University of Lille

Thesis financed by **Région Haut de France**

Graduate School ED MADIS-631

Laboratory CRIStAL

Thesis defended by **Aurélie Saulquin**

Academic Field **Computer Science**

FPGA-Based Neuromorphic Architecture for Spiking Neural Network Emulation

Directors

Director : **Prof. Samy Meftali**
CRIStAL, Univ. de Lille

Co-Director : **Prof. Pierre Boulet**
CRIStAL, Univ. de Lille

Jury

Referees
Prof. Sylvain Saïghi
IMS Bordeaux, Univ. de Bordeaux
Prof. Frédéric Pétrot
TIMA Lab, Univ. de Grenoble-Alpe

Examiners
Prof. Virginie Hoel
IEMN, Univ. de Lille

Université de Lille
42 rue Paul Duez
59000 Lille

www.univ-lille.fr

Université de Lille

Thèse Financé par **Région Haut de France**

Ecole Graduée ED MADIS-631

Laboratoire CRIStAL

Thèse défendue par **Aurélie Saulquin**

Domaine Universitaire **Informatique Appliquée**

Architecture Neuromorphique Basé sur FPGA pour l'Emulation de Réseaux de Neurones Impulsionnels

---

soutenue le 12 décembre 2025

Directeurs

Directeur : **Prof. Samy Meftali**
CRIStAL, Univ. de Lille

Co-Directeur : **Prof. Pierre Boulet**
CRIStAL, Univ. de Lille

Jury

Rapporteurs
Prof. Sylvain Saïghi
IMS Bordeaux, Univ. de Bordeaux
Prof. Frédéric Pétrot
TIMA Lab, Univ. de Grenoble-Alpe

Examinateurs
Prof. Virginie Hoel
IEMN, Univ. de Lille

Université de Lille
42 rue Paul Duez
59000 Lille

www.univ-lille.fr

Je dédie ma thèse à Clélia Simon

Remerciements

Je souhaite tout d'abord exprimer ma profonde gratitude envers le Professeur Samy Meftali qui a dirigé ma thèse et à mon co-encadrant, le Professeur Pierre Boulet, pour leur encadrement rigoureux et le soutien durant ces trois années de doctorat. Leurs conseils avisés et leur accompagnement, aussi bien scientifique qu'humain, m'ont permis d'affiner ma démarche scientifique. Je leur suis particulièrement reconnaissante pour la confiance qu'ils ont placée en moi dès le début de mon stage de Master il y a de cela quatre ans.

Je tiens également à remercier les membres du jury qui ont accepté d'évaluer mes travaux de thèse. J'aimerais remercier les rapporteurs de ma thèse, le Professeur Frédéric Pétrot et le Professeur Sylvain Saïghi pour leurs observations et recommandations qui ont permis d'améliorer et d'enrichir mon manuscrit. J'aimerais également remercier la Professeure Virginie Hoel pour avoir accepté d'être examinatrice de ma thèse.

Je remercie aussi mes collègues de travail, notamment Thomas, Mazdack, Soukaina Dima ainsi qu'Eva pour nos échanges scientifiques stimulants, de même que pour les moments conviviaux partagés au cours de ces années. Ma reconnaissance va également au personnel du laboratoire CRIStAL, de l'école doctorale MADIS et de la plateforme de recherche IRCICA, dont le professionnalisme a grandement contribué au bon déroulement de mes travaux de recherches dans un environnement de travail fonctionnel et stimulant propice à l'épanouissement scientifique.

Je souhaite aussi témoigner ma profonde reconnaissance envers ma famille, dont le soutien inconditionnel m'a permis de faire face aux difficultés. Merci à mes parents, Gérard et Sylvette, pour leur présence, leur confiance et leur bienveillance durant mon long parcours universitaire. Je remercie par ailleurs mon frère Anthony, ainsi que ses enfants Abigaëlle et Axel, pour leur encouragement et leur affection qui m'ont apporté du réconfort tout au long de mes études. Ma reconnaissance s'adresse également à ma cousine Cécile, à qui j'ai pu me confier et dont le soutien m'a donné force et espoir dans les périodes les plus éprouvantes.

Enfin, je tiens à remercier les communautés LGBTrans Way of Life et Marie's Cottage pour leur soutien et les discussions que j'ai pu avoir durant les derniers mois de ma thèse qui m'ont réchauffé le cœur durant mes moments d'hésitation. Mais surtout, je tiens à remercier la communauté de la Magic Box, qui m'a accompagné durant ces deux dernières années et avec qui j'ai pu partager mes peines et mes joies. Je suis particulièrement reconnaissante envers Antoine, Jordan, Salomé, Marie-Lou, Christophe et Stéphane pour tous ces instants de bonheur partagés en ligne ou en face-à-face. Surtout, j'aimerais remercier Clélia, à qui je dédie cette thèse, qui a été présente à mes côtés dans les meilleurs moments comme dans les épreuves les plus dures. Je la remercie de m'avoir aidé à me relever lorsque j'étais au plus bas et m'a soutenue sans faille jusqu'à l'aboutissement de ce travail. Au détour d'un discord bancal, j'ai fait la plus belle rencontre de ma vie.

This thesis has been prepared at the following research units.

CRIStAL
UMR 9189
Université de Lille - Campus scientifique
Bâtiment ESPRIT
Avenue Henri Poincaré
59655 Villeneuve D'ascq
France
Website: https://www.cristal.univ-lille.fr/

IRCICA
USR 3380
CAMPUS Haute-Borne CNRS IRCICA-IRI-RMN
Parc Scientifique de la Haute Borne
50 Avenue Halley
BP 70478
59658 Villeneuve D'ascq
France
Website: https://ircica.univ-lille.fr/

Jury Composition

Refrees

Prof. Sylvain Saïghi
IMS Bordeaux, Univ. de Bordeaux
Prof. Frédéric Pétrot
TIMA Lab, Univ. de Grenoble-Alpe

Examiners

Prof. Virginie Hoel
IEMN, Univ. de Lille

Thesis Directors

Director : **Prof. Samy Meftali**
CRIStAL, Univ. de Lille
Co-Director : **Prof. Pierre Boulet**
CRIStAL, Univ. de Lille

Short Abstract

Abstract

Artificial Intelligence (AI), and particularly Artificial Neural Networks (ANNs) have become central to modern computing with remarkable performances for complex task resolution. Their integration on embedded systems establishes the Artificial Intelligence of Things (AIoT).
However, ANNs remain challenging to deploy on such systems due to their complexity. To address this, researchers explore specialized hardware, model compression, and cloud solutions. Neuromorphic Computing offers an alternative solution, drawing inspiration from the brain by using Spiking Neural Networks (SNNs).
Although promising, neuromorphic chips are limited and poorly reconfigurable, making Field Programmable Gate Arrays (FPGAs) an interesting target for research. In this context, we developed ModNEF, an open-source modular FPGA architecture for SNN inference through the interconnection of independent modules, offering high implementation flexibility. We validated our architecture on standard neuromorphic datasets and on a use case for sperm whale detection in the Mediterranean Sea, demonstrating its capacity to perform classification tasks for embedded applications.
Keywords *: Neuromorphic accelerator, Spiking Neural Networks, Artificial Intelligence, Embedded Artificial Intelligence, Embedded Systems, Edge Computing, Parallel Architecture, Digital Architecture, Neuromorphic Architecture, FPGA*.

Résumé

L'intelligence artificielle (IA), et plus particulièrement les Réseaux de Neurones Artificiels (ANNs) ont pris une place centrale dans l'informatique moderne avec des performances remarquables pour la résolution de tâches complexes. Leur intégration dans des systèmes embarqués a donné naissance à l'Intelligence Artificielle des Objets (AIoT).
Cependant, le déploiement d'ANNs sur ces systèmes demeure difficile dû à leur complexité. Pour relever ce défi, les chercheurs étudient le recours à du matériel spécialisé, à la compression de modèles et à des solutions cloud. Le Calcul Neuromorphique offre une solution viable, s'inspirant du cerveau par l'utilisation de Réseaux de Neurones Impulsionnels (SNNs).
Bien que prometteuses, les puces neuromorphiques sont souvent limitées et peu reconfigurables, plaçant les Field Programmable Gate Arrays (FPGAs) comme des plateformes privilégiées pour la recherche. C'est dans ce contexte que nous avons développé ModNEF, une architecture modulaire libre sur FPGA permettant l'inférence de SNN complexes via l'interconnexion de modules indépendants, offrant ainsi une grande flexibilité d'implémentation. Nous avons validé notre architecture sur des jeux de données neuromorphiques standards ainsi que sur une étude de cas sur la détection de cachalots en mer Méditerranée, démontrant sa capacité à réaliser des tâches de classification pour des applications embarquées.
Mots Clés *: Accélérateur Neuromorphique, Réseaux de Neurones Impulsionnels, Intelligence Artificielle, Intelligence Artificielle Embarquée, Systèmes Embarqués, Calcule en Prériphérie, Architecture Parallèle, Architecture Numérique, Architecture Neuromorphique, FPGA*.

Abstract

Artificial Intelligence (AI), and particularly Artificial Neural Networks (ANNs) have become central to modern computing, demonstrating remarkable performances in complex task resolution. These promising results have motivated researchers in embedded computing to integrate ANN into embedded and resource-constrained systems, thereby establishing the emerging field, Artificial Intelligence of Things (AIoT). This new paradigm aims to combine artificial intelligence and interconnected devices while addressing the specific constraints of embedded systems.

However, the integration of ANNs into resource-constrained embedded systems presents significant challenges due to their algorithmic complexity and their computational requirements. A first solution is to offload data processing into the cloud. This approach introduces critical concerns regarding data privacy and execution latency. To enable in-edge computing, researchers have explored complementary solutions. A first approach is to develop new hardware architectures specialized for ANNs execution, such as Tensor Processing Units (TPUs), Application-Specific Integrated Circuits (ASICs), or Field Programmable Gate Arrays (FPGAs), more power efficient than conventional Graphic Processing Units (GPUs). In addition to hardware solutions, model compression methods are developed to minimize the computational complexity of models, thereby reducing their inference cost.

Another approach, proposed by Mead in 1988, draws inspiration on biological neural networks, particularly the human brain, to develop energy-efficient ANN systems. The neuromorphic computing paradigm represents information as temporal sequences of discrete events called spikes. These spikes are processed by Spiking Neural Networks (SNNs), which consist of the interconnection of spiking neuron models that accumulate incoming spikes from synapses into an internal variable, called membrane voltage. If the membrane voltage exceeds a threshold value, the neuron propagates output spikes through axonal connections.

Neuromorphic computing was initially conceived as an analog technology, reflecting the inherently analog nature of biological neuron systems. However, the development of fully analog SNNs presents significant challenges. Thereby, research starts to develop hybrid analog/digital and fully digital ASIC. While these specialized chips offer promising solutions for low-power embedded applications, their functionalities can be limited by their neuron model, their online learning rules, and their reconfigurability. In this context, FPGAs emerge as a compelling alternative to explore new architectural designs and deployment of low-power embedded applications.

However, many FPGA-based SNN emulators developed by the research community remain close source. At the beginning of the thesis in 2022, no open-source FPGA emulator was available. Today, in 2025, three emulators have been released under open-source licenses. In the aim to develop FPGA-based SNN implementation within our research team for embedded applications, we have developed ModNEF, an open source FPGA modular architecture. ModNEF is based on the interconnection of independent modules to emulate complex SNNs. Our architecture is highly flexible, offering several neuron models and different emulation strategies

to provide users with a high level of control in hardware implementations. In addition, ModNEF proposes a comprehensive software framework that supports the complete deployment pipeline, from model training to on-board implementation.
We have validated our architecture through comprehensive evaluation on standard neuromorphic datasets such as MNIST, N-MNIST, SHD, and DVS Gesture. We have analyzed and identified current architectural limitations and presented solutions proposed by ModNEF to overcome hardware constraints. Additionally, we have introduced quantization tools within the software framework to minimize the power consumption while maintaining software-level accuracy. This includes an in-depth study on Post-Training Quantization (PTQ) impacts and the presentation of the Quantization Aware Training (QAT) algorithm developed for ModNEF. Finally, we present a novel application for biodiversity monitoring in the Mediterranean Sea for sperm whale detection with a complete SOC integration. This case study demonstrates ModNEF's ability to perform low-power embedded classification tasks for environmental monitoring.

Résumé

L'Intelligence Artificielle (IA), en particulier les Réseaux de Neurones Artificiels (ANN), a pris une place centrale dans l'informatique moderne, en démontrant des performances remarquables dans la résolution de tâches complexes. Ces perspectives prometteuses ont motivé les chercheurs en informatique embarquée à intégrer les ANNs dans des systèmes embarqués et des systèmes contraints en ressources, créant ainsi le domaine émergent de l'Intelligence Artificielle des Objets (AIoT). Ce nouveau paradigme vise à combiner l'IA et les appareils connectés tout en tenant compte des contraintes spécifiques de ces systèmes.

Cependant, l'intégration de réseaux de neurones dans des systèmes embarqués à ressources limitées présente des défis importants, dus à la complexité algorithmique et aux besoins en calcul. Une première solution consiste à transférer le traitement dans le cloud. Cette approche soulève néanmoins des problématiques critiques en termes de protection des données et de latence d'exécution. Pour permettre le calcul en périphérie (*in-edge computing*), les chercheurs ont exploré plusieurs solutions complémentaires. Une première approche est le développement de nouvelles architectures matérielles spécialisées dans l'exécution de réseaux de neurones, comme les *Tensor Porcessing Units (TPU)*, les *Application Specific Integrated Circuits (ASIC)* ou les *Filed Programmable Gate Arrays, (FPGAs)*, plus efficaces énergétiquement que les *Graphic Processing Units (GPU)* conventionnels. En plus de ces solutions matérielles, des méthodes de compression de modèle ont été développées pour réduire la complexité calculatoire du modèle, réduisant ainsi le coût d'inférence.

Une autre approche, proposée par Mead en 1988, consiste à s'inspirer des réseaux de neurones biologiques, en particulier du cerveau humain, pour développer des systèmes de réseaux de neurones énergétiquement efficaces. Le paradigme du calcul neuromorphique représente l'information comme des séquences temporelles d'événement discret appelées impulsions (*spikes*). Ces impulsions sont traitées par des Réseaux de Neurones Impulsionnels (*Spiking Neural Networks*, SNNs), constitués de l'interconnexion de modèles de neurones impulsionnels qui accumulent les impulsions entrantes issues des synapses dans une variable interne appelée la tension de membrane. Si la tension de membrane dépasse une tension de seuil, le neurone génère une impulsion de sortie à travers les connexions axonales.

Le calcul neuromorphique a d'abord été conçu comme une technologie analogique, reflétant la nature intrinsèquement analogue des systèmes neuronaux biologiques. Cependant, le développement de réseaux impulsionnels complètement analogiques représente des défis significatifs. Les recherches se sont ainsi orientées vers des puces hybrides analogiques/numériques ou entièrement numériques. Bien que ces circuits spécialisés offrent des solutions prometteuses pour des applications embarquées à basse consommation énergétique, leurs fonctionnalités sont souvent limitées par le modèle de neurones, leur capacité à faire de l'apprentissage en ligne et leur degré de reconfigurabilité. Dans ce contexte, les FPGA apparaissent comme une alternative convaincante pour explorer de nouvelles conceptions architecturales et déployer des applications embarquées à faible consommation.

Cependant, de nombreux émulateurs de SNN sur FPGA développés par la communauté scien-

tifique demeurent propriétaires. Au début de cette thèse, en 2022, aucun émulateur FPGA libre n'était disponible. Aujourd'hui, en 2025, trois émulateurs sous licences libres ont été publiés. Dans le cadre du développement d'implémentation de SNN sur FPGA au sein de notre équipe pour des applications embarquées, nous avons développé ModNEF, une architecture modulaire FPGA libre. ModNEF repose sur l'interconnexion de modules indépendants permettant d'émuler de complexes réseaux impulsionnels. Notre architecture se distingue par une grande flexibilité, proposant plusieurs modèles de neurones et différents algorithmes d'émulation, offrant ainsi à l'utilisateur un haut niveau de contrôle sur l'implémentation matérielle. De plus, ModNEF propose un environnement logiciel complet couvrant l'ensemble de la chaîne de déploiement, de l'entraînement du modèle à son implantation sur carte.
Nous avons validé notre architecture sur plusieurs jeux de données neuromorphiques standards comme MNIST, N-MNIST, SHD et DVS Gesture. Nous avons analysé et identifié les limitations architecturales actuelles et nous avons présenté les solutions intégrées à ModNEF pour dépasser les contraintes matérielles. De plus, nous avons introduit des outils de quantification dans l'environnement logiciel pour minimiser la consommation énergétique tout en préservant la précision des modèles. Cela inclut une étude approfondie de l'impact de la quantification post-entraînement (*Post-Training Quantization, PTQ*) ainsi que la présentation de l'algorithme d'entraînement avec prise en compte de la quantification (*Quantization-Aware Traning, QAT*) que nous avons développée pour ModNEF. Enfin, nous avons présenté une application innovante pour la surveillance de la biodiversité en mer Méditerranée pour la détection de cachalots avec une intégration complète sur puce (*System-on-Chip, SOC*). Cette étude de cas illustre la capacité de ModNEF à exécuter des tâches de classification embarquées dans un système embarqué à basse consommation énergétique pour la surveillance de l'environnement.

Keywords *: Neuromorphic accelerator, Spiking Neural Networks, Artificial Intelligence, Embedded Artificial Intelligence, Embedded Systems, Edge Computing, Parallel Architecture, Digital Architecture, Neuromorphic Architecture, FPGA*.
Mots clé *: Accélérateur Neuromorphique, Réseaux de Neurones Impulsionnels, Intelligence Artificielle, Intelligence Artificielle Embarquée, Systèmes Embarqués, Calcule en Prériphérie, Architecture Parallèle, Architecture Numérique, Architecture Neuromorphique, FPGA*.

Publications

Articles

1. Aurélie Saulquin, Mazdak Fatahi, Pierre Boulet, and Samy Meftali. 2025. ModNEF: An Open Source Modular Neuromorphic Emulator for FPGA for Low-Power In-Edge Artificial Intelligence. ACM Trans. Archit. Code Optim. 22, 2, Article 73 (June 2025), 24 pages. https://doi-org.ressources-electroniques.univ-lille.fr/10.1145/3730581

Contents

Acronyms

R_{pa} Ratio Power Accuracy. xxxv–xxxviii, xlix, 111–113, 115, 118–121, 123, 124, 126, 127, 132, 133, 146–148

AC Alternative Current. 136

AER Address Event Representation. xliv, xlv, 35, 36, 38, 42–44, 48, 49, 51, 58, 60, 79, 91

AI Artificial Intelligence. ix, xi, xli, , 3, 4, 31, 69, 136, 166

AIoT Artificial Intelligence of Things. ix, xi, , 3, 4

AIS Arbiter-Induced Serialization. 48, 53, 90

ANN Artificial Neural Network. ix, xi, , 3–5, 9, 10, 16, 18, 20, 28, 29, 100, 116, 165

ASIC Application-Specific Integrated Circuit. xi, 4, 5, 22, 23

BLIF Beta Leaky Integrate and Fire. xxxi–xxxv, xliii, xlvi, xlix, 38–40, 61, 69, 74, 75, 86–88, 102, 104, 105, 107, 111–114, 118

BP Backpropagation. 17–20, 28, 29, 66

BPTT Backpropagation Through Time. 19, 66

BRAM Block Random Access Memory. xxxi–xxxvi, xxxix, xlvii, xlix, 42, 66, 68, 71, 73, 76, 78, 81–87, 89–91, 97, 111, 112, 115, 119–121, 123, 125, 126, 159, 160

CMOS Complementary Metal Oxide Semiconductor. 21, 22, 31, 135

CNN Convolutional Neural Network. xlii, 9, 15, 16, 20, 23, 28, 63

CORDIC COordinate Rotation DIgital Computer. 26, 27, 29, 165

CPU Central Processing Unit. xxxii, xxxiv, 5, 20, 23, 28, 67, 69–71, 75, 76, 79, 81, 82, 88, 90–92, 144, 158, 165

DAC Digital to Analog Converter. 21

DC Direct Current. 136

DDR Double Data RAM. 28, 42, 65

DFPQ Dynamic Fixed-Point Quantizer. xlviii, 108, 109, 112, 113

DSFQ Dynamic Scale Factor Quantizer. xlviii, l, 101, 103, 104, 107–109, 111–113, 116, 130–133

DSP Digital Signal Processing. xxxi–xxxv, xlix, 39, 66, 68, 70, 71, 75, 76, 78, 82–87, 89, 91, 111, 112, 119–121, 123, 160

DVS Dynamic Vision Sensor. 20, 74, 79

FF Flip-Flop. xxxii, xxxiii, 66, 68, 71, 75, 76, 78, 81–87, 89, 91, 97, 119–121, 160

FF-FC Feed Forward Fully Connected. xxxviii, xxxix, xli, xlii, 1, 14–16, 20, 23, 28, 34, 63, 69, 74, 77, 95, 157–160, 165

FIFO First In First Out. 49

FLOPS Floating-Point Operations Per Seconds. xlii, 3, 25

FPAA Field Programmable Analog Array. 20

FPGA Field Programmable Gate Array. ix, xi, xxxi–xxxiv, xxxvii–xxxix, xlvi, xlvii, li, , 1, 4–7, 9, 20, 23, 25–33, 37, 57, 58, 60, 61, 63, 65–71, 73, 75, 76, 78, 79, 81, 82, 84, 85, 87, 88, 90, 91, 93, 97, 98, 104, 108, 114, 116, 118, 120, 123, 125, 127, 135, 138, 142–149, 152, 154, 157–160, 163–165

FPQ Fixed-Point Quantizer. xxxv, xlviii, 101–108, 111–113, 117, 118, 130, 131, 133

FPT Full-Precision Training. xxxv–xxxvii, xlix–li, 116–121, 123–133, 142, 143, 157, 160

FSM Finite State Machine. xliv, xlv, 1, 42–44, 46–48, 50, 151, 152

GALS Globally Asynchronous Locally Synchronous. 22

GDDR Graphic Double Data Rate. 65

GPIO General Purpose Input/Output. li, 148, 149, 152, 153, 160

GPU Graphic Processing Unit. xi, xxxii, xxxiv, xxxvii, 3, 6, 20, 23, 28, 67–71, 75, 76, 79, 81, 82, 88, 90–92, 143–146, 148, 158, 165

HDF5 Hierarchical Data Format 5. 137

HH Hodgkin-Huxley. xlii, 11, 12, 25, 26

IF Integrate and Fire. xxxvii, xli, xliv, xlviii, 12, 13, 27, 30, 34, 37, 40, 41, 59, 60, 65, 86, 97, 105, 106, 109, 143, 146

IoT Internet of Things. 3, 22

IR Internal Recurrence. 90, 91

LIF Leaky Integrate and Fire. xli, xlii, 12, 13, 23–27, 30, 34, 37, 59, 60, 136

LIFO Last In First Out. 49

LSB Less Significant Bit. 35

LUT Look-Up Table. xxxi–xxxvi, xlvii, 26, 27, 29, 66, 68, 71, 75, 76, 78, 81–87, 89, 91, 97, 119–121, 160, 165

LUTRAM Look Up Table Random Access Memory. 66, 68, 71, 76, 78, 82, 84, 85, 87, 89, 91, 119, 121, 160

MMQ Min-Max Quantizer. xlviii, l, li, 101, 103, 104, 107–109, 111–113, 116, 130, 131, 133

ModNEF Modular Neuromorphic Emulator for FPGA. ix, xi, xii, xiv, xxxi–xxxiii, xxxvii, xliii, xlvi, xlvii, li, , 6, 7, 32–38, 41, 46, 47, 49, 51–53, 55–63, 65, 66, 69–71, 73, 75, 77–83, 86, 89, 92, 95, 97, 101, 103, 104, 114, 116, 117, 125, 131, 133–135, 138, 139, 142–146, 148, 149, 159–161, 163–166

MS `Merger+Splitter`. xlvi, 52–54, 90, 91

MSB Most Significant Bit. xlviii, 35, 102

MSE Mean Squared Error. xlviii, 104–109

MSTE Mean Spike Time Error. xlviii, 105–107, 109

NN Neural Network. 3, 4, 98, 99

NPU Neural Processing Unit. 26

OPS Operations Per Seconds. xli, 3, 4

PCIe Peripheral Component Interconnect Express. 49

PTQ Post-Training Quantization. xii, 59, 66, 67, 70, 75, 78, 81, 84, 85, 87, 88, 90, 97, 99–101, 104, 108, 112–115, 118, 120, 123, 125, 127, 132–134, 143, 158, 163, 164

PWL Piecewise linear. 29

QAT Quantization Aware Training. xii, xxxv–xxxvii, xlix–li, 59, 97, 99–101, 115–134, 142–144, 148, 160, 164

RAM Random Access Memory. xliv, 43

RNN Recurrent Neural Network. xli, xlii, 14–16, 19, 28

RRAM Resistive Random Access Memory. 21

SAIF Switching Activity Interchange Format. 156

SCE Spike Counter Error. xlviii, 105–107

SCNN Spiking Convolutional Neural Network. 28

SFPQ Static Fixed-Point Quantizer. xlviii, 108, 109, 112

SLIF Simplified Leaky Integrate and Fire. xxxi–xxxiv, xliii, xlvi, xlviii, xlix, 40, 41, 61, 69–71, 74–76, 86, 87, 95, 104, 105, 107–109, 112–114

SNN Spiking Neural Network. ix, xi, xlvi, li, , 4–6, 9–23, 25–34, 38, 41, 51, 60–62, 69, 70, 75, 88, 95, 97, 100, 101, 105, 107, 116, 133, 135, 136, 138, 160, 163–166

SOC System On Chip. xii, li, 1, 135, 138, 139, 146, 148, 150, 152, 155, 157, 160, 161, 164

SRAM Static Random Access Memory. 21

SRLIF Shift Register Leaky Integrate and Fire. xxxii–xxxiv, xliii, xlviii, xlix, 39, 40, 77, 79, 81, 86, 87, 89, 92, 104, 105, 107, 108, 112–115, 120, 143, 145

STBP Spatio-Temporal Backpropagation. 19

STDP Spike-Timing-Dependent Plasticity. xlii, 17, 18, 22, 29, 100, 165

TPU Tensor Processing Unit. xi

TTFS Time To First Spike. 13

UART Universal Asynchronous Receiver Transmitter. xxxiii, xxxv, xxxviii, xxxix, xliii, xlv, xlvii, 34–37, 49–51, 56–60, 68, 69, 80, 82, 90–95, 143, 145, 148, 149, 154–156, 158, 164

VHDL Verry High Speed Integrated Circuit Hardware Description Language. 33, 55–57, 61, 91, 153

List of Tables

List of Figures

1 Introduction and Motivation

Artificial Intelligence (AI), and particularly Artificial Neural Networks (ANNs), have rapidly evolved into a central place in both research and mainstream application, fueled by the democratization of generative AIs such as ChatGPT [139] or Mid Journey [128]. These technologies demonstrate remarkable performance in complex tasks, including environmental perception and decision-making, far surpassing the reach of traditional algorithms. Such advancements open promising perspectives for their integration into embedded Internet of Things (IoT) systems, giving rise to the emerging field of the Artificial Intelligence of Things (AIoT).

1.1 Embedded AI and their Limitations

The application range of AI, and particularly ANNs for embedded and IoT systems has expanded significantly in recent years. Well-established applications include autonomous vehicles [119], medical applications [195], robotics [17] and computer vision [174]. Additionally, emerging fields like computer-assisted agriculture [25] or industrial automation [45] demonstrate the growing versatility of AI-based systems.
However, the integration of ANNs into embedded devices presents significant challenges. The high algorithmic complexity of Neural Networks (NNs) and the substantial computational requirements [31, 213] often exceed the capabilities of resource-constrained systems. Due to the mathematical representation of ANNs, Graphic Processing Units (GPUs) offer a viable solution for NN deployment in embedded environments [11]. However, their high-power consumption remains a critical limitation for low-power in-edge applications.
Figure 1.1 represents the relation between power consumption (W) and computing capacity (Operations Per Seconds (OPS))[1] for the Jetson product family of embedded GPUs developed by Nvidia. Overall, power consumption increased with the available computing capacity from 5 W for the least computing efficient to 40 W for the most capable device.
The calculation capacity limits the size of the NN that can be implemented. Larger models require more computational resources, which increase power consumption. Consequently, high-performance embedded GPUs are unsuitable for applications with strict power budgets, while low-power devices cannot support large or computationally intensive models.

1. According to Nvidia's documentation, computing throughput is expressed using two different metrics: TFlops for Tera Floating-Point Operations Per Seconds (FLOPS) and TOPS for Tensor OPS

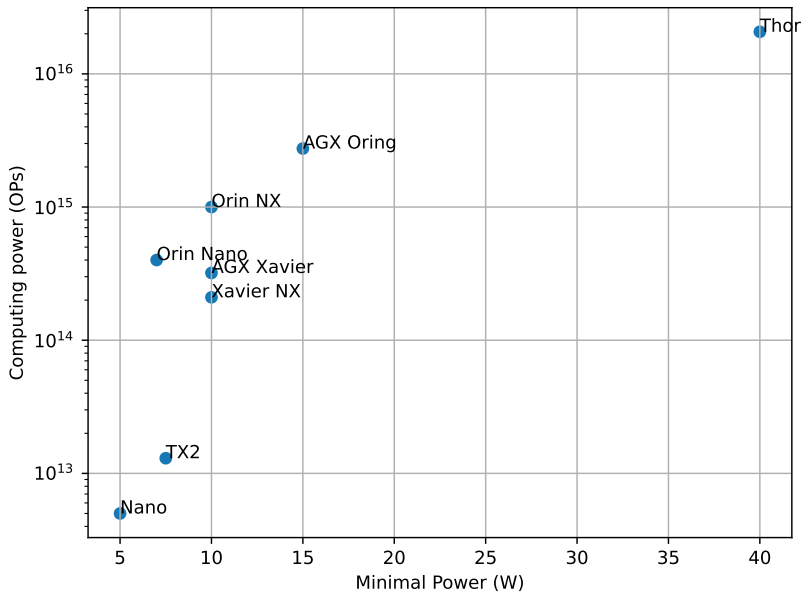

Figure 1.1 – Relation between power consumption (W) and calculation capacity (OPS) for Jetson products, the AI product range for embedded applications from Nvidia [178]. The power consumption increases with the calculation capacity from 5 W for the less power-intensive and computing-performance chip to 40 W for the biggest device.

To address the power constraints of ANN execution, researchers have explored multiple solutions. One approach involves offloading data processing to the cloud [31, 170], which eliminates embedded processing needs but introduces critical concerns regarding data privacy and computational latency. Alternative hardware solutions, more suitable for ANN inference, have emerged, such as Application-Specific Integrated Circuits (ASICs), Field Programmable Gate Arrays (FPGAs) [170, 169]. Complementing these hardware solutions, algorithm-level optimization, such as pruning or quantization, has been proposed to reduce the computational complexity of NN [43].
Another promising approach involves shifting the computing paradigm toward the bio-inspired neuromorphic computing paradigm that is less power-consuming. This novel computing paradigm, pioneered by Mead in 1988, draws inspiration from the human brain, which performs complex neuron computations with remarkable energy efficiency. Unlike traditional ANNs, neuromorphic computing adopts sparse event-driven and distributed processing in order to drastically reduce consumption [126, 120].

1.2 Spiking Neural Network as a solution for low-power in-edge AI

Spiking Neural Networks (SNNs) have emerged as a promising candidate for in-edge embedded AIoT applications. Neuromorphic computing, and especially Spiking Neural Networks (SNNs), are considered the third generation of neural networks [120] and

fundamentally differ from conventional ANNs by providing a more biologically plausible representation of brain structure and function.
In SNNs, information is encoded as temporal sequences of discrete events called spikes. A spike is a timestamped signal that can be mathematically represented as a Dirac pulse or implemented as a single-bit pulse signal in digital architectures. These spikes are processed by a specific neuron model that accumulates incoming spikes from synapses in their membrane potential voltage. When this potential exceeds a threshold voltage, the neuron emits an output spike that is propagated to other neurons via the axon. If no incoming spikes are detected, the neuron's membrane potential is not updated and slowly decreases to the rest potential.
The energy efficiency of SNNs stems from two characteristics. First, SNNs were initially designed as analog systems, where neurons and synapses are implemented using analog electronics, resulting in a lower energy consumption than digital architecture. Seconds, and more significantly, their information representation contributes substantially to energy saving. The sparsity of spikes in SNNs drastically reduces memory access operations, which are necessary for synaptic weight access, and neuron state updates frequency. This contrasts with conventional ANNs, which perform memory read and neuron operations regardless of input activity [114]. Consequently, even in digital implementations, SNNs demonstrate superior power efficiency compared to their equivalent ANNs [105].
As we previously mentioned, SNNs were originally developed as an analog technology [126]. However, purely analog implementations present significant challenges, particularly on synapses, memory storage, and scalability for large-scale networks. Consequently, researchers focus their work on hybrid digital/analog architectures or fully digital architectures, leveraging custom ASICs or reconfigurable FPGAs to overcome these limitations while maintaining the energy efficiency of neuromorphic computing. FPGAs appear as a good entry point for hardware neuromorphic architecture, offering distinct advantages over ASIC neuromorphic chips such as SpiNNaker. Their cost-effectiveness and development flexibility make them particularly attractive for research and prototyping. The inherent reconfigurability of FPGA architectures enables simulation acceleration, allowing researchers to implement various network topologies. While not achieving ultra-low power consumption compared to other neuromorphic chips, their robustness against noise and power fluctuations [111, 105] is a strong positive aspect for embedded applications such as autonomous vehicles [202] or satellite observations [4].

1.3 Motivations

To fully exploit the potential of neuromorphic computing and SNNs for low-power and embedded applications, we must transition from software-based SNN simulators and Von Neumann-based architectures such as Central Processing Units (CPUs)

and GPUs toward dedicated neuromorphic hardware. While commercial solutions like SpiNNaker [143, 78], BrainScaleS [161, 146] or Xylo chips [21] enable efficient low-power SNN inference [162, 198], these solutions can be expensive and may have limitations in terms of implementation possibilities, such as network topology, spiking neuron models, or online learning capabilities.

FPGAs emerge as an optimal compromise between cost, flexibility, and power efficiency. Their reconfigurable nature makes them ideal for both neuromorphic research and application-specific design. However, most, if not all, proposed emulators remain closed-source, forcing researchers to develop their own tools. This situation has led to a proliferation of tools and a lack of reproducibility in experiments.

Aiming to develop FPGA-based research within our research team and noting the absence, at least in 2022 at the beginning of this thesis, of an open-source FPGA emulator. We were motivated to develop a new neuromorphic FPGA emulator and offer it as an open-source tool to the community.

To address these challenges, we developed ModNEF, a new neuromorphic FPGA emulator designed as an open-source tool for the research community. Our solution serves as an accessible entry point to neuromorphic hardware research, especially in FPGA-based architectures. ModNEF provides a versatile hardware platform capable of emulating various networks with different topologies or neuron models. The emulator features a dual design philosophy: it operates as a user-friendly "black box" solution accessible to researchers without any background in FPGA or hardware design, while simultaneously offering a flexible foundation for future FPGA neuromorphic development, thus supporting the integration of new mechanisms.

The design requirements for our emulator are structured around these three principles:

— **Flexibility**: The emulator must be capable of emulating a wide variety of networks with various network topologies, particularly focusing on feed forward and recurrent topologies, with multiple spiking neuron models, including heterogeneous neural networks. Beyond network flexibility, it should provide fine-gained control over hardware-specific metrics, enabling users to optimize hardware resources, memory usage, power consumption, and latency of the hardware implementation.
— **Accessibility**: The emulator must be intuitive enough to be used by researchers without FPGA and hardware design expertise. Simultaneously, the source code must be comprehensible to SNN hardware developers, serving as an entry point for new research that leads to the next requirement.
— **Extensibility**: The emulator should serve as a foundation for future research and development. It is crucial for our tools to support the development and integration of new neuron models, emulation algorithms, convolutional kernels, or learning methods.

1.4 Outline

This thesis is organized as follows :
Chapter 2 provides a comprehensive survey of neuromorphic computing hardware research with a particular focus on FPGA-based implementations. This chapter aims to identify the key techniques developed in the fields and identify the limitations of the works on FPGA neuromorphic architecture.
Chapter 3 presents ModNEF, our novel neuromorphic architecture. We present in detail the FPGA architecture and the software framework developed, which provide a complete pipeline from model training to FPGA integration.
Chapter 4 evaluates ModNEF's performance on standard benchmark datasets. We present the observed limitations of our architecture and the solutions proposed by ModNEF to address these limitations.
Chapter 5 focuses on quantization techniques for ModNEF architectures. We conduct a depth study on quantization impacts and propose methods to mitigate the negative impact of quantization on accuracy for low-power deployment.
Finally, Chapter 6 demonstrates a real-world application of ModNEF for embedded application on biodiversity monitoring in the Mediterranean Sea through the ULP Cochlea ANR Project.

2 State of the art

In this chapter, we provide an overview of the state of the art regarding Spiking Neural Networks (SNNs), with a particular focus on hardware implementation, especially on FPGA design, and tools.
First, in Section 2.1, we introduce a general background of SNN by presenting the different network topologies, spiking neuron models, and learning methods used in SNN. Next, in Section 2.2, we will introduce hardware implementation of SNN with different implementation technologies such as analog, mixed analog-digital, and fully digital. Following this, in Section 2.3, we focus on FPGA architecture designs and problematics.

2.1 Spiking Neural Network: a State of the Art

In this section, we provide an overview of SNN implementations. We first briefly introduce the main mechanisms of SNN in Section 2.1.1. Then, in Section 2.1.2, we present the major mathematical models of spiking neurons. In Section 2.1.3, we discuss the different spike encoding formats and their impact on SNN performances. Next, in Section 2.1.4, we present the different network topologies. Finally, to conclude this section, we explore various learning methods used in neuromorphic computing in Section 2.1.5.

2.1.1 Introduction to Spiking Neural Network

Spiking Neural Networks (SNNs) are considered the third generation of Artificial Neural Networks (ANNs) [189].
The first generation of ANN was proposed in 1943 by McCulloch and Pitts [125], where neuron activation functions were the Heaviside function and synaptic weights were represented in a binary format.
The second generation of ANN, which remains the most widely used today, is based on continuous activation functions and real weights.
Further historical breakthroughs can be cited. Firstly, the backpropagation algorithm introduced by Rumelhart et al. in 1986 [158] revolutionized the training of multi-layer neural networks. In 1998, LeCun et al. [98] demonstrated the effectiveness of Convolutional Neural Networks (CNNs) for handwritten digit classification, paving the way for modern computer vision applications. Finally, the work of Hinton et al. [76] introduced a computational method for training deep neural networks, marking the beginning of the deep learning era.

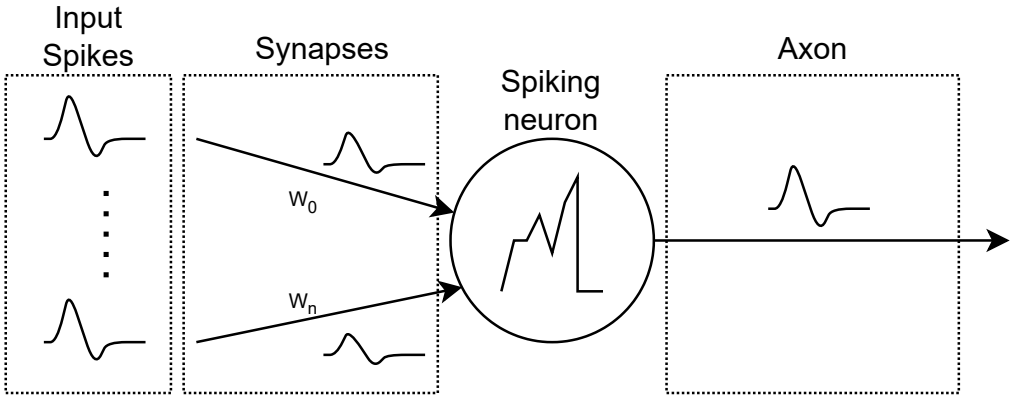

Figure 2.1 – Spiking neuron diagram. Input spikes are received from synapses, which modify the amplitude of the spikes. The weighted spikes are accumulated into the membrane potential. When the membrane voltage exceeds a threshold value, the membrane is reset and an output spike is emitted through the Axon.

Both previous generations draw inspiration from the brain, modeling computation as networks of interconnected neuron-like units. In 1988, Mead introduced the third generation by proposing neuromorphic systems that mimic not only the structure but also the data representation and dynamics of biological neurons [126, 121].
Contrary to the two previous generations of ANN, neuromorphic computing represents data as analog spikes. As shown in Figure 2.1, input spikes are received from synapses. These spikes are modulated by real synaptic weights and then accumulate into the membrane voltage. If the potential exceeds a threshold value, an output spike is transmitted to other neurons via the axon. In computer science, spikes can be considered simple Dirac pulses, which can be encoded using a 1-bit variable.
The particular structure and data representation of SNN lead to a reduction of power consumption. Unlike traditional ANN, the internal neuron state and the synaptic weight read operations are performed only when incoming spikes are detected. Thanks to the sparsity of spikes within SNN, the memory read and update operations frequency is reduced, leading, theoretically, to a lower power consumption [108, 105]. However, this reduction in power consumption must be nuanced by several important design choices [23, 105, 48, 14], which we will highlight in the next section, starting with the neuron model used in our SNN.

2.1.2 Spiking Neuron Mathematical Models

In this subsection, we will present different spiking neuron models used in neuromorphic computing.
The first formal model of neuronal electrical behavior, illustrated in Figure 2.2, was introduced by Louis Lapicque in 1907 [95, 1]. He proposed that neurons accumulate

input current in membrane potential over time. If the membrane voltage reaches a threshold value, the membrane potential is reset. This early formulation, known as the integrate-and-fire model, laid the foundation for many subsequent spiking neuron models.

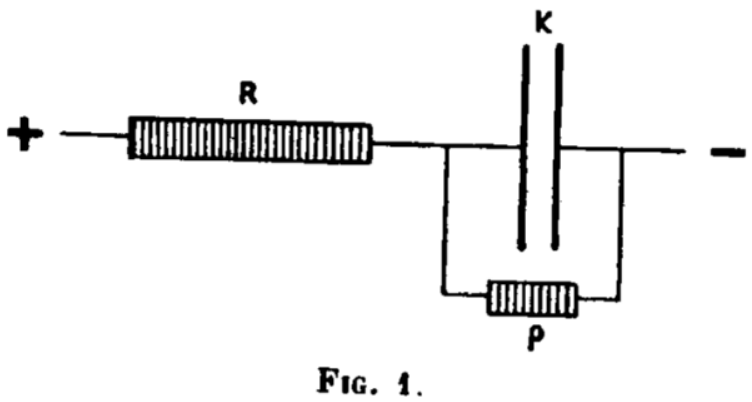

Figure 2.2 – Electronic representation of a neuron, as proposed by Louis Lapicque in *"Recherches quantitatives sur l'excitation électrique des nerfs traitée comme une polarisation"* [95].

In 1952, Hodgkin and Huxley [77] described the most biologically accurate neuron model. It consists of 4 differential equations, with 9 additional differential equations to describe the parameters of the first 4 equations. The mathematical complexity of the Hodgkin-Huxley (HH) model poses several challenges for hardware implementation or large-scale SNN simulation.
To address this issue, other neuron models have been developed to simplify the mathematical model while maintaining a high degree of biological realism, such as the Morris-Lecar model [97] or the FitzHugh-Nagumo model [85]. Izhikevich proposes a biologically realistic neuron model with only two differential equations, parameterized by 4 constants that determine the firing mode of the neuron model [83]. Equation 2.1 describes the membrane potential dynamic, while Equation 2.2 represents the membrane recovery. When the membrane voltage exceeds the threshold value, Equation 2.3 is applied to $V(t)$ and $u(t)$.

$$\frac{dV(t)}{dt} = 0.04V(t)^2 + 5V(t) + 140 - u(t) + I(t) \tag{2.1}$$

$$\frac{du(t)}{dt} = a(bV(t) - u(t)) \tag{2.2}$$

$$\text{if} V(t) \geq 30 \text{ then } \begin{cases} V(t) = c \\ u(t) = u(t) + d \end{cases} \tag{2.3}$$

In contrast to biologically realistic models such as the HH or Izhikevich models, simpler models like the Integrate and Fire (IF) neuron model [26] have been developed. The IF neuron model, described by Equation 2.4, consists of the accumulation of input current into the membrane potential.

$$C\frac{dV(t)}{dt} = I(t) \tag{2.4}$$

A major limitation of this model is that it does not consider the membrane potential decay when no incoming spikes are detected, which can be useful for processing temporal data and taking into account spike temporal information. The Leaky Integrate and Fire (LIF) neuron model addresses this limitation by incorporating the membrane decay mechanism while maintaining a simple mathematical model. A LIF model is described by Equation 2.5, where τ represents the membrane potential decay constant.

$$\tau \cdot \frac{dV(t)}{dt} = -V(t) + RI(t) \tag{2.5}$$

In most cases, the input current $I(t)$ is calculated as the weighted sum of incoming spikes, as shown in Equation 2.6. Here, w_i represents the synaptic weight between the post-synaptic and pre-synaptic neurons, while $x_i(t)$ is the input spike. In computer science, analog spikes can be represented as a binary variable that takes the value of 1 if the pre-synaptic neuron emits a spike and 0 otherwise.

$$I(t) = \sum_i w_i \cdot x_i(t) \tag{2.6}$$

Figure 2.3 represents the membrane dynamic response to the same input stimuli through different neuron models. The IF model exhibits the simplest membrane dynamics, while the LIF neuron model shows a more complex dynamic due to exponential membrane decay. The Izhikevich model, however, displays a more complex dynamic with an exponential increase and recovery mechanism after each spike emission.
Most spiking neuron models consist of differential equations. To run an SNN, it is necessary to solve these equations. One approach to solving differential equations is to discretize time and apply iterative methods such as the Euler or Runge-Kutta 3 method [61, 122, 90]. This approach necessitates updating the neuron state regularly and is known as **clock-driven**. However, this approach reduces the evenemential benefits of SNNs. To simulate or emulate SNNs in an **event-driven** manner, it is necessary to solve the differential equations before simulation. The LIF neuron model can be easily solved, but more complex neuron models such as the Izhikevich neuron model are too complex to be solved analytically [90].

2.1.3 Spike Encoding

Spike encoding schemes are a crucial aspect in the neuromorphic community [135, 163, 24] and have a significant impact on model performance.

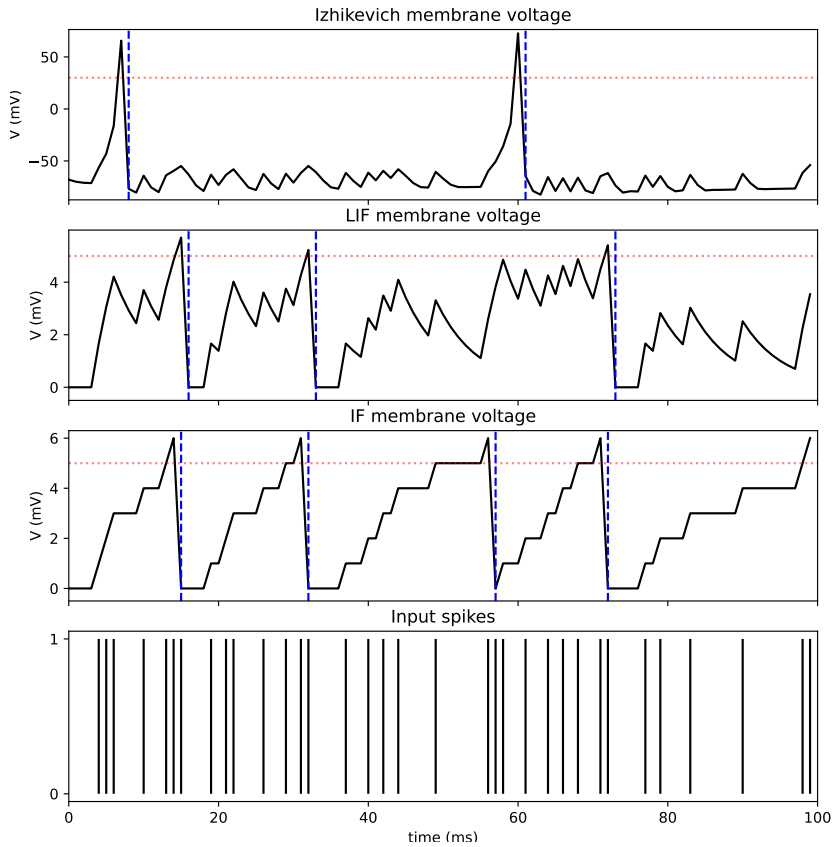

Figure 2.3 – Comparison of membrane voltage dynamics between Izhikevich, LIF and IF neuron models. The dotted red lines indicate the threshold value, and the dashed vertical blue lines mark the spike transmission events. From bottom to top, the first graph shows the incoming spikes received by the neurons during the simulation. The second graph illustrates the membrane dynamic of the IF neuron model; the third graph represents the membrane potential evolution of the LIF model; and the top graph depicts the membrane voltage evolution of the Izhikevich model.

The issue of spike encoding arises when input data comes from non-neuromorphic sensors, as is the case with datasets like MNIST [99] or CIFAR-10 [92]. Because spiking neurons cannot directly understand data provided by sensors, it is necessary to convert data into spike representation.

A common way to convert input data into spikes is to use a Poisson generator [73]. However, other conversion methods can be used, such as rate coding [24] or temporal coding systems like Time To First Spike (TTFS) encoding [12, 163, 60].

Fan et al. [47] have highlighted the importance of spike encoding on both accuracy and power consumption. Similarly, Barchid et al. [14] have shown the impact of encoding schemes on power consumption, depending on the scheme used. Guo et al. [60] have demonstrated that TTFS is a highly efficient encoding scheme for achieving high accuracy and low-power consumption in SNN inference.

The spike encoding method problematic is no longer arise when data are directly pro-

vided as spikes from neuromorphic or event-based sensors, such as DVS cameras [157, 109, 151], neuromorphic cochlea [96, 171, 87], or other event-based sensory systems [18, 69].
In this section, we highlight the importance of data encoding methods for SNN and neuromorphic hardware performance. In this thesis, we focus our experiments with a natural neuromorphic dataset. The question of encoding systems was therefore not studied.

2.1.4 Network Topology

In this section, we provide an overview of different network topologies.
The simplest network topology is the Feed Forward Fully Connected (FF-FC) topology [44]. FF-FC networks are composed of one or more layers, where each layer is connected to the next layer. In a FF-FC network, a layer cannot be connected to a layer placed before itself. Data is thus processed in a forward manner. Figure 2.4 represents a 2-layer FF-FC network where all layers are connected with dense connections, without feedback connections.

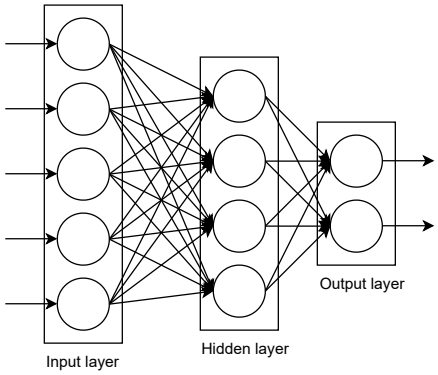

Figure 2.4 – FF-FC network topology with a hidden layer and an output layer, each connected with dense connections. The first layer input consists of a single input and single output neuron, which copies input spikes to the next layer. Unlike the input layer, the hidden layer comprises multiple inputs and multiple outputs, where each neuron receives inputs from all neurons from the previous layers and is connected to all neurons of the next layer. The output layer consists of multiple inputs and single output.

In contrast to the FF-FC topology, the Recurrent Neural Network (RNN) topology allows for feedback connections within the network. We can distinguish two types of RNN topologies based on the level of recurrent connections. The first is local recurrency, where recurrent connections are confined to a single layer, meaning neurons within a layer are connected to other neurons within the same layer. The second is global recurrency, which creates feedback connections across layers [214]. Figure 2.5

illustrates a recurrent network topology, with local recurrence in blue within the hidden layer and global recurrence in red from the output layer to the hidden layer. We differentiate these two network topologies because their hardware implementations differ, making it necessary to distinguish between them.

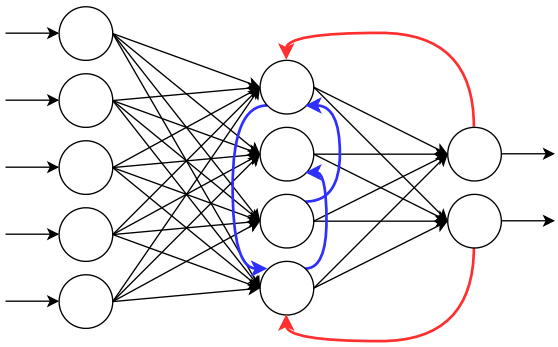

Figure 2.5 – RNN network topology. Black arrows represent dense forward connections, similar to the FF-FC topology. However, additional connections are implemented: blue arrows indicate local recurrent connections within the same layer, while red arrows represent global connections, connecting neurons from a layer to neurons in the previous layer.

Another well-used network topology is the Convolutional Neural Network (CNN), illustrated in Figure 2.6. This topology has demonstrated its ability to process images [156, 173, 150]. In a CNN, data is first processed by convolutional kernels, which are applied to the input data to extract information such as edges or textures. The result of the convolution is then passed to a pooling layer, which extracts essential information and reduces the size of the data matrix, thereby condensing the information. These two specific layers are iterated several times, and the output of the convolution is passed to a FF-FC network, which performs the classification task.

In SNN, the convolution process is adapted to spike processing. Abderrahmane et al. [4] apply convolution to non-spiking data and then transform the convolution map to spikes before the classification part of the network. Xing et al. [196] propose a cyclic recurrent CNN where convolutional and pooling layers are connected in a cyclic manner to store an internal state for the classification task.

Finally, another network topology is reservoir computing [164, 205], illustrated in Figure 2.7. In reservoir computing, input spikes are sent into a reservoir where neurons are connected randomly with fixed synaptic weights. The output is read from several neurons in the reservoir, which are connected with trained synaptic weights. Because only the weights of the output neurons are trained, the model training duration is reduced.

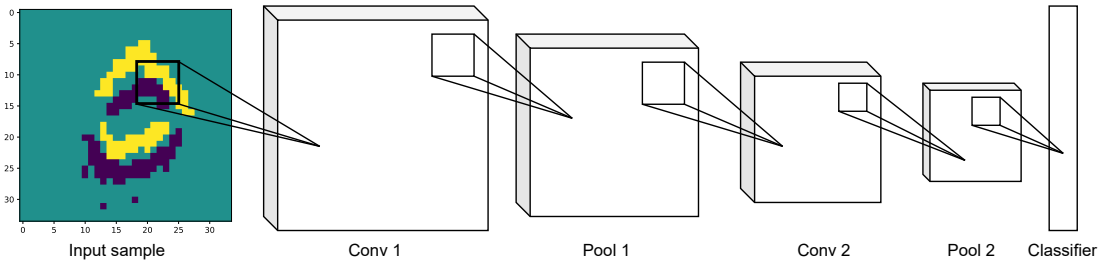

Figure 2.6 – CNN network topology illustration. The network consists of a succession of convolution layers and pooling layers, two stages in this example, followed by an output layer with dense connections. Each input neuron of the convolutional kernel processes a region of the previous layers or input data and applies a convolution operation to generate an output smaller feature map.

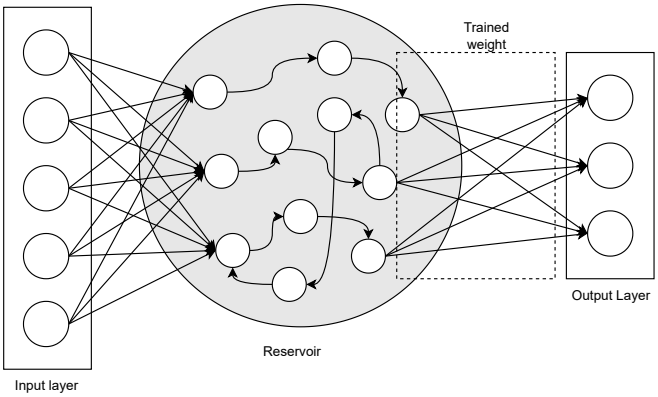

Figure 2.7 – Reservoir computing network topology. The input neuron transmits input data to the reservoir, which consists of multiple neurons randomly connected. The output layer is connected randomly to other neurons within the reservoir. Unlike on FF-FC or RNN topologies, only the output synaptic weights are trained, while the synaptic weights inside the reservoir remain random.

2.1.5 Learning Methods

In this section, we will introduce several important points about training methods applied to SNNs. Due to the particular data representation and specific neuron models in SNNs, the learning methods must be adapted. We can identify three main approaches to training SNNs. Firstly, in continuity with biological inspiration, we can use biological learning methods based on Hebb's rule. Secondly, we can adapt learning methods from the second generation of ANNs to SNNs, such as gradient-based rules. Finally, we can first train an ANN and then convert the trained model into a SNN [39, 100].

In this section, we will first focus on bio-inspired learning methods in Section 2.1.5, as

these methods are widely used in hardware implementations of SNNs. Then, we will present Backpropagation (BP)-based learning methods in Section 2.1.5.

Bio-Inspired Learning Methods

Beyond neuron and network models, the bio-inspired approach of neuromorphic computing also motivates the use of learning rules derived from biological mechanisms, such as Spike-Timing-Dependent Plasticity (STDP) or delay-based plasticity.

The Hebbian learning rule was introduced by Hebb in 1949 [72] and can be summarized by the idea: *"Neurons that fire together, wire together"*. However, Hebb did not provide a mathematical equation for the weight update function. Based on Hebb's rule, Bi et al. [16] introduced the Spike-Timing-Dependent Plasticity (STDP) learning rule. The learning rule is described by Equation 2.7 and illustrated in Figure 2.8. The parameter Δt represents the time difference between the post-synaptic (output) spike t_{post} and the pre-synaptic (input) spike t_{pre}. If Δt is positive and small, it means the input spike has a significant impact on the neuron's spike emission, and the connection between the two neurons increases. Conversely, if Δt is negative, it means the incoming spike has no impact on the neuron's activity, and the weight between these two neurons decreases. STDP is an attractive learning method for hardware implementation due to its weight update operation local to neurons. However, STDP is generally less efficient than gradient-based learning methods, resulting in lower model accuracy [184].

$$\Delta w = \begin{cases} A_+ \cdot \exp \frac{-\Delta t}{\tau_+} & \text{if } \Delta t \geq 0 \\ -A_- \cdot \exp \frac{\Delta t}{\tau_-} & \text{if } \Delta t < 0 \end{cases} \tag{2.7}$$

Another bio-inspired learning method is the delay learning method [179]. Contrary to STDP, where we modify the synaptic weight, in delay learning, we modify the delay between the spike reception and the spike integration into the membrane potential [67]. Figure 2.9 illustrates the impact of delay learning on membrane potential. Before the delay learning, represented in left graphs, the two input spikes are too spaced to trigger an output spike. However, after the learning phase, as shown on the right graphs, the two input spikes are close enough to trigger an output spike without modifying the synaptic weight between the neurons. In this example, we did not modify the weight between neurons during learning, but it is possible to learn delay and synaptic weight [168].

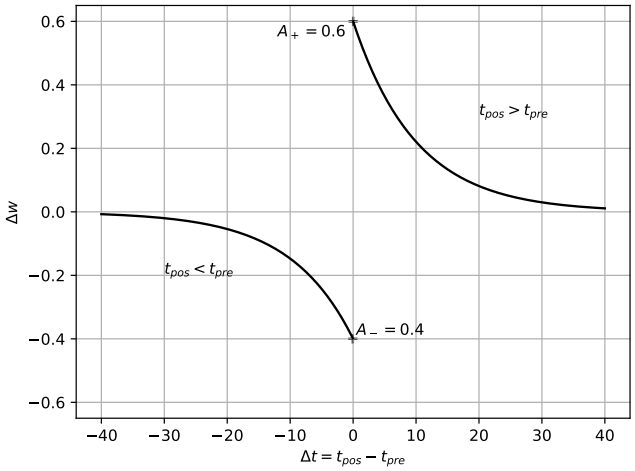

Figure 2.8 – Δw dynamic, depending on Δt for the STDP learning rule. The maximum value corresponds to $A_+ = 0.6$ and the minimum value corresponds to $A_- = 0.4$, with $\tau_+ = \tau_- = 10ms$. If the incoming spike is accumulated just before the output spike emission, the STDP interprets this spike as important and increases the corresponding weight, thereby giving more importance to this spike.

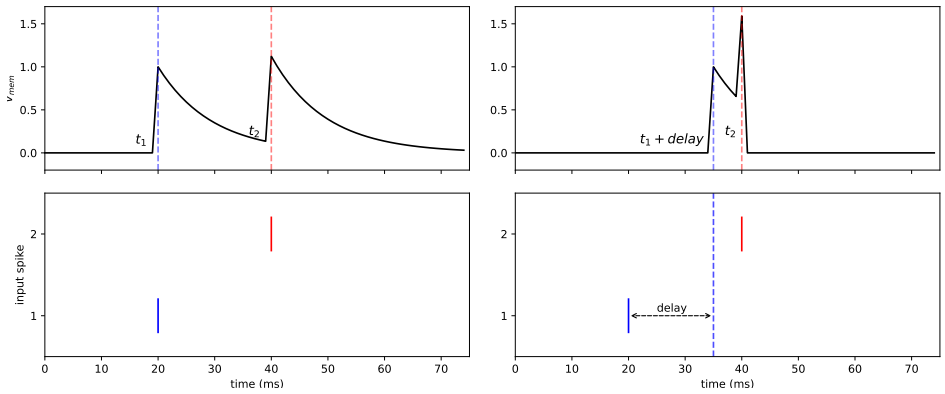

Figure 2.9 – Delay learning representation. Left graphs represent the membrane dynamic before learning, and right graphs show the membrane potential after delay learning. In the right graph, the blue spike is delayed by 15 ms, resulting in an output spike generation since the spikes are accumulated within a smaller time window.

Other bio-inspired learning methods, such as dendrite-based [204] or STDP variation [203] are proposed.

Backpropagation-Adapted Learning Rules

In contrast to bio-inspired methods, we can adapt learning methods used to train ANN for SNN training, and more specifically, the backpropagation method.
Backpropagation (BP) was introduced by Rumelhart et al. in 1986 [158] and has proven

to be a very efficient approach for neural network learning. However, this learning method requires differentiable neuron activation functions, which is not the case for spiking neuron activation functions. In fact, the membrane reset after spike emission can be considered as a Heaviside function $S(V(t)) = \Theta(V(t) - V_{th})$, which is not differentiable [101, 80, 134].
The major challenge for applying the BP method is thus to make the spiking neuron activation function differentiable. To overcome this problem, Neftci et al. [134] propose to smooth the Heaviside function used for spike generation with a surrogate gradient, making the membrane potential reset differentiable during the backward pass and thus compatible with BP learning. Many surrogate functions can be defined, provided they approximate the Heaviside function with a differentiable function. In Figure 2.10, we represent the surrogate gradient with two different surrogate functions: the sigmoid function in orange [210] and the arc tan function in green [49].

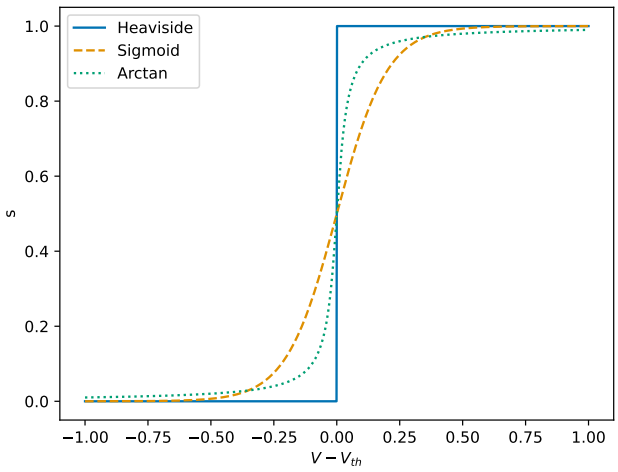

Figure 2.10 – Approximation of the Heaviside function using two surrogate gradient functions, based on the difference between the membrane potential and the threshold voltage. The original Heavisde function is presented in blue, the sigmoid approximation is represented by the orange dashed line, and the Arctan approximation is illustrated with the green dotted line.

Due to the importance of time in SNNs, it is also possible to use BP-based learning methods with additional temporal information. We can mention Backpropagation Through Time (BPTT), initially used to train RNN networks [39, 194]. Other methods have also been proposed, such as Spatio-Temporal Backpropagation (STBP) by Wu et al. [194].

2.1.6 Conclusion

In this section, we present a new generation of neural networks that draw inspiration from the human brain, not only in their network topology but also in the way data

is represented within the brain. In neuromorphic computing, data is represented as spikes, which are timestamped events (Section 2.1.1). Spikes can be produced directly from neuromorphic sensors such as Dynamic Vision Sensor (DVS) cameras or generated by conversion methods (Section 2.1.3). These spikes are processed by spiking neurons, a particular neuron model that accumulates input spikes within a membrane potential and emits an output spike when the potential exceeds a threshold voltage. A particular mechanism of spiking neurons is the membrane voltage decay mechanism, which occurs when no input spikes are received. Thanks to this particularity, SNNs appear as good candidates for temporal data processing (Section 2.1.2). As with other generations of ANNs, SNNs can be arranged with different network topologies, from simple topologies such as Feed Forward Fully Connected (FF-FC) to more complex ones such as Convolutional Neural Network (CNN) or reservoir computing (Section 2.1.4). Due to the particularities of SNNs, learning methods must be adapted, either by drawing inspiration from biology or by adapting already known methods such as BP learning methods (Section 2.1.5).

2.2 Neuromorphic Hardwares

The sparse activity within SNNs leads to the following conclusions: the neuron state is updated when new informations, i.e., spikes, are detected, and the synaptic weight read operation frequency is reduced. These two points present SNNs as an interesting neural network technology for low-power tasks. However, running SNNs on Central Processing Units (CPUs) or Graphic Processing Units (GPUs) does not consider these particularities, which is a major reason why researchers aim to create dedicated hardware for SNN emulation.
Due to their biological inspiration and the use of analog neuron models, analog hardware implementations appear as a natural approach for SNN hardware implementation. However, as we will see in Section 2.2.1, analog implementations raise several challenges. To address these limitations, researchers propose alternative strategies such as mixed analog-digital architectures, presented in Section 2.2.2, or fully digital architectures, discussed in Section 2.2.3.

2.2.1 Analog Hardware Architectures

In his paper, Mead [126] conceptualizes neuromorphic computing as an analog neural network technology. This is primarily because the human brain, like analog circuits, operates in an analog and asynchronous manner. Additionally, digital hardware systems, due to their design, consume more energy than dedicated analog hardware.
One approach to implementing analog SNNs is to use Field Programmable Analog Arrays (FPAAs), an analog equivalent of FPGAs, as the hardware target [133, 164, 166]. However, the majority of analog architectures consist of custom chips [164].

To implement SNNs on analog chips, it is necessary to implement both the neuron model and the synaptic weight memory. Implementing a spiking neuron model in analog technology appears to be feasible and efficient in terms of energy efficiency. Sourikopoulos et al. propose a spiking neuron model using 65 nm sub-threshold Complementary Metal Oxide Semiconductor (CMOS) technology that consumes only 4 fJ per neuron [175].
The major challenge of analog implementation is the synaptic weight implementation. A significant innovation for synaptic weights was the implementation of the memristor component, theorized in 1971 by Chua [36] and first implemented in 2008 by Strukov et al. [176]. The memristor can be described as a passive resistor where the resistive value can be controlled by an input voltage. Memristors appear to be good candidates for analog synaptic weights with online learning capabilities [32, 62]. Additionally, Resistive Random Access Memory (RRAM) also appears to be a promising approach for analog synaptic weight memory [186].
Another major challenge of analog hardware architecture is the spike routing, and internal communication can become a significant limitation due to the high number of neurons and spikes [164].
Other researchers propose to implement neurons analogically and synaptic weights with digital memory, leading to a new type of architecture that combines both digital and analog technologies. We will present this approach in the next section.

2.2.2 Mixed Analog and Digital Architectures

In the previous section, we highlighted two major challenges of implementing SNNs on fully analog hardware: synaptic weight memory and spike routing. Several researchers propose to maintain analog neuron models while implementing synaptic weights or spike routing with digital technology.
Neurogrid [15] proposes an analog representation of neurons and synapses. Synapses are localized near the neurons and consist of analog conductances that emit input currents when an input spike is received. The digital part of Neurogrid consists of the spike routing scheme, which is based on a tree topology [127].
BrainScaleS [161, 146] also proposes to implement analog neuron models with digital synaptic memory. In BrainScale2 [146], analog neuron models can be parameterized by digital parameters stored in an 80-bit Static Random Access Memory (SRAM) and converted using a 10-bit resolution Digital to Analog Converter (DAC). The synaptic weights are locally stored in a 6-bit SRAM. In addition to neuron parameters, and contrary to Neurogrid, the synaptic weights are stored in a 6-bit SRAM memory. Thanks to the network topology, these weight memories can be placed locally on neurons.
Similarly, DYNAPs [131, 154] propose to implement neuron models with analog technology and synaptic weights with digital SRAM. The major contribution of DYNAPs is the spike routing scheme based on a mixed topology, featuring a 2D-mesh topology

at the local level and a tree-based routing strategy at a higher level [130].
These architectures can achieve low-power consumption thanks to their analog computational components and a high level of flexibility due to the digital synaptic weight memory or digital event routing schemes, making these boards highly adaptable.

2.2.3 Digital Architectures

In contrast to analog architectures, researchers have also focused their studies on fully digital architectures for SNN emulation. As discussed in the previous section, digital implementations can address some challenges of fully analog implementations, such as synaptic weight storage and spike routing. Fully digital architectures appear to be good candidates for low-power and large-scale SNN deployment, thanks to the maturity and robustness of CMOS technology.
A well-known digital neuromorphic architecture is SpiNNaker [143, 27, 124]. SpiNNaker is based on the interconnection of ARM Cortex M4F processors. In SpiNNaker 2 [124, 58], each ARM processor can emulate up to a thousand neurons with a million synaptic connections [78], depending on the neuron model emulated by the neurocore. The major contribution of SpiNNaker is the spike routing scheme and the interconnection of ARM processors. Each neurocore, i.e., ARM processor, implements an internal router to route generated events. Routers are connected with a toric topology, demonstrated as the best topology for event routing [145, 193, 58].
Another notable digital architecture is TrueNorth developed by IBM [6]. Similar to SpiNNaker, TrueNorth is composed of multiple neurosynaptic cores, each emulating 256 neurons and 64,000 synapses. TrueNorth chip implements 4,096 neurosynaptic cores interconnected in a 2D-mesh topology with routers. An interesting design choice of TrueNorth is the use of a Globally Asynchronous Locally Synchronous (GALS) approach. Neurosynaptic core communication and circuit control operate asynchronously in contrast to neuron emulation, which runs synchronously. This results in high-speed communication and minimizes the number of clock transitions, thereby reducing dynamic power consumption. The power consumption depends on the number of active synapses and spikes, but the authors consider 65 mW as typical power consumption.
We can also mention Loihi, developed by Intel [41, 42, 141]. Like the two previous architectures, Loihi implements several neurocores to emulate a collection of neurons, up to a maximum of 8,192 neurons per core in Loihi 2. Unlike other chips, Loihi implements an undefined number of synapses by using an internal synaptic memory, and the maximum number of synapses will depend on the synaptic format. An interesting feature of Loihi is the online learning engine provided by the neurocore, enabling the chip to run STDP learning rules on the chip.
Other ASIC neuromorphic systems are proposed for general purposes [118] or dedicated tasks such as embedded IoT systems [81], vision processing [155], or audio pro-

cessing [21]. We can also mention the Akida AKD1000 chip, developed by BrainChip [5, 136]. The chip can emulate a wide variety of neural network topologies, such as FF-FC, recurrent and CNN using the LIF neuron model. Additionally, the Akida chip supports online learning and is compatible with the TensorFlow framework [123] and CNN2SNN toolkit developed by BrainChip.

In this section, we focused our presentation on ASIC digital architectures. However, ASIC construction can be costly and lacks of reconfigurability. FPGAs appear to be a suitable alternative for cost-effective and highly flexible implementations. As this is the main subject of the thesis, we will dedicate the next section to focusing on FPGA implementation methods of SNNs.

2.3 SNN Implementation on FPGA

As discussed in the previous section, neuromorphic computing was initially conceived as a fully analog approach. However, fully analog hardware architectures present several significant challenges. While mixed analog-digital and fully digital ASICs can effectively address these challenges, these solutions come with their limitations. Specifically, ASIC development is associated with high costs, extended development timelines, and inherent architectural inflexibility once manufactured.

One initial approach to implementing SNNs flexiblely involves using CPUs and GPUs. While these offer a high level of flexibility, they are typically too power-intensive for low-power embedded applications. Field Programmable Gate Arrays (FPGAs) emerge as promising hardware targets for low-power and flexible implementations. Despite the relatively high-power consumption of FPGAs due to memory access operations [105], they have been successfully deployed in embedded tasks such as robotics [23] and in-orbit satellite systems [4].

Even for non-embedded applications, FPGAs can serve as effective hardware accelerators, providing a cost-efficient entry point for neuromorphic hardware acceleration. They offer high computational performance and reconfigurability, enabling users to implement various network topologies [90, 50, 82, 149, 111].

Implementing SNNs on FPGAs presents unique challenges and issues due to hardware constraints and platform specificities. This section focuses on the major challenges of SNN implementation on FPGA hardware targets. In Section 2.3.1, we will introduce the subject by discussing emulation strategies, an important aspect for understanding the implementation of spiking neuron models detailed in Section 2.3.2. Then, in Section 2.3.3, we will address the challenges introduced by synapses in terms of memory consumption and the implementation of continuous synaptic models. Subsequently, we will see what network topologies are implemented on FPGA. Before focusing on interesting FPGA neuromorphic emulators in Section 2.3.5, we will present implementations of learning methods on FPGAs in Section 2.3.4.

2.3.1 Emulation Paradigm: Clock-Driven VS Event-Driven

As we previously explained in Section 2.1.2, most spiking neuron models are described by one or more differential equations. To emulate neurons, it is necessary to solve these equations. The method of how we solve the equation can result in an opposite emulation paradigm. In this section, we will so present the two major emulation strategies, **Clock-Driven** and **Event-Driven**, which are essential to understanding the neuron implementation presented in the next section.
The **Clock-Driven** emulation paradigm consists of updating neuron state regularly, even in the absence of new spikes. A major drawback of the emulation paradigm is to lose the benefits of event sparsity, which reduce the neuron update and the memory access operations, power-hungry operations [105]. However, this emulation paradigm greatly facilitates the resolutions of differential equations by using iterative solving methods such as Euler or Runge-Kutta [61, 122, 90], as it shows in Equation 2.8.

$$V[t+1] = V[t] + \frac{dv}{dt} \tag{2.8}$$

In opposition to Clock-Driven emulation paradigm, we can keep the event-based approach of neuromorphic computing and use **Event-driven** emulation paradigm. This emulation paradigm keeps the benefits of spike sparsity, but it is necessary to solve differential equation(s) over time. For simple neuron models such as LIF, equation resolution can be easily done but result in a more complex mathematical operation. Li et al. [108] illustrate this drawback with the following example: the solved Equation 2.9 involves addition, multiplication, and exponential function. With clock-driven approach, $\Delta t = 1$ and $\exp(-\frac{1}{\tau}) = \alpha$ and can be simplified by a combination of addition and multiplication, as illustrated by Equation 2.10. Other neuron models cannot be solved, such as the Izhikevich model, which is too complex to be solved.

$$v = v_{rest} + (v - v_{rest}) \exp(-\frac{\Delta t}{\tau}) \tag{2.9}$$

$$v = v_{rest} + (v - v_{rest})\alpha \tag{2.10}$$

In their work, Li et al. [108] summarize the two emulation strategies as follows: *"The clock-driven computing results in more frequent computation, but its computational complexity at each time is less than that of the event-driven computing technique."* To leverage the advantages of both approaches while mitigating their drawbacks, they propose an adaptive emulation paradigm that dynamically switches between clock-driven and event-driven modes based on the spiking activity inside the network.
It is important to differentiate Clock-Driven and Event-Driven, which define how and when neuron states are updated independently to hardware architecture, and synchronous and asynchronous, which are specific to hardware architecture using a clock signal or not.

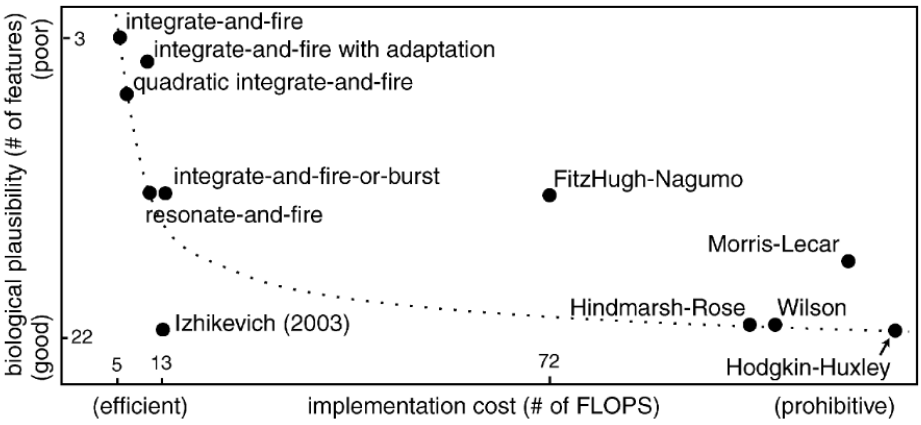

Figure 2.11 – Izhikevich neuron classification depending on biological plausibility and computational complexity from [84]. The abscise axis represents the computational complexity of the neuron model, measured by the number of FLOPS, while the ordinate axis indicates the level of biological inspiration. The HH model is positioned at the bottom right due to its high biological plausibility, placing it at the bottom, but its high mathematical complexity places the model to the right. The LIF models are less computationally complex, positioning them to the left, but their lower biological plausibility places them higher on the ordinate. The Izhikevich neuron model appears as the best trade-off between biological inspiration and computational cost.

2.3.2 Spiking Neuron Models

In the previous section, we focused on emulation strategy and how it can impact the manner of how we solve differential equations. In this section, we will focus on neuron models used in FPGA neuromorphic emulators.

The choice of neuron model has a significant impact on emulator performances. Since neurons are one of the most significant computational resources inside SNNs, neuron hardware implementation is crucial. Depending on mathematical complexity and emulation strategy, the hardware footprint of hardware neurons will differ and can restrict the number of neurons that can be implemented due to hardware limitations of FPGA. That can significantly impact power performances. Two solutions can be employed to leverage the hardware footprint of neuron models, firstly the choice of the neuron model and secondly the multiplexing of the neuron model.

In his paper, Izhikevich [84] has classified different neuron models depending on biological plausibility and computational cost, illustrated by Figure 2.11. As we can see, the computational cost, represented with FLOPS, of neuron models highly differs from different models. Even if the implementation cost is given for software cost, it gives a good estimation of the mathematical complexity and complexity of hardware implementation.

Several studies have implemented the HH neuron model on FPGA. However, due to mathematical complexity, it is necessary to reduce the mathematical complexity of

operations. Yaghini et al. [197] propose to use COordinate Rotation DIgital Computer (CORDIC) algorithm to implement exponential function. In [10], authors propose to linearize the exponential function to reduce the mathematical complexity of the neuron model. Another way to simplify the HH model is to approximate the exponential function with a set of base-2 functions [66]. Shama et al. [165] propose to combine the two previous methods to implement the HH neuron model.

The same previously described methods have been used to implement the Morris-Lecar neuron model [70, 102] or the FitzHugh-Nagumo model [209, 137].

According to Figure 2.11, the Izhikevich neuron model offers the best trade-off between computational complexity and biological plausibility. As for previous neuron models, the CORDIC algorithm [74, 187, 181] has been employed to implement multiplications and quadratic therm. It is also possible to fit Izhikevich parameters to simplify multiplications into shift registers, as demonstrated in [103]. Liu et al. [113] propose to implement a stochastic-based Izhikevich neuron model that achieves low hardware consumption with high computational precision. In [65] and [152], authors propose to replace the quadratic function by an optimizing $cosh$ function based on shift registers and addition. We can finally notice a Look-Up Table (LUT)-based implementation where $v[t]$ is used as the read address to access the next pre-calculated $v[t+1]$ [8].

However, as we can see in Figure 2.11, the most computationally efficient neuron model is the Leaky Integrate and Fire (LIF) neuron model. Even if it is not a bio-plausible neuron model, its computational complexity is enough to run SNN and achieve good accuracy. The LIF neuron model offers the best trade-off between computational efficiency and computational complexity. The major challenge for the LIF implementation is how the decay mechanism operates with multiplication, a power-hungry operation. A first solution is to fix the decay factor as a power of 2 and so replace multiplication with a shift register [29, 28, 51, 7]. Another solution proposed by Gupta et al. is to linearize the decay and so replace multiplication by subtraction [63, 129]. We can also cite the CORDIC algorithm used in [192] to implement the exponential function that occurs when the LIF neuron model is solved.

To minimize neuron model hardware impact, we can change the neuron model for a simpler model, adapt neuron parameters to simplify mathematical operations, or implement these operations with a hardware-efficient algorithm. Another solution to minimize the neuron's impact is to time-multiplex the hardware neuron [2]. SPLEAT emulator [4, 3] proposes to multiplex a single hardware neuron per Neural Processing Unit (NPU) and implement one NPU per layer. In another study, Liu et al. [114] implement 16 hardware neurons and multiplex these neurons to emulate all SNN's neurons.

In this section, we focused on the implementation of neuron models on FPGA. Because the neuron model is a predominant computational resource, the choice of neuron model can have a significant impact on emulator performances. The HH neuron model is highly bioplausible but also highly hardware expensive. Izhikevich reduced

computational complexity and kept high bio-inspiration degrees. If the bio-inspiration is not an important point, a simpler model such as IF or LIF can be used. In addition to the neuron model choice, some hardware optimization methods can be applied, such as shift registers for multiplications, CORDIC algorithms LUT with already computed values, or stochastic computing. Another way to minimize the hardware impact of the neuron model, several works proposed a time-multiplex hardware neuron model to emulate several neurons with a single hardware circuit.

2.3.3 Synapses Models

The previous section was focused on neuron models, one of the most present computational resources. The other major computational resources are synapses. In this section, we will focus on synapses implementations on FPGA neuromorphic emulators and the memory issue raised by synaptic weight storage.
Some works propose to implement a continuous model of synaptic weight [188, 206], but most FPGA works implement synaptic weight and input current as a weighted sum described by Equation 2.11. To reduce memory consumption, Tang et al. propose an implementation of weight-binarized SNN to reduce as much as possible the memory consumption [180].

$$I(t) = \sum_i w_i \cdot x_i(t) \tag{2.11}$$

Other researchers propose to implement synaptic delay to support SNN trained with delay-learning methods, previously described in Section 2.1.5. Zheng et al. [215] propose programmable delay parameters for synapses. Leone et al. propose an implementation of delay mechanisms at an axon level [106].
Synapses and more specifically synaptic weight implementation is a major challenge due to the high consumption of memory used to store synaptic weight. Lemaire et al. [105] study has shown the significant impact of memory access on energy consumption. Moreover, due to the limited amount of memory on FPGA boards, the resolution of synaptic weight can have a significant impact on network size.

2.3.4 Network Topologies

In the two previous sections, we identified two challenges for SNN implementation on FPGA. The first one is the choice neuron model. Depending on the neuron model and the emulation strategy of the emulator, Clock-Driven or Event-Driven, the hardware footprint of neurons is a first limiting factor for network implementation. For similar reasons, synapses also limit implementation possibilities due to the constrained memory resources available within FPGAs. In this section, we will focus on other challenges specific to network topology implemented on FPGAs.

FF-FC and RNN are relatively simple network topologies, and their implementations are well understood. Thanks to the reconfigurability of FPGAs, each layer can be implemented within a neuron emulation unit that emulates a group of neurons. These units can be connected with point-to-point connections and transfer spikes directly from a layer to the next [4, 28]. For recurrent networks, neuron emulation units support recurrence as an internal mechanism, and the overall architecture remains similar to that previously described [28].
Other works propose to implement Spiking Convolutional Neural Network (SCNN) network topologies, which are particularly interesting for image classification [90]. Most of these works focus on convolutional kernel implementation, and one particularly interesting method is kernel subdivision operations. Chen et al. [33] propose a subdivision of the kernel into channels to process convolution in a parallel way, and feature maps are then regrouped with a tree adder. In DeepFire2 [13], the authors propose a kernel filter with a configurable level of parallelism to fit the best trade-off between hardware footprint and execution latency.
Some works propose an interesting approach that leverages the advantages of both ANN and SNN, creating hybrid neural networks [104, 208]. Lemaire et al. [104] propose to use non-spiking CNN to extract feature maps of static input images with a low latency and then convert feature maps to spikes and take advantage of spike sparsity for low-power classification.
Regardless of the network topology, implementing a large-scale neural network on FPGAs is a significant challenge due to hardware resource limitations. Larger chips, such as Zynq Ultrascale [50], can be used to increase network size by increasing the chip capacity. BiCoSS implements large SNNs with multiple Cyclone V FPGA chips connected in a tree topology with efficient spike routing [201, 200]. Wang et al. propose to multiplex neuron circuit and propose a neuron organization into hyper-column to minimize hardware footprint with additional Double Data RAM (DDR) memory to emulate from 20 millions to 2.6 billion neurons [188].

2.3.5 Learning Methods Implementation

In the previous sections, we have focused on major challenges for SNNs FPGA implementations. In all cases, the emulation strategy, spiking neuron model, network topology, and synapses model must be carefully designed and chosen to run, at least, SNN inference. Another major challenge for SNN implementation is the learning algorithm. In FPGA emulators, most works prefer to use offline training on CPU or GPU and then program internal FPGA memory with trained weight. However, several studies have proposed to implement online algorithms. In this section, we will focus on challenges in implementing online learning algorithm implementation on FPGAs.
Due to high mathematical complexity and the needs of a global view of the networks, the Backpropagation (BP) learning algorithm is not well-used for online learning meth-

ods. Several works propose digital implementation of BP for neuromorphic emulators [164, 138]. If we consider non-spiking ANNs implementation, more articles can be found of FPGAs implementation of BP [93]. Using a co-processor to run the learning algorithm appears as a viable solution to minimize the hardware impact of the learning algorithm and simplify the implementation of complex mathematical operations [142]. Fully FPGA implementation appears limited with lower accuracy [89] or constraints on network size [185].
Hebbian-based learning methods are preferred by researchers due to their local implementation and simpler mathematical operations. The most well-known Hebbian learning method is STDP, previously detailed in Section 2.1.5 and described by Equation 2.12.

$$\Delta w = \begin{cases} A_+ \cdot \exp \frac{-\Delta t}{\tau_+} & \text{if } \Delta t \geq 0 \\ -A_- \cdot \exp \frac{\Delta t}{\tau_-} & \text{if } \Delta t < 0 \end{cases} \tag{2.12}$$

The major challenge in implementing STDP is exponential functions. Similar to neuron model implementation, several researchers have proposed to use CORDIC to implement STDP function [74, 167]. Another method involves using LUT with pre-calculated values to replace exponential operations by a read operation [115, 108, 172]. Other researchers propose approximating exponential functions with the Piecewise linear (PWL) method [10, 88, 57]. To simplify the computation of $\Delta t = t_{pos} - t_{pre}$, researchers propose to use shift registers to delay input spikes and calculate Δt for each synapse [115, 108, 30]. Other works propose variation of STDP with tiplet-based STDP [148] where an additional pre-synaptic or post-synaptic spike is considered to compute Δw [94, 203, 88, 71, 200].
Others learning methods are implemented on FPGAs, such as the dendrite-based learning method [201, 204], where the learning method depends on spike timing and on dendritic activity or delay learning methods [91].

2.3.6 Notable FPGA Emulators

In the previous sections, we presented several challenges in implementing SNNs on FPGAs. The choice of neurons and synapses appears crucial and can limit the network topology possibilities. While most researchers train their network with offline devices, efforts are being made to develop online learning methods. In this section, we will present two important emulators: SPLEAT in Section 2.3.6 and Spiker+ in Section 2.3.6.

SPLEAT

SPiking Low-power Event-based ArchiTecture (SPLEAT) [3, 4, 104] is a neuromorphic emulator developed by the EDGE research team of the LEAT laboratory.

SPLEAT is based on the interconnection of configurable Neural Processing Modules (NPMs), each emulating a layer of the network. NPMs can be configured to emulate different layer topologies, such as convolutional, pooling, and dense layers. SPLEAT proposes IF and LIF neuron models, but authors consider the IF model sufficient for static data processing.
To achieve low-power inference, SPLEAT adopts an event-driven emulation strategy, which is simplified by the use of the IF neuron model. The authors also propose a time-multiplexing neuron model to minimize hardware resource consumption and, therefore, power consumption. However, this approach increases the inference speed but remains fast enough for high-speed classification [2].
An interesting feature of SPLEAT is its ability to run hybrid neural networks with formal convolutional kernels to extract feature maps. These can be converted into spikes with a spike generation module that converts input data into spikes with rate or temporal coding.
SPLEAT has been utilized for in-orbit processing of satellite imagery for cloud segmentation using a hybrid neural network. The neural network was embedded in a Cyclone V FPGA on an OPSSAT satellite, demonstrating SPLEAT's capability to perform embedded classification tasks with high-speed and low-power consumption.

Spiker+

Spiker+ is an open-source FPGA emulator presented by Carpegna et al. [29, 28] in 2023, developed in parallel with our works.
Spiker+ proposes to emulate each layer of the network with different modules, one per layer. Each module is composed of internal synaptic weight memory, a layer control unit, and a neuron model implemented in parallel to optimize the inference latency. Spiker+ offers three neuron models with different degrees of complexity, which can be described by Equation 2.13. If $\beta = \alpha = 1$, the model reduces to an IF neuron model. If $\beta \neq 1$ and $\alpha = 1$, the neuron model corresponds to a first-order LIF. Finally, if $\beta \neq 1$ and $\alpha \neq 1$, we obtain a second-order neuron model. For each neuron model, Spiker+ proposes hard reset $V_m = 0$ and soft reset $V_m = V_m - V_{th}$. Spiker+ appears to be highly configurable in terms of the neuron model.

$$\begin{cases} I_{syn}[t] = \alpha \cdot I_{syn}[t-1] + W \cdot s_{in}[t] \\ V_m[t] = \beta \cdot V_m[t-1] + I_{syn}[t-1] \end{cases} \tag{2.13}$$

In addition to the FPGA emulator, Spiker+ offers a configuration framework based on SNNTorch to enable users to create, train, and implement SNNs with their emulator without requiring hardware design knowledge. The framework can be found on GitHub with many examples and a video tutorial: `https://github.com/smilies-polito/Spiker`.

2.4 Conclusion

In this chapter, we present a state-of-the-art review on hardware implementation of SNNs.

Neuromorphic computing is a novel computational approach inspired by the human brain and is considered the third generation of Artificial Intelligence (AI) (Section 2.1.1). In neuromorphic computing, data are represented as timestamped events called spikes (Section 2.1.3). These spikes are processed by Spiking Neural Networks (SNNs), which consist of the interconnection of spiking neuron models (Section 2.1.2) and appears to be a viable candidate for low-power processing. Thanks to their ability to consider temporal information, SNNs are well-suited for temporal data processing. Due to this novel approach, classical learning methods must be adapted, or new learning methods inspired by the human brain can be used (Section 2.1.4).

To achieve low-power, neuromorphic computing must move away from Von Neumann architectures, and specific hardware architectures must be developed (Section 2.2). Initially designed as analog technology, SNNs have led to the proposal of fully analog architectures. However, the development of fully analog chips presents several challenges, particularly in terms of scalability and synapses implementations (Section 2.2.1). Mixed digital-analog approaches offer solutions to increase network scalability and implement synapses with digital memory (Section 2.2.2). In contrast to the two previous approaches, researchers propose a fully digital architecture based on the well-known CMOS technology, which appears highly scalable with low-power performances (Section 2.2.3).

Field Programmable Gate Arrays (FPGAs) appear to be a good entry point for fully digital architecture and hardware implementations. Thanks to their reconfigurability, FPGAs can be used to propose new design methods and hardware implementation innovations with low development time and cost while maintaining good latency and power performances (Section 2.3). Several points are important when designing FPGA emulators. The emulation strategy can have a significant effect on the power consumption of the emulator (Section 2.3.1). Moreover, the emulation strategy can have a significant impact on neuron implementation, a crucial design point where, depending on the mathematical complexity, can affect power, latency, and scalability of the emulator (Section 2.3.2). The synapse model is also a crucial point due to the limited memory provided by FPGAs, which can limit the scalability of network topology and increase power consumption (Section 2.3.3). Due to limitations of hardware resources, network implementations can be limited, and researchers propose several solutions to implement various network topologies (Section 2.3.4). To increase the computational capability of emulators, researchers propose to implement online learning methods to enable architectures for continuous learning (Section 2.3.5).

FPGA implementations represent an active research domain with the following challenges :

— **Mathematical model implementation**: The implementation of mathematical models, such as neuron or synapse models, learning rules, or other mechanisms such as convolutional kernels, is a crucial point on hardware implementation. Because neurons and synapses are the most important computational resources on SNN, their hardware implementation has a significant impact on architecture scalability, latency, and power performances.
— **Hardware constraints**: Due to limited computational and memory resources on FPGAs, the neurons and synapses model, the size of the network, and the implementation of additional features, such as learning methods, appear as a major challenge.
— **Maintaining event-driven computation**: Most works propose a clock-driven emulation strategy where neuron states are updated even in the absence of input spikes. This approach is often chosen due to the high mathematical complexity induced by event-driven algorithms. However, event-driven has been proven to be the best emulation strategy for low-power classification.

Many proposals from researchers exist to address these challenges. However, the vast majority of these solutions consist of close-source implementation, which forces researchers to re-implement a full architecture to propose a new feature. The need for open-source emulators that provide a common set of tools for researchers appears as crucial to advance research on SNN implementation on FPGAs. In the next chapter, we will present ModNEF, an open-source emulator, highly configurable with a stack of tools enabling users to create, train, and deploy an SNN on FPGA. After a complete presentation of our tools, we will compare our architecture with the other existing propositions.

3 ModNEF: a Modular Neuromorphic Emulator for FPGA

In the introduction, we set the following motivations :

- **Flexibility**: The emulator must be capable of emulating a wide variety of networks with various network topologies, particularly focusing on feed forward and recurrent topologies, with multiple spiking neuron models, including heterogeneous neural networks. Beyond network flexibility, it should provide fine-gained control over hardware-specific metrics, enabling users to optimize hardware resources, memory usage, power consumption, and latency of the hardware implementation.
- **Accessibility**: The emulator must be enough intuitive to be used by researchers without FPGA and hardware design expertise. Simultaneously, the source code must be comprehensible to SNN hardware developers, serving as an entry point for new research, which leads to the next requirement.
- **Extensibility**: The emulator should serve as a foundation for future research and development. It is crucial for our tools to support the development and integration of new neuron models, emulation algorithms, convolutional kernels, or learning methods.

In addition to these motivations, the state-of-the-art explorations raise additional challenges, specific to FPGA implementation:

- **Mathematical model implementation**: The implementation of mathematical models, such as neuron or synapse models, learning rules, or other mechanisms such as convolutional kernels, is a crucial point on hardware implementation. Because neurons and synapses are the most important computational resources on SNN, their hardware implementation has a significant impact on architecture scalability, latency, and power performances.
- **Hardware constraints**: Due to limited computational and memory resources on FPGAs, the neurons and synapses model, the size of the network, and the implementation of additional features, such as learning methods, appear as a major challenge.
- **Maintaining event-driven computation**: Most works propose a Clock-Driven emulation strategy where neuron states are updated even in the absence of input spikes. This approach is often chosen due to the high mathematical complexity induced by Event-Driven algorithms. However, event-driven has been proven to be the best emulation strategy for low-power classification.

In this chapter, we will present in detail our tool, Modular Neuromorphic Emulator for FPGA (ModNEF) [159] and how our tools answer to our initial motivations and address the research challenges. Written in Verry High Speed Integrated Circuit Hardware De-

scription Language (VHDL) and designed as a modular architecture, ModNEF proposes different neuron models, three based on the LIF neuron model and a IF neuron model. Each neuron is proposed with two different emulation algorithms and recurrent layers. These modules communicate through point-to-point connections and provide additional modules to manage modules interconnections, allowing users to develop complex network topology.

In the first Section 3.1, we will present the global architecture design. In Section 3.2, we will detail the neuron emulation module, including neuron mathematical models and neuron emulation algorithms. The third section, Section 3.3, will cover the various communication modules for on-chip and off-chip communication. Following that, Section 3.4 will address recurrent layers and their implementation in ModNEF. Before concluding if ModNEF answers to the presented motivations, we will present, in Section 3.5, the software high-level library provided to simplify the use of our emulator.

3.1 General Architecture

As the name suggests, ModNEF is a modular architecture. ModNEF provides several independent modules that users can connect to create more complex architectures capable of emulating SNN. In this section, we will describe the general aspects of ModNEF's architecture. We will first discuss how ModNEF implements an SNNs. After this brief presentation, we will present the computational paradigm adopted in ModNEF. Following that, we will introduce the spike representation within the architecture before detailing the common communication protocol. Finally, we will explain how the emulation process operates as a pipeline, where each module processes a different emulation step over time.

Figure 3.1 represents an example of a 2-layer FF-FC network implemented with ModNEF architecture. The Universal Asynchronous Receiver Transmitter (UART) module, which can be interpreted as the input layer of the network, receives input spikes from the host computer and transmits these to the module, which emulates the hidden layer. The hidden layer module and output layer module are connected with the common data bus, and the output layer sends output spikes to the UART component, which will transmit them to the computer.

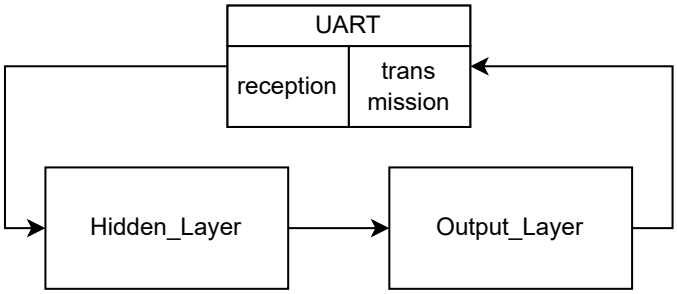

Figure 3.1 – Example of 2-layer network implemented in ModNEF. The architecture consists of three modules: two modules to emulate the two layers of the model and a UART module. The reception process of the UART module transmits input spikes to the hidden layer, emulating the input layer. Meanwhile, the transmission process receives spikes from the output layer module and transmits the output spikes to the host.

ModNEF adopts a clock-driven emulation strategy instead of an event-driven approach. This means that neuron states are updated at regular intervals, even in the absence of input spikes. This approach simplifies neuron update operation, as event-driven architectures require storing the last update date to accurately calculate membrane decay. Furthermore, Li et al.'s study [108] demonstrates that a clock-driven paradigm can be advantageous when spike density is high. In event-driven architectures, it is necessary to store and process generated events, which can negate the energy savings from fewer neuron updates if many events need to be processed and routed. Although the clock-driven paradigm may be less biologically realistic, it offers a favorable trade-off between computational complexity and power consumption.

In ModNEF, similar to many digital neuromorphic architectures [4, 143, 29], spikes are represented using the Address Event Representation (AER) format [19]. Instead of representing a spike as a 1-bit signal (i.e., 1 for emitted and 0 for not emitted), the AER format represents the spike as a neuron address. In some cases, such as with TrueNorth [6, 207], the address represents the address of the neuron that must receive the spike. However, in most cases, including ModNEF, the address represents the emitter neuron [207]. An advantage of this spike representation is the ability to include additional information within the data packet. For example, the binaural neuromorphic cochlea developed by Angel Jiménez-Fernández et al. [87] includes a 1-bit Most Significant Bit (MSB) to identify the sensor (left or right) that emitted the spike and a 1-bit Less Significant Bit (LSB) for spike polarity (positive or negative). The Darwin emulator [118] includes the date of spike emission. In ModNEF, it is not necessary to add more information to the data bus beyond the AER data. Due to point-to-point connectivity, the same neuron address can be shared by multiple neurons, provided they are not emulated within the same neuron emulation module. In other words, each emulation neuron module operates as an independent address space.

In ModNEF, AER packets are transmitted via point-to-point data bus using a common

transmission protocol. Figure 3.2 illustrates this data transmission protocol. When the transmitter module updates the neuron state, it sets the *emu_busy* signal to 1, indicating to the receiver module that it is ready to start the synchronization phase. The synchronization phase is based on a 4-phase handshake protocol using *req* and *ack* signals. Once the two modules are synchronized, the AER data is written onto the *AER* parallel bus. Additionally, the *spike_flag* signal is set to 1 to indicate whether the address on the *AER* bus represents a spike. This is necessary due to the arbitration phase, which will be described later in the section dedicated to emulation strategies (Section 3.2.2). After all data has been transmitted, the *emu_busy* signal returns to 0, concluding the spike transmission phase.

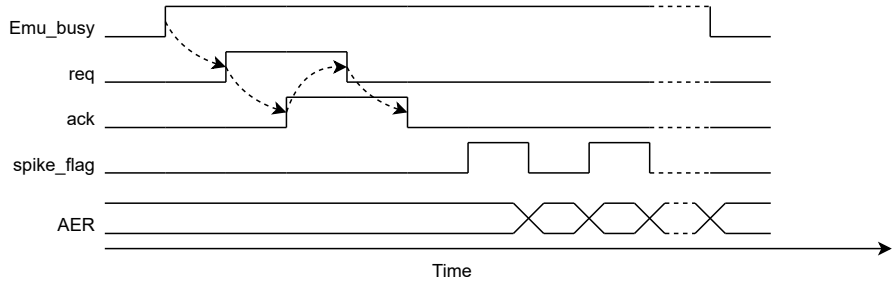

Figure 3.2 – Inter-module transmission protocol. The communication process begins when the transmitter sets the *emu_busy* at 1. The transmitter then initiates the synchronization process using a 4-phase handshake protocol. Once both processes are synchronized, the transmitter writes the neuron address onto the *AER* data bus and signals the presence of spike by controlling the *spike_flag* signals.

In ModNEF, the spike transmission + neuron update operations and spike reception processes operate in parallel within the neuron emulation unit. This means that while receiving input spikes, the neuron emulation unit updates the neuron state with previously received events and transmits spikes generated from previously calculated inputs. This mechanism, which will be described in more detail in the next section, results in a pipeline emulation. Figure 3.3 illustrates the emulation state process through time on the previous network illustrated by Figure 3.1. The first row of the UART module represented the transmission process (process which transmits output spikes to the computer), and the second row represents the reception process (process which receives input spikes). For the layer module, the first row represents the input spike reception process, and the second row represents the neuron update and transmission process. Due to the pipeline mechanism, the UART module needs to send extra empty steps, named as "**e**" in the figure, to receive spikes from the output layer.

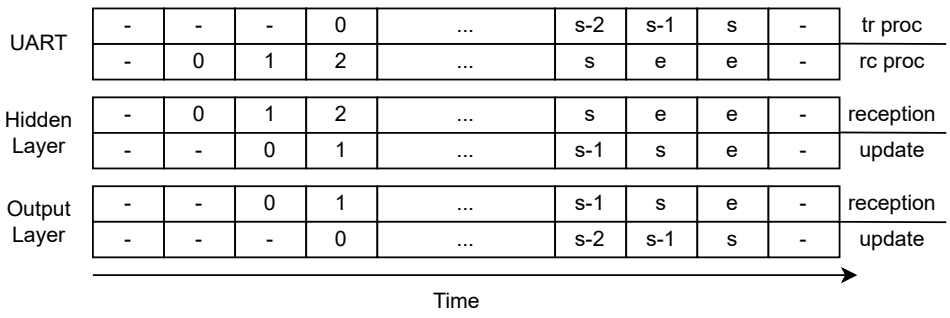

Figure 3.3 – ModNEF emulation pipeline. Each table illustrates the internal emulations state of both reception and update+transmission processes for each module. In ModNEF, the default scheduling scheme activates all emulation modules in parallel. Since spike reception and the neuron update plus the output spike transmission are executed concurrently, the emulation step of the output layer module is delayed compared to the preceding modules. Consequently, empty steps must be transmitted to "push" the data to the UART transmission process.

3.2 Neuron Emulation Modules

As discussed in Section 2.3.2, the Leaky Integrate and Fire (LIF) neuron model offers a favorable trade-off between biomimetic fidelity and mathematical complexity, while the IF model is the simplest. More complex neurons, such as the Izhikevich model, are developed by others FPGA researchers. However, previous works during my Master internship demonstrate this neuron model has a high power consumption due to multiplication. Consequently, three neuron models proposed by ModNEF are based on the LIF, each featuring distinct membrane dynamics and varying impacts on power consumption and hardware resource utilization. The last neuron model is based on IF model, which is the most hardware optimized neuron model with sufficient complexity for static data processing [4]. To enable users to tailor the architecture to their specific needs, we propose two different emulation strategies, each with unique implications for power consumption, hardware resource utilization, and inference time. In Section 3.2.1, we will describe the different mathematical and hardware conceptions of neuron models. In Section 3.2.2, we will detail and compare the different emulation strategies proposed in ModNEF.

3.2.1 Spiking Neuron Model

ModNEF proposes four different neuron models, three based on the Leaky Integrate and Fire (LIF) model [35] and the last one based on the Integrate and Fire (IF) model. We will first present the LIF-based neuron model and then the simpler IF neuron model. The membrane voltage dynamic, described by Equation 3.1, is inspired by the SNNTorch simulator used during network training [46]. In this model, the membrane

voltage V_{mem} accumulates input current I_{in}. The membrane decay is represented by $\tau = R \cdot C < 1$, which decreases the membrane potential. When the membrane potential reaches the threshold value V_{th}, a spike is emitted, and V_{mem} is reset to the resting value V_{rest}.

$$\begin{cases} \tau \frac{dV_{mem}(t)}{dt} = -(V_{mem}(t) + RI_{in}(t)) \text{ ,if } V_{mem}(t) < V_{th} \\ V_{mem}(t) = V_{rest} \text{ , else} \end{cases} \tag{3.1}$$

In digital hardware or software implementations of SNN, the differential equation is typically solved using the Euler method [61, 199]. The equation is transformed as the Equation 3.2, with $\beta = 1 - \frac{\Delta t}{\tau}$:

$$V_{mem}[t+1] = \begin{cases} \beta V_{mem}[t] + W I_{in}(t) \text{ ,if } V_{mem}[t] < V_{th} \\ V_{rest} \text{ , else} \end{cases} \tag{3.2}$$

In SNNTorch, the W factor in front of the input current I_{in}, initially set as $W = (1 - \beta)$, is considered a learnable weight with an independent dynamic separate from the β value.

Based on these neuron equations (eq. 3.1 and eq. 3.2), the first neuron model proposed by ModNEF is the Beta Leaky Integrate and Fire (BLIF) neuron model, named after the coefficient β used for membrane decay. Equation 3.3 presents the membrane dynamics of this neuron model.

$$V_{mem}[t+1] = \begin{cases} Rest(V_{mem}[t]) \text{ if } V_{mem}[t] > V_{threshold} \\ (V_{mem}[t] + I_{in}[t]) \cdot \beta \text{ else} \end{cases} \tag{3.3}$$

As we can see, the model differs from the SNNTorch model by multiplying $I_{in}[t]$ by β. This difference arises from how the input current I_{in} is calculated in ModNEF. As we explain in Section 2.1.2, the classical way to calculate the input current is described by the following Equation 3.4.

$$I_{in}(t) = \sum_i w_i \cdot x_i(t) \tag{3.4}$$

However, in ModNEF, as initially designed, the synaptic weight between the emulated neuron n and the spike received from neuron *AER* is directly added to the membrane voltage register. This approach avoids the need for additional registers, given the sequential reception of input spikes.

$$V_n[t] = V_n[t] + w_{n,AER}, \text{ if } spike\_flag = 1 \tag{3.5}$$

Another interesting point is the `Rest` function call when a neuron emits a spike. The BLIF model offers two different reset mechanisms: the hard reset, where the potential is reset to 0, and the soft reset, where the voltage is calculated as the difference

between the membrane voltage and the threshold voltage [4, 28]. The `Rest` function is described in the following equation 3.6.

$$Rest(V_{mem}[t+1]) = \begin{cases} 0 \text{ if } reset = zeros \\ V_{mem}[t] - V_{th} \text{ if } reset = subtract \end{cases} \tag{3.6}$$

Figure 3.4 shows the architecture of the BLIF architecture with hard reset in the left part and with the soft reset at the right.

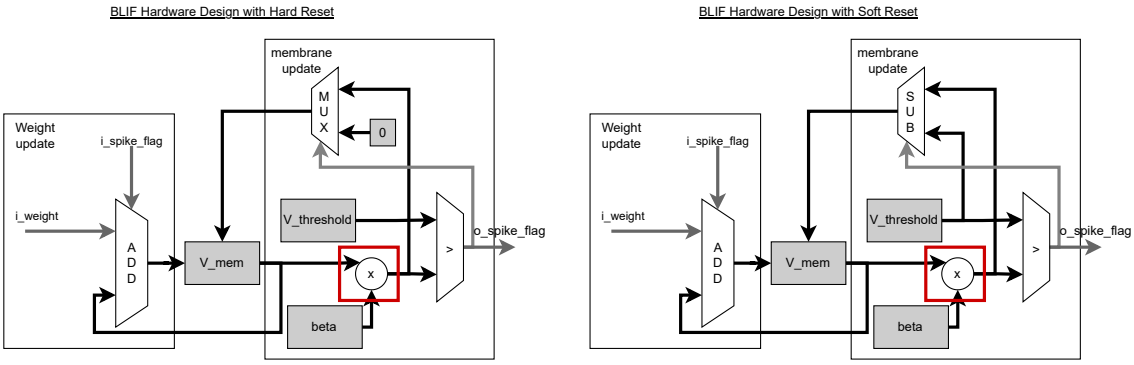

Figure 3.4 – BLIF hardware design with hard reset in left and soft reset in right. The input weight is accumulated into the membrane voltage register via an adder, controlled by the *i_spike_flag* signal. During the update operation, the membrane potential is multiplied by the beta value and compared to the threshold value. The output spike is used as a control flag for a multiplexer or a subtractor depending on the reset method, determining whether the next membrane voltage is written or the reseted mebrane potential based on the output spike flag.

The major problem with the BLIF neuron model it is the use of multiplication operations. Processing multiplications requires the use of Digital Signal Processing (DSP), which is power-hungry and consumes significant hardware resources. It would be beneficial to replace these multiplications with less power-intensive operations.
A potential solution to address this issue is to replace the multiplication with a shift register. By considering Equation 3.3 with $\beta = 1 - \frac{\Delta t}{\tau} < 1$, we can expand the multiplication to obtain Equation 3.7. In the specific case where $\frac{\Delta t}{\tau} = 2^{-s}$, the multiplication can be replaced by a shift register, as shown in Equation 3.8.

$$V_{mem}[t+1] = \begin{cases} Rest(V_{mem}[t]) \text{ if } V_{mem}[t] > V_{threshold} \\ (V_{mem}[t] + I_{in}[t]) - \frac{\Delta t}{\tau}(V_{mem}[t] + I_{in}[t]) \cdot \text{ else} \end{cases} \tag{3.7}$$

$$V_{mem}[t+1] = \begin{cases} Rest(V_{mem}[t]) \text{ if } V_{mem}[t] > V_{threshold} \\ (V_{mem}[t] + I_{in}[t]) - (V_{mem}[t] + I_{in}[t]) >> s \cdot \text{ else} \end{cases} \tag{3.8}$$

This neuron model is referenced as Shift Register Leaky Integrate and Fire (SRLIF), illustrated by Figure 3.5, and is used in Spiker+, for example [29, 28]. As for the BLIF neuron

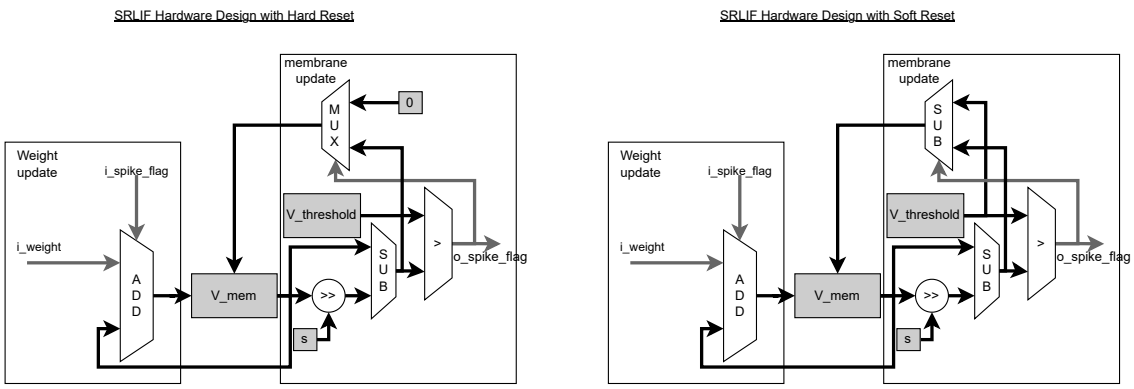

Figure 3.5 – SRLIF hardware design with hard reset, in left, and soft reset, in right. The architecture shares similarities with the BLIF design, but the membrane update mechanism differs. The membrane voltage is first shifted, and the shifted value is subtracted from the non-shifted value. The resulting value is then passed on the spike detector circuit, while the output spike flag is passed to the multiplexer or a subtractor depending on the reset mechanism.

model, SRLIF proposes two reset mechanisms: the hard reset where membrane voltage is set to 0 and the soft reset where threshold voltage is subtracted from membrane voltage.

Another solution to replace the multiplication in the initial neuron model, described in Equation 3.3, is to linearize the membrane decay by using subtraction instead of multiplication. This approach was proposed by Gupta et al. [63]. The neuron model, named Simplified Leaky Integrate and Fire (SLIF) is described in Equation 3.9 and illustrated in Figure 3.6.

$$V_{t+1} = \begin{cases} V_{rest} \text{ if } V_t < V_{min} \\ V_{rest} \text{ if } V_t > V_{threshold} \\ V_t + I_t - V_{leak} \text{ else} \end{cases} \tag{3.9}$$

The last neuron implemented is the Integrate and Fire (IF) neuron model described by Equation 3.10. This neuron model is characterized by its mathematical simplicity by accumulating input spikes without considering membrane decay. As for the previous SLIF neuron model, only the hard reset mechanism is implemented. This neuron model appears as a suitable solution for hardware implementation, especially when temporal information is not a crucial part of the input data [4]. The hardware architecture, depicted in Figure 3.7, comprises the input spike accumulation circuit and a comparator that controls the multiplexer, which applies the reset when an output spike is detected.

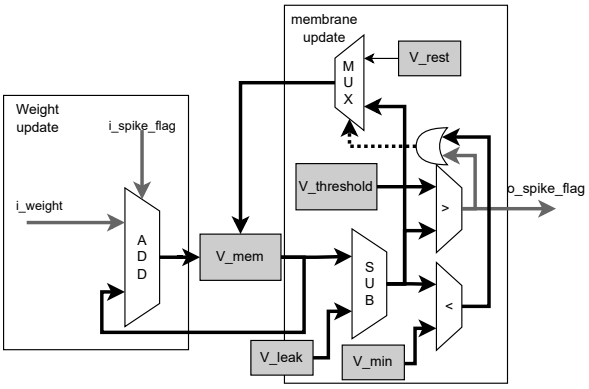

Figure 3.6 – SLIF hardware design. The membrane potential register is passed to a subtractor component to apply the membrane leakage. The result is then compared with two values: the threshold value and the minimum voltage value. The two resulting comparison flags are passed to an OR gate, which controls a multiplexer to apply the reset mechanism or not.

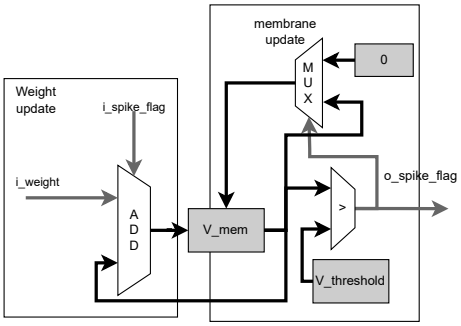

Figure 3.7 – IF neuron hardware design. The membrane potential is directly passed into a comparator, which controls the multiplexer to activate the reset mechanism.

$$V_{t+1} = \begin{cases} 0 \text{ if } V_t > V_{threshold} \\ V_t + I_t \text{ else} \end{cases} \tag{3.10}$$

The computational variables, used to represent the input current, membrane voltage, and neuron hyperparameters, bitwidth can be configured by users. The minimal variable size is determined based on the minimal and maximal amplitudes of the membrane voltage and input current. The bitwidth of these variables can be automatically determined during the VHDL code generation. During this thesis, most SNNs employ 16-bit variables to encode the internal state of neurons.

3.2.2 Emulation Strategy

As discussed in the previous section, ModNEF proposes four neuron models. Each model can be implemented using two different emulation strategies, or algorithms. The first strategy is parallel emulation, where each neuron has its own hardware circuit, allowing neuron update operations to run in parallel. The second strategy is sequential emulation, where a single hardware neuron is multiplexed to emulate the same group of neurons.

First, we will describe the common points between the two emulation strategies. In Section 3.2.2, we will detail the architecture of the parallel emulation strategy. Then, in Section 3.2.2, we will focus on the sequential emulation strategy.

The first common point between the two emulation strategies is the memory used to store the synaptic weights of the emulated neurons. The synaptic memory is implemented using Block Random Access Memory (BRAM) instead of DDR memory to enhance the temporal performance of memory access operations and preserve locality between memory and computational units, a crucial aspect in biological neural networks. The input AER address is used as the read address to read all synaptic weights between the neuron represented by the AER data and the emulated neuron with 1 clock cycle latency. Data organization is shown in figure 3.8.
The second common point between the two emulation strategies is the data reception Finite State Machine (FSM) illustrated by Figure 3.9. The *i_emu_busy* signal acts as an enable signal. The first two states facilitate process synchronization, while the third state waits for data reception. Due to the 1-clock cycle latency of memory read operations, the *i_spike_flag* signal must also be delayed by 1 clock cycle, resulting in the *spike_flag* signal. The reception FSM returns to the `idle` state when the transmitter module no longer needs to send data, indicated by the reset of *i_emu_busy* and when the last received spikes have been read from memory.

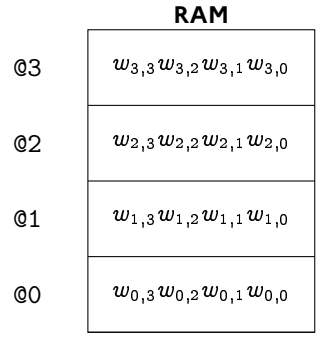

Figure 3.8 – Organization of synaptic weights on internal module memory. Each memory address represents the address of the input neuron. At each address, all synaptic weights between the input neuron and the emulated neurons are stored. The binary representation of each weight is concatenated into a single binary word, which is separate by giving only the specific part of this word to the neuron circuit.

Unlike the reception process, the transmission FSM operates differently between the parallel and sequential strategies. We will now describe each emulation strategy independently to highlight the major differences between them.

Parallel Emulation Strategy

First, we will describe the first emulation strategy designed: the parallel strategy. The parallel emulation strategy, illustrated in Figure 3.10, assigns each neuron its own hardware circuit, previously described in Section 3.2.1. Each neuron circuit locally stores the membrane voltage and the input current in a local register, as explained in the previously referenced section.

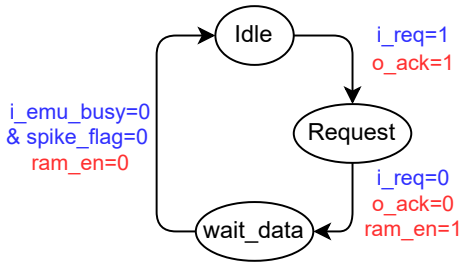

Figure 3.9 – Spike reception FSM diagram. When the transmitter module requests data reception, the reception process handles the synchronization phase. Following this, the FSM process waits for the completion of data reception, indicated by two conditions: the reset of the *i_emu_busy* signal and the reset of the *spike_flag* signal. This internal signal represents the input signal *i_spike_flag* delayed by a single clock cycle to account for the clock-cycle RAM read operation.

Thanks to the parallel implementation, the membrane update operation can be performed in parallel within a single clock cycle. The input current can be added to the internal membrane voltage register because the neuron update is performed during the spike reception synchronization. The neuron spike output is connected to a binary word that is sent to the input of the arbiter component. Each bit of this word corresponds to the spike output of a neuron: if a neuron emits a spike, the corresponding bit is set to 1; otherwise, it is set to 0.

The arbiter reads this binary word to convert the single-bit spike representation into AER and controls the *o_spike_flag* signal. The arbiter algorithm is detailed in Algorithm 1. The arbitration phase begins when the *start* signal is set to one by the transmission FSM. During this phase, a counter iterates through the *input_spikes* binary word, examining one bit per clock cycle. When the arbiter detects a spike, it updates the output *AER* bus and the *spike_flag* signal. The arbitration phase concludes once all bits of *input_spikes* have been processed.

The transmission FSM is described in Figure 3.11. The transmission process begins when the global signal *i_start_emu* is set to 1, indicating that the modules must compute a new emulation step. The FSM waits for the neuron update and checks if an arbitration phase is necessary. If no spikes have been emitted during the emulation step, the transmission process returns to the idle state to avoid unnecessary operations. If spikes have been emitted, the process signals the arbiter to start the arbitration operation by setting the *start_arb* flag to 1. Then, the FSM waits for the arbitration to complete before returning to the idle state. The total number of clock cycles required to emulate spiking neurons in the parallel module is given by equation 3.11, where $T(n)$ is the number of clock cycles necessary to emulate n neurons and F is the firing frequency of the emulated neuron group.

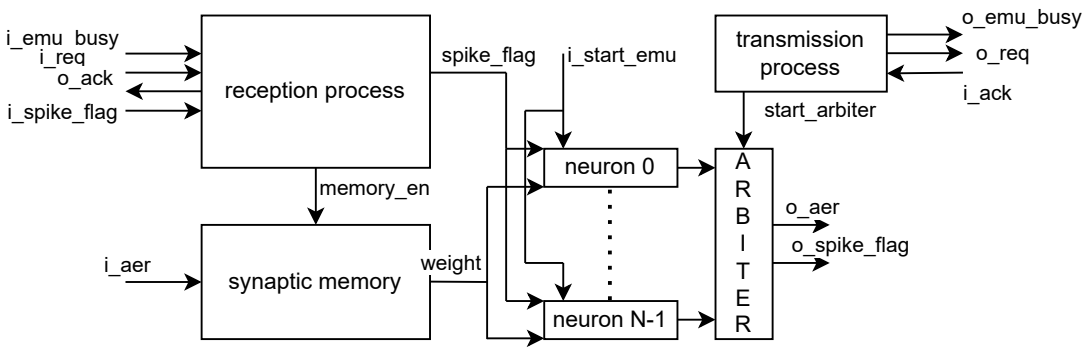

Figure 3.10 – Internal architecture of Parallel Emulation module. The reception process manages synchronization and controls both the memory enable flag and the spike flag. The AER data bus is directly connected to the synaptic weight memory, retrieving the synaptic weights with hardware partitioning for each implemented neuron. In the parallel emulation strategy, each emulated neuron is emulated by its own dedicated hardware circuit. The output of each neuron is connected to the arbiter component, which is controlled by the transmission process responsible for synchronization with the next module.

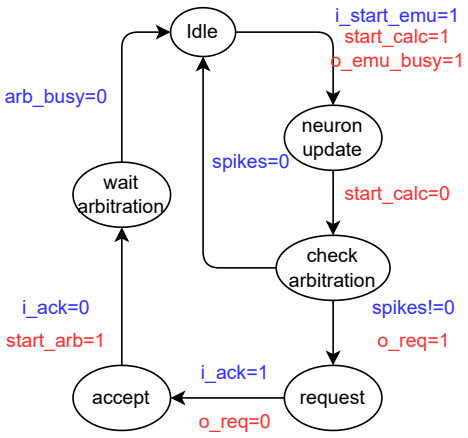

Figure 3.11 – Spike transmission FSM in Parallel Strategy module. When the FSM receives the *start_emu* flag, it triggers the neuron update operation. If no spikes are emitted, the FSM returns to the `Idle` state. Otherwise, the process synchronizes with the next module and will initiate the arbitration phase. Once the arbiter has completed its operation, the FSM transitions back to the `Idle` state.

Algorithm 1 Arbiter algorithm. The transmission process controls the start flag. The arbiter examines each input spike, encoded as a binary word, and controls the *spike_flag* signal and the *AER* data bus based on the spike value. This algorithm explains the necessity of implementing the *spike_flag* signal in the communication protocol. Since the arbiter can examine a neuron that has not emit a spike, a dedicated signal is essential to indicate the presence or absence of a spike or not.

Input: $start, input\_spikes[N]$
Output: $aer, spike\_flag$

$counter \leftarrow 0$
$run \leftarrow 0$
$aer \leftarrow 0$
$spike\_flag \leftarrow 0$
if $start = 1$ **then**
    $run \leftarrow 1$
end if
if $run = 1$ **then**
    if $counter = N$ **then**
        $run \leftarrow 0$
        $aer \leftarrow 0$
        $spike\_flag \leftarrow 0$
    else
        if $input\_spikes[counter] = 1$ **then**
            $spike\_flag \leftarrow 1$
            $aer \leftarrow counter$
        else
            $spike\_flag \leftarrow 0$
        end if
        $counter \leftarrow counter + 1$
    end if
end if

$$T(n) = 2 + F \cdot (2 + n) \tag{3.11}$$

Sequential Emulation Strategy

In addition to the parallel emulation strategy, ModNEF proposes a sequential emulation strategy, illustrated in Figure 3.12. In the sequential module, a single hardware neuron circuit is multiplexed to emulate the entire group of neurons. Since we simulate a group of neurons rather than a single neuron with the hardware circuit, it is necessary to save the neuron state in an additional memory. Unlike the parallel emulation strategy, the neuron update operation is performed during the reception of input spikes for the next emulation step. Consequently, the input current cannot be directly added to the membrane voltage; instead, it must be stored in another internal memory.

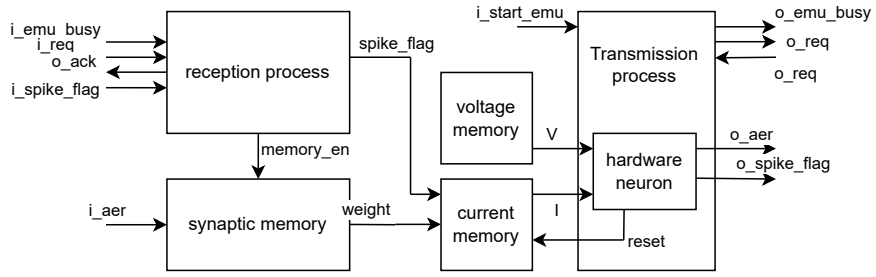

Figure 3.12 – Internal architecture of sequential emulation module. The reception process and the synaptic memory operate as for the parallel architecture. Two additional memories are implemented to store the membrane potential and input current for each emulated neuron. The transmission process manages syncronization with the next module and uses an internal hardware neuron circuit to sequentially emulate each neuron one by one.

The FSM of the sequential module is described in Figure 3.13. The transmission process begins with synchronization using a 4-phase handshake. Then, neuron emulation starts. The neuron emulation process is divided into three steps:

1. **Get Voltage**: Previous membrane voltage and input current are summed and stored into an internal register.
2. **Update Voltage**: The activation function is applied to the neuron membrane voltage.
3. **Set Voltage**: Spike detection and reset operations are performed, and the new membrane voltage is written into the voltage memory.

If all neurons have been emulated, the FSM transitions to the `emulation finish` state. If not, it returns to the `get voltage` state to emulate the next neuron. The total number of clock cycles required to emulate all neurons is given by equation 3.12, where n represents the number of emulated neurons.

$$T(n) = 2 + 3 \cdot n + 1 \tag{3.12}$$

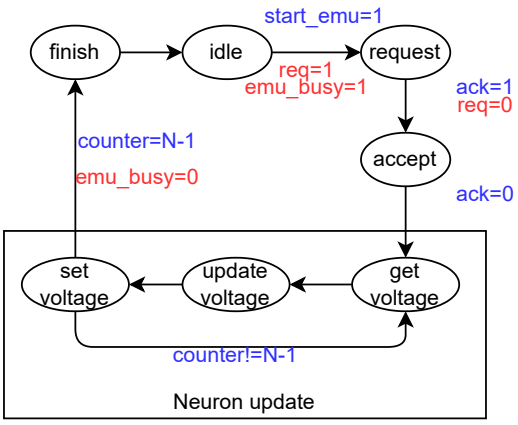

Figure 3.13 – Spike transmission FSM in sequential strategy module. The FSM begins with synchronization upon detecting the update emulation trigger, then enters the emulation loop. During this loop, the process reads membrane voltage and input current from memories, then the membrane voltage is updated and the spike is detected. At the end of each iteration, the reset mechanism is applied, and the updated membrane voltage is saved for the next emulate step. This loop repeats for all neurons before the FSM returns to the `Idle` state.

3.3 Communication Modules

In addition to the neuron emulation modules, ModNEF offers a range of specialized modules to enhance its functionality. These include two on-chip communication modules, which will be described in Section 3.3.1, and three off-chip communication modules, which we will describe in detail in Section 3.3.2.

3.3.1 On-Chip Communication Modules

ModNEF includes two specialized on-chip communication modules: the `Splitter` module and the `Merger` module. These modules affect the organization of the data bus.

The `Splitter` module splits one input data bus into two separate data buses, while the `Merger` module merges two input data buses into a single data bus. We will first describe the `Splitter` module and then the `Merger` module, which is more complex than the `Splitter`.

The `Splitter` module divides one input data bus into two separate data buses. The internal architecture of the `Splitter` module performs a simple `AND` operation be-

tween the two input acknowledgments. This operation signals the transmitter when both receiver modules are ready to receive data.

The `Merger` module is more complex than the `Splitter` module. It combines two input data buses into a single data bus by arbitrating communication between the two input modules and the output module. The FSM of the `Merger` unit is described in Figure 3.14. The `Merger` gives priority to the data bus connected to the *a* port. During the **`transmit_a`** or **`transmit_b`** state, the corresponding input *AER* and *spike_flag* signals are copied to the output *AER* and *spike_flag* signals. During the data transmission of one module, the second module will be put in standby. It results in what we call Arbiter-Induced Serialization (AIS). Due to arbitration of synchronization of the two input modules, their neuron update operations are serialized instead of a parallel running.

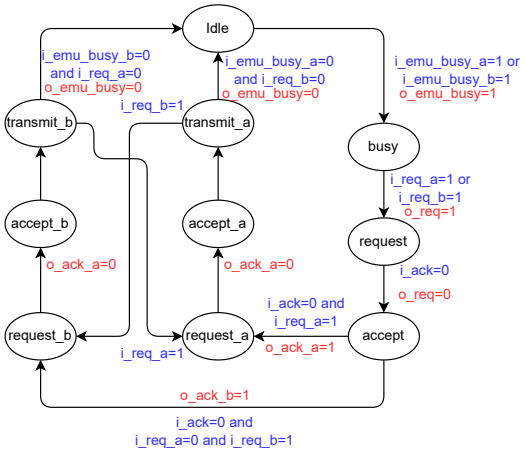

Figure 3.14 – Merger module FSM. When the process detects that at least one module requires data transmission, it synchronizes to the next module. After synchronization process, the FSM grants access to one of the input modules, prioritizing the module connected to port *a*. Once the module completes its data transmission, the FSM either grants access to the other module or returns to the `Idle` state.

As mentioned in Section 3.1, each neuron emulation module has its own address space, which can potentially be shared between the two input modules. The `Merger` module re-addresses these address spaces to prevent address overlap. This mechanism is illustrated in Figure 3.15. The address space of the module connected to *b* port is shifted before the address space of *a* port. It is crucial to consider this re-addressing in the internal synaptic weight output module.

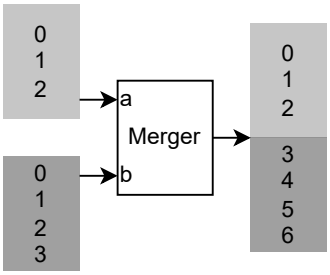

Figure 3.15 – Merger address space readdressing. The two input modules share overlapping addresses. The `Merger` reassigns the address space of the module connected to port *b* as the continuation of the address space of modules connected to port *a*, ensuring contiguous and unified address mapping.

3.3.2 Off-Chip Communication Modules

In addition to the on-chip communication modules, ModNEF offers two main off-chip communication modules, named `Uart_XStep` and `Uart_Classifier`, both based on the UART communication protocol. The UART interface was chosen for two main reasons. First, a fully functional UART implementation was already developed from prior work, enabling us to concentrate our efforts on the architecture design rather than peripheral development. Second, UART offers superior power efficiency for low-power applications, unlike alternatives such as Ethernet or Peripheral Component Interconnect Express (PCIe), which, while faster, introduce greater complexity and higher power consumption for bare-metal implementation.
These modules are responsible for receiving input spikes from the host device and transmitting them to the module, effectively serving as the input layer. They also receive spikes generated by the output layer module(s) and transmit these spikes to the host device, typically a computer in our experiments. Each module offers an alternative version that sends execution time in addition to output spikes. These versions are referred to with the module name extended with the `_Timer` suffix.
The internal architecture of the UART communication modules is very similar and is illustrated in Figure 3.16. Each module consists of a UART communication component and three internal controllers.
The internal UART component handles UART bus processing with internal read and write controllers. UART data are stored in internal queues: a First In First Out (FIFO) queue for input data and a Last In First Out (LIFO) queue for output data. This setup ensures that data is received and transmitted in the correct order to maintain the emulation step sequence. Input AER data are sent using a protocol described in Figure 3.17. The data packet starts with the `0x4B` header, followed by the total number of bytes the read controller must read. The controller then receives the different emula-

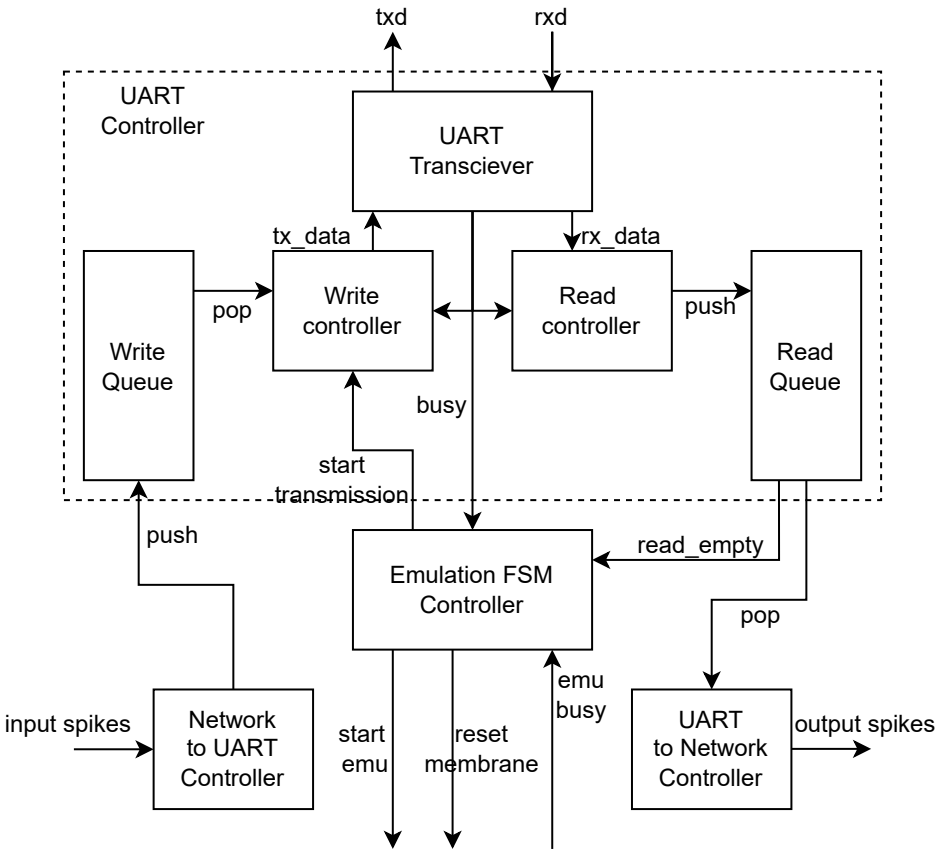

Figure 3.16 – Internal architecture of UART modules. All UART modules share similar architecture. Each module consists of three FSMs: one for transmitting spikes to the network, another for receiving output spikes, and a third for controlling the emulation steps. Additionally, a subcomponent manages UART communication, composed by a communication handler, read and write controllers, and internal memories.

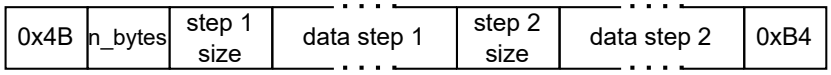

Figure 3.17 – UART transmission protocol. The data packet begins with a header byte equal to `0x4B`, followed by the number of bytes in the packet. The packet then includes the emulation steps using the AER format, with each step starting with the number of spikes it contains, followed by the spike data. Once all emulations steps are transmitted, either `0xBB` or `0xB4` is sent, indicating whether the controller must reset the neuron membrane voltage or not.

tion steps, each beginning with the number of AER events in that step. After receiving all bytes, the driver sends `0xB4` to conclude the data transmission. Alternatively, the tail byte `0xBB`, can be sent to signal the emulation controller to trigger the *reset_membrane* signal, which resets all emulated neuron membrane voltages to 0.
In addition to the UART controller, the communication modules implement three different controllers. The first is the emulation controller, which manages the *start_emu* flag when it detects a new emulation step. This detection is based on the UART busy state and the *emu_busy* signals, which are set to 1 when at least one module is active. The emulation controller also manages the *start_transmission* flag once all emulation steps have been completed, indicated by the absence of data in the read queue.
The second controller, the UART to network controller, reads input AER data from the read queue and sends this data to the SNN implemented on the chip. The controller begins by reading the number of input data items that need to be processed and then transmits them to the network.
The last controller, referenced in Figure 3.16 as Network to UART, receives spikes generated by the output layer and writes them into the internal UART write queue. This controller operates differently depending on the module type. In the `Uart_XStep` module, output spikes are sent directly to the internal queue. In the `Uart_Classifier` module, the controller counts the total number of spikes emitted by each neuron and sends this count, resulting in lower memory consumption and fewer data transmissions.
In addition to output spikes, the UART communication module can measure the onboard inference execution time. The module counts the number of clock cycles required to emulate each step and sends this count to the host device. The software driver then converts this count into time using the board's clock frequency.

3.4 Modules for Recurrent Layers

In this section, we will discuss the implementation of recurrence on ModNEF. As explained in Section 2.1.4, a recurrent network is one in which one or more neurons in a layer are connected to themselves or connected to previous feed forward layers. In this section, we will focus on local recurrence.
Neurons can be connected in a one-to-one manner, where each neuron is connected

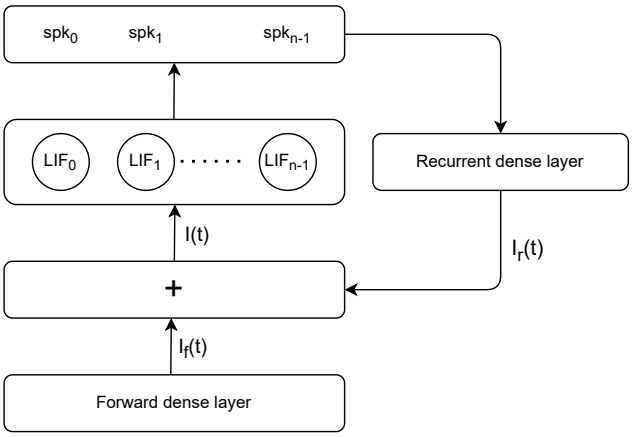

Figure 3.18 – Recurrent Network Topology at a Layer Level. Recurrent layers are implemented using two dense layers: one for the forward connections and another one for the recurrent connection. The input currents from both layers are summed and then passed to the neurons.

only to itself and not to other neurons in the layer. Alternatively, neurons can be connected in an all-to-all manner, where all neurons in the layer are connected to every other neuron. Another way to represent all-to-all recurrence is to use a first dense layer for feedforward connections and a second dense layer that takes as input the spikes emitted by the layer, as shown in Figure 3.18.
This results in a modification of the input current equation, as described in Equation 3.13. Here, Wf represents the forward synaptic weights, Xf is the input spikes, Wr are the synaptic weights of the recurrent dense layer, and Xr are the spikes emitted by the neuron layer during the last emulation step.

$$I_{in}(t) = \sum_i Wf_i \cdot Xf_i(t) + \sum_n Wr_n \cdot Xr_n(t) \tag{3.13}$$

In ModNEF, we focus on the approach where all neurons in a layer are connected to every other neuron. We propose two different methods to create recurrent connections between neurons within the same layer. In the first Section 3.4.1, we will describe a naïve approach using the `Merger` and `Splitter` modules, previously detailed in Section 3.3.1. In the second Section 3.4.2, we will detail the native recurrent module.

3.4.1 Recurrence based on Merger+ and Splitter Modules

The first method to implement recurrent topology with ModNEF is to use the two on-chip communication modules: the `Splitter` and the `Merger`. This method is named `Merger+Splitter` (MS). In this section, we assume you have already read Section 3.3.1 to understand how the `Merger` and `Splitter` modules work.
Figure 3.19 illustrates an implementation of a recurrent network topology using this method. The forward module, `Module 1`, which emulates 100 neurons in this exam-

ple, is connected to port *a* of the `Merger`. The output of `Module 2` is split into two separate data buses. The first bus connects to the next module, while the second bus connects to port *b* of the `Merger` module.

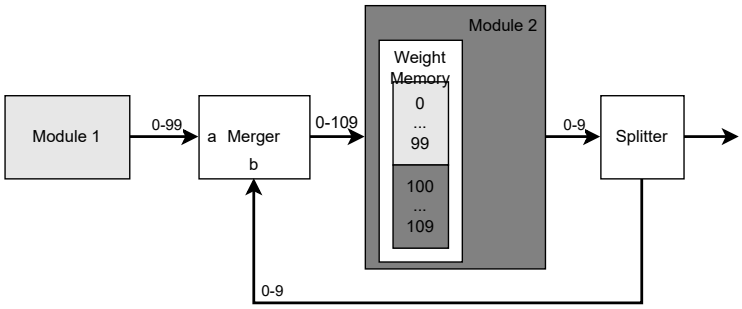

Figure 3.19 – MS based recurrent topology. The output of the recurrent module is divided using a `Splitter` module. One data bus is connected to the next module, while the other is connected to a `merger` module. The `Merger` connects the recurrent layer and the forward module, readdressing the input neurons address spaces. As a result, the recurrent weights are stored in the internal synaptic memory immediately following the forward weights.

Thanks to the re-addressing operation performed by the `Merger` module, the input neuron addresses in the example presented in Figure 3.19 range from 0 to 109. The recurrent module, `Module 2`, implements an internal weight memory with two address spaces. The first space from 0 to 99 stores the forward synaptic weights, while the second space, from 100 to 109, stores the recurrent synaptic weights.

Depending on which module the `Merger` selects to send its output spikes, the module will first compute one part of the input current and then the other part. In ModNEF, by convention, the forward module is connected to port *a* of the `Merger`, while the recurrent module is connected to port *b*.

Due to the `Merger`'s communication arbitration, there is an AIS of the neuron state update operation between the previous module, which provides forward input, and the recurrent module. Figure 3.20 illustrates this AIS operation. In the figure, the forward module transmits its spikes before the recurrent module. The recurrent module's spike transmission remains in the **`request`** state and must wait for the forward spike transmission to complete before starting the neuron update and spike transmission. This results in a lower inference speed.

Despite the AIS operation, this recurrence method is not limited to layer-level recurrence. It can also be used to create recurrent connections between two layers, thereby increasing the range of possible network topologies that can be implemented with ModNEF.

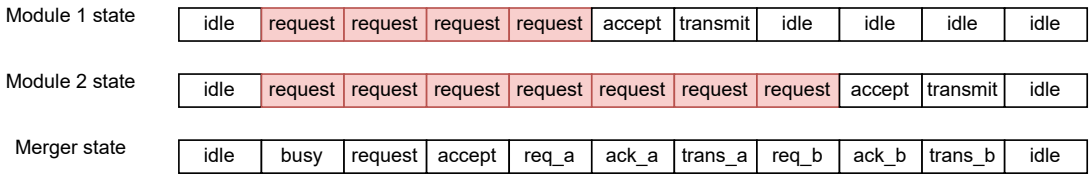

Figure 3.20 – Evolution of module states through time in `Merger+Splitter` recurrent architecture. Due to the `Merger`'s requirement to synchronize with the next module and selectively grant data transmission to one module while keeping the other in a waiting state. The neuron update of the recurrent module is delayed, which slows the emulation execution.

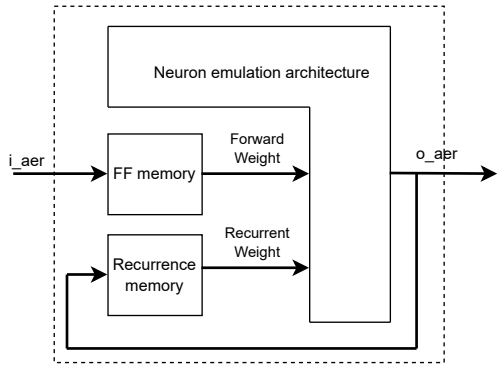

Figure 3.21 – Internal architecture of native recurrent modules. An additional memory is implemented to store the recurrent synaptic weights. The output *AER* bus is directly connected to the *address* port of this memory, enabling both input currents to be processed simultaneously.

3.4.2 Recurrent Based on Natural Recurrent Module

To address the major drawback of the `Merger+Splitter` (MS) recurrence method, we developed a native recurrent module where recurrence is directly handled locally within the module. In this section, we assume you have already read Section 3.2.2 to understand the initial architecture of neuron emulation modules.

The architecture of the native recurrent module is presented in Figure 3.21. The module implements a new internal memory to store recurrent synaptic weights. The output *AER* data bus is connected to the input read address of the memory, and the *o_spike_flag* signal is handled similarly to the input signal *i_spike_flag*.

Thanks to this architecture, the reception of forward spikes and recurrent (i.e., output) spikes can be processed in parallel, unlike the sequential processing imposed by the `Merger` module in the previously presented method.

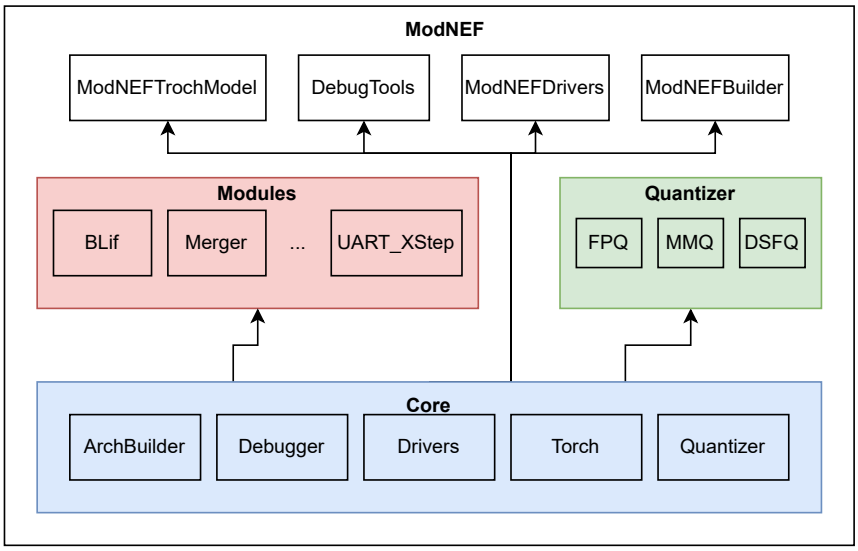

Figure 3.22 – ModNEF software library organization. The core module implements all base classes for creating ModNEF modules, drivers, or quantizers. Each module leverages these core base classes to build its software representation. Additionally, a quantizer submodule uses quantizer base classes to create quantization functions. Beyond these three submodules ModNEF library proposes four high-level utility classes to facilitate ModNEF model deployment.

3.5 ModNEF Software Tools

To make ModNEF accessible, we provide a collection of software tools that facilitate the use of the ModNEF architecture, from network training to onboard inference execution.

Figure 3.22 illustrates the organization of the software library. The foundation of the library is the core submodule, which provides all base classes necessary to create new software representations of ModNEF modules. The "modules" submodule regroups all already developed ModNEF modules created using these core classes. Alongside this, the quantizers submodule categorizes all quantization methods. Finally, four classes are provided for model creation:

— `ModNEFTorchModel` enables creating a PyTorch model compatible with ModNEF.
— `ModNEFBuilder` generates the corresponding VHDL file from a Python description, compatible with Torch models.
— `ModNEFDrivers` provides a collection of tools that generate the corresponding driver class from a YAML description file.
— `DebugTool` handles and processes output files produced during VHDL simulation.

In Section 3.5.1, we will describe the `Architecture Builder` tools. Following that, in Section 3.5.3, we will discuss ModNEF tools based on SNNTorch, including the `ModNEF`

neuron model, the **ModNEF**, and the **ModNEF model generator**. Finally, we will describe the ModNEF Rust **Driver**. The **Quantizer** tools will be covered in a subsequent Chapter 5.

3.5.1 ModNEF Architecture Builder

The first software tool provided by the ModNEF software library is the **Architecture Builder** tool. This tool enables users to create a ModNEF hardware architecture by describing it with high-level Python scripts.
The tool provides module classes that represent ModNEF hardware modules. Each ModNEF module has its own Python class and shares a common Python interface called **ModNEFArchMod**. These modules are invoked during VHDL code generation and convert neuron hyperparameters and synaptic weights using the user-defined quantization method.
To create a ModNEF architecture, the user will instantiate the modules and utilize the **ModNEFBuilder** Python class. This class has two significant roles. First, **ModNEFBuilder** stores the architecture as an oriented graph representation. The architecture modules represent the nodes of the graph and can be added to the internal graph by calling the *add_module(module)* method. To create a connection between modules, the user can call *add_link(source, target)*, which establishes a connection from the source module to the target. To simplify the creation of recurrent topologies, the user can call *add_link(source, source)* without needing to create a Merger and Splitter. The **ModNEFBuilder** will identify recurrent connections and add the necessary additional modules.
A particular case involves managing off-chip communication modules. The **ModNEFBuilder** defines a default data bus for I/O using the common communication protocol described in Section 3.1. To define a new entry point for the architecture, the user must call the *set_io(IOModule)* function. The IOModule must inherit from a specific class called **IOArch**, which implements the necessary methods specific to the I/O modules.
After creating the modules and defining the architecture topology, the user can generate the corresponding VHDL code. During code generation, the **ModNEFBuilder** will check for and develop any necessary recurrent connections. The code generator will then produce all required files, including a VHDL file with the architecture description and synaptic memory files used by the neuron emulation units.
A complete architecture creation script is provided in the following Code 3.1. First, we create the builder class with the name of the architecture and the network's input and output. Next, we create the UART communication module and the neuron emulation units, in this case, a SLif module with a parallel emulation strategy. We then add these modules by calling the *add_modules* method, which allows adding multiple modules in a single call. Afterward, we define the module connections using the *add_links*

method and set the UART module as the entry point of the architecture. Finally, we call the VHDL file generator, specifying the output file name and the clock signal name as arguments.

```
# creation of builder
builder = ModNEFBuilder(name = "mnist",
    input_layer_size = 784, output_layer_size = 10)

# create modules
uart = Uart_Classifier( name = "uart",
    input_layer_size = 784, output_layer_size = 10,
    clk_freq = 125_000_000, baud_rate = 921_600, queue_read_depth =
       10240,
    tx_name =  "uart_txd", rx_name = "uart_rxd")

unit = SLif(name = "neurons", input_neuron = 784, output_neuron = 10,
    v_threshold = 0.8, v_leak = 0.025, v_rest = 0.0, v_min = 0.0,
    strategy="Parallel", mem_init_file = "mnist.mem")
unit.weight_convert(weight_file = "./mnist_weight.txt")

# add modules and connection between module to builder
builder.add_modules([uart, unit])
builder.add_links([(uart, unit), (unit, uart)])
builder.set_io(uart)

# generate VHDL file
builder.to_vhdl(file_name = "modnef_mnist_arch.vhd", clock_name = "clock
   ")
```

Listing 3.1 – ModNEF architecture builder script of a single layer MNIST network. The script begins by instantiating a builder class and initializing each module of the architecture. Modules are then appended into the builder and interconnected through defined links. After specifying the entry point we generate the VHDL file.

With this organization, users can easily and quickly generate a ModNEF architecture and all necessary files, especially memory files. Thanks to the proposed Python interface, users can easily add new modules to the architecture generator library and introduce more functionalities to ModNEF. Moreover, the architecture builder can be used to generate a driver configuration, which will be described in the next section.

3.5.2 ModNEF Drivers

The ModNEF software library also provides a collection of Python classes used to transmit input spikes to the FPGA board and receive the output spikes resulting from hardware inference.
Each UART hardware module has its Python class equivalent. All these classes imple-

ment the `ModNEF_Driver` interface, which initializes the Rust driver class and provides methods to read and write data by manipulating the Rust driver.

The UART driver has been written in the Rust Programming Language to facilitate fast data manipulation and provide memory-safe tools for embedded systems using our ModNEF architecture. This Rust structure is utilized by a Python class, enabling users to easily communicate data with the FPGA and leverage Python's dataset packages.

Because the data transmission or reception format differs among the various UART hardware modules, each communication module is represented by a specific Python class. These classes implement the *run_sample(input_spikes, extra_step, reset_membrane, to_aer)* method to run an entire data sample. As discussed in Section 3.1, the ModNEF pipeline organization requires extra empty emulation steps to completely receive all emulation step outputs. The user can choose to send these extra emulation steps by using the `extra_step` parameter. The user can opt to reset the membrane potential after the end of sample emulation by setting the `reset_membrane` parameter to `True`. The UART module needs to convert input spikes with AER format before sending data to the FPGA. To do that, the drivers will use the *to_aer* function that converts input spikes to AER format. Since the input spike format can differ, we give the user the possibility to develop its own *to_aer* function, making drivers more flexible.

In addition, if the hardware UART sends the inference runtime, the driver will receive these extra bytes and convert the received clock counter to inference time.

To facilitate the creation of drivers, the architecture builder can generate a YAML configuration file. This file can be used to automatically generate a driver by calling the *load_driver_from_yaml(configuration, board_path)* method. If the user wants to create a new driver class, they can add it to the *drivers_dict* dictionary, which maps driver string identifiers to their Python classes.

3.5.3 ModNEF Torch Neuron Models

Although the neurons implemented in ModNEF are based on SNNTorch [46], our neuron models may differ slightly or significantly, as explained in Section 3.2.1. This can result in differences in model accuracy between the trained model and the onboard model.

To enable users to train network models with the ModNEF neuron model and minimize differences in accuracy between software and hardware, we provide a Python ModNEF neuron model based on the SNNTorch library, as described in the next Section 3.5.3.

In addition to the neuron model, we provide a PyTorch [144] module that implements additional functionalities, as described in section 3.5.3.

ModNEF Neuron Models

As discussed in Section 3.2.1, the neuron models proposed in ModNEF are based on the LIF model from the SNNTorch library and on the IF neuron model. However, ModNEF neuron models may differ from those proposed by SNNTorch, potentially leading to differences in accuracy between the software and onboard implementations. To minimize this accuracy gap, we have developed a ModNEF neuron model Python class inspired by the SNNTorch Python class.
We have developed a Python interface, `ModNEFNeuron`, which implements the `SpikingNeuron` interface from SNNTorch and includes additional features. In addition to running the ModNEF mathematical neuron model, the `ModNEFNeuron` class also allows for the direct conversion of the neuron model to its equivalent in the architecture builder and so integrates hardware description fields.
Moreover, `ModNEFNeuron` offers additional evaluation modes to assess hardware performance by simulating quantization or calculating hardware variables. These simulations help avoid overconsumption of hardware resources during architecture generation. These different simulation modes are controlled by the ModNEF network model, which will be described in the next section.

ModNEF Network Model

To assist users in developing ModNEF hardware models, we have developed an additional Network Model interface, based on PyTorch Module, that offers additional run modes.
In PyTorch, the `Module` class offers two run modes:

1. **Training**: Used to train the network model and activated by calling the method *`model.train()`*.
2. **Evaluation**: Used during model evaluation (i.e., inference) and activated by calling the method *`model.eval()`*.

In addition to the two run modes, the `ModNEFModel` interface offers four additional modes:

1. **Quantized Training**: Used to train the network model during QAT and activated by calling the method *`model.train(quant=True)`*.
2. **Quantized Evaluation**: Used to run model software evaluation with Post-Training Quantization (PTQ) and activated by calling the method *`model.eval(quant=True)`*.
3. **Hardware Estimation Evaluation**: This evaluation method assesses the model with PTQ and additionally computes several hardware specifications during evaluation. Currently, it calculates the maximal size of the UART internal memory component and the bitwidth of computational variable sizes for the neuron model. However, it is possible to compute more hardware specifications, such as

power consumption estimation. This evaluation method is activated by calling the method *model.eval(hardware=True)*.

4. **FPGA Evaluation**: The final evaluation method is FPGA evaluation. The model opens the UART driver and sends input spikes to the FPGA. This evaluation method is activated by calling *model.fpga_eval(board_path, driver_configuration)*.

Because the `ModNEFModel` can be used for both software and hardware inference, the *forward* function is decomposed into two different methods: *software_forward* and *fpga_forward*, to differentiate between the two forward hardware targets.

3.6 Conclusion

In this chapter, we presented our developed tool call Modular Neuromorphic Emulator for FPGA (ModNEF).
ModNEF is based on the interconnection of several independent modules with a common communication protocol. For the moment, ModNEF proposes four different types of modules: neuron emulation modules, which emulate a group of neurons with two different emulation algorithms, and three neuron models, intra-chip communication modules, which can manipulate module communication protocol and UART modules to communicate with the computer and transmit spikes.
Figure 3.23 presents an expanded version of the initial network example introduced in Figure 3.1. In this architecture, due to the modular architecture of ModNEF, each layer is implemented using a dedicated neuron module, configured with a specific neuron model and emulation strategy. These modules are independently configurable, providing users with fine-grained control over the hardware implementation. Inter-module communication on the protocol detailed in Section 3.1, where spikes are transmitted using a four-phase handshake synchronization in the AER format.
In addition to the architecture, ModNEF provide a software framework that allows the user to create an SNN compatible with ModNEF neuron models from the training to the FPGA deployment.
In the introduction of the chapter, based on the analysis of the state of the art, we raise several challenges that research tries to address:

— **Mathematical model implementation**: ModNEF proposes four different neuron models, three based on the LIF neuron model and the last one based on IF, with different membrane leakage mechanisms, each with different impact on architecture. Moreover, thanks to its modular architecture, users can develop new mathematical models to implement new mathematical models.
— **Hardware Constraints**: ModNEF proposes several emulation algorithms to get as much control over hardware implementation as possible. Moreover, we gave to the user the control on synaptic weight encoding bitwidth that gives to the user a high control on memory usage.

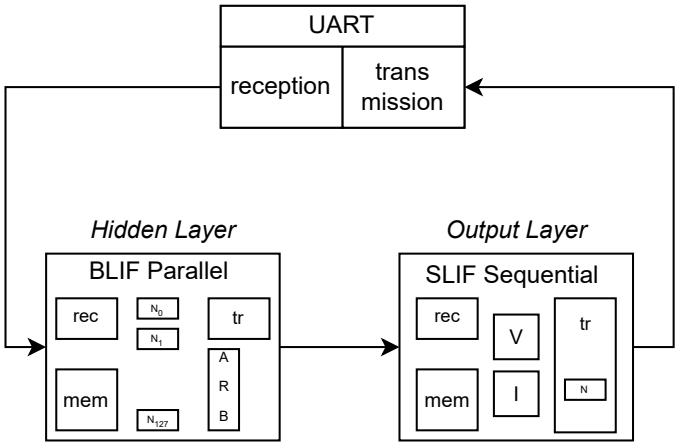

Figure 3.23 – Example of a 2-layer SNN implemented with ModNEF. The hidden layer is emulated with the BLIF neuron model with parallel emulation strategy, while the output layer is implemented with a sequential SLIF model.

— **Maintaining event-driven computation**: ModNEF adtops a clock-driven computing scheme. However, as one of the only open-source architectures, ModNEF can be used as a good starting point for further research on developing an event-driven architecture.

In addition to these challenges, we developed ModNEF with initial motivations. This first presentation provides preliminary evidence that our tool aligns with these objectives:

— **Flexibility**: Due to the variety of neuron modules and the inclusion of on-chip communication modules, ModNEF appears highly flexible, enabling users to create a wide variety of networks with complex topologies due to on-chip communication modules.

— **Accessibility**: ModNEF is provided with a Python software library, enabling users to train and deploy models without requiring hardware design expertise, using only Python descriptions. At the same time, the entire source code, from the Python library to VHDL files, is open-source, supporting our third motivation.

— **Extensibility**: As an open-source project, ModNEF allows researchers to access the code and create new modules. This is facilitated by a standard communication protocol and the modular design of both hardware architecture and software libraries.

Table 3.1 compares ModNEF with other notable FPGA emulators. We restrain our selection to works that propose complete emulators instead of works that focus on implementing a specific neuron model or mechanism. While the research community has recognized the lack of open-source solutions, the three existing open-source emulators, including ModNEF, were developed concurrently. ModNEF appears to be flex-

ible, offering more neuron models, configurable weight bitwidth and neuron update algorithms.
However, our architecture does have limitations: we employ the Clock-Driven paradigm, which is simpler than the Event-Driven paradigm but less power-efficient. Additionally, our tool lacks an online learning rule and convolutional kernel, reducing the network topology possibilities. Nevertheless, thanks to its modular design, ModNEF serves as a solid foundation for future research to address these constraints.
In this chapter we present architectural concepts of ModNEF and the software framework. However, we have not yet presented results demonstrating the architecture's ability to perform SNN inference tasks. In the next chapter, we will present different experiments demonstrating ModNEF capabilities and limitations.

Table 3.1 – Functionalities comparison between FPGA emulators. Only three emulators, developed in parallel and including ModNEF are available as open-source. ModNEF appears to be more configurable than other works but remains limited in their topology possibilities, the use of clock-driven emulation schemes, and the absence of online learning methods.

Design	**ModNEF**	Spiker+ [28]	SPLEAT [4]	NeuroCoreX [55]	Li et al. [108]	FPGA_NHAP [114]	Han et al. [68]	Fang et al. [47]
Open Source	**Yes**	**Yes**	No	**Yes**	No	No	No	No
Publication Date	2025	2024	2022	2025	2021	2022	2020	2019
Paradigm	Clock-Driven	Clock-Driven	**Event-Driven**	Clock-Driven	**Adaptative**	**Adaptative**	**Event-Driven**	**Event-Driven**
Online Learning	No	No	No	**STDP**	**STDP**	No	No	**STDP**
Neuron Model	**3 LIF + IF**	3 LIF	IF+LIF	LIF	IZ	LIF+IZ	LIF	LIF
Neuron Time Multiplexing	**Configurable**	No	Yes	Yes	Yes	Yes	N/A	Yes
Weight bw	**Configurable**	**Configurable**	N/A	8	N/A	16	16	16
Software Tools	**Yes**	**Yes**	**Yes**	**Yes**	N/A	N/A	N/A	N/A
Topology	FF-FC, Rec	FF-FC, Rec	**FF-FC, Rec, CNN**	FF-FC	FF-FC	FF-FC	FF-FC	FF-FC
Design	Gupta et al. [63]	DeepFire2 [13]	BiCoSS [201]	Zhang et al. [211]	Ali et al. [7]	Mishra et al. [129]	Wang et al. [188]	S2N2 [91]
Open Source	No	No	No	No	No	No	No	No
Publication Date	2020	2023	2022	2019	2024	2025	2018	2021
Paradigm	Clock-Driven	Clock-Driven	Clock-Driven	**Event-Driven**	Clock-Driven	**Event-Driven**	**Event-Driven**	N/A
Online Learning	No	No	**STDP+Axonal**	No	No	No	No	**Delay**
Neuron Model	Simplified LIF	IF	LIF	LIF	LIF	Simplified LIF	LIF	LIF
Neuron Time Multiplexing	No	Mixed	Yes	No	No	N/A	Yes	N/A
Weight bw	24	3/8	N/A	8	16	32	8	N/A
Software Tools	N/A	**Yes**	N/A	N/A	N/A	N/A	N/A	N/A
Topology	FF-FC	CNN	N/A	FF-FC	FF-FC	FF-FC	FF-FC - Rec	FF-FC - CNN

4 Test Bench

We previously described ModNEF, a modular neuromorphic emulator design for FPGA target. Due to its modular architecture and on-chip communication module, ModNEF can be used to emulate a wide variety of network topologies.
In this chapter, we will present various results demonstrating the emulator capacity, all available on the following github link: `https://gitlab.univ-lille.fr/bioinsp/modneftheseresult`. Since the IF neuron model was recently integrated into ModNEF, no benchmarks currently utilize this model. This neuron model will be used in a future chapter.
First, in Section 4.1 we will present the common experimental protocols and metrics we used to validate ModNEF architecture. Then we will present classification tasks based on feed forward network topology in Section 4.2. Thereafter, in Section 4.3, we will present classification tasks with recurrent network topology. After classification sections, we will explore the limitations of our architecture in Section 4.4 and, in the next Section 4.5, we will explore different architectural choices to prevent hardware limitations.

4.1 Experimental Protocol and Metrics

In this section, we will describe the global experimentation protocol we used during the thesis. First, in Section 4.1.1, we will describe the resources used during experimentation, including the FPGA board used for onboard inference and the laptop specifications used for training and software inference. We will then detail the experimental protocol followed during these tests in Section 4.1.2. Finally, in Section 4.1.3 we will outline all the metrics used to validate our experiments.

4.1.1 Experiments Resources

In this section, we will present the materials used during experimentation. We will first describe the FPGA target board and then the computer used for training and software inference. We will compare the speed and energy performance of the computer with that of the FPGA.
First, all experiments were conducted on an Arty Z7: Zynq-7000 SoC Development Board equipped with a XC7Z020-1CLG400C FPGA chip operating at a clock frequency of 125 MHz. The hardware resources are summarized in Table 4.1.
Larger FPGA boards, such as the Zynq UltraScale, were not used during experimentation due to equipment limitations in the laboratory. Due to the hardware resource limitations of the Arty Z7 board, we focused our experiments on small networks.
Because ModNEF does not support online training, the models were trained on an ASUS TUF Gaming F17 TUF707ZC4-HX094W laptop. This laptop is equipped with 16 GiB of DDR4 3200 MHz RAM, a 64-bit 12th Gen Intel(R) Core(TM) i5-12500H CPU, and a GeForce RTX 3050 Mobile GPU with 4 GiB of Graphic Double Data Rate (GDDR)6 memory.

Table 4.1 – Hardware resources capacity of Arty Z7 Zynq-700 SoC Development Board.

Resource	Capacity
LUT	53,200
LUTRAM	17,400
FF	106,400
DSP	220
BRAM (Kbit)	5,040

4.1.2 Experimental Protocol

In this section, we will describe the experimental protocol used to train and validate our models.
The experimental protocol follows these steps and is summarized in Figure 4.1:

1. **Dataset Selection**: We first select the target dataset for the experiments. The MNIST dataset was only used as a proof-of-concept dataset. For other experiments, we prefer natural neuromorphic datasets such as N-MNIST, SHD or DVS Gesture.
2. **Network Topology Definition**: The second step involves defining the network topology and layer hyperparameters, such as neuron parameters.
3. **Short Training Session**: To validate the network model, we run a short training session with a high learning rate and a low number of epochs to verify if the model can handle the classification task.
4. **First FPGA Implementation**: To verify if the trained model can be implemented on the FPGA board, we synthesize the circuit of this initial model. If the model does not fit into the FPGA, we need to find a smaller model that fits onto the board.
5. **Training**: Once the model is validated, we run a longer training session with a lower learning rate to achieve a more accurate model.
6. **Evaluations**: After training the model, we run several evaluations:
   a) Full Precision Software Evaluation
   b) Post-Training Quantization (PTQ) Software Evaluation
   c) FPGA Evaluation
7. **Metrics**: If the FPGA evaluation is considered good, with a low accuracy drop compared to the software evaluation, we gather the different metrics of the model.

To train our network, we used the SNNTorch simulator [46] with the ModNEF Torch neuron models (see Section 3.5.3 for more details). To train the different models, we employed the Backpropagation Through Time (BPTT) learning method, which is an adaptation of the BP learning method [194, 212, 20, 46, 158].
The classification process is based on a rate classifier, where the spikes generated by each neuron in the output layer are counted. The neuron that emits the most spikes is considered the winner of the classification phase.

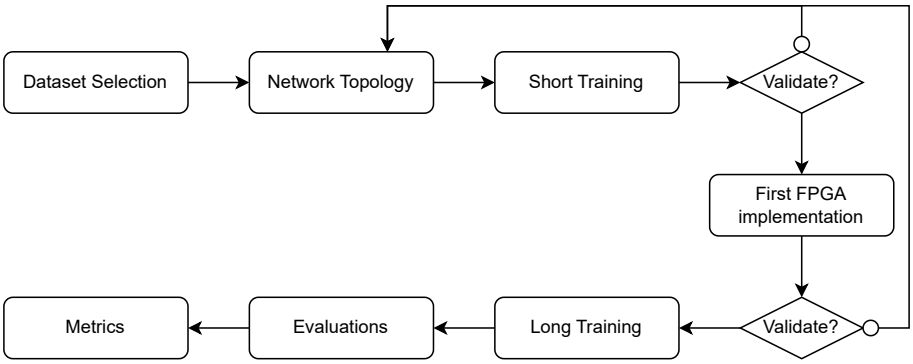

Figure 4.1 – Experimental protocol flow organization. The process stats by selecting the dataset and training a network with a low number of epochs to validate the network topology and the layer hyperparameters. If we achieve good accuracy, the network is implemented on FPGA to verify its compatibility with our board. If the model achieves good accuracy and can be deployed on our board, we train our model with a higher number of epochs, and we run comprehensive evaluation and metrics recording.

The final step of the experimentation process involves evaluating various metrics to assess our model's performance. In the next section, we will describe the chosen metrics.

4.1.3 Metrics Definition

To validate our experiments, we define various metrics to evaluate the performance of our model and the corresponding hardware architecture. We categorize these metrics into three groups, depending on whether they are shared between the software and hardware models or not.
The first metric we used to evaluate model performance is accuracy. We evaluate accuracy using three different methods:

1. **Full Precision**: Accuracy measured with the best model evaluated using SNNTorch without any quantization.
2. **PTQ Simulation**: Accuracy measured with the best trained model evaluated using the SNNTorch simulator, incorporating quantization during inference.
3. **FPGA**: Accuracy measured from FPGA inference.

The second category of metrics consists of metrics shared between software (i.e., CPU and GPU hardware target) models and hardware (i.e., FPGA hardware target) models. These shared metrics are:

1. **Inference Speed**: For software models, inference speed is measured during power estimation using PyJoules, which provides the execution time of the measured Python block. For the FPGA architecture, we use the `UART_XStep_Timer`, which counts and transmits the number of clock cycles during architecture execution (see Section 3.3.2 for more details). The execution time measurement excludes the latency of the communication protocol, as its implementation can be modified or optimized independently. Thus, including communication latency

would not provide a consistent evaluation of the core architecture performance. After receiving the counter value, we multiply it by the clock frequency to obtain the execution time of the inference.

2. **Energy Consumption**: For software energy consumption, we used the PyJoules Python library [22], which measures the energy consumption of a block of Python code. We measure the energy consumption of a batch forward pass and calculate the mean value across all batches. It is important to note that the energy consumption of GPUs strongly depends on the batch size. Therefore, the reported results should be interpreted with caution: they provide a reliable order of magnitude but not an absolute measure independent of the experimental setup. For FPGA energy consumption, we use the power estimator provided by Vivado and calculate the energy by multiplying the power estimation by the inference time: $E_{FPGA} = P_{vivado} * T_{inference}$.

The last category of metrics is available only for the FPGA model architecture and consists of the following metrics:

1. **Dynamic Power Consumption**: The dynamic power consumption is provided by vectorless Vivado power estimation.
2. **Static Power Consumption**: The static power consumption is provided by vectorless Vivado power estimation.
3. **BRAM Consumption**: The memory footprint of the architecture is provided by the Vivado utilization report. The metric follows the format: `A;B (X%)`, where `A` is the memory consumption of the network circuit, `B` is the memory consumption of the UART module, and `X` is the percentage of resource usage.
4. **FF Consumption**: This metric represents the consumption of Flip-Flop (FF) hardware resources. It follows the format: `A;B (X%)`, where `A` is the FF consumption of the network circuit, `B` is the FF consumption of the UART module, and `X` is the percentage of resource usage.
5. **LUT Consumption**: This metric represents the consumption of LUT hardware resources. It follows the format: `A;B (X%)`, where `A` is the LUT consumption of the network circuit, `B` is the LUT consumption of the UART module, and `X` is the percentage of resource usage.
6. **LUTRAM Consumption**: This metric represents the consumption of Look Up Table Random Access Memory (LUTRAM) hardware resources. It follows the format: `A;B (X%)`, where `A` is the LUTRAM consumption of the network circuit, `B` is the LUTRAM consumption of the UART module, and `X` is the percentage of resource usage.
7. **DSP Consumption**: This metric represents the consumption of DSP hardware resources. It follows the format: `A;B (X%)`, where `A` is the DSP consumption of the network circuit, `B` is the DSP consumption of the UART module, and `X` is the percentage of resource usage.

We noticed that the power estimation provided by Vivado with the `Uart_XStep_Timer` module can underestimate the DSP power consumption for unknown reasons. To mitigate this estimation error, the power estimation and resource usage report were conducted using the `Uart_XStep` module as the UART component. The differences between these two architectures are negligible and only

concern the UART component consumption, not the network implementation. We are currently working on a more accurate power estimation methodology. However, the tool is not yet developed and validated.

4.2 Classifications Results with Feed Forward

In this section, we will present the various experiments conducted to validate and evaluate the performance of the ModNEF architecture for FF-FC network topology. First, in Section 4.2.1, we will present the results on the MNIST dataset as a proof of concept. Next, in Section 4.2.2, we will discuss ModNEF performance with natural neuromorphic dataset classification tasks with the N-MNIST dataset.

4.2.1 Proof of concept with MNIST

To verify the proper functioning of the ModNEF architecture, we first used the MNIST dataset [99]. Although MNIST is not a neuromorphic dataset and is considered well-solved [101], it served as a proof of concept. Given that MNIST is widely used to benchmark the performance of various AI models, including SNN models [44], we use it to compare our work with other FPGA architectures for SNN implementation.

Experimental Setup

MNIST was introduced in a study by Lecun et al. in 1998 [98, 99]. It consists of 28x28 8-bit grayscale images of handwritten digits ranging from 0 to 9. The training set comprises 60,000 images, and the test set comprises 10,000 images.
Since MNIST is not a neuromorphic dataset, it is necessary to represent input data as temporal data. During training and software-based evaluation (on CPU and GPU), we sent the input data several times, to simulate temporal data. During FPGA evaluation, we convert input into spike representation. To achieve this, we use a Poisson generator [73, 44] to convert each pixel into a train of spikes, where the pixel value is used as the mean of a binomial distribution. The Poisson encoding is run as data preprocessing on the host computer, and the generated spikes are sent into the FPGA. In our case, we fix the number of time steps to 100 ms, sending the input data 100 times during software-based evaluation and training and generating a spike train of 100 ms with a 1 ms time step. The work of Fang et al. [47] highlights the importance of the conversion process, and the encoding scheme used can significantly impact the inference speed. To reduce the inference speed and energy consumption, we can change the encoding scheme or the number of steps.
The model used for the MNIST classification task is a simple FF-FC network topology. Table 4.2 summarizes modules configurations of the deployed model. The network consists of a hidden layer with 128 neurons and an output layer with 10 neurons, each implementing the SLIF neuron model. Given the simplicity of the MNIST classification task, the simplified membrane voltage dynamics of the SLIF model are sufficient to achieve high accuracy while maintaining lower power consumption compared to more complex models such as BLIF.
To reduce emulation latency, both layers are emulated using a `Parallel` strategy. The internal computational variables are encoded with 16 bits, while synaptic weights are represented using 8-bit encoding, balancing accuracy and resource efficiency.

Table 4.2 – Detailed module configurations for the 2-layer MNIST model. Both layers are emulated with the SLIF neuron model emulated in parallel with 16-bit computational variables and 8 bits to encode synaptic weights.

Layer	N neuron	Neuron model	Strategy	Variable size	Synaptic Weight
Hidden Layer	128	SLIF	Parallel	16	8
Output Layer	10	SLIF	Parallel	16	8

Experimental Results

We will present the results of our experiments.

Table 4.3 summarizes the accuracy values obtained using different inference methods. As shown, the accuracy measured with PTQ is higher than the full precision accuracy, likely due to reaching a different optimal point. Additionally, the FPGA accuracy is higher than both other evaluation methods. This can be attributed to two factors: the previously mentioned PTQ optimization and the spike conversion used during FPGA inference.

Table 4.3 – Accuracy results for the 2-Layer MNIST model. In this scenario, the onboard evaluation achieves better accuracy than both quantized software and the full precision evaluation. This improvement can be attributed to reaching better local minima due to quantization error and the spike conversion process specific to the FPGA evaluation.

Evaluation Method	Accuracy (%)
Full Precision	96.66
PTQ Simulation	96.58
FPGA	97.06

Since the network is relatively small, the hardware resource consumption is not very high, as shown in Table 4.4. The primary resource consumed is memory, with 27.14% usage. Because we used the SLIF neuron model, no DSP resources were used due to the absence of multiplication operations, resulting in low dynamic power consumption.

Table 4.5 references the inference speed and energy consumption of the model on different hardware targets. The FPGA model achieves classification tasks 3.7 times faster than CPU and GPU. Thanks to this high inference speed and the low-power consumption of FPGA, the energy consumption for classification tasks is 557 to 953 times better than other hardware targets.

MNIST was used as a proof-of-concept dataset, and these metrics demonstrate that ModNEF is a promising architecture for high-speed, low-power classification tasks. ModNEF can achieve accuracy comparable to software-based evaluation methods while offering superior inference speed, even in real-time, and lower power and energy consumption.

In addition to serving as a proof of concept, the MNIST dataset was also used as a comparison dataset with other works that implement SNN architectures on FPGA chips.

Table 4.4 – Hardware metrics results for the 2-Layer MNIST model. The BRAM is the most consumed resource due to synaptic weight storage, followed by the LUT usage driven by the parallel emulation strategy. Since the model uses the SLIF neuron model, DSP consumption remains at 0, leading to low dynamic power consumption.

Metrics	Results
Dynamic Power (mW)	90
Static Power (mW)	109
BRAM (KBit)	1080 ; 288 (27.14%)
FF	2616 ; 381 (2.82%)
LUT	8935 ; 640 (18.00%)
LUTRAM	0 ; 384 (2.21%)
DSP	0 ; 0 (0.00%)

Table 4.5 – Shared metrics result for the 2-Layer MNIST model. The FPGA target delivers the fastest hardware performance, significantly better than software-based targets. This high inference speed, combining with low-power consumption, results in a substantial reduction in energy consumption.

Target	CPU	GPU	FPGA
Inference Speed (ms)	0.437	0.469	0.116
Energy Consumption (mJ)	20.97	12.27	0.022

Comparison with other works

Table 4.6 presents a comparison of ModNEF's performance with other works in the field.
Since ModNEF and Spiker+ share a common network topology, dataset transformation, and hardware architecture, we will focus our comparison with Spiker+. In fact, a bigger network topology or a different neuron model can significantly impact the model and hardware performances.
ModNEF achieves higher accuracy than Spiker+ but requires more memory, as Spiker+ uses 4-bit synaptic weights encoding, leading to lower power consumption. ModNEF appears as faster than Spiker+. However, it is important to notice the latency on ModNEF is calculated without input and output spike transmission while Spiker+ takes these steps into consideration [28].
Overall, ModNEF delivers competitive accuracy and power efficiency results compared to state-of-the-art solutions. To improve accuracy, larger neural networks, such as those in the FPGA-NHAP [114] architecture, Abderrahmane et al.'s work [3], or Han et al. [68] study, could be adopted.
We can see that the power consumption per implemented neuron is comparable to other similar works. However, there is a notable gap between clock-driven and event-driven approaches. In clock-driven architectures, the neuron state is updated at each

emulation step, even without new information, leading to higher power consumption per neuron. In contrast, event-driven architectures update neuron states only when new information arrives, resulting in lower power consumption. Liu et al. [114] demonstrate that clock-driven architectures can be advantageous if the spike density within the network is high. In event-driven systems, spikes must be processed, and the energy impact can be significant if the number of spikes is high.

Table 4.6 – Comparison between ModNEF and other FPGA emulators. This comparison focuses on Spiker+ due to similar network topology and hardware architecture. While ModNEF achieves higher accuracy, Spiker+ demonstrates better power performance due to lower BRAM consumption. However, thanks to its high inference speed, ModNEF reaches better energy consumption than Spiker+.

Design	**ModNEF**	**Spiker+ [28]**	**FPGA-NHAP [114]**	**Abderrahmane et al. [2]**
Paradigm	Clock Driven	Clock Driven	Clock/Event Driven	Event Driven
FPGA	XC7Z020-1CLG400C	XC7Z020	XC7K325T	5CGXFC7C7F23C8
Clock Frequency (MHz)	125	100	200	83.51
Topology	784-128-10	784-128-10	784-1024-1024-10	784-300-300-10
BRAM (Kbit)	1368	**640**	3006	8676
Accuracy	**97.06%**	93.85%	97.70%	98.00%
Inference time (ms)	**0.116**	0.780	4.8	N/A
Power (W)	0.190	**0.180**	0.351	N/A
Power/neurons (mW/neurons)	1.37	**1.30**	0.17	N/A
Energy (μJ)	**21.99**	140.4	1,684	N/A
Design	**Han et al. [68]**	**Li et al. [108]**	**Zhang et al. [211]**	**Fang et al. [47]**
Paradigm	Event Driven	Clock/Event Driven	Event Driven	Event Driven
FPGA	XC7Z045	Virtex-7VC707	xc7vx690t	Cyclone V
Clock Frequency (MHz)	200	100	Asynchronous	75
Topology	784-1024-1024-10	784-200-100-10	784-512-384-10	784-600-10
BRAM (Kbit)	1530 + 1GB of DDR3	N/A	8192	N/A
Accuracy	97.06%	92.93%	98.00%	95.66%
Inference time (ms)	6.21	3.15	1.1	6.65
Power (W)	0.246	1.6	0.7	1.029
Power/neurons (mW/neurons)	0.12	7.27	0.0077	1.68
Energy (μj)	1,527	5,040	770	6,842

4.2.2 N-MNIST Dataset: a Neuromorphic Equivalent to MNIST

Because MNIST is not a natural neuromorphic dataset, we used another neuromorphic dataset called N-MNIST, which is comparable to MNIST in terms of classification task simplicity. In this section, we will present our result with the N-MNIST dataset. We will first introduce the experiments and then describe the different results.

Experimental Setup

N-MNIST, or Neuromorphic MNIST [140], is a natural neuromorphic equivalent of the MNIST dataset. N-MNIST was created by recording MNIST static images with Posch et al. DVS camera [151]. The camera was moved to simulate the motion of MNIST samples, thereby generating events from the DVS camera. The sequence of camera movements is decomposed into three different saccades, each approximately 100 ms in duration, resulting in a total sample duration of 300 ms. The output data are represented as timestamped events with X and Y coordinates, the date of spike emission, and the polarity of the event, which indicates whether the camera detected a positive or negative brightness change.
Before model training and evaluation, we transform the N-MNIST samples into frames, where events between the beginning and the end of each frame are accumulated, as represented in Figure 4.2. In our experiments, we separate each N-MNIST sample into 50 frames.

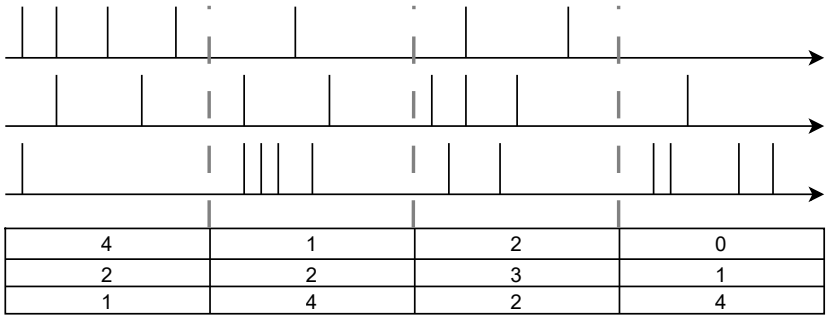

Figure 4.2 – Graphical representation of the n_time_bins transformation. The upper graph illustrates the spike trains from different input neurons, with time steps delimited by dashed gray lines. The lower table represents the transformation results, where the number of spikes in each train is accumulated for every frame.

We trained two different models for the N-MNIST classification task. Both models share the same FF-FC 34x34x2-128-10 topology.
The first model is described in Table 4.7. Both layers are implemented with the BLIF neuron model implemented with a parallel strategy. The second model is described in Table 4.8. To minimize the power consumption of the model, we decide to use the SLIF neuron model, more power efficient since this neuron model does not use multiplication. To maintain high precision of our model, we implement the output layer with the BLIF neuron model, which maintains an exponential membrane voltage decay. Both models emulated their neuron with the parallel emulation strategy to decrease the model latency with 16 bits to encode membrane voltage and 8 bits to encode synaptic weight.

Table 4.7 – Detailed module configurations of the architecture used to implement the 2-layer N-MNIST model. Both layers are implemented with the BLIF neuron model implemented with parallel emulation strategy.

Layer	N neuron	Neuron model	Strategy	Variable size	Synaptic Weight
Hidden Layer	128	BLIF	Parallel	16	8
Output Layer	10	BLIF	Parallel	16	8

Table 4.8 – Modules configuration details use to implement the 2-layer N-MNIST model. In contrast to the first model, and since modules on ModNEF are independent, we implement the hidden layer with the SLIF model to minimize the power consumption of our model. The output layer remains implemented with the BLIF neuron model.

Layer	N neuron	Neuron model	Strategy	Variable size	Synaptic Weight
Hidden Layer	128	SLIF	Parallel	16	8
Output Layer	10	BLIF	Parallel	16	8

Experimental Results

In this section, we will present the experimental results of the two implemented networks.
Table 4.9 summarizes the accuracy of the different models. As we can see, there is a slight drop in accuracy between the $Model_{BB}$ and the $Model_{SB}$, likely due to differences in the initial state and the varying neuron dynamics of the hidden layer.

Table 4.9 – Accuracy for 2-Layer N-MNIST models. Both models achieve similar accuracy, and the evaluation methods do not significantly impact the accuracy.

Evaluation Method	$Model_{BB}$ Accuracy (%)	$Model_{SB}$ Accuracy (%)
Full Precision	97.36	96.65
PTQ Simulation	97.39	96.48
FPGA	97.37	96.47

As it referred in Table 4.10 the two different models have similar LUT and FF consumption. However, $Model_{BB}$, due to its higher DSP consumption, requires more power than $Model_{SB}$.
As shown in Table 4.11, both models outperform the CPU and GPU in terms of energy consumption and computational time. However, the $Model_{SB}$ achieves inference with a similar computational duration but with lower energy consumption, decreasing from 37 μJ for $Model_{BB}$ to 16 μJ for $Model_{SB}$ due to lower consumption of DSP.
These experiments demonstrate the capability of ModNEF to process natural neuromorphic data in real-time with low-energy consumption. Moreover, we show that ModNEF can implement heterogeneous SNN topologies with different neuron models, resulting in varying impacts on power consumption and hardware resource usage,

Table 4.10 – Hardware metrics results for 2-Layer N-MNIST models. The $Model_{SB}$, using the SLIF neuron model for the hidden layer, shows reduced DSP and FF consumption, leading to a significant decrease in power consumption. BRAM consumption remains equal for both models, as the memory usage is determined by the number of parameters rather than the neuron model.

Metrics	$Model_{BB}$	$Model_{SB}$
Dynamic Power (mW)	473	153
Static Power (mW)	121	115
BRAM (KBit)	4158 ; 252 (87.50%)	4158 ; 252 (87.50%)
FF	5856 ; 355 (5.84%)	2795 ; 352 (2.96%)
LUT	8583 ; 503 (17.08%)	8954 ; 493 (17.76%)
LUTRAM	0 ; 96 (0.55%)	0 ; 96 (0.55%)
DSP	138 ; 0 (62.73%)	10 ; 0 (4.55%)

Table 4.11 – Shared metrics results for 2-Layer N-MNIST models. Both models on FPGA deliver comparable inference speed, but the $Model_{SB}$ model achieves significantly lower energy consumption due to lower power consumption. Overall, both models exhibit better energy consumption on FPGA compared to GPU and CPU targets.

Model	$Model_{BB}$			$Model_{SB}$		
Target	CPU	GPU	FPGA	CPU	GPU	FPGA
Inference Speed (ms)	0.39	0.24	0.062	0.30	0.26	0.06
Energy Consumption (mJ)	17.81	6.94	0.037	15.84	7.37	0.016

without significant differences in execution time and accuracy.

4.3 Recurrent Topology in ModNEF

In the previous sections, we presented two experiments using different datasets but with a common network topology, specifically the FF-FC network topology. Other network topologies can be implemented with ModNEF, such as recurrent networks. Recurrent networks appear to be good candidates for sequential data processing, such as audio or video data, due to their ability to maintain hidden states through recurrence [196, 214].
In this section, we will present two experiments based on recurrent networks. The first experiment, presented in Section 4.3.1, involves a local recurrent network using the SHD audio dataset. The second experiment, presented in Section 4.3.2, involves global recurrence with the DVS Gesture dataset.

4.3.1 Local Recurrence Architecture with SHD Dataset

The first experiment with a recurrent network topology involves a local recurrent network using the Spiking Heidelberg Digits (SHD) audio dataset. We will first describe the experimental setup, including a presentation of the dataset and a description of the dataset transformation and model.

Experimental Setup

The Spiking Heidelberg Digits (SHD) dataset [38] is an audio dataset composed of 10,000 spoken digits, ranging from 0 to 9, in both English and German. The spoken digits are recorded in high audio quality and then converted into spikes using a software cochlea simulator inspired by the work of Sieroka et al. [171], resulting in 700 output channels.
Our network model consists of a 700-200-20 topology with local recurrence on the hidden layer and output layer. Table 4.12 represents the layer configuration of our model. We used the SRLIF neuron model for both layers to maintain an exponential membrane potential decay while keeping a low-power architecture.
As with the N-MNIST dataset, we apply a frame transformation using the `n_time_bins` Tonic transformation. After experimentation, we determined that 75 frames provide the best transformation in terms of accuracy.

Experimental Results

In this section, we will present the different results obtained with the SHD dataset.

Table 4.12 – Detailed module configurations used to implement the 2-layer SHD model. To minimize the power consumption of our model, we choose to use the SRLIF neuron model to maintain exponential decay with low power consumption.

Layer	N neuron	Neuron model	Strategy	Variable size	Synaptic Weight
Hidden Layer	200	SRLIF	Parallel	16	8
Output Layer	20	SRLIF	Parallel	16	8

Table 4.13 summarizes the accuracy results obtained with our model. Due to the limited training set and the complexity of the classification task, it is difficult to achieve high accuracy with a 2-Layer network. However, we achieve a similar accuracy for a comparable network topology to other works with 71.4% [38], 77.5% [147], and 72.99 % [29].

Table 4.13 – Accuracy results with the SHD model. A 1.5% accuracy drop is observed on the FPGA target compared to the full precision evaluation, attributed to quantization errors that reduce synaptic weight precision.

Evaluation Method	Accuracy (%)
Full Precision	77.59
PTQ Simulation	76.87
FPGA	76.16

The use of recurrent modules increases power consumption due to higher constraints on the neuron model, especially to account for recurrent spikes in the membrane potential. For a similar number of neurons, the model consumes more power and hardware resources than the previous model. The usage of BRAM increases from the previous model, firstly because more BRAM is needed to store recurrent weights. Secondly, since the forward and recurrent BRAM memories are separated into two different memory components, the Vivado synthesis tool will separate these memories, increasing BRAM consumption due to the limited amount of BRAM on FPGAs. This results in the use of at least 18 Kbit of BRAM, even if less than 18 Kbit are required, thereby increasing memory and power consumption.

Table 4.14 – Hardware metrics results for the SHD model. The increase in BRAM and LUT usage, due to the recurrence, leads to a significant rise in dynamic power consumption.

Metrics	Results
Dynamic Power (mW)	296
Static Power (mW)	115
BRAM (KBit)	2592 ; 252 (56.43%)
FF	4110 ; 356 (4.20%)
LUT	18415 ; 549 (35.65%)
LUTRAM	0 ; 256 (1.47%)
DSP	0 ; 0 (0.00%)

We achieve real-time classification operations in less than 0.5 seconds, which is the shortest SHD sample duration. Thanks to low-power consumption and high inference speed, ModNEF run classification results with an energy consumption better than software-based inference from 530 to 895 times.

Table 4.15 – Shared metrics results for SHD model. FPGA inference achieves the fastest execution, while CPU and GPU evaluation run at comparable speeds. The FPGA target demonstrates the lowest energy consumption overall. A notable observation is the lower energy consumption of the GPU. Despite both targets delivering similar inference speed, we can conclude the power consumption of the GPU is less than the CPU power consumption during this experiment, which is counterintuitive.

Target	CPU	GPU	FPGA
Inference Speed (ms)	0.59	0.59	0.13
Energy Consumption (mJ)	27.41	15.41	0.056

4.3.2 Global-level Recurrent Architecture with DVS Gesture Dataset

In the previous experimentation, we used a local-level recurrency where neurons of a given layer are only connected to other neurons within the same layer. Another recurrence topology consists of a global-level recurrence where layers of the network are first connected in a forward manner, with additional connections between a given layer and other previous layers. This architecture allows capturing more temporal information [182, 196, 214].
In this section, we will demonstrate the capability of ModNEF to run this kind of network topology with the DVS Gesture dataset. We will first introduce the experimentation and then present the results of dataset inference.

Experimental Setup

DVS Gesture [9] is a neuromorphic dataset consisting of 1,342 hand and arm gestures recorded with an event-based DVS camera [109] with a 128x128 sensor resolution. This camera generates AER data with two different event polarities. The DVS Gesture dataset is separated into 11 classes: 10 classes for gesture recognition and one additional class for other gestures.
We applied two different transformations to the dataset samples. The first is a frame transformation using the `n_time_bins` method, with a sample split of 100 different frames. The second transformation is the downscaling of input data, illustrated in Figure 4.3, which shows the same portion of a frame with and without downscaling. Due to memory limitations of our FPGA, we need to downscale the input sample from 128x128x2 to 32x32x2 to avoid memory overconsumption.
The network model is presented in Figure 4.4. The output layer is connected to itself and to the hidden layer, creating a global recurrence connection.
The ModNEF architecture is schematized in Figure 4.5. To create the global recurrency, we used the `SRLif_Parallel` module along with the `Merger` and `Splitter` modules. For model training and architecture generation, we developed a new module called `GRShiftLIF`. This module allows the SNNTorch model to take input spikes from other layers, and the VHDL module generator can produce the correct memory file, which is slightly different from the traditional SRLIF neuron module.
A detailed model configuration is presented in Table 4.16. As for the previous, we use the SRLIF neuron model to maintain exponential decay while minimizing the power consumption. Since the `Merger` module will increase the model latency, we decide

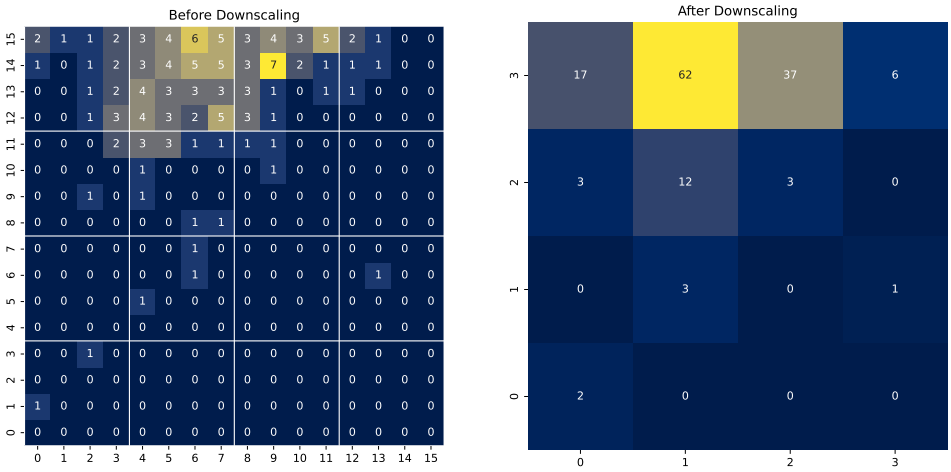

Figure 4.3 – Downscale representation of a region of a DVS Gesture sample. The left map shows the original data, illustrating the number of spikes per neuron. The right map represents the same region after downscale transformation, where spikes within each delineated area, marked by white lines on the original map, are accumulated and assigned to a shared neuron address. This transformation significantly reduces input data size but decreases information granularity.

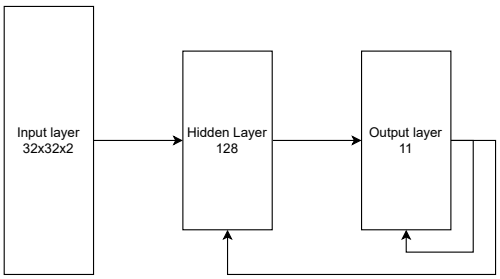

Figure 4.4 – DVS Gesture network topology. The network consists of an input layer connected to the hidden layer, which in turn connects the output layer. The output layer features both local recurrence, i.e., connected to itself, and global recurrence, with a feedback connection from the output layer to the hidden layer.

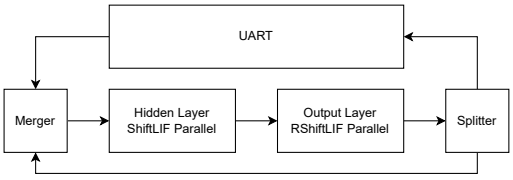

Figure 4.5 – DVS Gesture ModNEF architecture. The UART module connects to a `Merger` module, which establishes the global recurrence. The hidden layer is emulated by a `SRLif Parallel` module and is connected to the output layer module, implemented using a `RSRLif Parallel` module. The output data bus of the output module is split to enable the global recurrence.

Table 4.16 – Detailed module configurations for the DVS-Gesture model hardware implementation. Both layers are emulated with the SRLIF neuron model emulated in parallel with 16-bit computational variable size and 8 bits to encode synaptic weights.

Layer	N neuron	Neuron model	Strategy	Variable size	Synaptic Weight
Hidden Layer	128	SRLIF	Parallel	16	8
Output Layer	11	SRLIF	Parallel	16	8

to implement our neurons with the parallel emulation strategy to compensate for the latency increase.

Experimental Results

In this section, we will present the different results we obtain with our model.
Table 4.17 summarizes the accuracy of our model. With this simple topology, we achieve good accuracy with a smaller network model compared to other models, as shown in Table 4.18. Although ModNEF achieves lower accuracy than other models, our work demonstrates that similar accuracy can be achieved with a smaller network model by using global recurrence topology while maintaining low-power consumption.

Table 4.17 – Accuracy result of the DVS Gesture model. FPGA evaluation shows a 1% decrease in accuracy compared to the full-precision evaluation.

Evaluation Method	Accuracy (%)
Full Precision	84.37
PTQ Simulation	83.59
FPGA	83.20

Table 4.19 shows the hardware resource consumption of our architecture. As we can see, the BRAM consumption appears to be a limitation for our architecture, rather than LUT or FF resources. In the next section, we will investigate the limitations of our architecture.
Table 4.20 shows the inference speed and energy consumption comparison between CPU, GPU, and FPGA. As we can see, due to the complex architecture and the use of the `Merger`, we achieve classification tasks slower than the CPU and at a similar speed to the GPU. However, due to the low-power consumption of our device and the neuron model, the energy consumption is approximately 50 times better than CPU and GPU devices.

Table 4.18 – Model comparison for DVS Gesture. While ModNEF achieves lower accuracy than other models, our network is smaller, which may account for the reduced performance. However, this compact architecture enables the classification task to be executed with lower power consumption, compared to other FPGA-based architectures.

Design	ModNEF	Cheng et al. [34]	Liu et al. [110]	Frenkel et al. [54]
Topology	Global Recurrence	Feed Forward	Convolutional SNN	Local Recurrence
Model	2048-128-11	1024-512-11	144×144-p4-32c-p2-32c3-p2-512-256-11	256-256-10
Accuracy (%)	83.20	87.50	86.30	87.3
Target Device	XC7Z020-1CLG400C	Xilinx ZCU104	Zynq XA7Z020	ASIC
Power (mW)	280	720 [a]	420	42

[a] Unknown for DVS Gesture; the power consumption is based on the N-MNIST network with a slightly larger network topology.

Table 4.19 – Hardware metrics results for the DVS Gesture model. Due to the high BRAM consumption, the dynamic power represents the most important part of the power budget.

Metrics	Results
Dynamic Power (mW)	166
Static Power (mW)	115
BRAM (Kbit)	4212 ; 288 (89.29%)
FF	2623 ; 357 (2.80%)
LUT	10467 ; 515 (20.64%)
LUTRAM	0 ; 192 (1.10%)
DSP	0 ; 0 (0.00%)

Table 4.20 – Shared metrics result for DVS Gesture model. The FPGA target performs inference slower than other targets. This delay is primarily due to the high number of input spikes, which prolongs data transmission between the UART module and the hidden layer. Additionally, the sequential process induced by the `Merger` module further slows the inference. Nevertheless, due to the low-power consumption of the target, the FPGA energy consumption remains lower than other hardware targets.

Target	CPU	GPU	FPGA
Inference Speed (ms)	1.04	2.31	3.36
Energy Consumption (mJ)	50.72	56.19	0.94

4.4 Scalability and Limitations

In the previous network with DVS Gesture, we demonstrated the impact of hardware resource limitations on the network topology implementation possibilities.
In this section, we will demonstrate the scalability of ModNEF with the MNIST dataset to highlight the limitations of our architecture.
We used the MNIST dataset for two major reasons. The first reason is to reduce the size of the input layer compared to N-MNIST, thereby reducing the memory footprint of our model and allowing us to create deeper models. The second reason is the simplicity of the classification task, which can be easily trained and can readily support multiple-layer network topologies.
We trained six models with various network topologies. We varied two factors of our architecture: the number of layers and the number of neurons per layer. Table 4.21 summarizes all the models trained, referred to as Big MNIST Models (BMM).

Table 4.21 – Big MNIST Model summary. This table compares models trained with varying numbers of parameters, neurons, and layers.

Model name	Topology	Number of Neuron	Number of Synapses
BMM_2l	784-950-10	960	754,300
BMM_3l	784-392-196-10	594	386,120
BMM_3l2	784-1024-512-10	1,546	1,332,224
BMM_4l	784-588-392-196-10	1,186	770,280
BMM_4l2	784-256-128-64-10	456	242,304
BMM_5l	784-256-128-64-32-10	490	244,032

All previous models were implemented with default values, meaning all neuron emulation modules were implemented with the `Parallel` strategy, with signed 8-bit synaptic weights. We ran synthesis with this naïve method, and the results are shown in Table 4.22. As we can see, a naïve implementation does not work in our case due to the high number of neurons and synapses. Only BMM_4l2 and BMM_5l can be directly implemented due to their lower number of neurons and synapses, contrary to other models with more neurons and synapses.
In most cases, the BRAM and LUT are overused due to the high number of synapses and neurons, as shown in Figure 4.6, where there is a direct correlation between the number of synapses and resources, especially LUT and BRAM usage consumption.
The naïve method is limited due to the hardware limitations of our chip. To reduce the hardware footprint, we can apply three different methods:

1. Reduce the synaptic encoding bitwidth to reduce BRAM consumption.
2. Change emulation strategy from `Parallel` to `Sequential` to reduce computation resource usage, such as DSP, LUT and FF.
3. Reducing computational variable bitwidth is the last option to reduce hardware footprint. However, it can result in the introduction of signed overflow errors, resulting in a loss of accuracy.

Table 4.22 – Resource consumption, in percent, of big MNIST models with a naïve implementation method. Models with the highest number of parameters exceed BRAM capacity. The overconsumption of LUT arises because the large number of neurons exhausts all DSPs, thereby replaced by LUTs, resulting in overall overconsumption.

Ressource	BMM_2l	BMM_3l	BMM_3l2	BMM_4l	BMM_4l2	BMM_5l
BRAM	**152.86**	85.00	**245.36**	**172.86**	62.50	66.07
FF	33.12	20.70	53.22	40.80	17.59	17.14
LUT	**199.13**	**114.38**	**321.36**	**251.82**	79.66	92.53
LUTRAM	19.49	2.21	31.63	20.59	2.20	2.20
DSP	100.00	100.00	100.00	100.00	100.00	100.00

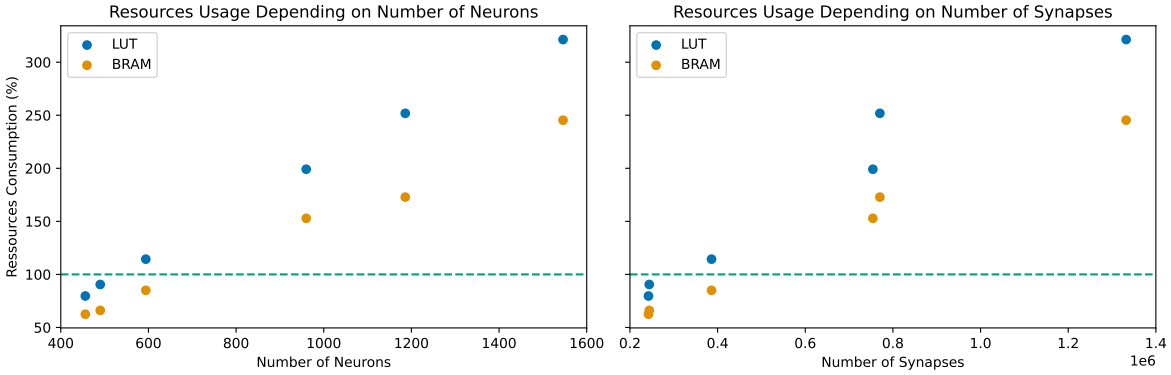

Figure 4.6 – LUT and BRAM usage, in percent, depending on the number of neurons on the left graph, and the number of synapses on the right graph. The green dashed line indicates the FPGA resource limits, while blue and orange marks represent the percentage of LUT and BRAM consumption, respectively.

Table 4.23 shows that by reducing the synaptic weight size and changing the neuron emulation strategy, we can fit models onto our board. However, for the most resource-intensive models, such as BMM_3l2 and BMM_4l, which have the most synapses and neurons, these modifications are not sufficient to implement the models on our FPGA board. To further reduce hardware resource consumption, we must also reduce the computational variable size to decrease the LUT consumption of our circuits.

We can thus highlight a first limitation of our architecture, which depends on the target FPGA board not having enough hardware resources to implement the model. However, this limitation is due to the experimental material and not the architecture design.

Table 4.24 shows the accuracy of our model. We can note a good model emulation with a low accuracy gap between software and hardware for most models. However, we observe an accuracy drop for two models. The model BMM_3l2 experienced two drops in accuracy: the first between full precision and PTQ accuracy due to low synaptic weight resolution, and the second between PTQ and FPGA due to low computational variable size, which can cause signed overflow errors during emulation, meaning the sign bit passed to 1 by dint of addition, or in contrast, passed to 0 by dint of

Table 4.23 – Model configuration and resource consumption of big MNIST models after applying reduction methods. By reducing synaptic bitwidth and variable size, BRAM consumption is minimized. The sequential emulation strategy reduces LUT usage, though it increases the FF utilization. These optimization methods also eliminate DSP usage for most models due to Vivado optimization.

Ressource	BMM_2l	BMM_3l	BMM_3l2	BMM_4l	BMM_4l2	BMM_5l
Synaptic bw (bits)	4	8	3	4	8	8
Strategy	Sequential	Sequential	Sequential	Sequential	Parallel	Sequential
Variable bw (bits)	8	16	8	8	16	16
BRAM	81.78	85.00	97.85	91.78	62.50	66.07
FF	22.44	27.78	35.93	27.73	17.60	22.97
LUT	49.44	54.10	78.08	59.45	79.65	45.15
LUTRAM	1.10	2.21	2.21	2.21	2.21	2.21
DSP	0.00	1.36	0.00	0.00	100.00	2.27

subtraction. The second model with an accuracy drop is BMM_2l, where the accuracy drop can be explained by the weight bitwidth reduction but also by the reduction of computational variable size, inducing signed overflow error. The impact of the reduction of computational variable bitwidth and the reduction of synaptic weight precision is not visible on the BMM_4l model, which is more robust to quantization than other models.

Table 4.24 – Accuracy results of Big MNIST models. For most models, accuracy remains nearly identical between full-precision and FPGA evaluation. The accuracy of the BMM_2l model decreased by 50% due to the reduction of variable size, potentially causing signed overflow on membrane voltage variables. The most pronounced accuracy loss occurs in the BMM_3l2 model, attributed to both reduced variable size and strict synaptic weight quantization.

Model	BMM_2l	BMM_3l	BMM_3l2	BMM_4l	BMM_4l2	BMM_5l
Synaptic bw (bits)	4	8	3	4	8	8
Strategy	Sequential	Sequential	Sequential	Sequential	Parallel	Sequential
Variable bw (bits)	8	16	8	8	16	16
Full Precision Accuracy (%)	96.65	96.39	96.01	96.86	96.58	96.45
PTQ Accuracy (%)	96.17	96.38	79.54	96.49	96.57	95.47
FPGA Accuracy (%)	44.63	95.93	9.45	91.66	96.88	95.35

We can conclude that our architecture demonstrates good scalability. As seen with the BMM_2l and BMM_3l2 models, hardware limitations are responsible for the accuracy drop due to lower weight resolution and signed overflow errors, which can occur if computational variable sizes are too small.

4.5 Impact of Design Choices on Architecture Performances

In the previous section, we present limitations of ModNEF and see the major limitation is due to hardware limitations of the target board, which can impact final accuracy. We highlight three methods to reduce the hardware footprint of our circuit:

1. Change emulation strategy from `Parallel` to `Sequential` that reduces LUT consumption.
2. Reduce synaptic weight bitwidth to reduce the BRAM consumption.
3. Reduce variable size to reduce computational complexity, thereby reducing computational circuit usage.

The first one is the emulation strategy, the second one is the synaptic weight bitwidth and the last one is the reduction of computational variable size.
In this section, we will present other possibilities to reduce the hardware footprint by analyzing diverse design choices. In addition to seeing the accuracy and the hardware footprint impact of these choices, we will also see the impact of these choices on inference speed.
First, in Section 4.5.1, we will study the impact of the neuron model on network implementation performances. In section 4.5.2 we will see with more details the impact of emulation strategy, and to finish in section 4.5.3 we will talk about the recurrence strategy, `Merger` + `Splitter` recurrence, or Internal Recurrence method on architecture performances.

4.5.1 Impact of the Neuron Model on Architecture Performances

In this section, we will describe the impact of the neuron model on architecture performance. We assume that you have read Section 3.2.1, which describes the different neuron models proposed in ModNEF. As we previously explained, since the IF model were integrated into ModNEF recently, the impact of the IF model is not studied in this section.
We trained three 2-Layer N-MNIST models with a FF 2312-128-10 architecture, each using a different neuron model. In this experiment, we used the `Parallel` emulation strategy to highlight the impact of the neuron model.
Table 4.25 summarizes the accuracy of different models. Because the membrane potential dynamics of BLIF and SRLIF neuron models are similar, it is expected that they achieve similar accuracy. However, the SLIF neuron model achieves a lower accuracy due to its linear decay and the difficulty in obtaining optimal neuron hyperparameters.
Table 4.26 compares the hardware resource consumption of different neuron models. The BLIF model consumes more dynamic power than other models due to the high consumption of DSP, a power-hungry resource, and the higher consumption of FF. The SRLIF model, although it consumes fewer hardware resources, consumes more power than the SLIF model. In all cases, memory consumption does not change because the number of synapses remains constant.
Table 4.27 summarizes the inference speed and energy consumption of models depending on the neuron model and target device. Since inference speed does not significantly depend on the neuron model (it can only vary if some neuron models emit fewer spikes than others), the inference speed remains stable across all models. In

Table 4.25 – Accuracy comparison depending on neuron model with N-MNIST dataset. The BLIF and SRLIF models deliver comparable accuracy, while the SLIF-based model achieves lower accuracy.

Evaluation Method	BLIF Accuracy (%)	SRLIF Accuracy (%)	SLIF Accuracy (%)
Full Precision	97.36	97.30	94.21
PTQ Simulation	97.39	97.29	94.09
FPGA	96.47	97.03	94.54

contrast, energy consumption is lower for SRLIF and SLIF models due to their lower power consumption, which is a result of not using DSP.
In terms of power consumption, the SRLIF neuron model is the best choice, with lower dynamic and static power consumption. Moreover, the SRLIF model offers better accuracy than the SLIF model, which has similar power consumption. However, in terms of resource consumption, the SLIF neuron model is slightly more efficient than other neuron models. Nevertheless, the SLIF neuron model is harder to train due to the linear membrane voltage decay. Although the BLIF neuron model is not efficient in terms of power consumption and hardware resource consumption, it is simpler to use than the SLIF neuron model and offers more hyperparameter possibilities, especially for membrane decay. A way to decrease power and hardware resource consumption for the BLIF model and other neuron models is to change the emulation strategy from `Parallel` to `Sequential`.

Table 4.26 – Hardware metrics comparison depending on neuron model for N-MNIST dataset. The BLIF-based exhbits higher power consumption due to its use of DSP resources. Although the SLIF-based model consumes fewer hardware resources overall, the SRLIF model demonstrates the lowest power consumption among the models. BRAM usage remains constant as the number of parameters is identical across all models.

Metrics	BLIF	SRLIF	SLIF
Dynamic Power (mW)	473	111	138
Static Power (mW)	121	114	115
BRAM (Kbit)	4158 ; 252 (87.50%)	4158 ; 252 (87.50%)	4158 ; 252 (87.50%)
FF	5952 ; 355 (5.93%)	2573 ; 352 (2.75%)	2617 ; 352 (2.79%)
LUT	8634 ; 504 (17.18%)	10447 ; 474 (20.53%)	9071 ; 493 (17.98%)
LUTRAM	0 ; 96 (0.55%)	0 ; 96 (0.55%)	0 ; 96 (0.55%)
DSP	138 ; 0 (62.73%)	0 ; 0 (0.00%)	0 ; 0 (0.00%)

4.5.2 Impact of the Emulation Strategy on Architecture Performances

As seen in Section 4.4, to reduce the hardware footprint of our larger models, we used the `Sequential` emulation unit instead of `Parallel`. In some cases, this was not suf-

Table 4.27 – Shared metrics comparison depending on neuron model on the N-MNIST dataset. The neuron model affects inference speed on CPU and GPU, whereas in FPGA target where all models execute inference at the same speed.

Model	BLIF			SRLIF			SLIF		
Target	CPU	GPU	FPGA	CPU	GPU	FPGA	CPU	GPU	FPGA
Inference Speed (ms)	0.26	0.24	0.06	0.34	0.29	0.06	0.31	0.28	0.06
Energy Consumption (mJ)	14.15	6.85	0.036	17.63	7.85	0.013	30.94	28.31	0.015

ficient, but it is a good starting point to reduce the hardware footprint of an SNN. In this section, we will study the impact of the choice of emulation strategy on architecture performance, especially in terms of power consumption, hardware footprint, and inference time. We assume you have already read Section 3.2.2 to clearly understand the major differences between the different architectures.
For this section, we used our previous model described in Section 4.2.2 with the N-MNIST dataset and the BLIF neuron model to highlight the differences between the architectures. To differentiate our architectures, we named our models with two letters, "P" or "S", corresponding to the emulation strategy of the first layer module and the second layer module, respectively. We thus have the following four different models:

- PP: Parallel-Parallel
- PS: Parallel-Sequential
- SP: Sequential-Parallel
- SS: Sequential-Sequential

Table 4.28 shows the accuracy of the initial model and all different model implementations. As we can see, the emulation strategy has no impact on accuracy because the synapses and neuron hyperparameters are the same, and the dataset is not affected by random operations.

Table 4.28 – Accuracy results for the N-MNIST dataset with different emulation strategies for each layer module. A drop in accuracy is observed between full-precision and FPGA evaluations. However, all FPGA-based evaluations maintain identical accuracy. As synaptic weights remain consistent across models, regardless of the emulation strategy used.

Evaluation Method	Accuracy (%)
Full Precision	97.36
PTQ Simulation	97.39
PP FPGA	97.3758
PS FPGA	97.3758
SP FPGA	97.3758
SS FPGA	97.3758

Table 4.29 summarizes the hardware resource consumption of our different architec-

tures. As we can see, due to the reduction in DSP usage, the power consumption decreases depending on the number of DSPs used. However, as seen in the previous table, the higher inference time means this reduction is not sufficient to improve energy consumption. The usage of the `Parallel` strategy increases LUT usage, whereas the `Sequential` module increases FF consumption due to additional internal memory.

Table 4.29 – Hardware metrics comparison between the different architectures depending on emulation strategy on the N-MNIST dataset. BRAM consumption remains consistent across strategies due to the identical implemented topology. Sequential emulation significantly reduces DSP usage, as a single hardware neuron emulates all neurons, leading to a substantial decrease in power consumption. Additionally, LUT usage is lower with the sequential strategy.

Metrics	PP	PS	SP	SS
Dynamic Power (mW)	472	423	160	156
Static Power (mW)	121	120	115	115
BRAM (Kbit)	4158 ; 252 (87.50%)	4158 ; 252 (87.50%)	4158 ; 252 (87.50%)	4158 ; 252 (87.50%)
FF	5952 ; 355 (5.93%)	5937 ; 355 (5.91%)	6748 ; 354 (6.67%)	6781 ; 354 (6.71%)
LUT	8634 ; 504 (17.18%)	8506 ; 504 (16.94%)	5843 ; 458 (11.84%)	5690 ; 459 (11.56%)
LUTRAM	0 ; 96 (0.55%)	0 ; 96 (0.55%)	0 ; 96 (0.55%)	0 ; 96 (0.55%)
DSP	138 ; 0 (62.73%)	129 ; 0 (58.64%)	11 ; 0 (5.00%)	2 ; 0 (0.91%)

Table 4.30 shows the inference speed and energy consumption of model implementations on different architectures. As expected, the sequential strategy increases the inference time. However, for the PS model, the inference time is not much higher than the PP model. This is because, in ModNEF, the inference time is dominated by the slowest neuron emulation model. If we consider the execution time formulas presented in Section 3.2.2, the execution time of the hidden layer with spike emission is $2 + 1 \cdot (2 + 128) = 132$, and the execution time of the second layer is $2 + 3 \cdot (10 + 1) = 35$. For the same reason, the inference time between SP and SS architectures is exactly the same because the execution time is dominated by the sequential module of the hidden layer.
However, in terms of energy consumption, due to the higher inference time, the energy consumption of SS and SP architectures is higher than that of PP and PS architectures. We can notice that the best model in terms of energy consumption is the PS architecture, due to the high speed of the largest layer and the low-power consumption of the second layer module.

4.5.3 Impact of the Recurrence Strategy on Architectures Performances

In this section, we will focus on the impact of recurrence strategies on architecture performance. We assume you have already read Sections 3.4 and 3.3.1 to clearly understand the different strategies and the presented results.
To highlight the differences between the two recurrence strategies, we trained a new model using the N-MNIST dataset with a 2312-128-10 topology, featuring recurrent hidden and output layers and SRLIF. The two implementations are referred to as:

Table 4.30 – Shared metrics comparison depending on emulation strategy. While the sequential emulation strategy increases inference speed, it also raises energy consumption despite reducing power usage. The PS architecture offers the best trade-off, maintaining high inference speed for the hidden layer and minimizing the power consumption.

Target	CPU	GPU	PP FPGA	PS FPGA	SP FPGA	SS FPGA
Inference Speed (ms)	0.26	0.23	0.0615	0.0617	0.1626	0.1626
Energy Consumption (mJ)	14.36	6.70	0.03651	0.03354	0.04473	0.04408

— `Merger+Splitter` (MS) recurrence strategy.
— Internal Recurrence (IR) recurrence strategy.

As we can see in Table 4.31, there is no impact of the recurrence strategy on accuracy, demonstrating the good functioning of both recurrence strategies.

Table 4.31 – Accuracy comparison between the two recurrence implementations. Since the recurrent implementation does not affect the weights encoding, both FPGA architectures achieve identical accuracy, similar to the full-precision accuracy.

Evaluation Method	Accuracy (%)
Full Precision	97.02
PTQ Simulation	96.79
MS FPGA	96.76
IR FPGA	96.76

Table 4.32 summarizes the hardware resource consumption of both implementations. The MS strategy consumes less power than the Internal Recurrence (IR) strategy. This power difference is mostly due to a significant gap in BRAM consumption, which arises from how internal memory is implemented. The Vivado synthesis tool can only use 36Kb or 18Kb BRAM blocks. Because the IR recurrence strategy implements two different internal memories, the recurrence memory, which is smaller than the forward memory, must be implemented with independent BRAM blocks. This increases BRAM consumption and, consequently, increases power consumption.
However, as we can see in Table 4.33, there is a significant impact on inference speed due to the Arbiter-Induced Serialization (AIS) mechanism introduced by the `Merger` module. Although the power consumption is higher for MS, the IR recurrence strategy achieves better energy performance due to its 16 times faster inference speed.

4.5.4 Impact of the UART Module on Architecture Performances

In this section, we will focus on the impact of the UART component on emulator performance. In previous sections, we did not discuss the UART impact on performance, firstly because all UART modules were configured in the same way throughout all previous experimentation, and secondly because, due to limitations we will explore in this section, the UART transmission module will soon be replaced by more efficient communication modules.

Table 4.32 – Hardware metrics comparison between the different recurrence implementations. Since the IR architecture utilizes additional BRAM to store recurrent weights, increasing resource usage and resulting in higher power consumption.

Metrics	MS	IR
Dynamic Power (mW)	173	209
Static Power (mW)	115	117
BRAM (Kbit)	4158 ; 252 (87.50%)	4734 ; 252 (98.93%)
FF	2623 ; 352 (2.80%)	2593 ; 352 (2.77%)
LUT	10475 ; 460 (20.55%)	11418 ; 474 (22.35%)
LUTRAM	0 ; 96 (0.55%)	0 ; 96 (0.55%)
DSP	0 ; 0 (0.00%)	0 ; 0 (0.00%)

Table 4.33 – Shared metrics comparison between the different recurrence implementations. The MS architecture exhibits slower inference compared to the IR architecture, primarly due to the sequential update operations induced by the `Merger` module, resulting in higher energy consumption.

Target	CPU	GPU	MS FPGA	IR FPGA
Inference Speed (ms)	0.39	0.32	1.01	0.061
Energy Consumption (mJ)	19.45	8.57	0.029	0.02

For this section, we assume you have read Section 3.3.2 to understand the role of the UART component in the architecture.
The UART data protocol was chosen due to its ease of implementation in both hardware and software. The VHDL UART transceiver was previously developed and utilized during a prior internship. Due to time constraints, we decided to retain UART and focus on developing more modules and conducting additional experiments rather than developing other communication modules.
In this section, we will compare the `UART_XStep_Timer` and `UART_Classifier_Timer`. We developed two additional drivers to gather more information about the duration of different operations:

- **Pre-Processing (PeP)**: Data preparation, transforming input spikes to AER format, and determining which emulation step can be sent to the FPGA.
- **Write Processing (WP)**: Packaging AER data into arrays of bytes according to the UART protocol and accessing the UART system.
- **Data Transmission (DT)**: Bytes transmission.
- **Inference Latency (IL)**: Onboard inference latency.
- **Data Reception (DR)**: Bytes reception.
- **Read Processing (RP)**: Reading system access to UART and unpacking to obtain output AER data.
- **Post-Processing (PoP)**: Updating output results.

PeP and PoP can be measured directly using the time library in Python. IL is directly

obtained from the received data. However, the DT and DL values cannot be directly measured and must be computed based on the length of the data packet and the baud rate. Figure 4.7 represents a schematic view of the UART pipeline and how we measure the duration of different operations.
For each UART module, we changed different UART hyperparameters. We first changed the baud rate from 921,600 (the highest supported baud rate on our board) to 460,800 and the size of the internal data read memory. If the driver detects that it cannot send all spikes of a sample, because the onboard module cannot fully store all generated spikes, it will send the spikes with multiple data transmissions. The memory depth was calculated based on the maximum number of spikes in the dataset, which is 7,868, and the average number of spikes per sample, which is 4,202. To highlight the importance of memory, we ran another architecture without enough memory to read all input spikes.
Table 4.34 summarizes the different architectures we evaluated. To minimize the impact of network consumption, we used the SRLIF N-MNIST network previously presented in Section 4.5.1 due to its low-power and low hardware resource consumption, which highlights the power consumption differences between different architectures. The architecture name is composed as follows: UX or UC for `Uart_XStep` or `Uart_Classifier`, respectively, the first digit of the baud rate (9 for 921,600, 4 for 460,800), and the first digit of the read memory (8 for 8,000 words, 3 for 3,000 word depth).

Table 4.34 – Summary of UART architectures with various baud rates for each UART module. The baud rate directly influences data transmission time, while the read memory size affects the number of communications required to transmit all spikes in the samples.

Parameters	UX98	UX48	UX93	UC98	UC48	UC93
Module type	XStep	XStep	XStep	Classifier	Classifier	Classifer
Baud rate	921,600	460,800	921,600	921,600	460,800	921,600
Read memory	8,000	8,000	3,000	8,000	8,000	3,000

Tables 4.35 summarize the operation durations per sample. The baud rate has a significant impact on experimentation duration by increasing the transmission and reception operations. However, in all cases, the longest UART operation is the data preparation operation. The inference speed is not impacted by UART and is negligible compared to other operations. The DT operation is only impacted by the baud rate but not by the architecture type, due to the same input data. However, because the data reception format is not the same between `Uart_XStep` and `Uart_Classifier`, the data reception duration is highly different.
If we consider the total UART operation duration for a single sample, the difference between `Uart_XStep` and `Uart_Classifier` can be considered negligible. However, due to the high number of samples, the total runtime is lower with `Uart_Classifier` than with `Uart_XStep`.
Figure 4.8 represents the execution time distribution across all different architectures. The most important parameter is the baud rate, as other operations have similar execution times.
This section represents a major limitation of ModNEF compared to CPU and GPU plat-

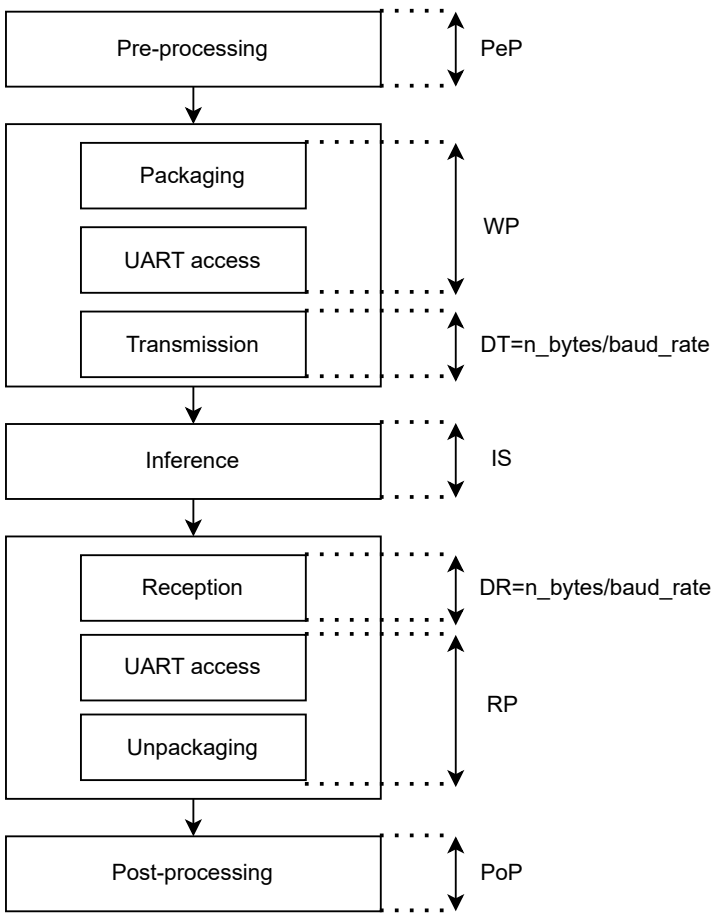

Figure 4.7 – Schematic representation of UART operation stage and how we measure execution time of different stages. The duration of each operation stage can be measured using Python or FPGA tools except for the data transmission and reception, which must be calculated based on the number of bytes and the baud rate.

Table 4.35 – Summary of UART architectures. The `UART_Classifier` appears as the fastest UART module due to its reduced output data volume, which minimizes the data transmission and the data unpackaging time. As expected, the baud rate significantly impacts the sample runtime.

Operation duration (s)	UC98	UC48	UC93	UX98	UX48	UX93
PeP	0.12	0.11	0.11	0.12	0.12	0.12
WP	0.043	0.086	0.015	0.043	0.086	0.043
DT	9.2e-3	18e-3	9.2e-3	9.2e-3	18e.4-3	9.2e-3
IS	6.18e-5	6.18e-5	6.18e-5	6.18e-5	6.16e-5	6.18e-5
DR	1.63e-5	3.26e-5	3.12e-5	12.17e-5	24.33e-5	12.17e-5
RP	0.058	0.11	0.10	0.065	0.12	0.066
PoP	1.72e-5	1.56e-5	2.89e-5	8.49e-5	8.58e-5	8.49e-5
Total	0.21	0.31	0.22	0.23	0.32	0.23

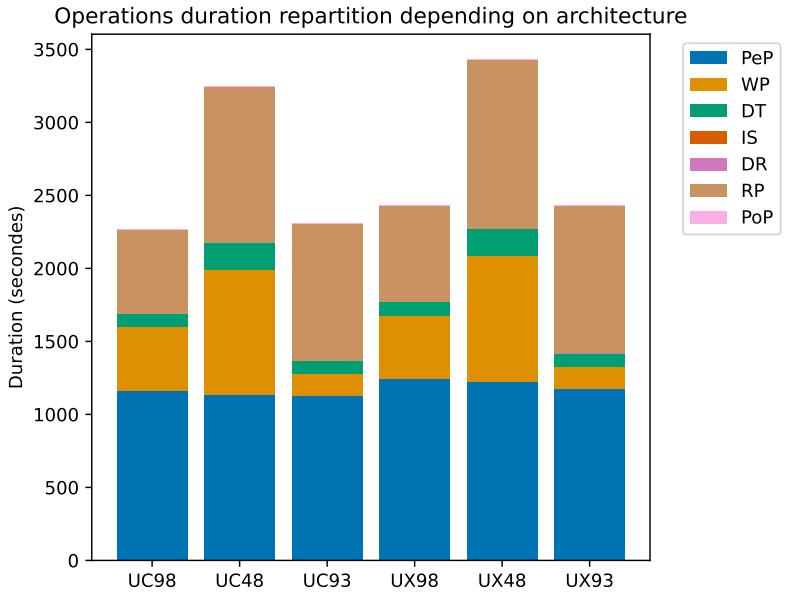

Figure 4.8 – UART operation duration repartition depends on architecture type. The majority of time budget is dominated by data preparation and UART access. In contrast, inference time and data unpackaging represent negligible portions of the overall operation duration.

forms. Even if the onboard execution is equal to or better than software execution, the total experimentation duration is much higher due to UART data transmission.

4.6 Conclusion

In this chapter, we prove the capability of our architecture to run SNN models with various network topologies.
We first demonstrate the good functioning of our emulator with the MNIST dataset and prove that ModNEF is a strong candidate for low-power and high inference speed compared to other works. We also demonstrate the capability of ModNEF to run Feed Forward Fully Connected (FF-FC) network topologies with neuromorphic datasets for simple classification tasks such as N-MNIST dataset. In addition to FF-FC topologies, ModNEF is able to run recurrent networks with local or global recurrence.
We also demonstrate the scalability of our architecture, which can run deep SNNs or SNNs with a high number of neurons and synapses. The major limitation, to the best of our knowledge, is the availability of hardware resources on the target board, rather than design limitations.
The other major limitation of our architecture is the use of the UART transmission component, which is used to transmit input spikes and receive output spikes. Due to the low communication speed of UART, the total experimentation duration is greatly increased. This limitation can be countered by using other communication protocols, such as Ethernet or PCI Express.
To reduce the hardware footprint of the architecture, we can use different neuron models, emulation strategies, and, for recurrent networks, different recurrence strategies. Each of these design choices has a different impact on hardware footprint, power consumption, inference speed, and energy consumption. Emulation strategy and recurrence strategy have no impact on accuracy, but the choice of neuron model can significantly affect accuracy due to the complexity of training with the SLIF neuron model or limited neuron decay hyperparameter values. If these methods are not sufficient, it is possible to reduce the bitwidth of computational variables and synaptic weights. Reducing the bitwidth of computational variables has a high impact on accuracy due to the introduction of signed overflow errors.
A better option can be to reduce the bitwidth of synaptic weights. In the next chapter, we will study the impact of quantization methods on emulator performance.

5 Quantization in ModNEF

In the previous chapter 4, particularly in Section 4.4, we highlighted several limitations of ModNEF, mostly due to FPGA hardware resource limitations. To bypass these limitations, we can firstly choose a bigger FPGA, but ModNEF proposes several tools to limit the hardware usage of FPGA resources by modifying the emulation algorithm from `Parallel` to `Sequential`, which decreases the LUT and FF consumption. However, the problem of BRAM consumption remains and highly limits the network topology, as shown in the previous chapter in Section 4.3.2. Memory restrictions force us to apply downsampling of DVS Gestures data or limit the network size, as we see in Section 4.4. To reduce memory consumption, it is necessary to reduce the encoding bitwidth of synaptic weights by applying a more restrictive **quantization** method.
Initially, quantization is a computational constraint due to the FPGA hardware target. To minimize computational resources, most FPGA emulators convert floating-point values and operations to integers to minimize the computational cost of their architecture [75]. Quantization is applied to neuron hyperparameters and synaptic weights and can highly impact model performances in terms of power consumption of computational resource usage. However, due to the introduction of quantization error, we can observe a drop in accuracy between full-precision, i.e., floating-point, models and quantized models. To minimize this accuracy drop while keeping advantage of quantization, it is crucial to dedicate a chapter about quantization methods and study their impact on ModNEF model performances.
In this chapter, we will focus on studying the impact of quantization in the ModNEF architecture depending on multiple aspects such as quantization method, dataset, neuron model, and quantization bitwidth target. Firstly, in Section 5.1, we will introduce a theoretical background for quantization. Then, in Section 5.2, we will present the different quantization methods proposed in the ModNEF framework. In Section 5.3, we will study the impact of Post-Training Quantization (PTQ) on model performances. Before concluding, in Section 5.4, we will present the implementation of the Quantization Aware Training (QAT) algorithm on the ModNEF framework.
As in the previous chapter, the IF neuron model, recently introduced in ModNEF, has not yet been explored in this chapter.

5.1 Theoretical Background of Quantization

In this section, we will present a theoretical background of quantization methods. Firstly, we will present basic concepts of quantization in Section 5.1.1. Then in Section 5.1.2, we will talk about the two different quantization schemes. To finish, in Section 5.1.3, we will focus on SNN quantization.

5.1.1 Introduction to Quantization

Quantization is a mathematical method to map an input set, generally composed of continuous values, to an output set smaller than the initial set and generally composed

of discontinuous values [59, 56]. Quantization, in addition to other methods such as distillation, compression, or pruning, appears as a suitable solution to decrease the computational complexity of NN inference and therefore reduce the computational time and energy consumption [132, 56].
We can categorize the quantization methods with two main criteria. First, we can distingue *uniform* and *non-uniform* quantization.
Figure 5.1 illustrates a uniform quantizer. The illustrated quantizer is defined by Equation 5.1, where x_q represents the quantized value, x_f is the floating-point value, and S is the scale factor, which is the quantizer step. In uniform quantization, each value from the initial set is quantized with the same quantizer resolution.

$$x_q = \lfloor \frac{x_f}{S} \rceil \tag{5.1}$$

In opposition to uniform quantization, Figure 5.2 represents a non-uniform quantizer. This plotted quantizer is defined by Equation 5.2, where $sign(x)$ is a function that returns the sign of x_f. In non-uniform quantization, the quantizer resolution is reduced for extreme values and increased for middle values, prioritizing medium values instead of extreme values. This quantization method can be interesting at first sight for NN quantization. In fact, in NN, weight distribution follows a normal distribution centered near to 0, and, in fact, non-uniform quantization demonstrates higher accuracy than uniform quantization [116]. However, because quantized values are not directly mapped to their float values, additional operations are necessary to find the original values. This can raise several problems for hardware implementation, such as in FPGA [116].

$$x_q = \lfloor sign(x_f) \cdot ln(1 + \frac{|x_f|}{S}) \rceil \tag{5.2}$$

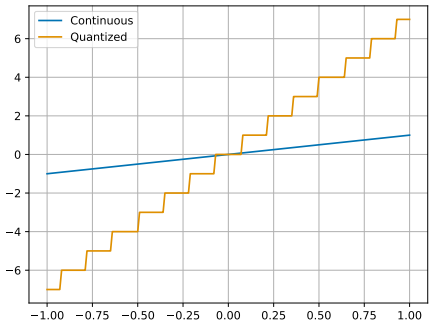

Figure 5.1 – Uniform quantization. The initial set, represented by the blue curve, is a continuous set in the range $[-1; 1]$. This input set is mapped to a quantized set, presented in the orange plot, composed of only 16 integer values.

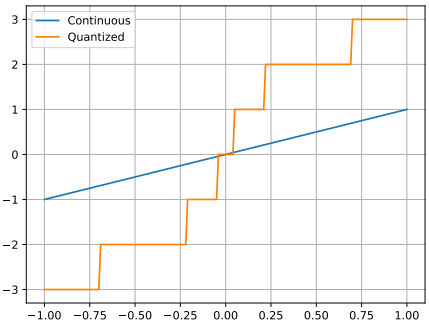

Figure 5.2 – Non-uniform quantization. The initial set in blue is mapped to the quantized set in orange, with a higher resolution for values in the range $[-0.25; 0.25]$ and a lower resolution for values outside this range.

The first criterion for quantization was uniform or non-uniform quantizers. The second criterion is the symmetry of the quantizer. Figures 5.3 and 5.4 illustrate the main difference between symmetric and asymmetric quantizers. A symmetric quantizer is

a quantizer where the initial set is symmetric: $[-a; a]$ will be mapped to the quantized set: $[-Q; Q]$. With an asymmetric quantizer, the initial set is asymmetric, meaning the lower and upper bounds are different: $[b; a]$, therefore, the quantized output set remains symmetric: $[-Q; Q]$.

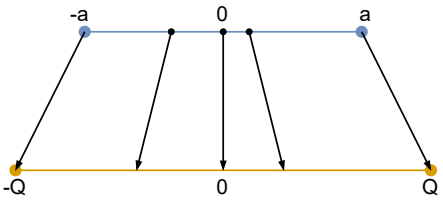

Figure 5.3 – Schematic representation of a Symmetric quantizer. The initial range is centered at 0 with the same number of values and mapped to the output set, also centered on 0.

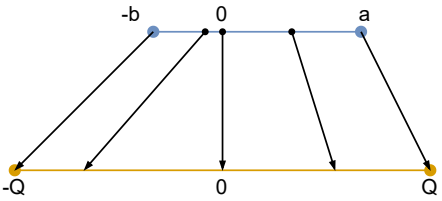

Figure 5.4 – Schematic representation of an asymmetric quantizer. In this case, the initial range is not centered at 0. However, the output set stays centered on 0.

As for non-uniform quantizers, asymmetric quantizers can raise several problems for hardware implementation. Due to asymmetry, a given quantized value x_q can represent a real value x_f; therefore, the quantized value $-x_q$ is not directly mapped to the real value $-x_f$ since a part of the input set is not quantized with the same quantization resolution as the other parts of the set.
Figure 5.5 summarizes the two criteria of the quantizer. Quantizers are sorted by column depending on whether they are symmetric or asymmetric and in rows depending on whether they are uniform or not.

5.1.2 Post-Training Quantization and Quantization Aware Training

Independently of the quantization function, there are different quantization strategies. We call quantization strategy the moment of when and how we apply the quantization function to the initial values. We can denote two main quantization strategies in NN quantization:

— **Post-Training Quantization (PTQ)**: In PTQ, the model is trained using standard full-precision training methods. After the training phase, the synaptic weights and neuron hyperparameters are quantized to the target bitwidth. This method is simple and does not necessitate a retraining phase but can lead to a drop in accuracy, especially for low-bit resolution. Several works [153, 183, 107] have proposed calibration or fine-tuning strategies and several methods to mitigate the accuracy degradation.

— **Quantization Aware Training (QAT)**: In QAT, the quantization process is considered during the training, or retraining, process [56]. A model can be trained directly with QAT, or an existing model previously trained with full precision can be fine-tuned with a retraining phase with QAT. The way in which the quantization is considered can depend on the learning methods. If QAT is applied to gradient-based learning methods, the QAT algorithm can be described as follows: during the forward pass, the synaptic weights are quantized; thereby, during the backward pass, the synaptic weights and the gradient calculation remain

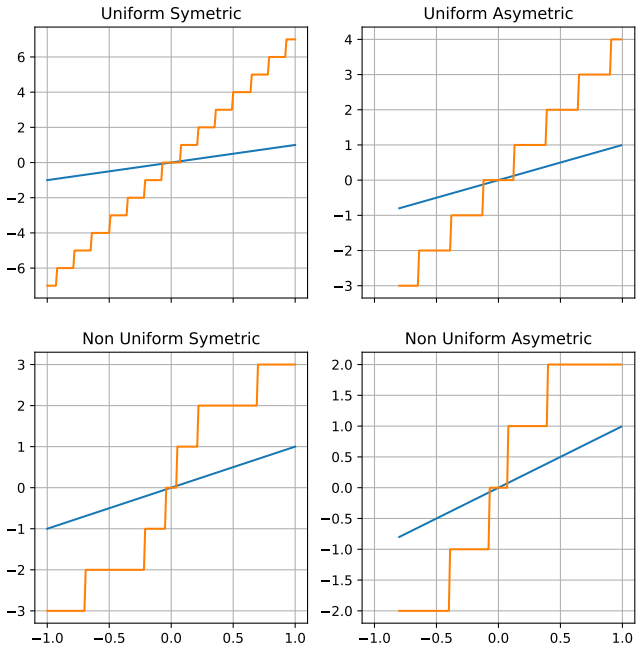

Figure 5.5 – Quantizer comparison depending on criteria. Columns compare *Symmetric* and *Asymmetric* quantizers, and rows compare *Uniform* and *Non-Uniform* quantizers.

in full-precision [56]. The model will thus converge into a state where the optimal model stores quantized weights and so mitigate the accuracy drop by reducing the quantization error and the quantization noise. Even if many works focus on QAT applied to gradient-based learning methods [56, 116, 112], the QAT methods can be applied to other learning methods such as STDP [79]. A major limitation of the QAT methods is the computational cost to retrain a model. Even if we use the QAT methods from the beginning of the training, the computational cost of synaptic weight quantization must be considered.

To choose between QAT and PTQ, we should ask ourselves if, firstly, the drop of accuracy induced by the PTQ is acceptable and, secondly, the additional computational cost of QAT is acceptable [132]. Depending on the quantization target bitwidth and the application, the answers to these questions can change; for example, below 4-bit resolution, the impact of PTQ becomes too important and the QAT becomes essential [132].

5.1.3 Quantization in Spiking Neural Networks

We previously introduced generality about quantization. In this section, we will present quantization in the particular case of SNN.

As for ANN, the quantization of synaptic weights is a crucial part of SNN quantization. The quantization of synaptic weights will introduce a quantization error due to lower resolution. However, contrary to ANN, where inference is not executed through time,

the quantization error introduced in SNN weights can be accumulated into the membrane voltage through time, which can result in a higher sensitivity to quantization.
In addition, SNNs uses internal neuron variables to represent membrane voltage and input current, for example. These internal neuron variables must be stored in memory, and the computational resources will be more expensive for high bitwidth [153, 183].
In addition to synaptic weights and internal variables, the neuron hyperparameters must also be quantized. A major problem of hyperparameter quantization is that the quantization error can change the neuron dynamics with a different threshold or membrane leakage value [107].
We can denote several libraries that propose quantization schemes that take the specificities of SNN to create novel quantization schemes.
NeuronQuant [107] proposes a mixed-precision PTQ scheme. NeuronQuant first proposes to quantize synaptic weights to minimize the quantization gap between the synaptic weights and the threshold value. Moreover, NeuronQuant proposes to compute the target bitwidth dynamically depending on a score that indicates the quantization impact on a neuron. This results in a good tradeoff between power consumption, which decreases thanks to the quantization, and accuracy.
SQUAT [183] proposes to separate synaptic weight and neuron state, i.e., internal variables, quantization. Synaptic weights are trained in full-precision or with the QAT algorithm, and, in a separate way, the internal state of neurons is quantized.
The Q-SpiNN framework [153] proposes to run quantization exploration to select the best quantization scheme, depending on quantization function and bitwidth, and select the best model/quantization pair.

5.2 Quantization Method in ModNEF

The ModNEF framework proposes three different quantization methods, all of which are derived from a quantizer base class :

1. Fixed-Point Quantizer (FPQ) presented in Section 5.2.1
2. Dynamic Scale Factor Quantizer (DSFQ) presented in Section 5.2.2
3. Min-Max Quantizer (MMQ) presented in Section 5.2.3

The base class applies quantization using Equation 5.3, where Q denotes the quantization function specific to the quantizer. Beyond quantization, the base class has the capability to apply the clamping function and can also simulate quantization impact by unquantizing quantized values. The quantization simulation is illustrated by Equation 5.4, where Q^{-1} is the inverse function of the quantization function. This quantization simulation feature enables application of quantization in software SNNTorch-based simulations during the inference phase to predict onboard results or during the training phase to do QAT.

$$x_{int} = \lfloor Q(x_f) \rceil \tag{5.3}$$

$$x_s = Q^{-1}(clamp(\lfloor Q(x_f) \rceil)) \tag{5.4}$$

It is important to note that in ModNEF, the quantizers are parameterized based on the synaptic weights, meaning their scaling factors and limits are determined using the weight distribution and target bitwidth. However, these same quantizer parameters

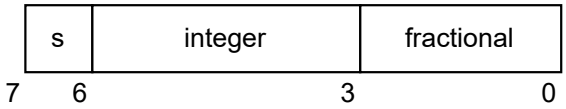

Figure 5.6 – Signed fixed-point representation. In this example, the MSB is the sign flag; the bits between 6 and 3 are used to represent the integer parts, and the bits below 3 will be used to encode the fractional parts. In this example, bit 3 is referred to as the point position.

are also applied to computational variables such as neuron hyperparameters, input current, and membrane voltage. While these variables typically use a higher bitwidth than synaptic weights, applying the same quantization parameters does not lead to overflow issues since the wider representation range can accommodate the quantized values without saturation.

5.2.1 Fixed-Point Quantizer

The first implemented quantizer is the widely recognized Fixed-Point Quantizer (FPQ) [64]. The fixed-point representation is a binary decimal format where a portion of the binary word is allocated to encode the integer part of the number, and the remaining bits are used to encode the fractional part, as illustrated in Figure 5.6. The bit that separates the integer and the fractional part is known as the fixed-point position.
The quantization function is described by Equation 5.5, where fp represents the fixed point position. The fp parameter can either be manually fixed by the user or can be determined during the initialization phase. During initialization, the integer part size is determined by the integer part of synaptic weights, while the remaining bits are allocated for the fractional part.

$$x_{int} = x_f \cdot 2^{fp} \tag{5.5}$$

Due to the simplicity of the quantization function, all neuron models can support this quantization method. However, it is necessary to provide the fixed point position to the BLIF neuron model. In fact, due to multiplication, the two scale factors (that of the membrane voltage variable and that of the β hyperparameter) are accumulated, as is shown in Equation 5.6. This scale factor accumulation can be addressed by specifying the fixed point position and eliminating the additional scale factor with a simple shift register, since the scale factor is a power of two.

$$x_{int} \cdot y_{int} = x_f 2^{fp} \cdot y_f 2^{fp} = x_f \cdot y_f \cdot 2^{2fp} \tag{5.6}$$

5.2.2 Dynamic Scale Factor Quantizer

Fixed point quantization has a significant drawback: it may not utilize all possible values provided by the binary word. For example, to encode values within the range $[-1.5, 1.5]$, the FPQ quantizer must be configured to encode values in $[-2.0, 1.98]$, and several values remain unused, which reduces the precision of the quantizer. To address this issue, the scale factor, equal to 2^{fp} for fixed-point quantizer can be determined dynamically based on the range of input data, synaptic weights in our case [117].

The quantization function is presented in Equation 5.7 and is very similar to the fixed-point quantizer. However, contrary to the fixed-point quantizer, the scale factor s_x is determined by Equation 5.8, which is dynamically calculated depending on the maximum value of input data.

$$x_{int} = \frac{x_f}{s_X} \tag{5.7}$$

$$s_X = \frac{max(|x|)}{2^{b-sign}} \tag{5.8}$$

This quantization method reduces quantization error by dynamically computing the extreme values of quantized data. This can be an advantage, but this method can also give too much importance to extreme, therefore rare, values. However, a major drawback of this quantization method is its inability to perform multiplication.
As we explained with Equation 5.6, any multiplication between quantized values results in an accumulation of their scale factor. While the scale factor of FPQ quantizer is a power of two, the extra scale factor can be removed by a simple shift register. In contrast, the scale factor of DSFQ is not a power of two. After a multiplication, correcting the accumulation of scale factors required additional costly operations. For this reason, DSFQ is not recommended or neuron models involving multiplications.

5.2.3 Min-Max Quantizer

The last quantization method is the Min-Max Quantizer (MMQ) [86], described by Equation 5.9. This method is based on the Min-Max scaling technique, where the initial range is $[x_{min}; w_{max}]$ and the target range is $[b_{min}; b_{max}]$. Here, b_{max} and b_{min} represent the maximum and minimum values of the integer data, respectively, which can vary depending on the bitwidth and whether the data are signed or unsigned. In most cases, we set $b_{min} = -b_{max}$ and $x_{min} = -x_{max} = \max(|x|)$ to maintain symmetry in the quantization method and represent synaptic weights without distortion.

$$x_{int} = \frac{(x_f - x_{min}) \cdot (b_{max} - b_{min})}{w_{max} - w_{min}} + b_{min} \tag{5.9}$$

Once again, this quantization method does not support multiplication due to addition and subtraction and the non-trival, i.e., not equal to a power of two, scale factor.

5.2.4 Conclusion

In this section, we present the three different quantization methods proposed in ModNEF :

1. Fixed-Point Quantizer (FPQ)
2. Dynamic Scale Factor Quantizer (DSFQ)
3. Min-Max Quantizer (MMQ)

The main advantage of FPQ method is its support for multiplication. As we explained in Sections 5.2.1 and 5.2.2, the scale factors used to quantize value are accumulated during multiplication. In FPQ, since the scale factor is a power of 2, the additional factor can be efficiently removed with a simple shift register. However, in DSFQ, the scale factor is not a power of 2 and to remove the additional scale factor, it is necessary

Table 5.1 – Quantizer method supported by each neuron model. A "V" indicates support for the quantizer method, while "X" denotes lack of support. The FPQ quantizer is universally supported across all neuron models. Other quantizers are incompatible with the BLIF neuron model due to the multiplication, which not supported by other quantizers.

Quantizer	BLIF	SLIF	SRLIF
FPQ	V	V	V
DSFQ	X	V	V
MMQ	X	V	V

to divide the result of multiplication, which is not efficient for FPGA implementation. In Min-Max Quantizer (MMQ), the result of multiplication is more complicated than a simple accumulation of scale factors, and as for DSFQ, this quantizer does not support multiplication.
Table 5.1 references the supported quantizer for each neuron model. Because the BLIF model uses multiplication, only FPQ is supported. Other neurons can support all quantizers because in SRLIF, the multiplication is replaced with a shift register and the scale factor is not accumulated. And in the SLIF model, the exponential membrane decay is replaced by linear membrane decay.

5.3 Post Training Quantization Impact on Architecture Performances

In the previous section, we presented the different quantizers provided in ModNEF. In this section, we will study the impact of different quantizers at different levels. Firstly, in Section 5.3.1, we will focus on the impact of quantizers at a neuron level by studying the impact of quantizers and bitwidth on the membrane voltage dynamic and on spike emission. Then, in Section 5.3.2, we will zoom out and investigate the impact of quantization at a network level using Post-Training Quantization (PTQ).

5.3.1 Quantization Impact at a Neuron Level

In this section, we will examine the impact of quantization methods at a neuron level depending on the quantization methods, target bitwidth and neuron models. To achieve this, we define a single neuron with a single synaptic weight fixed at 1, and we set the threshold voltage at 5.0 for all neuron models.

Metrics Definition

To compare quantization methods, we will evaluate their performances based on bitwidth and on the neuron model. To achieve this comparison, we will simulate a single neuron with a single synaptic weight equal to 1, which can be quantized without quantization error. We determine the quantization error with three different errors.
The first metric is the Mean Squared Error (MSE) defined in Equation 5.10, where V_i is full precision, i.e., floating point, membrane voltage of step i and $\hat{V}_i$ is quantized membrane voltage at step i.

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(V_i - \hat{V}_i)^2 \tag{5.10}$$

The MSE error measures the quantization impact on neuron dynamics but does not capture its effect on spike emission. To highlight the impact of the quantization method on spike emission, we introduce two different metrics. The first one is Mean Spike Time Error (MSTE), defined in Equation 5.11, where t_i^f is the spike emission date of the i-th spike during the floating-point simulation and t_i^q the spike emission date during the quantized simulation. However, in some specific cases that we will examine later, it is possible one of the simulations can emit more spikes than the others. To address this scenario, we define the Spike Counter Error (SCE) presented by Equation 5.12.

$$MSTE = \frac{1}{N_{match}}\sum_{i=1}^{N_{match}}(|t_i^f - t_i^q|) \tag{5.11}$$

$$SCE = |N^f - N^q| \tag{5.12}$$

Impact Depending on Neuron Model

In this section, we will examine the impact of quantization depending on the neuron model. In this section, we will exclusively use FPQ because it is the only quantization method supported by all neuron models. We will fix the decimal point position at position $bw - 1$, where bw is the target bitwidth.
Figure 5.7 illustrates the different error metrics depending on the neuron model for the FPQ quantizer. In the first plot, which represents the MSE error, we observe the SLIF membrane dynamic is the most sensitive, with an average MSE of 1.99 instead of 1.25 for the BLIF neuron model. However, the SLIF MSE error is very high for low resolution and approaches zero for higher resolution, contrary to the BLIF model, which got a more linear MSE dynamic.
However, in SNN, the timing of spike emissions is more important than membrane voltage dynamics. In the middle plot, we can observe that the BLIF neuron model has a high MSTE error primarily due to a higher number of additional spikes emitted by the BLIF model compared to other neuron models. This increase in spike emissions can be attributed to the quantization error introduced in the neuron hyperparameters and, more specifically, the quantization of membrane decay parameters.
Figure 5.8 illustrates the evolution of membrane leakage parameters. The left plot represents the β parameter of the BLIF neuron model, and the right plot shows the V_{leak} parameter of the SLIF neuron. As observed, at low bitwidth resolution, the difference between the initial value (in red) and the quantized value (in blue) is very high. For 3-bit resolution, the membrane decay is equal to 1 for β and 0 for V_{leak} meaning the neuron models are turned to an IF model. In contrast to these two neuron models, the SRLIF model implements the membrane decay with a shift register, where the shift value is not quantized. This implementation makes the SRLIF model more robust against quantization effects.
In this section, we studied the quantization impact on neuron models with the FPQ quantizer. Depending on the metric we consider, the SLIF and BLIF neuron models

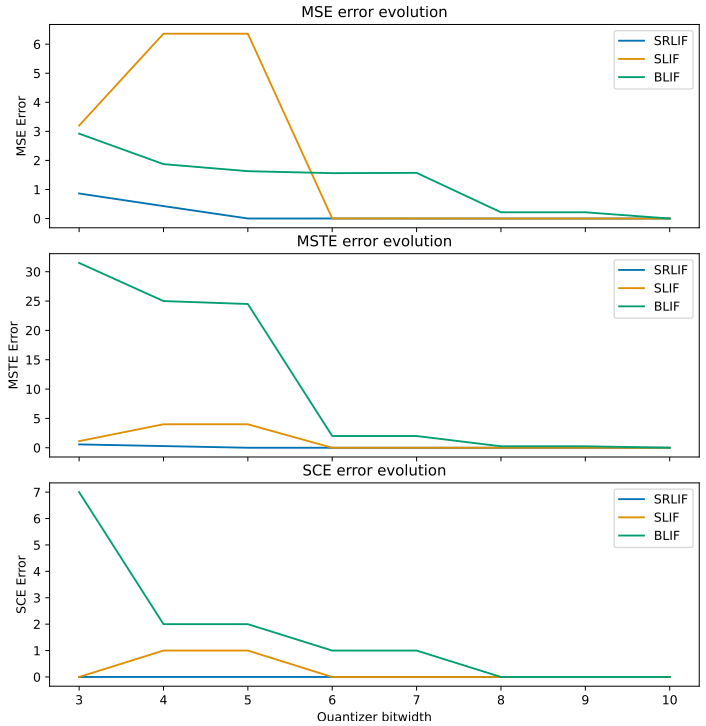

Figure 5.7 – Quantization error metrics for a single neuron with the FPQ quantization method. The graphs represent the error evolution depending on the bitwidth and the neuron models. The upper graph represents the MSE error, the middle graph illustrates the MSTE error, and the lower graph shows the SCE error.

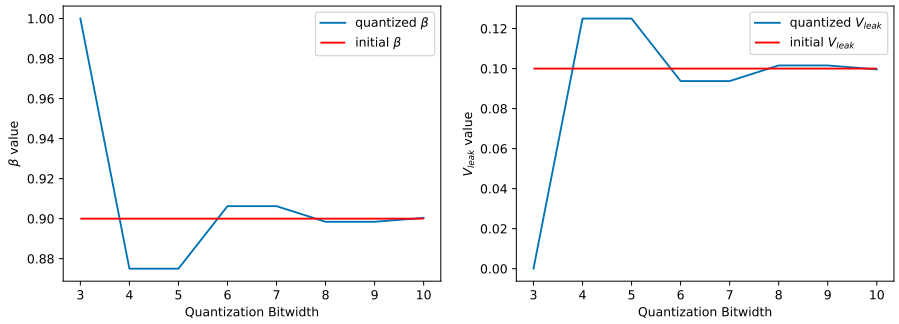

Figure 5.8 – Evolution of leakage parameters depending on the quantizer target bitwidth. The left plot represents the evolution of the β parameter, and the right plot illustrates the V_{leak} parameter. The red line is the full precision value, and the blue curve is the quantized value. In both graphs, the quantized value remains similar to the initial value for high resolution but tends to negligible values, 1 for β and 0 for V_{leak}, resulting in an IF neuron model.

are more sensitive to quantization, mostly due to the quantization error of the quantization of membrane decay hyperparameters. This initial study was conducted exclusively with FPQ quantizer. In the next section, we will compare these different metrics depending on the quantization method.

Impact Depending on Quantization Method

In this section, we will investigate the impact of quantization based on the quantization method. Our analysis in this section will only focus on SLIF and SRLIF neuron models. As the BLIF neuron model only supports the FPQ method, was thus examined in the previous section.
Figure 5.9 illustrates the different error metrics of SRLIF and SLIF neuron models. The first notable observation from the upper graphs in both figures is the absence of differences between FPQ and DSFQ. We can explain this because, in this specific case, because the synaptic weight is equal to 1, the scale factor of DSFQ $s_X = \frac{1}{2^{b-1}}$. According to Equation 5.7 $x_{int} = \frac{x_f}{s_X} = x_f \cdot 2^{b-1}$ and so the DSFQ mimics the FPQ method.
Beyond this specificity, we observe that the best quantization method varies depending on the neuron model. For SRLIF model, the MMQ quantizer is better than DSFQ and FPQ in opposition to the SLIF neuron model, where FPQ and DSFQ are better than MMQ.

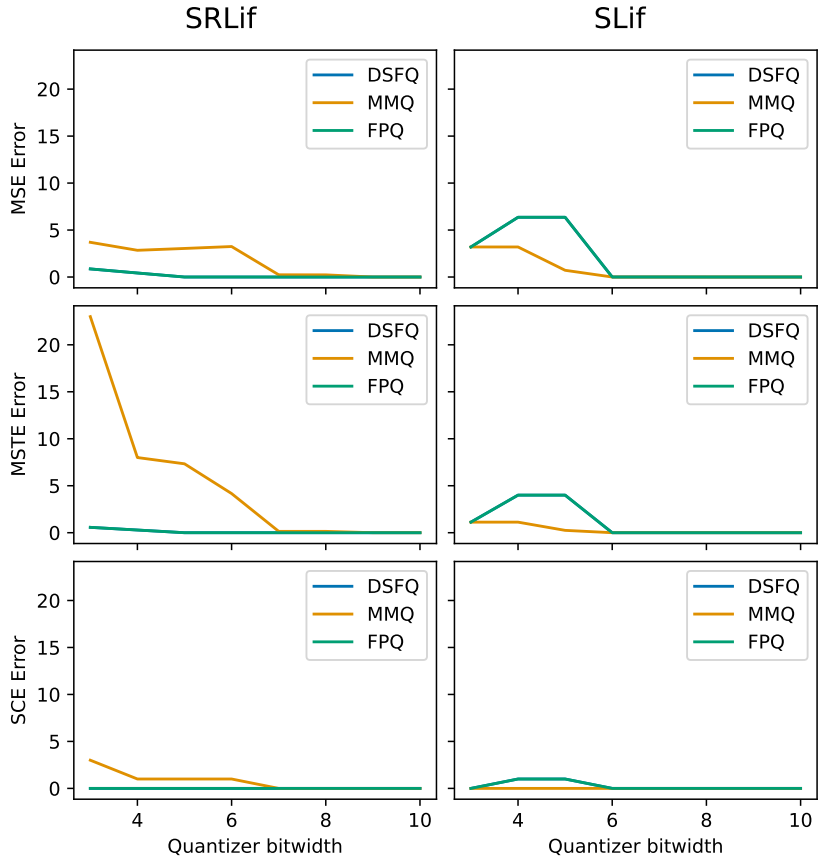

Figure 5.9 – Error metrics for the SRLIF neuron model in left and the SLIF neuron model in right depending on the quantizer target bitwidth. The upper graph represents the MSE error, the middle graph illustrates the MSTE error, and the lower graph shows the SCE error. Because, in this specific case for a single neuron with a single synaptic weight fixed at 1, the scale factor of the DSFQ and the FPQ are equal, resulting in the same curves.

However, the most critical aspect in SNN is the spike emission timing represented in

the middle plot of both figures. As for MSE metrics, the MMQ quantizer is less precise than DSFQ and FPQ for the SRLIF neuron model. For SLIF neuron model, DSFQ and FPQ appear to be more accurate.

5.3.2 Quantization Impact at a Network Level

In this section, we will examine the impact of PTQ at a network level. Our analysis will focus on the relative error described by Equation 5.13. Furthermore, we will limit our comparison to software quantization simulation, i.e., PTQ, to avoid long runs on FPGA. We previously observed in Chapter 4, the PTQ software results are accurate with FPGA results.

$$E = \frac{|Acc_f - Acc_q|}{Acc_f} \cdot 100 \tag{5.13}$$

We also introduce two new notations for Fixed-Point Quantizer (FPQ). As explained in Section 5.2.1, the fixed-point position can be manually set or can be automatically determined during quantization initialization. As we will observe in the following sections, the method by which the fixed-point position is determined can be crucial. We differentiate between Static Fixed-Point Quantizer (SFPQ) where the point position is fixed by the user (equal to $bw - 1$ in all cases in this study), and Dynamic Fixed-Point Quantizer (DFPQ) where the point position is dynamically determined.
To enable quantization simulation, which involves applying quantization and then dequantizing values, we use an override of the `eval` function used in PyTorch models. This enhanced method accepts an additional argument to enable the quantization simulation, as demonstrated with this line of code: `model.eval(quant=True)`.
Firstly, we will focus our studies on a specific use case with N-MNIST dataset in Section 5.3.2. Following by an investigation into the impact of quantization on hardware resource usage in Section 5.3.2. Before concluding, Section 5.3.2 will be dedicated to analyzing other parameters that can significantly affect the impact of quantization on accuracy.

Quantization Impact on Accuracy: N-MNIST Use Case

In this section, we trained three different networks, one for each neuron model, using the network topology: $2312 - 10$ trained during 10 epochs to highlight quantization impact.
Figure 5.10 illustrates the quantization impact, illustrated by the relative error, on different networks, categorized by neuron model in rows and by quantizer method by columns.
The first notable observation is the low quantization error when quantization bitwidth is at least 5 bits. An exception to this observation is the MMQ method, where the relative error becomes negligible after 6 bits.
Another key observation is the negative impact of fixed-point positioning and, more generally, the loss of precision in quantizers based on minimal and maximal values, such as DSFQ and MMQ. Figure 5.11 illustrates the weight distribution after quantization. The blue dotted line represents the extreme values of the original weights, while the red dashed line indicates the extreme values of the quantized weights. The SFPQ quantizer quantized fewer synaptic weights to 0 compared to other methods, as it

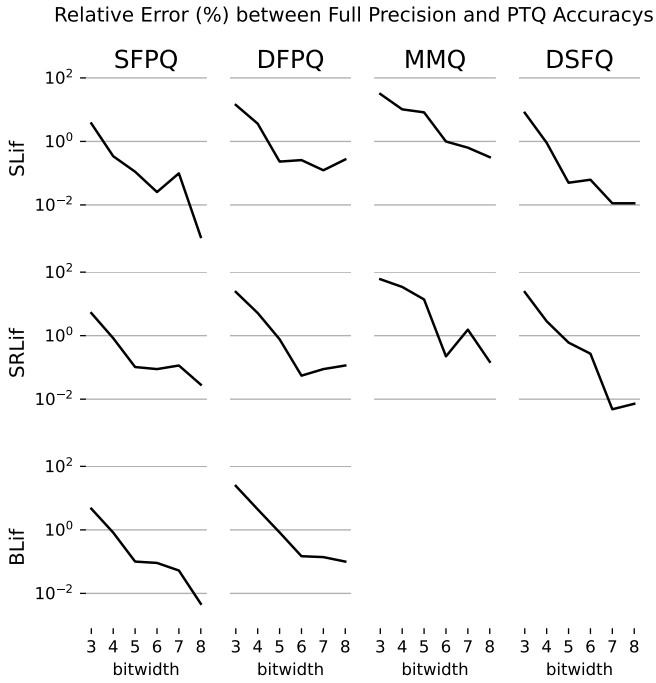

Figure 5.10 – Relative error evolution in percents with a logarithimic scale, by quantizer bitwidth, neuron model, and quantizer method. The figure is organized by neuron model in rows and by quantizer method in columns. The relative error decreases when the quantizer resolution. The MMQ quantizer exhibits lower precision compared to other quantizers. Among the two fixed-point quantizers, the SFPQ quantizer demonstrates greater robustness than the DFPQ. Although the DSFQ is more sensitive than both fixed-point quantizers at low resulotion, the DSFQ achieves higher precision at high resolution.

does not assign quantization levels to extreme and rare values. In contrast, quantizers whose parameters are determined based on minimal and maximal values, encode these rare values, resulting in a higher number of synaptic weights being quantized to zero.

If we consider the neuron model, the SLIF neuron model appears to be the least sensitive to quantization, with lower relative error than other neuron models. This hypothesis, based on Figure 5.10, is corroborated by Figure 5.12, where the SLIF neuron model achieves superior performance relative to other neuron models. This can be explained by, considering the previous Section 5.3.1 where the SLIF neuron model demonstrated a high MSE error in membrane dynamics but low MSTE error, resulting in higher resistance to quantization at a network model because information is transmitted through spikes and their emission timing. However, it is important to note that we trained our model with low membrane decay parameters $V_{leak} = 0.01$ which makes the SLIF model behave similarly to the IF neuron model. Additional experiments are necessary to determine whether the SLIF neuron model is inherently more resistant to quantization or if we have studied a particular case where the membrane dynamic is less sensitive due to the low value of membrane decay.

To select an appropriate quantization function, two main aspects mus be considered. First, the neuron model plays a crucial role. As we previously explained, not all neu-

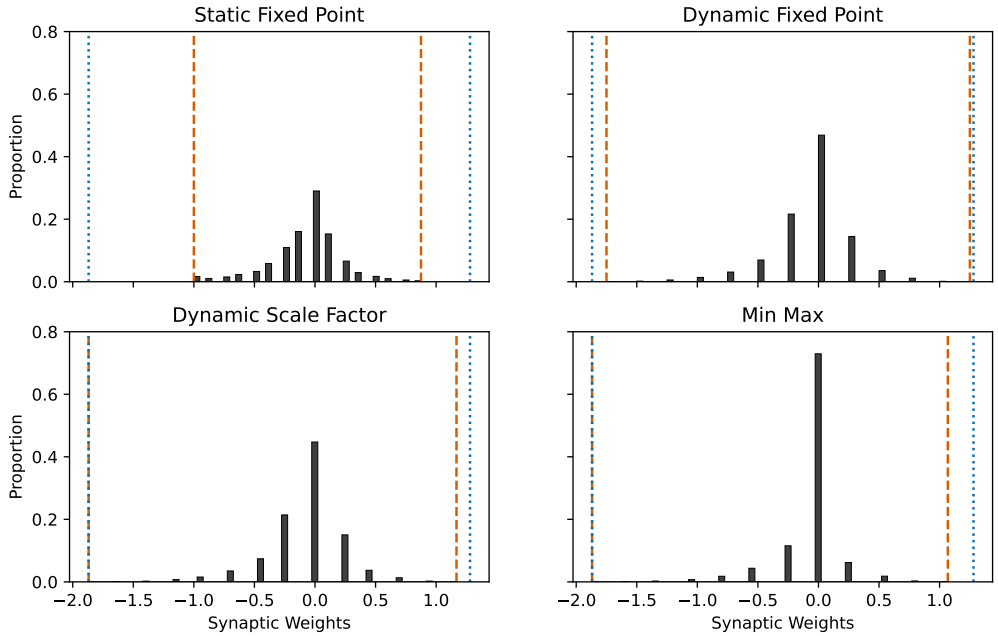

Figure 5.11 – Comparison of weight distribution after quantization. The blue dotted lines represent the minimal and maximal values of the original weight distribution, while the red dashed lines indicate the extreme values of the quantized weights. The height of each bin corresponds to the proportion of synaptic weights at the given value.

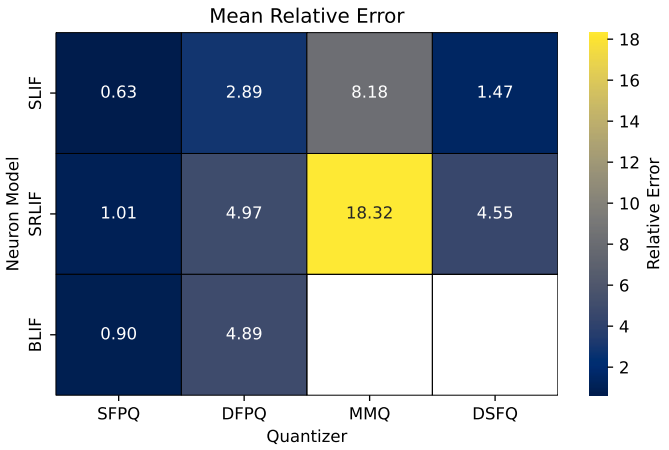

Figure 5.12 – Matrix representation of average relative error depending on the neuron model as rows and the quantization methods as columns.

ron models are compatible with all quantization. DSFQ and MMQ cannot be used with the BLIF neuron model that relies on multiplication due to the non-power-of-two scale factor. Moreover, each neuron model seems to exhibit a different sensitivity to quantization errors introduced by each quantizer.
The second important aspect is the distribution of synpatic weight. Since DSFQ and MMQ are based on extreme value, which can be rare, the resulting quantizer may allocate more precision to these rare values. When these extreme values are below the minimum bound used in FPQ, DSFQ and MMQ may achieve a higher effective precision by adapting their scale factor to actual synaptic weight distribution.

Quantization Impact on Hardware Performances

In the previous section, we studied the impact of quantization on accuracy metrics. In this section, we will explore the impact of quantization on hardware performance, especially on memory consumption and dynamic power consumption.
Figure 5.13 illustrates the memory consumption of the previously trained models, categorized by neuron model and quantizer target bitwidth. Since memory consumption depends solely on the number of synaptic weights within the model, the consumption levels for the three models are equivalent. As expected, reducing the quantizer bitwidth results in a linear decrease in memory consumption.
Figure 5.14 illustrates the dynamic power consumption (in mW) depending on the neuron model and quantizer target bitwidth. Due to the utilization of DSP, the BLIF-based model has higher power consumption compared to other neuron models. However, since the BRAM consumption decreases with lower bitwidth, the overall power consumption of the model also reduces.
To determine the optimal tradeoff between power consumption and accuracy, we can define a new metric called Ratio Power Accuracy (R_{pa}) (in mW/%) defined by Equation 5.14. This metric represents the amount of power consumed per percentage of accuracy. Figure 5.15 represents the R_{pa} ratio as a function of the quantizer target bitwidth, quantization method, and sorted neuron model. A lower R_{pa} indicates a better tradeoff between power consumption and accuracy.

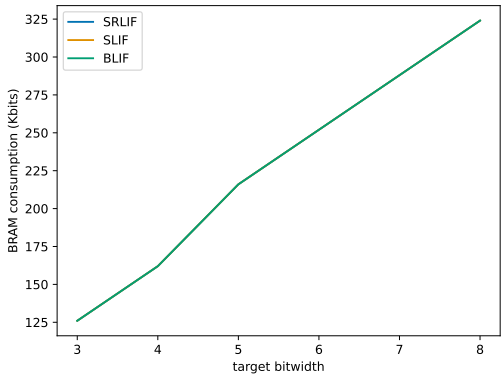

Figure 5.13 – BRAM consumption (Kbit) evolution through quantizer bitwidth depending on the neuron models. Because the BRAM consumption depends solely on the number and the resolution of the synaptic weights, the three curves are confused.

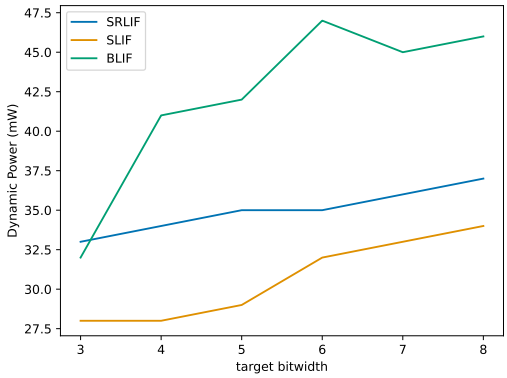

Figure 5.14 – Dynamic power consumption (mW) evolution depending on the quantizer bitwidth and the neuron model. The BLIF-based model consumes more power due to the use of DSP expected for 3-bit quantizer bitwidth. In all cases, the power consumption increases linearly through synaptic weight resolution.

$$R_{pa} = \frac{P}{Acc} \tag{5.14}$$

For the SRLIF neuron model, the R_{pa} decreases as the bitwidth increases. Contrary to the SRLIF, the R_{pa} for the BLIF neuron model increases with higher quantizer resolution. In the middle graph, we observe that the R_{pa} of the SLIF neuron model reaches an optimal value for 4 bits. We can also observe that the MMQ quantizer has a higher R_{pa} compared to other quantizers, followed by SFPQ, DSFQ, and DFPQ.

Quantization Impact Depending on Other Parameters

In the previous section, we studied the impact of quantization on accuracy and hardware performance using a specific network configuration. All these networks share the same topology, dataset, dataset transformation, and training parameters. In this section, we will present a non-exhaustive list of other parameters that can influence the impact of quantization on network performance.

One parameter that can significantly impact the quantization sensitivity is the number of training epochs. Figure 5.16 illustrates the evolution of PTQ quantization for 4-bit resolution. The network topology used was the same network topology $2312 - 10$ with the SRLIF neuron model. The first observation is the DSFQ and MMQ; based on extreme values, relative error highly increases when the number of epochs increases. In contradiction, FPQ-based methods decrease when the number of training epochs increases. However, the DFPQ method increases after 40 training epochs.

Figure 5.17, which represents the evolution of synaptic weight distribution as a function of the number of training epochs, can help to explain the previous observations. For

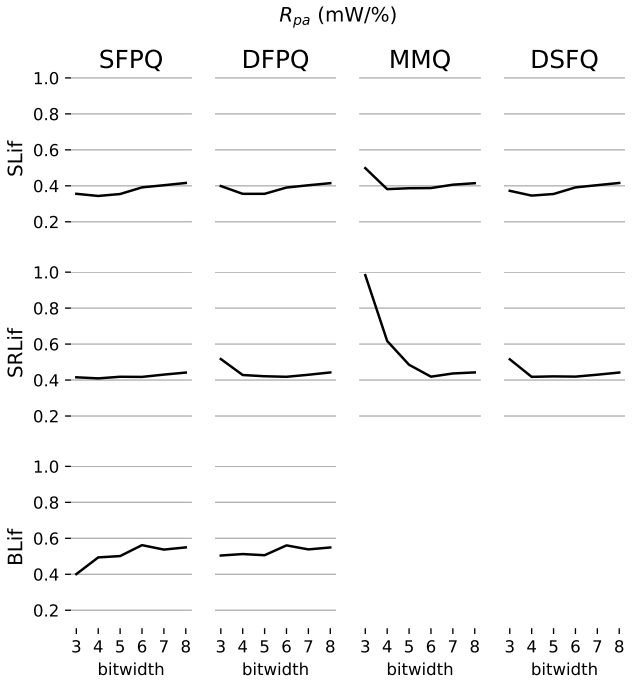

Figure 5.15 – R_{pa} ratio evolution depending on quantizer bitwidth and sorted by neuron model. The R_{pa} represents the best tradeoff between power consumption and accuracy. For SRLIF and SLIF neuron models, the curve decreases, reaches a minimal point, and then increases. The exception is the BLIF neuron model.

DSFQ and MMQ, the increase in relative error can be attributed to the extreme values shown in blue, which are not representative of the overall weight distribution, represented by the standard distribution in green. However, FPQ-based quantizers use a fixed resolution, and a broader distribution of synaptic weights will utilize a larger range of possible quantized values. However, after 40 epochs, the minimal falls below -1.0, causing the DFPQ to mimic the behavior of extreme value-based quantizers. Beyond this point, the relative error of DFPQ decreases for the same reasons previously discussed.

Another parameter that can significantly affect the robustness of quantization is data transformations. For more details about the transformation used, you can read Section 4.2.2. Figure 5.18 illustrates the impact of quantization on a single-layer $2312 - 10$ model based on the SRLIF neuron model trained over 10 epochs. The results were obtained for 4-bit PTQ. The number of frames can have a substantial impact on quantization error, particularly for MMQ and DFPQ quantization methods. This error reduction can be explained by the reduction of quantization error accumulation during an emulation step. As each emulation step, which is equivalent to the number of frames, increases, the number of spikes per frame decreases, thereby potentially mitigating the accumulation of quantization errors. However, no in-depth studies have been conducted to confirm this hypothesis.

Finally, the chosen dataset can have a significant influence on quantization impact. Previous experiments were done on the N-MNIST dataset. However, changing the dataset can result in variations in quantization dynamics. Figure 5.19 illustrates the

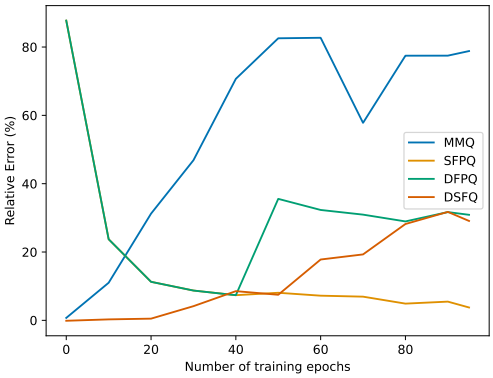

Figure 5.16 – Evolution of relative error depending on the number of epochs and the quantization methods. The model is a single layer with the SRLIF neuron model with the N-MNIST dataset.

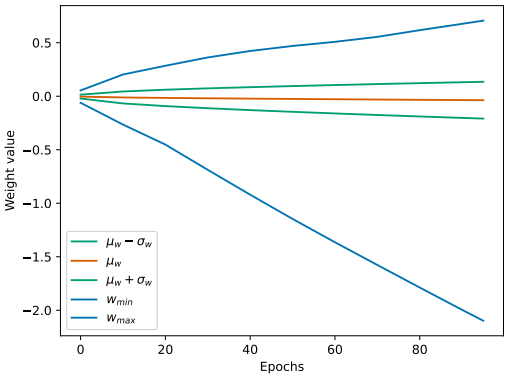

Figure 5.17 – Weight distribution of the network through the number of trained epochs. The red line represents the average value of weights, the green curves represent the standard deviation of the weight distribution, and the blue lines represent the extreme positive and negative values. The extreme values, especially the negative values, highly increase when the number of epochs increases, making the extreme-based quantizer less precise.

quantization dynamics depending on quantizer bitwidth for the SHD dataset. Due to a higher noise level in the SHD, the impact of quantization is more pronounced.

5.3.3 Conclusion

In this section, we studied the effects of Post-Training Quantization (PTQ) with ModNEF neuron models. The study was focused solely on software quantization simulation, but as we demonstrated in the previous Chapter 4, the quantization simulation is accurate with FPGA results.
In Section 5.3.1, we examined the impact of quantization at a neuron level. This straightforward approach led us to formulate several hypotheses, as follows:

1. The BLIF neuron model is the most sensitive to quantization, followed by the SLIF and SRLIF models.
2. The more accurate quantization method is highly dependent on the neuron model.

These hypotheses were confirmed in Section 5.3.2, where we studied the impact of quantization at a network level with a particular focus on a specific network topology: an N-MNIST single-layer network: $2312 - 10$. Observations made during this section confirmed our hypothesis, indicating that the BLIF neuron model is particularly sensitive, while the SLIF model is less sensitive to quantization compared to other models.

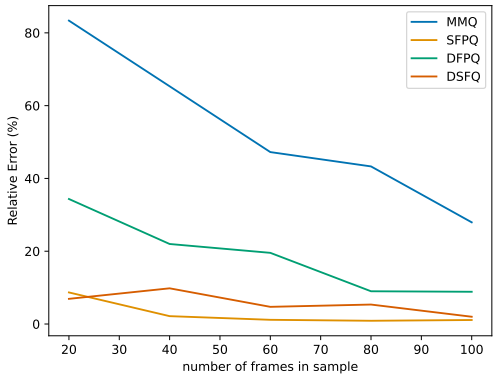

Figure 5.18 – Relative error evolution depends on the number of frames. The model is a single-layer model based on the SRLIF neuron model. The quantizer was parameterized for 4-bit resolution.

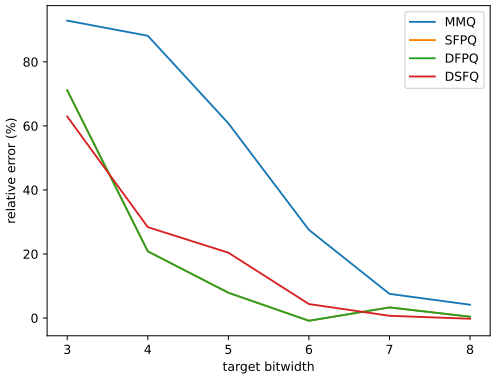

Figure 5.19 – Relative error evolution depends on the quantizer target bitwidth for the SHD dataset. The model is composed of two layers based on the SRLIF neuron model.

However, these observations must be considered with critical hindsight. Our findings are based on a single network topology with a specific dataset and dataset transformations. Section 5.3.2 highlights the importance of other parameters affecting quantization robustness, such as the number of training epochs, the data transformation, or the dataset. Therefore, observations made during this section may be subject to change with different network topologies or datasets.
Although these observations are limited, Section 5.3.2 highlights the effect of quantization on hardware performance. Since BRAM consumption is directly related to the quantization bitwidth, lower bitwidth resolutions result in decreased BRAM and power consumption. However, accuracy also decreases with lower synaptic weight resolution. Even though we have defined the R_{pa} metrics to find the best tradeoff between power consumption and accuracy, it would be highly beneficial to find a way to achieve low weight resolution with a moderate drop in accuracy. This is the promise of Quantization Aware Training (QAT), a learning method that considers quantization during training to significantly mitigate the drop in accuracy after PTQ.

5.4 Quantization Aware Training in ModNEF

In the previous section, we studied the impact of Post-Training Quantization (PTQ) on accuracy, power consumption, and hardware resources, particularly BRAM consumption. For low synaptic weight resolution, both BRAM consumption and power consumption decrease linearly. However, accuracy can drastically drop due to quantization errors. We found that some parameters, such as the neuron model, the quantization method, the dataset transformation, and the number of training epochs, can mitigate the quantization impact and minimize the accuracy drop. Nevertheless, the accuracy is always impacted, and this drop can mitigate the power consumption gain. To achieve low bitwidth resolution and significantly mitigate the accuracy drop, it is possible to train our model by considering already quantized weights and the effect of quantization on synaptic weights, hyperparameters, and membrane voltage. This can

be accomplished by using the Quantization Aware Training (QAT) algorithm. To propose the most comprehensive framework for SNN FPGA integration, we implemented the QAT algorithm within the ModNEF framework. In this section, we will present the QAT integrated into the ModNEF framework. We introduce a new notation, Full-Precision Training (FPT) to differentiate between the two training methods: training without quantization, referred to as FPT, and training with quantization, referred to as QAT.
In Section 5.4.1, we will examine how QAT is implemented within the ModNEF framework. Section 5.4.2 will present the initial results of QAT runs. Following this, in Section 5.4.3, we will explore methods to reduce bitwidth for ultra-low bitwidth resolution. Before concluding, we will highlight in Section 5.4.4 the limitations of the QAT algorithm.

5.4.1 Quantization-Aware Training Algorithm in ModNEF

ModNEF implements its own QAT algorithm instead of relying on existing Python libraries such as Brevitas [53], for two main reasons:

- The documentation of the library is unclear, and it was not evident whether Brevitas propose quantizers comparable to the ModNEF quantizers. Additionally, the process of implementing new quantizers compatible with Brevitas is poorly documented, making the development of the ModNEF quantizers difficult.
- Brevitas was primarily designed for quantizing ANNs and not for SNN. However, in SNN, neuron membrane voltage and neuron hyperparameters must also be quantized in addition to synaptic weight, which is not done by Brevitas, which only provides quantized synaptic weights.

In ModNEF, QAT is implemented at two different levels.

1. The first level of implementation is the layer/neuron group level. Each ModNEF neuron model incorporates a `QuantLinear`, which is an override class of PyTorch's `Linear` class. The `QuantLinear` will store the synaptic weight in a floating-point representation. During the **forward pass**, `QuantLinear` can accept a ModNEF quantizer. If a quantizer is provided, the class will simulate the quantization of synaptic weight (see Section 5.2 and Equation 5.4 for more details) and will use the quantized weights for linear PyTorch functions. Otherwise, the floating-point weights are used. The membrane voltage and input currents, which are outputs of `QuantLinear`, are summed, and then the neuron membrane update function is applied. Quantization is then applied to membrane voltage before spike detection, and we return output spikes and quantized membrane voltage, which is used for the backward pass. The **backward pass** remains unchanged and is performed in floating-point. Figure 5.20 illustrates the forward pass implemented in ModNEF. The white cells represent the operations always executed; the blue cell is only executed in FPT, and the red cells are only executed during QAT.
2. The second level is the training loop level, as described by Algorithm 2. Before training begins, and if it is necessary, the algorithm initializes the model quantizers. The algorithm then enters the training loop. Within this loop, the model quantizes neuron hyperparameters if the quantizer parameters have changed during the previous epoch. The dataset is then passed through the network. After training, the model will clamp the synaptic weights to remain within the quantizer range. Since DSFQ and MMQ use dynamic parameters, changes in synaptic

weights during epoch training necessitate an update of quantizer parameters. This update is performed before the weight clamping by setting the `force_init` parameter to `True`.

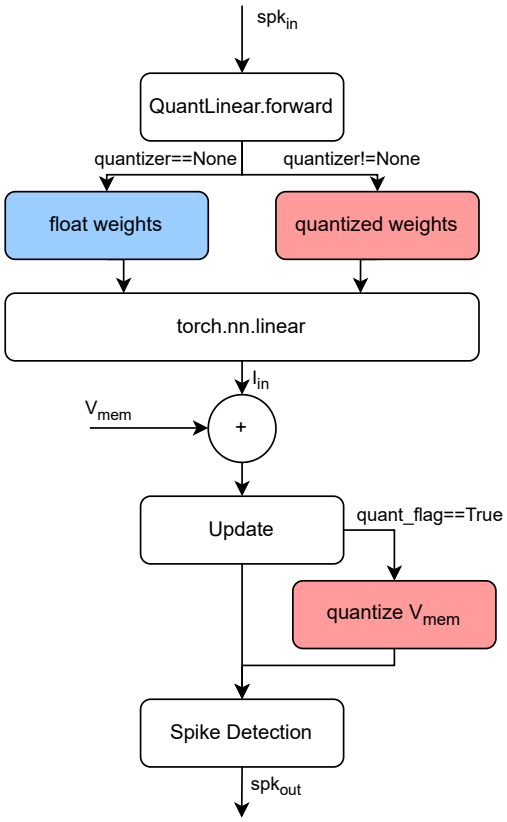

Figure 5.20 – Forward execution flow at a neuron level. Cells in white are always executed, cells in blue are executed only during FPT, and red ones are only executed during QAT.

Algorithm 2 ModNEF Training Loop. The algorithm begins by initializing the model's quantizers. During each epoch, the quantizers are updated. After the training loop, the synaptic weights are clamped, and the quantizer's internal parameters are updated.

```
if qat = True then
    model.init_quantizer()
end if
for each epoch e in epochs do
    if qat = True then
        model.quantize_hp()
    end if
    for each batch in DataLoader do
        Forward pass
        Backward pass
    end for
    if qat = True then
        model.clamp(fore_init=True)
    end if
end for
```

5.4.2 First Results

In the previous section, we introduced the main ideas of the QAT algorithm implemented in the ModNEF framework. In this section, we will present the initial results of QAT. We conducted our experimentation on the N-MNIST dataset as a proof of concept and on the SHD dataset, which is more sensitive to quantization. For these experiments, we exclusively used the FPQ quantizer with a fixed-point position equal to $bw - 1$, where bw is the target quantizer bitwidth.

For each experiment, we have generated a "seed" model, which is an untrained model used to ensure that we train the same initial model. This approach highlights the effect of the training method by ignoring the variability due to random initialization. The model trained with the FPT method was inferred with two different bitwidth. We ran the model with an 8-bit quantizer, a common bitwidth used since the beginning of the thesis, and 4-bit, which is the bitwidth used for QAT runs. For QAT, we choose 4-bit because, as we observed in the previous section, the impact of quantization becomes significant at this bitwidth, which is high enough to simplify training. In a future section, we will explore methods to achieve lower resolution.

Proof of Concept with N-MNIST Dataset

In this section, we will present results regarding the use case with the N-MNIST dataset. We chose this dataset because this dataset is a relatively simple classification task and is not highly sensitive to quantization, especially with the FPQ quantizer, as we saw in Section 5.3.2.
The model trained uses the BLIF neuron model and follows the topology: $2312-128-10$. The model was trained over 50 epochs.
Table 5.2 presents the accuracy results of three models. The models FPT_8 and FPT_4 are the same model trained using the FPT algorithm but with different quantizer bitwidth, respectively 8 and 4 bits. The QAT_4 model was trained using the QAT algorithm with 4-bit bitwidth. Since the full precision evaluation, i.e., without quantization, is not relevant for the QAT_4 model, we do not show full precision accuracy on the table results. The two key inference methods are PTQ simulation and FPGA accuracy.
For FPT_8, there is no significant drop in accuracy due to high resolution. However, for the FPT_4, we observe a 3% drop in accuracy between full precision and PTQ. In contrast, the QAT_4 model achieves similar accuracy as FPT_8 but with a lower quantizer bitwidth.

Table 5.2 – Accuracy results for the N-MNIST dataset with different training methods and synaptic weight resolution. The FPT_4 model achieves lower accuracy compared to the FPT_8 model, whereas the QAT_4 model achieves comparable accuracy despite its lower resolution, demonstrating the positive impact of the QAT algorithm.

Evaluation Method	FPT_8	FPT_4	QAT_4
Bitwidth	8	4	4
Full Precision	97.50	97.50	-
PTQ Simulation	97.44	94.33	96.77
FPGA	97.33	93.34	96.77

The accuracy of the QAT_4 model is slightly lower than the accuracy of the FPT_8 model. Therefore, in terms of accuracy, there are no significant gains from using the QAT algorithm. However, in terms of power and energy consumption, the QAT algorithm becomes more advantageous due to its lower synaptic resolution.
Table 5.3 compares the energy and power consumption between the models. Since both the FPT_4 and QAT_4 use lower bitwidth than the FPT_8, their power consumption is significantly reduced. However, if we compare the R_{pa}, which represents the power

consumed to achieve 1 % accuracy, the QAT_4 model demonstrates better performance than other models.
The FPT_8 model achieves higher accuracy than other models, but to achieve this higher accuracy, the model consumes more power. In contrast, the FPT_4 model achieves lower power consumption but achieves less accuracy. The QAT_4 model emerges as the best trade-off, achieving similar accuracy to the FPT_8 model but with the same power consumption as the FPT_4 model, resulting in a lower R_{pa}.

Table 5.3 – Energy and power consumption of N-MNIST models with different training methods and synaptic weight resolution. Both 4-bit models exhibit identical energy and power consumption due to their shared network topology and weight resolution. However, the QAT_4 model achieves a better R_{pa} by combining lower power consumption while maintaining high accuracy.

Target	FPT_8	FPT_4	QAT_4
Inference Speed (μs)	41.56	41.56	41.56
Total Power (mW)	611	366	366
Energy Consumption (μJ)	25.39	15.21	15.21
R_{pa} (mW/%)	6.28	3.92	3.78

The power reduction can be explained by examining Table 5.4. The reduction of quantizer bitwidth leads to a reduction of BRAM consumption. Additionally, the reduction of encoding bitwidth has resulted in a decrease in DSP usage. This reduction can be attributed to internal Vivado optimizations, which have determined that DSP units are not necessary for multiplication due to lower synaptic weight bitwidth.

Table 5.4 – Hardware metrics results comparison between FPT and QAT models for the N-MNIST dataset. The power reduction observed in 4-bit models compared to the 8-bit model is driven by decreased BRAM and DSP usage. The bitwidth reduction prompts Vivado to implement multiplications using LUT instead of DSP, further contributing to observed power savings.

Metrics	FPT_8	FPT_4	QAT_4
Dynamic Power (mW)	489	253	253
Static Power (mW)	113	113	113
BRAM (Kbit)	4158 ; 252 (87.50%)	2088 ; 252 (46.43%)	2088 ; 252 (46.43%)
FF	6205 ; 355 (6.17%)	5068 ; 355 (5.10%)	5068 ; 355 (5.10%)
LUT	9035 ; 536 (17.99%)	11905 ; 537 (23.39%)	11905 ; 537 (23.39%)
LUTRAM	0 ; 96 (0.55%)	0 ; 96 (0.55%)	0 ; 96 (0.55%)
DSP	138 ; 0 (62.73%)	0 ; 0 (0.00%)	0 ; 0 (0.00%)

This initial use case validates the QAT algorithm, demonstrating that it can achieve similar results to those of FPT training but with lower synaptic weight resolution. This resolution reduction leads to a reduction of power consumption and hardware resource

usage. Since the model consumes less power while maintaining similar accuracy, the R_{pa} ratio decreases, indicating that less power is required to achieve 1 % accuracy.
Due to the lower complexity of N-MNIST dataset, even a suboptimal training configuration can yield acceptable performance. However, these performances may not generalize to more challenging dataset such as SHD.

Algorithm Validation with the SHD Dataset

In this section, we will present results obtained on the SHD dataset. Unlike N-MNIST, which is a relatively simple classification task, SHD is more sensitive to quantization and can thus provide a more rigorous validation of the QAT algorithm.
For this study, we used the SRLIF neuron model with a 700-200-20 network topology, and the model was trained over 75 epochs.
Table 5.5 presents the accuracy of inferred models. To enhance clarity, we used the same model nomenclature as the previous section. As we observed, the impact of quantization is more significant on the SHD dataset compared to the N-MNIST dataset, with accuracy drops of 15% for 4-bit bitwidth.
The QAT_4 model achieves a similar accuracy than FPT_8 with 71.18%. However, when we compare models with similar quantization bitwidth, the QAT_4 model demonstrates 27% higher accuracy than the FPT model.

Table 5.5 – Accuracy results on the SHD dataset with different training methods and synaptic weight resolution. Due to the increased noise in the SHD dataset, models exhibit greater sensitivity to quantization. The FPT_8 model has a 4% drop in accuracy and 30% loss for the FPT_4 model. In contrast, the QAT_4 model achieves accuracy comparable to full-precision evaluations of FPT models.

Evaluation Method	FPT_8	FPT_4	QAT_4
Bitwidth	8	4	4
Full Precision	71.61	71.61	-
PTQ Simulation	72.05	44.86	72.18
FPGA	67.01	43.21	70.04

In terms of power consumption, presented in Table 5.6, the 4-bit models consume less power than the 8-bit model. When we consider the R_{pa}, we observe the QAT_4 model achieve the lowest R_{pa}, making it the best model if we want to maximizing accuracy while minimizing power consumption.
An interesting observation is the higher R_{pa} value for the FPT_4 model compared to the FPT_8 model. Although the FPT_8 model consumes more power, its superior accuracy results in a lower R_{pa} compared to the FPT_4.
Contrary to the N-MNIST use case, where the power consumption reduction could be explained by decreased BRAM and DSP consumption. The usage of the SRLIF neuron model avoids the usage of DSP; the power reduction cannot be explained by a reduction of DSP usage as in the N-MNIST use case. Therefore, the reduction of power consumption can be attributed to lower BRAM consumption, as shown in Table 5.7.
The BRAM reduction leads to a reduction of FF and LUT usage, which can also contribute to the observed power reduction.

Table 5.6 – Energy and power consumption of SHD models with different training methods and synaptic weight resolution. All models maintain similar inference speed, but the 4-bit models benefit from lower power consumption, resulting in reduced energy consumption. Notably, while the FPT_4 model achieves a higher R_{pa}, even though power consumption is reduced, the accuracy drop remains too important to compensate for the power decrease.

Target	FPT_8	FPT_4	QAT_4
Inference Speed (μs)	109.5	109.71	109.48
Total Power (mW)	331	268	268
Energy Consumption (μJ)	36.24	29.40	29.34
R_{pa} (mW/%)	4.69	4.75	3.98

Table 5.7 – Hardware metrics results comparison between FPT and QAT models for the SHD use case. Power consumption reduction is attributed to reduced BRAM and LUT usage.

Metrics	FPT_8	FPT_4	QAT_4
Dynamic Power (mW)	219	158	158
Static Power (mW)	112	110	110
BRAM (Kbit)	1692 ; 252 (38.57%)	864 ; 252 (22.14%)	864 ; 252 (22.14%)
FF	5660 ; 352 (5.65%)	4048 ; 352 (4.14%)	4048 ; 352 (4.14%)
LUT	23784 ; 526 (45.70%)	16562 ; 510 (32.09%)	16562 ; 510 (32.09%)
LUTRAM	0 ; 128 (0.74%)	0 ; 128 (0.74%)	0 ; 128 (0.74%)
DSP	0 ; 0 (0.00%)	0 ; 0 (0.00%)	0 ; 0 (0.00%)

The use case with the SHD dataset confirmed the effective performance of the QAT algorithm. Similar to the results obtained with the N-MNIST dataset, we achieve slightly lower but similar accuracy to the FPT results. However, when we compare models with the same synaptic resolution, we attained better accuracy, consequently, a better R_{pa} than the FPT model.

5.4.3 Reach Lower Bit Resolution with Quantizer Scheduler

In the previous section, we studied the performances of the QAT algorithm with a fixed target bitwidth for both experiments. It is reasonable to assume that if the QAT algorithm works with 4-bit resolution, the algorithm will achieve good performance for higher target bitwidth. However, we need to validate the algorithm for lower resolution.
A first naïve approach is to run QAT training with 3-bit resolution. Figure 5.21 illustrates the loss evolution through epochs during QAT training for the N-MNIST dataset. When running QAT training with 3-bit, the model fails to learn, resulting in a linear loss.
This phenomenon can be explained with Figure 5.22. The blue curve represents the weight distribution of the initial seed model, i.e., without training, and the two vertical

red lines represent the minimal quantizer step, equal to $\frac{1}{2^{3-1}} = 0.25$. Since all synaptic weights are below the quantizer step, the initial quantization will quantize all synaptic weights to 0. Consequently, the input current is equal to 0, the membrane voltage remains at 0, preventing the gradient from being computed, and the layer does not emit output spikes.
A solution to address this issue is to start training with higher bitwidth and gradually reduce quantizer bitwidth. To achieve this, we developed **`QuantizerScheduler`**, which, similar to the learning rate scheduler proposed by PyTorch, progressively reduces the quantization bitwidth. The following code shows the definition of a quantizer scheduler. Every `T` epoch, the quantizer will reduce the quantizer bitwidth by creating a new quantizer with the defined lambda function `quantizationMethod`. To avoid saving models that are not trained on the target bitwidth, **`QuantizerScheduler`** defines a method `save_model()` that returns `True` if the target bitwidth is reached.

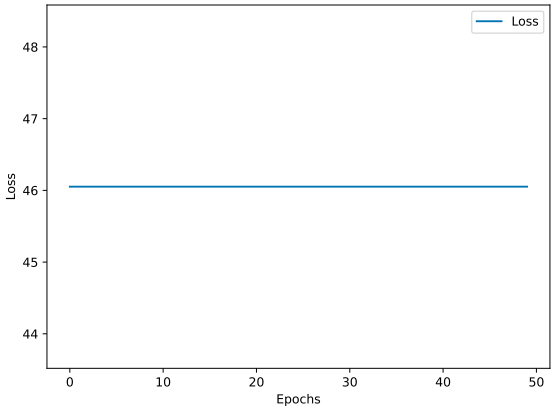

Figure 5.21 – Loss evolution for 3-bits QAT training for the N-MNIST dataset. Because the synaptic weights of the untrained model are too small, the first quantization results in all 0 weights, disabling the model to modify the membrane voltage and so disabling the gradient calculation.

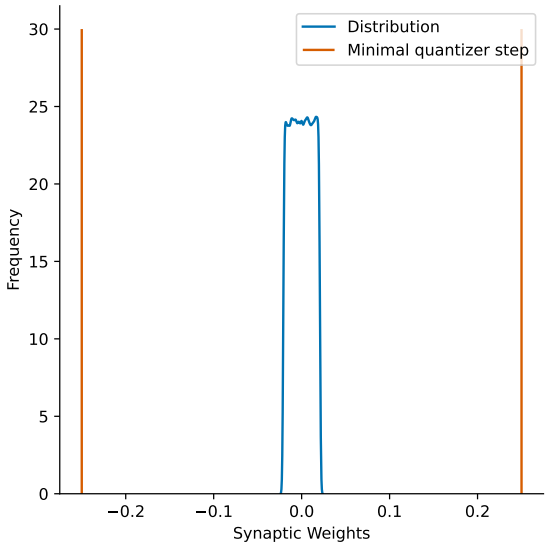

Figure 5.22 – Weight distribution (blue curve) of seed, i.e., untrained, model with quantizer minimal step illustrated by the two red lines.

We tested our bitwidth scheduler on the two previous datasets, N-MNIST as a proof of concept for achieving low synaptic weight resolution and SHD and DVS Gesture datasets to validate our methods.

Proof of Concept with the N-MNIST Dataset

We trained three different models, all based on the $2312-128-10$ network topology. As in the previous section, we trained a full-precision model with different quantization bitwidth targets: FPT_8, FPT_3, and FPT_2 with 8, 3, and 2-bit resolution, respectively.

```
QuantizerScheduler(model=model, # model to train
                   bit_range=(8, 3), # limits of the bitwidth domain
                   T=3, # number of epochs after we change bitwidth
                   quantizationMethod=lambda n : FixedPointQuantizer(n,
                      n-1, True) # definition of new quantizer
                   )
```

Listing 5.1 – Definition of Quantizer Scheduler in the ModNEF framework. We schedule the quantizer of the model from 8 to 3 while reducing the quantizer bitwidth every 3 epochs.

Additionally, we trained two models with the QAT algorithm: QAT_3 trained with 3-bit synaptic resolution and QAT_2 trained with 2-bit resolution.
Table 5.8 summarized the accuracy obtained during inference. As we can see, the QAT_3 model achieves a similar but slightly lower accuracy than the FPT_8 model. The accuracy drop becomes more important with 2-bit quantization. However, if we compare the two 2-bit models, FPT_2 and QAT_2, we achieve better accuracy with QAT training, similar to that of the FPT_3 model.

Table 5.8 – Accuracy results with quantizer bitwidth scheduler on the N-MNIST dataset. The accuracy loss of FPT-based models is more pronounced due to lower resolution. While QAT-based models maintain comparable accuracy, the drastic reduced resolution significantly degrades the accuracy, especially for the QAT_2 model.

Evaluation Method	FPT_8	FPT_3	FPT_2	QAT_3	QAT_2
Bitwidth	8	3	2	3	2
Full Precision	97.54	97.54	97.54	-	-
PTQ simulation	97.50	94.04	70.10	96.24	94.63
FPGA	97.46	94.11	70.63	96.05	94.31

In terms of power consumption, Table 5.9 highlights the benefit of low-resolution quantization. Thanks to the lower bitwidth, the power consumption of the 3 and 2-bit models significantly decreases compared to the FPT_8 model. This power reduction can be explained by the lower BRAM consumption and the avoidance of DSP usage, as discussed in Section 5.4.2. Due to lower power consumption and a mitigated accuracy drop of QAT models, QAT_3 and QAT_2 models achieve lower R_{pa} ration than FPT models. A particular observation is the higher R_{pa} ratio of the FPT_2 model compared to the FPT_3 model. Even though the FPT_2 model consumes less power, the accuracy drop of FPT_2 is too significant to compensate for the reduction in power consumption.
Figures 5.23 and 5.24 illustrate the accuracy in the upper graphs and the quantizer bitwidth in the lower graphs, through epochs with respectively 3 and 2 bits target resolution. The dashed red lines represent the epochs where the target bitwidth is reached. From the 3-bit resolution, the accuracy starts to drop in both trainings. However, for the 3-bit resolution, the accuracy is maintained around at 95%. In contrast, for the 2-bit training, the accuracy globally decreases with a high variability once the 2-bit resolution is reached. This behavior can be explained by the ultra-low resolution and the limited possible synaptic weight values: 8 values for 3-bit and 4 values for 2-

Table 5.9 – Energy and power consumption of N-MNIST models trained with bitwidth scheduler. All models maintain comparable inference speed but achieve lower energy consumption due to reduced power consumption. While QAT-based models achieve good accuracy with lower power consumption, their R_{pa} values remain minimal compared to FPT-based models.

Target	FPT_8	FPT_3	FPT_2	QAT_3	QAT_2
Inference Speed (μs)	41.56	41.56	41.57	41.57	41.56
Total Power (mW)	611	220	205	220	205
Energy Consumption (μJ)	25.39	9.14	8.52	9.14	8.52
R_{pa} (mW/%)	6.27	2.33	2.90	2.29	2.17

bit resolution. This drastic reduction of values set increases the training difficulties because a small change of synaptic weights can lead to significant differences in accuracy.

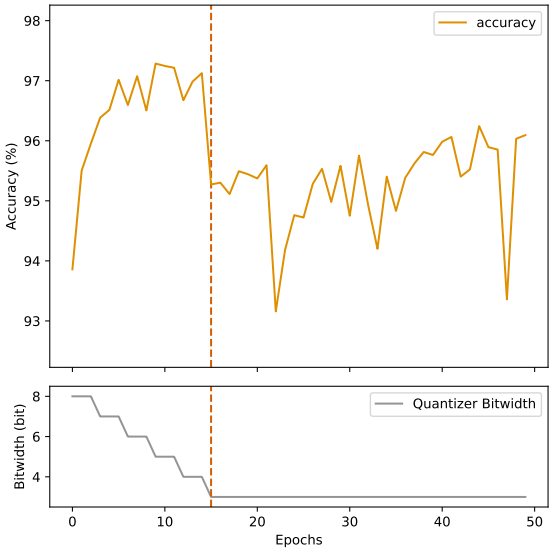

Figure 5.23 – Accuracy, illustrated in the upper graph, and quantizer bitwidth, presented in the lower graph, evolution during training through epochs for 3-bit target quantizer bitwidth.

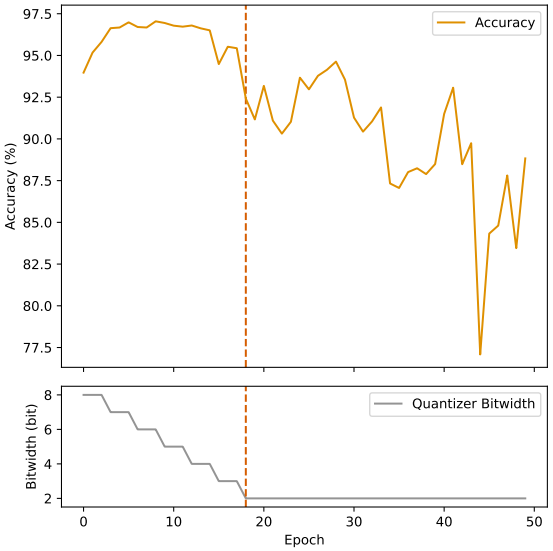

Figure 5.24 – Accuracy, illustrated in the upper graph, and quantizer bitwidth, presented in the lower graph, evolution during training through epochs for 2-bit target quantizer bitwidth.

Validation with the SHD Dataset

In the previous section, we demonstrated the usefulness of the quantizer scheduler to achieve low bitwidth in the N-MNIST dataset. As we previously observed, the N-MNIST dataset is a simple classification task where good results can be achieved with a non-optimal algorithm. In this section, we will present results on the SHD dataset. As in previous experiments, we trained a full precision model, denoted as FPT, and

two QAT models, denoted as QAT. All models followed the $700 - 200 - 20$ network topology. Thanks to the modular architecture of ModNEF, we can generate mixed-precision networks with different quantization bitwidth depending on the module. In this section, we quantized our model with two quantization schemes. Models denoted with a $3 - 3$ subscript refer to models where both layers are quantized with 3-bit resolution. In the other case, models denoted with $3 - 8$ refer to models where the hidden layer is quantized with 3-bit resolution and the output layer is quantized with 8-bit resolution. To ignore the quantization bitwidth scheduling for the output layer, we deactivate the scheduling by setting the `allow_scheduling` flag during quantizer initialization to false, as shown in the following code.

```
FixedPointQuantizer(bitwidth=8, fixed_point=7, signed=True,
    allow_scheduling=False)
```

Listing 5.2 – Example of a Fixed Point ModNEF quantizer to quantize signed values at an 8-bit bitwidth with a fixed position fixed at the bit 7. The allow_scheduling set at False disables the quantizer scheduler to change the quantizer bitwidth.

Table 5.10 presents the accuracy of all inferred models. Due to low resolution, the FPT_{3-3} and FPT_{3-8} models achieve low accuracy. We can notice that the FPT_{3-8} model does not achieve significantly higher accuracy than the FPT_{3-3} model. We can explain this behavior because the hidden layer is highly quantized, preventing the output layer from effectively processing the spikes generated by the hidden layers. However, QAT-based models achieve good accuracy results. The QAT_{3-3} models achieve lower accuracy than the FPT_8 model, from 72.19% to 64.38% due to the low resolution. The QAT_{3-8} model appears as a good trade-off in terms of accuracy with 69.15%, which is only 3% lower than the FPT_8 model, compared to a 7% drop with the QAT_{3-3} models.

Table 5.10 – Accuracy results with quantizer bitwidth scheduler on SHD dataset. Quantization severely degrades the accuracy of FPT-based models. In contrast QAT-based models achieve good accuracy, nearly less than the full-precision models. The QAT_{3-8} model performs the best accuracy, since the high resolution of the output layer minimizes the quantization impact.

Evaluation Method	FPT_8	FPT_{3-3}	FPT_{3-8}	QAT_{3-3}	QAT_{3-8}
Bitwidth	8	3-3	3-8	3-3	3-8
Full Precision	72.19	72.19	72.19	-	-
PTQ simulation	72.19	16.52	18.66	65.36	70.09
FPGA	72.05	15.40	17.63	64.38	69.15

Table 5.11 references the power consumption of different models. A notable observation is the absence of power reduction with a lower bitwidth quantizer.
This observation can be explained with Table 5.12, which detailed the power consumption estimation generated by Vivado. We can see the BRAM power consumption decrease with lower bitwidth as expected. However, the Slice and Signals power consumption increases with lower bitwidth resolution. This can be explained by the

non-standard bitwidth. Our hypothesis is that, in the previous experiments, this non-standard format was mitigated by the layer size equal to a power of two. However, because the hidden layer size in this network is fixed at 200 neurons, Vivado cannot optimize memory and will use additional resources for memory mapping.
Because of this power increase, the quantized models achieve a lower R_{pa} ratio than the FPT_8 models. To achieve a better R_{pa}, one solution, based on our previously described hypothesis, can be to define another model with a hidden layer size as a power of 2. It is important to note that this hypothesis was not clearly identified and tested due to a lack of time.

Table 5.11 – Energy and power consumption of SHD model trained with bitwidth scheduler. Contrary to previous experiments, reducing resolution does not decrease power consumption due to the irregular BRAM size, which increases the power consumption. It results in higher R_{pa}s values.

Target	FPT_8	FPT_{3-3}	FPT_{3-8}	QAT_{3-3}	QAT_{3-8}
Inference Speed (μs)	109.53	109.60	109.57	109.76	109.83
Total Power (mW)	331	338	339	338	339
Energy Consumption (μJ)	36.25	37.05	39.97	37.10	37.23
R_{pa} (mW/%)	4.59	21.95	19.23	5.25	4.90

Table 5.12 – Power consumption repartition comparision between FPT and QAT models for the SHD models trained with the bitwidth quantizer scheduler. Power consumption of BRAM decreases, but the Slice and Signals power increase while the circuit of memory multiplexing increases due to irregular memory size.

Target	FPT_8	FPT_{3-3}	FPT_{3-8}	QAT_{3-3}	QAT_{3-8}
Clock	28	28	27	28	27
Slice	76	89	90	89	90
Signals	79	85	86	85	86
BRAM	35	24	25	24	25
Static	112	111	111	111	111
Total	331	338	339	338	339

Application to Recurrent Network with DVS Gesture

In this section, we will present results on models based on the DVS Gesture dataset. The trained models are based on the following network topology: $3200 - 128_R - 11_R$, where the subscript R indicates that the layer is locally recurrent, i.e., connected to itself. We previously explained the data transformation applied to DVS Gesture in Section 4.3.2.
Table 5.13 summarizes the accuracy results of different models. As in previous experiments, the QAT model achieves similar accuracy to the high-resolution FPT model but

with a lower resolution.

Table 5.13 – Accuracy results with quantizer bitwidth scheduler on the DVS Gesture dataset. The QAT_4 model achieves similar accuracy than the FPT_8 model and doubles the accuracy compared to the FPT_4 model, highlighting the effectiveness of the QAT training method.

Evaluation Method	FPT_8	FPT_4	QAT_4
Bitwidth	8	4	4
Full Precision	86.32	86.32	-
PTQ simulation	85.61	47.27	84.77
FPGA	83.71	44.14	81.44

It results in lower power consumption, as shown in Table 5.14. Thanks to the lower power consumption with similar accuracy, the R_{pa} ratio decreases and achieves the lowest value with the QAT model.

Table 5.14 – Energy and power consumption of DVS Gesture models. All models perform classification tasks at comparable speed. Since the power consumption of 4-bit models is reduced, the energy consumption of these models decreases. Combining the high accuracy and the low power consumption, the QAT_4 model achieves a better R_{pa} ratio.

Target	FPT_8	FPT_4	QAT_4
Inference Speed (ms)	3.31	3.32	3.31
Total Power (mW)	358	309	309
Energy Consumption (mJ)	1.18	1.03	1.02
R_{pa} (mW/%)	4.28	7.00	3.79

Thank to the bitwidth scheduler, we have been able to train a recurrent model for the DVS Gesture classification task. The QAT model achieves similar accuracy to the model trained in full-precision. However, due to the lower synaptic weight resolution, the power consumption of the QAT model decreases, thereby achieving the best R_{pa} ratio.

5.4.4 Limitations of the Quantization Aware Training Algorithm

In the previous sections, we demonstrated the good behavior of our QAT algorithm implementation. The QAT algorithm performed well, achieving similar accuracy to the FPT algorithm results but with a lower power consumption.
In this section, we will discuss the identified limitations of the QAT implementation and the impact of this training method on model and training performances.

Training Duration

A first limitation of our algorithm is the impact of the QAT algorithm on the calculation duration.

As previously described in Figure 5.20, the QAT algorithm necessitates additional calculations due to the quantization of synaptic weights and membrane voltage. Therefore, we can expect that these additional operations will increase the duration of the training phase.
Table 5.15 references several trained networks on the N-MNIST dataset, showing the average computational duration during the QAT training and the FPT training. The experiments were conducted on the computer previously described in Section 4.1.1 with CUDA activated.
In all cases, the QAT algorithm is slower than the FPT training. It is difficult to identify the parameters that will influence the training duration. This increase in training duration can become crucial if many networks are trained, such as in optimization experiments run by our colleague Thomas Firmin [52], where hundreds of models are trained.

Table 5.15 – Training phase duration comparison between QAT training and FPT training. While QAT training increases the training duration, there is no clear correlation between this increase and the number of layers, the number of neurons, or the number of parameters.

Topology	QAT Duration (s)	FPT Duration (s)	Variation (%)
2312-950-10	93.70	76.24	22.88
2312-392-196-10	90.53	80.52	12.43
2312-1024-512-10	107.61	81.75	31.64
2312-588-392-196-10	103.42	93.43	10.69
2312-256-128-64-10	101.05	88.91	13.66
2312-256-128-64-32-10	111.44	97.16	14.70

The use of the QAT algorithm increases the calculation time during the training phase. This increase is due to quantization of synaptic weights and membrane voltage, which necessitates additional calculations. Although in our experiments, these increases can be considered as not very significant, in some studies, such as optimization tasks, this additional calculation can become important, making this increase a first limitation of our algorithm.

Training Convergence

Another impact of the QAT algorithm is the convergence speed. As we began to explain with N-MNIST in Section 5.4.3, the use of the QAT algorithm can have a significant impact on training convergence. In this section, we will use the training data from the N-MNIST dataset model presented in Section 5.4.2, the SHD model presented in Section 5.4.2, and results on the DVS Gesture model presented in Section 5.4.3.
Figure 5.25 illustrates the two loss functions for each algorithm during N-MNIST training. Due to the significantly lower domain of synaptic weights in QAT compared to FPT, with only 16 possible values, the QAT loss is higher than the FPT loss. However, in the N-MNIST case, both training algorithms converge at a similar speed but with a higher loss value for QAT.

The SHD loss highlights the impact of quantization. Figure 5.26 compares the loss function of both training algorithms. As we observe, the QAT loss is less stable than the FPT loss. A significant observation is the difference in convergence speed. The convergence speed gap between the QAT and FPT algorithms is more predicted in the SHD use case than on N-MNIST. This gap can be attributed to the highly reduced domain of synaptic weight with quantization, combined with the SHD classification task being more difficult than N-MNIST due to the presence of noise.

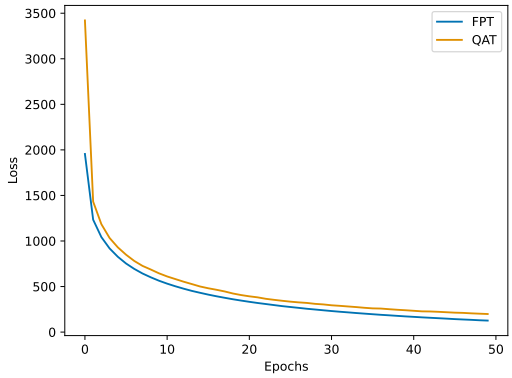

Figure 5.25 – Loss function comparison between the FPT loss and the QAT loss obtained during the N-MNIST models training.

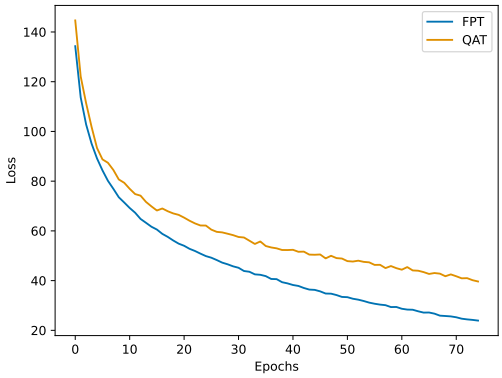

Figure 5.26 – Loss function comparison between the FPT loss and the QAT loss obtained during the SHD model training.

Figure 5.27 compares the accuracy through epochs during the N-MNIST training for both training algorithms. As expected from examining the loss function in Figure 5.25, both accuracys converge at the same speed. However, the final accuracy is lower for the QAT algorithm than the FPT algorithms due to the lower resolutions of synaptic weights.
The accuracy evolution on the SHD dataset, illustrated in Figure 5.28, provides more relevant insights. We can observe that the accuracy converges more slowly during the QAT training than on the FPT training. Specifically, convergence occurs at approximately 20 epochs during FPT training and 35 epochs for the QAT training. Another significant observation is the higher variability of the accuracy during QAT training, which can also be explained by low synaptic weight resolutions.
The two previously studied cases begin to highlight the impact of the QAT algorithm compared to the FPT training method. The DVS Gesture models present a significant impact from the use of the QAT training method.
Figure 5.29 illustrates the evolution of loss during the training, comparing the FPT and QAT algorithms. When the quantizer target bitwidth decreases and reaches 4-bits, the loss increases due to the drop in precision. After 15 epochs, the QAT loss decreases, but at a slower rate than the FPT model. However, after 70 epochs, the QAT loss increases and is stuck in an unstable state until the end of the training. This can be explained firstly by the increase in loss in the FPT training, which is shifted due to bitwidth scheduling. However, contrary to FPT loss, which decreases after this jump, the QAT loss cannot decrease again and remains in an unstable state.
This unstable loss is visible in Figure 5.30, where we observe an accuracy drop from

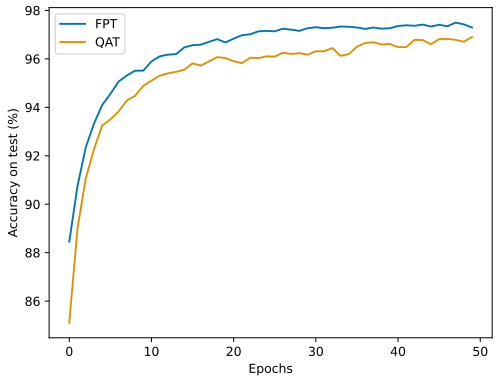

Figure 5.27 – Accuracy evolution on the test set through epochs during the FPT and QAT N-MNIST models trainings.

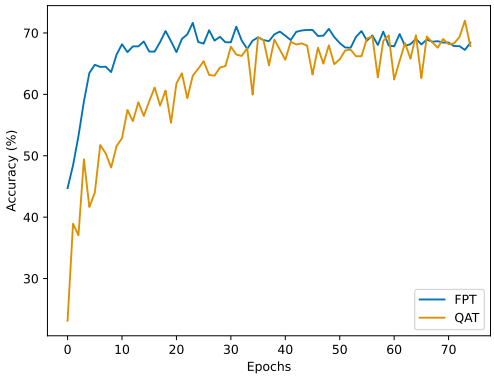

Figure 5.28 – Accuracy evolution on the test set through epochs during the FPT and QAT SHD models trainings.

80% to 15% starting from 70 epochs. Subsequently, the accuracy remains low, under 40%, and is highly variable, a behavior not observed during the FPT run.
The use of the QAT algorithm introduces high variability during the training phase. In some cases, such as with the N-MNIST or SHD dataset, this variability, even though it is present, does not have a significant impact on the final results. However, in the DVS Gesture training, the variability and the difficulty of training stuck the loss into an unstable state until the end of the training.

Quantizer Support

Another major limitation is the quantizer support. Since the beginning of this section dedicated to QAT, we have conducted our experiments with the FPQ quantizer without integer parts. By studying the impact of the quantizer choice for QAT, we reveal a major limitation of our algorithm. We tested other quantizer methods on the N-MNIST and SHD datasets, previously described in Section 5.4.2. In this section, we will focus our studies on DSFQ and MMQ because the FPQ quantizer was previously studied. To highlight the quantizer impact, we will only present the SHD model results, but the N-MNIST results are available on the GitLab repository.
Figure 5.31 represents the loss evolution during the training phase with a logarithmic scale on the ordinate axis with the DSFQ quantizer. A notable observation is the exponential increase of loss from 20 epochs, with the loss value rising from approximately 100 to 10e7. This observation can also be seen in Figure 5.32, which shows a drop in accuracy starting from 20 epochs. This behavior can be explained by two points:

1. The scale factor is determined by the extreme values of synaptic weights. This means that extreme values can take a higher importance than mean values, as we previously explained in Section 5.3.2. During QAT, because we update the scale factor after each epoch, the quantizer will give more importance to extreme values. Consequently, the average values will be quantized to similar values due to the lower bitwidth.
2. As we previously explained in the first point and in Section 5.4.1, we update the quantizer scale factor before weight clamping. Additionally, the change of scale

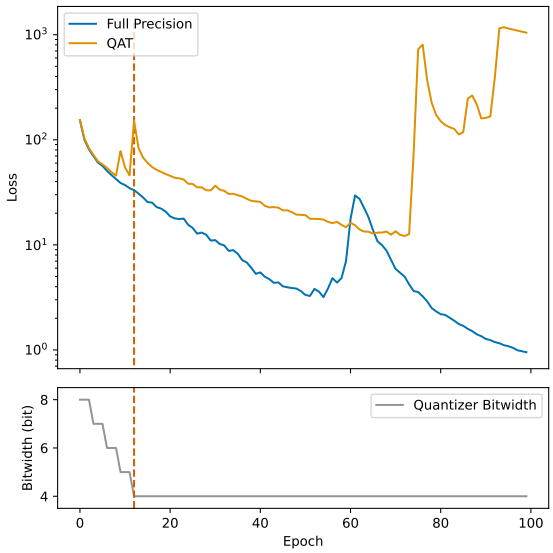

Figure 5.29 – Loss evolution comparison between the FPT and QAT training for the DVS Gesture dataset. The upper graph represents the loss evolution, and the lower graph represents the quantizer bitwidth evolution through epochs.

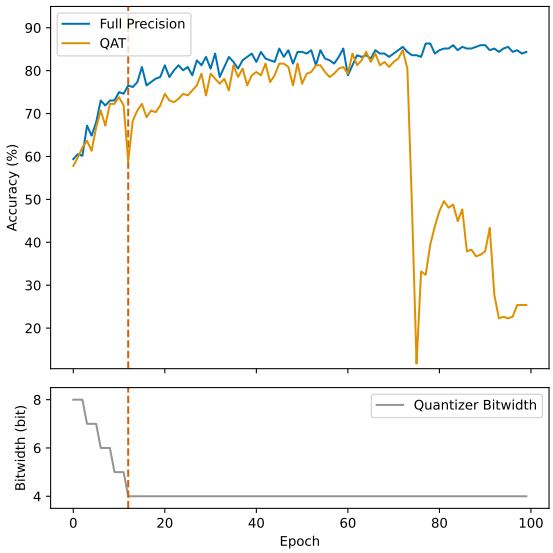

Figure 5.30 – Accuracy evolution comparison between FPT and QAT training for the DVS Gesture dataset. The upper graph represents the accuracy evolution, and the lower graph represents the quantizer bitwidth evolution through epochs.

factor can introduce variability in synaptic weight values, which can introduce variability for the gradient calculation.

The other quantizer that dynamically computes quantizer parameters is the MMQ quantizer. Figure 5.33 represents the loss evolution. As in the previous DSFQ quantizer case, the loss evolution with the MMQ quantizer adopts a curious dynamic from 20 epochs. However, contrary to the previous case, the network stops learning, resulting in a linear loss. This observation is confirmed by Figure 5.34, where accuracy decreases from 5 epochs and reaches the random value, equal to 5% for the SHD dataset. The two previous explanations can be applied to this behavior. However, because the quantizer dynamic is different, it results in a different loss and accuracy evolution.

A potential solution to prevent this degeneration is to fix the synaptic weight range, as it is done with the FPQ quantizer. Because this degeneration is induced by the recalculation of the scale factor and thus the range of the quantizer, we can manually fix a maximal range to avoid the increase of the scale factor. However, due to lack of time, this solution was not implemented.

5.4.5 Conclusion

In this section, we presented the implementation of Quantization Aware Training (QAT) on the ModNEF framework.

In Section 5.4.1, we present the algorithm implemented in the framework, which is similar to Full-Precision Training (FPT) but uses a quantized version of synaptic weight

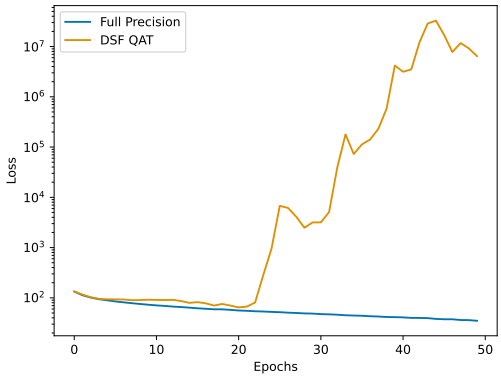

Figure 5.31 – Loss evolution through epochs with a logarithmic scale for the SHD dataset training with the DSFQ quantizer: the blue curve represents the loss evolution during the FPT training, and the orange curve represents the loss of QAT training.

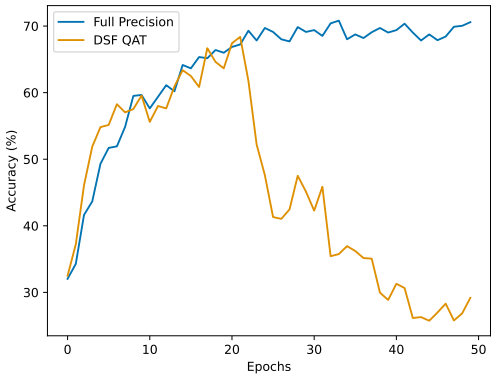

Figure 5.32 – Accuracy evolution through epochs for the SHD dataset training with the DSFQ quantizer. The blue curve shows the evolution of accuracy during the FPT training, and the orange curve shows the accuracy during the QAT training.

used during the forward pass.
In Section 5.4.2, we present N-MNIST and SHD use cases, comparing results between FPT + PTQ and QAT models. Models trained with QAT achieve similar accuracy to high-resolution FPT models. However, thanks to their lower synaptic resolution, the power consumption of the QAT models significantly decreases, making these models the best tradeoff between accuracy and power consumption, as given by the R_{pa} ratio.
To achieve lower quantizer bitwidth, a naive approach is not possible due to the low synaptic weight values of untrained models. To address this issue, we present a Quantizer Scheduler in Section 5.4.3. This scheduler allows reaching lower synaptic weight resolution and thus further reduces the R_{pa} ratio. However, due to low resolution, the presented models begin to show the limitations of the QAT algorithm.
These limitations were presented in Section 5.4.4. A first limitation is the additional computational time, which can become a problem if you want to run hundreds of models, such as in an optimization task. The major limitation of the QAT algorithm is the concern of model convergence. Due to the lower range of synaptic weights, the model can have difficulties to converge, or, as in the DVS Gesture case, will remain stuck in an unstable state until the end of the training phase.

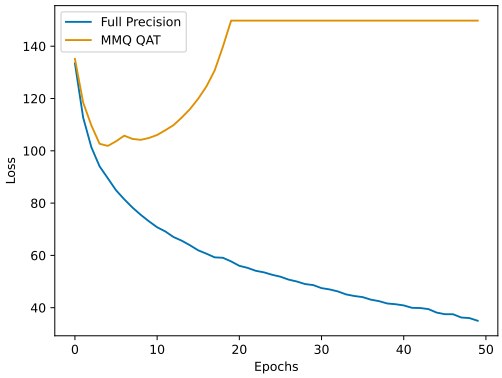

Figure 5.33 – Loss evolution through epochs with a logarithmic scale for the SHD dataset training with MMQ quantizer: the blue curve represents the loss evolution during the FPT training, and the orange curve represents the loss of QAT training.

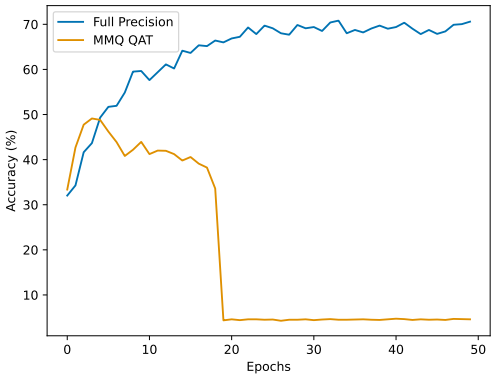

Figure 5.34 – Accuracy evolution through epochs for the SHD dataset training with MMQ quantizer. The blue curve shows the evolution of accuracy during the FPT training, and the orange curve shows the accuracy during the QAT training.

5.5 Conclusion

In this chapter, we investigate the impact of quantization on Spiking Neural Network (SNN) and more particularly on the ModNEF architecture.
In Section 5.1, we present a short theoretical background about quantization. We introduce two important notions of this chapter: the Post-Training Quantization (PTQ), where a quantization function is applied before the training, and the Quantization Aware Training (QAT), where we consider the quantization during the training.
In the ModNEF framework, we implement three different quantization methods, presented in Section 5.2. In ModNEF, we implement three quantization methods:

1. Fixed-Point Quantizer (FPQ)
2. Dynamic Scale Factor Quantizer (DSFQ)
3. Min-Max Quantizer (MMQ)

In Section 5.3, we studied the impact in post-quantization of our different quantizers. We identified several limitations of our quantizers, particularly when the quantizer parameters are calculated depending on synaptic weight values and not fixed. This method gives more importance to the extreme and rare values, increasing the quantization impact on network performance and resulting in a high drop of accuracy.
To mitigate the accuracy drop, we presented our implementation of the QAT algorithm on the ModNEF framework in Section 5.4. Thanks to the QAT algorithm, we achieve similar accuracy to models trained in full-precision. However, due to the lower synaptic weight resolution, the power consumption of the QAT models is, in general, lower, resulting in a better tradeoff between power consumption and accuracy.
To identify the tradeoff, we introduce a new metric called Ratio Power Accuracy (R_{pa}), which indicates the power consumed by the architecture to reach 1% in accuracy. This

metric can be used to compare similar models and identify the model with the best tradeoff between power consumption and accuracy.
In this chapter, we present the quantization tools implemented in ModNEF and we demonstrate their impact on PTQ and QAT models. We can achieve good results with both methods, each with their limitations. We also highlight the limitations of our quantizers, especially quantizers where internal parameters are calculated dynamically. More work is thus necessary to propose better quantizers that decrease the quantization impact on models.

6 ULP Cochlea: a ModNEF Application for Bio-Diversity Monitoring

In the previous chapters, we introduced our architecture, ModNEF. We demonstrated its behavior with well-known datasets such as MNIST, N-MNIST, SHD, and DVS Gesture, demonstrating the capability of ModNEF to perform SNN inferences for established classification tasks. However, as outlined in our motivations, ModNEF can be used in embedded classification tasks.
In this chapter, we will present an application of the ModNEF architecture and framework for an embedded classification task within the ULP Cochlea project. The primary objective of this project is to monitor biodiversity in the Mediterranean Sea, with a particular focus on sperm whales. ModNEF will be utilized to classify, in real time, the various sounds in the sea, that are transformed into spikes using an analog cochlea.
In Section 6.1, we will present the ULP Cochlea project, providing an overview of the project's goals, a description of the dataset and a brief introduction of the developed analog cochlea sensor. Then, in Section 6.2, we will present the networks we trained for this project and the initial results of FPGA implementation. Finally, in Section 6.3, we will describe the integration of the ModNEF architecture for a fully embedded system, from sensory systems to data processing.

6.1 ULP Cochlea Project Presentation

In this section, we will present the ULP Cochlea project and the role of ModNEF within this project. First, in Section 6.1.1, we will introduce the ULP Cochlea project. Following that, in Section 6.1.2, we will describe the analog cochlea sensor. Then, in Section 6.1.3, we will present the datasets generated with this cochlea. Finally, in Section 6.1.4, we will explain the role of ModNEF in this project and within the embedded system.

6.1.1 Project Presentation

The ULP Cochlea project is an interdisciplinary project involving IEMN, CRIStAL, and LIS laboratories. It involves the development of a fully bio-inspired System On Chip (SOC) from sensor to data processing with ultra-low power consumption (ranging from 1 to 10 mW). Neuromorphic computing aims to minimize the environmental footprint of sperm whale monitoring in the Mediterranean Sea. Currently, sensor systems are based on active microcontroller SOCs that consume significant power, necessitating regular battery changes and, consequently, frequent boat trips to the monitoring sites. Bio-inspired computing methods appear as a viable solution to substantially decrease the power consumption of SOC, thereby reducing the frequency of battery replacement operation.
The ULP Cochlea project is centered around four main objectives :

1. **Fully Analog Bio-inspired Cochlea**: The first objective is to develop an analog bio-inspired cochlea. This device will convert an input analog audio signal into a spike representation by using CMOS subthreshold technology.

2. **Develop an AI processing chip**: To process the spikes from analog cochlea, it is essential to develop an ultra-low-power (ranging from 1 mW to 10 mW) AI processing chip. This chip will implement an SNN to detect the presence of sperm whales.
3. **Audio Events Detection and Localization**: The two previously described tools will be used for audio event detection and localization. These events can be sea species, such as sperm whales in our case, or boat detection to help preserve ecosystems.
4. **Provide an Overview of the Acoustic Environment**: By using the cochlea, we aim to provide an overview of the acoustic environment on Mediterranean Seas.

The IEMN laboratory will be involved in developing the analog cochlea sensor, while the LIS laboratory will provide data from existing sensory systems. In our case, the 2XS team from the CRIStAL laboratory, we focuses on developing SNN models for sperm whale classification and providing an AI chip for embedded classification tasks.

6.1.2 Cochlea Presentation

In the previous section, we presented the main objectives of the ULP Cochlea project. In this section, we will present the analog cochlea developed by the IEMN team.
Inspired by the functioning of the human cochlea, the analog cochlea consists of a bank of 16 channels. Each channel comprises a low-pass filter with a specific cutoff frequency, depending on the channel's position. The high-frequency component, extracted by the current channel, is sent to a half-wave rectifier, while the remaining low-frequency component is passed to the next stage of the cascade filter. The rectifier converts the Alternative Current (AC) signal to a Direct Current (DC) signal, which is then sent to an analog LIF neuron model to generate output spikes. The architecture of the cochlea is presented in slide 12 of the slideshow [40] and in Figure 6.1. The cochlea was fabricated using CMOS TMSC 65 nm technology.
The final cochlea system is decomposed of two different cochlea circuits, each with distinct channel cutoff frequency. The first cochlea covers a frequency range from 90 to 12 kHz, while the second cochlea spans from 9 to 1 kHz. This separation was implemented to avoid stability problems. It was decided to divide the cochlea into two sub-cochlea to address stability concerns to avoid, but also because an 8-channel cochlea is easier to characterize than a 16-channel cochlea. The output of the first cochlea filter bank is connected to the input of the second cochlea, resulting in a 16-channel output cochlea.

6.1.3 Datasets Presentation

Two datasets were generated with the analog cochlea: a 2-classes dataset and a 10-classes dataset.
Both datasets were generated using the same method. The cochlea was placed under probes. An input audio WAV file was sent to the cochlea sensor. The output spikes are converted into 0/1 V signals, where 1 V represents an output spike. These binary signals were recorded using an oscilloscope and saved into CSV files at an 800 ns time resolution.
The first created dataset consists of a 2-classes dataset with a "sperm whale" class and a "non-sperm whale" class. Each sample lasts 10 ms, with an 800 ns sampling

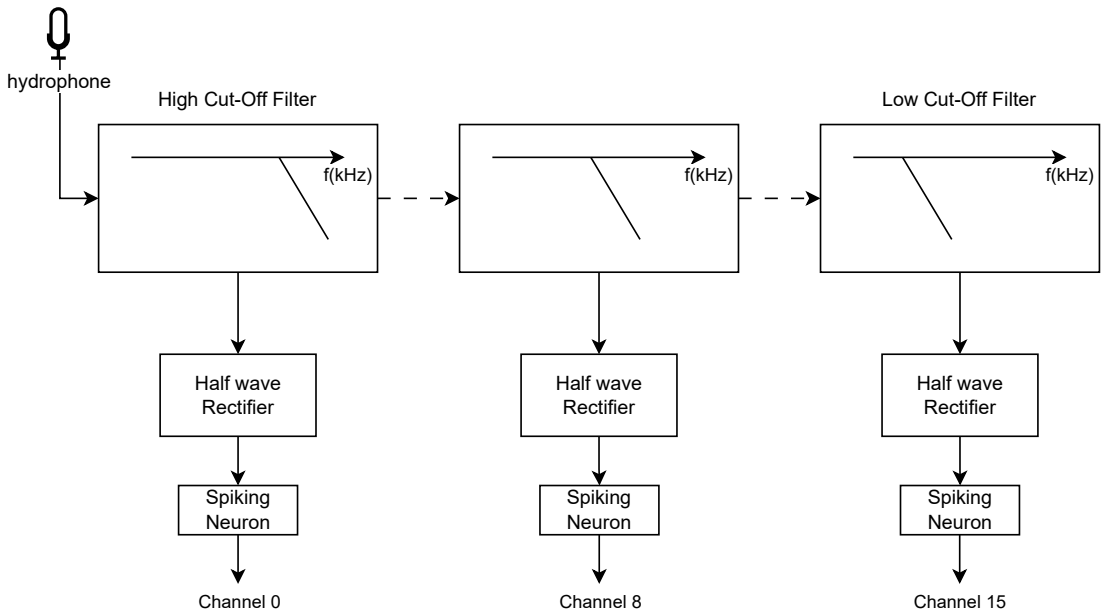

Figure 6.1 – Schematic representation of the cochlea circuit inspired by [40]. The input signal is provided by a hydrophone. Each channel is composed of a low-pass filter, a half-wave rectifier, and a spiking neuron based on the LIF neuron model. The residual signal of a filter is connected to the next stage of the filter bank.

rate, resulting in 12500 points for each sample. The input audio files are sourced from animal documentaries and are distributed as follows: 500 samples for the sperm whale class and 500 samples for the non-sperm whale class. This results in a small dataset of only 1,000 samples with a random test-train split, which can lead to variation in accuracy with each run. This dataset is so limited by its size and the dataset handler provided, but several studies will be conducted to optimize the dataset.
The second dataset is a 10-classes dataset based on the Watkins Marine Mammal Sound Dataset [190, 160, 191]. The initial dataset consists of 15,000 samples generated from 2,000 records of more than 60 species. In our case, we select only 10 species:

1. White Whale
2. False Killer Whale
3. Atlantic Spotted Dolphin
4. Bowhead Whale
5. Clymene Dolphin
6. Long-Finned Pilot Whale
7. Melon Headed Whale
8. Northern Right Whale
9. Ross Seal
10. Sperm Whale

The generated dataset consists of 9,500 samples, with 7,611 samples in the train set and 1,908 samples in the test set. The output CSV files are converted into the Hierarchical

Data Format 5 (HDF5) format, which is also used for other spiking datasets such as SHD and SSC dataset [38].
Because the last dataset was developed and presented toward the end of the thesis, in July 2025, we will mostly focus our works on the 2-classes dataset.

6.1.4 Role of ModNEF in the Project

In the previous section, we have presented the ULP Cochlea project, the dataset created, and the analog cochlea used to generate spikes from input audio.
One of the goals of the project is to provide a fully neuromorphic SOC with ultra-low power consumption, ranging between 1 and 10 mW. However, due to the complexity of developing analog SNN, we decided to establish an intermediate goal. This involves creating a low-power SNN implementation that will activate a more power-hungry, yet more powerful, processing unit capable of handling more complex operations. This intermediate circuit is represented in Figure 6.2. The purpose of the SNN is to trigger a wake-up signal that will activate a more powerful and power-hungry processing unit, which will execute more complex sound processing tasks for specimen identification or echolocation.

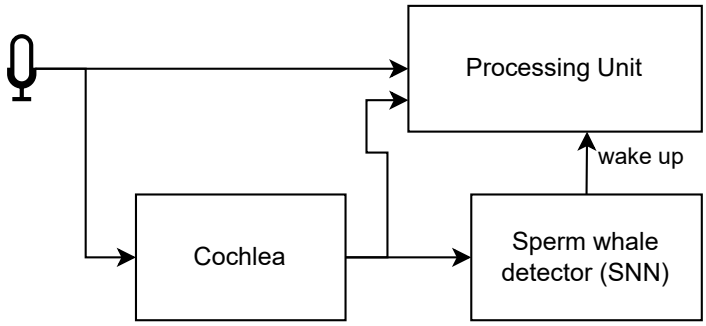

Figure 6.2 – Schematic representation of the ULP Cochlea SOC. The hydrophone is connected to the cochlea, which is connected to the SNN chip. The SNN triggers wake-up signals to activate the processing unit, which then performs more complex tasks.

To minimize the power consumption, the SNN must be implemented on a low-power architecture. A first approach is to use a neuromorphic chip such as Xylo [21], a neuromorphic chip dedicated to audio processing. Another intermediate solution is to use an FPGA, which provides a good tradeoff between power consumption and computational capabilities.
As an intermediate solution, the use of FPGA has been chosen for several reasons :

- Analog SNN is very complex and long to develop.
- Currently, our research team has no neuromorphic chip for embedded classification.
- ModNEF was already developed and has proven its capability to perform classification tasks. Further, due to the high number of I/O on the FPGA, it is possible to connect the output of the cochlea to the FPGA with low additional circuitry.

The other solutions have not been abandoned and will be explored in future works by our research team. However, the FPGA solution with the ModNEF architecture appears

to be a good initial solution.

6.2 Trained Networks and First Implementations

In the previous section, we presented the ULP Cochlea project. We presented the analog cochlea developed by the IEMN research team and the two datasets created with this cochlea. The ModNEF architecture will be used as the initial solution for executing classification tasks within the embedded SOC by triggering a wake-up signal to activate a more powerful, yet power-hungry, processing unit capable of performing more complex tasks.
In this section, we will present the various networks trained using the 2-classes dataset. As previously explained, the 10-classes dataset has been developed too recently to propose a complete study. Moreover, the 2-classes dataset appears to be the most suitable dataset for the initial role of ModNEF, which involves detecting sperm whales and triggering a wake-up signal.
In Section 6.2.1, we will present an initial network topology and dataset transformation exploration conducted by our colleague Mazdack Fatahi. Building on this preliminary study, we will present the trained networks with the ModNEF framework in Section 6.2.2. Finally, in Section 6.2.3, we will discuss the inference results on various hardware targets.

6.2.1 Network Exploration

An initial exploration of network topology and dataset transformation was conducted by our colleague Mazdack Fatahi. This work served as a baseline to select network topology and dataset transformation for training the ModNEF-based models. In this section, we will present the results of this exploration.
The exploration focused on three key points:

1. **Network Topology**: Four different network topologies were explored, described by the following list. The classification is based on the spike rate of the output neuron. If the output neuron emits more than 0.5 spikes per time step, the network detects a sperm whale.
   — 16-1
   — 16-32-1
   — 16-64-1
   — 16-128-1
2. **Number of Time Step**: The number of time steps was also explored through six different values: 1, 2, 4, 5, 10, and 20 time steps. The number of time steps represents the number of frames and can be compared to the `n_time_bins` parameter of the tonic frame transformation.
3. **Binary Transformation**: The binary transformation refers to a transformation where the number of spikes is ignored, as shown in Figure 6.3. If, at least, one spike is accumulated during the time step, the binary transformation will consider only 1 spike in that time step. The impact of this transformation was examined during the exploration.

Figure 6.4 summarizes the results of the exploration. The upper graph illustrates the accuracy based on the network topology and the number of time steps without binary

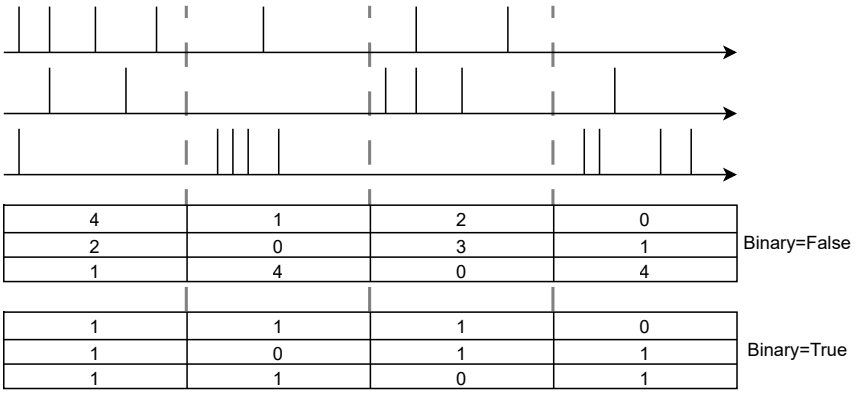

Figure 6.3 – Representation of Binary Transformation. The first graph illustrates the input spikes over time, with the frames delimited by dashed gray lines. The first table represents the spike transformation without binary transformation, where the values represent the exact number of spikes accumulated during the time window. In contrast, the lower table shows the spike transformation with binarization. In this case, if at least one spike has been detected within the time window, the number of spikes is always equal to 1, regardless of the actual number of input spikes.

transformation. In contrast, the lower graph represents the accuracies with binary transformation.

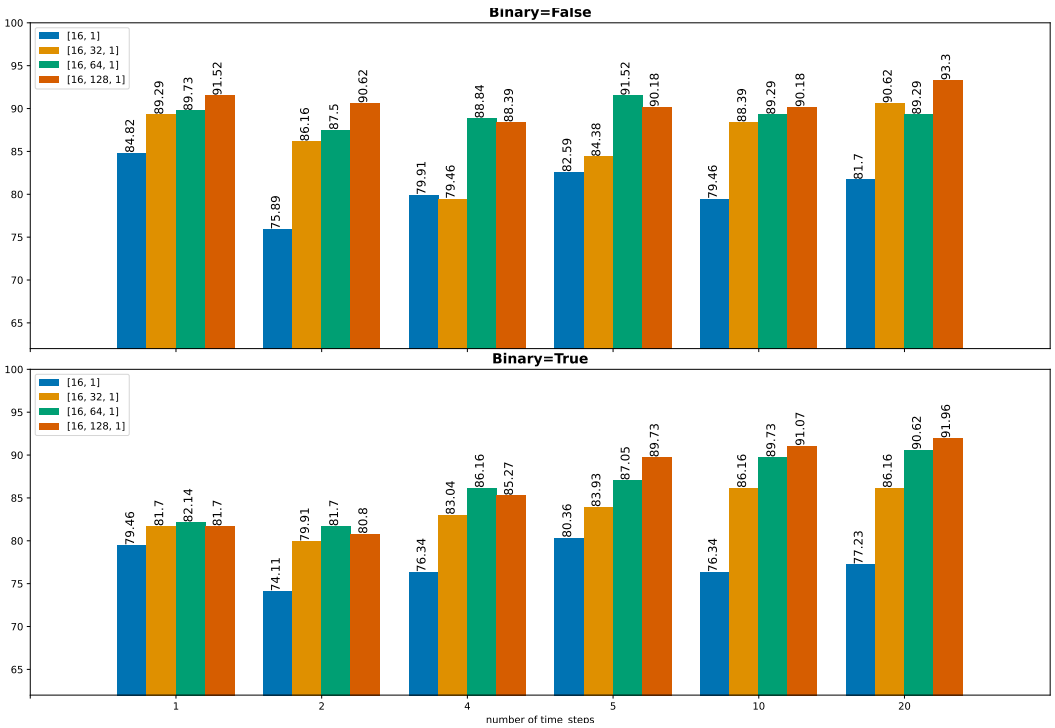

Figure 6.4 – Results of the exploration phase for the 2-classes ULP Cochlea dataset. The upper graph illustrates the accuracy based on network topology and the number of time steps without binary transformation. The lower graph represents the accuracy of the different networks for various time steps with the application of the binary transformation.

A initial observation is the negative impact of the binary transformation. In most cases, applying the binary transformation decreases the accuracy for a given network topology and number of time steps. This can be explained by the loss of information that occurs when the binary transformation is applied.

A second observation pertains to the impact of the network topology. In most cases, the accuracy increases as the number of neurons increases. Notably, there is a significant gain in accuracy for two-layer networks compared to the single-neuron network.

The impact of the number of time steps is not as clear as the previously explored parameters. With binary transformation, we can see a tendency for accuracy to increase as the number of time steps increases. This is likely due to the significant loss of information caused by the combination of a low number of time steps and binary transformation. However, the accuracy gain without binary transformation is not as evident as for models trained with binary transformation. Depending on the network topology, the optimal accuracy is achieved at different numbers of time steps.

Table 6.1 summarizes the best accuracy for each network topology. At first glance, one might infer that smaller networks reach the best accuracy with a lower number of time steps, whereas larger networks require a higher number of time steps. However, examining Figure 6.4 reveals that the evolution of accuracy depending on the time steps is not significant and remains relatively stable.

This initial study highlights several key points for the continuation of our work. Firstly, the binary transformation significantly impacts the accuracy of the networks, with the best accuracy achieved without its application. Generally, larger network topologies

Table 6.1 – Best accuracy for each network topology and the associated number of time steps without binary transformation. The number of time steps does not directly correlate with the accuracy of the network. For smaller networks, the best accuracy is achieved for low time steps. In contrast, the two largest reach the best accuracy with the highest number of time steps.

Network Topology	Accuracy	Time Step
16-1	84.82	1
16-32-1	91.52	5
16-64-1	90.62	20
16-128-1	93.3	20

achieve higher accuracy, with a notable difference between single-layer networks and two-layer networks. The number of time steps does not have a significant impact on the accuracy.
It is important to note that these models were trained using the SNNTorch neuron model and are therefore incompatible with the ModNEF architecture. In the next section, we will present the selected parameters and the networks we trained with the ModNEF framework.

6.2.2 ModNEF-based Models

In the previous section, we presented initial network topologies and dataset transformations conducted by our colleague Mazdack Fatahi. Based on this experiment, we will now present the selected parameters.
All models and results are available on the following GitLab: `https://gitlab.univ-lille.fr/bioinsp/ulp_cochlea_fpga`.
All models were trained over 5,000 epochs using the `BCEWithLogitsLoss` loss function optimized for binary classification. The dataset was split with 750 samples for the training set and 250 samples for the test set, with a fixed seed to maintain consistent training and test sets across all experiments.
Table 6.2 summarizes the model trained with the ModNEF framework. To maximize the accuracy, we decide not to use the binary transformation.
Most models are based on the same exploration parameters, utilizing the 16-128-1 network topology and 20 time steps. Initially, a model was trained with the FPT training algorithm. However, as we will discuss in the results section, the drop in accuracy was significant. Therefore, we decided to train our models with the QAT algorithm.
We trained two models with this network topology using the QAT algorithm, each with two different quantization target bitwidth. The first model was trained with 8 bits, a conventional value for bitwidth. The second model was trained with the lowest bitwidth without a significant drop in accuracy. However, due to the small size of the dataset, the model trained with lower target bitwidth did not achieve sufficient accuracy, converging to 88% of accuracy compared to 93% for the initial model.
Two additional models were trained with only one time step for two distinct reasons. The initial reason was to utilize the single neuron model to minimize the hardware impact of the FPGA implementation. Since the best accuracy for this model is achieved

Table 6.2 – Networks trained for the 2-classes ULP Cochlea dataset using the ModNEF framework. Most models are trained using the QAT algorithm and are based on the same exploration parameters. Two models were specifically trained with single-time steps to utilize the IF neuron model.

Name	Topology	Binary	Num Steps	Training	Neuron	Weight bitwidth
FPT_16_128_1_s20	16-128-1	False	20	FPT	SRLIF	10
QAT_16_128_1_w8_s20	16-128-1	False	20	QAT	SRLIF	8
QAT_16_128_1_w7_s20	16-128-1	False	20	QAT	SRLIF	7
QAT_16_128_1_s1	16-128-1	False	1	QAT	IF	8
QAT_16_1_s1	16-1	False	1	QAT	IF	10

with one time step, we selected this configuration to maximize the network's accuracy.
The second reason was the use of the IF neuron model. As there is no need for a membrane decay mechanism, we could employ the IF neuron model, which is the simplest neuron model. Therefore, the use of a single frame was driven by hardware optimization, selecting the best network topology, and dataset transformation.

6.2.3 Inference Results

In the previous section, we presented the different models trained with the ModNEF framework and the rationale behind the selection of these model parameters, specifically in terms of network topology and dataset transformation.
In this section, we will present the results of our model inference across different hardware targets. As with previous experiments, we will discuss the results obtained on the computer described in Section 4.1.1 with CUDA activated, which will be referred as GPU target for the reminder of this section. The second hardware target is the FPGA Arty-z7 20 board with UART communication.
In addition to these hardware targets, we will also present results on a RasTech Raspberry Pi 5 equipped with an ARM Cortex-A7 processor running at 2.4GHz with 8 GB of RAM. Our objective is to compare the FPGA inference with the inference on an embedded board. Due to time constraints and the lack of availability of other embedded boards within our research team, we selected a Raspberry Pi board as an embedded hardware target.
Table 6.3 summarizes the different accuracys depending on the hardware target. For software accuracy on the GPU and Raspi hardware targets, we present the accuracy with PTQ. The initial value corresponds to the accuracy of models trained during the exploration experiments.
The FPT_16_128_1_s20 model achieves low accuracy due to quantization effects. Without quantization, the model achieves 91.66% accuracy. However, even with 10-bit synaptic weight resolution, the impact of quantization significantly degrades the model's performance.
To facilitate the readability of the following tables, we will not present further results for this model. Nevertheless, all results are available on the GitLab repository for ref-

Table 6.3 – Accuracy of the ModNEF-based models for the 2-classes ULP Cochlea dataset. The "Initial" accuracy refers to the accuracy obtained during the initial exploration. The GPU hardware target refers to the laptop GPU, Raspi refers to the Raspberry Pi 5, and FPGA refers to the Arty-Z7 20.

Model Name	Initial	GPU	Raspi	FPGA
FPT_16_128_1_s20	93.3	77.60	75.52	77.60
QAT_16_128_1_w8_s20	93.3	91.15	92.71	91.15
QAT_16_128_1_w7_s20	93.3	90.10	90.63	89.58
QAT_16_128_1_s1	91.52	89.06	88.54	88.54
QAT_16_1_s1	84.82	84.90	80.73	84.38

erence.

The corresponding networks trained with the QAT algorithm, QAT_16_128_1_w8_s20 and QAT_16_128_1_w7_s20 models, achieve similar accuracy than the FPT_16_128_1_s20 full precision model. Thanks to the QAT training, the accuracy of these models remains high, close to the initial accuracy. The model trained with 7-bit quantizer bitwidth achieves lower accuracy than the 8-bit model. Additional results are necessary to identify the best model between these two networks.

Another model that achieves accuracy around 90% is the QAT_16_128_1_s1 model, although the accuracy drop starts to become significant on Raspi and FPGA targets. The other single time step model, QAT_16_1_s1, achieves lower accuracy compared to other models but maintains similar accuracy to the initial model trained with SNNTorch.

A notable observation is the difference in accuracy between the GPU and the Raspi targets. It is important to notice that we fixed a common seed for all random operations and set the CUDA backend deterministic. This accuracy difference between the GPU and Raspi cannot be explained by the randomness of operations. This difference can be explained by three main points:

— Operations and floating values representation, as well as their hardware implementations, can differ between CPU and GPU, which can significantly affect the inference. Further, the CPU of the Raspberry Pi is an ARM processor, which may implement floating-point operations differently than the Intel processor and Nvidia GPUs.
— The version of the PyTorch library differs between the laptop and GPU target and the Raspberry Pi target, which could explain the computational difference observed.
— The generated seed used for random split is different between the two targets. Because the dataset is small and networks are highly sensitive, it can result in different sample processing, resulting in different accuracy.

This accuracy gap between these two hardware targets remains acceptable, ranging from 0.10% to 1.5%; expect for the QAT_16_1_s1. This model exhibits a 4% accuracy gap between GPU and Raspi targets, which becomes significant.

Table 6.4 summarizes the inference time, in milliseconds, of the different models depending on the hardware target.

Table 6.4 – Inference time, in milliseconds, of the models trained with the ModNEF framework for the 2-classes ULP Cochlea dataset. In most cases, the slowest hardware target is the GPU, followed by Raspi and FPGA targets. The only exception is for the smallest model, where the Raspi achieves faster inference times than the FPGA target.

Model Name	GPU	Raspi	FPGA
`QAT_16_128_1_w8_s20`	1.91	1.08	0.049
`QAT_16_128_1_w7_s20`	1.90	1.09	0.049
`QAT_16_128_1_s1`	1.75	0.066	0.048
`QAT_16_1_s1`	1.77	0.016	0.046

In most cases, the GPU is the slowest hardware target, followed by the Raspi target. The FPGA remains the fastest hardware target. However, the inference speed of the `QAT_16_1_s1` model on the Raspi target is 4 times faster than the FPGA inference. This can be explained by the combination of the small network size, the low number of time steps, and the use of an optimized library. In the FPGA, the spike transmission between the UART and the layer modules is a time-consuming operation and can slow down the inference speed.

The parameters influencing the inference time vary depending on the hardware target. For the GPU target, the inference time is highly influenced by the number of time steps rather than the network size, due to the parallelism of model inference on the GPU. For the Raspi target, because of sequential execution, both the number of time steps and the network topology significantly influence the inference time. Finally, the inference time for the FPGA remains stable, primarily due to the time required for sending the input spikes to the layers, which is a time-consuming operation because of the high number of input spikes.

Table 6.5 references the energy consumption of the different models depending on the hardware target. The energy estimation for the GPU target was obtained using the PyJoule Python library [22]. The power consumption of the Raspi target was determined by measuring the power consumption of the Raspberry Pi using the `vcgencmd -pmic_read_adc` command line. Then, we monitored the execution time of the inference with the `process_time` Python function, more precise than the `time` function, and applied the equation $E = P \cdot t$. The energy consumption of the FPGA target was obtained with the vectorless Vivado power report and the measured inference time.

As expected, the GPU hardware target is the most energy-intensive. Moreover, the energy consumption does not significantly decrease with smaller network topologies. In contrast, the energy consumption of the Raspi target decreases depending on the network size and the number of time steps. The Raspi target achieves inference with better energy efficiency, exhibiting 14 to 900 times lower energy consumption than the GPU hardware target.

The FPGA demonstrates the best performance in terms of energy consumption, ranging from 7 to 230 better than the Raspi energy consumption. Since the inference time, referenced in Table 6.4, does not vary significantly depending on the models, the reduction in energy consumption is primarily due to decreased power consumption. The power consumption decreases with the smaller network topologies, explained by the reduction of synaptic resolution and the change of the neuron model from the SRLIF

Table 6.5 – Energy consumption, in millijoules, of the different ModNEF-based models for the 2-classes ULP Cochlea dataset. The GPU target is the most energy-consuming, with an increase from 14 to 900 times compared to the Raspi hardware target. The FPGA target remains the best hardware target in terms of energy efficiency.

Model Name	GPU	Raspi	FPGA
QAT_16_128_1_w8_s20	42.04	2.84	0.012
QAT_16_128_1_w7_s20	42.66	2.86	0.010
QAT_16_128_1_s1	39.00	0.17	0.008
QAT_16_1_s1	39.61	0.043	0.006

model to the IF model.
We present the different metrics depending on the model and on the hardware target. To determine the best model, we can also consider the R_{pa} metric, which represents the power (in mW) consumed to achieve 1% accuracy. The R_{pa} ratio is presented in Table 6.6.

Table 6.6 – R_{pa} ratio depending on the models and the hardware target for the 2-classes ULP Cochlea dataset. The ratio remains relatively stable for the GPU and the Raspi targets. However, a significant decrease from 2.61 to 1.56 can be observed for the FPGA hardware target.

Model Name	GPU	Raspi	FPGA
QAT_16_128_1_w8_s20	240.97	28.25	2.61
QAT_16_128_1_w7_s20	248.83	28.91	2.17
QAT_16_128_1_s1	250.13	29.51	1.89
QAT_16_1_s1	262.40	32.90	1.56

As expected from the previous tables, the R_{pa} of the GPU hardware target exhibits the worst R_{pa} ratio, with approximately 250 mW/% for all models. The R_{pa}s ratios of the Raspi models are 10 times better than those of the GPU.
The best R_{pa} values are achieved with the FPGA hardware target, which performs approximately 10 times better than the Raspi R_{pa}s. The lowest R_{pa} is achieved with the QAT_16_1_s1 model. Even though this model achieves the lowest accuracy, the network is small, and the power consumption of this model decreases sufficiently to significantly impact the R_{pa} ratio.
To determine the best model and hardware target, we can base our selection on the three following metrics:

1. **Accuracy**: The accuracy directly reflects the performance of the model. The accuracy is to be maximized.
2. **Energy Consumption**: A lower energy consumption is preferable for low-power embedded SOC.
3. R_{pa} **ratio**: The R_{pa} ratio represents the power consumed to reach 1% f accuracy. A lower R_{pa} indicates better power usage.

The inference time was not considered since all models on all hardware targets achieve real-time inference.
Depending on these specified criteria, the best models for each metric are as follows:

— **Accuracy**: According to Table 6.3, the best model in terms of accuracy is the `QAT_16_128_1_w8_s20` on Raspi hardware target.
— **Energy consumption**: According to Table 6.5, the best model in terms of energy consumption is the `QAT_16_1_s1` on Raspi target.
— R_{pa} **ratio**: According to Table 6.6, the best model in terms of R_{pa} ratio is the `QAT_16_1_s1` on the FPGA hardware target.

If we want to find the best model depending on all these three metrics, we can calculate a composite score based on these three metrics. Firstly, we normalize the values with Equation 6.1. Then, using Equation 6.2, we calculate the model score. In our case, we give the same importance to all metrics.

$$X_{norm} = \frac{X - X_{min}}{X_{max} - X_{max}} \tag{6.1}$$

$$Score = \frac{Acc_{norm} + 1 - E_{norm} + 1 - Rpa_{norm}}{3} \tag{6.2}$$

According to the composite score, the three best models are summarized in Table 6.7. The best model is the `QAT_16_128_1_w8_s20` model on FPGA with a score of 0.95. This is followed closely by the same network on the Raspi hardware target with a score of 0.94. The third best model is the `QAT_16_128_1_w7_s20` model on the FPGA target. Additional score values are available in the GitLab repository.

Table 6.7 – Best models and hardware target according to the composite score. The top model is the `QAT_16_128_1_w8_s20` running on FPGA, due to the high accuracy, low-energy consumption, and R_{pa} ratio. The same model running on Raspberry Pi achieves a similar score, thanks to its highest accuracy, even though its energy consumption and R_{pa} metrics are higher. Finally, the `QAT_16_128_1_w7_s20` model is the third best model, thanks to its very low-energy consumption and R_{pa}, which mitigates its lower accuracy score.

Model	Hardware target	Acc_{norm}	E_{norm}	Rpa_{norm}	Score
`QAT_16_128_1_w8_s20`	FPGA	0.87	0.0001	0.009	0.95
`QAT_16_128_1_w8_s20`	Raspi	1.0	0.066	0.11	0.94
`QAT_16_128_1_w7_s20`	FPGA	0.74	9.4e-5	0.008	0.91

This scoring method can be critiqued for assigning the same importance to the three metrics. Further discussions are necessary to determine the best composite score by adjusting the weights of each metric according to their importance in the projects.

6.2.4 Conclusion

In this section, we present the network topology and the dataset transformation used for FPGA implementation of the sperm whale detection network.
An initial study conducted by our colleague Mazdack Fatahi was used to determine the best configuration for our experiments. In terms of network topology, the best topology identified is the 16-128-1. However, the 16-1 topology also emerged as an

interesting option due to its simplicity, which can lead to a lower power consumption architecture. This study also examined the dataset transformations. It was found that binary transformation negatively impacts accuracy. Additionally, the number of time steps does not significantly impact the accuracy, although the best accuracies are achieved with 20 time steps.
Based on this study, we decided to train several models using the ModNEF framework. Since the models trained with full precision achieved lower accuracy due to quantization effects, we focused our study on models trained with the QAT method. Additionally, we compared models across three different hardware targets: GPU, Raspberry Pi, and FPGA targets.
Depending on the metrics considered, the optimal configuration may vary. We propose a composite score considering the accuracy, energy consumption, and R_{pa} ratio. This approach aims to identify the best trade-off among all models.
Our experiments serve as a good starting point for the SOC deployment but remain artificial. Specifically, the input samples are loaded from a dataset and transmitted to the network in a manner that may not be realistic when considering the final SOC architecture. In the next section, we will present a more realistic SOC architecture.

6.3 From Sensor to FPGA : Pipeline Implementations

In the previous section, we presented a first implementation of models trained with the ModNEF framework. Implemented models achieved good accuracy and low-energy consumption, making the FPGA architecture a promising method for low-power wake-up engines. However, a major limitation of our previous work is the unrealistic method to transmit input spikes. Samples were loaded from a dataset with a Python library and sent to the FPGA with the UART communication protocol. This data reception interface serves as a good starting point but remains unsuitable for a SOC deployment.
In this section, we will present a SOC-compatible architecture based on the previously trained models integrated into an architecture suitable for SOC integration.
Figure 6.5 illustrates the main concept of the connection between the analog cochlea and the FPGA, which will execute the classification model based on the ModNEF architecture. The output of cochlea, ranging from 0 to 1V, must be amplified to 3.3V to be recognized by the FPGA General Purpose Input/Output (GPIO). The input signal is sent to spike detectors, which detect input spikes and transmit them to the ModNEF models. The ModNEF model will trigger the wake-up signal if it detects a sperm whale.
In section 6.3.1, we will present the model implementations and the additional components developed for the SOC integration. Finally, in Section 6.3.2, we will present the inference results obtained with two different levels of SOC simulation.

6.3.1 Pipeline FPGA Architecture

In this section, we will present the implementation of SOC-based models and the additional components developed.
Before studying the network implementation and the newly developed components, it is important to explain the role of the UART module in classical architecture. The role of the UART component can be summarized by the following points:

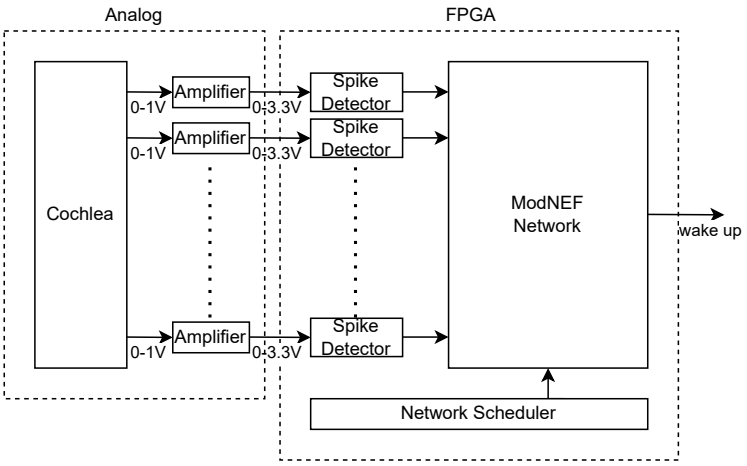

Figure 6.5 – Schematic representation of the connection between the analog cochlea and the FPGA. The output of the cochlea, which ranges from 0 to 1 V, must be amplified to be detected by the FPGA GPIO. The input signal is sent to the spike detector, which will detect input spikes and send these to the ModNEF network.

— **Input spike reception**: The module will receive input spikes from the computer and will transmit them to the network.
— **Network Scheduler**: After receiving input spikes from the computer, the UART module will schedule the network by detecting emulation steps and triggering the *start_emu* signal, which indicates to the layer modules when they must update their internal state.
— **Classification**: The UART component receives the spikes from the output layer and will transmit them to the computer, which runs the classification process with the PyTorch library.

To bypass the usage of the UART component, it is necessary to implement components that replace the role of the UART component.
Based on the different trained models, we can define two different model architectures.
Figure 6.6 illustrates the network implementation for 2-Layer-based models.
In these architectures, the input spikes from the spike detectors are accumulated in the accumulator component, which represents the input layer previously emulated by the UART component. The output layer is connected to both the classifier and the early stop detector. The early stop detector halts the network update executions based on several conditions to save energy. However, the early stop detector is not implemented for the `QAT_16_128_1_s1` model due to its single emulation step.
In contrast to the 2-Layer models, a specific architecture was developed for the `QAT_16_1_s1` model illustrated by Figure 6.7.
Due to its simplicity, the network is implemented without the ModNEF architecture. The network consists of a single neuron component and is scheduled by a pulse generator. The neuron is composed by a bank of synaptic weight registers that return the weight value if the input spike is at 1; otherwise, the registers return 0. The hard-coded synaptic weights are sent into a 4-stage tree adder and accumulated into the mem-

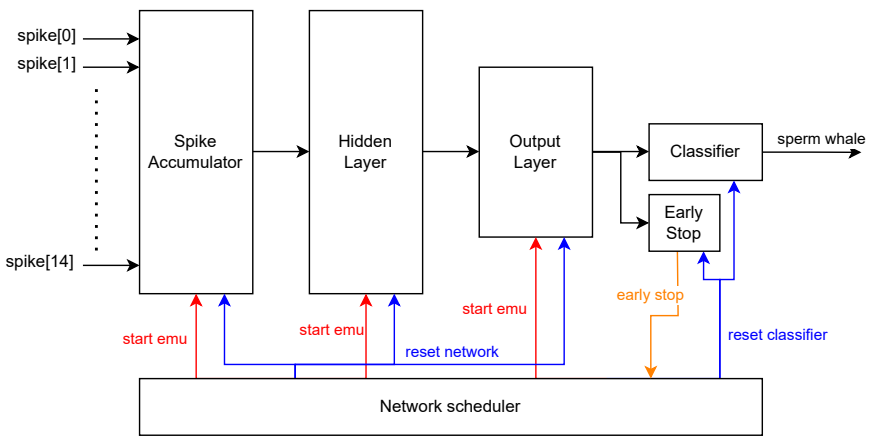

Figure 6.6 – Architecture for 2-layer models designed for SOC deployment. The input spikes are accumulated in a spike accumulator, which represents the input layer. The output layer is connected to both the classifier and early stop detector. A network scheduler component is utilized to trigger the emulation process and clear the network's internal state.

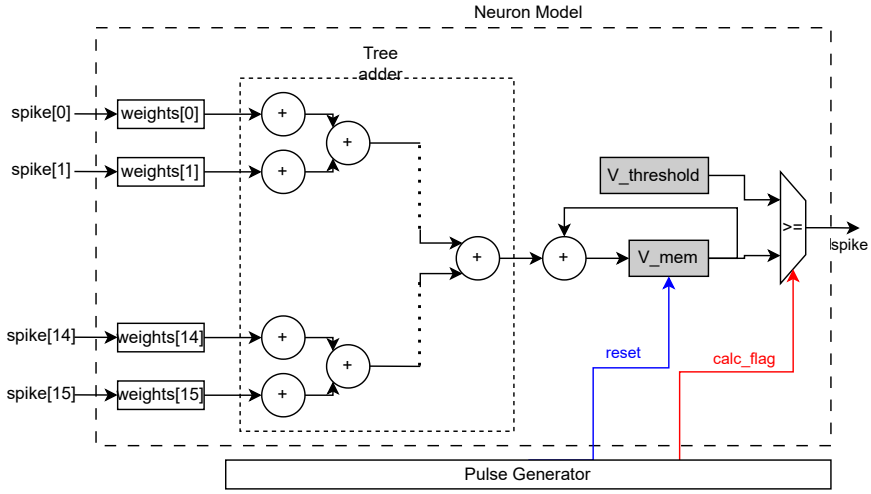

Figure 6.7 – Architecture for the single neuron model for the SOC deployment. The input spikes are passed to the corresponding synaptic weight register. The input current is computed using a tree adder and accumulated into the membrane voltage register. Due to its simplicity, a simple pulse generator is sufficient to schedule the neuron.

brane voltage.
In the next sections, we will present the newly developed components.

Spike Detector

The input spike detection is processed by the `spike_detector` component, illustrated by Figure 6.8. The spike detector consists of a three-stage D latch pipeline designed to debounce the input signals. The number of stages in the pipeline was

determined through board experimentation with the Raspberry Pi 5. We selected the smallest number of stages to maintain a low latency and detect the spike as soon as possible. When a spike is detected, the spike detector generates a single clock cycle signal.

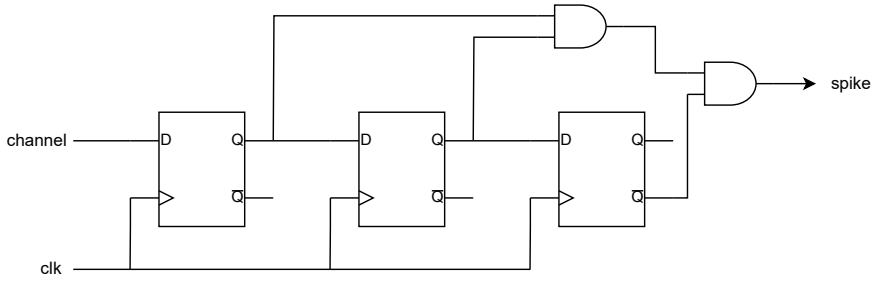

Figure 6.8 – Spike detector diagram. The three-stage D latch is used to debounce the signal while maintaining a low latency and detecting the spike as soon as the cochlea sends it.

Network Scheduler and Early Stop

The second process that must be implemented is the network scheduling. Figure 6.9 illustrates the FSM process of the network scheduling process.
When the *start_emu* signal, controlled by a pulse generator, is set to 1, the scheduler initiates the network scheduling of each layer. It begins with the layer emulation and waits for the layer to complete its execution. Once a layer finishes its process, the scheduler proceeds to schedule the next layer.
At the beginning of a sample process, the component will reset the layer state and the classifier. We decided to clear the network state at the beginning of the sample to increase the duration of the wake-up signal for the last emulation step. This prevents the wake up signal from potentially being set to 1 for only one clock cycle, which operates at 125 MHz, too fast for signal processing.
We can observe a special transition in the `Idle` state when the *i_early_stop* signal is set to 1. We implement an early stop detector mechanism to disable the emulation if it is not necessary. Because the early stop mechanism is correlated with the classifier component, we will describe the early stop mechanism in the next section.

Classifier and Early Stop

The classification component is relatively simple. The classifier module will receive spikes generated by the output layer. The classifier counts these spikes; if the number of received spikes exceeds a spike gap, the classifier component sets the output signal at 1.
The *spike_flag* signal and the output classifier signals are also connected to the early stop detector. This component determines whether it is necessary to continue network updates or if the update operations can be halted, thereby saving power.
The early stop equation is defined by Equation 6.3. We trigger the early stop under two conditions. First, if we have already detected a sperm whale ($classifier = 1$), it is unnecessary to continue the inference process. The second condition ($SpikeCounter +$

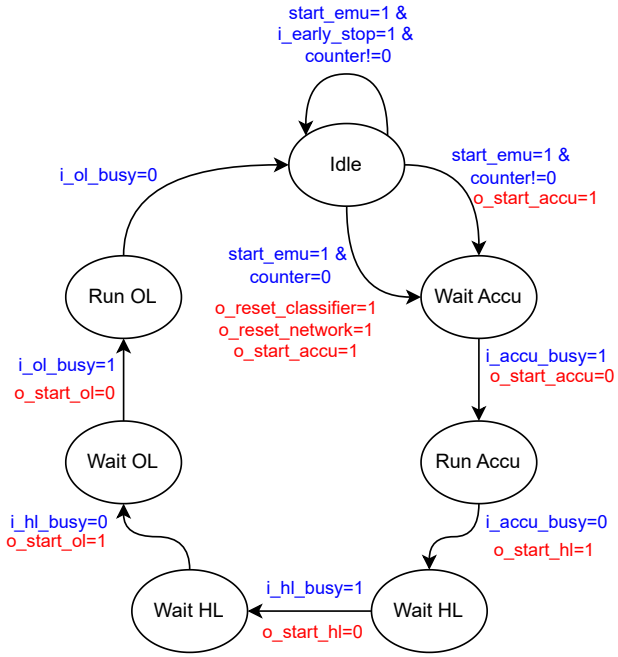

Figure 6.9 – FSM representation of the network scheduling. The *start_emu* signal is triggered by a pulse generator. Depending on the early stop, the emulation process may not start. In addition to the network scheduling, the scheduler triggers the reset signals at the beginning of a new sample process.

$TotalStep-1+StepCounter) < SpikeGap$ is satisfied if there is not enough time to reach the spike gap to detect a sperm whale. Here, $SpikeCounter$ represents the number of spikes generated by the output layer. $StepCounter$ determines the number of emulation steps, and the $TotalStep$ parameter represents the number of time steps on the sample.

$$EarlyStop = (class = 1)\&(SpikeCounter+TotalStep-1-StepCounter) < SpikeGap \quad (6.3)$$

6.3.2 SOC Simulation Results

In this section, we will present the results of our SOC pipeline implementation at two different levels of realism. Due to the necessity of additional circuitry to connect the FPGA and the cochlea chip, we decided to simulate the SOC with two different levels of realism:

1. **Vivado Software Simulation**: To validate our design, we ran a Vivado simulation with a software-based cochlea.
2. **Raspberry Pi 5 Emulation**: To validate the functional behavior of the implemented circuit, we emulated the cochlea using a Raspberry Pi 5. We connect the Raspberry Pi GPIOs to the FPGA GPIOs with jumper wires.

In these experiments, we calculated the accuracy of the model across the entire dataset. To simulate the cochlea, we read the CSV files generated by the oscilloscope used to record the output of the cochlea chip. Consequently, we did not have access to the specific samples sent to the initial network. This approach leads to an overestimation of the accuracy. However, we previously presented the initial accuracy of the trained models, and we will compare the accuracy across the entire dataset to compare the same results.
First, we will present the two experimental setups, followed by a presentation and comparison of the results based on the model implementation setup.

Vivado Simulation Experimental Setup

The first experimental setup consists of a software simulation using the Vivado simulation tool. This experiment is limited due to the lack of hardware specification considerations but allows us to validate the main concepts of the different components.
To run the simulation, we developed a VHDL component that simulates the output of the analog cochlea by reading the CSV files generated by the oscilloscope. The output of the network is recorded by a VHDL process into a buffer CSV file and is regularly read by the Python script that runs the simulation.
The Python script that executes the simulation is presented in Algorithm 3. For each file, the script modifies the input file in the test bench file and then calls a TCL script to execute the Vivado simulation. Once the simulation is finished, the script will read the buffer file, update the accuracy, and save the results on NumPy arrays.

Algorithm 3 Vivado Simulation Script: The algorithm reads all dataset files. For each file, the test bench file is modified, and the simulation is executed with this file. Once the simulation is finished, the accuracy and the results array are updated. After all files have been simulated, the output results are saved.

```
for each file f in datasets do
    test bench modification
    run simulation
    update accuracy
    update y_true and y_pred
end for
save results
```

Raspberry Pi 5 Experimental Setup

The Raspberry Pi 5 setup is more complex than the Vivado simulation setup due to its hardware aspect.
We first designed a cochlea simulator in the Rust programming language to try to achieve the most precise timing measurements and simulate the cochlea in the most realistic manner possible. However, because the Raspberry Pi 5 cannot control the GPIO at 1.25 MHz, the oscilloscope frequency, we increased all time constant by a factor 100 to meet the timing constraints of the Raspberry Pi GPIOs.
In addition to the time constant, we added an enable signal, which is active at low, to synchronize the two devices.
Algorithm 4 represents the main steps of the Cochlea simulation on the Raspberry Pi. The Rust program reads all dataset files and extracts the input samples from the file.

Then, the Raspberry Pi starts the emulation process by setting the *enable* signal to low and sends the output spikes generated by the cochlea to the FPGA. After sending all the data, the Raspberry Pi reads the *spermwhale* signal controlled by the FPGA and sets the FPGA into an idle mode by setting the *enable* signal to 1.

Algorithm 4 Raspberry Pi Cochlea Simulation Algorithm: The algorithm reads all dataset files. For each file, the algorithm extracts the sample and provides it to the Cochlea simulator. The cochlea sets the *enable* to 0 and transmits the sample data. When the transmission is finished, the cochlea reads the classification results and updates results variables.

```
for each file f in datasets do
    enable ← 1
    for each sample s in f do
        Read sample
        enable ← 0
        for each step cs in s do
            now ← Instant :: now()
            channels ← cs
            channels ← 0
            while now.elapsed() < WAIT_DURATION do
                spin_loop()
            end while
        end for
        res ← spermwhale
        enable ← 1
    end for
    update accuracy
end for
save results
```

A photograph of the final circuit is illustrated in Figure 6.10. The Raspberry Pi, on the left side of the photo, is connected with jumper wires to the Arty-Z7 20 FPGA on the right. The colored wires represent the channel connections, the gray wires the control signals, including the *enable* and the *spermwhale* signals, and the black wire represents the common ground to ensure a shared voltage reference.
To verify the interconnection circuit, we conducted several experiments using models specifically designed to generate output spikes under certain conditions. An output spike is generated if at least $N + 1$ spikes are written to the channel N. For example, at least 1 spike must be written to the channel 0 to generate an output spike. Using various architectures, we verified the correct functioning of wiring and hardware implementation of the components. It was also during these experiments that we determined the number of D latch stages for the spike detector.

Results

In this section, we will present the results from the three following configurations:

1. UART FPGA implementation
2. Vivado Simulation
3. Raspberry Pi Emulation

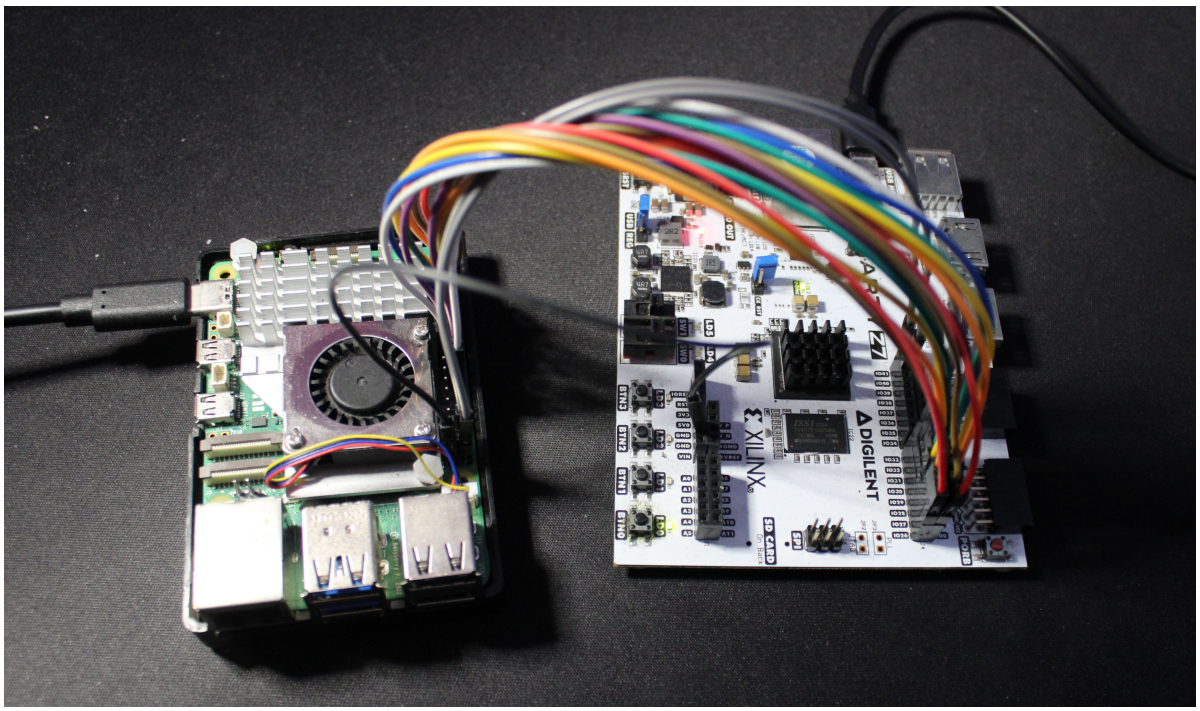

Figure 6.10 – Connection between the Raspberry Pi 5, on the left, and the Arty-Z7 20, on the right. The colored wires represent the channel interconnections, the gray wires the control signal, specifically the *enable* and *spermwhale* signals, and the black wire the common ground.

Table 6.8 summarizes all accuracy for all models across all configurations. All configurations achieve similar accuracy, proving the good behavior of the SOC-based architectures. The only significant drop in accuracy occurs for the `QAT_16_1_s1` model in Vivado Simulation configuration, where the accuracy decreases from 84% and 83% to 80%. However, we place less importance on this configuration since it is a pure software simulation without any hardware configurations. This accuracy drop can be explained by the output record process not being properly tested due to a lack of development time.

Table 6.8 – Accuracy comparison in the complete 2-classes ULP Cochlea dataset depends on the model and configuration. All three configurations achieve similar accuracy, except for the Vivado configuration of the `QAT_16_1_s1`, which shows a 4% drop in accuracy.

Configuration	UART	Vivado	Raspberry
`QAT_16_128_1_w8_s20`	92.75	91.4	92.5
`QAT_16_128_1_w7_s20`	90.63	89.3	89.8
`QAT_16_128_1_s1`	91.41	90.00	91.1
`QAT_16_1_s1`	84.26	80.8	83.1

Table 6.9 references the power consumption for all models in both UART and Raspberry configurations. The Vivado configuration is not shown as it is a pure software

simulation. We can observe a significant reduction in power consumption between the two implementations. One explanation for this is the elimination of the UART component, which saves power by reducing the hardware usage. However, upon closely examining the power report, the different update processes also contribute to the reduction in power reduction.

Table 6.9 – Vectorless power consumption estimation, in mW, depending on the models for UART and Raspberry configurations. We observe a notable power reduction from 1.19 to 1.27 times better for the Raspberry Pi implementations.

Configuration	UART	Raspberry
QAT_16_128_1_w8_s20	238	199
QAT_16_128_1_w7_s20	195	171
QAT_16_128_1_s1	167	155
QAT_16_1_s1	133	110

Table 6.10 shows a comparison of power consumption by architecture component for the QAT_16_128_1_w8_s20 model, depending on the implementation configuration. The power consumption of the input layer, the UART component for the UART configuration, and the accumulator for the Raspberry Pi configuration are equal for both configurations. However, the power estimation for the hidden layer changes significantly. Given that the layer implementation is very similar, the power reduction can be attributed to the neuron state update mechanism.
We can also demonstrate the impact on power consumption of the early stop mechanism. Since the effect of the early stop is highly dependent on execution, we ran power estimations with and without the early stop with a SAIF file. The Switching Activity Interchange Format (SAIF) format is an ASCII format that records the toggle rate of different signals based on a simulation. This switching activity can be used by the Vivado power estimator to provide a more accurate power consumption by using more realistic signal activity.
To demonstrate the positive impact of the early stop detector, we conducted simulations using the QAT_16_128_1_w8_s20 and QAT_16_128_1_w7_s20 models, which both utilize the early stop detector. We simulated these architectures with a reduced dataset with 20 samples from the False class and 20 samples from the True class, resulting in a simulation duration of 400 ms.

Table 6.10 – Vectorless power estimation, in mW, comparison between the UART and the Raspberry configuration for the QAT_16_128_1_w8_s20 model. The input layer consumption, represented by the UART module of the accumulator depending on the configuration, does not change. However, we can observe a significant drop in power consumption for the hidden layer. The reports do not provide further details about the other architecture components.

Module	UART	Raspberry
Input Layer (UART/Accumulator)	5	5
Hidden Layer	122	83

Table 6.11 – Power consumption, in mW, of both models using the early stop detector with and without the implementation of the early stop mechanism. We can observe a significant drop for the QAT_16_128_1_w8_s20 model from 230 to 199. The effect of the early stop mechanism appears less significant for the QAT_16_128_1_w7_s20 model, with a reduction of only 6 mW.

Early Stop	Yes	No
QAT_16_128_1_w8_s20	199	230
QAT_16_128_1_w7_s20	172	178

Table 6.11 references the power estimation, in mW, of both models with and without the early stop mechanism. Depending on the model, the impact of this mechanism varies, but the early stop tends to reduce the power consumption overall. For the QAT_16_128_1_w8_s20, the decrease is more pronounced with a 13% reduction, compared to the QAT_16_128_1_w7_s20 model, which shows only a 3% reduction in power consumption. This difference can be explained by the sparsity of the output layers. The output layer of the first model is likely more active than that of the second model. The conditions of the early detector need to be further validated for the QAT_16_128_1_w7_s20 model. Additionally, this lower reduction can be attributed to the lower accuracy of the model, resulting in a lower validation frequency of the first condition, $classifier = 1$.
Through these experiments, particularly with the Raspberry Pi implementation, we have demonstrated the behavior of our FPGA architecture and models for an embedded SOC integration. Depending on the model selected, the power performance and the accuracy will vary, and further discussions are necessary to determine the best model.
To achieve lower power consumption, we can utilize a smaller FPGA chip that is more dedicated for SOC integration than our Arty-Z7 20, such as the CMOD A7 35t FPGA [37]. It is also important to note that the SOC integration was not demonstrated with a direct connection between the sensor and the FPGA. Therefore, additional experiments are necessary to fully demonstrate the SOC integration.

6.4 Multi Classes Dataset

In the two previous sections, we presented results on the 2-classes ULP Cochlea dataset. Another dataset created with the analog cochlea is a 10-classes dataset based on the Watkins Marine Mammal Sound Dataset [190, 160, 191]. This dataset is composed of approximately 9,500 samples split into 7,600 samples in the training set and 1,900 samples in the test set.
We trained two different models to run the classification task. The first model is a simple FF-FC model with a 16-128-10 network topology. Since the input data consists of sound, we also developed a recurrent network topology with the 16-128_R-10_R configuration. Due to a lack of time, the models' parameters and topologies were not fine-tuned, and we used the FPT training method over 75 epochs to simplify the network training.
Table 6.12 shows the accuracies of both models depending on the inference methods. First, the impact of quantization is not significant for either model. We can observe

a drop in accuracy for the recurrent model between the software (full precision and PTQ simulation) and the FPGA inference, which can be explained by the quantization. Another notable observation is the lower performance of the recurrent model compared to the forward model. Our hypothesis is that the recurrent model was not sufficiently fine-tuned to achieve better accuracy. Additional experiments are therefore necessary to confirm this hypothesis.

Table 6.12 – Accuracy results for both trained models with the 10-classes ULP Cochlea dataset, depending on the inference method. The recurrent model achieves lower accuracy than the FF-FC model, and a significant drop in accuracy can be observed for this model. The impact of quantization is not significant in these networks.

Evaluation Method	FF-FC Model	Rec Model
Full Precision	88.57	86.80
PTQ Simulation	88.20	86.91
FPGA	88.09	83.08

Figures 6.11 and 6.12 represent the confusion matrices of the FF-FC model and the recurrent model, respectively. In both models, we can observe that the models failed to detect class 1, which represents the False Killer Whale. This can be explained by the very low number of samples in the dataset, with only 7 samples in the training set and 2 samples in the test set. For the recurrent model, the classification of class 5, the Long-Fined Pilot Whale, is particularly poor, further decreasing the final accuracy.
Table 6.13 summarizes the inference time and the energy consumption of both models on different hardware targets. As for the previous experiments, the lowest energy consumptions are achieved for the FPGA inference. A notable observation is the same inference time on FPGA for both models. This can be explained by the spike transmission between the UART component and the hidden layer component, which is a time-intensive operation.

Table 6.13 – Inference time and energy consumption of both models for the 10-classes ULP Cochlea classification task, depending on hardware target. For both models, the FPGA inference appears as the fastest and achieves the lowest energy consumption. The inference time on FPGA for both models is equal, indicating that the spike transmission between the UART module and the hidden layer is the most time-intensive operation. However, due to recurrent weight memories, the power consumption of the recurrent model is higher than the FF-FC, resulting in a higher energy consumption.

Model	FF-FC Model			Recurrent Model		
Target	CPU	GPU	FPGA	CPU	GPU	FPGA
Inference Speed (ms)	0.48	0.70	0.094	0.61	0.80	0.094
Energy Consumption (mJ)	24.58	25.40	0.023	31.31	29.29	0.028

The difference in terms of energy consumption between both models can be explained by examining Table 6.14. The power consumption of the recurrent model is higher than that of the FF-FC model. This increase in power consumption can be

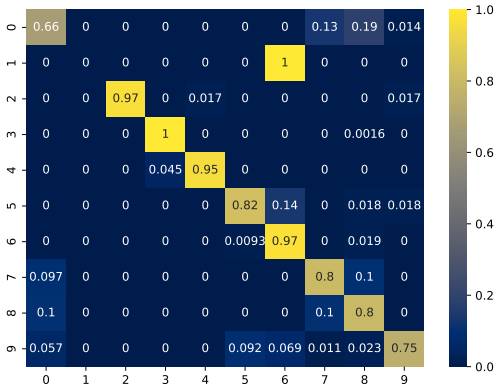

Figure 6.11 – Confusion Matrix of the FPGA inference for the FF-FC model for the 10-classes ULP Cochlea dataset classification task. The class 1, which represents the False Killer Whale species, is always confused with the Melon Headed Whale. This confusion significantly degrades the final accuracy.

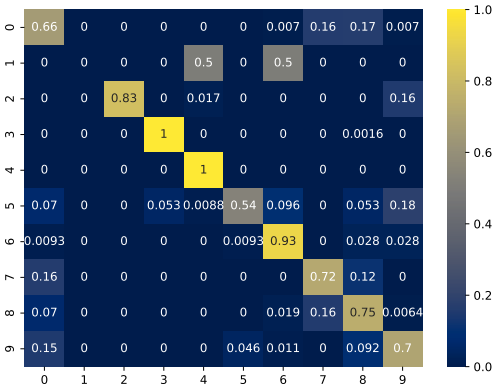

Figure 6.12 – Confusion Matrix of the FPGA inference for the recurrent model for the 10-classes ULP Cochlea dataset classification task. Class 1, which represents the False Killer Whale species, is not recognized at all and is confused with the Clymene Dolphin (class 4) and the False Killer Whale (class 6). Additionally, the Long-Finned Pilot Whale is correctly classified with only 54% accuracy. These two species degrade the final model's accuracy.

mostly attributed to the increase of BRAM consumption, while other resources achieve similar levels of occupation. This overconsumption of memory is due to the memory required to store the recurrent synaptic weights.
We demonstrated the capability of ModNEF to run inference for biodiversity classification tasks using a more complex spiking dataset. Currently, the best model, in terms of accuracy and energy consumption, is the FF-FC model, which achieves higher accuracy and lower power consumption than the recurrent model. However, we have also identified several limitations of the dataset, particularly for the False Killer Whale (class 1), which degrades the accuracy due to a low number of samples in both training and test sets. Additional experiments are necessary to optimize the dataset and the models to achieve better accuracy.

Table 6.14 – Hardware metrics for both models for the 10-classes ULP Cochlea dataset. The recurrent model consumes more power than the FF-FC model. This increase can be explained by the additional BRAM usage, which is necessary to store the additional recurrent synaptic weights.

Metrics	FF-FC Model	Rec Model
Dynamic Power (mW)	144	184
Static Power (mW)	109	111
BRAM (KBits)	576 ; 252 (16.43%)	1152 ; 252 (27.86%)
FF	3596 ; 344 (3.70%)	2599 ; 344 (2.77%)
LUT	14995 ; 508 (29.14%)	11385 ; 510 (22.36%)
LUTRAM	0 ; 96 (0.55%)	0 ; 96 (0.55%)
DSP	0 ; 0 (0.00%)	00 ; 0 (0.00%)

6.5 Conclusion

In this chapter, we present the ULP Cochlea project. The interdisciplinary project involves collaboration among electronic researchers, computer scientists, and biologists to monitor the sperm whale population on the Mediterranean Sea. The main goal of this project is to deploy ultra-low-power SOC with an analog neuromorphic cochlea that will convert the environmental sounds into spikes, which will then be processed by an SNN. As an intermediate step, due to challenges in developing an analog SNN chip, we have decided to use a digital neuromorphic chip, in particular an FPGA, as a wake-up engine. This digital chip will activate a more power-hungry processor when necessary.

To achieve this task, we developed two different datasets: a 2-classes dataset and a 10-classes dataset, both converted into spikes using the analog cochlea chip. However, because the 10-classes dataset was developed too recently, we focused our work on the 2-classes dataset.

Based on an initial network and dataset exploration conducted by our colleague Mazdack Fatahi, we developed five different models using the ModNEF framework. Because the model trained by the FPT algorithm was too sensitive to quantization, we focused our work on QAT-based models. We demonstrated the viability of these models with classical FPGA implementation. However, these implementation methods are relatively artificial compared to a SOC integration due to the data reception.

To prove the viability ModNEF for the project, we developed specific architecture for SOC integration. We validated the SOC architecture using software Vivado Simulation to confirm the functional behavior of our architecture. To validate our architecture in a more realistic manner, we emulated the output of the cochlea using a Raspberry Pi 5. We connected the Raspberry Pi GPIOs and the FPGA GPIOs to emulate the electrical SOC circuit. With both configurations, we achieved comparable accuracy but with a lower power consumption, thanks to the update process and the early stop mechanism.

We also trained models with the 10-classes dataset. Although we achieved good accuracy and power consumption, additional experiments are necessary to improve mod-

els performances.
We demonstrated the capability of ModNEF to be implemented on embedded SOC for complex classification tasks for biodiversity monitoring. Additional experiments are necessary for the SOC integration and for the 10-classes dataset models.

7 Conclusion and Future Work

In this thesis, we have presented Modular Neuromorphic Emulator for FPGA (ModNEF), an open-source modular FPGA architecture for SNN deployment for embedded low-power classification tasks. The primary motivation behind the development of this tool has been the lack of open-access FPGA architecture proposed by researchers, which compelled us to develop a new architecture and offer it as an open-source solution. As an open-source project, ModNEF is designed to be highly flexible, enabling users to deploy a wide variety of network topologies with different neuron models and emulation algorithms. In addition to this deployment flexibility, ModNEF provides a high level of control over the hardware implementation, offering various neuron models, emulation strategies, and module parameters, each with different impacts on hardware implementation and power consumption.

In Chapter 2, we have presented a comprehensive state-of-the-art on SNN hardware implementation, with a particular focus on FPGA implementations. Through this study, we have identified three major challenges:

- The implementation of complex mathematical operations in neuron models, learning rules, or synapse models, such as exponential operation.
- The limited hardware resources highly limit the network implementation, both in terms of the number of neurons due to limited computational resources and in terms of the number of parameters by the limited amount of memory.
- The event-driven computing scheme appears as the most energy-efficient computing scheme, but its implementation on FPGA remains challenging. The clock-driven strategy appears as the simplest solution for FPGA integration, even if it degrades power performances.

To address these challenges and meet our motivations, we have presented ModNEF in Chapter 3. ModNEF is based on the interconnection of independent modules that communicate using a common protocol. Due to this modular architecture, ModNEF offers various numbers of neuron models and emulation strategies and can support the integration of new mechanisms, such as new neuron models or learning algorithms. In addition to the FPGA architecture, ModNEF provides a Python framework that allows users to train and deploy a ModNEF model on FPGA without requiring knowledge of hardware design. According to Table 3.1, our work appears as a competitive emulator with a high level of configurability.

In Chapter 4, we have demonstrated the capability of ModNEF to perform classification tasks for several well-known datasets, such as MNIST, used as proof of concept, N-MNIST, SHD, and DVS Gesture datasets. We have also presented the identified limitations of ModNEF, which primarily consist of hardware constraints of the target FPGA, and proposed solutions to mitigate these limitations. The choice of emulation strategy and the reduction of encoding synaptic weight bitwidth appear to be effective solutions to minimize the impact of the models on hardware implementations. However, reducing the precision of synaptic weights leads to a decrease in classification performances.

The impact of quantization has been thoroughly investigated in Chapter 5. Two major quantization schemes exist for quantifying synaptic weights: Post-Training Quantiza-

tion (PTQ), where the quantization function is applied after the training phase, and the Quantization Aware Training (QAT), where the quantization is considered during the training phase. PTQ appears to be a simple solution for model quantization. However, depending on the target bitwidth, the neuron model, and the quantization algorithm, the impact on accuracy can be significant, with a high drop accuracy. QAT training can minimize the negative impact of quantization, but the training phase can be more challenging with a more sensitive loss and sometimes model degeneration during training. Nevertheless, QAT appears to be a good solution to achieve high accuracy, similar to full-precision training, while remaining a low synaptic resolution, leading to lower power consumption.
After presenting and studying ModNEF architecture and behavior, we present an application for embedded classification tasks within the ULP Cochlea ANR Project. The ULP Cochlea project aims to develop a SOC neuromorphic system to monitor the sperm whale population on the Mediterranean Sea. An analog cochlea has been developed by the IEMN research team, which can convert input audio signals to a spiking representation. ModNEF has been chosen as the initial solution for spike processing due to its availability and can be used as a wake-up engine to detect sperm whales and activate a more powerful processor for more complex processing. We trained several models, each with different accuracy and energy consumption, and integrated these models on an architecture compatible with SOC integration, maintaining similar accuracy to UART-based FPGA deployment.
Considering the four last chapters of the thesis, we can conclude that ModNEF addresses our initial motivations:

- **Flexible**: We have demonstrated that ModNEF can implement various network topologies with feed-forward or recurrent SNN. Because ModNEF modules are independent, we can implement heterogeneous networks with different neuron models.
- **Accessible**: The ModNEF framework offers a user-friendly implementation of ModNEF-based models from training to FPGA deployment, without requiring knowledge of FPGA.
- **Upgradable**: Due to its modular architecture, ModNEF appears to be highly flexible. As we shown in Chapters 4 and 6, new modules can be integrated, as long as the common communication protocol is implemented, into the hardware architecture and on the software framework thanks to base classes.

7.1 Future Work

The current version of ModNEF demonstrates its capabilities, yet there remain several areas where it can be further improved.
In Chapter 4, we have identified the data transmission as a bottleneck, which significantly reduces the inference latency. Therefore, in the short term, we plan to enhance the communication protocol for transmitting spikes from the computer to the FPGA. The current UART communication protocol is a good starting point but remains limited by its low baud rate. To improve the communication protocol, we will integrate ModNEF architecture with the Zynq processor implemented on several FPGAs, thereby gaining access to Ethernet ports or SD card ports to enhance data transmission.
According to Table 3.1, we have identified three major areas for improvement for long-term work.

First, we plan to implement the Event-Driven computing scheme, updating the neuron state only when an incoming event is detected. Unlike the Clock-Driven computing scheme, which regularly updates the neuron state even in the absence of input data, the event-driven architecture updates neuron states only when new information must be processed. This leads to a reduction of the number of operations, resulting in a lower power consumption [68, 211, 47, 2]. As we explained in Section 2.3.1, a major limitation of this method is the need to solve differential equations of the neuron models, resulting in a more complex mathematical model. Moreover, the processing of network spikes can increase the power consumption of the architecture [114]. To address these challenges, several works propose a hybrid approach with an adaptive computing scheme depending on the number of spikes [114, 108]. Our solution is to determine the computing scheme at a module level based on software estimation. A module that receives a number of spikes above a certain threshold will adopt the clock-driven computing scheme. In contrast, if a module receives a low number of spikes, the event-driven computing scheme can be used to save power consumption by not activating the module at each emulation step. This approach differs from the two previously cited works [114, 108], which change the computing scheme at a global level.

The second identified limitation of ModNEF is the lack of online and on-chip learning rules. The research community focuses their efforts on local learning rules such as delay learning [91], dendrite learning [201, 204], but most works focus on STDP [55, 108, 201]. The major challenge in implementing STDP is the exponential function, a cost-intensive operation in FPGA. Some works propose to use the CORDIC algorithm to simplify the exponential function [74, 167] while others use LUTs with pre-calculated values [115, 108, 172]. The implementation of the exponential function is a common challenge with the previously described point, as event-driven neuron models also require it. To reduce the hardware cost of the exponential function, it is possible to time-multiplex the hardware circuit to update multiple synaptic weights within a single hardware circuit. Further research should be conducted to determine whether STDP is a viable learning rule for FPGA implementation while other learning rules can be more efficient.

The third improvement for ModNEF, according to Table 3.1 support for convolutional networks represents a potential future improvement. Most FPGA emulators focus on FF-FC and Recurrent network topologies, thanks to their implementation simplicity. However, convolutional network topologies have proven efficient for image [4, 90] and DVS data classification [9, 110, 177]. The implementation of convolution kernels remains a significant challenge. Some works propose parallelizing the convolution operation to achieve low latency [33], while DeepFire2 [13] proposes configuring the level of parallelism to achieve the best trade-off between hardware cost and execution latency. Implementing convolution in ModNEF appears promising to enhance both flexibility and completeness of the architecture.

Finally, a limitation of our work is the lack of direct comparison with other neuromorphic hardware platforms. While we evaluated ModNEF against traditional CPUs and GPUs, such comparisons are inherently biased due to the fundamental architectural differences between von Neumann systems and event-driven neuromorphic hardware. Future work should prioritize benchmarking ModNEF against FPGA-based SNN emulators (e.g., Spiker+ [29]) and dedicated neuromorphic chips (e.g., SpiNNaker [143] or Xylo [21]).

Additionally, comparing SNNs and ANNs on similar architectures would provide in-

sights into their relative strengths for low-power embedded AI. However, it is critical to note that SNNs are ultimately designed for analog or mixed-signal implementations [126, 175], which promise orders-of-magnitude improvements in power efficiency compared to digital-only solutions.

7.2 Acknowledgments

We thank IRCICA (Institut de Recherche sur les Composants logiciels et matériels pour l'Information et la Communication Avancée) for granting access to the neuromorphic research platform and Région Hauts-de-France for a PhD grant.

7.3 Code availability

ModNEF source code is available on GitLab of the University of Lille: `https://gitlab.univ-lille.fr/bioinsp/ModNEF`, the presented networks are available on `https://gitlab.univ-lille.fr/bioinsp/modneftheseresult`, and the models trained for the ULP Cochlea project are available on `http://gitlab.univ-lille.fr/bioinsp/ulp_cochlea_fpga`.

Bibliography

[1] L. F Abbott. « Lapicque's introduction of the integrate-and-fire model neuron (1907) ». *Brain Research Bulletin* 50.5 (Nov. 1, 1999), pp. 303–304. ISSN: 0361-9230. DOI: 10.1016/S0361-9230(99)00161-6. URL: https://www.sciencedirect.com/science/article/pii/S0361923099001616 (visited on 05/02/2025).

[2] Nassim Abderrahmane, Edgar Lemaire, and Benoît Miramond. « Design Space Exploration of Hardware Spiking Neurons for Embedded Artificial Intelligence ». *Neural Networks* 121 (Jan. 1, 2020), pp. 366–386. ISSN: 0893-6080. DOI: 10.1016/j.neunet.2019.09.024. URL: https://www.sciencedirect.com/science/article/pii/S0893608019303041 (visited on 05/13/2024).

[3] Nassim Abderrahmane and Benoît Miramond. « Neural coding: adapting spike generation for embedded hardware classification ». *2020 International Joint Conference on Neural Networks (IJCNN)*. 2020 International Joint Conference on Neural Networks (IJCNN). ISSN: 2161-4407. July 2020, pp. 1–8. DOI: 10.1109/IJCNN48605.2020.9207702. URL: https://ieeexplore.ieee.org/abstract/document/9207702 (visited on 05/06/2024).

[4] Nassim Abderrahmane et al. « SPLEAT: SPiking Low-power Event-based ArchiTecture for in-orbit processing of satellite imagery ». *2022 International Joint Conference on Neural Networks (IJCNN)*. 2022 International Joint Conference on Neural Networks (IJCNN). ISSN: 2161-4407. July 2022, pp. 1–10. DOI: 10.1109/IJCNN55064.2022.9892277. URL: https://ieeexplore.ieee.org/abstract/document/9892277 (visited on 04/08/2024).

[5] admin. *Akida*. BrainChip. URL: https://brainchip.com/technology/ (visited on 11/29/2025).

[6] Filipp Akopyan et al. « TrueNorth: Design and Tool Flow of a 65 mW 1 Million Neuron Programmable Neurosynaptic Chip ». *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 34.10 (Oct. 2015). Conference Name: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, pp. 1537–1557. ISSN: 1937-4151. DOI: 10.1109/TCAD.2015.2474396.

[7] Asmer Hamid Ali, Mozhgan Navardi, and Tinoosh Mohsenin. *Energy-Aware FPGA Implementation of Spiking Neural Network with LIF Neurons*. Nov. 3, 2024. DOI: 10.48550/arXiv.2411.01628. arXiv: 2411.01628[cs]. URL: http://arxiv.org/abs/2411.01628 (visited on 05/19/2025).

[8] Abdulaziz S. Alkabaa et al. « An Investigation on Spiking Neural Networks Based on the Izhikevich Neuronal Model: Spiking Processing and Hardware Approach ». *Mathematics* 10.4 (Jan. 2022). Number: 4 Publisher: Multidisciplinary Digital Publishing Institute, p. 612. ISSN: 2227-7390. DOI: 10.3390/math10040612. URL: https://www.mdpi.com/2227-7390/10/4/612 (visited on 05/16/2025).

[9] Arnon Amir et al. « A Low Power, Fully Event-Based Gesture Recognition System ». *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). Honolulu, HI: IEEE, July 2017, pp. 7388–7397. ISBN: 978-1-5386-0457-1. DOI: 10.1109/CVPR.2017.781. URL: https://ieeexplore.ieee.org/document/8100264/ (visited on 06/28/2023).

[10] Masoud Amiri, Soheila Nazari, and Karim Faez. « Digital realization of the proposed linear model of the Hodgkin-Huxley neuron ». *International Journal of Circuit Theory and Applications* 47.3 (2019). _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cta.2596, pp. 483–497. ISSN: 1097-007X. DOI: 10.1002/cta.2596. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/cta.2596 (visited on 05/16/2025).

[11] Li Minn Ang and Kah Phooi Seng. « GPU-Based Embedded Intelligence Architectures and Applications ». *Electronics* 10.8 (Jan. 2021). Publisher: Multidisciplinary Digital Publishing Institute, p. 952. ISSN: 2079-9292. DOI: 10.3390/electronics10080952. URL: https://www.mdpi.com/2079-9292/10/8/952 (visited on 08/12/2025).

[12] Daniel Auge et al. « A Survey of Encoding Techniques for Signal Processing in Spiking Neural Networks ». *Neural Processing Letters* 53.6 (Dec. 1, 2021), pp. 4693–4710. ISSN: 1573-773X. DOI: 10.1007/s11063-021-10562-2. URL: https://doi.org/10.1007/s11063-021-10562-2 (visited on 05/01/2025).

[13] Myat Thu Linn Aung et al. « DeepFire2: A Convolutional Spiking Neural Network Accelerator on FPGAs ». *IEEE Trans. Comput.* 72.10 (Oct. 1, 2023), pp. 2847–2857. ISSN: 0018-9340. DOI: 10.1109/TC.2023.3272284. URL: https://doi.org/10.1109/TC.2023.3272284 (visited on 09/09/2025).

[14] Sami Barchid et al. « Spiking neural networks for frame-based and event-based single object localization ». *Neurocomputing* 559 (Nov. 28, 2023), p. 126805. ISSN: 0925-2312. DOI: 10.1016/j.neucom.2023.126805. URL: https://www.sciencedirect.com/science/article/pii/S0925231223009281 (visited on 05/01/2025).

[15] Ben Varkey Benjamin et al. « Neurogrid: A Mixed-Analog-Digital Multichip System for Large-Scale Neural Simulations ». *Proceedings of the IEEE* 102.5 (May 2014). Conference Name: Proceedings of the IEEE, pp. 699–716. ISSN: 1558-2256. DOI: 10.1109/JPROC.2014.2313565.

[16] Guo-qiang Bi and Mu-ming Poo. « Synaptic Modifications in Cultured Hippocampal Neurons: Dependence on Spike Timing, Synaptic Strength, and Postsynaptic Cell Type ». *The Journal of Neuroscience* 18.24 (Dec. 15, 1998), pp. 10464–10472. ISSN: 0270-6474. DOI: 10.1523/JNEUROSCI.18-24-10464.1998. URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC6793365/ (visited on 05/07/2025).

[17] Zhenshan Bing et al. « A Survey of Robotics Control Based on Learning-Inspired Spiking Neural Networks ». *Frontiers in Neurorobotics* 12 (July 6, 2018). Publisher: Frontiers. ISSN: 1662-5218. DOI: 10.3389/fnbot.2018.00035. URL: https://www.frontiersin.org/articles/10.3389/fnbot.2018.00035 (visited on 04/18/2024).

[18] Tom Birkoben et al. « A spiking and adapting tactile sensor for neuromorphic applications ». *Scientific Reports* 10.1 (Oct. 14, 2020). Publisher: Nature Publishing Group, p. 17260. ISSN: 2045-2322. DOI: 10.1038/s41598-020-74219-1. URL: https://www.nature.com/articles/s41598-020-74219-1 (visited on 05/01/2025).

[19] K.A. Boahen. « Point-to-point connectivity between neuromorphic chips using address events ». *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing* 47.5 (May 2000). Conference Name: IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing, pp. 416–434. ISSN: 1558-125X. DOI: 10.1109/82.842110.

[20] Sander M. Bohte, Joost N. Kok, and Han La Poutré. « Error-backpropagation in temporally encoded networks of spiking neurons ». *Neurocomputing* 48.1 (Oct. 1, 2002), pp. 17–37. ISSN: 0925-2312. DOI: 10.1016/S0925-2312(01)00658-0. URL: https://www.sciencedirect.com/science/article/pii/S0925231201006580 (visited on 03/27/2025).

[21] Hannah Bos and Dylan Muir. *Sub-mW Neuromorphic SNN audio processing applications with Rockpool and Xylo*. Sept. 20, 2022. DOI: 10.48550/arXiv.2208.12991. arXiv: 2208.12991[cs]. URL: http://arxiv.org/abs/2208.12991 (visited on 05/13/2025).

[22] Aurélien Bourdon et al. « PowerAPI: A Software Library to Monitor the Energy Consumed at the Process-Level ». *ERCIM News*. Special Theme: Smart Energy Systems 92 (Jan. 2013). Ed. by ERCIM. Publisher: ERCIM, pp. 43–44. URL: https://inria.hal.science/hal-00772454 (visited on 04/25/2025).

[23] Maxence Bouvier et al. « Spiking Neural Networks Hardware Implementations and Challenges: a Survey ». *ACM Journal on Emerging Technologies in Computing Systems* 15.2 (June 4, 2019), pp. 1–35. ISSN: 1550-4832, 1550-4840. DOI: 10.1145/3304103. arXiv: 2005.01467[cs]. URL: http://arxiv.org/abs/2005.01467 (visited on 10/17/2022).

[24] Romain Brette. « Philosophy of the Spike: Rate-Based vs. Spike-Based Theories of the Brain ». *Frontiers in Systems Neuroscience* 9 (Nov. 10, 2015). ISSN: 1662-5137. DOI: 10.3389/fnsys.2015.00151. URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4639701/ (visited on 11/25/2020).

[25] Fanyu Bu and Xin Wang. « A smart agriculture IoT system based on deep reinforcement learning ». *Future Generation Computer Systems* 99 (Oct. 1, 2019), pp. 500–507. ISSN: 0167-739X. DOI: 10.1016/j.future.2019.04.041. URL: https://www.sciencedirect.com/science/article/pii/S0167739X19307277 (visited on 04/18/2024).

[26] A. N. Burkitt. « A Review of the Integrate-and-fire Neuron Model: I. Homogeneous Synaptic Input ». *Biological Cybernetics* 95.1 (July 1, 2006). Company: Springer Distributor: Springer Institution: Springer Label: Springer Number: 1 Publisher: Springer-Verlag, pp. 1–19. ISSN: 1432-0770. DOI: 10.1007/s00422-006-0068-6. URL: https://link.springer.com/article/10.1007/s00422-006-0068-6 (visited on 05/02/2025).

[27] Andrea Calimera, Enrico Macii, and Massimo Poncino. « The Human Brain Project and neuromorphic computing ». *Functional Neurology* 28.3 (2013), pp. 191–196. ISSN: 0393-5264. URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3812737/ (visited on 04/09/2022).

[28] Alessio Carpegna, Alessandro Savino, and Stefano Di Carlo. « Spiker+: a framework for the generation of efficient Spiking Neural Networks FPGA accelerators for inference at the edge ». *IEEE Transactions on Emerging Topics in Computing* (2024), pp. 1–15. ISSN: 2168-6750. DOI: 10.1109/TETC.2024.3511676. URL: https://ieeexplore.ieee.org/abstract/document/10794606 (visited on 04/25/2025).

[29] Alessio Carpegna, Alessandro Savino, and Stefano Di Carlo. « Spiker: an FPGA-optimized Hardware accelerator for Spiking Neural Networks ». *2022 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. 2022 IEEE Computer Society Annual Symposium on VLSI (ISVLSI). ISSN: 2159-3477. July 2022, pp. 14–19. DOI: 10.1109/ISVLSI54635.2022.00016. URL: https://ieeexplore.ieee.org/document/9911998 (visited on 02/15/2024).

[30] Andrew Cassidy, Andreas G. Andreou, and Julius Georgiou. « A combinational digital logic approach to STDP ». *2011 IEEE International Symposium of Circuits and Systems (ISCAS)*. 2011 IEEE International Symposium of Circuits and Systems (ISCAS). ISSN: 2158-1525. May 2011, pp. 673–676. DOI: 10.1109/ISCAS.2011.5937655.

[31] Zhuoqing Chang et al. « A Survey of Recent Advances in Edge-Computing-Powered Artificial Intelligence of Things ». *IEEE Internet of Things Journal* 8.18 (Sept. 2021), pp. 13849–13875. ISSN: 2327-4662. DOI: 10.1109/JIOT.2021.3088875. URL: https://ieeexplore.ieee.org/document/9453402 (visited on 04/25/2025).

[32] Rajneesh Chaurasiya et al. « Emerging higher-order memristors for bio-realistic neuromorphic computing: A review ». *Materials Today* 68 (Sept. 1, 2023), pp. 356–376. ISSN: 1369-7021. DOI: 10.1016/j.mattod.2023.08.002. URL: https://www.sciencedirect.com/science/article/pii/S1369702123002572 (visited on 04/25/2025).

[33] Qinyu Chen et al. « Skydiver: A Spiking Neural Network Accelerator Exploiting Spatio-Temporal Workload Balance ». *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41.12 (Dec. 2022), pp. 5732–5736. ISSN: 1937-4151. DOI: 10.1109/TCAD.2022.3158834. URL: https://ieeexplore.ieee.org/abstract/document/9733037 (visited on 05/20/2025).

[34] Xi Cheng et al. « A 1024-Neuron 1M-Synapse Event-Driven SNN Accelerator for DVS Applications ». *2024 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2024 IEEE International Symposium on Circuits and Systems (ISCAS). ISSN: 2158-1525. May 2024, pp. 1–5. DOI: 10.1109/ISCAS58744.2024.10558416. URL: https://ieeexplore.ieee.org/abstract/document/10558416 (visited on 04/18/2025).

[35] Sayeed Shafayet Chowdhury, Chankyu Lee, and Kaushik Roy. *Towards Understanding the Effect of Leak in Spiking Neural Networks*. June 15, 2020. DOI: 10.48550/arXiv.2006.08761. arXiv: 2006.08761[cs]. URL: http://arxiv.org/abs/2006.08761 (visited on 03/12/2025).

[36] L. Chua. « Memristor-The missing circuit element ». *IEEE Transactions on Circuit Theory* 18.5 (Sept. 1971), pp. 507–519. ISSN: 2374-9555. DOI: 10.1109/TCT.1971.1083337. URL: https://ieeexplore.ieee.org/abstract/document/1083337 (visited on 05/12/2025).

[37] *Cmod A7 Reference Manual - Digilent Reference*. URL: https://digilent.com/reference/programmable-logic/cmod-a7/reference-manual (visited on 08/09/2025).

[38] Benjamin Cramer et al. « The Heidelberg Spiking Data Sets for the Systematic Evaluation of Spiking Neural Networks ». *IEEE Transactions on Neural Networks and Learning Systems* 33.7 (July 2022). Conference Name: IEEE Transactions on Neural Networks and Learning Systems, pp. 2744–2757. ISSN: 2162-2388. DOI: 10.1109/TNNLS.2020.3044364. URL: https://ieeexplore.ieee.org/document/9311226 (visited on 04/19/2024).

[39] Manon Dampfhoffer et al. « Backpropagation-Based Learning Techniques for Deep Spiking Neural Networks: A Survey ». *IEEE Transactions on Neural Networks and Learning Systems* 35.9 (Sept. 2024), pp. 11906–11921. ISSN: 2162-2388. DOI: 10.1109/TNNLS.2023.3263008. URL: https://ieeexplore.ieee.org/abstract/document/10097504 (visited on 05/09/2025).

[40] Francois Danneville et al. « Ultra low power Cochlea for biodiversity monitoring ». *Colloque BIOCOMP 2023*. Banyuls - sur - Mer, France, Nov. 2023. URL: https://hal.science/hal-04463709.

[41] Mike Davies et al. « Loihi: A Neuromorphic Manycore Processor with On-Chip Learning ». *IEEE Micro* 38.1 (Jan. 2018). Conference Name: IEEE Micro, pp. 82–99. ISSN: 1937-4143. DOI: 10.1109/MM.2018.112130359.

[42] Mike Davies et al. « Advancing Neuromorphic Computing With Loihi: A Survey of Results and Outlook ». *Proceedings of the IEEE* 109.5 (May 2021). Conference Name: Proceedings of the IEEE, pp. 911–934. ISSN: 1558-2256. DOI: 10.1109/JPROC.2021.3067593.

[43] Lei Deng et al. « Model Compression and Hardware Acceleration for Neural Networks: A Comprehensive Survey ». *Proceedings of the IEEE* 108.4 (Apr. 2020), pp. 485–532. ISSN: 1558-2256. DOI: 10.1109/JPROC.2020.2976475. URL: https://ieeexplore.ieee.org/document/9043731 (visited on 08/12/2025).

[44] Peter U. Diehl and Matthew Cook. « Unsupervised learning of digit recognition using spike-timing-dependent plasticity ». *Frontiers in Computational Neuroscience* 9 (Aug. 3, 2015). Publisher: Frontiers. ISSN: 1662-5188. DOI: 10.3389/fncom.2015.00099. URL: https://www.frontiersin.org/journals/computational-neuroscience/articles/10.3389/fncom.2015.00099/full (visited on 03/28/2025).

[45] Pierpaolo Dini et al. « Overview of AI-Models and Tools in Embedded IIoT Applications ». *Electronics* 13.12 (Jan. 2024). Publisher: Multidisciplinary Digital Publishing Institute, p. 2322. ISSN: 2079-9292. DOI: 10.3390/electronics13122322. URL: https://www.mdpi.com/2079-9292/13/12/2322 (visited on 08/12/2025).

[46] Jason K Eshraghian et al. « Training spiking neural networks using lessons from deep learning ». *Proceedings of the IEEE* 111.9 (2023), pp. 1016–1054.

[47] Haowen Fang et al. « An Event-driven Neuromorphic System with Biologically Plausible Temporal Dynamics ». *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD). ISSN: 1558-2434. Nov. 2019, pp. 1–8. DOI: 10.1109/ICCAD45719.2019.8942083. URL: https://ieeexplore.ieee.org/abstract/document/8942083 (visited on 09/04/2024).

[48] Haowen Fang et al. « Encoding, model, and architecture: systematic optimization for spiking neural network in FPGAs ». *Proceedings of the 39th International Conference on Computer-Aided Design*. ICCAD '20. New York, NY, USA: Association for Computing Machinery, Dec. 17, 2020, pp. 1–9. ISBN: 978-1-4503-8026-3. DOI: 10.1145/3400302.3415608. URL: https://doi.org/10.1145/3400302.3415608 (visited on 02/06/2023).

[49] Wei Fang et al. *Incorporating Learnable Membrane Time Constant to Enhance Learning of Spiking Neural Networks*. Aug. 17, 2021. DOI: 10.48550/arXiv.2007.05785. arXiv: 2007.05785[cs]. URL: http://arxiv.org/abs/2007.05785 (visited on 05/09/2025).

[50] Edris Zaman Farsa, Arash Ahmadi, and Oliver Keszocze. « Reconfigurable Digital FPGA Implementations for Neuromorphic Computing: A Survey on Recent Advances and Future Directions ». *IEEE Transactions on Emerging Topics in Computational Intelligence* (2025). Conference Name: IEEE Transactions on Emerging Topics in Computational Intelligence, pp. 1–23. ISSN: 2471-285X. DOI: 10.1109/TETCI.2025.3551934. URL: https://ieeexplore.ieee.org/document/10946177/?arnumber=10946177 (visited on 04/03/2025).

[51] Edris Zaman Farsa et al. « A Low-Cost High-Speed Neuromorphic Hardware Based on Spiking Neural Network ». *IEEE Transactions on Circuits and Systems II: Express Briefs* 66.9 (Sept. 2019), pp. 1582–1586. ISSN: 1558-3791. DOI: 10.1109/TCSII.2019.2890846. URL: https://ieeexplore.ieee.org/document/8600358 (visited on 05/19/2025).

[52] Thomas Firmin, Pierre Boulet, and El-Ghazali Talbi. « Parallel hyperparameter optimization of spiking neural networks ». *Neurocomputing* 609 (Dec. 7, 2024), p. 128483. ISSN: 0925-2312. DOI: 10.1016/j.neucom.2024.128483. URL: https://www.sciencedirect.com/science/article/pii/S0925231224012542 (visited on 05/01/2025).

[53] Giuseppe Franco, Alessandro Pappalardo, and Nicholas J Fraser. *Xilinx/brevitas*. 2025. DOI: 10.5281/zenodo.3333552. URL: https://doi.org/10.5281/zenodo.3333552.

[54] Charlotte Frenkel and Giacomo Indiveri. « ReckOn: A 28nm Sub-mm2 Task-Agnostic Spiking Recurrent Neural Network Processor Enabling On-Chip Learning over Second-Long Timescales ». *2022 IEEE International Solid-State Circuits Conference (ISSCC)*. 2022 IEEE International Solid-State Circuits Conference (ISSCC). Vol. 65. ISSN: 2376-8606. Feb. 2022, pp. 1–3. DOI: 10.1109/ISSCC42614.2022.9731734. URL: https://ieeexplore.ieee.org/document/9731734 (visited on 04/18/2025).

[55] Ashish Gautam et al. « NeuroCoreX: An Open-Source FPGA-Based Spiking Neural Network Emulator with On-Chip Learning » (2025). Publisher: arXiv Version Number: 1. DOI: 10.48550/ARXIV.2506.14138. URL: https://arxiv.org/abs/2506.14138 (visited on 09/09/2025).

[56] Amir Gholami et al. *A Survey of Quantization Methods for Efficient Neural Network Inference*. June 21, 2021. DOI: 10.48550/arXiv.2103.13630. arXiv: 2103.13630[cs]. URL: http://arxiv.org/abs/2103.13630 (visited on 06/12/2025).

[57] Shaghayegh Gomar and Arash Ahmadi. « Digital Multiplierless Implementation of Biological Adaptive-Exponential Neuron Model ». *IEEE Transactions on Circuits and Systems I: Regular Papers* 61.4 (Apr. 2014), pp. 1206–1219. ISSN: 1558-0806. DOI: 10.1109/TCSI.2013.2286030. URL: https://ieeexplore.ieee.org/document/6656004 (visited on 05/21/2025).

[58] Hector A. Gonzalez et al. *SpiNNaker2: A Large-Scale Neuromorphic System for Event-Based and Asynchronous Machine Learning*. Jan. 9, 2024. DOI: 10.48550/arXiv.2401.04491. arXiv: 2401.04491[cs]. URL: http://arxiv.org/abs/2401.04491 (visited on 05/13/2025).

[59] R.M. Gray and D.L. Neuhoff. « Quantization ». *IEEE Transactions on Information Theory* 44.6 (Oct. 1998), pp. 2325–2383. ISSN: 1557-9654. DOI: 10.1109/18.720541. URL: https://ieeexplore.ieee.org/document/720541 (visited on 07/25/2025).

[60] Wenzhe Guo et al. « Neural Coding in Spiking Neural Networks: A Comparative Study for Robust Neuromorphic Systems ». *Frontiers in Neuroscience* 15 (Mar. 4, 2021). Publisher: Frontiers. ISSN: 1662-453X. DOI: 10.3389/fnins.2021.638474. URL: https://www.frontiersin.orghttps://www.frontiersin.org/journals/neuroscience/articles/10.3389/fnins.2021.638474/full (visited on 05/01/2025).

[61] Wenzhe Guo et al. « Toward the Optimal Design and FPGA Implementation of Spiking Neural Networks ». *IEEE Transactions on Neural Networks and Learning Systems* 33.8 (Aug. 2022). Conference Name: IEEE Transactions on Neural Networks and Learning Systems, pp. 3988–4002. ISSN: 2162-2388. DOI: 10.1109/TNNLS.2021.3055421.

[62] Yilong Guo et al. « Unsupervised Learning on Resistive Memory Array Based Spiking Neural Networks ». *Frontiers in Neuroscience* 13 (Aug. 6, 2019). Publisher: Frontiers, p. 812. ISSN: 1662-453X. DOI: 10.3389/fnins.2019.00812. URL: https://www.frontiersin.org/journals/neuroscience/articles/10.3389/fnins.2019.00812/full (visited on 05/11/2024).

[63] Shikhar Gupta, Arpan Vyas, and Gaurav Trivedi. « FPGA Implementation of Simplified Spiking Neural Network ». *2020 27th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*. 2020 27th IEEE International Conference on Electronics, Circuits and Systems (ICECS). Nov. 2020, pp. 1–4. DOI: 10.1109/ICECS49266.2020.9294790. URL: https://ieeexplore.ieee.org/abstract/document/9294790 (visited on 04/25/2025).

[64] Suyog Gupta et al. *Deep Learning with Limited Numerical Precision*. Feb. 9, 2015. DOI: 10.48550/arXiv.1502.02551. arXiv: 1502.02551[cs]. URL: http://arxiv.org/abs/1502.02551 (visited on 11/27/2025).

[65] Saeed Haghiri et al. « Multiplierless Implementation of Noisy Izhikevich Neuron With Low-Cost Digital Design ». *IEEE Transactions on Biomedical Circuits and Systems* 12.6 (Dec. 2018), pp. 1422–1430. ISSN: 1940-9990. DOI: 10.1109/TBCAS.2018.2868746. URL: https://ieeexplore.ieee.org/document/8454469 (visited on 05/16/2025).

[66] Saeed Haghiri et al. « High Speed and Low Digital Resources Implementation of Hodgkin-Huxley Neuronal Model Using Base-2 Functions ». *IEEE Transactions on Circuits and Systems I: Regular Papers* 68.1 (Jan. 2021), pp. 275–287. ISSN: 1558-0806. DOI: 10.1109/TCSI.2020.3026076. URL: https://ieeexplore.ieee.org/document/9235538 (visited on 05/16/2025).

[67] Ilyass Hammouamri, Ismail Khalfaoui-Hassani, and Timothée Masquelier. *Learning Delays in Spiking Neural Networks using Dilated Convolutions with Learnable Spacings*. Dec. 1, 2023. DOI: 10.48550/arXiv.2306.17670. arXiv: 2306.17670[cs]. URL: http://arxiv.org/abs/2306.17670 (visited on 05/07/2025).

[68] Jianhui Han et al. « Hardware implementation of spiking neural networks on FPGA ». *Tsinghua Science and Technology* 25.4 (Aug. 2020). Conference Name: Tsinghua Science and Technology, pp. 479–486. ISSN: 1007-0214. DOI: 10.26599/TST.2019.9010019. URL: https://ieeexplore.ieee.org/document/8954866/metrics#metrics (visited on 08/19/2024).

[69] Joon-Kyu Han et al. « Artificial Olfactory Neuron for an In-Sensor Neuromorphic Nose ». *Advanced Science* 9.18 (2022). _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/advs.202106017, p. 2106017. ISSN: 2198-3844. DOI: 10.1002/advs.202106017. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/advs.202106017 (visited on 05/01/2025).

[70] Mohsen Hayati et al. « Digital Multiplierless Realization of Two Coupled Biological Morris-Lecar Neuron Model ». *IEEE Transactions on Circuits and Systems I: Regular Papers* 62.7 (July 2015), pp. 1805–1814. ISSN: 1558-0806. DOI: 10.1109/TCSI.2015.2423794. URL: https://ieeexplore.ieee.org/document/7127066 (visited on 05/16/2025).

[71] Zhen He et al. « A Low-Cost FPGA Implementation of Spiking Extreme Learning Machine With On-Chip Reward-Modulated STDP Learning ». *IEEE Transactions on Circuits and Systems II: Express Briefs* 69.3 (Mar. 2022). Conference Name: IEEE Transactions on Circuits and Systems II: Express Briefs, pp. 1657–1661. ISSN: 1558-3791. DOI: 10.1109/TCSII.2021.3117699.

[72] D. O. Hebb. *The Organization of Behavior: A Neuropsychological Theory*. New York: Psychology Press, Apr. 11, 2005. 279 pp. ISBN: 978-1-4106-1240-3. DOI: 10.4324/9781410612403.

[73] Professor David Heeger. « Poisson Model of Spike Generation » ().

[74] Moslem Heidarpur et al. « CORDIC-SNN: On-FPGA STDP Learning With Izhikevich Neurons ». *IEEE Transactions on Circuits and Systems I: Regular Papers* 66.7 (July 2019), pp. 2651–2661. ISSN: 1558-0806. DOI: 10.1109/TCSI.2019.2899356. URL: https://ieeexplore.ieee.org/abstract/document/8660497 (visited on 04/25/2025).

[75] Don Lahiru Nirmal Hettiarachchi, Venkata Salini Priyamvada Davuluru, and Eric J. Balster. « Integer vs. Floating-Point Processing on Modern FPGA Technology ». *2020 10th Annual Computing and Communication Workshop and Conference (CCWC)*. 2020 10th Annual Computing and Communication Workshop and Conference (CCWC). Jan. 2020, pp. 0606–0612. DOI: 10.1109/CCWC47524.2020.9031118. URL: https://ieeexplore.ieee.org/abstract/document/9031118 (visited on 06/11/2025).

[76] Geoffrey E. Hinton, Simon Osindero, and Yee-Whye Teh. « A Fast Learning Algorithm for Deep Belief Nets ». *Neural Computation* 18.7 (July 2006), pp. 1527–1554. ISSN: 0899-7667. DOI: 10.1162/neco.2006.18.7.1527. URL: https://ieeexplore.ieee.org/abstract/document/6796673 (visited on 05/02/2025).

[77] A. L. Hodgkin and A. F. Huxley. « A quantitative description of membrane current and its application to conduction and excitation in nerve ». *The Journal of Physiology* 117.4 (Aug. 28, 1952), pp. 500–544. ISSN: 0022-3751. URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC1392413/.

[78] Sebastian Höppner et al. *The SpiNNaker 2 Processing Element Architecture for Hybrid Digital Neuromorphic Computing*. Aug. 15, 2022. DOI: 10.48550/arXiv.2103.08392. arXiv: 2103.08392[cs]. URL: http://arxiv.org/abs/2103.08392 (visited on 05/13/2025).

[79] S. G. Hu et al. « Quantized STDP-based online-learning spiking neural network ». *Neural Comput. Appl.* 33.19 (Oct. 1, 2021), pp. 12317–12332. ISSN: 0941-0643. DOI: 10.1007/s00521-021-05832-y. URL: https://doi.org/10.1007/s00521-021-05832-y (visited on 07/29/2025).

[80] Dongsung Huh and Terrence J. Sejnowski. *Gradient Descent for Spiking Neural Networks*. June 19, 2017. DOI: 10.48550/arXiv.1706.04698. arXiv: 1706.04698[q-bio]. URL: http://arxiv.org/abs/1706.04698 (visited on 03/27/2025).

[81] Innatera. *Spiking Neural Processor T1*. URL: https://innatera.com/products/spiking-neural-processor-t1 (visited on 05/12/2025).

[82] Murat Isik. *A Survey of Spiking Neural Network Accelerator on FPGA*. July 8, 2023. DOI: 10.48550/arXiv.2307.03910. arXiv: 2307.03910[cs]. URL: http://arxiv.org/abs/2307.03910 (visited on 04/14/2025).

[83] E.M. Izhikevich. « Simple model of spiking neurons ». *IEEE Transactions on Neural Networks* 14.6 (Nov. 2003), pp. 1569–1572. ISSN: 1045-9227. DOI: 10.1109/TNN.2003.820440.

[84] E.M. Izhikevich. « Which Model to Use for Cortical Spiking Neurons? » *IEEE Transactions on Neural Networks* 15.5 (Sept. 2004), pp. 1063–1070. ISSN: 1045-9227. DOI: 10.1109/TNN.2004.832719. URL: http://ieeexplore.ieee.org/document/1333071/ (visited on 05/02/2025).

[85] Eugene M. Izhikevich and Richard FitzHugh. « FitzHugh-Nagumo model ». *Scholarpedia* 1.9 (Sept. 23, 2006), p. 1349. ISSN: 1941-6016. DOI: 10.4249/scholarpedia.1349. URL: http://www.scholarpedia.org/article/FitzHugh-Nagumo_model (visited on 05/02/2025).

[86] Benoit Jacob et al. *Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference*. Dec. 15, 2017. DOI: 10.48550/arXiv.1712.05877. arXiv: 1712.05877[cs]. URL: http://arxiv.org/abs/1712.05877 (visited on 11/27/2025).

[87] Angel Jiménez-Fernández et al. « A Binaural Neuromorphic Auditory Sensor for FPGA: A Spike Signal Processing Approach ». *IEEE Transactions on Neural Networks and Learning Systems* 28.4 (Apr. 2017). Conference Name: IEEE Transactions on Neural Networks and Learning Systems, pp. 804–818. ISSN: 2162-2388. DOI: 10.1109/TNNLS.2016.2583223.

[88] Ehsan Jokar and Hamid Soleimani. « Digital Multiplierless Realization of a Calcium-Based Plasticity Model ». *IEEE Transactions on Circuits and Systems II: Express Briefs* 64.7 (July 2017), pp. 832–836. ISSN: 1558-3791. DOI: 10.1109/TCSII.2016.2621823. URL: https://ieeexplore.ieee.org/document/7707387 (visited on 05/21/2025).

[89] Tatsuya Kaneko, Hiroshi Momose, and Tetsuya Asai. « An FPGA Accelerator for Embedded Microcontrollers Implementing a Ternarized Backpropagation Algorithm ». *2019 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. 2019 International Conference on ReConFigurable Computing and FPGAs (ReConFig). ISSN: 2640-0472. Dec. 2019, pp. 1–8. DOI: 10.1109/ReConFig48160.2019.8994795. URL: https://ieeexplore.ieee.org/abstract/document/8994795 (visited on 05/21/2025).

[90] Mehrzad Karamimanesh et al. « Spiking neural networks on FPGA: A survey of methodologies and recent advancements ». *Neural Networks* 186 (June 1, 2025), p. 107256. ISSN: 0893-6080. DOI: 10.1016/j.neunet.2025.107256. URL: https://www.sciencedirect.com/science/article/pii/S0893608025001352 (visited on 04/29/2025).

[91] Alireza Khodamoradi, Kristof Denolf, and Ryan Kastner. « S2N2: A FPGA Accelerator for Streaming Spiking Neural Networks ». *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '21. New York, NY, USA: Association for Computing Machinery, Feb. 17, 2021, pp. 194–205. ISBN: 978-1-4503-8218-2. DOI: 10.1145/3431920.3439283. URL: https://dl.acm.org/doi/10.1145/3431920.3439283 (visited on 08/19/2024).

[92] Alex Krizhevsky, Geoffrey Hinton, et al. « Learning multiple layers of features from tiny images » (2009).

[93] Shivani Kuninti and S Rooban. « Backpropagation Algorithm and its Hardware Implementations: A Review ». *Journal of Physics: Conference Series* 1804.1 (Feb. 2021). Publisher: IOP Publishing, p. 012169. ISSN: 1742-6596. DOI: 10.1088/1742-6596/1804/1/012169. URL: https://dx.doi.org/10.1088/1742-6596/1804/1/012169 (visited on 05/21/2025).

[94] Corey Lammie et al. « Efficient FPGA Implementations of Pair and Triplet-Based STDP for Neuromorphic Architectures ». *IEEE Transactions on Circuits and Systems I: Regular Papers* 66.4 (Apr. 2019). Conference Name: IEEE Transactions on Circuits and Systems I: Regular Papers, pp. 1558–1570. ISSN: 1558-0806. DOI: 10.1109/TCSI.2018.2881753.

[95] L Lapicque. « Recherches quantitatives sur l'excitation electrique des nerfs ». *Journal de physiologie et de pathologie générale* 9 (1907), pp. 620–635.

[96] J. Lazzaro et al. « Silicon auditory processors as computer peripherals ». *IEEE Transactions on Neural Networks* 4.3 (May 1993). Conference Name: IEEE Transactions on Neural Networks, pp. 523–528. ISSN: 1941-0093. DOI: 10.1109/72.217193.

[97] Harold Lecar. « Morris-Lecar Model ». *Scholarpedia* 2.10 (Oct. 23, 2007), p. 1333. ISSN: 1941-6016. DOI: 10.4249/scholarpedia.1333. URL: http://scholarpedia.org/article/Morris-Lecar_Model,%20http://www.scholarpedia.org/article/Morris-Lecar_Model (visited on 04/18/2022).

[98] Y. Lecun et al. « Gradient-based learning applied to document recognition ». *Proceedings of the IEEE* 86.11 (Nov. 1998), pp. 2278–2324. ISSN: 1558-2256. DOI: 10.1109/5.726791. URL: https://ieeexplore.ieee.org/abstract/document/726791 (visited on 05/02/2025).

[99] Yann LeCun and Corinna Cortes. « MNIST handwritten digit database » (2010). URL: http://yann.lecun.com/exdb/mnist/ (visited on 04/12/2024).

[100] Chankyu Lee et al. « Enabling Spike-Based Backpropagation for Training Deep Neural Network Architectures ». *Frontiers in Neuroscience* 14 (Feb. 28, 2020). Publisher: Frontiers. ISSN: 1662-453X. DOI: 10.3389/fnins.2020.00119. URL: https://www.frontiersin.orghttps://www.frontiersin.org/journals/neuroscience/articles/10.3389/fnins.2020.00119/full (visited on 05/05/2025).

[101] Jun Haeng Lee, Tobi Delbruck, and Michael Pfeiffer. « Training Deep Spiking Neural Networks Using Backpropagation ». *Frontiers in Neuroscience* 10 (Nov. 8, 2016). Publisher: Frontiers. ISSN: 1662-453X. DOI: 10.3389/fnins.2016.00508. URL: https://www.frontiersin.org/journals/neuroscience/articles/10.3389/fnins.2016.00508/full (visited on 03/28/2025).

[102] Alexander J. Leigh, Moslem Heidarpur, and Mitra Mirhassani. « A High-Accuracy Digital Implementation of the Morris–Lecar Neuron With Variable Physiological Parameters ». *IEEE Transactions on Circuits and Systems II: Express Briefs* 69.10 (Oct. 2022), pp. 4138–4142. ISSN: 1558-3791. DOI: 10.1109/TCSII.2022.3187623. URL: https://ieeexplore.ieee.org/document/9812745 (visited on 05/16/2025).

[103] Alexander J. Leigh, Mitra Mirhassani, and Roberto Muscedere. « An Efficient Spiking Neuron Hardware System Based on the Hardware-Oriented Modified Izhikevich Neuron (HOMIN) Model ». *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2020 IEEE International Symposium on Circuits and Systems (ISCAS). ISSN: 2158-1525. Oct. 2020, pp. 1–2. DOI: 10.1109/ISCAS45731.2020.9181154. URL: https://ieeexplore.ieee.org/document/9181154 (visited on 05/16/2025).

[104] Edgar Lemaire et al. « An FPGA-Based Hybrid Neural Network Accelerator for Embedded Satellite Image Classification ». *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2020 IEEE International Symposium on Circuits and Systems (ISCAS). ISSN: 2158-1525. Oct. 2020, pp. 1–5. DOI: 10.1109/ISCAS45731.2020.9180625. URL: https://ieeexplore.ieee.org/abstract/document/9180625 (visited on 05/20/2025).

[105] Edgar Lemaire et al. « An Analytical Estimation of Spiking Neural Networks Energy Efficiency ». *Neural Information Processing*. Ed. by Mohammad Tanveer et al. Cham: Springer International Publishing, 2023, pp. 574–587. ISBN: 978-3-031-30105-6. DOI: 10.1007/978-3-031-30105-6_48.

[106] Gianluca Leone, Luigi Raffo, and Paolo Meloni. « A Bandwidth-Efficient Emulator of Biologically-Relevant Spiking Neural Networks on FPGA ». *IEEE Access* 10 (2022), pp. 76780–76793. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2022.3192826. URL: https://ieeexplore.ieee.org/document/9834334 (visited on 05/21/2025).

[107] Haomin Li et al. « NeuronQuant: Accurate and Efficient Post-Training Quantization for Spiking Neural Networks ». *Proceedings of the 30th Asia and South Pacific Design Automation Conference*. New York, NY, USA: Association for Computing Machinery, Mar. 4, 2025, pp. 734–740. ISBN: 979-8-4007-0635-6. URL: https://doi.org/10.1145/3658617.3697716 (visited on 07/26/2025).

[108] Sixu Li et al. « A Fast and Energy-Efficient SNN Processor With Adaptive Clock/Event-Driven Computation Scheme and Online Learning ». *IEEE Transactions on Circuits and Systems I: Regular Papers* 68.4 (Apr. 2021). Conference Name: IEEE Transactions on Circuits and Systems I: Regular Papers, pp. 1543–1552. ISSN: 1558-0806. DOI: 10.1109/TCSI.2021.3052885. URL: https://ieeexplore.ieee.org/abstract/document/9336327 (visited on 04/22/2024).

[109] Patrick Lichtsteiner, Christoph Posch, and Tobi Delbruck. « A 128\times 128 120 dB 15 ?s Latency Asynchronous Temporal Contrast Vision Sensor ». *IEEE Journal of Solid-State Circuits* 43.2 (Feb. 2008), pp. 566–576. ISSN: 1558-173X. DOI: 10.1109/JSSC.2007.914337. URL: https://ieeexplore.ieee.org/abstract/document/4444573 (visited on 04/17/2025).

[110] Hanwen Liu et al. « A Low Power and Low Latency FPGA-Based Spiking Neural Network Accelerator ». *2023 International Joint Conference on Neural Networks (IJCNN)*. 2023 International Joint Conference on Neural Networks (IJCNN). ISSN: 2161-4407. June 2023, pp. 1–8. DOI: 10.1109/IJCNN54540.2023.10191153. URL: https://ieeexplore.ieee.org/document/10191153 (visited on 04/18/2024).

[111] Jihong Liu and Chengyuan Wang. « A Survey of Neuromorphic Engineering–Biological Nervous Systems Realized on Silicon ». *2009 IEEE Circuits and Systems International Conference on Testing and Diagnosis*. 2009 IEEE Circuits and Systems International Conference on Testing and Diagnosis. ISSN: 2324-8491. Apr. 2009, pp. 1–4. DOI: 10.1109/CAS-ICTD.2009.4960772. URL: https://ieeexplore.ieee.org/abstract/document/4960772 (visited on 05/12/2024).

[112] Shiya Liu, Nima Mohammadi, and Yang Yi. « Quantization-Aware Training of Spiking Neural Networks for Energy-Efficient Spectrum Sensing on Loihi Chip ». *IEEE Transactions on Green Communications and Networking* 8.2 (June 2024). Conference Name: IEEE Transactions on Green Communications and Networking, pp. 827–838. ISSN: 2473-2400. DOI: 10.1109/TGCN.2023.3337748. URL: https://ieeexplore.ieee.org/document/10332927 (visited on 03/16/2025).

[113] Wei Liu et al. « SC-IZ: A Low-Cost Biologically Plausible Izhikevich Neuron for Large-Scale Neuromorphic Systems Using Stochastic Computing ». *Electronics* 13.5 (Jan. 2024). Number: 5 Publisher: Multidisciplinary Digital Publishing Institute, p. 909. ISSN: 2079-9292. DOI: 10.3390/electronics13050909. URL: https://www.mdpi.com/2079-9292/13/5/909 (visited on 05/16/2025).

[114] Yijun Liu et al. « FPGA-NHAP: A General FPGA-Based Neuromorphic Hardware Acceleration Platform With High Speed and Low Power ». *IEEE Transactions on Circuits and Systems I: Regular Papers* 69.6 (June 2022). Conference Name: IEEE Transactions on Circuits and Systems I: Regular Papers, pp. 2553–2566. ISSN: 1558-0806. DOI: 10.1109/TCSI.2022.3160693. URL: https://ieeexplore.ieee.org/document/9745256 (visited on 04/18/2024).

[115] Yu Liu, Sai Sourabh Yenamachintala, and Peng Li. « Energy-efficient FPGA Spiking Neural Accelerators with Supervised and Unsupervised Spike-timing-dependent-Plasticity ». *J. Emerg. Technol. Comput. Syst.* 15.3 (May 30, 2019), 27:1–27:19. ISSN: 1550-4832. DOI: 10.1145/3313866. URL: https://dl.acm.org/doi/10.1145/3313866 (visited on 04/18/2025).

[116] Zechun Liu et al. *Nonuniform-to-Uniform Quantization: Towards Accurate Quantization via Generalized Straight-Through Estimation*. Apr. 7, 2022. DOI: 10.48550/arXiv.2111.14826. arXiv: 2111.14826[cs]. URL: http://arxiv.org/abs/2111.14826 (visited on 07/25/2025).

[117] Thomas Louis. « Conventional or bio-inspired? Strategies for optimizing neural network energy efficiency in resource-constrained environments » (), p. 53.

[118] De Ma et al. « Darwin: A neuromorphic hardware co-processor based on spiking neural networks ». *Journal of Systems Architecture* 77 (June 1, 2017), pp. 43–51. ISSN: 1383-7621. DOI: 10.1016/j.sysarc.2017.01.003. URL: https://www.sciencedirect.com/science/article/pii/S1383762117300231 (visited on 02/02/2023).

[119] Yifang Ma et al. « Artificial intelligence applications in the development of autonomous vehicles: a survey ». *IEEE/CAA Journal of Automatica Sinica* 7.2 (Mar. 2020). Conference Name: IEEE/CAA Journal of Automatica Sinica, pp. 315–329. ISSN: 2329-9274. DOI: 10.1109/JAS.2020.1003021. URL: https://ieeexplore.ieee.org/abstract/document/9016391 (visited on 04/18/2024).

[120] Wolfgang Maass. « Networks of spiking neurons: The third generation of neural network models ». *Neural Networks* 10.9 (Dec. 1, 1997), pp. 1659–1671. ISSN: 0893-6080. DOI: 10.1016/S0893-6080(97)00011-7. URL: https://www.sciencedirect.com/science/article/pii/S0893608097000117 (visited on 04/19/2024).

[121] Wolfgang Maass. « Networks of spiking neurons: The third generation of neural network models ». *Neural Networks* 10.9 (Dec. 1, 1997), pp. 1659–1671. ISSN: 0893-6080. DOI: 10.1016/S0893-6080(97)00011-7. URL: https://www.sciencedirect.com/science/article/pii/S0893608097000117 (visited on 05/05/2025).

[122] T. Makino. « A Discrete-Event Neural Network Simulator for General Neuron Models ». *Neural Computing & Applications* 11.3 (June 1, 2003), pp. 210–223. ISSN: 1433-3058. DOI: 10.1007/s00521-003-0358-z. URL: https://doi.org/10.1007/s00521-003-0358-z (visited on 04/27/2022).

[123] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[124] Christian Mayr, Sebastian Hoeppner, and Steve Furber. *SpiNNaker 2: A 10 Million Core Processor System for Brain Simulation and Machine Learning*. Nov. 6, 2019. DOI: 10.48550/arXiv.1911.02385. arXiv: 1911.02385[cs]. URL: http://arxiv.org/abs/1911.02385 (visited on 05/13/2025).

[125] Warren S. McCulloch and Walter Pitts. « A logical calculus of the ideas immanent in nervous activity ». *The bulletin of mathematical biophysics* 5.4 (Dec. 1, 1943), pp. 115–133. ISSN: 1522-9602. DOI: 10.1007/BF02478259. URL: https://doi.org/10.1007/BF02478259 (visited on 05/01/2025).

[126] C. Mead. « Neuromorphic electronic systems ». *Proceedings of the IEEE* 78.10 (Oct. 1990). Conference Name: Proceedings of the IEEE, pp. 1629–1636. ISSN: 1558-2256. DOI: 10.1109/5.58356.

[127] Paul Merolla et al. « A Multicast Tree Router for Multichip Neuromorphic Systems ». *IEEE Transactions on Circuits and Systems I: Regular Papers* 61.3 (Mar. 2014). Conference Name: IEEE Transactions on Circuits and Systems I: Regular Papers, pp. 820–833. ISSN: 1558-0806. DOI: 10.1109/TCSI.2013.2284184.

[128] *Midjourney*. Midjourney. URL: https://www.midjourney.com/website (visited on 08/12/2025).

[129] Saras Mani Mishra et al. « FPGA-Based Adaptive LIF Neuron for High-Speed, Energy-Efficient Spiking Neural Network ». *IEEE Transactions on Circuits and Systems for Artificial Intelligence* (2025), pp. 1–11. ISSN: 2996-6647. DOI: 10.1109/TCASAI.2025.3568365. URL: https://ieeexplore.ieee.org/document/10994369/ (visited on 05/19/2025).

[130] Saber Moradi et al. « A memory-efficient routing method for large-scale spiking neural networks ». *2013 European Conference on Circuit Theory and Design (ECCTD)*. 2013 European Conference on Circuit Theory and Design (ECCTD). Sept. 2013, pp. 1–4. DOI: 10.1109/ECCTD.2013.6662203.

[131] Saber Moradi et al. « A Scalable Multicore Architecture With Heterogeneous Memory Structures for Dynamic Neuromorphic Asynchronous Processors (DYNAPs) ». *IEEE Transactions on Biomedical Circuits and Systems* 12.1 (Feb. 2018). Conference Name: IEEE Transactions on Biomedical Circuits and Systems, pp. 106–122. ISSN: 1940-9990. DOI: 10.1109/TBCAS.2017.2759700.

[132] Markus Nagel et al. *A White Paper on Neural Network Quantization*. June 15, 2021. DOI: 10.48550/arXiv.2106.08295. arXiv: 2106.08295[cs]. URL: http://arxiv.org/abs/2106.08295 (visited on 03/25/2025).

[133] Aishwarya Natarajan and Jennifer Hasler. « Hodgkin–Huxley Neuron and FPAA Dynamics ». *IEEE Transactions on Biomedical Circuits and Systems* 12.4 (Aug. 2018). Conference Name: IEEE Transactions on Biomedical Circuits and Systems, pp. 918–926. ISSN: 1940-9990. DOI: 10.1109/TBCAS.2018.2837055. URL: https://ieeexplore.ieee.org/abstract/document/8410807 (visited on 05/11/2024).

[134] Emre O. Neftci, Hesham Mostafa, and Friedemann Zenke. « Surrogate Gradient Learning in Spiking Neural Networks: Bringing the Power of Gradient-Based Optimization to Spiking Neural Networks ». *IEEE Signal Processing Magazine* 36.6 (Nov. 2019). Conference Name: IEEE Signal Processing Magazine, pp. 51–63. ISSN: 1558-0792. DOI: 10.1109/MSP.2019.2931595. URL: https://ieeexplore.ieee.org/document/8891809 (visited on 04/12/2024).

[135] *NeuroBench: Advancing Neuromorphic Computing through Collaborative, Fair and Representative Benchmarking*. Apr. 15, 2023. arXiv: 2304.04640[cs]. URL: http://arxiv.org/abs/2304.04640 (visited on 04/20/2023).

[136] Open Neuromorphic. *Akida - BrainChip*. URL: https://open-neuromorphic.org/neuromorphic-computing/hardware/akida-brainchip/ (visited on 11/29/2025).

[137] M. Nouri et al. « Digital multiplierless implementation of the biological FitzHugh–Nagumo model ». *Neurocomputing* 165 (Oct. 1, 2015), pp. 468–476. ISSN: 0925-2312. DOI: 10.1016/j.neucom.2015.03.084. URL: https://www.sciencedirect.com/science/article/pii/S092523121500466X (visited on 05/16/2025).

[138] Marco Aurelio Nuno-Maganda et al. « Hardware implementation of Spiking Neural Network classifiers based on backpropagation-based learning algorithms ». *2009 International Joint Conference on Neural Networks*. 2009 International Joint Conference on Neural Networks. ISSN: 2161-4407. June 2009, pp. 2294–2301. DOI: 10.1109/IJCNN.2009.5178912. URL: https://ieeexplore.ieee.org/abstract/document/5178912 (visited on 05/21/2025).

[139] OpenAI et al. *GPT-4 Technical Report*. Mar. 4, 2024. DOI: 10.48550/arXiv.2303.08774. arXiv: 2303.08774[cs]. URL: http://arxiv.org/abs/2303.08774 (visited on 08/12/2025).

[140] Garrick Orchard et al. « Converting Static Image Datasets to Spiking Neuromorphic Datasets Using Saccades ». *Frontiers in Neuroscience* 9 (Nov. 16, 2015). Publisher: Frontiers, p. 437. ISSN: 1662-453X. DOI: 10.3389/fnins.2015.00437. URL: https://www.frontiersin.orghttps://www.frontiersin.org/journals/neuroscience/articles/10.3389/fnins.2015.00437/full (visited on 04/25/2025).

[141] Garrick Orchard et al. « Efficient Neuromorphic Signal Processing with Loihi 2 ». *2021 IEEE Workshop on Signal Processing Systems (SiPS)*. 2021 IEEE Workshop on Signal Processing Systems (SiPS). ISSN: 2374-7390. Oct. 2021, pp. 254–259. DOI: 10.1109/SiPS52927.2021.00053. URL: https://ieeexplore.ieee.org/abstract/document/9605018 (visited on 05/13/2025).

[142] Francisco Ortega-Zamorano et al. « Efficient Implementation of the Backpropagation Algorithm in FPGAs and Microcontrollers ». *IEEE Transactions on Neural Networks and Learning Systems* 27.9 (Sept. 2016), pp. 1840–1850. ISSN: 2162-2388. DOI: 10.1109/TNNLS.2015.2460991. URL: https://ieeexplore.ieee.org/abstract/document/7192642 (visited on 05/21/2025).

[143] Eustace Painkras et al. « SpiNNaker: A 1-W 18-Core System-on-Chip for Massively-Parallel Neural Network Simulation ». *IEEE Journal of Solid-State Circuits* 48.8 (Aug. 2013). Conference Name: IEEE Journal of Solid-State Circuits, pp. 1943–1953. ISSN: 1558-173X. DOI: 10.1109/JSSC.2013.2259038.

[144] Adam Paszke et al. *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. Dec. 3, 2019. DOI: 10.48550/arXiv.1912.01703. arXiv: 1912.01703[cs]. URL: http://arxiv.org/abs/1912.01703 (visited on 03/21/2025).

[145] Cameron Patterson et al. « Scalable communications for a million-core neural processing architecture ». *Journal of Parallel and Distributed Computing*. Communication Architectures for Scalable Systems 72.11 (Nov. 1, 2012), pp. 1507–1520. ISSN: 0743-7315. DOI: 10.1016/j.jpdc.2012.01.016. URL: https://www.sciencedirect.com/science/article/pii/S0743731512000287 (visited on 01/27/2023).

[146] Christian Pehle et al. « The BrainScaleS-2 Accelerated Neuromorphic System With Hybrid Plasticity ». *Frontiers in Neuroscience* 16 (Feb. 24, 2022). Publisher: Frontiers. ISSN: 1662-453X. DOI: 10.3389/fnins.2022.795876. URL: https://www.frontiersin.org/journals/neuroscience/articles/10.3389/fnins.2022.795876/full (visited on 05/12/2025).

[147] Nicolas Perez-Nieves and Dan F. M. Goodman. *Sparse Spiking Gradient Descent*. Jan. 13, 2022. DOI: 10.48550/arXiv.2105.08810. arXiv: 2105.08810[cs]. URL: http://arxiv.org/abs/2105.08810 (visited on 04/12/2025).

[148] Jean-Pascal Pfister and Wulfram Gerstner. « Triplets of Spikes in a Model of Spike Timing-Dependent Plasticity ». *Journal of Neuroscience* 26.38 (Sept. 20, 2006). Publisher: Society for Neuroscience Section: Articles, pp. 9673–9682. ISSN: 0270-6474, 1529-2401. DOI: 10.1523/JNEUROSCI.1425-06.2006. URL: https://www.jneurosci.org/content/26/38/9673 (visited on 05/21/2025).

[149] Quoc Trung Pham et al. « A review of SNN implementation on FPGA ». *2021 International Conference on Multimedia Analysis and Pattern Recognition (MAPR)*. 2021 International Conference on Multimedia Analysis and Pattern Recognition (MAPR). Oct. 2021, pp. 1–6. DOI: 10.1109/MAPR53640.2021.9585245.

[150] Patrick Plagwitz et al. « SNN vs. CNN Implementations on FPGAs: An Empirical Evaluation ». *Applied Reconfigurable Computing. Architectures, Tools, and Applications*. Ed. by Iouliia Skliarova et al. Cham: Springer Nature Switzerland, 2024, pp. 3–18. ISBN: 978-3-031-55673-9. DOI: 10.1007/978-3-031-55673-9_1.

[151] Christoph Posch et al. « Retinomorphic Event-Based Vision Sensors: Bioinspired Cameras With Spiking Output ». *Proceedings of the IEEE* 102.10 (Oct. 2014). Conference Name: Proceedings of the IEEE, pp. 1470–1484. ISSN: 1558-2256. DOI: 10.1109/JPROC.2014.2346153. URL: https://ieeexplore.ieee.org/abstract/document/6887319 (visited on 04/03/2025).

[152] Junran Pu et al. « A Low-Cost High-Throughput Digital Design of Biorealistic Spiking Neuron ». *IEEE Transactions on Circuits and Systems II: Express Briefs* 68.4 (Apr. 2021), pp. 1398–1402. ISSN: 1558-3791. DOI: 10.1109/TCSII.2020.3023825. URL: https://ieeexplore.ieee.org/document/9195442 (visited on 05/16/2025).

[153] Rachmad Vidya Wicaksana Putra and Muhammad Shafique. « Q-SpiNN: A Framework for Quantizing Spiking Neural Networks ». *2021 International Joint Conference on Neural Networks (IJCNN)*. 2021 International Joint Conference on Neural Networks (IJCNN). ISSN: 2161-4407. July 2021, pp. 1–8. DOI: 10.1109/IJCNN52387.2021.9534087. URL: https://ieeexplore.ieee.org/document/9534087 (visited on 07/26/2025).

[154] Ole Richter et al. « DYNAP-SE2: a scalable multi-core dynamic neuromorphic asynchronous spiking neural network processor ». *Neuromorphic Computing and Engineering* 4.1 (Jan. 2024). Publisher: IOP Publishing, p. 014003. ISSN: 2634-4386. DOI: 10.1088/2634-4386/ad1cd7. URL: https://dx.doi.org/10.1088/2634-4386/ad1cd7 (visited on 05/12/2025).

[155] Ole Richter et al. « Speck: A Smart event-based Vision Sensor with a low latency 327K Neuron Convolutional Neuronal Network Processing Pipeline ». *Nature Communications* 15.1 (May 25, 2024), p. 4464. ISSN: 2041-1723. DOI: 10.1038/s41467-024-47811-6. arXiv: 2304.06793[cs]. URL: http://arxiv.org/abs/2304.06793 (visited on 05/13/2025).

[156] Bodo Rueckauer et al. « Conversion of Continuous-Valued Deep Networks to Efficient Event-Driven Networks for Image Classification ». *Frontiers in Neuroscience* 11 (Dec. 7, 2017). Publisher: Frontiers. ISSN: 1662-453X. DOI: 10.3389/fnins.2017.00682. URL: https://www.frontiersin.orghttps://www.frontiersin.org/journals/neuroscience/articles/10.3389/fnins.2017.00682/full (visited on 05/05/2025).

[157] P.-F. Ruedi et al. « A 128 × 128 pixel 120-dB dynamic-range vision-sensor chip for image contrast and orientation extraction ». *IEEE Journal of Solid-State Circuits* 38.12 (Dec. 2003). Conference Name: IEEE Journal of Solid-State Circuits, pp. 2325–2333. ISSN: 1558-173X. DOI: 10.1109/JSSC.2003.819169.

[158] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. « Learning representations by back-propagating errors ». *Nature* 323.6088 (Oct. 1986). Publisher: Nature Publishing Group, pp. 533–536. ISSN: 1476-4687. DOI: 10.1038/323533a0. URL: https://www.nature.com/articles/323533a0 (visited on 03/28/2025).

[159] Aurélie Saulquin et al. « ModNEF : An Open Source Modular Neuromorphic Emulator for FPGA for Low-Power In-Edge Artificial Intelligence ». *ACM Trans. Archit. Code Optim.* (Apr. 16, 2025). Just Accepted. ISSN: 1544-3566. DOI: 10.1145/3730581. URL: https://dl.acm.org/doi/10.1145/3730581 (visited on 04/17/2025).

[160] Laela Sayigh et al. « The Watkins Marine Mammal Sound Database: An online, freely accessible resource ». *Proceedings of Meetings on Acoustics* 27.1 (Feb. 21, 2017), p. 040013. ISSN: 1939-800X. DOI: 10.1121/2.0000358. URL: https://doi.org/10.1121/2.0000358 (visited on 08/06/2025).

[161] Johannes Schemmel et al. « A wafer-scale neuromorphic hardware system for large-scale neural modeling ». *2010 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2010 IEEE International Symposium on Circuits and Systems (ISCAS). ISSN: 2158-1525. May 2010, pp. 1947–1950. DOI: 10.1109/ISCAS.2010.5536970. URL: https://ieeexplore.ieee.org/document/5536970 (visited on 04/19/2024).

[162] K. Schreiber et al. « Closed-loop experiments on the BrainScaleS-2 architecture ». *Proceedings of the 2020 Annual Neuro-Inspired Computational Elements Workshop*. NICE '20. New York, NY, USA: Association for Computing Machinery, June 18, 2020, pp. 1–3. ISBN: 978-1-4503-7718-8. DOI: 10.1145/3381755.3381776. URL: https://dl.acm.org/doi/10.1145/3381755.3381776 (visited on 06/06/2025).

[163] Catherine Schuman et al. « Evaluating Encoding and Decoding Approaches for Spiking Neuromorphic Systems ». *Proceedings of the International Conference on Neuromorphic Systems 2022*. ICONS '22. New York, NY, USA: Association for Computing Machinery, Sept. 7, 2022, pp. 1–9. ISBN: 978-1-4503-9789-6. DOI: 10.1145/3546790.3546792. URL: https://dl.acm.org/doi/10.1145/3546790.3546792 (visited on 05/01/2025).

[164] Catherine D. Schuman et al. *A Survey of Neuromorphic Computing and Neural Networks in Hardware*. May 19, 2017. DOI: 10.48550/arXiv.1705.06963. arXiv: 1705.06963[cs]. URL: http://arxiv.org/abs/1705.06963 (visited on 04/19/2024).

[165] Farzin Shama, Saeed Haghiri, and Mohammad Amin Imani. « FPGA Realization of Hodgkin-Huxley Neuronal Model ». *IEEE Transactions on Neural Systems and Rehabilitation Engineering* 28.5 (May 2020). Conference Name: IEEE Transactions on Neural Systems and Rehabilitation Engineering, pp. 1059–1068. ISSN: 1558-0210. DOI: 10.1109/TNSRE.2020.2980475. URL: https://ieeexplore.ieee.org/abstract/document/9037076 (visited on 04/23/2024).

[166] Mst Shamim Ara Shawkat and Sajid Hasan. « Review of Hardware Implementation of SNN ». *2023 IEEE 66th International Midwest Symposium on Circuits and Systems (MWSCAS)*. 2023 IEEE 66th International Midwest Symposium on Circuits and Systems (MWSCAS). ISSN: 1558-3899. Aug. 2023, pp. 743–747. DOI: 10.1109/MWSCAS57524.2023.10405990. URL: https://ieeexplore.ieee.org/abstract/document/10405990 (visited on 05/11/2024).

[167] Shirui Sheng et al. « An Energy-Efficient and High-Accuracy Spiking Neural Network Utilizing Asynchronous CORDIC for On-FPGA STDP Learning ». *2024 IEEE Asia Pacific Conference on Circuits and Systems (APCCAS)*. 2024 IEEE Asia Pacific Conference on Circuits and Systems (APCCAS). ISSN: 2768-3516. Nov. 2024, pp. 571–575. DOI: 10.1109/APCCAS62602.2024.10808853. URL: https://ieeexplore.ieee.org/abstract/document/10808853 (visited on 05/21/2025).

[168] Sumit Bam Shrestha and Garrick Orchard. *SLAYER: Spike Layer Error Reassignment in Time*. Sept. 5, 2018. DOI: 10.48550/arXiv.1810.08646. arXiv: 1810.08646[cs]. URL: http://arxiv.org/abs/1810.08646 (visited on 05/07/2025).

[169] Shakhrul Iman Siam et al. « Artificial Intelligence of Things: A Survey ». *ACM Trans. Sen. Netw.* 21.1 (Jan. 27, 2025), 9:1–9:75. ISSN: 1550-4859. DOI: 10.1145/3690639. URL: https://dl.acm.org/doi/10.1145/3690639 (visited on 08/12/2025).

[170] Matthew Sibanda, Ernest Bhero, and John Agee. « AI Edge Processing - A Review of Distributed Embedded Systems ». *2023 31st Southern African Universities Power Engineering Conference (SAUPEC)*. 2023 31st Southern African Universities Power Engineering Conference (SAUPEC). Jan. 2023, pp. 1–6. DOI: 10.1109/SAUPEC57889.2023.10057624. URL: https://ieeexplore.ieee.org/abstract/document/10057624 (visited on 08/12/2025).

[171] Norman Sieroka, Hans Günter Dosch, and André Rupp. « Semirealistic models of the cochlea ». *The Journal of the Acoustical Society of America* 120.1 (July 1, 2006), pp. 297–304. ISSN: 0001-4966. DOI: 10.1121/1.2204438. URL: https://doi.org/10.1121/1.2204438 (visited on 04/12/2025).

[172] Jeongyong Sim, Sunghwan Joo, and Seong-Ook Jung. « Comparative Analysis of Digital STDP Learning Circuits Designed Using Counter and Shift Register ». *2019 34th International Technical Conference on Circuits/Systems, Computers and Communications (ITC-CSCC)*. 2019 34th International Technical Conference on Circuits/Systems, Computers and Communications (ITC-CSCC). June 2019, pp. 1–4. DOI: 10.1109/ITC-CSCC.2019.8793424. URL: https://ieeexplore.ieee.org/abstract/document/8793424 (visited on 05/21/2025).

[173] Jan Sommer et al. « Efficient Hardware Acceleration of Sparsely Active Convolutional Spiking Neural Networks ». *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41.11 (Nov. 2022), pp. 3767–3778. ISSN: 1937-4151. DOI: 10.1109/TCAD.2022.3197512. URL: https://ieeexplore.ieee.org/document/9925683 (visited on 05/05/2025).

[174] Joon Boum Song et al. « A Flexible Machine Vision AI System for Edge-Oriented Deep Learning Accelerators ». *2023 International Technical Conference on Circuits/Systems, Computers, and Communications (ITC-CSCC)*. 2023 International Technical Conference on Circuits/Systems, Computers, and Communications (ITC-CSCC). June 2023, pp. 1–4. DOI: 10.1109/ITC-CSCC58803.2023.10212811. URL: https://ieeexplore.ieee.org/document/10212811 (visited on 04/18/2024).

[175] Ilias Sourikopoulos et al. « A 4-fJ/Spike Artificial Neuron in 65 nm CMOS Technology ». *Frontiers in Neuroscience* 11 (Mar. 15, 2017). Publisher: Frontiers. ISSN: 1662-453X. DOI: 10.3389/fnins.2017.00123. URL: https://www.frontiersin.org/journals/neuroscience/articles/10.3389/fnins.2017.00123/full (visited on 05/12/2025).

[176] Dmitri B. Strukov et al. « The missing memristor found ». *Nature* 453.7191 (May 2008). Publisher: Nature Publishing Group, pp. 80–83. ISSN: 1476-4687. DOI: 10.1038/nature06932. URL: https://www.nature.com/articles/nature06932 (visited on 05/12/2025).

[177] Pengfei Sun et al. « Towards parameter-free attentional spiking neural networks ». *Neural Networks* 185 (May 1, 2025), p. 107154. ISSN: 0893-6080. DOI: 10.1016/j.neunet.2025.107154. URL: https://www.sciencedirect.com/science/article/pii/S0893608025000334 (visited on 08/12/2025).

[178] *Systèmes Embedded de NVIDIA pour les machines autonomes de nouvelle génération*. NVIDIA. URL: https://www.nvidia.com/fr-fr/autonomous-machines/embedded-systems/ (visited on 11/19/2025).

[179] Aboozar Taherkhani et al. « A review of learning in biologically plausible spiking neural networks ». *Neural Networks* 122 (Feb. 1, 2020), pp. 253–272. ISSN: 0893-6080. DOI: 10.1016/j.neunet.2019.09.036. URL: https://www.sciencedirect.com/science/article/pii/S0893608019303181 (visited on 04/23/2024).

[180] Chengcheng Tang and Jie Han. « Hardware Efficient Weight-Binarized Spiking Neural Networks ». *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2023 Design, Automation & Test in Europe Conference & Exhibition (DATE). ISSN: 1558-1101. Apr. 2023, pp. 1–6. DOI: 10.23919/DATE56975.2023.10136955. URL: https://ieeexplore.ieee.org/abstract/document/10136955 (visited on 04/29/2025).

[181] Uchechukwu Leo Udeji and Martin Margala. « FPGA Implementation of Addition-based CORDIC-SNN With Izhikevich Neurons ». *2022 IEEE 35th International System-on-Chip Conference (SOCC)*. 2022 IEEE 35th International System-on-Chip Conference (SOCC). ISSN: 2164-1706. Sept. 2022, pp. 1–6. DOI: 10.1109/SOCC56010.2022.9908081. URL: https://ieeexplore.ieee.org/abstract/document/9908081 (visited on 05/21/2025).

[182] Marzieh Hassanshahi Varposhti, Mahyar Shahsavari, and Marcel van Gerven. *Energy-Efficient Spiking Recurrent Neural Network for Gesture Recognition on Embedded GPUs*. Aug. 23, 2024. DOI: 10.48550/arXiv.2408.12978. arXiv: 2408.12978[cs]. URL: http://arxiv.org/abs/2408.12978 (visited on 04/17/2025).

[183] Sreyes Venkatesh, Razvan Marinescu, and Jason K. Eshraghian. *SQUAT: Stateful Quantization-Aware Training in Recurrent Spiking Neural Networks*. Apr. 15, 2024. DOI: 10.48550/arXiv.2404.19668. arXiv: 2404.19668[cs]. URL: http://arxiv.org/abs/2404.19668 (visited on 03/16/2025).

[184] Alex Vigneron and Jean Martinet. « A critical survey of STDP in Spiking Neural Networks for Pattern Recognition ». *2020 International Joint Conference on Neural Networks (IJCNN)*. 2020 International Joint Conference on Neural Networks (IJCNN). ISSN: 2161-4407. July 2020, pp. 1–9. DOI: 10.1109/IJCNN48605.2020.9207239. URL: https://ieeexplore.ieee.org/abstract/document/9207239 (visited on 05/07/2025).

[185] Huan Minh Vo. « Implementing the on-chip backpropagation learning algorithm on FPGA architecture ». *2017 International Conference on System Science and Engineering (ICSSE)*. 2017 International Conference on System Science and Engineering (ICSSE). ISSN: 2325-0925. July 2017, pp. 538–541. DOI: 10.1109/ICSSE.2017.8030932. URL: https://ieeexplore.ieee.org/abstract/document/8030932 (visited on 05/21/2025).

[186] Weier Wan et al. « A compute-in-memory chip based on resistive random-access memory ». *Nature* 608.7923 (Aug. 2022). Number: 7923 Publisher: Nature Publishing Group, pp. 504–512. ISSN: 1476-4687. DOI: 10.1038/s41586-022-04992-8. URL: https://www.nature.com/articles/s41586-022-04992-8 (visited on 05/23/2023).

[187] Kuanchuan Wang et al. « Large-scale Bio-inspired FPGA Models for Path Planning ». *IEEE Transactions on Biomedical Circuits and Systems* (2023). Conference Name: IEEE Transactions on Biomedical Circuits and Systems, pp. 1–12. ISSN: 1940-9990. DOI: 10.1109/TBCAS.2023.3302993.

[188] Runchun M. Wang, Chetan S. Thakur, and André van Schaik. « An FPGA-Based Massively Parallel Neuromorphic Cortex Simulator ». *Frontiers in Neuroscience* 12 (Apr. 10, 2018), p. 213. ISSN: 1662-4548. DOI: 10.3389/fnins.2018.00213. URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5902707/ (visited on 11/23/2022).

[189] Xiangwen Wang, Xianghong Lin, and Xiaochao Dang. « Supervised learning in spiking neural networks: A review of algorithms and evaluations ». *Neural Networks* 125 (May 1, 2020), pp. 258–280. ISSN: 0893-6080. DOI: 10.1016/j.neunet.2020.02.011. URL: https://www.sciencedirect.com/science/article/pii/S0893608020300563 (visited on 05/01/2025).

[190] William A. Watkins et al. « Sound Database of Marine Animal Vocalizations Structure and Operations. » (Aug. 1, 1992). Number: WHOI9231. URL: https://apps.dtic.mil/sti/html/tr/ADA256661/ (visited on 08/06/2025).

[191] *Watkins Marine Mammal Sound Database*. URL: https://whoicf2.whoi.edu/science/B/whalesounds/index.cfm (visited on 08/06/2025).

[192] Jiajun Wu et al. « Efficient Design of Spiking Neural Network With STDP Learning Based on Fast CORDIC ». *IEEE Transactions on Circuits and Systems I: Regular Papers* 68.6 (June 2021), pp. 2522–2534. ISSN: 1558-0806. DOI: 10.1109/TCSI.2021.3061766. URL: https://ieeexplore.ieee.org/document/9366935 (visited on 05/19/2025).

[193] Jian Wu, Steve Furber, and Jim Garside. « A Programmable Adaptive Router for a GALS Parallel System ». *2009 15th IEEE Symposium on Asynchronous Circuits and Systems*. 2009 15th IEEE Symposium on Asynchronous Circuits and Systems. ISSN: 1522-8681. May 2009, pp. 23–31. DOI: 10.1109/ASYNC.2009.17.

[194] Yujie Wu et al. « Spatio-Temporal Backpropagation for Training High-Performance Spiking Neural Networks ». *Frontiers in Neuroscience* 12 (May 23, 2018). Publisher: Frontiers. ISSN: 1662-453X. DOI: 10.3389/fnins.2018.00331. URL: https://www.frontiersin.org/journals/neuroscience/articles/10.3389/fnins.2018.00331/full (visited on 03/27/2025).

[195] Li Xiaoxue et al. « Review of medical data analysis based on spiking neural networks ». *Procedia Computer Science*. Tenth International Conference on Information Technology and Quantitative Management (ITQM 2023) 221 (Jan. 1, 2023), pp. 1527–1538. ISSN: 1877-0509. DOI: 10.1016/j.procs.2023.08.138. URL: https://www.sciencedirect.com/science/article/pii/S1877050923008967 (visited on 04/18/2024).

[196] Yannan Xing, Gaetano Di Caterina, and John Soraghan. « A New Spiking Convolutional Recurrent Neural Network (SCRNN) With Applications to Event-Based Hand Gesture Recognition ». 14 (Nov. 17, 2020). Publisher: Frontiers. ISSN: 1662-453X. DOI: 10.3389/fnins.2020.590164. URL: https://www.frontiersin.orghttps://www.frontiersin.org/journals/neuroscience/articles/10.3389/fnins.2020.590164/full (visited on 04/15/2025).

[197] Safa Yaghini Bonabi et al. « FPGA implementation of a biological neural network based on the Hodgkin-Huxley neuron model ». *Frontiers in Neuroscience* 8 (2014), p. 379. ISSN: 1662-453X. URL: https://www.frontiersin.org/article/10.3389/fnins.2014.00379 (visited on 04/18/2022).

[198] Yexin Yan et al. « Comparing Loihi with a SpiNNaker 2 prototype on low-latency keyword spotting and adaptive robotic control ». *Neuromorphic Computing and Engineering* 1.1 (July 2021). Publisher: IOP Publishing, p. 014002. ISSN: 2634-4386. DOI: 10.1088/2634-4386/abf150. URL: https://dx.doi.org/10.1088/2634-4386/abf150 (visited on 06/06/2025).

[199] Shiyu Yang et al. « An Efficient FPGA Implementation of Izhikevich Neuron Model ». *2020 International SoC Design Conference (ISOCC)*. 2020 International SoC Design Conference (ISOCC). ISSN: 2163-9612. Oct. 2020, pp. 141–142. DOI: 10.1109/ISOCC50952.2020.9333014.

[200] Shuangming Yang et al. « Scalable Digital Neuromorphic Architecture for Large-Scale Biophysically Meaningful Neural Network With Multi-Compartment Neurons ». *IEEE Transactions on Neural Networks and Learning Systems* 31.1 (Jan. 2020). Conference Name: IEEE Transactions on Neural Networks and Learning Systems, pp. 148–162. ISSN: 2162-2388. DOI: 10.1109/TNNLS.2019.2899936. URL: https://ieeexplore.ieee.org/abstract/document/8668700 (visited on 08/19/2024).

[201] Shuangming Yang et al. « BiCoSS: Toward Large-Scale Cognition Brain With Multigranular Neuromorphic Architecture ». *IEEE Transactions on Neural Networks and Learning Systems* 33.7 (July 2022). Conference Name: IEEE Transactions on Neural Networks and Learning Systems, pp. 2801–2815. ISSN: 2162-2388. DOI: 10.1109/TNNLS.2020.3045492. URL: https://ieeexplore.ieee.org/document/9319553/?arnumber=9319553 (visited on 08/19/2024).

[202] Shuangming Yang et al. « Smart Traffic Navigation System for Fault-Tolerant Edge Computing of Internet of Vehicle in Intelligent Transportation Gateway ». *IEEE Transactions on Intelligent Transportation Systems* 24.11 (Nov. 2023). Conference Name: IEEE Transactions on Intelligent Transportation Systems, pp. 13011–13022. ISSN: 1558-0016. DOI: 10.1109/TITS.2022.3232231. URL: https://ieeexplore.ieee.org/document/10014015 (visited on 08/19/2024).

[203] Shuangming Yang et al. « Integrating Visual Perception With Decision Making in Neuromorphic Fault-Tolerant Quadruplet-Spike Learning Framework ». *IEEE Transactions on Systems, Man, and Cybernetics: Systems* 54.3 (Mar. 2024). Conference Name: IEEE Transactions on Systems, Man, and Cybernetics: Systems, pp. 1502–1514. ISSN: 2168-2232. DOI: 10.1109/TSMC.2023.3327142. URL: https://ieeexplore.ieee.org/document/10318216 (visited on 08/19/2024).

[204] Shuangming Yang et al. « NADOL: Neuromorphic Architecture for Spike-Driven Online Learning by Dendrites ». *IEEE Transactions on Biomedical Circuits and Systems* 18.1 (Feb. 2024). Conference Name: IEEE Transactions on Biomedical Circuits and Systems, pp. 186–199. ISSN: 1940-9990. DOI: 10.1109/TBCAS.2023.3316968. URL: https://ieeexplore.ieee.org/document/10255238/?arnumber=10255238 (visited on 08/19/2024).

[205] Yang Yi et al. « FPGA based spike-time dependent encoder and reservoir design in neuromorphic computing processors ». *Microprocessors and Microsystems* 46 (Oct. 1, 2016), pp. 175–183. ISSN: 0141-9331. DOI: 10.1016/j.micpro.2016.03.009. URL: https://www.sciencedirect.com/science/article/pii/S0141933116300060 (visited on 05/05/2025).

[206] Hongzhi You and Kunpeng Zhao. « Neuromorphic Implementation of a Continuous Attractor Neural Network With Various Synaptic Dynamics ». *IEEE Access* 9 (2021), pp. 109224–109240. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2021.3101290. URL: https://ieeexplore.ieee.org/abstract/document/9502127 (visited on 05/19/2025).

[207] Aaron R. Young et al. « A Review of Spiking Neuromorphic Hardware Communication Systems ». *IEEE Access* 7 (2019). Conference Name: IEEE Access, pp. 135606–135620. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2019.2941772.

[208] Amirreza Yousefzadeh et al. « Hybrid Neural Network, An Efficient Low-Power Digital Hardware Implementation of Event-based Artificial Neural Network ». *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2018 IEEE International Symposium on Circuits and Systems (ISCAS). ISSN: 2379-447X. May 2018, pp. 1–5. DOI: 10.1109/ISCAS.2018.8351562. URL: https://ieeexplore.ieee.org/abstract/document/8351562 (visited on 05/20/2025).

[209] Abdulhamid Zahedi, Saeed Haghiri, and Mohsen Hayati. « Multiplierless Digital Implementation of Time-Varying FitzHugh–Nagumo Model ». *IEEE Transactions on Circuits and Systems I: Regular Papers* 66.7 (July 2019), pp. 2662–2670. ISSN: 1558-0806. DOI: 10.1109/TCSI.2019.2899361. URL: https://ieeexplore.ieee.org/document/8660414 (visited on 05/16/2025).

[210] Friedemann Zenke and Surya Ganguli. « SuperSpike: Supervised Learning in Multilayer Spiking Neural Networks ». *Neural Computation* 30.6 (June 1, 2018), pp. 1514–1541. ISSN: 0899-7667. DOI: 10.1162/neco_a_01086. URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC6118408/ (visited on 05/09/2025).

[211] Jilin Zhang et al. « An Asynchronous Reconfigurable SNN Accelerator With Event-Driven Time Step Update ». *2019 IEEE Asian Solid-State Circuits Conference (A-SSCC)*. 2019 IEEE Asian Solid-State Circuits Conference (A-SSCC). Nov. 2019, pp. 213–216. DOI: 10.1109/A-SSCC47793.2019.9056903. URL: https://ieeexplore.ieee.org/document/9056903 (visited on 09/04/2024).

[212] Wenrui Zhang and Peng Li. « Temporal Spike Sequence Learning via Backpropagation for Deep Spiking Neural Networks ». *Advances in Neural Information Processing Systems*. Vol. 33. Curran Associates, Inc., 2020, pp. 12022–12033. URL: https://proceedings.neurips.cc/paper/2020/hash/8bdb5058376143fa358981954e7626b8-Abstract.html (visited on 05/08/2024).

[213] Zhaoyun Zhang and Jingpeng Li. « A Review of Artificial Intelligence in Embedded Systems ». *Micromachines* 14.5 (May 2023). Publisher: Multidisciplinary Digital Publishing Institute, p. 897. ISSN: 2072-666X. DOI: 10.3390/mi14050897. URL: https://www.mdpi.com/2072-666X/14/5/897 (visited on 08/12/2025).

[214] Dongcheng Zhao et al. « GLSNN: A Multi-Layer Spiking Neural Network Based on Global Feedback Alignment and Local STDP Plasticity ». *Frontiers in Computational Neuroscience* 14 (Nov. 12, 2020). Publisher: Frontiers. ISSN: 1662-5188. DOI: 10.3389/fncom.2020.576841. URL: https://www.frontiersin.orghttps://www.frontiersin.org/journals/computational-neuroscience/articles/10.3389/fncom.2020.576841/full (visited on 04/14/2025).

[215] Huanliang Zheng et al. « Balancing the Cost and Performance Trade-Offs in SNN Processors ». *IEEE Transactions on Circuits and Systems II: Express Briefs* 68.9 (Sept. 2021), pp. 3172–3176. ISSN: 1558-3791. DOI: 10.1109/TCSII.2021.3090422. URL: https://ieeexplore.ieee.org/abstract/document/9459780 (visited on 05/21/2025).

FPGA-Based Neuromorphic Architecture for Spiking Neural Network Emulation by *Aurélie Saulquin*

Abstract

Artificial Intelligence (AI), and particularly Artificial Neural Networks (ANNs) have become central to modern computing with remarkable performances for complex task resolution. Their integration on embedded systems establishes the Artificial Intelligence of Things (AIoT).
Therefore, ANN integration on such systems raises major challenges due to model complexity. While some researchers propose to offload processing to the cloud, others propose new hardware architectures or compression algorithms for in-edge computing. Another alternative approach, neuromorphic computing, consists of drawing inspiration from the brain by using Spiking Neural Networks (SNNs) to treat spiking information.
Although promising, neuromorphic chips are limited and poorly reconfigurable, making FPGAs an interesting target for research. In this context, we developed ModNEF, an open-source modular FPGA architecture for SNN inference through the interconnection of independent modules, offering high implementation flexibility. We validated our architecture on standard neuromorphic datasets and on a use case for sperm whale detection in the Mediterranean Sea, demonstrating its capacity to perform classification tasks for embedded applications.
Keywords *: Neuromorphic accelerator, Spiking Neural Networks, Artificial Intelligence, Embedded Artificial Intelligence, Embedded Systems, Edge Computing, Parallel Architecture, Digital Architecture, Neuromorphic Architecture, FPGA*

Résumé

L'intelligence artificielle, et plus particulièrement les Réseaux de Neurones Artificiels (ANNs) ont pris une place centrale dans l'informatique moderne avec des performances remarquables pour la résolution de tâches complexes. Leur intégration dans des systèmes embarqués a donné naissance à l'Intelligence Artificielle des Objets (AIoT).
Cependant, l'intégration d'ANN dans ces systèmes présente des défis majeurs liés à leur complexité. Une solution est de traiter les données dans le cloud, mais de nouvelles architectures matérielles ou des méthodes de compression permettent de garder le calcul en périphérie. Une approche alternative, le calcul neuromorphique, consiste à s'inspirer du cerveau en utilisant des réseaux de neurones impulsionnels (SNNs) pour traiter l'information sous forme d'impulsion.
Bien que prometteuses, les puces neuromorphiques sont souvent limitées et peu reconfigurables, plaçant les FPGA comme des plateformes privilégiées pour la recherche. C'est dans ce contexte que nous avons développé ModNEF, une architecture modulaire libre sur FPGA permettant l'inférence de SNN complexes via l'interconnexion de modules indépendants, offrant ainsi une grande flexibilité d'implémentation. Nous avons validé notre architecture sur des jeux de données neuromorphiques standards ainsi que sur une étude de cas sur la détection de cachalots en mer Méditerranée, démontrant sa capacité à réaliser des tâches de classification pour des applications embarquées.
Mots Clés *: Accélérateur Neuromorphique, Réseaux de Neurones Impulsionnels, Intelligence Artificielle, Intelligence Artificielle Embarquée, Systèmes Embarqués, Calcule en Prériphérie, Architecture Parallèle, Architecture Numérique, Architecture Neuromorphique, FPGA*