

SPIKING NEURON ARCHITECTURE FOR INFORMATION PROCESSING

Project Report - Group 27



GODAWITA B.M.K.K. (E/18/113)
RALAPANAWA K.B.D.N.B. (E/18/277)
RATHNAYAKA. (E/18/295)

Project Supervisors: Dr. Isuru Dasanayaka

Department of Electrical and Electronic Engineering

Faculty of Engineering

University of Peradeniya, Sri Lanka

March 2024

Acknowledgement

We are grateful for the kind help and guidance provided by many people throughout this effort. Their involvement was vital to the project's success and progress.

We would like to begin by thanking our project supervisor, Dr. Isuru Dassanayake, for his significant assistance and guidance throughout *EE 405 - Undergraduate Project I*. His incisive feedback, intelligent leadership, and ongoing encouragement all contributed significantly to the project's progress and success. We are grateful for his dedication and the depth of expertise he offered with us as we worked to complete the project.

We would also want to thank Dr. R.D.B. Ranaweera, the course coordinator, for his support and the outstanding framework he provided for the course. His contributions aided in the creation of an environment in which we could effectively focus and develop our idea.

Finally, we'd want to thank our lecturers, peers, and everyone else who contributed ideas, support, and encouragement. Their participation was critical in ensuring the project's success, and we are extremely grateful for their contributions.

E/18/113 – Godawita B.M.K.K.

E/18/277 – Ralapanawa K.B.D.N.B.

E/18/295 – Rathnayaka R.M.J.B.

Department of Electrical and Electronics Engineering,
Faculty of Engineering,
University of Peradeniya.

Abstract

This research aims to improve information processing through the use of spiking neuron designs, with a specific emphasis on increasing the efficiency of unit step function approximations. To this goal, we looked into two different methods: *Chebyshev approximation* and *Piecewise Linear Gradient Approximation (PLGA)*. Our comparison research revealed which strategy achieves the optimal balance of computing efficiency and accuracy. In addition, we studied the basic principles of spiking neuron models, notably the *Leaky Integrate-and-Fire (LIF) model*, to better understand the dynamics of spiking neurons while mimicking brain-like processing. In addition to architectural research, we investigated several spike encoding strategies that affect the performance of spiking neural networks (SNNs).

Our findings provide vital insights into speeding up information processing in SNNs, potentially boosting their applicability in neuromorphic computing and other advanced neural computation disciplines.

CONTENTS

Acknowledgement	i
Abstract	ii
CONTENTS	iii
List Of Figures	iv
List Of Tables	v
List Of Abbreviations	vi
Introduction	1
1.1 Bio-inspired Neural Networks	1
1.2 Artificial Neural Network	3
1.3 Neural Networks	4
Spiking Neuron Architecture	6
2.1 Introduction	6
2.2 Data Encoding Techniques	9
2.3 Leaky-integrate and Fire Neuron	21
2.4 Mathematical Derivation of LIF	22
Mathematical Approximation of Unit Step Function	25
3.1 Sigmoid Approximation	25
3.2 Chebyshev Approximation	27
3.3 Piecewise Linear Gradient Approximation	29
Hardware Implementation of the SNN	31
Power Analysis	32
Conclusion	33
References	34

List Of Figures

Figure 1: Classification with a Decision Boundary	1
Figure 2: Basic parts of a neuron (Wikimedia Commons)	2
Figure 3: Artificial Neuron Model	3
Figure 4: Structure of a Neural Network	4
Figure 5: Backpropagation	5
Figure 6: Typical Spiking Neural Network Architecture	6
Figure 7: Spiking Neuron Models	7
Figure 8: Biphasic STDP	8
Figure 9: Rate Coding	11
Figure 10: Accuracy in Rate Coding	12
Figure 11: RC circuit model of the Latency Coding	14
Figure 12: RC model - Voltage variation along with time	15
Figure 13: Time vs Input current	15
Figure 14: Spike time vs Input value	16
Figure 15: Accuracy in Latency Coding	17
Figure 16: Delta Modulation	18
Figure 17: Spike generation with time	19
Figure 18: Delta modulated spikes	19
Figure 19: Accuracy using Delta Modulation	20
Figure 20: membrane potential characteristics of LIF neuron	21
Figure 21: RC Model	21
Figure 22: Unit Step Function	24
Figure 23: Derivative of Unit Step Function	24
Figure 24: Unit step approximation and Derivative of the approximation	25
Figure 25: Loss Curves	26
Figure 26: Throughput, Train loss and Test loss	26
Figure 27: Accuracy of Sigmoid Approximation on test data set	26
Figure 28: Chebyshev Approximation of the Unit Step Function	27
Figure 29: Latency and Throughput in Chebyshev Approximation	28
Figure 30: Loss Curves in Chebyshev Approximation	28
Figure 31: Inferencing Results	28
Figure 32: Piecewise Linear Function	29
Figure 33: Loss Curves in Piecewise Linear Approximation	30
Figure 34: Throughput and Latency in Piecewise Linear Approximation	30
Figure 35: Power Calculation of Inferencing of SNN	31
Figure 36: Timing Diagram of Inferencing in SNN	31

List Of Tables

Table 1: Similarities between Biological and Artificial Neuron

4

List Of Abbreviations

ANN	Artificial Neural Network
SNN	Spiking Neural Network
STPD	Spike Time Dependent Plasticity
MNIST	Modified National Institute of Standard & Technology
DVS	Dynamic Vision Sensor
LIF	Leaky Integrate and Fire
PLGA	Piecewise Linear Gradient Approximation

Chapter 1

Introduction

1.1 Bio-inspired Neural Networks

To increase the probability of survival, animals use a process called decision-making. For example, identifying prey and predators, avoiding threats and escape or even choose good food in between spoiled foods. All these natural instincts can be broken down to simple decision-making processes. By understanding the process of decision-making clearly and the mechanism behind it is the novel focus of the state-of-the-art neural network technology.

Neurons

Nature has developed numerous methods for data processing inside the brain. However, before building a machine that mimics the brain, a few techniques have evolved for data processing. The artificial neuron model is one of the best architectures for classification tasks.

Why we need neurons?

When considering the dataset shown in Figure 1, it can be separated by a line. What is the simplest decision boundary that can be drawn to separate the two classes? Yes, it is a **linear decision boundary**.

- $Y = W_0 + W_1X_1 + W_2X_2$
- Decision boundary is a D-1 plane
- D+1 parameters: D weights and a bias
- $y = \sigma(W^T X + W_0)$

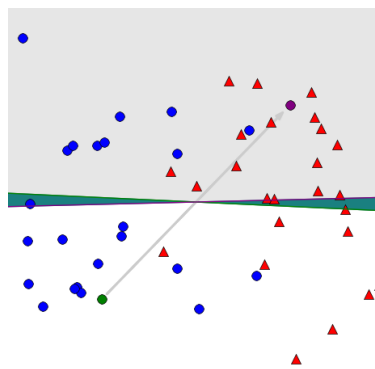


Figure 1: Classification with a Decision Boundary

▪ Biological Neuron

Neurons are fundamental element of the nervous system, and it can generate *electrical potential (Action potential)* to transmit the information. There are 3 basic functionalities in the neuron shown below in Figure 2.

1. Receive information from outside
2. Process the received information
3. Output the signal according to the processing mechanism

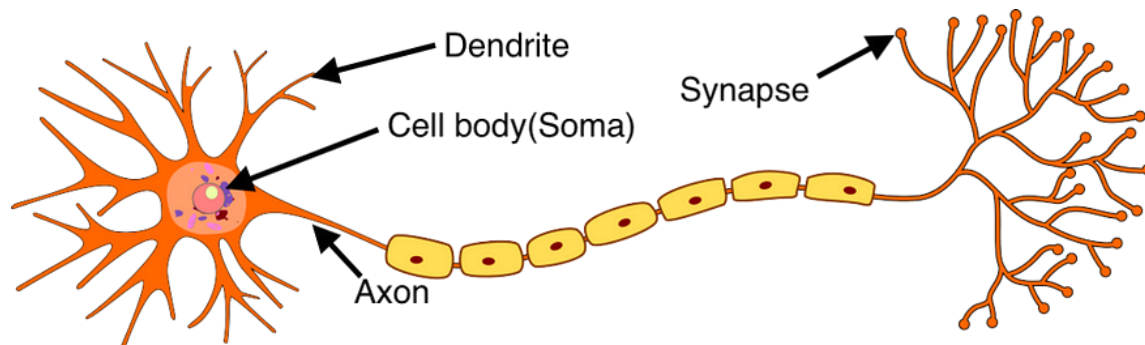


Figure 2: Basic parts of a neuron (Wikimedia Commons)

1. **Dendrite:** Responsible units for the receive the incoming information
2. **Cell body (Soma):** Processing of input signals and decision-making unit
3. **Axon:** Get processed information from neuron to relevant ends.
4. **Synapse:** Connection between axon and other neuron dendrites.

▪ Working Mechanism

Dendrites collect input data, while most of the processing is done in the cell body. Incoming signals can be of two types, the *excitatory signals* which make the neuron to fire or send an electrical impulse or the *inhibitory signals* which restrain the firing of the neurons.

Large quantities of input signals are typically received by most of the neurons through their *dendritic trees*. A single neuron can have innumerable numbers of dendrites and may receive many thousands of input signals. The decision for a neuron to be excited into firing an impulse or not, is dependent on the total of all the excitatory and inhibitory signals fed into it. The processing of this information happens in *soma* which is neuron cell body. When and if the neuron does fire, the nerve impulse, which is known as an *action potential*, travels along the axon.

The target ends receive the information via axon terminal, which are at the end of the axon. An axon splits up too many branches as it needs.

1.2 Artificial Neural Network

Artificial neural networks are the center of the decision-making mechanism in the modern computers. Comprehend unstructured data, make generalization and inferences with a greater accuracy are some of the basic functionalities that a neural network can achieve.

The basic unit of Artificial Neural Network (ANN) is known as *perceptron*. This is modeled based on the functionalities of biological neurons. Each artificial neuron (shown in Figure 3), has the similar functionalities. Following,

1. Takes inputs from the input layer
2. Weighs them separately and sum up
3. Pass the summation through a non-linear function to process the output

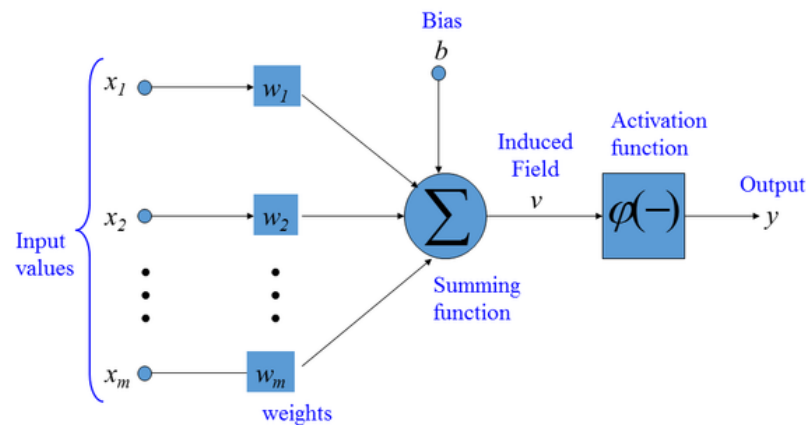


Figure 3: Artificial Neuron Model

In this method, there are inputs that are multiplied by weights, and these weight values direct the forward propagation. Backward propagation is used for fine-tuning the weight values of the layers.

The perceptron consists of 4 parts:

1. **Input Layer:** Acts like dendrites in biological neurons
2. **Weights and Bias:** Inputs multiplied with weights and add bias values
3. **Activation Function:** This decide whether to fire the neuron or not fire
4. **Output Layer:** Final output from the neuron passed to the next layer

Biological Neuron vs Artificial Neuron

Biological Neuron	Artificial Neuron
Dendrites	Input
Cell Body (Soma)	Node
Axon	Output
Synapse	Interconnection

Table 1: Similarities between Biological and Artificial Neuron

1.3 Neural Networks

A neural network is an interconnected network of neurons, as described in earlier sections. Neural networks are mainly used for classification tasks. The neural network architecture is represented in Figure 4.

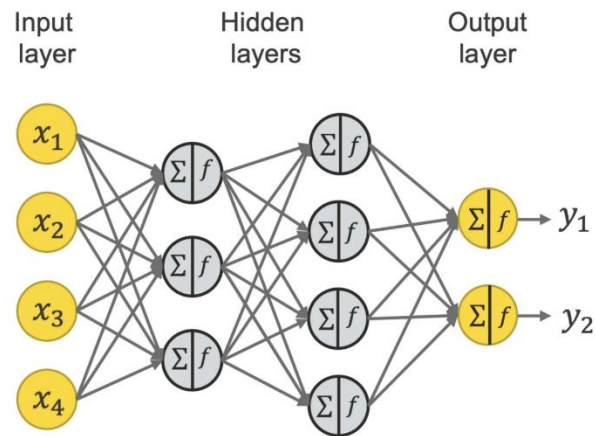


Figure 4: Structure of a Neural Network

As shown in the figure 4, it has a couple of layers that integrate artificial neurons, and through the combined contributions of all the neurons, it can perform multiclass classification.

1.3.1 Forword propagation

In forward propagation, the process begins by summing up all the weighted input values. The resulting sum is then non-linearly activated.

Let's denote the classes t using $+1$ and -1 . The discriminant function becomes $\mathbf{y} = \mathbf{W}^T \mathbf{x}$. For a pair of data (\mathbf{x}_i, t_i) , the condition $(t_i(\mathbf{W}^T \mathbf{x}_i) \geq 0)$ holds. A misclassification occurs when $t_i(\mathbf{W}^T \mathbf{x}_i) < 0$. To address this, we define a step activation function and aim to minimize misclassifications by optimizing the expression $\min \sum -t_i(\mathbf{W}^T \mathbf{x}_i)$. Gradient Descent can be used to find the optimal \mathbf{W} . However, to improve efficiency, the error calculation should only include misclassified samples.

1.3.2 Backward Propagation

As mentioned in the previous session, the weight parameters play a crucial role in the neural network. During the learning process, the neural network updates these weight values in a proper manner, which is accomplished through the process of backpropagation (shown in Figure 5).

The intelligence of neural networks is primarily developed in the following manner: First, the network guesses the classification based on the given input parameters (forward propagation) using randomly initialized weight values. This initial guess is typically incorrect because the weight values are random. The network then calculates the loss (Error), which represents how much the prediction deviates from the correct classification. In this research, we primarily used binary cross-entropy loss.

Next, using optimization techniques, the network recalculates the weight values in reverse order during backpropagation. Here, the network tries to adjust the weight values to find the minimum of the first derivative of the loss function with respect to the weight values.

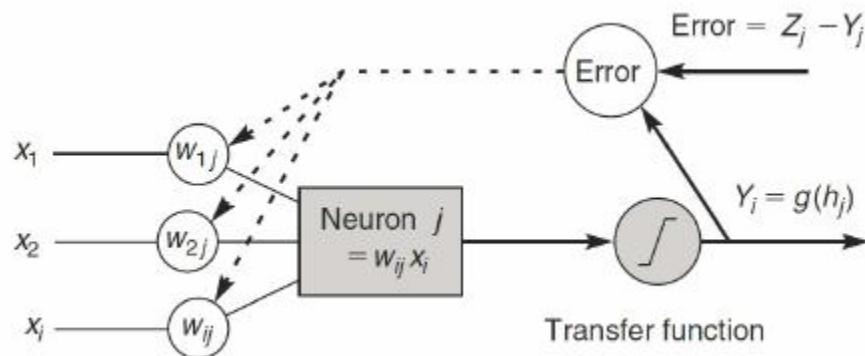


Figure 5: Backpropagation

Chapter 02

Spiking Neuron Architecture

2.1 Introduction

Spiking Neural Networks (SNNs) were created in computational neuroscience to simulate the behavior of organic neurons. As a result, the Leaky-Integrate-and-Fire (LIF) model was created, which describes neural activity as integrating incoming spikes with poor dispersion (leakage) into the environment.

Spiking Neural Networks lack a generic linear structure (shown in Figure 6). In this regard, it lacks a layer other than the input and output layers. Instead of crisp layers, they use more complicated structures such as loops or multi-directional connections to transmit data between neurons. Due of their complexity, they require different types of training and learning methods. To adapt to spike behavior, procedures like as back-propagation must be changed.

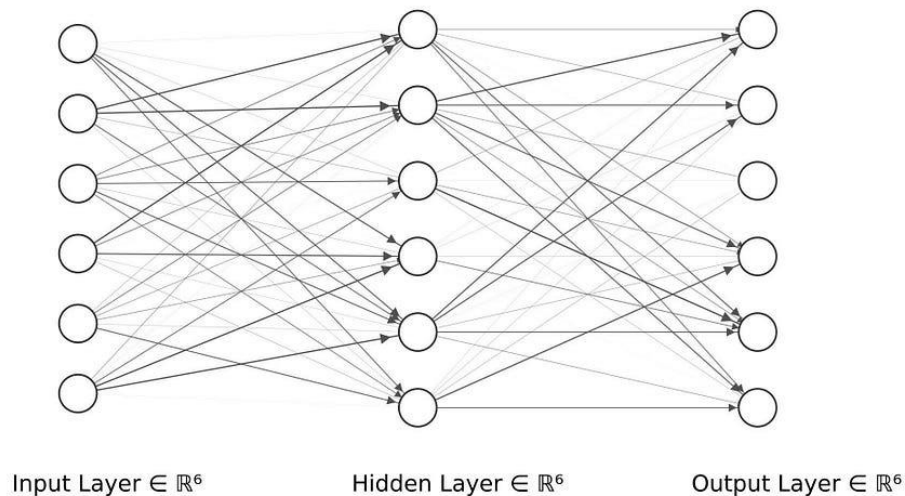


Figure 6: Typical Spiking Neural Network Architecture

Neuron Model

Each neuron in an SNN is represented by a mathematical model that mimics the activity of a real neuron. The most frequent model is the leaky integrate-and-fire (LIF) neuron model. It is made up of a membrane potential that stores incoming signals and a threshold that controls when the neuron fires a spike.

Here are some of the neuron models,

- Hindmarsh-Rose Neuron Model
- Fitzhugh-Nagumo Neuron Model
- Izhikevich Neuron Model
- Morris-Lecar Neuron Model

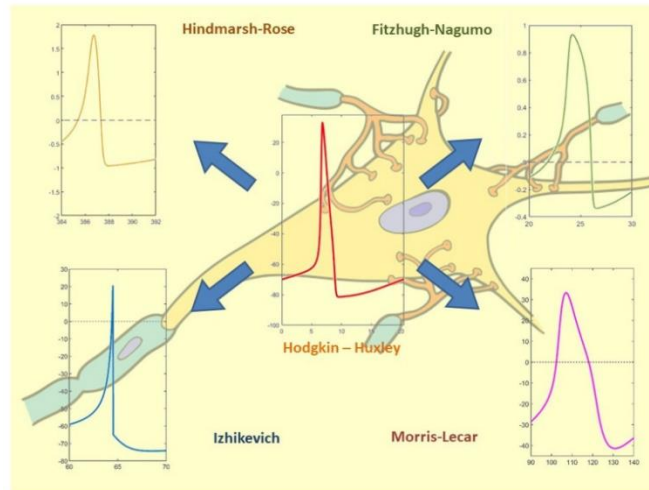


Figure 7: Spiking Neuron Models

Spike Generation

When the membrane potential approaches or surpasses a particular threshold, the neuron fires a spike or action potential. This spike indicates the neuron's output signal. After firing, the membrane potential is reset to its resting state, and a refractory period may be imposed, during which the neuron is briefly receptive to incoming inputs.

Spike Propagation

Neurons generate spikes, which are conveyed to neighboring neurons via weighted connections. The weight of a connection impacts the spike's effect on the receiving neuron's membrane potential. Typically, the weight is changed in accordance with learning criteria, such as spike-timing-dependent plasticity (STDP), which strengthens or weakens connections depending on the relative timing of pre- and postsynaptic events.

Spike Time Dependent Plasticity (STDP)

Spike-timing-dependent plasticity (STDP) is a critical learning mechanism inspired by the brain's synaptic plasticity. It adjusts the strength of synapses based on the relative timing of spikes from *pre-synaptic* and *post-synaptic* neurons. This temporal aspect of STDP makes it a powerful method for training Spiking Neural Networks (SNNs), a class of neural networks that mimic the behavior of biological neurons more closely than traditional artificial neural networks (ANNs).

The phenomenology of STDP is generally described as a biphasic exponentially decaying function (shown in Figure 8). That is, the instantaneous change in weights is given by,

$$\begin{aligned}\Delta W &= A_+ e^{(t_{pre}-t_{post})/\tau_+} && ; \text{if } t_{post} > t_{pre} \\ \Delta W &= -A_- e^{-(-t_{pre}-t_{post})/\tau_-} && ; \text{if } t_{post} < t_{pre}\end{aligned}$$

Where ΔW denotes the change in the synaptic weight, A_+ and A_- determine the maximum amount of synaptic modification (which occurs when the timing difference between presynaptic and postsynaptic spikes is close to zero), τ_+ and τ_- determine the ranges of pre-to-post synaptic interspike intervals over which synaptic strengthening or weakening occurs. Thus, $\Delta W > 0$ means that postsynaptic neuron spikes after the presynaptic neuron.

This model captures the phenomena that repeated occurrences of presynaptic spikes within a few milliseconds **before** postsynaptic action potential lead to long-term potentiation (LTP) of the synapse, whereas repeated occurrences of presynaptic spikes **after** the postsynaptic ones lead to long-term depression (LTD) of the same synapse.

The latency between presynaptic and postsynaptic spike (Δt) is defined as,

$$\Delta t = t_{pre} - t_{post}$$

Where t_{pre} and t_{post} are the timings of the presynaptic and postsynaptic spikes, respectively.

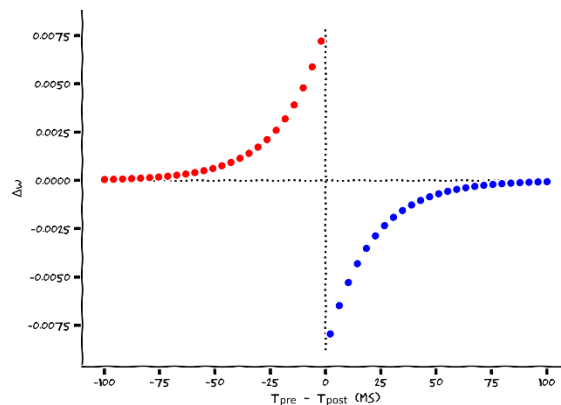


Figure 8: Biphasic STDP

2.2 Data Encoding Techniques

2.2.1 Introduction to Spike Encoding and Spike Encoding Methods

One of the main concepts in spiking neural networks (SNNs) is that all information must be encoded through spikes because SNNs aim to replicate how our brain processes it. Unlike standard neural networks that receive continuous signals, SNNs operate on spikes in short bursts of electrical activity. The shapes and time of these spikes are important, which is why data encoding into spikes has an essential role when it comes to the design of SNNs.

Spikes or action potentials are how neurons communicate within the human brain to relay information swiftly and effectively. Spike encoding is how in SNNs data comes to be represented and processed. Encoding plays a major role in how SNNs behave while doing tasks such as pattern recognition, processing sensory data, or controlling autonomous systems.

To create and train our SNNs, we will use the ‘*Snntorch*’ which is a PyTorch-based library. Snntorch allows for flexible experimentation with different spike encoding strategies to determine which method best suits our requirements.

For static datasets such as MNIST, which do not inherently change over time., the main strategies to them for SNNs are:

- Replicated Input Method: The simplest way to address this is by giving the same image multiple times at each time step t . While easy to realize, they do not leverage the temporal processing characteristics of SNNs.
- Spike Train Conversion: A more interesting approach is to convert the input image into a sequence of spikes in time, where every pixel can emit spikes based on its intensity. And so now the SNN can capitalize on its strength of dealing with temporal data, in a way more closely resembles to how biological neural networks work.

The *snntorch.spikegen* module can inherently support several methods for encoding spikes:

- Rate Coding (*spikegen.rate*): Information is represented by spikes which occur at a higher frequency when input values are high. Though this method is simple, it does not effectively utilize the temporal dynamics of SNNs.
- Latency Coding (*spikegen.latency*): Information is given by the mean firing rate at the onset of a stimulus. This works well, because using fewer spikes is energy efficient and processes faster.

- Delta Modulation (*spikegen.delta*): Then for every update time (e.g. delta), it reacts by spiking, its main function being to signal changes of the inputs over this interval and keep redundant information down to represent slow varying signals in a compressed way.

These encoding methods have unique characteristics, which determine their applicability to different types of problems. We will dig into each method in the next sections, comparing their performance based on snntorch.

2.2.2 Rate Coding

In shaping how learning will be affected, in this project the focus was on rate coding, one of the key strategies that are used to encode information through the frequency of neural spikes within spiking neural networks (SNNs). It represents data by changing the number of times neurons fire spikes per unit time. A stronger input to a neuron causes the latter to fire at a greater rate.

$$V = \frac{N_{spike}}{T}$$

Rate coding works on the premise that an input feature strength is represented by how frequently a neuron fires. Think of each input value as a probability for the neuron to spike in subsequent time steps. Just as with convolutional layers in neural networks, activation values of neurons can be interpreted visually. large weights mean high activation frequency ratios between pixels that the neuron is convoluting. On the other hand, low pixel values lead to fewer spikes The process is very like a coin flip where the probability of getting heads (a spike) is equally likely to its magnitude.

For example, when applied to images such as in the case of the MNIST dataset it allows us to change each pixel intensity into a spike rate. Whenever the pixel is bright, this causes a higher spike rate in the corresponding neuron; whereas a dark pixel and its related neuron hardly spike. This transformation allows SNNs to better process visual input (as shown in Figure 9).

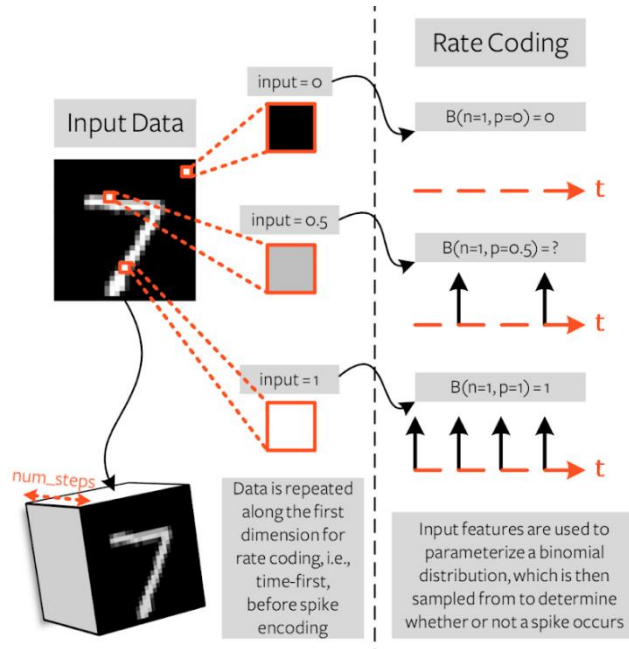


Figure 9: Rate Coding

Rate coding in SNNs works by translating continuous input data into spikes. The firing rate of each neuron is proportional to the value it receives as input. In a given interval, the level of activity in that neuron indicates how strong was the input by the number of spikes.

More acutely, if you have input image data that has highly varying pixel intensities, range encoding of these values may naturally lead to spikes in frequency. In the beginning, SNN also adapts some bio-processing, to reduce the background and Visual data for easy processing and analysis in computational systems.

There are three main types of Rate Coding:

- Count Rate Coding: This calculates the mean of number based on time.

$$V = \frac{N_{spike}}{T}$$

- Density Rate Coding: Averaging the spikes over multiple runs gives a graph of how frequently neurons fire. This is not how it works in nature but they can be useful, in artificial systems at least for evens out outcomes. (Average over several runs)

$$p(t) = \frac{1}{\Delta t} \frac{N_{spike}(t: t + \Delta t)}{K}$$

- **Rate Coding by Population:** this technique averages spikes produced in a set of neurons over time. It accounts for the combined activity of an entire population of neurons in response to input and enables encoding large amounts information via activities across many individual neurons.

$$A(t) = \frac{1}{\Delta t} \frac{N_{spike}(t: t + \Delta t)}{K}$$

Implementation and Advantages

Rate coding is a simple way to encode information in spiking neural networks (SNNs). Rate coding is the simplest mechanism to describe since it converts only raw input intensity into an aggregate firing rate which makes it so much easier for analysis and implementation. One of its nice features is that it does quite well in decay time considering Poisson noise averaging the rates out helps to smooth over individual spikes not firing as they should.

Rate coding is less efficient because it typically requires lots of spikes to transmit reliable information and so its proxy timing tends to be delayed compared with methods that rely on the precise timing of spikes. Moreover, rate coding is seen in some sensory systems but for other parts of the brain that have richer emergent behavior (emerge a higher-level function), it could be more efficient or faster processing than methods used by older regions.

With the MNIST dataset, using rate coding scored 29% accuracy in my experiments. Rate coding you can see, is pretty handy but ultimately limited in accurately translating pixel data to predictions.

```
Epoch 1, Loss: 1.9505195617675781
Epoch 2, Loss: 1.846865177154541
Epoch 3, Loss: 1.9342751502990723
Epoch 4, Loss: 1.8251029253005981
Test Accuracy: 29.29%
```

Figure 10: Accuracy in Rate Coding

Applications and Implications

This is perfect if you need something that simple and reliable (as in rate coding). It is particularly effective for encoding sensory information and is often used in the early layers of SNNs. However, it has its downfalls.

Rate coding is open to more concern. Although it can be seen that rate coding is present in the peripheral sensory systems, whether this represents the primary method of encoding in more difficult brain regions remains unclear. Consider the following points:

- **Power Consumption:** Some things might be power-hungry to accomplish with too frequent spikes. A limited number of studies suggest that rate coding may account for relatively little neuron activity in some brain areas, implying instead that the brain conserves energy through other mechanisms.
- **Reaction Times:** Although human reaction times are approximately 250ms. But as a neuron only fires every ~100 milliseconds (10Hz) this greatly limits the amount of action potentials that can be processed each time. This implies that rate coding may not be the optimal way of information processing.

However, despite these challenges, we are convinced that rate coding is useful. This model is substantially robust due to its noise resistance and continuous spiking can facilitate learning (neurons that fire together, and wire together). Rate coding can be applied to increase firing rates in cases where training SNNs is challenging.

2.2.3 Latency Coding

This is more complex form of encoding used in spiking neural networks (SNNs) where information is transferred based on the precise timing of spikes. Rate coding uses the frequency of spikes to encode information, whereas latency coding refers to the exact moment a spike occurs.

Concept of Latency Coding

If so, we can exploit the fact that latency coding has to do with when a neuron spikes. This method namely makes use of the timing information of single spikes. After all, if a neuron spikes after just having been fired upon with an input signal from another cell (very shortly thereafter), that is because it says the input is "ON". If the spike is delayed, this means a weaker or less urgent signal.

The beauty of latency coding lies in its widespread capacity for encoding precise temporal details. However, for tasks requiring exact timing, latency coding can give more information than with rate codes.

The spikegen can also be used to do latency coding. Tools such as snntorch often include a latency function. In this method, every spike has a timing that corresponds to the intensity of each feature used as an input.

- Early Spiking of Bright Pixels: Image datasets such as MNIST inherently contain bright pixels.
- Two pixels later, the presence of a dark pixel fires.

This is similar to the way a capacitor charges in an RC circuit. An easier way to understand it is:

- 1) **Current Injected:** The input feature is interpreted as a current source injected into an RC model. This current slowly charges a capacitor.
- 2) **Trigger Voltage:** A spike is created when the capacitor reaches a certain trigger voltage.
- 3) **Time of Spike:** the capacitor takes time to charge and reach a trigger voltage that is directly proportional to the input feature value. So high values charge faster and end up having spikes sooner also when low-value input comes in we get different events later than with the higher one.

The RC Model

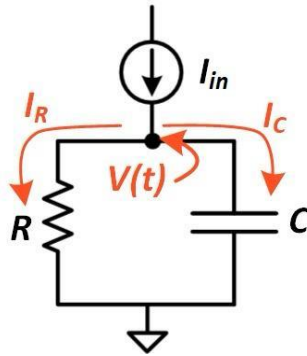


Figure 11: RC circuit model of the Latency Coding

$$\begin{aligned}
 I_{in} &= I_r + I_c \\
 I_{in} &= \frac{V(t)}{R} + C \frac{dV(t)}{dt} \\
 I_{in}R &= V(t) + RC \frac{dV(t)}{dt} \\
 V(t) &= I_{in}R + C e^{-t/RC} \quad ; \text{ where } t = 0, V(t) = 0 \\
 C &= -I_{in}R \\
 V(t) &= I_{in}R[1 - C e^{-t/RC}]
 \end{aligned}$$

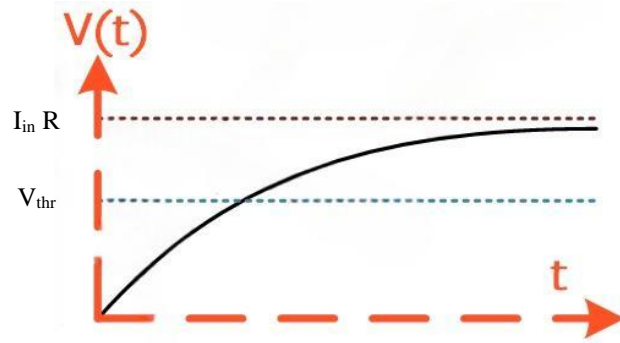


Figure 12: RC model - Voltage variation along with time

The spike is emitted when $V(t)$ reaches the threshold V_{thr} . So,

$$t = RC \left[\ln \left(\frac{I_{in} R}{I_{in} R - V_{thr}} \right) \right]$$

For simplicity,

$$t = \tau \left[\ln \left(\frac{I_{in} R}{I_{in} R - V_{thr}} \right) \right]$$

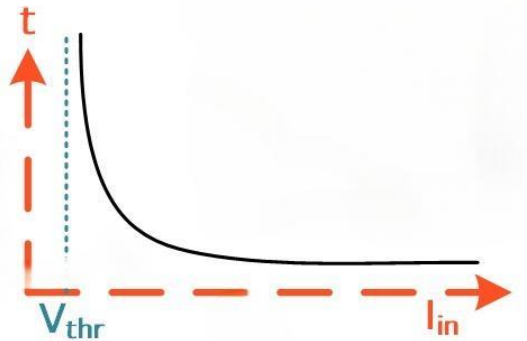


Figure 13: Time vs Input current

This recursion can be described mathematically but essentially it says that big inputs will cause faster spikes and small inputs slower spikes (as shown in Figure 13).

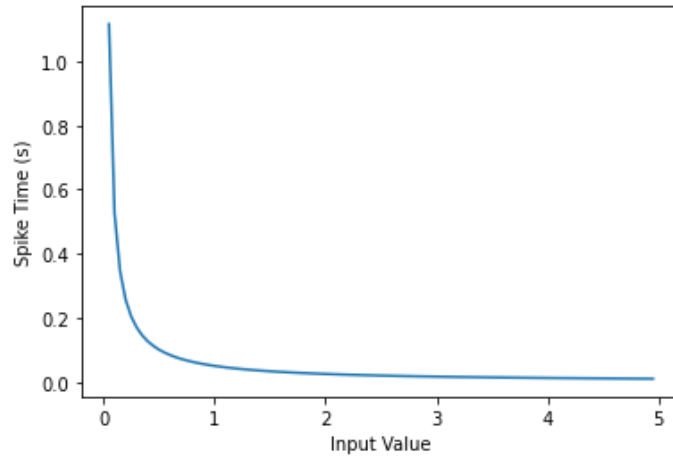


Figure 14: Spike time vs Input value

Advantages and Disadvantages

Advantages:

- Latency coding is very energy efficient. A SNN is power efficient because it performs encoding of information using precise spike timings instead of utilizing multiple spikes to encode the same data. Meaning, you need fewer spikes and thus consume less computational resources.
- High Temporal Resolution: This method has a very fine time resolution, which is necessary for tasks that need detailed temporal encoding. However, it is also a framework that permits advanced and accurate capturing of time-evolving information which makes this toolbox ideal for applications such as auditory processing where the precise timing of sound signals matters.

Disadvantages:

- Sensitivity to Noise: As latency coding is based on spike timing, tiny variations in the precise time of firing can have a dramatic impact on emissions. This makes it more prone to noise which can destroy the data and cause an error.

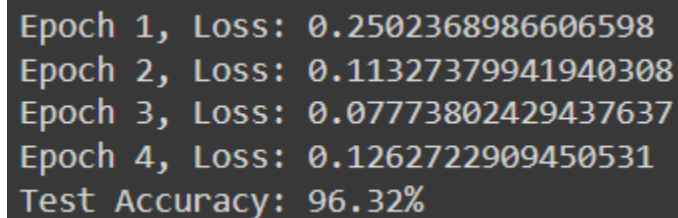
Applications and Implications

This can be very useful for applications where the time dimension is a significant factor while using other *coding. Thus, it is perfect for fast-running auditory processing tasks that require accurate timing of sound signals or rapid sequence visual pattern detection with high temporal rate. In operations like these, latency coding can give you a more precise read on what the data means.

Although latency coding has not been widely used because it is complex to implement, the combination of encoding elaborate timing information with low power consumption makes this technique attractive for some applications. However, the typical noise sensitivity and requirement for high precision mean that it is commonly used in combination with other coding methods to increase its performance or robustness.

While in practice, latency coding means a conversion of input features to spike timings such that the SNN can harness this accurate temporal data. This is especially beneficial for real-time, energy-efficient processing in robotic systems and incarnations the task of time-sensitive computations etc.

For use cases for my implementation of latency coding, I tested it on the simple MNIST dataset and saw precision rise to unprecedented levels when a whopping 96% accuracy was reached. This finding underscores how latency coding has the capacity to improve performance in cases where absolute timing is a crucial resource, suggesting applicability to high-precision tasks.



```
Epoch 1, Loss: 0.2502368986606598  
Epoch 2, Loss: 0.11327379941940308  
Epoch 3, Loss: 0.07773802429437637  
Epoch 4, Loss: 0.1262722909450531  
Test Accuracy: 96.32%
```

Figure 15: Accuracy in Latency Coding

2.2.4 Delta Modulation

Spike encoding refers to delta modulation which is specifically developed to represent changes in the input signals well by creating spikes as a function of differences between two consecutive values. This approach simulates the event-driven nature of biological neurons, which are more likely to fire when a stimulus changes rather than for constant stimuli. There is also delta modulation, which encodes "shifts" or "deltas" of data to follow fast-changing signals.

The fundamental idea of delta modulation is based on how some biological systems like the retina process information only when there occurs a change in input nodes. If you stare long enough at a single, inanimate scene over time there is an effective input-decay that occurs with your photoreceptor cells. The hereditary mechanism ensures that to changes or events, rather than continuous input, neurons respond with the highest sensitivity, which in turn makes them very low-power devices and trainable by the environment.

Delta modulation in artificial neural networks has been demonstrated to work on an event-driven basis. It represents input changes by emitting a spike whenever the change between two subsequent inputs exceeds some threshold. This has application in time-series or dynamic inputs where we are more home on detecting and encoding changes rather than the absolute values.

Delta modulation is the process of checking the variation between the consecutive data points of a time-series input. The *snntorch.delta()* function, for instance, evaluates all the potential differences among successively characteristics, or features, across all time buckets of a time-series tensors. If this dissimilarity is positive and greater than a given threshold, a spike is initiated.

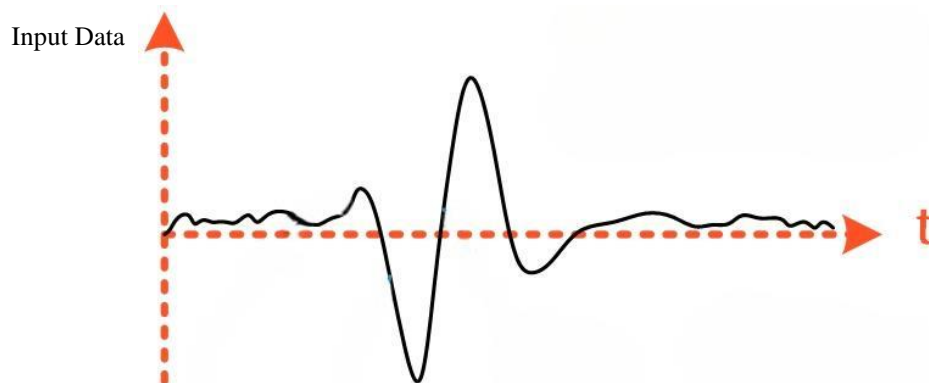


Figure 16: Input Data to Delta Modulation

Data Preparation: Beginning with a time-series tensor indicating some signal.

Delta Conversion: It is the tensor you will pass into the `spikegen.` function on a predetermined threshold. The function will now cast the data by rising spikes provided each consecutive value is greater than a given threshold. This would create spikes in the above operation for 'meaningful changes' and ignore all those minor fluctuations (as shown in Figure 17).



Figure 17: Spike generation with time

Sensitivity adjustment: Sensitivity can adjust by tweaking the threshold value. The more the system senses, a higher such value will impede triggering greater no of spikes for smaller fluctuations.

Processing Negative Changes: It is Intriguing that delta function has a parameter called off spike, which can capture negative change by producing “off-spikes”. This is handy for when you care about both an increase and a decrease of the signal. It would create further spikes for large negative changes, offering a better set of direct signal dynamics (as shown in Figure 18).

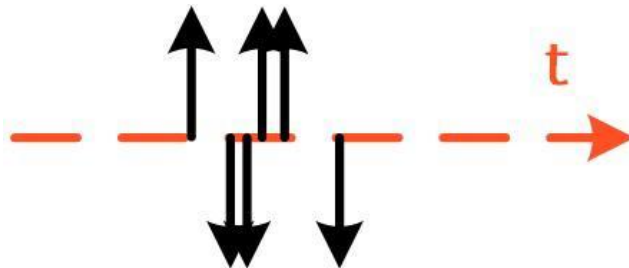


Figure 18: Delta modulated spikes

Advantages and Disadvantages

Advantages:

- **Efficient Event-Driven:** Delta modulation borrows its mechanism from biological neurons, being implemented on event-driven planes which makes it very efficient in encoding time-varying signals. This solution can obtain even up to 7 times of energy saving in neuromorphic hardware, as it reduces massively the number of spikes fired at stable inputs
- **Lesser Data Redundancy:** Delta modulation generates spikes only when there are significant changes in input, so it reduces redundancy present in data. This improves computational efficiency, avoiding to process or store information that is not needed.

Disadvantages:

- **Noise Sensitivity:** Delta modulation depends on finding differences, so it is especially susceptible to noise. Small noise in the input can cause these spikes if we do not set the threshold with extra care and this might give wrong results
- **Poor Static Information Representation:** Delta modulation cannot represent the signals that are not changing over time in an effective manner. For a complete solution that would require both dynamic information and static, this limitation might not be that effective.

Applications and Discussion

Delta modulation is ideal for use in applications where the detection of changes takes precedence, e.g. event-based sensors and dynamic vision systems. Such systems, which include the Dynamic Vision Sensor (DVS), take advantage of delta modulation to directly and quickly address visual input by concentrating on changes as opposed to static scenes. This results in the same low-latency, energy-efficient operation that is typical of a biological vision system with adaptive, event-driven behavior.

Delta modulation is used alone in spiking neural networks (SNNs), but rather with other encoding techniques to keep strengths and weaknesses at bay. It is the perfect solution for a recorded signal that has variations in time but it can also be combined with (rate coding) or replaced by rate-coding and latency-based approaches to tackle not only stimuli of stasis nature but low fidelity measurements.

This technique was useful for encoding pixel intensity changes over time, especially in images with moving patterns when experimented with delta modulation on the MNIST dataset. The higher accuracy of 86.35% (as shown in Figure 19) that was achieved in these tests demonstrates the potential delta modulation has to efficiently capture relevant information without sacrificing sensitivity or specificity. The main problem with this method is its fine-tuning parameters to have the best results on different kinds of data: threshold, and least count size.

```
targets shape: torch.Size([128])
outputs shape: torch.Size([100, 128, 10])
targets shape: torch.Size([128])
outputs shape: torch.Size([100, 16, 10])
targets shape: torch.Size([16])
Test Accuracy: 86.35%
```

Figure 19:Accuracy using Delta Modulation

2.3 Leaky-integrate and Fire Neuron

Basically, the leaky integrate-and-fire model works by instantaneously increasing its potential as soon as it receives an input spike, and then it decays exponentially until it reaches the threshold value. This works as shown in the following Figure 20, the membrane potential is reset after reaching the threshold value, and then it outputs a spike.

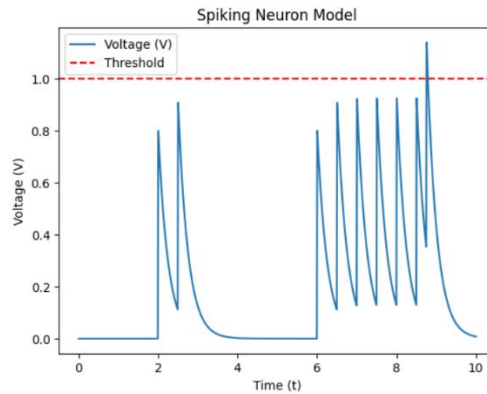


Figure 20: membrane potential characteristics of LIF neuron

Due to this characteristic of the LIF neuron, we have modeled it using the following RC circuit.

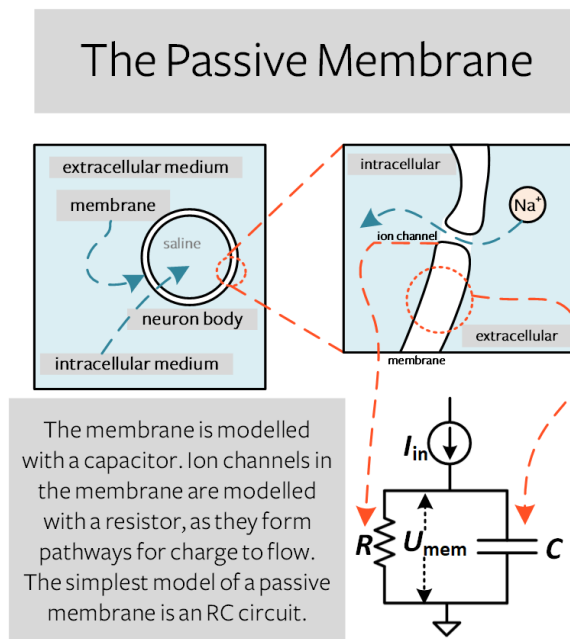


Figure 21: RC Model

2.4 Mathematical Derivation of LIF

When a time-varying current $I_{in}(t)$, injected into a neuron either through electrical stimulation or from other neurons. The total current in the circuit must be conserved.

This can be expressed as:

$$I_{in}(t) = I_R + I_C$$

Here, I_R represents the current through the resistor, and (I_C) represents the capacitive current. According to Ohm's Law, the membrane potential (U_{mem}), which is the voltage measured across the resistor, is proportional to the current through the resistor:

$$I_R(t) = \frac{U_{mem}(t)}{R}$$

The capacitance (C) relates the charge stored on the capacitor (Q) to the membrane potential ($U_{mem}(t)$) by:

$$Q = C \cdot U_{mem}(t)$$

The rate of change of the charge on the capacitor gives us the capacitive current:

$$\frac{dQ}{dt} = I_C(t) = C \cdot \frac{dU_{mem}(t)}{dt}$$

Substituting this into the conservation equation,

$$I_{in}(t) = \frac{U_{mem}(t)}{R} + C \cdot \frac{dU_{mem}(t)}{dt}$$

Rearranging terms and obtain,

$$R \cdot C \cdot \frac{dU_{mem}(t)}{dt} = -U_{mem}(t) + R \cdot I_{in}(t)$$

Here, the right-hand side of the equation has units of voltage. To match the units on both sides of the equation, we note that the term $(\frac{dU_{mem}(t)}{dt})$ on the left-hand side has units of voltage per time. Thus, $(R \cdot C)$ must have units of time. We refer to $\tau = R \cdot C$ as the time constant of the circuit:

$$\tau \cdot \frac{dU_{mem}(t)}{dt} = -U_{mem}(t) + R \cdot I_{in}(t)$$

Above equation describes how the passive membrane of the neuron responds to inputs over time. The solution to this linear differential equation reveals that the membrane potential decays exponentially with the time constant (τ).

Assuming the neuron starts with an initial membrane potential U_0 and no further input ($I_{in}(t) = 0$), the solution to the differential equation is:

$$U_{mem}(t) = U_0 \cdot e^{-\frac{t}{\tau}}$$

This expression shows that the membrane potential decreases exponentially from its initial value (U_0) with a time constant (τ).

Now say some arbitrary time-varying current $I_{in}(t)$ is injected into the neuron, be it via electrical stimulation or from other neurons. The total current in the circuit is conserved

Computational implementation of LIF

The current source is related to the inputs of the neuron, and the membrane potential is equivalent to the voltage difference across the $1k$ resistor. The Leaky Integrate-and-Fire (LIF) neuron model is used in the forward propagation equation. To evaluate this computationally we have used Forward Euler method and the representation of the differential equation is given in (Eq 2.1).

$$\tau \frac{dU(t)}{dt} = -U(t) + RI_{in}(t)$$

For simplicity, the subscript of ($U(t)$) is omitted.

First, let's solve this derivative without taking the limit ($\Delta t \rightarrow 0$):

$$\tau \frac{U(t + \Delta t) - U(t)}{\Delta t} = -U(t) + RI_{in}(t)$$

For a small enough (Δt), this provides a good approximation of continuous-time integration. Isolating the membrane potential at the following time step gives:

$$U(t + \Delta t) = U(t) + \frac{\Delta t}{\tau} (-U(t) + RI_{in}(t))$$

$$U[t + 1] = \beta U[t] + Wx[t + 1] - r[t]$$

(Eq 2.1)

Where β is the decay constant, W is the synaptic weight, $x[t + 1]$ is the input at time $t+1$, and $r[t]$ is the rest term. Due to the LIF neuron model, the output shows a Heaviside step transfer function because the output is 1 when the membrane potential is larger than the threshold value and 0 otherwise (shown in Figure 22 and Figure 23).

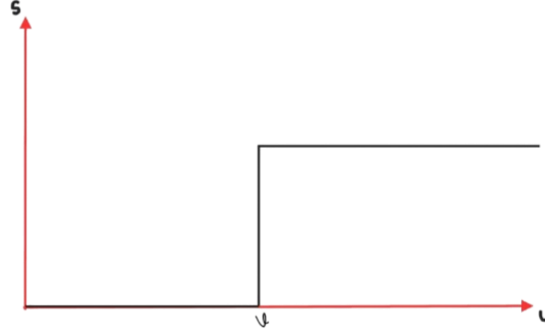


Figure 22: Unit Step Function

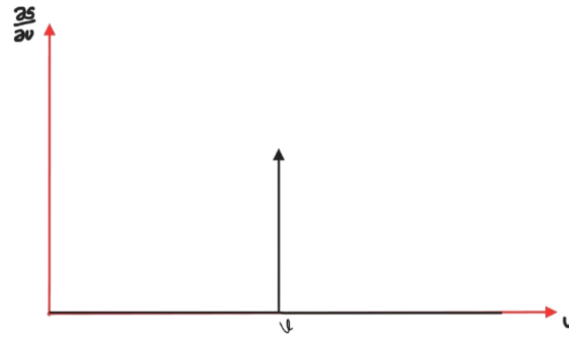


Figure 23: Derivative of Unit Step Function

Now, a new problem arises due to the first derivative of the step function, which is the impulse function. This is a non-derivable discrete function, and this issue adversely affects the training process because optimization techniques require the derivative $\frac{ds}{du}$, which tends to infinity in this case.

Chapter 3

Mathematical Approximation of Unit Step Function

3.1 Sigmoid Approximation

Firstly, when considering the dead neuron problem, we found a solution by approximating the unit step function using a sigmoid approximation. The sigmoid function provides a smooth approximation of the step function. The function is given by (Eq. 3.1), and its graphical representation is shown in Figure 24.

$$f(x) = \frac{1}{1+e^{-x}} \quad (\text{Eq. 3.1})$$

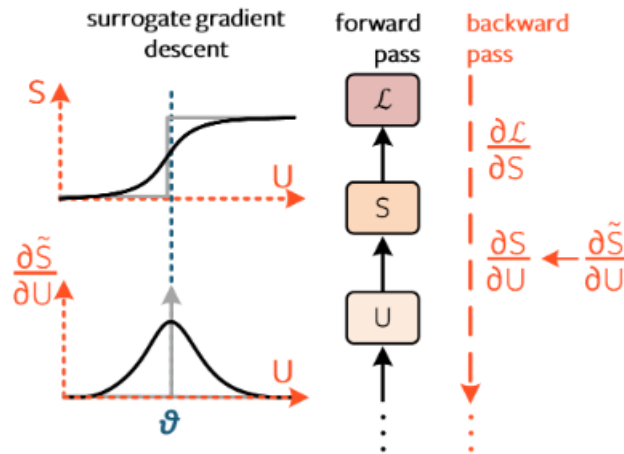


Figure 24: Unit step approximation and Derivative of the approximation

However, due to the exponential calculations, the computational cost was high. Therefore, when the training processes the throughput is lower than the artificial neural network but the accuracy was good.

Even though, the throughput is reduced, but the accuracy is comparable to that of the sigmoid function. It has demonstrated great accuracy, as shown in the following figures.


```

Iteration [450], Throughput: 881.21 samples/s
Epoch [1/1], Step [451], Train Loss: 15.9240, Test Loss: 15.0614
Iteration [451], Throughput: 841.21 samples/s
Iteration [452], Throughput: 882.64 samples/s
Iteration [453], Throughput: 938.82 samples/s
Iteration [454], Throughput: 914.07 samples/s
Iteration [455], Throughput: 890.25 samples/s
Iteration [456], Throughput: 911.79 samples/s
Iteration [457], Throughput: 809.13 samples/s
Iteration [458], Throughput: 892.61 samples/s
Iteration [459], Throughput: 914.20 samples/s
Iteration [460], Throughput: 893.01 samples/s
Iteration [461], Throughput: 906.74 samples/s
Iteration [462], Throughput: 882.08 samples/s
Iteration [463], Throughput: 848.50 samples/s
Iteration [464], Throughput: 932.10 samples/s
Iteration [465], Throughput: 941.26 samples/s
Iteration [466], Throughput: 906.44 samples/s
Iteration [467], Throughput: 921.56 samples/s
Iteration [468], Throughput: 861.33 samples/s
Epoch [1/1], Throughput: 646.92 samples/s Train Loss: 15.1298, Test Loss: 14.5822

```

Figure 26: Throughput, Train loss and Test loss

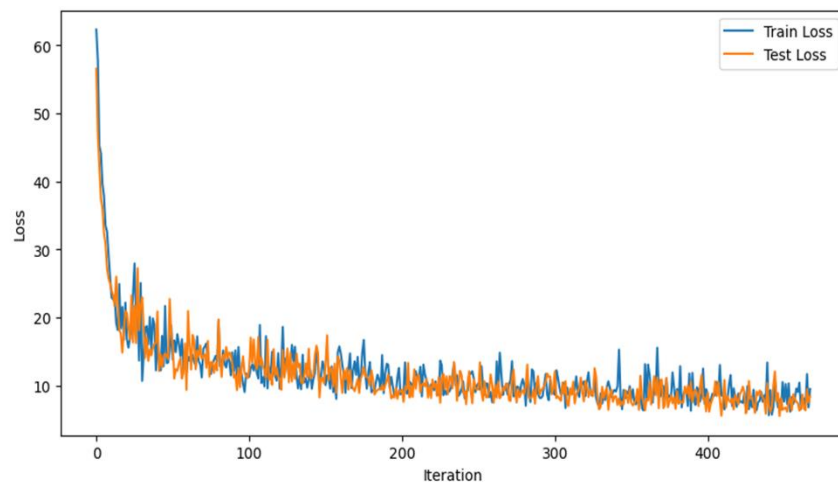


Figure 25: Loss Curves

After the training process, inferencing was performed using 10% of the dataset.

```

Total correctly classified test set images: 9272/10000
Test Set Accuracy: 92.72%

```

Figure 27: Accuracy of Sigmoid Approximation on test data set

3.2 Chebyshev Approximation

Then, to find a more computationally efficient method, we used Chebyshev polynomial approximations.

Chebyshev approximations did a great job when considering the approximation of the unit step function, but a new problem arose regarding the ripple effect of the Chebyshev approximations. Due to these ripple effects, the calculation of the derivative of the unit step function was negatively impacted. As a result, the ripple effect in the derivative led to lower performance in the optimization function. However, in terms of power consumption and computational complexity, their performance was better.

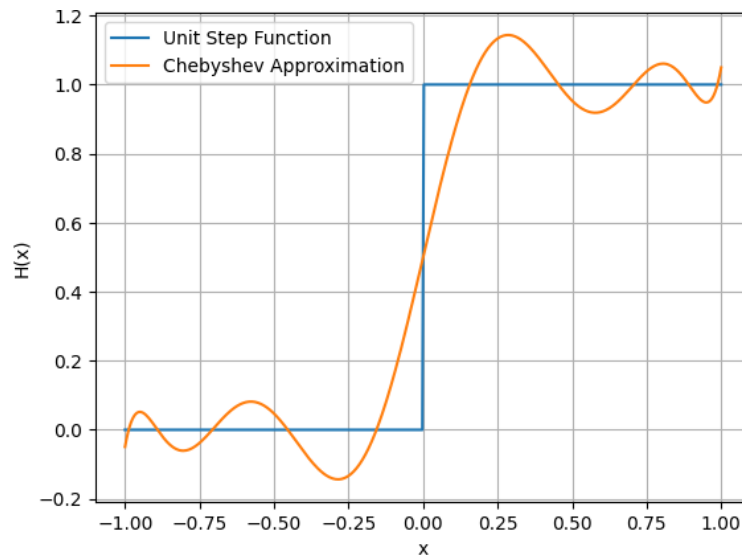


Figure 28: Chebyshev Approximation of the Unit Step Function

Due to the results of throughput, the Chebyshev approximation performs better than the sigmoid function. However, when considering accuracy, the performance drops.

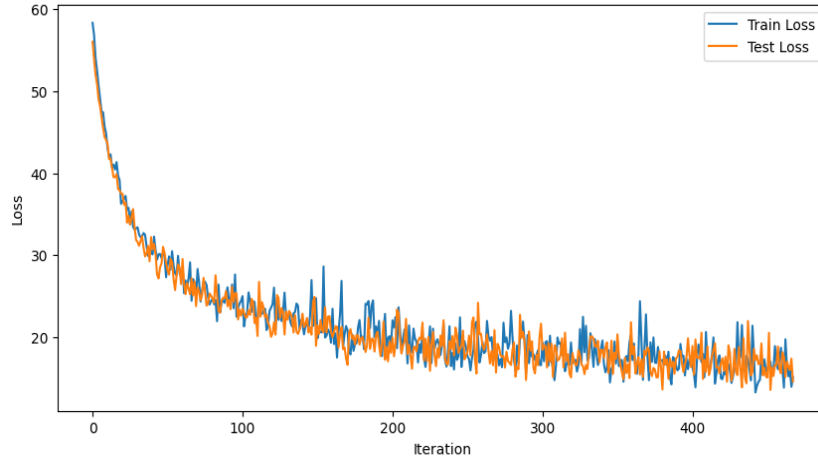


Figure 30: Loss Curves in Chebyshev Approximation

Iteration [451]	Latency: 0.0651s,	Throughput: 1968.33 samples/s
Iteration [452]	Latency: 0.0612s,	Throughput: 2090.80 samples/s
Iteration [453]	Latency: 0.0817s,	Throughput: 1566.96 samples/s
Iteration [454]	Latency: 0.0627s,	Throughput: 2041.75 samples/s
Iteration [455]	Latency: 0.0679s,	Throughput: 1884.99 samples/s
Iteration [456]	Latency: 0.0673s,	Throughput: 1901.32 samples/s
Iteration [457]	Latency: 0.0749s,	Throughput: 1708.41 samples/s
Iteration [458]	Latency: 0.0629s,	Throughput: 2034.20 samples/s
Iteration [459]	Latency: 0.0648s,	Throughput: 1974.93 samples/s
Iteration [460]	Latency: 0.0633s,	Throughput: 2023.03 samples/s
Iteration [461]	Latency: 0.0661s,	Throughput: 1935.47 samples/s
Iteration [462]	Latency: 0.0621s,	Throughput: 2061.58 samples/s
Iteration [463]	Latency: 0.0648s,	Throughput: 1976.60 samples/s
Iteration [464]	Latency: 0.0618s,	Throughput: 2072.61 samples/s
Iteration [465]	Latency: 0.0634s,	Throughput: 2018.66 samples/s
Iteration [466]	Latency: 0.0657s,	Throughput: 1947.64 samples/s
Iteration [467]	Latency: 0.0636s,	Throughput: 2011.75 samples/s
Iteration [468]	Latency: 0.0738s,	Throughput: 1735.55 samples/s
Epoch [1/1]	Latency: 48.1897s,	Throughput: 1245.08 samples/s

Figure 29: Latency and Throughput in Chebyshev Approximation

Figure 31 shows the inferencing results of SNN.

```
Total correctly classified test set images: 9181/10000
Test Set Accuracy: 91.81%
```

Figure 31: Inferencing Results

3.3 Piecewise Linear Gradient Approximation

After observing the impact of the ripple effect in Chebyshev functions, we realized that instead of approximating the step function, it would be more effective to approximate the derivative of the step function, which is the impulse function. Approximating the unit step function requires the approximation to be differentiable, but by focusing on the impulse function, we eliminate the need for the function to be differentiable or continuous. Therefore, we have implemented a piecewise linear approximation for the unit impulse function. (Eq. 3.2 shows the mathematical representation of the piecewise linear approximation for the unit impulse function.

$$f(x) = \begin{cases} 0 & ; if \ x < -0.5 \\ x + 1 & ; if \ -0.5 \leq x < 0 \\ 1 - x & ; if \ 0 \leq x < 0.5 \\ 0 & ; if \ 0.5 \leq x \end{cases} \quad (Eq. 3.2)$$

The graphical representation of the unit step function is shown in **Figure 29**. Here, it clearly illustrates that the approximation of the unit step function is a triangular function, which is more suitable for representing an impulse than sigmoid and Chebyshev approximations.

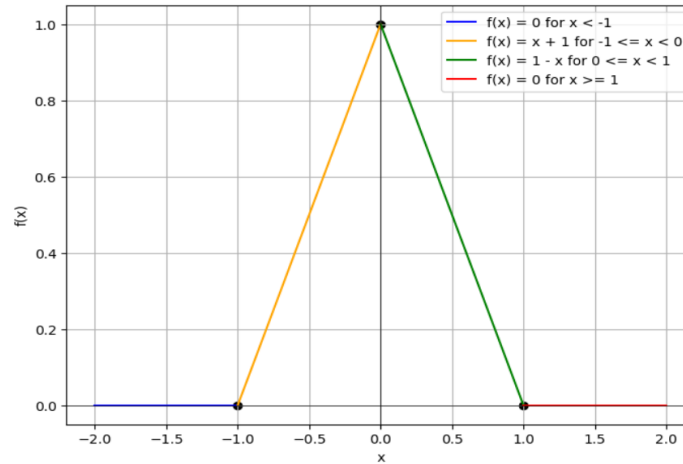


Figure 32: Piecewise Linear Function

Then, using the piecewise linear gradient approximation function, we re-evaluated our neural network model and trained it on the MNIST dataset as done with the previous approximations. The loss data is shown in Figure 33, and the throughput data is presented in Figure 34.

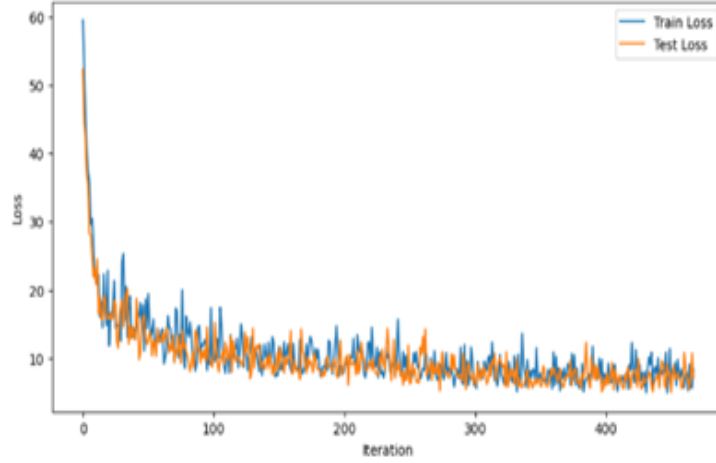


Figure 33: Loss Curves in Piecewise Linear Approximation

```
Epoch [1/1], Step [1], Train Loss: 60.6571, Test Loss: 54.5423
Iteration [1] Throughput: 2027.98 samples/s
Epoch [1/1], Step [51], Train Loss: 14.9203, Test Loss: 13.4249
Iteration [51] Throughput: 2063.12 samples/s
Epoch [1/1], Step [101], Train Loss: 14.0429, Test Loss: 17.1441
Iteration [101] Throughput: 2038.69 samples/s
Epoch [1/1], Step [151], Train Loss: 10.4271, Test Loss: 11.4463
Iteration [151] Throughput: 2015.89 samples/s
Epoch [1/1], Step [201], Train Loss: 7.4426, Test Loss: 8.5950
Iteration [201] Throughput: 2014.58 samples/s
Epoch [1/1], Step [251], Train Loss: 9.2920, Test Loss: 10.0605
Iteration [251] Throughput: 2033.55 samples/s
Epoch [1/1], Step [301], Train Loss: 11.5108, Test Loss: 8.6424
Iteration [301] Throughput: 2039.80 samples/s
Epoch [1/1], Step [351], Train Loss: 6.9007, Test Loss: 9.3507
Iteration [351] Throughput: 2023.32 samples/s
Epoch [1/1], Step [401], Train Loss: 11.3627, Test Loss: 8.9569
Iteration [401] Throughput: 1781.49 samples/s
Epoch [1/1], Step [451], Train Loss: 9.9615, Test Loss: 6.0782
Iteration [451] Throughput: 2032.84 samples/s
```

Figure 34: Throughput and Latency in Piecewise Linear Approximation

Chapter 4

Hardware Implementation of the SNN

After all the software implementations, we needed to measure the power consumption of the SNN model during inference. We used the Vivado platform to evaluate the power calculations and timing diagrams of our Verilog code. We also inferred the MNIST data. Figure 35 shows the power calculation results obtained from the Vivado and VCS simulations. Here, we plugged the weights trained using the piecewise approximation into the hardware code and ran it.

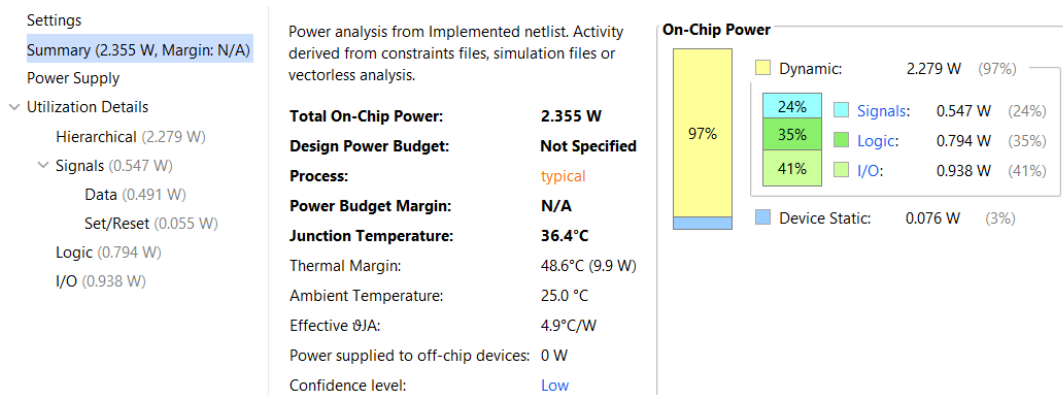


Figure 35: Power Calculation of Inferencing of SNN

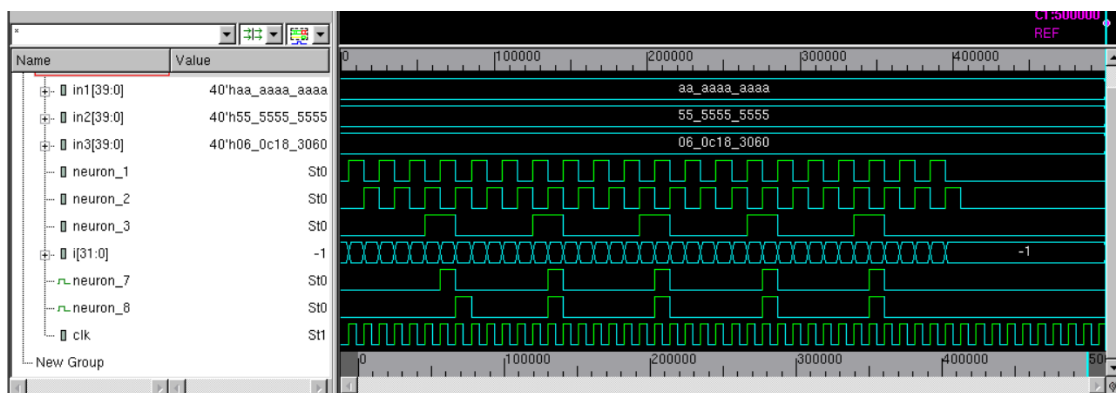


Figure 36: Timing Diagram of Inferencing in SNN

Power Analysis

The power analysis was performed using *Vivado*, which gave us an insight into the detailed on-chip power consumption. From Figure X, we observe the total on-chip power consumed by our SNN implementation is 2.355 W. Dynamic power, which comprises 97% of the total power, is concentrated among signals, logic, and I/O, respectively, with power consumptions of 0.547 W (24%), 0.794 W (35%), and 0.938 W (41%).

- **Signals:** By and large, in SNNs, 24% of the dynamic power is utilized by the signals, which, in principle, shows the energy usage in transferring data between the different components.
- **Logic:** The logic elements are performing the execution of network operations and dissipate 35% of dynamic power, indicating the computational overhead of the SNN.
- **I/O:** Input/output operations form the very basic level of communication with external devices and thus take the lion's share at 41%.

Our SNN has some advantages in comparison to its general neural network hardware implementation counterpart. Since SNNs are event-driven, they consume lower power due to fewer spikes (events) against continuous processing in GNNs, especially in the logic and I/O part.

Conclusion

This project investigated how to improve information processing in spiking neural networks (SNNs) by accelerating unit step function approximations. We succeeded in identifying the trade-offs between computational efficiency and accuracy by comparing the effectiveness of Chebyshev and Piecewise Linear Gradient Approximations (PLGA). While Chebyshev approximations reduced computational costs, the ripple effects had a negative impact on optimization. In contrast, the PLGA approach demonstrated more consistent performance, making it a better alternative for tackling the dead neuron problem in backpropagation. Furthermore, by investigating spiking neuron designs and spike encoding methodologies, our research helps to advance neuromorphic computing, which requires energy-efficient and physiologically inspired models. Our findings highlight the promise of SNNs in practical applications, paving the door for further research into optimizing spike-based information processing systems.

References

- [1] Fortuna, L. and Buscarino, A. (2023) ‘Spiking neuron mathematical models: A compact overview’, *Bioengineering*, 10(2), p. 174. doi:10.3390/bioengineering10020174.
- [2] NeuroCortex.AI (2023) *Spiking neural network architectures*, *Medium*. Available at: <https://medium.com/@theagipodcast/spiking-neural-network-architectures-e6983ff481c2>
- [3] *Piecewise linear approximation: Piecewise linear approximation - Cornell University Computational Optimization Open Textbook - Optimization Wiki*. Available at: https://optimization.cbe.cornell.edu/index.php?title=Piecewise_linear_approximation (
- [4] *SNNTORCH documentation: snnTorch Documentation - snntorch 0.9.1 documentation*. Available at: <https://snntorch.readthedocs.io/en/latest/> (Accessed: 20 July 2024).