, spike_l10, out_l10 = conv2d(input_l10, weights_l10,
        biases_l10, batch_size_l10, in_height_l10, in_width_l10, in_channels_l10,
        filter_height_l10, filter_width_l10, num_filters_l10, padding_l10,
        stride_l10)
432     a = output_l10 ## sum(Wx+b)

```

```

433 z = thresh*a                                ## z = Vthr * a
434 v = (v_prev + z)                            ## integrate mem_potentail
435 spike_timestep = np.zeros((batch_size_l10, 1, output_width_l10,
    num_filters_l10))    ## to store spikes for current timestep
436 for f in range (num_filters_l10):
437     for y in range(0, 1*2, 2):
438         window = v[:, y:y+2, :, f]
439         m=int(y/2)
440         if (np.any(window-thresh >= 0)):        # compare with thresh
441             spike_timestep[:, m, :, f] = 1      # spike
442             v[:, y:y+2, :, f] = v[:, y:y+2, :, f] - thresh # subtract mem      v
443             [v[:, y:y+2, :, f][v[:, y:y+2, :, f] < 0] = 0    # reLU
444         else:
445             spike_timestep[:, m, :, f] = 0
446 spike_list.append(spike_timestep)
447 v_prev = v.copy()
448
449 spike_l10 = np.array(spike_list)
450 print('spikes of layer_10', spike_l10.shape)
451 print('spikes of layer_10: \n', spike_l10)
452
453 ### flatten _ layer1 ----- layer 11
454 spike_list = []
455 for i in range (sim_time//dt):
456     spike_timestep = np.zeros((16))
457     input_l11 = spike_l10[i, :, :, :, :]

```

```

458 out_l11 = flatten(input_l11)
459 spike_timestep = out_l11
460 spike_list.append(spike_timestep)
461 spike_l11 = np.array(spike_list)
462 print('spikes of layer_11', spike_l11.shape)
463 print('spikes of layer_11: \n', spike_l11)
464
465
466 ### dense _ layer1 ----- layer 12
467 num_neurons_l12 = 120
468 weights_l12 = np.load("/amuneeb/proj/chb-mit/snn_toolbox/
    simple_network_py_implementation/net_eeg_conv_imp/w_dense_1.npy")
469 biases_l12 = np.array ([-2.89347232e-03, -1.97030464e-03, -2.58013532e-02,
    -2.26076180e-03,
470 -1.61345594e-03, 4.80355881e-02, -1.45725962e-02, -1.94588106e-03,
471 -1.58211272e-02, -2.41675936e-02, -4.81813867e-03, -1.60752926e-02,
472 -2.60997098e-02, 8.08483048e-04, -1.87503418e-03, -1.59602240e-02,
473 -1.21144112e-02, -7.69026764e-03, -3.54138482e-03, -1.90637745e-02,
474 -6.02030614e-03, 6.96597015e-03, -4.15502908e-03, -8.48357473e-03,
475 5.66895828e-02, 3.93913569e-06, -1.02648055e-02, -7.55583169e-03,
476 4.31414600e-03, -1.67496596e-02, 4.99117263e-02, -8.67244229e-03,
477 -2.18656287e-03, -4.84777964e-04, -4.30188095e-03, -5.81546873e-03,
478 -4.84305294e-03, -3.75242764e-03, 2.82588545e-02, -1.47308679e-02,
479 -3.16311460e-04, 4.15385254e-02, 1.15323793e-02, -7.82333501e-03,
480 -9.03647859e-03, -2.52858512e-02, -1.19263411e-03, -1.72258893e-04,
481 4.72685173e-02, 6.06105314e-04, 9.43084597e-05, 2.17288621e-02,
482 4.49128170e-03, -5.89371426e-03, -2.73019518e-03, -1.16024166e-02,

```

```

483 -1.12419315e-02, -2.28942707e-02, -8.67724977e-03, -2.48444942e-03,
484 -3.96019295e-02,  7.67870946e-03,  9.74680646e-04, -5.20151109e-03,
485 -2.86603649e-03, -2.49973256e-02,  4.62319069e-02,  2.48483587e-02,
486 6.81748763e-02, -5.23431553e-03, -4.43614088e-03, -1.98337417e-02,
487 -1.88898260e-03,  5.27999923e-02, -8.54129530e-03, -2.48958729e-03,
488 -1.49456877e-03, -1.30833907e-03,  2.74262335e-02,  7.96418190e-02,
489 -3.95716121e-03,  1.33066522e-02, -1.98221710e-02, -7.67724996e-04,
490 -4.61282860e-03,  2.35603321e-02, -6.26598159e-03, -2.73150820e-02,
491 -4.16427465e-05,  1.54193770e-02, -3.05898627e-03, -1.82450470e-02,
492 -4.94179223e-03, -7.00877467e-03,  1.90658087e-03, -2.87549640e-03,
493 -2.43801554e-03, -5.03720343e-03, -1.22205412e-03, -3.00784665e-03,
494 -1.94955431e-02, -9.88522917e-03, -2.72144121e-03, -2.26976303e-03,
495 3.69720794e-02, -1.63976790e-03,  4.24182974e-02, -1.74020324e-02,
496 3.96727398e-03, -5.34299714e-03,  4.67100143e-02, -4.90705296e-03,
497 -5.20496536e-03, -2.43666349e-03, -2.68621389e-02,  6.37066318e-03,
498 -1.66043770e-02,  5.58567680e-02, -1.52451666e-02,  2.85103405e-03])
499
500 # dense layer with thresholding and spike generation
501 v = np.zeros(num_neurons_l12)
502 v_prev = np.zeros(num_neurons_l12)
503 spike_list = []
504 for k in range (sim_time//dt):
505     input_l12 = spike_l11[k]
506     output_l12, mem_l12, spikes_l12, out_l12 = dense_layer_relu(input_l12,
507                                                                    weights_l12, biases_l12, num_neurons_l12)
507     a = output_l12                                ## sum(Wx+b)
508     z = thresh*a                                  ## z = Vthr * a

```

```

509 v = (v_prev + z)                                ## integrate mem_potentail
510 spike_timestep = []                             ## to store spikes for current
    timestep
511 for l in range (z.shape[0]):
512     if ((v[l]-thresh) >= 0):                     ## compare with thresh
513         spike_timestep.append(1)                 ## spike
514         v[l] = v[l] - thresh                     ## subtract mem_potentail
515     else:
516         spike_timestep.append(0)
517         spike_list.append(spike_timestep)
518         v_prev = v.copy()
519
520 spike_l12 = np.array(spike_list)
521 print('spikes of layer_12: \n', spike_l12.shape)
522 print('spikes of layer_12: \n', spike_l12)
523
524
525 ### dense _layer2 ----- layer 13
526 num_neurons_l13 = 84
527 weights_l13 = np.load("/amuneeb/proj/chb-mit/snn_toolbox/
    simple_network_py_implementation/net_eeg_conv_imp/w_dense_2.npy")
528 biases_l13 = np.array ([-4.63681947e-03,  3.77070904e-03,  7.38779642e-03,
    7.60984747e-03,
529 5.89874061e-03, -8.48719012e-03,  1.12406835e-02,  2.16511986e-03,
530 1.79820117e-02,  2.98310611e-02,  8.99855047e-03, -2.86474684e-03,
531 0.00000000e+00,  2.24189996e-03,  2.29859166e-02,  1.14884831e-04,
532 -8.20845552e-03,  2.08682162e-04,  3.24500501e-02,  2.91870558e-04,

```

```

533 -3.49193555e-03, -4.95689642e-03,  2.01518144e-02, -2.72922334e-03,
534 -2.15328811e-03,  4.64394409e-03, -5.99887129e-03, -2.42604199e-03,
535 -5.19911572e-03, -6.87270984e-03,  2.08941405e-03,  1.38646550e-02,
536 5.06390166e-03,  1.94915067e-02, -1.60766789e-03, -1.42720528e-03,
537 -5.82066691e-03,  1.66413113e-02, -3.39099992e-04,  1.23159569e-02,
538 2.11493969e-02,  1.68305356e-02, -1.05989194e-02,  7.56980339e-03,
539 1.63009781e-02, -4.22499795e-03, -1.10558409e-03,  3.25271413e-02,
540 1.05673171e-04,  1.16314311e-02,  9.88232903e-04,  3.90085531e-03,
541 1.44925462e-02, -2.47723213e-03,  1.86881598e-03,  2.78021507e-02,
542 -1.93179108e-03,  1.08054737e-02,  2.37229051e-05,  8.65442026e-03,
543 2.31688400e-03, -9.24572907e-03,  7.48282718e-03,  9.07519460e-03,
544 1.38185527e-02,  1.06354058e-02,  1.11667942e-02, -4.46523249e-04,
545 -2.79341545e-03,  8.70064134e-04,  9.83607862e-03,  2.57344008e-03,
546 -8.37018073e-04, -3.07562761e-04, -4.28229244e-03,  3.51762515e-03,
547 1.09176477e-03, -3.96167906e-03,  2.63773222e-02, -2.86396686e-03,
548 3.10602915e-02,  1.67376585e-02,  1.51891634e-02,  1.93437748e-02])
549
550 # dense layer with thresholding and spike generation
551 v = np.zeros(num_neurons_l13)
552 v_prev = np.zeros(num_neurons_l13)
553 spike_list = []
554 for k in range (sim_time//dt):
555     input_l13 = spike_l12[k]
556     output_l13, mem_l13, spikes_l13, out_l13 = dense_layer_relu(input_l13,
557         weights_l13, biases_l13, num_neurons_l13)
558     a = output_l13                                ## sum(Wx+b)
559     z = thresh*a                                  ## z = Vthr * a

```

```

559 v = (v_prev + z)                                ## integrate mem_potential
560 spike_timestep = []                             ## to store spikes for current
    timestep
561 for l in range (z.shape[0]):
562     if ((v[l]-thresh) >= 0):                     ## compare with thresh
563         spike_timestep.append(1)                 ## spike
564         v[l] = v[l] - thresh                     ## subtract mem_potential
565     else:
566         spike_timestep.append(0)
567         spike_list.append(spike_timestep)
568         v_prev = v.copy()
569 spike_l13 = np.array(spike_list)
570 print('spikes of layer_13: \n', spike_l13.shape)
571 print('spikes of layer_13: \n', spike_l13)
572
573
574 ### dense _ layer3 ----- layer 14
575 num_neurons_l14 = 2
576 weights_l14 = np.load("/amuneeb/proj/chb-mit/snn_toolbox/
    simple_network_py_implementation/net_eeg_conv_imp/w_dense_3.npy")
577 biases_l14 = np.array ([ -0.03762753,  0.0376226 ])
578 # dense layer with thresholding and spike generation
579 v = np.zeros(num_neurons_l14)
580 v_prev = np.zeros(num_neurons_l14)
581 spike_list = []
582 for k in range (sim_time//dt):
583     input_l14 = spike_l13[k]

```



```

584 output_l14, mem_l14, spikes_l14, out_l14 = dense_layer_relu(input_l14,
    weights_l14, biases_l14, num_neurons_l14)
585 a = output_l14                                ## sum(Wx+b)
586 z = thresh*a                                  ## z = Vthr * a
587 v = (v_prev + z)                              ## integrate mem_potentail
588 spike_timestep = []                            ## to store spikes for current
    timestep
589 for l in range (z.shape[0]):
590     if ((v[l]-thresh) >= 0):                    ## compare with thresh
591         spike_timestep.append(1)                ## spike
592         v[l] = v[l] - thresh                    ## subtract mem_potentail
593     else:
594         spike_timestep.append(0)
595         spike_list.append(spike_timestep)
596         v_prev = v.copy()
597 spike_l14 = np.array(spike_list)
598 print('spikes of layer_l14: \n', spike_l14.shape)
599 print('spikes of layer_l14: \n', spike_l14)
600
601 # sum up the spike counts for each class over all time steps
602 spike_counts = np.sum(spike_l14, axis=0)
603 # find the class with the maximum spike count
604 winning_class = np.argmax(spike_counts)
605 print("Winning class:", winning_class)
606 print("Spike counts:", spike_counts)

```

Listing B.1: Python Jupyter Notebook Code for PSCNN development in Numpy

# Appendix C

## Python code Fully Spiking CNN

### C.1 Fully Spiking CNN model

For the better understanding of each and every block of SNNs a customized Fully spiking CNN model is developed using Numpy library in which the first layer of FSCNN is also spiking as compared to the SCNN, which makes it compatible with connecting Level crossing ADCs without the need of any encoder block.

### C.2 Python Jupyter notebook code for FSCNN

All the required libraries need to be installed on a system running given code and the required paths are need to be updated to read and write data.

```
1 ##### import all the required libraries #####
```

```

2 import os
3 import sys
4 import time
5 import matplotlib.pyplot as plt
6 import numpy as np
7 import configparser
8 import tensorflow as tf
9 import numpy as np
10 import sys
11 np.set_printoptions(threshold=sys.maxsize)
12 print(np.__version__)
13
14 # define configurations
15 sim_time = 50
16 thresh = 1
17 dt = 1
18
19 ##### input data #####
20 cc=np.load("/amuneeb/proj/chb-mit/snn_toolbox/
           simple_network_py_implementation/net_eeg_conv_imp/arr_0.npy")
21 print (cc.shape)
22 x = cc[47,:]
23 x = x.reshape((23, 256, 1))
24 print(x.shape)
25
26 ##### code for spiking input #####
27 ### generate random scaled and constrained unique numbers ###

```

```

28 lower_scale = 0.0
29 upper_scale = 1.0
30 v_lsb = 0.02
31 n_steps = int(upper_scale/v_lsb)
32 print (n_steps)
33
34 random_matrix = np.unique(np.round(np.random.uniform(lower_scale,
    upper_scale, size=n_steps) / v_lsb) * v_lsb)
35 while len(random_matrix) < n_steps:
36 additional_numbers = np.unique(np.round(np.random.uniform(lower_scale,
    upper_scale, size=n_steps) / v_lsb) * v_lsb)
37 random_matrix = np.unique(np.concatenate([random_matrix, additional_numbers
    ]))
38 random_matrix = random_matrix[:n_steps]
39 print(random_matrix.shape)
40 print(random_matrix)
41
42 ### shuffle to create N unique matrices ###
43 num_arrays = n_steps
44 shuffled_arrays = []
45 print (random_matrix.shape)
46
47 for _ in range(num_arrays):
48     shuffled_array = random_matrix.copy()
49     np.random.shuffle(shuffled_array)
50     shuffled_arrays.append(shuffled_array)
51

```

```

52 shuffled_arrays = np.array(shuffled_arrays)
53 print (shuffled_arrays.shape)
54 print("First Shuffled Array:")
55 print(shuffled_arrays[0,:])
56
57 ##### comparison of original and random generated matrices #####
58 orig_matrix = x
59 rand_matrix = shuffled_arrays[0,:]
60 print (rand_matrix.shape)
61 out_matrix = []
62 out_matrixs = []
63
64 for k in range (n_steps):
65     out_matrix = (orig_matrix[:,:::] > rand_matrix[k]).astype(int)
66     out_matrixs.append(out_matrix)
67 out_matrixs = np.array(out_matrixs)
68
69 # Print the shapes of input_matrix and output_matrix
70 print("Orig Matrix Shape:", orig_matrix.shape)
71 # print("Orig Matrix:", orig_matrix)
72 print("Rand Matrix Shape:", rand_matrix.shape)
73 # # print("Rand Matrix:", rand_matrix)
74 print("Output Matrix Shape:", out_matrixs.shape)
75
76 ##### code for concatenation of spiking input #####
77 ##### to make the input similar to one that #####
78 ##### will be generated by the LC-ADC #####

```

```

79 ##### concatenated to reduce the time dimension and make a single row #####
80 concatenated_matrix = np.concatenate(out_matrixs, axis=2).reshape(23,
    256*50, 1)
81 print("Concatenated Matrix Shape:", concatenated_matrix.shape)
82 # to verify the concatenation of time domain in rows is okay
83 print (out_matrixs[:, 0, 0, 0])
84 print (concatenated_matrix[0, 0:50, 0])
85
86 new_ins23 = np.empty((23, 249, 400, 1))
87 for i in range(23):
88     for j in range(249):
89         new_ins23[i, j, :, :] = concatenated_matrix[i, j*50 : j*50 + 400, :]
90 print (new_ins23.shape)
91
92 q = np.array([1, 2, 3])
93 w = np.array([4, 5, 6])
94 e = q * w
95 print ("e_shape", e.shape)
96 print (e)
97
98 weights=np.load("/amuneeb/proj/chb-mit/snn_toolbox/
    simple_network_py_implementation/net_eeg_conv_imp/w_conv_1.npy")
99 biases = np.array ([ 1.          , -0.21174717, -0.45101094,  0.8199355 ])
100 out_ch1 = np.zeros((23, 249, 50, 4))
101 for l in range (23):
102     for m in range(249):
103         for n in range (50):

```

```

104     y_in = new_ins23[l, m, [n, n+50, n+100, n+150, n+200, n+250, n+300, n
+350]]
105     for k in range(4):
106         for ii in range(1):
107             for jj in range(1):
108                 for kk in range(1):
109                     y_in = y_in.reshape(8)
110                     o_mul = np.array(y_in) * np.array(weights[ii, :, kk, k])
111                     o_sum = np.sum(o_mul)
112                     out_ch1[l][m][n][k] = o_sum + biases[k]
113 out_ch1.shape
114 print (out_ch1.shape)
115
116 v = np.zeros((23, 249, 4))
117 v_prev = np.zeros((23, 249, 4))
118 spike_list = []
119 for i in range(sim_time//dt):
120     a = out_ch1[:, :, i, :]
121     z = thresh*a                                ## z = Vthr * a
122     v = (v_prev + z)                            ## integrate mem_potentail
123     spike_timestep = np.zeros((23, 31, 4))    ## to store spikes, current
                                                timestep
124     for f in range(4):
125         for y in range(23):
126             for x in range(0, 8*31, 8):
127                 window = v[y,x:x+8,f]
128                 m=int(x/8)

```

```

129         if (np.any(window-thresh >= 0)): ## compare with thresh
130             spike_timestep[y, m, f] = 1          # spike
131             v[y,x:x+8,f] = v[y,x:x+8,f] - thresh # subtract mem
132             v[y,x:x+8,f][v[y,x:x+8,f] < 0] = 0    # reLU
133         else:
134             spike_timestep[y, m, f] = 0
135         spike_list.append(spike_timestep)
136         v_prev = v.copy()
137 spike_l2 = np.array(spike_list)
138
139 print('spikes of layer_2', spike_l2.shape)
140 print('spikes of layer_2: \n', spike_l2)
141
142
143 ### conv _ layer2 ----- layer 4
144 filter_height_l4 = 1
145 filter_width_l4 = 16
146 num_filters_l4 = 16
147 padding_l4 = 0
148 stride_l4 = 1
149
150 in_height_l4 = 23
151 in_width_l4 = 31
152 in_channels_l4 = 4
153
154 weights_l4 = np.load("/amuneeb/proj/chb-mit/snn_toolbox/
    simple_network_py_implementation/net_eeg_conv_imp/w_conv_2.npy")

```



```

155 biases_l4 = np.array ([-0.1198286 , -0.1224545 , -0.04240831,  0.12511705,
156 -0.09632934, -0.09495766,  0.01580807, -0.15692943,
157 -0.0649002 , -0.10462347, -0.02581768, -0.08982677,
158 -0.09719819, -0.09956122, -0.1272312 ,  0.00084822])
159
160 output_height_l4 = (in_height_l4 - filter_height_l4 + 2 * padding_l4) //
    stride_l4 + 1
161 output_width_l4  = (in_width_l4 - filter_width_l4 + 2 * padding_l4) //
    stride_l4 + 1
162 # perform convolution with thresholding and spike generation
163 v          = np.zeros((output_height_l4, output_width_l4, num_filters_l4))
164 v_prev     = np.zeros((output_height_l4, output_width_l4, num_filters_l4))
165 spike_list = []
166 for i in range (sim_time//dt):
167     input_l4 = spike_l2[i, :, :, :]
168     output_l4 = conv2d(input_l4, weights_l4, biases_l4, in_height_l4,
        in_width_l4, in_channels_l4, filter_height_l4, filter_width_l4,
        num_filters_l4, padding_l4, stride_l4)
169     a = output_l4                                ## sum(Wx+b)
170     z = thresh*a                                  ## z = Vthr * a
171     v = (v_prev + z)                              ## integrate mem_potential
172     spike_timestep = np.zeros((output_height_l4, 4, num_filters_l4)) ## to
        store spikes for current timestep
173     for f in range (num_filters_l4):
174         for y in range(output_height_l4):
175             for x in range(0, 4*4, 4):
176                 window = v[y, x:x + 4, f]

```

```

177     m=int(x/4)
178     if (np.any(window-thresh >= 0)):          # compare with thresh
179         spike_timestep[y, m, f] = 1           # spike
180         v[y,x:x+4,f] = v[y,x:x+4,f] - thresh # subtract mem
181         v[y,x:x+4,f][v[y,x:x+4,f] < 0] = 0    # reLU
182     else:
183         spike_timestep[y, m, f] = 0
184     spike_list.append(spike_timestep)
185     v_prev = v.copy()
186
187 spike_l4 = np.array(spike_list)
188 print('spikes of layer_4', spike_l4.shape)
189 print('spikes of layer_4: \n', spike_l4)
190
191
192 ### conv _ layer3 ----- layer 6
193 filter_height_l6 = 1
194 filter_width_l6 = 8
195 num_filters_l6 = 16
196 padding_l6 = -1
197 stride_l6 = 1
198
199 in_height_l6 = 23
200 in_width_l6 = 4
201 in_channels_l6 = 16
202

```

```

203 weights_l6=np.load("/amuneeb/proj/chb-mit/snn_toolbox/
    simple_network_py_implementation/net_eeg_conv_imp/w_conv_3.npy")
204 biases_l6 = np.array ([-0.04850903, -0.12406814, -0.08137448, -0.07195912,
205 0.12712198, -0.06715503, -0.04275383, -0.05668198,
206 -0.06211237, -0.04940676, -0.02966597, -0.0338957 ,
207 0.03929811, -0.0486182 , -0.03896089, 0.0025328 ])
208
209 if padding_l6 == -1:
210     # Calculate new padded height and width
211     padded_height = (in_height_l6 - 1) * stride_l6 + filter_height_l6
212     padded_width = (in_width_l6 - 1) * stride_l6 + filter_width_l6
213     pad_top = (padded_height - in_height_l6) // 2
214     pad_left = (padded_width - in_width_l6) // 2
215     # Create a new input array with padded dimensions
216     padded_input = np.zeros((padded_height, padded_width, in_channels_l6))
217
218     # Update input dimensions and padding value
219     in_height_l6_padded = padded_height
220     in_width_l6_padded = padded_width
221     padding_l6 = 0 # Set padding to 0 for the rest of the function
222     output_height_l6 = (in_height_l6_padded - filter_height_l6 + 2 *
        padding_l6) // stride_l6 + 1
223     output_width_l6 = (in_width_l6_padded - filter_width_l6 + 2 * padding_l6)
        // stride_l6 + 1
224
225 # perform convolution with thresholding and spike generation
226 v = np.zeros((output_height_l6, output_width_l6, num_filters_l6))

```

```

227 v_prev = np.zeros((output_height_l6, output_width_l6, num_filters_l6))
228 spike_list = []
229 padding_l6 = -1
230 for i in range (sim_time//dt):
231     input_l6 = spike_l4[i, :, :, :]
232     output_l6 = conv2d(input_l6, weights_l6, biases_l6, in_height_l6,
233                        in_width_l6,
234                        in_channels_l6, filter_height_l6, filter_width_l6, num_filters_l6,
235                        padding_l6, stride_l6)
236     a = output_l6                                ## sum(Wx+b)
237     z = thresh*a                                ## z = Vthr * a
238     v = (v_prev + z)                            ## integrate mem_potentail
239     spike_timestep = np.zeros((output_height_l6, 1, num_filters_l6)) ## to
240     store spikes for current timestep
241     for f in range (num_filters_l6):
242         for y in range(output_height_l6):
243             for x in range(0, 1*4, 4):
244                 window = v[y, x:x + 4, f]
245                 m=int(x/4)
246                 if (np.any(window-thresh >= 0)):      # compare with thresh
247                     spike_timestep[y, m, f] = 1      # spike
248                     v[y,x:x+4,f] = v[y,x:x+4,f] - thresh # subtract mem
249                     v[y,x:x+4,f][v[y,x:x+4,f] < 0] = 0  # reLU
250                 else:
251                     spike_timestep[y, m, f] = 0
252     spike_list.append(spike_timestep)
253     v_prev = v.copy()

```

```

251 spike_l6 = np.array(spike_list)
252 print('spikes of layer_6', spike_l6.shape)
253 print('spikes of layer_6: \n', spike_l6)
254
255
256 ### conv _ layer4 ----- layer 8
257 filter_height_l8 = 16
258 filter_width_l8 = 1
259 num_filters_l8 = 16
260 padding_l8 = 0
261 stride_l8 = 1
262
263 in_height_l8 = 23
264 in_width_l8 = 1
265 in_channels_l8 = 16
266
267 weights_l8 = np.load("/amuneeb/proj/chb-mit/snn_toolbox/
    simple_network_py_implementation/net_eeg_conv_imp/w_conv_4.npy")
268 biases_l8 = np.array ([-0.07730611, -0.01295044, -0.13072164, -0.11829069,
269 -0.06862988, -0.06416431, -0.13668312,  0.1565233 ,
270 0.02352257, -0.03312271, -0.02246861,  0.1085486 ,
271 -0.01628919, -0.00659762, -0.07357125, -0.01929061])
272
273 output_height_l8 = (in_height_l8 - filter_height_l8 + 2 * padding_l8) //
    stride_l8 + 1
274 output_width_l8  = (in_width_l8 - filter_width_l8 + 2 * padding_l8) //
    stride_l8 + 1

```

```

275 # perform convolution with thresholding and spike generation
276 v      = np.zeros((output_height_l8, output_width_l8, num_filters_l8))
277 v_prev = np.zeros((output_height_l8, output_width_l8, num_filters_l8))
278 spike_list = []
279 for i in range (sim_time//dt):
280     input_l8 = spike_l6[i, :, :, :]
281     output_l8 = conv2d(input_l8, weights_l8, biases_l8, in_height_l8,
        in_width_l8, in_channels_l8, filter_height_l8, filter_width_l8,
        num_filters_l8, padding_l8, stride_l8)
282     a = output_l8                                ## sum(Wx+b)
283     z = thresh*a                                ## z = Vthr * a
284     v = (v_prev + z)                             ## integrate mem_potential
285     spike_timestep = np.zeros((2, output_width_l8, num_filters_l8))    ## to
        store spikes for current timestep
286     for f in range (num_filters_l8):
287         for y in range(0, 2*4, 4):
288             window = v[y:y+4,:,f]
289             m=int(y/4)
290             if (np.any(window-thresh >= 0)):        # compare with thresh
291                 spike_timestep[m, :, f] = 1        # spike
292                 v[y:y+4,:,f] = v[y:y+4,:,f] - thresh # subtract mem          v[y:y
        +4,:,f][v[y:y+4,:,f] < 0] = 0    # reLU
293             else:
294                 spike_timestep[m, :, f] = 0
295             spike_list.append(spike_timestep)
296             v_prev = v.copy()
297 spike_l8 = np.array(spike_list)

```

```

298 print('spikes of layer_8', spike_l8.shape)
299 print('spikes of layer_8: \n', spike_l8)
300
301
302 ### conv _layer5 ----- layer 10
303 filter_height_l10 = 16
304 filter_width_l10 = 1
305 num_filters_l10 = 16
306 padding_l10 = -1
307 stride_l10 = 1
308
309 in_height_l10 = 2
310 in_width_l10 = 1
311 in_channels_l10 = 16
312
313 weights_l10=np.load("/amuneeb/proj/chb-mit/snn_toolbox/
    simple_network_py_implementation/net_eeg_conv_imp/w_conv_5.npy")
314 biases_l10 = np.array ([ 0.03151372, -0.04661058,  0.0132325 ,  0.00946984,
315 -0.11233933, -0.09899666, -0.00634665,  0.01023328,
316 0.09918611, -0.00097357,  0.0489762 ,  0.02648545,
317 0.03093834,  0.00122427, -0.01635609, -0.11589059])
318
319 output_height_l10 = (in_height_l10 - filter_height_l10 + 2 * padding_l10) //
    stride_l10 + 1
320 output_width_l10  = (in_width_l10 - filter_width_l10 + 2 * padding_l10) //
    stride_l10 + 1
321

```

```

322 if padding_l10 == -1:
323     # Calculate new padded height and width
324     padded_height = (in_height_l10 - 1) * stride_l10 + filter_height_l10
325     padded_width = (in_width_l10 - 1) * stride_l10 + filter_width_l10
326     pad_top = (padded_height - in_height_l10) // 2
327     pad_left = (padded_width - in_width_l10) // 2
328     # Create a new input array with padded dimensions
329     padded_input = np.zeros((padded_height, padded_width, in_channels_l10))
330     # Update input dimensions and padding value
331     in_height_l10_padded = padded_height
332     in_width_l10_padded = padded_width
333     padding_l10 = 0 # Set padding to 0 for the rest of the function
334     output_height_l10 = (in_height_l10_padded - filter_height_l10 + 2 *
        padding_l10) // stride_l10 + 1
335     output_width_l10 = (in_width_l10_padded - filter_width_l10 + 2 *
        padding_l10) // stride_l10 + 1
336 # Perform convolution with thresholding and spike generation
337 v = np.zeros((output_height_l10, output_width_l10, num_filters_l10))
338 v_prev = np.zeros((output_height_l10, output_width_l10, num_filters_l10))
339 spike_list = []
340 padding_l10 = -1
341 for i in range (sim_time//dt):
342     input_l10 = spike_l8[i, :, :, :]
343     output_l10 = conv2d(input_l10, weights_l10, biases_l10, in_height_l10,
        in_width_l10, in_channels_l10, filter_height_l10, filter_width_l10,
        num_filters_l10, padding_l10, stride_l10)
344     a = output_l10 ## sum(Wx+b)

```



```

345     z = thresh*a                                ## z = Vthr * a
346     v = (v_prev + z)                            ## integrate mem_potentail
347     spike_timestep = np.zeros((1, output_width_l10, num_filters_l10)) # to
        store spikes for current timestep
348     for f in range (num_filters_l10):
349         for y in range(0, 1*2, 2):
350             window = v[y:y+2, :, f]
351             m=int(y/2)
352             if (np.any(window-thresh >= 0)):      # compare with thresh
353                 spike_timestep[m, :, f] = 1        # spike
354                 v[y:y+2, :, f] = v[y:y+2, :, f] - thresh # subtract mem
355                 v[y:y+2, :, f][v[y:y+2, :, f] < 0] = 0    # reLU
356             else:
357                 spike_timestep[m, :, f] = 0
358             spike_list.append(spike_timestep)
359             v_prev = v.copy()
360 spike_l10 = np.array(spike_list)
361 print('spikes of layer_10', spike_l10.shape)
362 print('spikes of layer_10: \n', spike_l10)
363
364
365 ### flatten _ layer1 ----- layer 11
366 spike_list = []
367 for i in range (sim_time//dt):
368     spike_timestep = np.zeros((16))
369     input_l11 = spike_l10[i, :, :, :]
370     out_l11 = flatten(input_l11)

```

```

371 spike_timestep = out_l11
372 spike_list.append(spike_timestep)
373 spike_l11 = np.array(spike_list)
374 print('spikes of layer_l1', spike_l11.shape)
375 print('spikes of layer_l1: \n', spike_l11)
376
377
378 ### dense _ layer1 ----- layer 12
379 num_neurons_l12 = 120
380 weights_l12=np.load("/amuneeb/proj/chb-mit/snn_toolbox/
    simple_network_py_implementation/net_eeg_conv_imp/w_dense_1.npy")
381 biases_l12 = np.array ([-2.89347232e-03, -1.97030464e-03, -2.58013532e-02,
    -2.26076180e-03,
382 -1.61345594e-03,  4.80355881e-02, -1.45725962e-02, -1.94588106e-03,
383 -1.58211272e-02, -2.41675936e-02, -4.81813867e-03, -1.60752926e-02,
384 -2.60997098e-02,  8.08483048e-04, -1.87503418e-03, -1.59602240e-02,
385 -1.21144112e-02, -7.69026764e-03, -3.54138482e-03, -1.90637745e-02,
386 -6.02030614e-03,  6.96597015e-03, -4.15502908e-03, -8.48357473e-03,
387 5.66895828e-02,  3.93913569e-06, -1.02648055e-02, -7.55583169e-03,
388 4.31414600e-03, -1.67496596e-02,  4.99117263e-02, -8.67244229e-03,
389 -2.18656287e-03, -4.84777964e-04, -4.30188095e-03, -5.81546873e-03,
390 -4.84305294e-03, -3.75242764e-03,  2.82588545e-02, -1.47308679e-02,
391 -3.16311460e-04,  4.15385254e-02,  1.15323793e-02, -7.82333501e-03,
392 -9.03647859e-03, -2.52858512e-02, -1.19263411e-03, -1.72258893e-04,
393 4.72685173e-02,  6.06105314e-04,  9.43084597e-05,  2.17288621e-02,
394 4.49128170e-03, -5.89371426e-03, -2.73019518e-03, -1.16024166e-02,
395 -1.12419315e-02, -2.28942707e-02, -8.67724977e-03, -2.48444942e-03,

```

```

396 -3.96019295e-02,  7.67870946e-03,  9.74680646e-04, -5.20151109e-03,
397 -2.86603649e-03, -2.49973256e-02,  4.62319069e-02,  2.48483587e-02,
398 6.81748763e-02, -5.23431553e-03, -4.43614088e-03, -1.98337417e-02,
399 -1.88898260e-03,  5.27999923e-02, -8.54129530e-03, -2.48958729e-03,
400 -1.49456877e-03, -1.30833907e-03,  2.74262335e-02,  7.96418190e-02,
401 -3.95716121e-03,  1.33066522e-02, -1.98221710e-02, -7.67724996e-04,
402 -4.61282860e-03,  2.35603321e-02, -6.26598159e-03, -2.73150820e-02,
403 -4.16427465e-05,  1.54193770e-02, -3.05898627e-03, -1.82450470e-02,
404 -4.94179223e-03, -7.00877467e-03,  1.90658087e-03, -2.87549640e-03,
405 -2.43801554e-03, -5.03720343e-03, -1.22205412e-03, -3.00784665e-03,
406 -1.94955431e-02, -9.88522917e-03, -2.72144121e-03, -2.26976303e-03,
407 3.69720794e-02, -1.63976790e-03,  4.24182974e-02, -1.74020324e-02,
408 3.96727398e-03, -5.34299714e-03,  4.67100143e-02, -4.90705296e-03,
409 -5.20496536e-03, -2.43666349e-03, -2.68621389e-02,  6.37066318e-03,
410 -1.66043770e-02,  5.58567680e-02, -1.52451666e-02,  2.85103405e-03])
411
412 # dense layer with thresholding and spike generation
413 v = np.zeros(num_neurons_l12)
414 v_prev = np.zeros(num_neurons_l12)
415 spike_list = []
416 for k in range (sim_time//dt):
417     input_l12 = spike_l11[k]
418     output_l12 = dense_layer_relu(input_l12, weights_l12, biases_l12,
419                                   num_neurons_l12)
419     a = output_l12                                ## sum(Wx+b)
420     z = thresh*a                                  ## z = Vthr * a
421     v = (v_prev + z)                              ## integrate mem_potentail

```

```

422 spike_timestep = []                ## to store spikes for current
    timestep
423 for l in range (z.shape[0]):
424     if ((v[l]-thresh) >= 0):        ## compare with thresh
425         spike_timestep.append(1)    ## spike
426         v[l] = v[l] - thresh        ## subtract mem_potential
427     else:
428         spike_timestep.append(0)
429 spike_list.append(spike_timestep)
430 v_prev = v.copy()
431 spike_l12 = np.array(spike_list)
432 print('spikes of layer_12: \n', spike_l12.shape)
433 print('spikes of layer_12: \n', spike_l12)
434
435
436 ### dense _ layer2 ----- layer 13
437 num_neurons_l13 = 84
438 weights_l13=np.load("/amuneeb/proj/chb-mit/snn_toolbox/
    simple_network_py_implementation/net_eeg_conv_imp/w_dense_2.npy")
439 biases_l13 = np.array ([-4.63681947e-03,  3.77070904e-03,  7.38779642e-03,
    7.60984747e-03,
440 5.89874061e-03, -8.48719012e-03,  1.12406835e-02,  2.16511986e-03,
441 1.79820117e-02,  2.98310611e-02,  8.99855047e-03, -2.86474684e-03,
442 0.00000000e+00,  2.24189996e-03,  2.29859166e-02,  1.14884831e-04,
443 -8.20845552e-03,  2.08682162e-04,  3.24500501e-02,  2.91870558e-04,
444 -3.49193555e-03, -4.95689642e-03,  2.01518144e-02, -2.72922334e-03,
445 -2.15328811e-03,  4.64394409e-03, -5.99887129e-03, -2.42604199e-03,

```

```

446 -5.19911572e-03, -6.87270984e-03, 2.08941405e-03, 1.38646550e-02,
447 5.06390166e-03, 1.94915067e-02, -1.60766789e-03, -1.42720528e-03,
448 -5.82066691e-03, 1.66413113e-02, -3.39099992e-04, 1.23159569e-02,
449 2.11493969e-02, 1.68305356e-02, -1.05989194e-02, 7.56980339e-03,
450 1.63009781e-02, -4.22499795e-03, -1.10558409e-03, 3.25271413e-02,
451 1.05673171e-04, 1.16314311e-02, 9.88232903e-04, 3.90085531e-03,
452 1.44925462e-02, -2.47723213e-03, 1.86881598e-03, 2.78021507e-02,
453 -1.93179108e-03, 1.08054737e-02, 2.37229051e-05, 8.65442026e-03,
454 2.31688400e-03, -9.24572907e-03, 7.48282718e-03, 9.07519460e-03,
455 1.38185527e-02, 1.06354058e-02, 1.11667942e-02, -4.46523249e-04,
456 -2.79341545e-03, 8.70064134e-04, 9.83607862e-03, 2.57344008e-03,
457 -8.37018073e-04, -3.07562761e-04, -4.28229244e-03, 3.51762515e-03,
458 1.09176477e-03, -3.96167906e-03, 2.63773222e-02, -2.86396686e-03,
459 3.10602915e-02, 1.67376585e-02, 1.51891634e-02, 1.93437748e-02])
460
461 # dense layer with thresholding and spike generation
462 v = np.zeros(num_neurons_l13)
463 v_prev = np.zeros(num_neurons_l13)
464 spike_list = []
465 for k in range (sim_time//dt):
466     input_l13 = spike_l12[k]
467     output_l13 = dense_layer_relu(input_l13, weights_l13, biases_l13,
468                                   num_neurons_l13)
469     a = output_l13                                ## sum(Wx+b)
470     z = thresh*a                                  ## z = Vthr * a
471     v = (v_prev + z)                              ## integrate mem_potential

```

```

471 spike_timestep = []                ## to store spikes for current
    timestep
472 for l in range (z.shape[0]):
473     if ((v[l]-thresh) >= 0):        ## compare with thresh
474         spike_timestep.append(1)    ## spike
475         v[l] = v[l] - thresh        ## subtract mem_potential
476     else:
477         spike_timestep.append(0)
478 spike_list.append(spike_timestep)
479 v_prev = v.copy()
480 spike_l13 = np.array(spike_list)
481 print('spikes of layer_13: \n', spike_l13.shape)
482 print('spikes of layer_13: \n', spike_l13)
483
484
485 ### dense _ layer3 ----- layer 14
486 num_neurons_l14 = 2
487 weights_l14=np.load("/amuneeb/proj/chb-mit/snn_toolbox/
    simple_network_py_implementation/net_eeg_conv_imp/w_dense_3.npy")
488 biases_l14 = np.array ([-0.03762753,  0.0376226 ])
489 # Dense layer with thresholding and spike generation
490 v = np.zeros(num_neurons_l14)
491 v_prev = np.zeros(num_neurons_l14)
492 spike_list = []
493 for k in range (sim_time//dt):
494     input_l14 = spike_l13[k]

```

```

495 output_l14 = dense_layer_relu(input_l14, weights_l14, biases_l14,
    num_neurons_l14)
496 a = output_l14                                ## sum(Wx+b)
497 z = thresh*a                                  ## z = Vthr * a
498 v = (v_prev + z)                              ## integrate mem_potential
499 spike_timestep = []                            ## to store spikes for current
    timestep
500 for l in range (z.shape[0]):
501     if ((v[l]-thresh) >= 0):                    ## compare with thresh
502         spike_timestep.append(l)               ## spike
503         v[l] = v[l] - thresh                    ## subtract mem_potential
504     else:
505         spike_timestep.append(0)
506 spike_list.append(spike_timestep)
507 v_prev = v.copy()
508 spike_l14 = np.array(spike_list)
509 print('spikes of layer_l14: \n', spike_l14.shape)
510 print('spikes of layer_l14: \n', spike_l14)
511
512 # sum up the spike counts for each class over all time steps
513 spike_counts = np.sum(spike_l14, axis=0)
514 # find the class with the maximum spike count
515 winning_class = np.argmax(spike_counts)
516 print("Winning class:", winning_class)
517 print("Spike counts:", spike_counts)

```

Listing C.1: Python Jupyter Notebook Code for FSCNN development in Numpy

# **Appendix D**

## **Verilog code for FSCNN**

### **D.1 Verilog HDL implementation**

The layer by layer hardware implementation of finalized, customized fully spiking convolutional neural network is done using Verilog HDL language. The HDL-based implementation, verilog code ready for FPGA and subsequent ASIC implementation is given in the next section.

### **D.2 Verilog code for FSCNN**

All the required libraries need to be installed on a system running given code and the required paths are need to be updated to read and write data.



## D.3 Classifier

```
1 `timescale 1ns / 1ps
2 module classifier
3     (
4         CLK,
5         nRST,
6         data_in,
7         data_valid,
8         start_flag,
9         data_out,
10        ch_out,
11        ch_ready,
12        ch_seizure,
13        out_final,
14        out_data,
15        spike_out,
16        pool_done_l1_400
17    );
18
19    input    CLK;
20    input    nRST;
21    input    data_in;
22    output   data_valid;
23    output   start_flag;
24    output   ch_ready;
25    output   ch_seizure;
26    output   spike_out;
```

```

27  output    pool_done_l1_400;
28  output          [7:0] data_out;
29  output          [11:0] out_data;
30  output signed [11:0] ch_out;
31  output signed [11:0] out_final;
32
33  wire  data_valid;
34  wire  pool_done_l1_8_400;
35  wire [8:0] count_pool1;
36
37  reg [11:0] W1_1 = 12'b1_111_0100_1100; // -180
38  reg [11:0] W2_1 = 12'b1_110_0100_1101; // -435
39  reg [11:0] W3_1 = 12'b1_111_0101_1000; // -168
40  reg [11:0] W4_1 = 12'b1_111_0001_1101; // -227
41  reg [11:0] W5_1 = 12'b1_110_1001_0111; // -361
42  reg [11:0] W6_1 = 12'b0_000_1101_1011; // 219
43  reg [11:0] W7_1 = 12'b1_110_0101_1110; // -418
44  reg [11:0] W8_1 = 12'b1_110_1000_0111; // -377
45
46  reg [11:0] W1_2 = 12'b0_010_0001_0000; // 528
47  reg [11:0] W2_2 = 12'b0_001_1110_1101; // 493
48  reg [11:0] W3_2 = 12'b0_000_0111_0100; // 116
49  reg [11:0] W4_2 = 12'b1_111_1101_1101; // -35
50  reg [11:0] W5_2 = 12'b1_111_1110_0110; // -26
51  reg [11:0] W6_2 = 12'b1_110_1010_1111; // -337
52  reg [11:0] W7_2 = 12'b0_000_1001_1000; // 152
53  reg [11:0] W8_2 = 12'b1_101_1111_0011; // -525

```

```

54
55 reg [11:0] W1_3 = 12'b0_001_1110_0101; // 485
56 reg [11:0] W2_3 = 12'b0_001_0000_0110; // 262
57 reg [11:0] W3_3 = 12'b1_111_0101_1001; // -167
58 reg [11:0] W4_3 = 12'b1_111_0011_1110; // -194
59 reg [11:0] W5_3 = 12'b0_000_0110_1111; // 111
60 reg [11:0] W6_3 = 12'b1_110_1010_1001; // -343
61 reg [11:0] W7_3 = 12'b0_010_1110_0001; // 737
62 reg [11:0] W8_3 = 12'b1_111_0001_1011; // -229
63
64 reg [11:0] W1_4 = 12'b1_101_1011_0000; // -592
65 reg [11:0] W2_4 = 12'b1_111_1110_1011; // -21
66 reg [11:0] W3_4 = 12'b1_111_1101_0110; // -42
67 reg [11:0] W4_4 = 12'b0_001_0100_1111; // 335
68 reg [11:0] W5_4 = 12'b1_111_0011_0001; // -207
69 reg [11:0] W6_4 = 12'b1_111_0010_1010; // -214
70 reg [11:0] W7_4 = 12'b1_101_0001_1111; // -737
71 reg [11:0] W8_4 = 12'b1_110_1100_1101; // -307
72
73 reg [11:0] b1 = 12'b0_011_1110_1000; // 1000
74 reg [11:0] b2 = 12'b1_111_0010_1100; // -212
75 reg [11:0] b3 = 12'b1_110_0011_1101; // -451
76 reg [11:0] b4 = 12'b0_011_0011_0100; // 820
77
78 // Passing Channel's information for the seizure detection
79 data_in_reg input_data(
80 .CLK(CLK),

```

```

81 .nRST(nRST),
82 .data_in(data_in),
83 .data_valid(data_valid),
84 .start_flag(start_flag),
85 .data_out(data_out),
86 .pool_done_l1_400(pool_done_l1_400),
87 .count_pool1(count_pool1),
88 .pool_done_l1_8_400(pool_done_l1_8_400)
89 );
90
91 seizure_detect SEIZURE_1(
92     .CLK(CLK),
93     .nrst(nRST),
94     .data_in(data_out),
95     .W8_1(W8_1), .W7_1(W7_1), .W6_1(W6_1), .W5_1(W5_1),
96     .W4_1(W4_1), .W3_1(W3_1), .W2_1(W2_1), .W1_1(W1_1),
97     .W8_2(W8_2), .W7_2(W7_2), .W6_2(W6_2), .W5_2(W5_2),
98     .W4_2(W4_2), .W3_2(W3_2), .W2_2(W2_2), .W1_2(W1_2),
99     .W8_3(W8_3), .W7_3(W7_3), .W6_3(W6_3), .W5_3(W5_3),
100    .W4_3(W4_3), .W3_3(W3_3), .W2_3(W2_3), .W1_3(W1_3),
101    .b1(b1), .b2(b2), .b3(b3), .b4(b4),
102    .out(ch_out),
103    .data_valid(data_valid),
104    .ready(ch_ready),
105    .seizure(ch_seizure),
106    .out_final(out_final),
107    .out_data(out_data),

```

```

108         .spike_out(spike_out)
109     );
110
111 max_pool_1 uut_max_pool1(
112     .CLK(CLK),
113     .nRST(nRST),
114     .data_in_frm_conv(spike_out),
115     .counter(count_pool1),
116     .data_valid(data_valid),
117     .pool_done_l1_8_400(pool_done_l1_8_400)
118 );
119 endmodule

```

Listing D.1: Verilog code for FSCNN development, classifier

## D.4 Data input register

```

1 `timescale 1ns / 1ps
2 module data_in_reg
3     (
4         input wire CLK,
5         input wire nRST,
6         input wire data_in,
7         output wire start_flag,
8         output reg data_valid,
9         output reg pool_done_l1_400,
10        output reg pool_done_l1_8_400

```

```

11     output reg [7:0] data_out,
12     output reg [8:0] count_pool1,
13 );
14
15 reg [49:0] mem [255:0];    // Buffer to store input data of size (256 *
    50) for 1-sec window EEG signal
16 reg [9:0] num_neuron = 0;  // 256 neurons at input
17 reg [5:0] neuron_adr = 0;  // 50 location of each neuron
18 reg [5:0] count;          // counter to count and monitor the location of
    neuron from 50
19 reg [7:0] count_neuron;    // counter to count and monitor the neuron from
    256
20 reg [3:0] count_pool_11_8_400 = 4'b0;    // to count pool window size
21 wire      pool_done_11_8_400;            // pooling done signal
22
23 always @(posedge CLK or negedge nRST)
24 begin
25     if (nRST) begin
26         // Reset memory and address
27         neuron_adr <= 6'b0;
28         num_neuron <= 10'b0;    // changed it to 10 bit to see changes
29         count <= 6'b0;
30         count_neuron <= 8'b0;
31         count_pool1 <= 9'b0;
32     end else begin
33         // Store data into memory
34         mem[num_neuron][neuron_adr] <= data_in;    // Data buffer

```

```

35
36 // Increment address
37 if (neuron_adr == 49) begin // to check the end of 1-sec window
38     neuron_adr <= 6'b0;
39     // Increment num_neuron if neuron_adr reaches its maximum value
40     if (num_neuron == 255) begin // to check neuron address, from 256
41         num_neuron <= 10'b0; // to make it back to 0
42     end else begin
43         num_neuron <= num_neuron + 1; // incrementer for num_neuron
44     end
45 end else begin
46     neuron_adr <= neuron_adr + 1;
47 end
48 if (num_neuron >= 8) begin // for 1st conv layer kernel size is 1*8 so
49     // reaches to 8 the data from AFE is considered as valid
50     // to be precessed further
51     data_valid <= 1;
52 end else begin
53     data_valid <= 0;
54 end
55 end
56 end
57
58 assign start_flag = data_valid; // flag indicator to tell that
59     data is valid

```

```

60 // start operations on data now
61 // Output data from memory
62 always @(posedge CLK)
63 begin
64     if(start_flag) begin
65         data_out[0] <= mem[0+count_neuron][count];
66         data_out[1] <= mem[1+count_neuron][count];
67         data_out[2] <= mem[2+count_neuron][count];
68         data_out[3] <= mem[3+count_neuron][count];
69         data_out[4] <= mem[4+count_neuron][count];
70         data_out[5] <= mem[5+count_neuron][count];
71         data_out[6] <= mem[6+count_neuron][count];
72         data_out[7] <= mem[7+count_neuron][count];
73         if (count_pool1 <= 399) begin // counter to check that, when
            max pooling layer is done by choosing max value from all 8 neurons from
            the output of 1st Conv layer
74             count_pool1 <= count_pool1 +1;
75         end else begin
76             count_pool1 <= 9'b0;
77             count_pool_11_8_400 <= count_pool_11_8_400 + 1;
78         end
79         if (count <= 49) begin
80             count <= count+1;
81         end else begin
82             count <= 6'b0;
83             count_neuron <= count_neuron + 1;
84         end

```



```

85     end else begin
86         data_out[7:0] <= 8'bx;
87     end
88 end
89
90 assign pool_done_l1_8_400 = (count_pool_l1_8_400 == 4'd8)? 1'b1 : 1'b0;
91 endmodule

```

Listing D.2: Verilog code for input data register

## D.5 Spiking convolution layer

```

1  `timescale 1ns / 1ps
2  module seizure_detect
3      (
4          CLK,
5          nrst,
6          data_in,
7          W8_1,W7_1,W6_1,W5_1,W4_1,W3_1,W2_1,W1_1,
8          W8_2,W7_2,W6_2,W5_2,W4_2,W3_2,W2_2,W1_2,
9          W8_3,W7_3,W6_3,W5_3,W4_3,W3_3,W2_3,W1_3,
10         b1,b2,b3,b4,
11         out,
12         ready,
13         seizure,
14         out_final,
15         data_valid,

```

```

16         out_data,
17         spike_out
18     );
19     input CLK;
20     input nrst;
21     input ready;
22     input data_valid;
23     input [7:0] data_in;          //in1,in2,in3,in4,in5,in6,in7,in8;
24     input [11:0] W8_1,W7_1,W6_1,W5_1,W4_1,W3_1,W2_1,W1_1;          // weights
        kernel 1
25     input [11:0] W8_2,W7_2,W6_2,W5_2,W4_2,W3_2,W2_2,W1_2;          // weights
        kernel 2
26     input [11:0] W8_3,W7_3,W6_3,W5_3,W4_3,W3_3,W2_3,W1_3;          // weights
        kernel 3
27     input [11:0] b1,b2,b3,b4;          // Biases
28     output seizure;
29     output reg spike_out;
30     output signed [11:0] out;
31     output reg signed [11:0] out_data;
32     output reg signed [11:0] out_final;
33
34     wire signed [11:0] temp_wire;
35     wire signed [11:0] data_to_adder1;
36     wire signed [11:0] data_to_adder2;
37     wire signed [11:0] data_to_adder3;
38     wire signed [11:0] data_to_adder4;
39     wire signed [11:0] data_to_adder5;

```

```

40 wire signed [11:0] data_to_adder6;
41 wire signed [11:0] data_to_adder7;
42 wire signed [11:0] data_to_adder8;
43
44 reg signed [11:0] temp_out= 12'b0;
45 reg [5:0] count;
46 reg seizure;
47 reg [2:0] bank_sel;
48 reg [14:0] mem [7:0];
49 reg signed [11:0] accumulator_reg;
50
51 // Multiplexer which acts as a data gating to send//
52 // weights only to be accumulated when data at the//
53 // inout is 1 otherwise send 0 for now but will skip//
54 // addition latter on in new version ///////////////
55 assign data_to_adder1 = (data_in[0]) ? W1_1 : 12'b0;
56 assign data_to_adder2 = (data_in[1]) ? W2_1 : 12'b0;
57 assign data_to_adder3 = (data_in[2]) ? W3_1 : 12'b0;
58 assign data_to_adder4 = (data_in[3]) ? W4_1 : 12'b0;
59 assign data_to_adder5 = (data_in[4]) ? W5_1 : 12'b0;
60 assign data_to_adder6 = (data_in[5]) ? W6_1 : 12'b0;
61 assign data_to_adder7 = (data_in[6]) ? W7_1 : 12'b0;
62 assign data_to_adder8 = (data_in[7]) ? W8_1 : 12'b0;
63
64 // Addition of W1'X1 + W2'X2 + W3'X3 + .....
65 add_wx WX_sum_block
66 (

```

```

67     .in1(data_to_adder1),
68     .in2(data_to_adder2),
69     .in3(data_to_adder3),
70     .in4(data_to_adder4),
71     .in5(data_to_adder5),
72     .in6(data_to_adder6),
73     .in7(data_to_adder7),
74     .in8(data_to_adder8),
75     .bias(b1),
76     .out(out)
77 );
78 assign temp_wire = out;
79 ///////////////////////////////////////////////////
80
81     ***** //////////////////////////////////*
82
83 ///// Reset accumulator to 0 on reset signal
84 always @(posedge CLK or negedge nrst)
85 begin
86     if (!nrst) begin
87         accumulator_reg <= 0;
88         count <= 6'b0;
89         temp_out <= 12'b0;
90         spike_out <= 0;
91     end else if (count>0 && count < 50 && data_valid) begin
92         accumulator_reg = accumulator_reg + temp_wire;
93     end
94 end

```

```

93
94 always @(posedge CLK or negedge nrst)
95 begin
96     if (!nrst) begin
97         count <= 6'b0;
98         temp_out <= 12'b0;
99     end else begin
100         if (count < 50 & data_valid) begin
101             count <= count +1;
102         end else begin
103             count = 6'b0;
104         end
105     end
106 end
107
108
109 // Resetting the membrane potential if it reaches to Threshold value//
110 // which is 1000 in this case to keep weights in integer domain //
111 always @(*)
112 begin
113     if(accumulator_reg >= 1000) begin
114         spike_out = 1;
115         accumulator_reg = accumulator_reg - 11'd1000;
116     if (accumulator_reg >= 0) begin                ///Relu
117         accumulator_reg = accumulator_reg;
118     end else begin
119         accumulator_reg = 0;

```

```

120     end
121 end else begin
122     spike_out = 0;
123 end
124 end
125
126 endmodule

```

Listing D.3: Verilog code for spiking convolution layer

## D.6 Max pooling layer

```

1 `timescale 1ns / 1ps
2 module max_pool_1(
3     CLK,
4     nRST,
5     data_in_frm_conv,
6     counter,
7     data_valid,
8     pool_done_l1_8_400
9 );
10 input    CLK;
11 input    nRST;
12 input    data_in_frm_conv;    // taking data from spike_out of conv1
13 input [8:0] counter;
14 input    pool_done_l1_8_400;
15 output    data_valid;

```

```

16
17 reg [49:0] mem1;    // memory in which spike data from conv layer is stored.
18 reg [49:0] mem2;    // memory in which spike data from conv layer is stored.
19 reg [49:0] mem3;    // memory in which spike data from conv layer is stored.
20 reg [49:0] mem4;    // memory in which spike data from conv layer is stored.
21 reg [49:0] mem5;    // memory in which spike data from conv layer is stored.
22 reg [49:0] mem6;    // memory in which spike data from conv layer is stored.
23 reg [49:0] mem7;    // memory in which spike data from conv layer is stored.
24 reg [49:0] mem8;    // memory in which spike data from conv layer is stored.
25
26 // counters for all the above memory locations //
27 // to keep the track of number of spike sin each //
28 // memory location //
29 reg [5:0] cnt1 = 6'b0;
30 reg [5:0] cnt2 = 6'b0;
31 reg [5:0] cnt3 = 6'b0;
32 reg [5:0] cnt4 = 6'b0;
33 reg [5:0] cnt5 = 6'b0;
34 reg [5:0] cnt6 = 6'b0;
35 reg [5:0] cnt7 = 6'b0;
36 reg [5:0] cnt8 = 6'b0;
37
38 reg [5:0] max_temp;
39 reg max_pool1_done;
40 wire [5:0] max_value;
41 reg [2:0] max_cnt;
42 wire [2:0] register_with_max_value;

```

```

43 wire [49:0] data_to_conv2;
44 reg [2:0] cnt_pool = 3'b0;
45 wire pool_done_l1;
46 wire pool_done_l1_8_400;
47 reg [3:0] count_pool1_8_400= 4'b0;
48
49 always @(posedge CLK)
50 begin
51     if (counter >= 0 && counter <= 49) begin
52         mem1[counter] <= data_in_frm_conv; // taking data(0,1) from conv1
53         if (data_in_frm_conv) begin // check if spike ==1
54             cnt1 <= cnt1 +1; // increment the counter
55         end else begin // else
56             cnt1 <= cnt1; // keep it same, do not increment counter
57             cnt1 <= cnt1; // keep it same, do not increment counter
58         end
59     end else if (counter >= 50 && counter <= 99) begin
60         mem2[counter-50] <= data_in_frm_conv;
61         if (data_in_frm_conv) begin
62             cnt2 <= cnt2 +1;
63         end else begin
64             cnt2 <= cnt2;
65         end
66     end else if (counter >= 100 && counter <= 149) begin
67         mem3[counter-100] <= data_in_frm_conv;
68         if (data_in_frm_conv) begin
69             cnt3 <= cnt3 +1;

```



```

70     end else begin
71         cnt3 <= cnt3;
72     end
73 end else if (counter >= 150 && counter <= 199) begin
74     mem4[counter-150] <= data_in_frm_conv;
75     if (data_in_frm_conv) begin
76         cnt4 <= cnt4 +1;
77     end else begin
78         cnt4 <= cnt4;
79     end
80 end else if (counter >= 200 && counter <= 249) begin
81     mem5[counter-200] <= data_in_frm_conv;
82     if (data_in_frm_conv) begin
83         cnt5 <= cnt5 +1;
84     end else begin
85         cnt5 <= cnt5;
86     end
87 end else if (counter >= 250 && counter <= 299) begin
88     mem6[counter-250] <= data_in_frm_conv;
89     if (data_in_frm_conv) begin
90         cnt6 <= cnt6 +1;
91     end else begin
92         cnt6 <= cnt6;
93     end
94 end else if (counter >= 300 && counter <= 349) begin
95     mem7[counter-300] <= data_in_frm_conv;
96     if (data_in_frm_conv) begin

```

```

97     cnt7 <= cnt7 +1;
98     end else begin
99         cnt7 <= cnt7;
100 end
101 end else if (counter >= 350 && counter <= 399) begin
102     mem8[counter-350] <= data_in_frm_conv;
103     if (data_in_frm_conv) begin
104         cnt8 <= cnt8 +1;
105     end else begin
106         cnt8 <= cnt8;
107     end
108 end else if (counter == 400) begin
109     cnt1 <= 6'b0;
110     cnt2 <= 6'b0;
111     cnt3 <= 6'b0;
112     cnt4 <= 6'b0;
113     cnt5 <= 6'b0;
114     cnt6 <= 6'b0;
115     cnt7 <= 6'b0;
116     cnt8 <= 6'b0;
117     cnt_pool <= cnt_pool+1;
118 end
119 end
120
121 always @*
122 begin
123     max_temp = 6'd0; // Initialize max_temp to all zeros

```

```
124 max_cnt = 3'd0; // Initialize max_cnt to an invalid value
125 if (counter == 399) begin
126     max_pool1_done = 1;
127     if (cnt1 > max_temp) begin
128         max_temp = cnt1;
129         max_cnt = 3'd0;
130     end
131     if (cnt2 > max_temp) begin
132         max_temp = cnt2;
133         max_cnt = 3'd1;
134     end
135     if (cnt3 > max_temp) begin
136         max_temp = cnt3;
137         max_cnt = 3'd2;
138     end
139     if (cnt4 > max_temp) begin
140         max_temp = cnt4;
141         max_cnt = 3'd3;
142     end
143     if (cnt5 > max_temp) begin
144         max_temp = cnt5;
145         max_cnt = 3'd4;
146     end
147     if (cnt6 > max_temp) begin
148         max_temp = cnt6;
149         max_cnt = 3'd5;
150     end
```

```

151  if (cnt7 > max_temp) begin
152      max_temp = cnt7;
153      max_cnt = 3'd6;
154  end
155  if (cnt8 > max_temp) begin
156      max_temp = cnt8;
157      max_cnt = 3'd7;
158  end
159  end else begin
160      max_temp = 6'd0;
161      max_cnt = 3'dx; // Reset max_cnt
162  end
163 end
164
165 assign pool_done_l1 = (cnt_pool == 3'd8)? 1'b1 : 1'b0;    // this will
    trigger the second convolution layer
166 assign pool_done_l1 = (counter == 9'd399)? 1'b1 : 1'b0;
167 assign max_value = max_temp;
168 assign register_with_max_value = max_cnt;
169
170 assign data_to_conv2 = (max_cnt == 0)?mem1:(max_cnt == 1)?mem2:(max_cnt ==
    2)?mem3:(max_cnt == 3)?mem4:(max_cnt == 4)?mem5:(max_cnt == 5)?mem6:(
    max_cnt == 6)?mem7:(max_cnt == 7)?mem8:50'bx;
171
172 conv_2 conv2_uut (
173     .CLK(CLK),
174     .nRST(nRST),

```

```

175     .counter(counter),
176     .data_valid(data_valid),
177     .data_to_conv2(data_to_conv2),
178     .pool_done_l1(pool_done_l1),
179     .pool_done_l1_8_400(pool_done_l1_8_400)
180 );
181 endmodule

```

Listing D.4: Verilog code for max pooling layer

## D.7 Add block for sign additions

```

1 `timescale 1ns / 1ps
2 module add_wx(
3     in1,in2,in3,in4,in5,in6,in7,in8,bias,
4     out
5 );
6     input  signed [11:0] in1,in2,in3,in4,in5,in6,in7,in8,bias;
7     output signed [11:0] out;
8
9     wire signed [11:0] out_tmp1,out_tmp2,out_tmp3,out_tmp4;
10    wire signed [11:0] out_tmp5,out_tmp6, out_tmp1_8;
11    sign_adder add_1_2 ( .a(in1), .b(in2), .cin(1'b0), .sum(out_tmp1) );
12    sign_adder add_3_4 ( .a(in3), .b(in4), .cin(1'b0), .sum(out_tmp2) );
13    sign_adder add_5_6 ( .a(in5), .b(in6), .cin(1'b0), .sum(out_tmp3) );
14    sign_adder add_7_8 ( .a(in7), .b(in8), .cin(1'b0), .sum(out_tmp4) );

```

```

15 sign_adder add_1to4 ( .a(out_tmp1[11:0]), .b(out_tmp2[11:0]), .cin(1'b0), .
    sum(out_tmp5) );
16 sign_adder add_5to8 ( .a(out_tmp3[11:0]), .b(out_tmp4[11:0]), .cin(1'b0), .
    sum(out_tmp6) );
17 sign_adder add_1to8 ( .a(out_tmp5[11:0]), .b(out_tmp6[11:0]), .cin(1'b0), .
    sum(out_tmp1_8) );
18 sign_adder add_1to8_b ( .a(out_tmp1_8[11:0]), .b(bias[11:0]), .cin(1'b0), .
    sum(out) );
19 endmodule

```

Listing D.5: Verilog code for sign adder

## D.8 Add block for sign adder

```

1 `timescale 1ns / 1ps
2 module sign_adder #(parameter BITS = 12)
3     (
4         a,
5         b,
6         cin,
7         sum
8     );
9     input signed [BITS-1:0] a;
10    input signed [BITS-1:0] b;
11    input cin;
12    output signed [BITS-1:0] sum;
13

```

```

14 wire cout;
15 wire c1,c2,c3;
16
17 adder_4bit uu1 (.a(a[3:0]), .b(b[3:0]), .cin(cin), .sum(sum[3:0]), .
    cout(c1));
18 adder_4bit uu2 (.a(a[7:4]), .b(b[7:4]), .cin(c1), .sum(sum[7:4]), .
    cout(c2));
19 adder_4bit uu3 (.a(a[11:8]), .b(b[11:8]), .cin(c2), .sum(sum[11:8]), .
    cout(c3));
20 endmodule

```

Listing D.6: Verilog code for adder

## D.9 4-bit adder

```

1 `timescale 1ns / 1ps
2 module adder_4bit(
3     a,
4     b,
5     cin,
6     sum,
7     cout
8 );
9 input    cin;
10 input    [3:0] a,b;
11 output    cout;
12 output    [3:0] sum;

```

```

13
14 wire [3:0] p,g,c;
15
16 assign p=a^b; //propagate
17 assign g=a&b; //generate
18 assign c[0]= cin;
19 assign c[1]= g[0] | (p[0]&c[0]);
20 assign c[2]= g[1] | (p[1]&g[0]) | p[1]&p[0]&c[0];
21 assign c[3]= g[2] | (p[2]&g[1]) | p[2]&p[1]&g[0] | p[2]&p[1]&p[0]&c[0];
22 assign cout= g[3] | (p[3]&g[2]) | p[3]&p[2]&g[1] | p[3]&p[2]&p[1]&g[0] | p
    [3]&p[2]&p[1]&p[0]&c[0];
23 assign sum=p^c;
24 endmodule

```

Listing D.7: Verilog code for 4-bit adder