

HDL Implementation of SNN for Event-based Dataset

B. Tech. Project Report

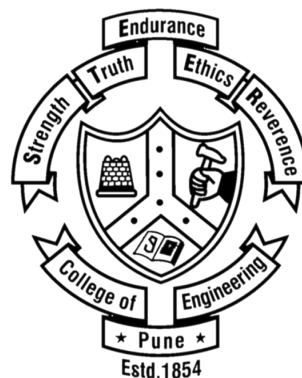
Submitted by

Rushikesh Gherde	111807014
Tarun Kumar Allamsetty	111807029
Juhi Wani	111807044

Under the guidance of

Dr. Vanita Agarwal

College of Engineering, Pune



DEPARTMENT OF ELECTRONICS AND
TELECOMMUNICATION,
COLLEGE OF ENGINEERING, PUNE

April-May 2022

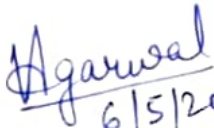
**DEPARTMENT OF ELECTRONICS AND
TELECOMMUNICATION,
COLLEGE OF ENGINEERING, PUNE**

CERTIFICATE

Certified that this project, titled "HDL Implementation of SNN for Event-based Dataset" has been successfully completed by

Rushikesh Gherde	111807014
Tarun Kumar Alamsetty	111807029
Juhi Wani	111807044

and is approved for the partial fulfillment of the requirements for the degree of "B.Tech. Electronics and Telecommunication Engineering".


6/5/2022

SIGNATURE

Project Guide

Department of Electronics and
Telecommunication

College of Engineering Pune,
Shivajinagar, Pune - 5.



SIGNATURE

Head of Department

Department of Electronics and
Telecommunication

College of Engineering Pune,
Shivajinagar, Pune - 5.

Approval Sheet

This report entitled

HDL Implementation of SNN for Event-based Dataset

By

Rushikesh Gherde 111807014

Tarun Kumar Alamsetty 111807029

Juhi Wani 111807044

Is approved for the degree of

Bachelor of Technology

of

Department of Electronics and Telecommunication

College of Engineering Pune

(An Autonomous institute of Govt. of Maharashtra)

Examiners

Name

Signature

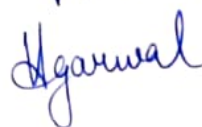
External Examiner

(Manoj Soman)






Guide

Dr. Vanita Agarwal



DECLARATION

We declare that this written submission represents our ideas in our own words and where others' ideas or words have been included, we have adequately cited and referenced the original sources. We also declare that we have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/ data/ fact source in our submission. We understand that any violation of the above will be cause for disciplinary action by the Institute and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.

NAME OF STUDENTS	MIS NO.	SIGNATURE
Rushikesh Gherde	111807014	
Tarun Kumar Allamsetty	111807029	
Juhi Wani	111807044	

Date: 6/5/2022

Place: Pune

Acknowledgements

We are grateful to our college and the Department of Electronics and Telecommunication for providing us with this opportunity to work on this project. It is their visionary objective to encourage students for this thesis-based project that has blossomed into this extraordinary opportunity.

To begin with, we sincerely thank our project guide Dr. Vanita Agarwal for her valuable suggestions and guidance towards the design and development of the project. Her constant motivation and support proved to be crucial.

Furthermore we would like to express our sincere gratitude to Dr. Mr. P. P. Bartakke, Dr. M. S. Sutaone and Mr. S. G. Mali for granting to us the access to the Centre of Excellence, College of Engineering Pune and the DGX Station GPU which were quintessential for the completion of the project.

The team is immensely grateful to Dr. S. P. Mahajan, Head of the Electronics and Telecommunications Department. It is his support and encouragement to students to work on a project of their field of interest that has made this possible.

Abstract

Neural networks have for long been the most popular tools used to perform complex brain-like operations like classification tasks, pattern recognition and path finding to name a few. Variations of neural networks include Artificial Neural Networks (ANN), Convolutional Neural Networks (CNN) and Recurrent Neural Networks (RNN). Although several advanced architectures and training algorithms have been consistently improving the performance of these neural networks, they are getting increasingly computationally expensive, high power consuming and require a extensive hardware.

Neuromorphic computing aims to lessen these drawbacks by operating on "Spiking Neural Networks" (SNN). SNNs have a *spiking neuron* as their primary unit which is a model of the biological neuron. SNNs aim to mimic the operation of the human brain in terms of encoding the data in the form of spike trains and propagating spikes along the network. These networks have the potential of working with complex time-dependent recognition problems apart from the basic recognition and classification problems- while consuming much less power and involving much less computation when compared to traditional neural networks. These promising prospectives have sparked the developed of specialised hardware in the form of "Neuromorphic Chips" that perform computations by closely emulating the brain's functioning.

This project discusses the design and training of a Spiking Neural Network that operates on an event-based dataset and the development of the synthesizable network in SystemVerilog for deployment on a FPGA device. The methodology for the design and development of the networks has been discussed. The detailed architecture has been elucidated along with discussions on the experiments undertaken and the results obtained.

Contents

List of Tables	ii
List of Figures	iv
List of Abbrevations	v
List of Symbols	vii
1 Introduction	1
1.1 Background	2
1.2 Aim	3
1.3 Motivation	3
1.4 Methodology	5
2 Literature Review	7
2.1 Spiking Neural Networks	7
2.1.1 The LIF Neuron Model	7
2.1.2 Learning SNN	9
2.2 Hardware Implementation of SNN	11
3 Experimental Work	13

3.1	Libraries Explored	13
3.2	Datasets Explored	14
3.3	Learning Methods Explored	14
3.4	HDL implementation of SNN	16
4	Implementation and Results	18
4.1	Software Implementation	19
4.1.1	Libraries used	19
4.1.2	Loading data	19
4.1.3	Architecture	20
4.1.4	Training	21
4.2	HDL Implementation	23
4.2.1	Passing Values to the Top Module	25
4.2.2	Fixed Point Representation of Numbers	25
4.2.3	Convolution module	26
4.2.4	LIF module	27
4.2.5	Flattening and Counter Modules	30
4.2.6	Method of Relaying Flags	30
4.3	Results and Discussion	31
4.3.1	Software	31
4.3.2	Hardware	37
4.3.3	Comparing Classical CNN with SNN	46
5	Conclusion and Future Scope	50
5.0.1	Conclusion	50
5.0.2	Future Scope	51

List of Tables

3.1	Details of libraries explored	14
4.1	CSNN architectures	20
4.2	Hyperparameters Priority	33
4.3	Accuracy	36

List of Figures

1.1	(a) Membrane of neuron (b) LIF circuit model	3
2.1	(a) Block diagram of LIF (b)Unrolled Recurrence block diagram of LIF	9
2.2	S vs U, $\frac{\partial S}{\partial U}$ vs U	10
2.3	Surrogate gradients	10
3.1	Timing diagram of Poisson Encoder	17
4.1	Forward pass of network for one timestep for above two architectures	21
4.2	Learning rate schedulers	22
4.3	HDL Block Diagram	24
4.4	Fixed point representation	25
4.5	Multichannel Convolution	26
4.6	State Diagram	29
4.7	LIF Block Diagram	29
4.8	Training Curves	31
4.9	Validation Curves	32
4.10	Tuning all hyperparameters	33
4.11	Intermediate values of val accuracy while hyper parameter tuning	34
4.12	Parallel coordinate values of hyperparameters	34
4.13	Tuning slope of surrogate gradient descent	35

4.14	Intermediate values for slope tuning	35
4.15	Full traning with hyperparamters	36
4.16	Neuron timing diagram	37
4.17	Synthesis models of neuron model	38
4.18	Implementation Results	39
4.19	Power Utilization Report	39
4.20	Convolution output	40
4.21	Convolution module synthesis results	41
4.22	Convolution module implementation results	41
4.23	Convolution module power utilization report	42
4.24	Synthesis models of convolution module's <i>flip_filter</i> task	43
4.25	Calculator Task	44
4.26	Simulation results	45
4.27	Convolution for SNN	47
4.28	Convolution for CNN	49

List of Abbreviations

ANN	Artificial Neural Network
API	Application Program Interface
CNN	Convolution nerual network
CPU	Central Processing Unit
DSP	Digital Signal Processing
DVS	Dynamic Vision Sensor
FPGA	Field Programmable Gate Array
GPU	Graphical Processing Unit
HDL	Hardware Descriptive Language
IC	Integrated Circuit
IF	Integrate and Fire
LIF	Leaky Integrate and Fire
MNIST	Modified National Institute of Standards and Technology
NN	Neural Networks
RAM	Random Access Memory
SCNN	Spiking Convolution network

SNN Spiking Neural Network

STDP Spike Time Dependent Plasticity

List of Symbols

1. β Decay paramter
2. τ Time constant of RC-Circuit
3. $U(t)$ Membrane Voltage
4. U_{thresh} Threshold Voltage
5. R Resistor
6. C Capacitor
7. I_{in} Input current of LIF
8. S_{out} Spike output of LIF
9. θ Threshold
10. Θ Unit step function
11. δ Impulse function
12. W Weight Matrix

Chapter 1

Introduction

As the limit to Moore's Law approaches, scientists and researchers are delving into new, disruptive methods hardware architectures. At the same time, software algorithms are being developed to widen the applications and improve the accuracy of neural networks so as to serve real life requirements. These complex algorithms are therefore computationally expensive and require robust and extensive hardware architecture to be compatible with them in terms of available resources and power requirements. Nonetheless, all of these developments fail to emulate the capabilities of a human brain- often dubbed as the most complex machine. As science gains more insights into the functioning of the mammalian brain, its emulation in terms of hardware and software via neuromorphic computing and spiking neural networks has sparked research as a potential solution to the limitations faced by traditional computing techniques.

This chapter provides some insight into the field of neuromorphic computing including past work and highlights the motivation for undertaking this project.

1.1 Background

Neuromorphic computing deals with modelling the elements in a system in close accordance with units present in the mammalian brain and nervous system. It functions over a network of spiking neurons. The first spiking neural network was developed in 1995 by J. Hopfield heralding the third generation of ANNs. Neural networks have evolved from the first generation "perceptrons" to the second generation feedforward networks, CNNs and RNNs to the biologically more tenable SNNs. The lucrative features of SNN have garnered the attention of major companies resulting in the development of neuromorphic hardware like the Intel TrueNorth, Loihi and SpiNNaker to name a few.

SNN neurons are based on mathematical representations of biological neurons. To model an SNN neuron, there are two major sets of approaches

1. **Conductance based models** describe how action potential of neuron are initiated and propagated. E.g. Hodgkin-Huxley model, Izhikevich model
2. **Threshold based models** generate an impulse at a certain threshold. E.g Leaky Integrate and Fire (LIF). LIF is the most popular model used for SNN neuron.

Biological neurons are surrounded by a thin membrane. This membrane is a lipid bilayer that separates the neuron's conducting saline solution from the extracellular media. LIF models it as a capacitor, as it's two conductive solutions separated by an insulator. The membrane also controls the inflow and outflow of ions. The membrane is impermeable to ions, but specific channels are opened when membrane is triggered by injecting current into them. LIF models it as a resistor. When the potential of the membrane

exceed a threshold then the neuron generates a spike. LIF is discussed further in Section 2.1.1

1.2 Aim

1. Training of an event based dataset on a Convolutional Spiking Neural Network and analysis of performance with change in hyper-parameters.
2. HDL implementation of the SNN and analysis of power consumption and resource utilization.

1.3 Motivation

The study of Neuromorphic computing is a blossoming research field in terms of designing and deploying architectures and training methods over specialised datasets as well their hardware implementation which makes the best use of the features of SNNs.

Several reasons contribute to the rising popularity of SNNs. According to the Moore's Law, the number of transistors on a chip doubles every two year. While this law held up for more than three decades, it is now reach-

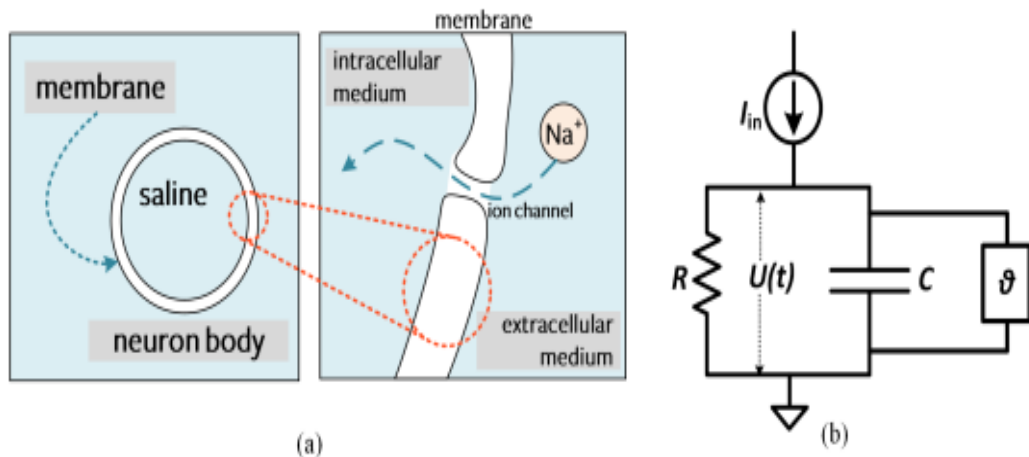


Figure 1.1: (a) Membrane of neuron (b) LIF circuit model

ing its limits. Today, the transistor gate width is approaching around 2nm which is merely around 10 times the size of a Silicon atom; thus putting a constraint on the transistor size. Thus scientists are exploring non-Von neumann architectures like the neuromorphic computing paradigm. Neuro-morphic computing provides a solution to the Von-neumann bottleneck that limits the data throughput due to the usage of a bus to carry data between the processor, memory and peripherals by using distributed architecture style composed of small units of memory and computation called 'neurons'.

Artificial Neural Networks (ANN) encode information in floating point values, whereas Spiking Neural Networks (SNN) encode information in terms of spikes. In an ANN the output of a fundamental unit only depends on the current input vector, but in SNN the output depends not only on the current spike but past spikes as well. Hence, the information transmission is distributed over time.

Neural networks undergo sparsification over time. Neurons send spikes less frequently and its output is zero for sometime resulting in less consumption of power. Since spikes are binary, applying weights is simply a memory lookup and add operation; unlike floating point matrix multiplications in traditional NNs.

In this project, we have worked on a "convolutional spiking neural network" that performs a classification function on an event-based dataset. Back-propagation using spikes has been used to train spiking neural network. Its performance in terms of accuracy, log loss and F1 score has been analysed. Further, the determined architecture has been prepared for FPGA synthesis using SystemVerilog and the resource utilization for the target device has been analysed.

1.4 Methodology

To implement a convolutional spiking neural network in software and HDL, the project has been divided into 2 parts. The first part deals with design of the architecture and training the event based dataset on the architecture finalised. It also involves analysing the change in performance of the network for different hyperparameters like network depth and width, neuron parameters, etc. The second part of the project deals with the HDL implementation of the network having good performance and its synthesis is feasible on the target FPGA device.

The following steps have been followed to achieve the aim:

1. Selection of a dataset based on potential applications.
2. Determination of neuron model, network type and training technique from those explored in the Literature Survey.
3. Experimenting with various architectures, neuron models, libraries and datasets to get an overview of the efficacy of the parameters, training time required and results obtained.
4. Finalisation of an architecture and implementation using a python library from among those experimented with.
5. Training the SNN using GPU available at college facility and analysing the performance achieved, followed by assessment of performance on changing the hyperparameters like network depth, layer width, batch size, LIF neuron parameters like β , threshold, reset mechanism.
6. Modelling the several layers in HDL like Verilog or SystemVerilog in the

form of synthesizable modules. The weights obtained from training the SNN are to be hard coded in the HDL code.

7. Synthesis and implementation of the HDL code and analysis of resource and power utilisation.

The subsequent chapters delineate the work done in detail. Chapter 2 covers literature review. Details about the LIF neuron model, learning methods used in SNN are discussed. Chapter 3 covers the experimental work which discusses libraries, datasets explored and why SystemVerilog was used for hardware implementation is discussed. Chapter 4 discusses the training of convolution spiking neural networks and synthesizing the network on hardware. The work is concluded in Chapter 5.

Chapter 2

Literature Review

2.1 Spiking Neural Networks

Spiking neural networks, the third generation of neural networks, attempt to bridge the gap between neuroscience and machine learning by performing computation using biologically realistic models of neurons.

The Literature Review undertaken for this project work revolves around three major heads: the neuron model, learning methods for SNN and the hardware implementation of SNNs.

2.1.1 The LIF Neuron Model

The Leaky Integrate and Fire (LIF) neuron model is a widely used neuron model in SNN applications. The LIF model remembers its previous inputs by adding each one to its membrane potential and then forgetting them using a leak (decay) parameter. This leak parameter represents the ion diffusion that occurs through the membrane when the cell's equilibrium is not reached. As discussed in Section 1.2, LIF can be modelled as a circuit. The differential equation RC circuit in Figure 1.1 is

$$\tau \frac{dU(t)}{dt} = -U(t) + I_{in}(t)R \quad (2.1)$$

where $\tau = RC$ is time constant. The solution of 2.1 for constant current is

$$U(t) = I_{in}(t)R + [U_0 - I_{in}(t)R]e^{-\frac{t}{\tau}} \quad (2.2)$$

On applying the forward Euler method 2.2 reduces to

$$U[t] = \beta U[t - 1] + (1 - \beta)I_{in}[t] \quad (2.3)$$

The weighting factor of an input is usually a learnable parameter in deep learning. In order to dissociate the effect of on the input $X[t]$, the coefficient of input current in Equation 2.3 $(1 - \beta)$ is subsumed into a learnable weight W , and the simplification $I_{in}[t] = WX[t]$ is produced. If LIF has multiple inputs the, W is a matrix and $X[t]$ becomes a vector

Thus membrane potential of LIF is given as

$$U[t] = \beta U[t - 1] + WX[t] - S_{out}[t - 1]\theta \quad (2.4)$$

When the membrane voltage exceed the threshold , then neuron generates a spike. It is represented as

$$S_{out}[t] = \begin{cases} 1, & \text{if } U[t] > \theta \\ 0, & \text{otherwise} \end{cases} \quad (2.5)$$

When a biological neuron spikes, then its membrane potential drops. This reset mechanism is captured in Equation 4.3. When spike is generated $S_{out}[t - 1] = 1$ so θ is subtracted , else $S_{out}[t - 1] = 0$ and no reset happens. Equations

from 2.1 to 2.5 are taken from [1]

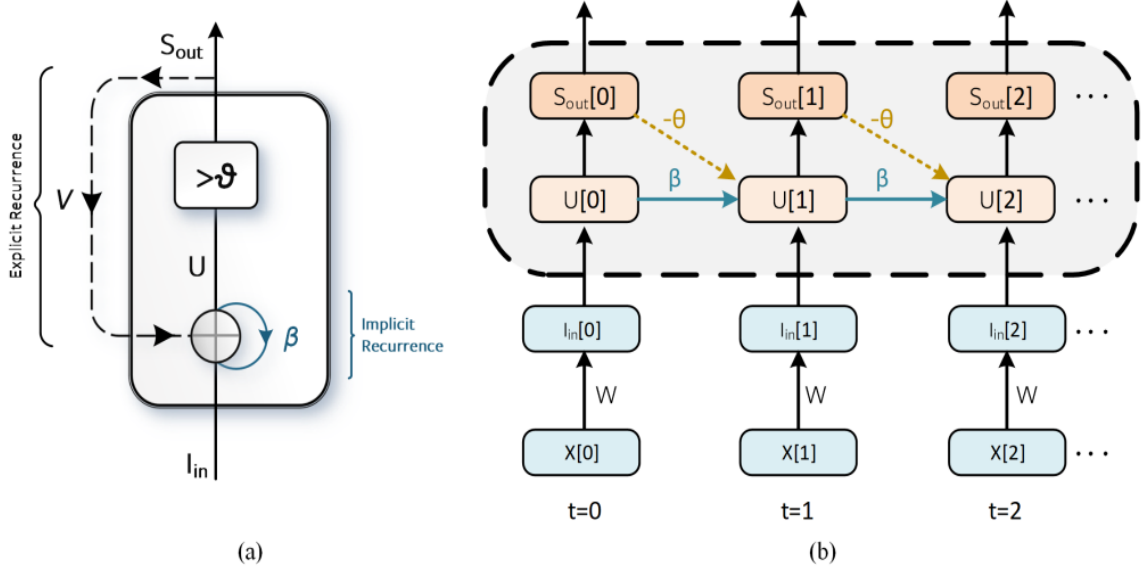


Figure 2.1: (a) Block diagram of LIF (b) Unrolled Recurrence block diagram of LIF

2.1.2 Learning SNN

Spiking neural networks training methods can be classified into three categories

1. **Shadow Training** is converting already existing ANN to SNN.
2. **Backpropagation Using Spikes** is training using error backpropagation rule.
3. **Local learning rules** Weight updates are a function of signals that are spatially and temporally local to the weight. E.g. STDP

In this project back-propagation using spikes is used as a training method. Alternate representation of Equation 2.5 is

$$S[t] = \Theta(U[t] - U_{thresh}) \quad (2.6)$$

where Θ is heavy side step function. On taking derivative of Equation 2.6

$$\frac{\partial S[t]}{\partial U} = \delta(U[t] - U_{thresh}) \quad (2.7)$$

where δ is the impulse function.

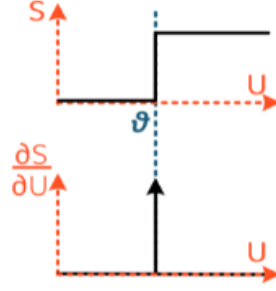


Figure 2.2: S vs U , $\frac{\partial S}{\partial U}$ vs U

The backpropagation algorithm's purpose is to minimise loss. To do so, the gradient of the loss is determined with respect to each learnable parameter using the chain rule from the final layer back to each weight. The gradient is then utilised to update the weights so that the error is always reduced ideally. As seen in Fig 2.2 the gradient is zero most of the time, so no weight update take place. This is known as "dead neuron" problem. This problem is addressed by using Surrogate gradients [2]. It proposes to replace the backward pass impulse function with functions like sigmoid, triangular etc to get non-zero gradients.

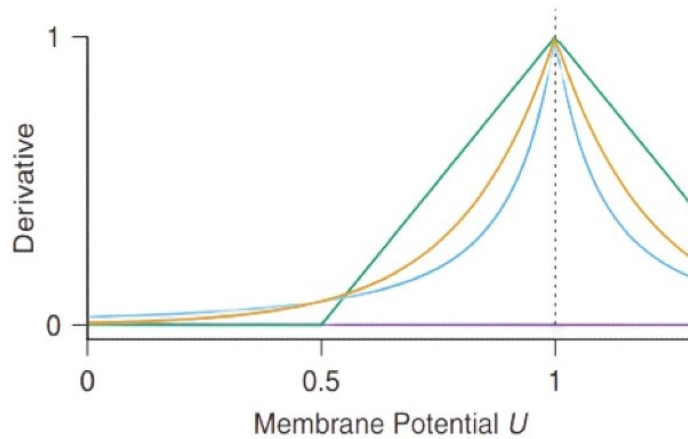


Figure 2.3: Surrogate gradients

2.2 Hardware Implementation of SNN

Spiking Neural Networks have been implemented in hardware on customised ICs as well as FPGAs. Throughout literature, several neuron models have been implemented, the most popular ones being the IF and the LIF models. Spike time dependent plasticity (STDP) is one of the common learning methods used. In order to implement a neuron on digital hardware, the neuron model equations have to be discretized and converted into the Euler form.

As a starting point, a top end neuromorphic chip was studied that gave an idea of the blocks involved in the architecture, the training techniques used and interconnection method. In [6] the architecture of Intel’s TrueNorth and some of its applications have been highlighted. It describes the simplified equation for the LIF neuron used. The gate level model for a single neuron has also been described in the publication.

A simplified SNN composed of 800 neurons has been implemented with real time learning in [5]. LIF neuron model and STDP learning rate has been used. The network spikes only in case of an event thereby simplifying the complexity. The STDP weights have been computed using a lookup table. This paper provides a good reference for modelling a neuron. It highlights the modules needed to construct a neuron and methods of generating spikes and reading weights.

In order to compare the complexity and efficacy of other neuron models, [7] has been referred. The FPGA implementation demonstrated in this paper is based on the Hodgkin-Huxley model. It provides insight into the basic design of a network consisting of a small number of sub-units, referred to as ‘mini-columns’ in the publication. The Euler form of the Hodgkin-Huxley model has been used and implemented using MUXs, adder, multiplier and

divider blocks.

In [10], an extensive neural core has been developed. A hierarchical modelling of the network and time multiplexing of the neural 'minicolumns' has been performed which proves to be effective in the number of neurons actually required to be synthesized. The publication highlights modelling over a million neurons which form the 'neural core' based on the LIF model. It employs the fact that only a small percentage of neurons are active concurrently and they can be reused for other operations at other times.

Other techniques for hardware implementation of SNNs have been explored by referring [8], [9] and [10].

Chapter 3

Experimental Work

This chapter deals with the extensive experimental work undertaken before closing in on the datasets, learning techniques, HDL techniques to be used in the project.

The experiments revolve around four major paths including the software libraries explored for training, datasets overviewd to best suit an SNN, several learning methods deployed and finally the techniques used in the hardware synthesis of the network.

3.1 Libraries Explored

As SNN is a developing field, the libraries are not mainstream enough as traditional deep learning libraries like pytorch, tensorflow etc. Some of the libraries experimented with are listed in Table 3.1.

Snntorch is used in this project due to its good documentation and user friendly API. As it is built on pytorch it has GPU support built in. It also goes hand in hand with pytorch-lightning , a pytorch research framework.

Libraries	Gpu support	Learning methods
Nengo	Yes	Hebbian, PES, RLS
Brian2	No	STDP
Bindsnet	True	Hebbian, STDP, MSTDP
Snntorch	True	Surrogate gradient descent

Table 3.1: Details of libraries explored

3.2 Datasets Explored

1. **MNIST (Modified National Institute of Standards and Technology database)** [11] is a large collection of handwritten digits. It has a training set of 60,000 examples, and a test set of 10,000 examples. Image size is 28*28 pixels and has 10 labels. As the dataset is static in nature and SNN expects inputs in form of spikes, the static images are converted to spikes using Poission Encoding.
2. **DVS 128** [12] is an event based dataset. The dataset was created by employing a TrueNorth neurosynaptic processor to recognise hand motions in real-time at minimal power from events transmitted live by a Dynamic Vision Sensor (DVS). Unlike typical frame-based cameras, which sample every pixel at a predetermined frame rate, the biologically inspired DVS transmits data only when a pixel detects a change. DVS dataset (DvsGesture) comprising 11 hand gesture categories from 29 subjects under 3 illumination conditions.

3.3 Learning Methods Explored

Two learning methods : STDP and surrogate gradient were explored on the MNIST dataset to get an idea of the architectures used, software support in

terms of libraries required, hardware support in terms of CPU RAM, GPU required and the time needed for training.

1. In an experiment inspired from [4] the network was trained in an unsupervised fashion using STDP. The network is divided into two tiers. The first layer is made up of excitatory neurons. The second layer is made up of excitatory and inhibitory neurons. Excitatory neurons are connected one to one to inhibitory neurons, while inhibitory neurons are connected back to all excitatory neurons except the one from whence they received connection. Once training is done, the network is ran through the entire dataset, and output neuron is associated with that label for which it fires the most. The predicted digit is determined by averaging the responses of each neuron per label and then choosing the label with the highest average firing rate. This model achieved 95% accuracy.
2. Convolution spiking neural network was trained using surrogate gradient, using fast sigmoid. The network architecture used is 12C5-MP2-64C5-MP2-1024FC10.
 - 12C5 : 12 convolution filters with 5x5 kernel
 - MP2: Max pool layer with kernel 2
 - 1024FC10 : Fully connected layer with 1024 inner features and 10 out features

The loss function used is cross entropy rate loss i.e. cross entropy loss is applied at every time step. Adam optimizer with learning rate $1e-2$ and momentum = (0.9 , 0.999). The accuracy achieved is 97.5%.

3.4 HDL implementation of SNN

The HDL implementation of the selected SNN architecture involves selection of a suitable synthesizable language and determination of the number of modules required, type of storage elements to be used and clock period required. In the experimentation phase, codes were initially written in Verilog. Due to several constraints attached to the language, it was decided to use SystemVerilog as the hardware descriptive language instead.

A simple LIF neuron was first modelled in Verilog. Verilog does not allow multi-dimensional ports to be declared, does not allow array slicing in case of multidimensional arrays and has limited provisions of modularity. These put severe constraints on the code since in case of a convolutional neural network, the inputs are all multichannel matrices. Apart from countering with these shortcomings, SystemVerilog also gives a provision for tasks and functions adding modularity and reusability to the code.

Operations of file read and write were performed in order to determine the bit width and format of the numerical values involved. Floating point representation of the numerical values considered was initially considered to maintain consistency with the default 64 or 32 bit floating point representation used in Python libraries. Using floating point representation improves the accuracy but is computationally expensive and complex. A simple addition operation requires use of clock based shift register in case of floating point computations. Owing to these limitations related to computations of floats, alternatives were considered. After taking into consideration the required precision for correct computation, a data size of 35 bits was considered to store the numbers in a fixed point format. Due to the binary nature of spikes, fixed point representation of decimals suffices to perform simple cal-

culations.

A trial on using static images as input instead of event based input was also performed. This required explicitly encoding the input in terms of spike train by either using rate or temporal coding. A Poisson's Spike Train generator circuit was designed in order to carry out rate based encoding of the input pixels. The output timing diagram for the same can be seen in Figure 3.1. The variable *spike* is the output spike train which goes to 1 when the input *pixel* is greater than random number generated shown in *i*.

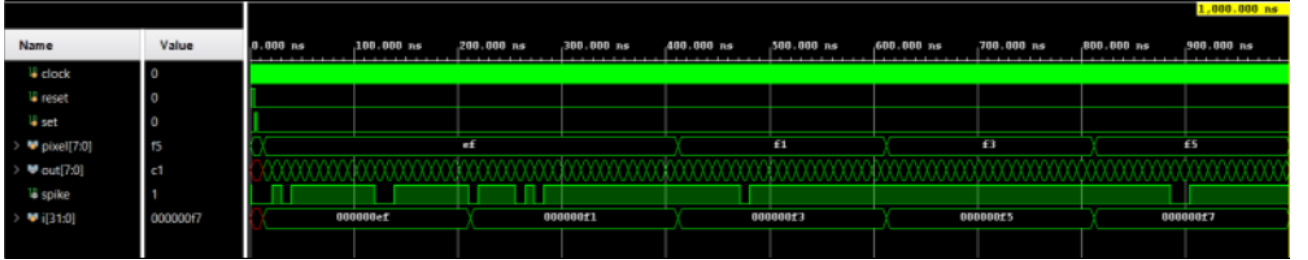


Figure 3.1: Timing diagram of Poisson Encoder

Further, a method of passing time spaced inputs to the top module was experimented with. The input was made to change after a definite clock period through the testbench which triggered the instantiated modules in the top module.

Finally, it was decided to model a LIF neuron for a convolutional spiking neural network. Multidimensional arrays were used to store inputs and outputs of various layers. A 35 bit fixed point format was used to represent the numerical values.

Chapter 4

Implementation and Results

The implementation was carried out in two parts: Software and HDL implementation.

The Software implementation of the SNN was done in a series of steps that included deciding the software libraries to be used, format of loading data, determination of the architecture followed by training.

The HDL implementation involved dividing the construction of the network into a series of modules and tasks representing the different layers of the SNN architecture. The details of these followed by the results of implementation have been presented.

4.1 Software Implementation

4.1.1 Libraries used

The following libraries were used in the code base.

- Pytorch-lightning : A pytorch research framework which reduces a lot of boiler plate code.
- Snntorch : A pytorch based library for implementing spiking neural networks
- Tonic : Used to load event based data, here used to load DVS128 dataset.
- Tensorboard : To visualize logs generated during training.
- Optuna : For hyperparameter tuning
- Qtorch : For quantizing network weights to Fixed point representation

4.1.2 Loading data

DVS128 is event based dataset, records events that is spikes based on timestamp, camera, (x,y) coordinate of the pixel generating the spike. The loaded data is accumulate events to frames by slicing along constant time into constant number of frames and also reshaping the data the sensor size. The dimension of sensor is (2, 128, 128). Finally shape of data is (num_time_bins, 2, 128, 128). The data is fed to the network for training with batch size of 16.

4.1.3 Architecture

The architecture used here is convolution spiking neural networks inspired from work presented in [3]. In Equation 4.3 the matrix W is a linear operator and is replaced with a convolution operator. Using convolution makes the number of learnable parameters independent of the input size. CNN are the go to architecture in image classification tasks, so extending it to spiking neural networks.

A stride = 2 was used in all convolution layers to avoid Maxpool layers. In case of repetition the for intermediate convolution blocks, padding is given and $in_{channels} = out_{channels}$ to preserve the dimensions after the first convolution.

12C5K2P2R means

- 12C means 12 out channels
- 5K means kernel size is 5
- 2P means padding of 2 is given, if P isn't mentioned the 0 padding
- 2R means repeat the convolution block twice i.e. add 1 block after main block, if R is not mentioned the no repetition.

1024FC11 means fully connected layer with 1024 $in_{features}$ and 11 $out_{features}$

D refers to the Dropout layer. Two distinct architectures have been trained and their details have been presented in Table 4.1.

Architecture	Parameters
4C5K2P-8C5K2P-8C3K1P-16C3K1P-D-1024FC11	14 K
8C5K2R-D-16C5K2R-D-32C3K2R-D-64C3K2R-D-2304FC11	106 k

Table 4.1: CSNN architectures

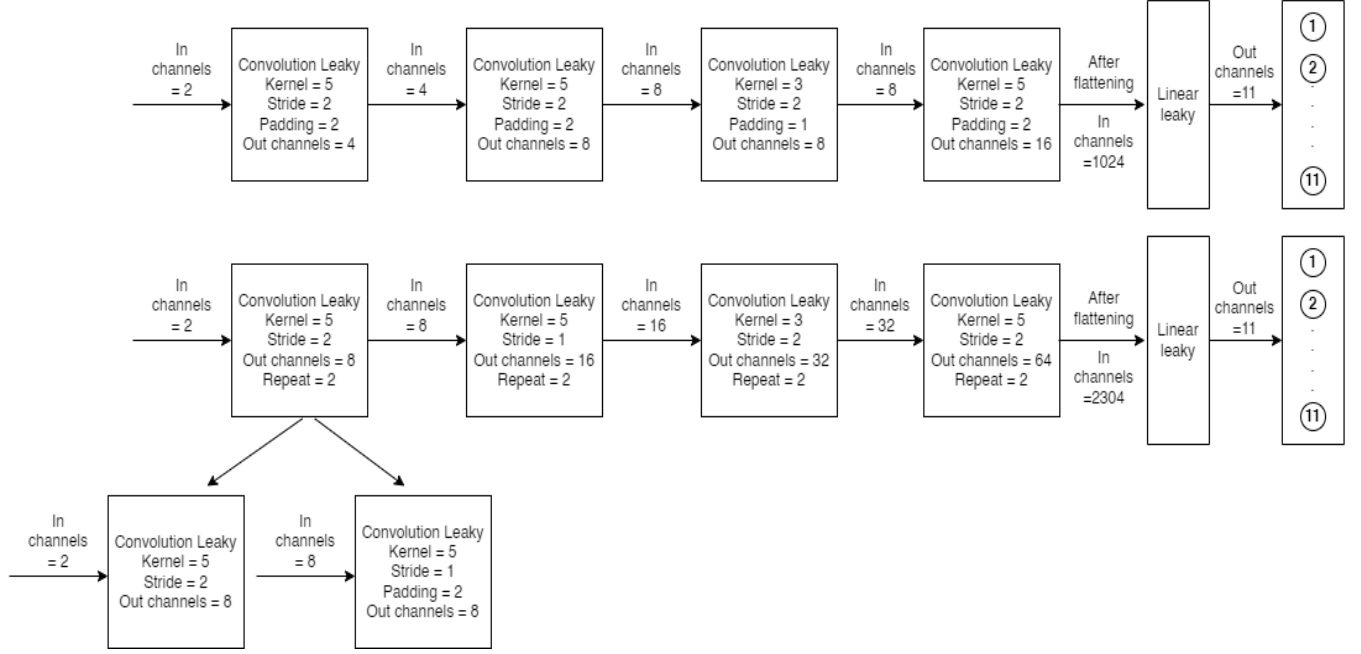


Figure 4.1: Forward pass of network for one timestep for above two architectures

4.1.4 Training

Training is done on NVIDIA DGX Station A100 which has 4 NVIDIA A100 gpus. DistributedDataParallel strategy is used to train across multiple gpus.

Optimizer used here is Adam optimizer with learning rate $1e-3$ and momentum = $(0.9, 0.999)$. Loss function used here is cross entropy rate loss. For first architecture in 4.1 training was done for maximum of 600 epochs. Learning rate schedulers are also used to change learning rate gradually for effective training. Learning rate schedulers used :

- Constant learning rate: Learning rate is kept constant with value of $1e-3$.
- Step decay learning rate: Decays the learning rate of each parameter group by gamma every step_size epochs.
- Cosine Annealing learning rate: Set the learning rate of the optimizer using a cosine annealing schedule.

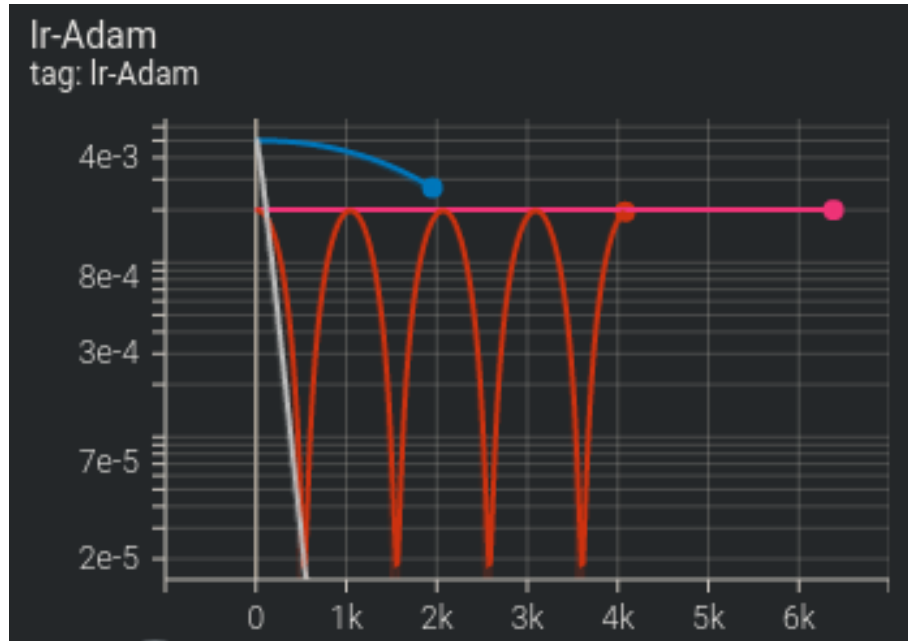


Figure 4.2: Learning rate schedulers

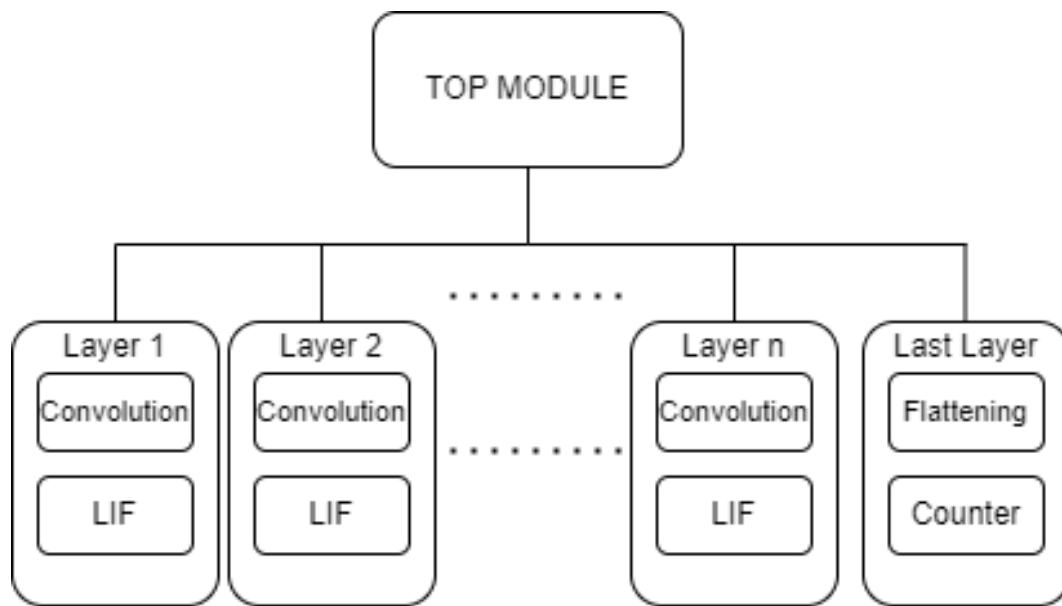
The following hyperparameters are fine tuned using Optuna for second architecture in 4.1 for 50 epochs. The best parameters were trained further for 400 epochs. The LIF hyperparameters used

- β
- threshold
- slope of surrogate gradient
- Learning rate
- Dropout layer

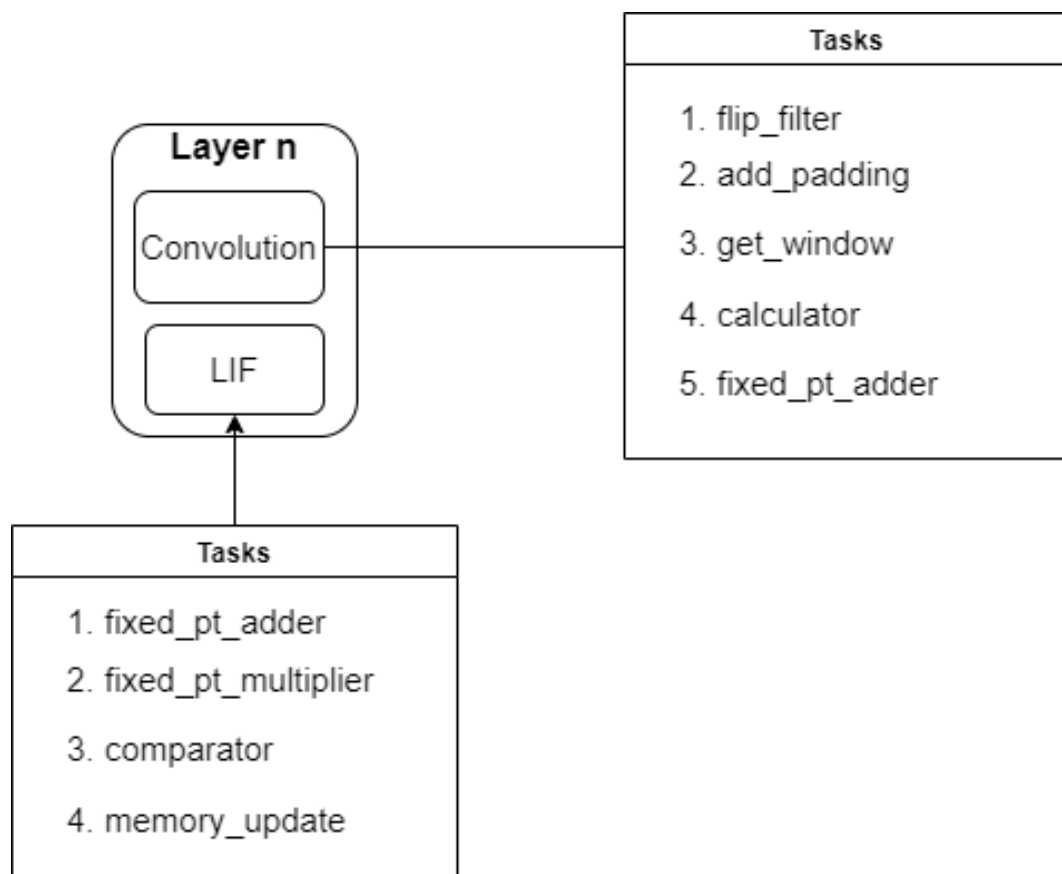
4.2 HDL Implementation

The HDL implementation aims to develop a synthesizable code for the architecture described in the previous section. The architecture implemented is 4C5K2P–8C5K2P–8C3K1P–16C3K1P–D–1024FC11. As seen in the block diagram in Section 4.1.3, each layer consists of a convolutional sublayer followed by a layer of neurons. The first layer has inputs as the data from the event-based dataset. Since the data is time varying, provision has been made to update the input register at every iteration. The last layer consists of linear layer which results into a flattened one-dimensional array, followed by a counter. The counter keeps account of the number of spikes fired at each of the 11 positions. The label corresponding to the array position with maximum spike count is the output of the network. The top module encapsulates all these individual layers which have to be run sequentially. The block diagram for the same is shown in Figure 4.3a below.

Each of the modules shown in the block diagram comprises SystemVerilog 'tasks' which helps to make the code modular and reusable since a lot of redundant operations have to be performed. Details of the various tasks are delineated in the following subsections and a graphical representation is shown in the Figure 4.3b.



(a) Block diagram of the top module



(b) Tasks in SNN layer

Figure 4.3: HDL Block Diagram

4.2.1 Passing Values to the Top Module

All the inputs need to be stored in registers to be passed to the sub-modules for computation. These inputs include the image data, filter/kernels and the biases for each layer which have been modelled as *registers* in the code. Owing to the size of data being dealt with, these registers cannot be modelled as input or output ports despite the advantage of filling in data in one go. Hence, data needs to be passed serially from the testbench.

A fast clock ticking at 5ns fills in the data at the positive edge of the clock and raises a flag once all the registers have been filled. The number of I/O ports are drastically reduced using this method since only the address or position where the data is to be stored and the data to be stored are needed.

4.2.2 Fixed Point Representation of Numbers

Most of the computations taking place in the network are involving small decimal numbers. The weights constituting the kernel are between -1 to 1. In such a case, the precision after the decimal point is of utmost importance. Nonetheless, only addition and subtraction operations have to be primarily performed. Using floating points is difficult to implement in hardware and its computations are complex. Hence, a fixed point representation of numbers has been chosen for this project.

A 35 bit fixed point number with 1 signed bit, 2 integer bits and 32 decimal bits has been chosen for representation. The signed bit is 1 and rest of the number is represented as 2's complement.

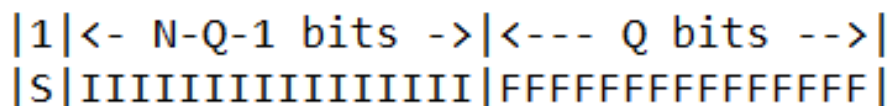


Figure 4.4: Fixed point representation

4.2.3 Convolution module

Convolution is performed between the output of the previous layer and the weight matrix, referred to as the filter or kernel. Traditionally convolution is a multiply-and-add operation. The kernel is of a much smaller dimension when compared to the input matrix. Hence a sliding window extracts a window of kernel's dimension and performs convolution on it. Since there are multiple channels in the input, output and kernel, a floating point addition operation is performed along the breadth of the matrix to output a matrix of the desired dimension and number of channels.

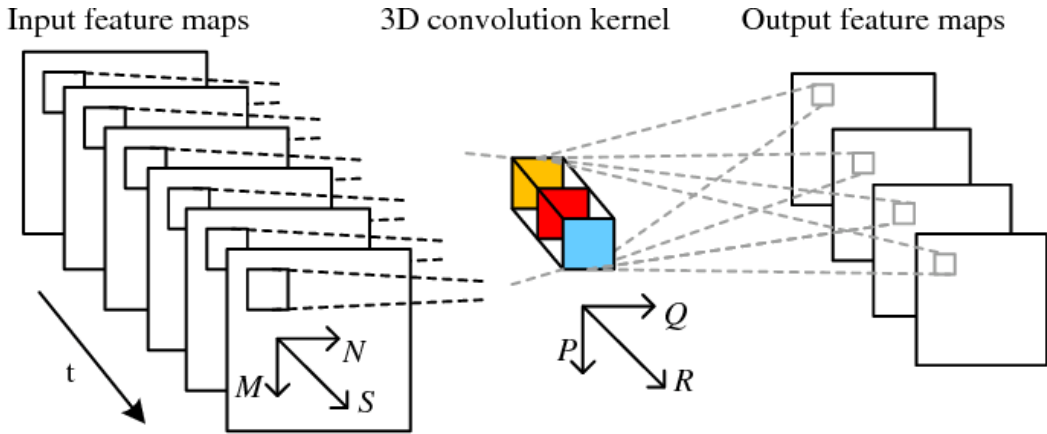


Figure 4.5: Multichannel Convolution

Since the previous layer is the LIF layer (except for the first layer), its output matrix will be a sparse binary matrix. Hence, the convolution operation is reduced to simply keep-if-1 operation. That is, if there is a '1' in the input matrix, we just store the value of the filter at that position in the output matrix instead of performing multiplications with '0's and '1's. If the value is '0' then it is simply discarded.

All these jobs are distributed among tasks:

1. **flip_filter:** This is the first step in convolution that flips the filter matrix along the row and column before the actual multiply-add operation

begins.

2. **add_padding:** For proper convolution to be computed between 2 matrices, a padding has to be added to the input matrix. This is layer of zeros appended to the input matrix.
3. **get_window:** This task works as the sliding window that moves across the input matrix. The (x,y,z) start indices are provided as input to the task guiding it to the location from where the window has to be extracted considering the stride as well. This window is then passed to the 'calculator' task for further processing.
4. **calculator:** This task performs the keep-if-1 operation mentioned above followed by the addition operation. The filter value is stored only if the value at the corresponding location in the input matrix is '1'. Else, a '0' is placed in the resultant matrix. The fixed_pt_adder task is then called to perform addition on the 35 bit values.
5. **fixed_pt_adder:** This task performs fixed point addition or subtraction operation considering a 35 bit fixed point number with 1 signed bit, 2 integer bits and 32 decimal bits.

4.2.4 LIF module

The LIF model remembers its previous inputs by adding each one to its membrane potential and then forgetting them using a leak (decay) parameter. In Verilog, a single neuron based on the LIF model has been implemented. The neuron's operation is determined by the input spikes it receives. These spikes can be thought of as the outputs of previous neurons in a wider network, simulating the operation of a 'synapse' in the brain.

This neuron's action is governed by three main currents: driving current (I_d), leakage current (I_{leak}), and spiking current (I_{sp}). As a result, the current obtained at the neuron's output can be represented as:

$$Cdv/dt = -(V - V_{rest})/R + I_d + I_{sp} \quad (4.1)$$

$$\text{Here, } (V - V_{rest})/R = I_{leak} \quad (4.2)$$

The membrane potential of LIF is given as

$$V[t] = \beta V[t - 1] + WX[t] - S_{out}[t - 1]\theta \quad (4.3)$$

When $V[t]$ is greater than the threshold voltage V_{th} the neuron gives a spike output and then enters a refractory period during which it does not respond to incoming spikes. The voltage drops after the spike is generated and it slowly returns to the resting potential V_{rest} .

The operation of the neuron is based on the input spikes received by the neuron. These spikes can be considered as the outputs of the preceding neurons in a larger network; thereby emulating the action of a neural 'synapse'. A state diagram has been devised for implementing the LIF neuron. The 4 important states of neuron operation: Resting, Rising, Spiking and Refraction are implemented.

The tasks used to fulfill these operation are:

1. **fixed_pt_adder and fixed_pt_multiplier** Perform multiplication and addition on numbers taking into consideration the signed bit, following rules of fixed point arithmetic.

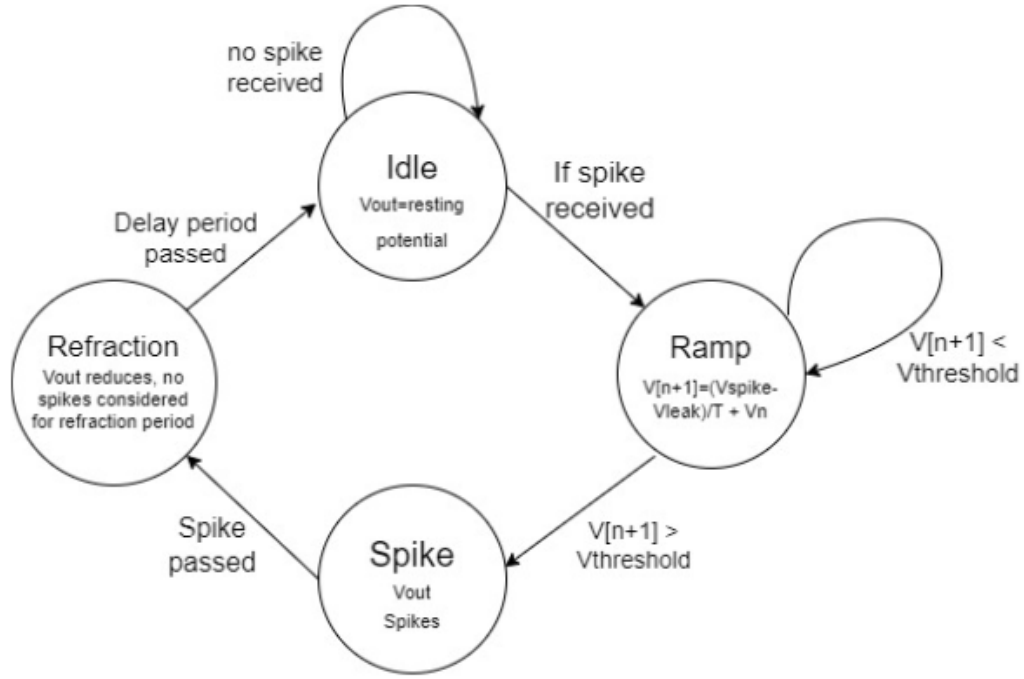


Figure 4.6: State Diagram

2. **comparator:** Compares the current membrane potential with the threshold voltage to determine whether a spike has to be fired or not.
3. **memory_update:** Since the neuron needs to hold the membrane potential reached from the previous iteration, this task takes input as the past potential, updates it for the current iteration and returns it to the top module to be passed as input in the next iteration.

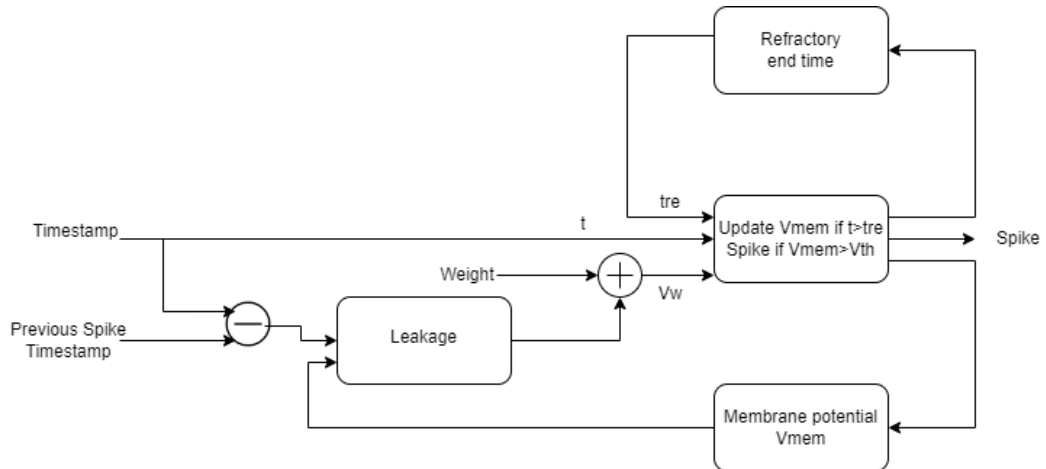


Figure 4.7: LIF Block Diagram

4.2.5 Flattening and Counter Modules

The flattening module acts as a linear layer converting large matrix arriving from the preceding layer to a linear array of size equal to the number of labels in the output: 11 in this case. This operation is performed by matrix multiplication. Again, since the input matrix is sparse binary, the computation is reduced to mere additions.

The output of the matrix multiplication is passed through a LIF layer which returns output as either 0 or 1 for each of the 11 positions in the array. The counter module keeps count of the ones appearing at each position in the flattened array across all iterations. The position holding maximum count corresponds to the output label.

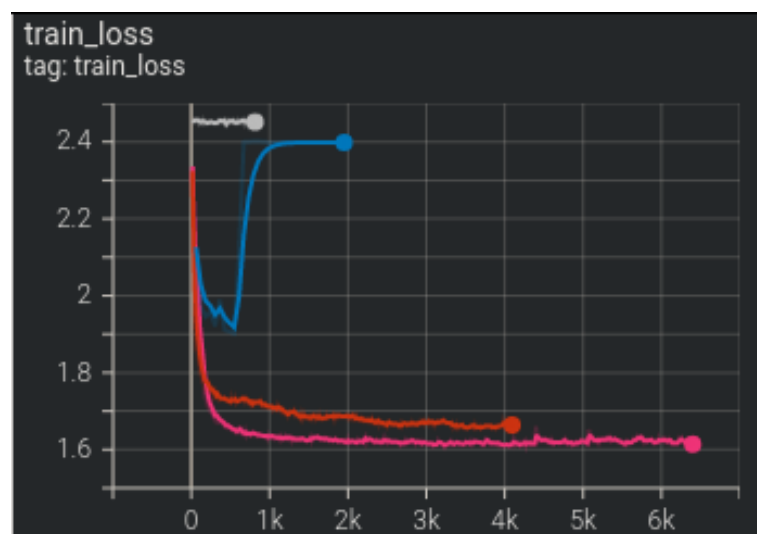
4.2.6 Method of Relaying Flags

The several layers and sub-layers in the network have to be executed sequentially: one following the other. But when the top module begins execution, all modules are run simultaneously. In order to enforce sequential operation, a method of relaying flags has been employed. Here, a given module executes only when its start flag is set by the previous module. After this module completes execution it clears the start flag and sets another flag which signals the next module to begin operation. This way tasks occur consecutively.

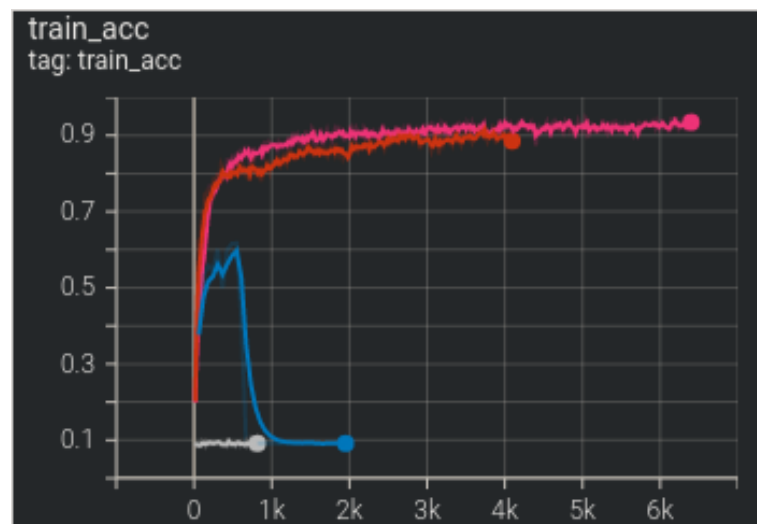
4.3 Results and Discussion

4.3.1 Software

Firstly training was done on first architecture of Table 4.1 As seen in Figure 4.8 and Figure 4.9 the results are similar for Cosine Annealing scheduler and no scheduler, so to reduce complexity.

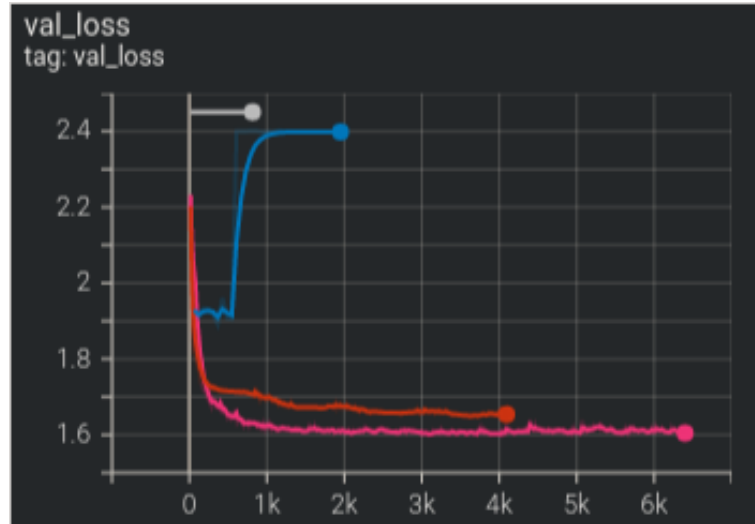


(a) Training Loss

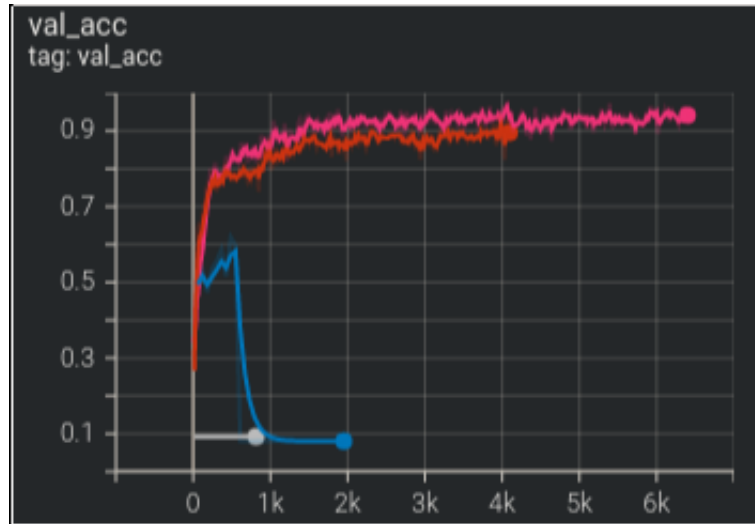


(b) Training Accuracy

Figure 4.8: Training Curves



(a) Validation Loss



(b) Validation Accuracy

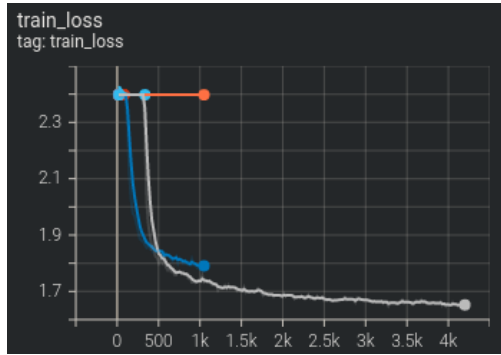
Figure 4.9: Validation Curves

In order to increase accuracy further the number of filters are increased in second architecture of Table 4.1. After hyperparameter tuning the priorities of hyperparameter are shown in Table 4.2. As it can be seen slope is the most important parameter it has been fine tuned further by keeping other value constant. Final values of the hyperparameters values are mentioned in Table 4.2.

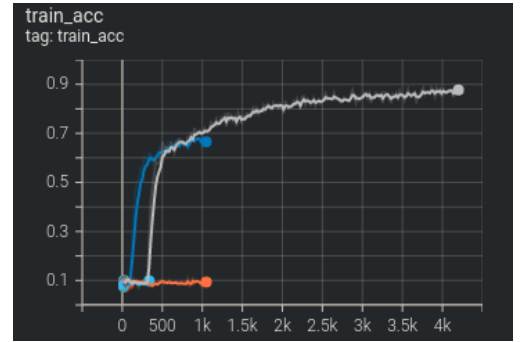
Plots shown in Figure 4.10 are for all hyperparameter tuning.

hyperparameters	Priority	Tuned Values
Slope of surrogate gradient	0.28	92
Beta	0.22	0.7
Threshold	0.22	0.3
Learning rate	0.16	10^{-3}
Dropout rate	0.13	0.1

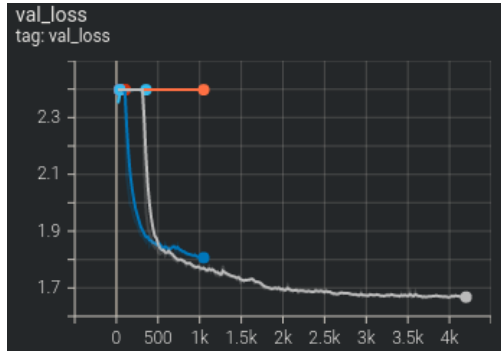
Table 4.2: Hyperparameters Priority



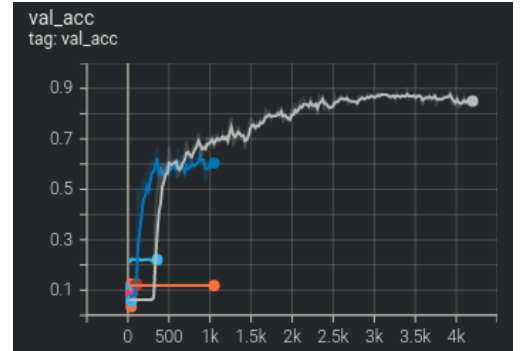
(a) Training Loss



(b) Training Accuracy



(c) Validation Loss



(d) Validation Accuracy

Figure 4.10: Tuning all hyperparameters

The following figure 4.13 is for tuning slope of surrogate gradient descent.

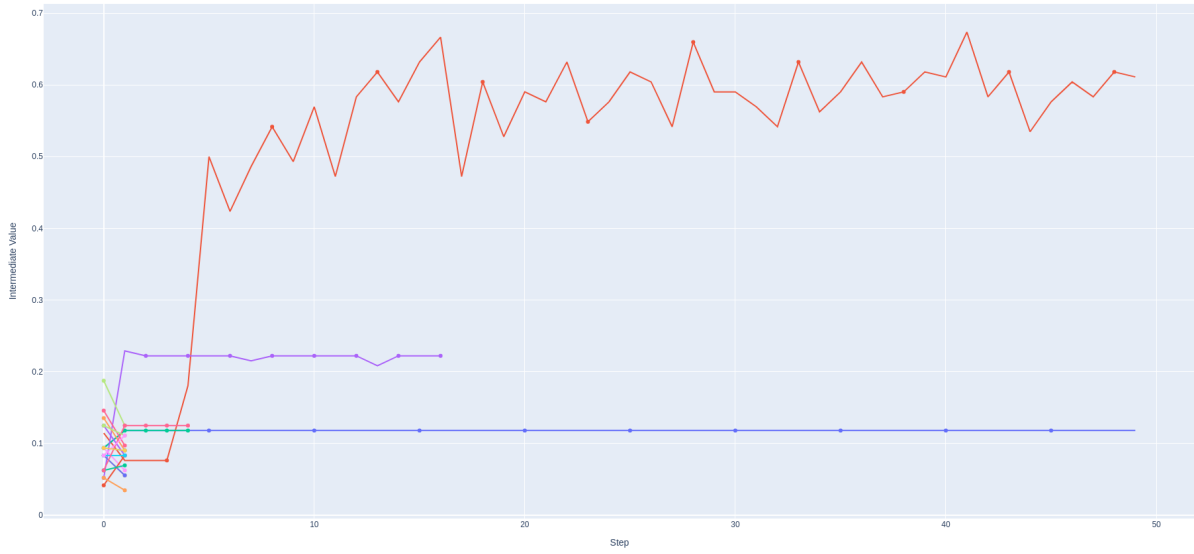


Figure 4.11: Intermediate values of val accuracy while hyper parameter tuning

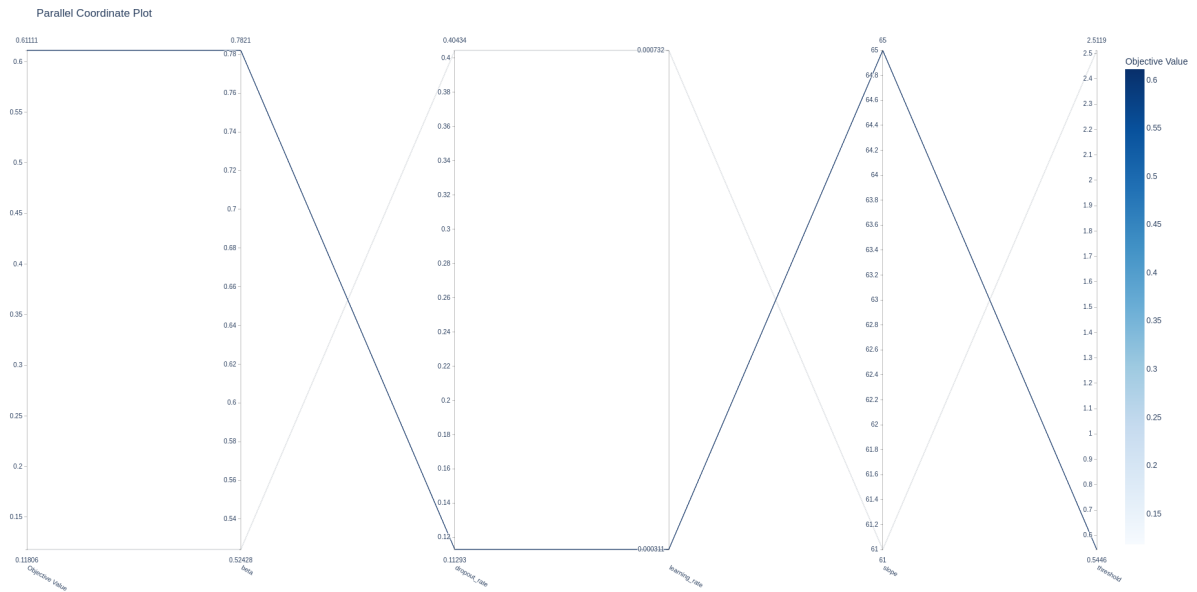
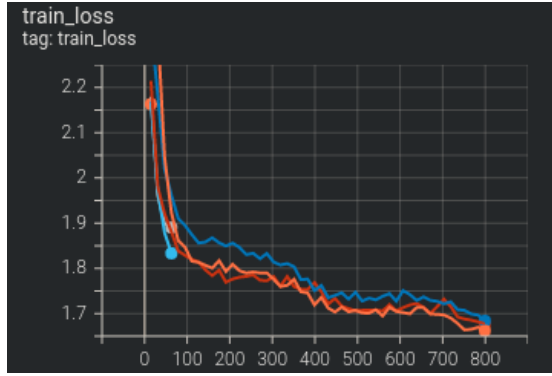
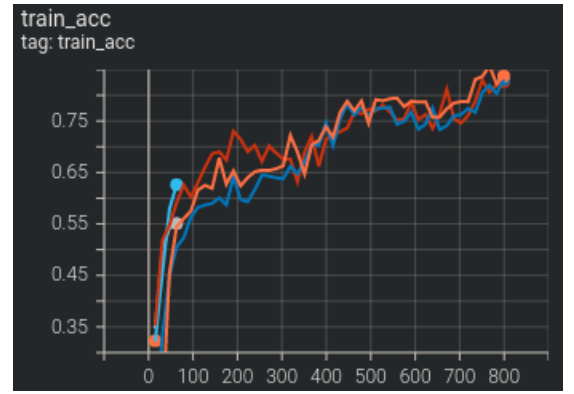


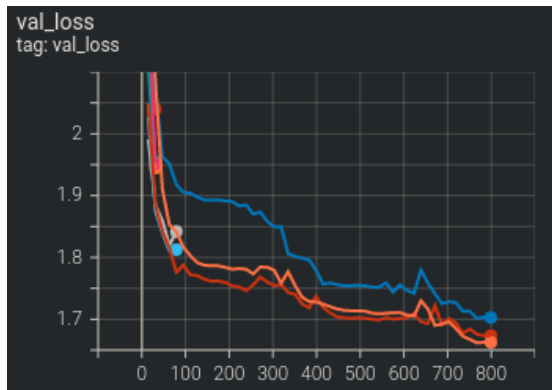
Figure 4.12: Parallel coordinate values of hyperparameters



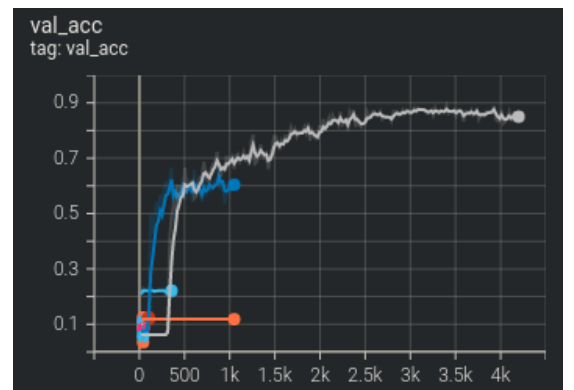
(a) Train Loss



(b) Train Accuracy



(c) Validation Loss



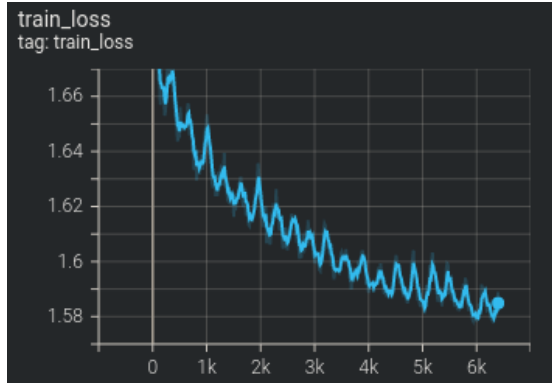
(d) Validation Accuracy

Figure 4.13: Tuning slope of surrogate gradient descent

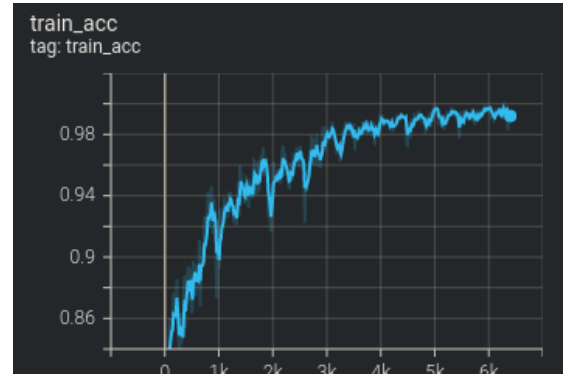


Figure 4.14: Intermediate values for slope tuning

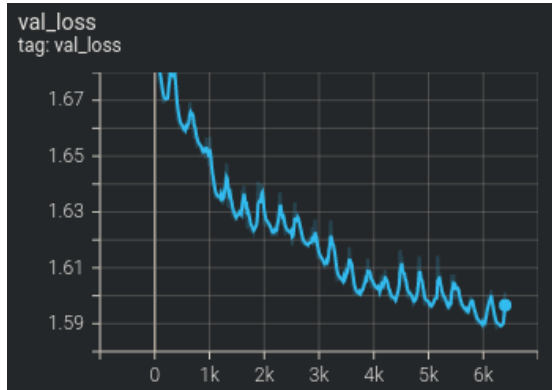
After obtaining the hyperparameter values the network was trained for another 400 epochs. The training , validation curves are discussed in Figure 4.15.



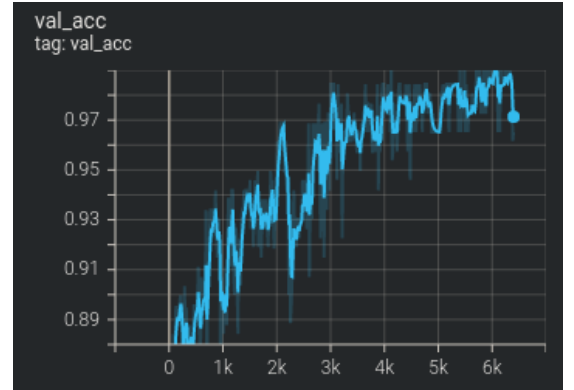
(a) Training Loss



(b) Training Accuracy



(c) Validation Loss



(d) Validation Accuracy

Figure 4.15: Full training with hyperparameters

The test accuracies for different architectures has been compared in Table 4.3. An improvement in the accuracy for the second architecture can be observed.

Architecture	Test Accuracy
4C5K2P-8C5K2P-8C3K1P-16C3K1P-D-1024FC11	0.80
8C5K2R-D-16C5K2R-D-32C3K2R-D-64C3K2R-D-2304FC11	0.88

Table 4.3: Accuracy

4.3.2 Hardware

The Artix-7 FPGA of xa7a75tcsg series has been used for synthesis and implementation in Xilinx Vivado software.

Neuron Model

The timing diagram for 1 neuron has been shown in the figure below.

- $\text{in}[2:0][2:0]$ is the set of inputs with corresponding weights stored
- in $\text{wt}[2:0][2:0][2:0]$.
- The variable $\text{out}[0][2:0]$ goes to 1 when $V_{\text{membrane}} > \text{Threshold voltage}$
 V_{membrane} returns to resting value after a delay accounting for the refractory period.

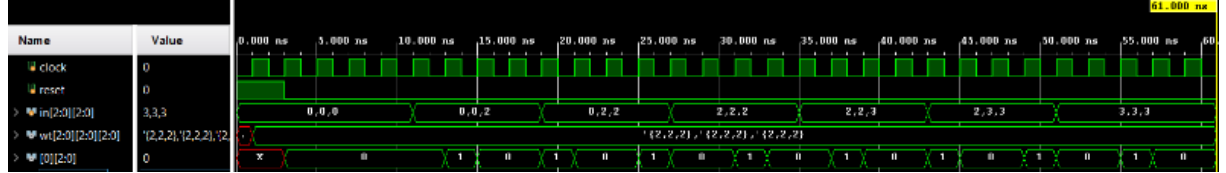


Figure 4.16: Neuron timing diagram

The synthesis results have been shown in Figure 4.17. The implementation results have been highlighted in Figure 4.18 along with the power utilization report in 4.19

As can be observed, less than 1% of the total resources are being used to model a single neuron. Also, most of the power is used in maintaining the state of the I/O ports.

1. Slice Logic

Site Type	Used	Fixed	Available	Util%
Slice LUTs*	57	0	47200	0.12
LUT as Logic	57	0	47200	0.12
LUT as Memory	0	0	19000	0.00
Slice Registers	0	0	94400	0.00
Register as Flip Flop	0	0	94400	0.00
Register as Latch	0	0	94400	0.00
F7 Muxes	0	0	31700	0.00
F8 Muxes	0	0	15850	0.00

7. Primitives

Ref Name	Used	Functional Category
LUT2	31	LUT
OBUF	19	IO
IBUF	19	IO
LUT4	17	LUT
LUT6	15	LUT
CARRY4	9	CarryLogic
LUT5	4	LUT
LUT3	4	LUT
LUT1	3	LUT

Figure 4.17: Synthesis models of neuron model



Figure 4.18: Implementation Results

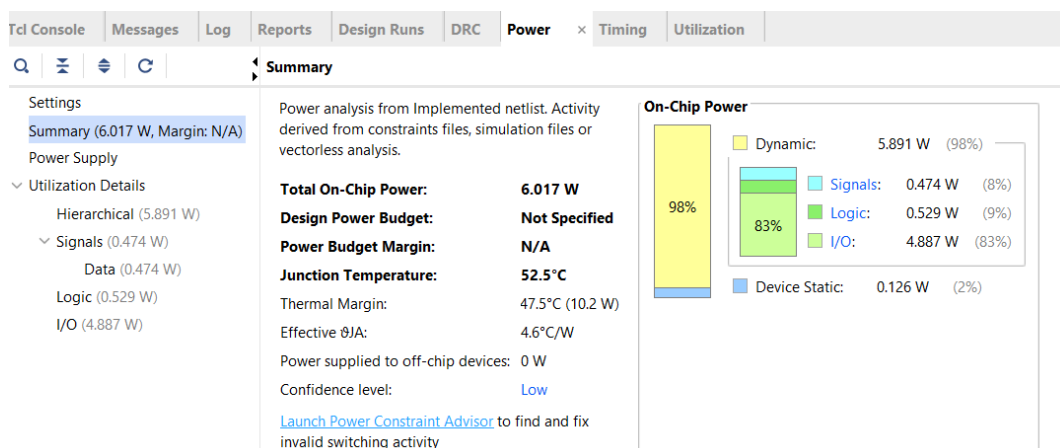


Figure 4.19: Power Utilization Report

Convolution Model

The convolution module is used to calculate the convolution of the kernel matrix with the input matrix which can be the initial image matrix or the weight matrix in the subsequent layers. This is a generalized module which can be scaled to desired number of matrix sizes. For this synthesis report, the model has been scaled down due to the excessive time required to carry out the synthesis.

For testing purposes, the input matrix and kernel have been hard-coded. The output of the code has been shown in the figure below.

```
Input Kernel:
  1  2  3
  1  2  3
  1  2  3
Flipped Kernel:
  3  2  1
  3  2  1
  3  2  1
Input matrix:
  1  1  1  1  1  1  1
  1  1  1  1  1  1  1
  1  1  1  1  1  1  1
  1  1  1  1  1  1  1
  1  1  1  1  1  1  1
  1  1  1  1  1  1  1
  1  0  0  0  1  0  0
Convolution:
18
18
18
18
18
18
15
13
15
```

Figure 4.20: Convolution output

The module is subdivided into tasks as highlighted in the preceding section. The major tasks being the *flip_filter* and the *calculator* tasks. Each of them have been synthesised individually apart from the synthesis of the entire module.

In Figures 4.21 through 4.23, the synthesis and implementation results of the scaled down convolution block are provided.

```

-----
Detailed RTL Component Info :
+---Registers :
          11 Bit   Registers := 446
           5 Bit   Registers := 4
           1 Bit   Registers := 2
+---Muxes :
      2 Input   1 Bit   Muxes := 608
-----

```

Figure 4.21: Convolution module synthesis results

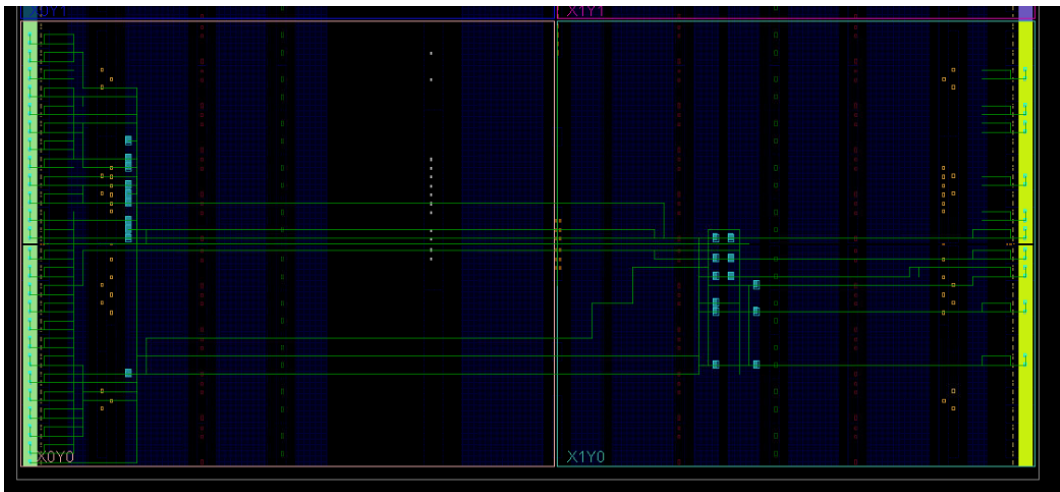


Figure 4.22: Convolution module implementation results

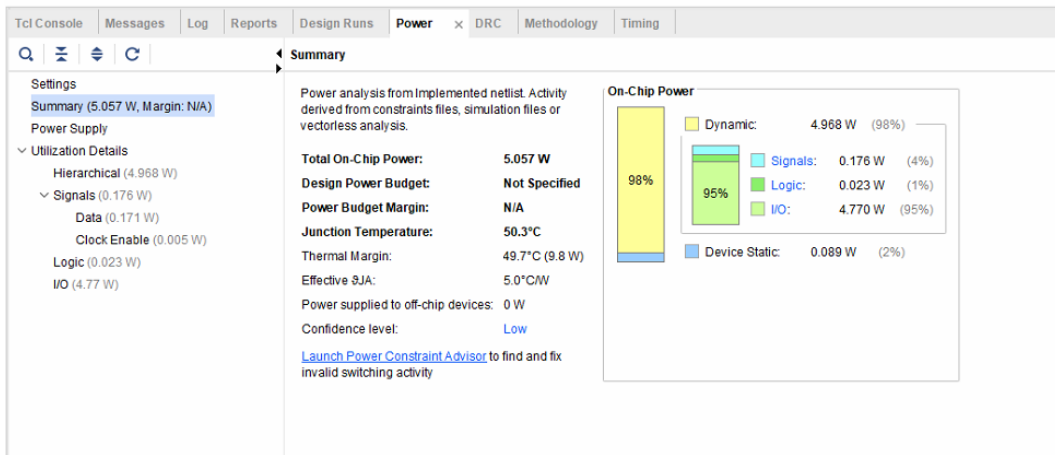


Figure 4.23: Convolution module power utilization report

The synthesis results of the major tasks in the convolution module are shown below.

1. Slice Logic

Site Type	Used	Fixed	Available	Util%
Slice LUTs*	8	0	47200	0.02
LUT as Logic	8	0	47200	0.02
LUT as Memory	0	0	19000	0.00
Slice Registers	110	0	94400	0.12
Register as Flip Flop	110	0	94400	0.12
Register as Latch	0	0	94400	0.00
F7 Muxes	0	0	31700	0.00
F8 Muxes	0	0	15850	0.00

4. IO and GT Specific

Site Type	Used	Fixed	Available	Util%
Bonded IOB	85	0	210	40.48
Bonded IPADs	0	0	2	0.00
PHY_CONTROL	0	0	6	0.00
PHASER_REF	0	0	6	0.00
OUT_FIFO	0	0	24	0.00
IN_FIFO	0	0	24	0.00
IDELAYCTRL	0	0	6	0.00
IBUFDS	0	0	202	0.00
PHASER_OUT/PHASER_OUT_PHY	0	0	24	0.00
PHASER_IN/PHASER_IN_PHY	0	0	24	0.00
IDELAYE2/IDELAYE2_FINEDELAY	0	0	300	0.00
ILOGIC	0	0	210	0.00
OLOGIC	0	0	210	0.00

5. Clocking

Site Type	Used	Fixed	Available	Util%
BUFGCTRL	1	0	32	3.13
BUFIO	0	0	24	0.00
MMCME2_ADV	0	0	6	0.00
PLLE2_ADV	0	0	6	0.00
BUFMRCE	0	0	12	0.00
BUFHCE	0	0	96	0.00
BUFR	0	0	24	0.00

7. Primitives

Ref Name	Used	Functional Category
FDRE	110	Flop & Latch
OBUF	56	IO
IBUF	29	IO
LUT6	5	LUT
LUT4	2	LUT
LUT5	1	LUT
BUFG	1	Clock

Figure 4.24: Synthesis models of convolution module's *flip_filter* task

The synthesis reports of the *calculator* block has been shown in Figure 4.25. As is evident, this task requires much more resources compared to the *flip_filter* module since the major comparisons and computations are performed in this module.

1. Slice Logic

Site Type	Used	Fixed	Available	Util%
Slice LUTs*	668	0	47200	1.42
LUT as Logic	668	0	47200	1.42
LUT as Memory	0	0	19000	0.00
Slice Registers	1155	0	94400	1.22
Register as Flip Flop	1155	0	94400	1.22
Register as Latch	0	0	94400	0.00
F7 Muxes	0	0	31700	0.00
F8 Muxes	0	0	15850	0.00

7. Primitives

Ref Name	Used	Functional Category
FDRE	1155	Flop & Latch
LUT6	349	LUT
LUT4	174	LUT
LUT5	104	LUT
LUT3	56	LUT
OBUF	55	IO
IBUF	54	IO
OBUFT	1	IO
BUFG	1	Clock

Figure 4.25: Calculator Task

Flattening Layer

The flattening module performs the final matrix multiplication resulting into a single dimensional array of 11 elements. These 11 elements represent the labels of the SNN for classification.

The output of the matrix multiplication is passed through a LIF layer which returns output as either 0 or 1 for each of the 11 positions in the array. The counter module keeps count of the ones appearing at each position in the flattened array across all iterations. The position holding maximum count corresponds to the output label.

As can be observed from Figure 4.26, the maximum count equalling 54 has been achieved for label 2, hence that is the output which has been verified with the software output results.

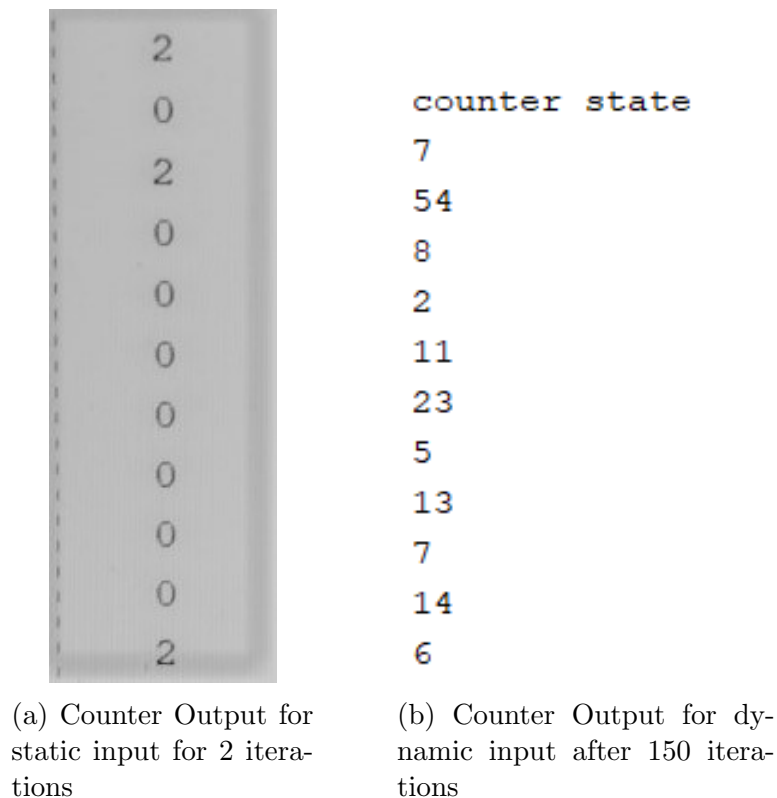


Figure 4.26: Simulation results

4.3.3 Comparing Classical CNN with SNN

The most crucial computation in both CNN and CSNN is convolution, which involves repetitive multiply-and-add operation in case of a traditional CNN and only a keep-if-non-zero operation in case of CSNN. This distinction will evidently produce less hardware for SNN s compared to CNN, which can be verified through synthesis results shown in Figures 4.27 and 4.28.

A 15 bit matrix of 11x11 elements has been considered for the purpose of demonstration. The results of synthesis show that SNN requires require only standard LUTs which are available in abundance on the FPGA device. On the other hand, CNN relies heavily on the usage of DSP blocks for computation. Around 67% of available DSP blocks have been used up for carrying out a single convolution operation. It is evident that when scaled up, the hardware resources would no longer be able to support the computations in case of CNNs. On the other hand, merely 1% of LUT slices have been used by a spike based convolution operation, providing room for scaling up.

```
1. Slice Logic
-----
```

Site Type	Used	Fixed	Available	Util%
Slice LUTs*	484	0	47200	1.03
LUT as Logic	484	0	47200	1.03
LUT as Memory	0	0	19000	0.00
Slice Registers	1936	0	94400	2.05
Register as Flip Flop	1936	0	94400	2.05
Register as Latch	0	0	94400	0.00
F7 Muxes	0	0	31700	0.00
F8 Muxes	0	0	15850	0.00

7. Primitives

Ref Name	Used	Functional Category
IBUF	2058	IO
OBUF	1936	IO
FDRE	1936	Flop & Latch
LUT6	363	LUT
LUT5	121	LUT
BUFG	1	Clock

3. DSP

Site Type	Used	Fixed	Available	Util%
DSPs	0	0	180	0.00

4. IO and GT Specific

Site Type	Used	Fixed	Available	Util%
Bonded IOB	3994	0	210	1901.90
Bonded IPADs	0	0	2	0.00
PHY_CONTROL	0	0	6	0.00
PHASER_REF	0	0	6	0.00
OUT_FIFO	0	0	24	0.00
IN_FIFO	0	0	24	0.00
IDELAYCTRL	0	0	6	0.00
IBUFDS	0	0	202	0.00
PHASER_OUT/PHASER_OUT_PHY	0	0	24	0.00
PHASER_IN/PHASER_IN_PHY	0	0	24	0.00
IDELAYE2/IDELAYE2_FINEDELAY	0	0	300	0.00
ILOGIC	0	0	210	0.00
OLOGIC	0	0	210	0.00

Figure 4.27: Convolution for SNN

1. Slice Logic

Site Type	Used	Fixed	Available	Util%
Slice LUTs*	0	0	47200	0.00
LUT as Logic	0	0	47200	0.00
LUT as Memory	0	0	19000	0.00
Slice Registers	0	0	94400	0.00
Register as Flip Flop	0	0	94400	0.00
Register as Latch	0	0	94400	0.00
F7 Muxes	0	0	31700	0.00
F8 Muxes	0	0	15850	0.00

7. Primitives

Ref Name	Used	Functional Category
IBUF	3873	IO
OBUF	1936	IO
DSP48E1	121	Block Arithmetic
BUFG	1	Clock

3. DSP

Site Type	Used	Fixed	Available	Util%
DSPs	121	0	180	67.22
DSP48E1 only	121			

4. IO and GT Specific

Site Type	Used	Fixed	Available	Util%
Bonded IOB	5809	0	210	2766.19
Bonded IPADs	0	0	2	0.00
PHY_CONTROL	0	0	6	0.00
PHASER_REF	0	0	6	0.00
OUT_FIFO	0	0	24	0.00
IN_FIFO	0	0	24	0.00
IDELAYCTRL	0	0	6	0.00
IBUFDS	0	0	202	0.00
PHASER_OUT/PHASER_OUT_PHY	0	0	24	0.00
PHASER_IN/PHASER_IN_PHY	0	0	24	0.00
IDELAYE2/IDELAYE2_FINEDELAY	0	0	300	0.00
ILOGIC	0	0	210	0.00
OLOGIC	0	0	210	0.00

Figure 4.28: Convolution for CNN

Chapter 5

Conclusion and Future Scope

5.0.1 Conclusion

This project aimed at training of an event based dataset on a spiking neural network and its subsequent implementation in HDL for deployment on as FPGA device. The DVS128 dataset developed by IBM has been used to train a 4 layer architecture of convolutional SNN. The training has achieved a fair accuracy of around 0.88, which is lower than some of the popular published works. The reason for the same being use of a simple architecture with a very limited depth trained for a considerably limited number of epochs. The neuron model used, architecture, training method and software libraries used have been decided upon thorough literature review and experimentation.

In the HDL implementation, the architecture has been modelled in SystemVerilog and results obtained through simulation in Xilinx Vivado. The HDL code has been so designed such that it is completely synthesizable, modular as well as extensible. Methods of serially passing values to the top module from the test bench for synthesis feasibility, relaying flags to accurately trigger modules sequentially have been employed which are important in the hardware design of any neural network. Synthesis and implementation

for an Artix-7 FPGA has been performed and the results have been discussed. Lastly, a comparison of the traditional CNN and the novel CSNN has been done based on the resource utilization for a convolution operation, thereby demonstrating that a neuromorphic architecture can perform the same operations with lesser resources compared to a traditional CNN.

In our future work, we aim to improve the accuracy of the architecture through optimization of various hyperparameters. We also aim to improve the resource utilization of the FPGA which is likely to be overutilized due to concurrent operation on a huge number of neurons which also involves some fixed point calculations. We have attempted to provide a general overview of the methodology involved in developing neuromorphic software and hardware on a small scale. This has opened several doors for further research and advancements.

5.0.2 Future Scope

The field of neuromorphic computing and Spiking Neural networks is a blossoming field which is currently in its nascent stage. It has quite a long way to go before it reaches a stage that the traditional neural networks and hardware are at today. This includes availability of benchmark datasets for SNNs, standard training methods, software packages and verified hardware modelling techniques. This in turn provides a huge opportunity for research in the field. Through this project we have attempted to train an SNN on an event-based dataset developed by IBM. We then have implemented the network for hardware using SystemVerilog.

Future scope for this project includes training of SNN on various other datasets to analyse the robustness and flexibility of the architecture and training mechanism. Another scope could be employing a different training

technique like STDP, Reinforced SNN, etc. to deduce what type of training technique performs best for what type of architecture and dataset.

On the hardware facet, different modelling techniques can be used. For example, use of multiplexing of resources to further reduce utilization can be explored. Self-learning hardware can also be developed to further the work done in this project. This involves building hardware where connections between blocks are made and several weights deduced while the code runs in real time.

Bibliography

- [1] Jason K. Eshraghian, Max Ward, Emre Neftci, Xinxin Wang, Gregor Lenz, Girish Dwivedi, Mohammed Bennamoun, Doo Seok Jeong, and Wei D. Lu. "Training Spiking Neural Networks Using Lessons From Deep Learning". arXiv preprint arXiv:2109.12894, September 2021.
- [2] E. O. Neftci, H. Mostafa and F. Zenke, "Surrogate Gradient Learning in Spiking Neural Networks: Bringing the Power of Gradient-Based Optimization to Spiking Neural Networks," in IEEE Signal Processing Magazine, vol. 36, no. 6, pp. 51-63, Nov. 2019, doi: 10.1109/MSP.2019.2931595.
- [3] Convolutional Spiking Neural Networks for Spatio-Temporal Feature Extraction Ali Samadzadeh, Fatemeh Sadat Tabatabaei Far, Ali Javadi, Ahmad Nickabadi, Morteza Haghiri Chehreghani
- [4] Peter U Diehl and Matthew Cook. Unsupervised learning of digit recognition using spike-timing-dependent plasticity. *Frontiers in Computational Neuroscience*, 9:99, 2015.
- [5] S. Gupta, A. Vyas and G. Trivedi, "FPGA Implementation of Simplified Spiking Neural Network," 2020 27th IEEE International Conference on Electronics, Circuits and Systems (ICECS), 2020, pp. 1-4, doi:10.1109/ICECS49266.2020.9294790.

- [6] F. Akopyan et al., "TrueNorth: Design and Tool Flow of a 65 mW 1 Million Neuron Programmable Neurosynaptic Chip," in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 34, no. 10, pp. 1537-1557, Oct. 2015, doi: 10.1109/TCAD.2015.2474396.
- [7] Yaghini Bonabi Safa, Asgharian Hassan, Safari Saeed, Nili Ahmadabadi Majid, "FPGA implementation of a biological neural network based on the Hodgkin-Huxley neuron model", *Frontiers in Neuroscience*, Volume 8, 2014, DOI=10.3389/fnins.2014.00379
- [8] T. Luo et al., "An FPGA-Based Hardware Emulator for Neuromorphic Chip With RRAM," in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 2, pp. 438-450, Feb. 2020, doi: 10.1109/TCAD.2018.2889670.
- [9] Kaveh Akbarzadeh-Sherbaf, Saeed Safari, Abdol-Hossein Vahabie, A digital hardware implementation of spiking neural networks with binary FORCE training, *Neurocomputing*, Volume 412, 2020, Pages 129-142, ISSN 0925-2312, <https://doi.org/10.1016/j.neucom.2020.05.044>.
- [10] Wang Runchun M., Thakur Chetan S., van Schaik André "An FPGA-Based Massively Parallel Neuromorphic Cortex Simulator", *Frontiers in Neuroscience*, vol. 12, 2018, DOI=10.3389/fnins.2018.00213, ISSN=1662-453X
- [11] Yann LeCun, Courant Institute, NYU; Corinna Cortes, Google Labs, New York; Christopher J.C. Burges, Microsoft Research, Redmond, <http://yann.lecun.com/exdb/mnist/>
- [12] IBM Research <https://research.ibm.com/interactive/dvsgesture/>