Evaluation of a Spiking-Neural Networks, as a Bio-Inspired Machine Learning Model

KAUSHIK DAS

A dissertation submitted in partial fulfilment
of the requirements for the degree of
Master of Science in Artificial Intelligence
of the
University of Aberdeen.

Department of Computing Science

2024

Declaration

No portion of the work contained in this document has been submitted in support of an application for a degree or qualification of this or any other university or other institution of learning. All verbatim extracts have been distinguished by quotation marks, and all sources of information have been specifically acknowledged.

Signed:

Date: 2024

Abstract

In this project, I have created a Spiking Neural Network (SNN) model specifically for image classification tasks, with a focus on leveraging spiking neuron layers to exploit temporal dynamics for enhanced pattern recognition. The key components of the model is a spiking neuron layer module, input encoder, and output decoder. The spiking neuron layer module will simulate neuron spiking behavior, incorporating membrane capacity and spike generation based on threshold mechanisms. Additionally, I will develop an input encoder to convert raw image data into spike sequences suitable for processing by the SNN, along with an output decoder to utilise the spike patterns into a format comparable to traditional neural network outputs. A comprehensive analysis will be conducted on the complexities involved in SNN models, including challenges in training, computational overheads, and hardware limitations, providing insights into scalability and applicability. The developed SNN model will be evaluated through image classification tasks, involving training on image datasets, parameter tuning, and assessing accuracy and efficiency. Furthermore, a comparative analysis will be conducted between the performance of the SNN model and a conventional Convolution Neural Network (CNN) model, focusing on accuracy, computational efficiency, and resource utilization, to highlight potential advantages of SNNs over traditional neural network approaches for image classification tasks.

Acknowledgements

I would like to express my deepest gratitude to Dr.Dewei Yi, my thesis advisor, for their invaluable guidance, patience, and support throughout the course of this research. Your insights and suggestions were crucial in shaping this dissertation.

My sincere appreciation also goes out to all my teachers for their timely support, and thoughtful advice in my life

I am deeply grateful to my family, especially my parents and friends, for their understanding and endless love, which provided me with the strength and peace of mind necessary to complete my studies.

Lastly, I would like to acknowledge my father for financially supporting this research, making my studies possible.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the branch of Artificial Intelligence, machine learning gives the power to the computers to learn from data and improve performance. In today's world, Machine learning is crucial for data driven decision making, automating tasks, and personalizing user experiences. But in today's ever changing dynamics of the world order, Machine Learning lacks in different aspects. Traditional Machine Learning struggle with complex pattern recognition, particularly in high-dimensional spaces or when dealing with non-linear relationships within data along with that when the data is not enough. Additionally, traditional Machine Learning models often lack dynamic adaptation capabilities, remaining static once trained and requiring extensive retraining to adapt to changing environments or data distributions. To address this uncertainty, as traditional Machine Learning techniques may struggle with ambiguous or incomplete data, leading to a saturation in performance, we can try some different options. In this aspect, Bio inspired Machine Leaning offers us unique advantages over traditional machine learning models. Bio-inspired approaches, such as evolutionary algorithms and spiking neural networks, based on natural selection and the operational mechanisms of biological neural networks to enhance adaptability and efficiency.

In the field of artificial intelligence, bio-inspired machine learning stands out as a compelling area of study that seeks to bridge the complex mechanisms of biological systems with the computational power of machine learning algorithms. Bio-inspired machine learning is a fascinating part of artificial intelligence that looks at nature for inspiration. The traditional machine learning models are also on a edge of a saturation, where no new work is going on to improve the uncertainty to face new problems when there are very small datasets available. The idea of looking to biology for clues on how to build smarter machines has been around for a long time. It influenced the early days of artificial intelligence. For example, the networks of artificial 'neurons' in computers were inspired by our brain's structure. By mimicking this, researchers hope to create computer programs that can learn from a small amount of data and handle new, unexpected situations better. In this report, we're going to explore Spiking Neural networks as a bio-inspired machine learning model works, and how spiking neural networks is being developed and used today to handle complexity and open a new window at a point of time where traditional machine learning models are at a point of saturation to delivering new way out to deal with energy time and memory complexity.

1.1 Objectives

The primary goal of this project is to develop and evaluate a Spiking Neural Network (SNN) model image classification and evaluate its performance with traditional Convolutional Neural Networks (CNNs). This challenge objectives to harness the bio-inspired properties of SNNs to probably improve computational efficiency and model responsiveness in processing visible statistics. The specific desires of the mission are mentioned as follows:

Create a Spiking Neuron Layer Module: Develop a modular spiking neuron layer that may be included into large neural network architectures. This module will consist of the vital mechanics to simulate neuron spiking behavior, together with membrane capacity dynamics and spike generation primarily based on a threshold mechanism.

Develop a Spiking Neural Network Model: Construct a entire SNN version that carries more than one spiking neuron layers. This model might be designed to manner photo records correctly, leveraging the temporal dynamics of spikes for greater sample popularity.

Implement Input Encoder and Output Decoder: Design and enforce an enter encoder that converts raw photo records into spike sequences appropriate for processing by way of the SNN. Additionally, expand an output decoder that could interpret the spike styles generated with the aid of the SNN into a shape that is comparable to the output of traditional neural networks, facilitating direct performance comparisons.

Analyze the SNN Models and Their Complexities: Thoroughly analyze the complexities involved in the SNN model, including the challenges in training such networks, the computational overheads, and the practical limitations of current hardware. This analysis will provide insights into the scalability and applicability of SNNs.

Perform Image Classification Tasks Using the Model: Utilize the developed SNN model to perform image class tasks. This will contain schooling the model on a dataset of photographs, tuning parameters to optimize performance, and evaluating the model's accuracy and efficiency.

Compare Results with CNN Models: One of our main aim is to have a comparative analysis between the performance of the SNN model and a conventional CNN model on the same image classification tasks. This comparison will focus on accuracy, computational efficiency, and resource utilization to assess the potential advantages of SNNs over traditional neural network approaches.

1.2 Outline of the Thesis

This dissertation is organised into 7 chapters, as described below.

Chapter 1: Introduction

A presentation to the topic area and the problem statement. This chapter introduces the scope,

objectives, and significance of the project, setting the stage for the detailed exploration to follow.

Chapter 2: Background

Chapter 3: Spiking Neural Networks

A detailed description of spiking neural networks, its key architectures, and components has been described. In this chapter we have also discussed the important models of spiking neurons and Synapses.

Chapter 4: Literature Review

A comparison of state-of-the-art solutions and the research work going on in the field of bio inspired machine learning models. We have also discussed the methods and the results of the papers based on the bio inspired machine learning models and their approach and analysis on uses of SNN .

Chapter 5: Neural Network Models

A description of Spiking neural network models and their working procedure , the model's different variants and complexities and work flow of the models

Chapter 6: Evaluation

The detailed comparisons of the experiments that has been performed and the detailed comparison of the results and accuracy of the SNN models and the comparison of results and accuracy based of the performance of SNN and CNN models.

Chapter 7: Conclusions

A summary and critical analysis of the realized work, as well as possible future development directions. This chapter reflects on the research outcomes, discusses the potential future work based on the spiking neural networks and discusses potential of SNN in near future.

Chapter 2

Background

Spiking neural networks (SNNs) represent a significant progressive step in the field of neural networks and computational neuroscience. Traditional artificial neural networks (ANNs) process information in a continuous manner and typically require a significant amount of computational resources, especially when dealing with large-scale data inputs. ANNs are based on a simplified mathematical model of a neuron, which integrates inputs to produce a continuous output. SNNs operate through discrete events or 'spikes.' These spikes are brief and localized bursts of electrical activity, similar to the action potentials in biological neurons.

One of the key feature of SNN is their ability to encode information in the timing of spikes, not just rate or the pattern of activity. This time-based encoding can lead to more efficient representations of information, especially in the situations that involves data processing using sensors such as speed, video streams or in robotics.

The energy efficiency of the spiking neural networks also gives it an edge over the traditional methods of machine learnings. Similar like biological neurons, the neurons of spiking neural networks also consume energy when they fire, unlike the traditional ANN that perform continuous computations. This aspect of SNN can reduce the power consumption rapidly and make it ideal for deploy in low power devices.

Moreover the structure of Spiking neural networks allows for more efficient learning mechanism. They can adapt and learn from new information dynamically, making them more appropriate for the conditions where input pattern continuously changes. The adaptive learning capability is very crucial for applications in autonomous vehicles and real time decision making systems like rockets or drones where learning from temporal data is critical.

The development of Spiking neural networks also signifies a step towards building neural networks that not only mimic the structure but also the functionality of the human brain, demonstrating more versatile and sophisticated AI systems. This evolution reflects a trend in computational neuroscience and artificial intelligence to develop models that are not only computationally effective but also closely aligned with biological systems came across thousand years of evolution.

2.1 Motivation

The motivation of using spiking neural networks in object detection revolves around their unique capabilities, which make them unique and opens a new horizon to enhance the real time processing, power efficiency and adaptability. These characteristics are important in crucial applications where decision making has to be unique depending on the situation.

2.1.1 Real-time and Streaming Data:

Spiking neural networks are designed from the concept to handle the real time data flow in a efficient way. The spiking mechanism of the SNN which tries to mimic the biological neurons, allows them to process inputs as they arrives, making is very much suitable for continuous data flow. This makes the whole system efficient to make decisions, where we are making a failed based decision making system. So whenever we make an algorithm, where we have given a pathway, what to do when everything goes wrong, spiking neural network based model may handle those situation better than the traditional neural networks.

2.1.2 Suitability for Edge Devices:

The energy efficiency of Spiking neural networks makes them ideal for integration into edge computing devices which require low power consumption for long operational period without sacrificing it's performance. This is particularly important in wearable techs and remote sensors used in various industrial and environmental monitoring applications.

2.1.3 Adaptability and Learning Capabilities:

Unlike the traditional neural networks which need significantly large amount of data, on which we can train our models and get rely on, Spiking neural networks can adapt more fluidly. The spiking neural network's learning mechanism allows it to adjust based on the situation based on the mechanisms such as spike timing dependent plasticity(STDP), where the network learns from the timing between input and output spikes, and continue the process of leaning and adaptation.

The use of Spiking Neural Networks (SNNs) in object detection systems is motivated by their efficiency, which is similar to biological systems, their ability to process information in real time, and their flexibility to adapt to new or changing situations. These qualities make SNNs an exciting option for improving object detection in many real-world applications, making these systems both more effective and efficient.

2.1.4 Aim of the Project

The main goal of the project is to understand how a spiking neural network works and find a way to adapt the spiking neural networks in the task of image classifications and object detections as there are not significant work has been done. While doing the research, we will try to focus to learn the dynamics of the spiking neural networks, adapt the different models that is been on theory, and make a some practical implementation of those different neuron and synaptic models to improve the process of image classification and object detection.

2.1.5 Research Areas

The project will explore several key features of Spiking neural networks to determine their effectiveness in image classification and object detection, where we will use different parameters that will check the temporal dynamics, threshold mechanism and generalization capabilities. The study will look into how SNNs use time and threshold-based firing to decrease false positives and increase detection reliability in noisy or varying settings by fine-tuning neuron threshold levels for better detection accuracy. The study will also focus on the SNN's ability to generalise from the limited data, important for the situations where the system must perform well in untrained situations.

While implementing the theories in a practical project work, we will try to create a model that will able to do an image classification task using the spiking neural network mechanism and different neurons and synaptic models, and then we will try to enhance the model to compare the efficiency of these models, and the efficiency/effect of different parameters in the model. Then for further use, we will try to make an object detection model that can detect/track a fast moving object.

Chapter 3

Spiking Neural Networks:

Spiking neural networks are composed of several key components that helps us to differentiate a Spiking neural networks from a traditional neural networks.

3.1 Components:

3.1.1 Spiking Neurons:

These are the fundamental computational units in SNNs, designed to mimic the behaviour of biological neurons. They produce spikes when stimulated, and their interactions are modelled over time.

3.1.2 Synapses:

Synapses are connections between neurons that transmit signals through spikes. Synapses can be excitatory, promoting the generation of spikes, or inhibitory, suppressing it. They often exhibit plasticity, changing their strength based on the neurons' spiking activity, thereby mimicking learning and memory processes.

3.1.3 Membrane Potential:

Each neuron has a membrane potential that changes dynamically based on incoming spikes. When this potential exceeds a certain threshold, the neuron fires a spike and then resets its potential, often followed by a refractory period during which it is less responsive or inactive.//
These three components are the main foundation of a spiking neural network. To demonstrate that, I have displayed how a spike gets generated. When the input current cross the threshold voltage, it generates a spike and the voltage again goes to reset.

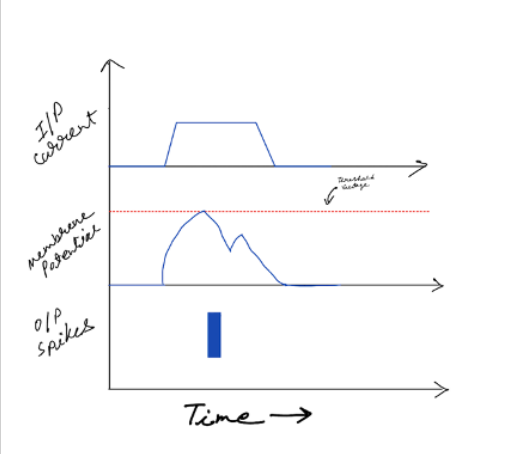

Figure 3.1: Neuronal Response with Single Spikes

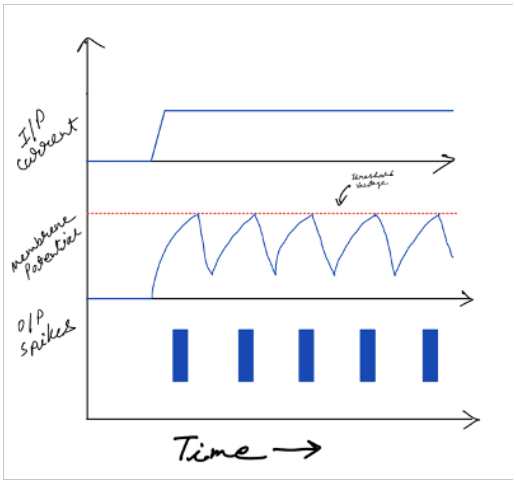

Figure 3.2: Neuronal Response with Multiple Spikes

In these images we can see the constant current is the main cause of generating the spikes. As there is a constant current, the membrane potential approaches towards the threshold voltage, and when it reaches the threshold voltage the neuron generates a spike. Then the membrane potential approaches towards the reset rapidly. Still there is a flow of current which triggers the membrane potential again to reach the threshold voltage and it again triggers the neuron to generating a spike again and the whole cycle continues until there is a change of input current with time.

There are other significant components and steps also comes in place to make the whole model work. The network topologies, the encoders and decoders, which are responsible for translating the data into spike patterns and again interpreting the spike patterns into usable data forms plays a crucial role. Along with that the learning rules and neuromorphic hardware which mimics the neurobiological architectures of the nervous system, designed specifically for efficiently running spiking neural networks makes the process more smoother. Together, these components enable spiking neural networks to simulate brain-like processing, making them suitable for tasks that require complex temporal dynamics and real-time operation.

3.2 Neuron Models:

Spiking neural networks (SNN) models are more closely how biological neural networks operate compared to the traditional artificial neural networks. In SNN the neurons are the key ingredient of the model. Neurons communicate primarily sending discrete spikes like signal over time, which is different than the continuous value based communication used in other type of neural networks.

1. Input neurons: These neurons are responsible for receiving and encoding external signals into spike patterns. They do not process the information but act as the interface between the external world and the network. The method of encoding can vary; some common techniques include rate encoding, where the frequency of spikes corresponds to the intensity of the stimulus, and temporal encoding, where the timing of spikes carries information.

2. Excitatory Neurons: These are the most common type of neurons in SNNs. They increase the electrical activity of other neurons they connect to. When an excitatory neuron fires a spike, it sends a signal that potentially brings the receiving neuron closer to its threshold for firing. In a biological context, these neurons typically release neurotransmitters like glutamate, which promotes action potentials in the target neurons.

3. Output Neurons: At the end of processing chains in the network, output neurons take the incoming spikes from other neurons and convert their activity into a form that can be sent out of the network to drive actions or further processing. In biological systems, these might be motor neurons that cause muscles to contract or other neurons that impact an organ's function.

There are different types of neuron models based on their firing patterns. We will discuss these models in details.

3.2.1 Leaky integrate and fire model:

Leaky integrate and fire model is a simple and yet effective model used in a computational neuroscience and artificial neural network particularly in spiking neural networks to simulate the electrical behaviour of neurons. It integrates the inputs it receives until the membrane potential reaches to its threshold, at which point it fires an action potential spike and resets to the neuron voltage. The model is valuable for its balance and simplicity and biophysical relevance and the mimicking and allowing study of biological inspired artificial neural networks without the computational complexity.

The idea behind the leaky integrate and fire model is to demonstrate the neurons as a simple equations and mimicking the behavioural pattern of biological neurons. The neuron accumulates input until its membrane potential reaches a specific threshold, at which point it fires to generate a spike and reset. The Leaky integrate and fire model enhances the phenomenon by utilizing the leaky term that causes the membrane potential to decay over time, demonstrating the natural leakiness of cell membranes.

Mathematical Equations:

1. Membrane Potential (V): Represents the neuron's current voltage relative to the outside of the cell. It is influenced by inputs and the leakiness of the membrane.

2. Input Current (I): The external stimuli applied to the neuron, which can be from synaptic transmissions or other sources. This current is what causes changes in the membrane potential.

3. Leakage: The LIF model accounts for the fact that neuronal membranes are not perfect insulators and gradually lose their stored charge over time. This leakage is proportional to the difference between the current membrane potential and the neuron's resting potential.

4. Threshold and Reset: The neuron has a voltage threshold (V_th). Upon reaching this threshold, it fires an action potential and the membrane potential is then reset to a lower value (V_reset), typically close to the resting potential. Along with that V_rest is the resting voltage of the membrane.

$$\frac{dV}{dt} = \frac{V_{rest} - V(t)}{\tau} + I(t)$$

Figure 3.3: Leaky Integrator Model Equation

$$\frac{dV}{dt} = -\frac{1}{\tau_m}(V - E_{\text{leak}}) + \frac{I}{C_m}$$

Figure 3.4: Modified Leaky Integrate-and-Fire Model Equation

τ_m. is the membrane time constant, which determines the rate of exponential decay due to leakage. E_leak is the leak potential, equivalent to the resting potential. Cm is the membrane capacitance, affecting how quickly the potential changes in response to input currents.

Demonstration: To showcase a Leaky integrate and fire model, I have created a function named "leaky_integrate_and_fire". Initially, I have defined the input parameters such as membrane potential ('mem'), an input value ('x'), a synaptic weight ('w'), a decay factor ('beta'), and a firing threshold('threshold'). The membrane potential changes over time by accounting for leakage (represented by beta) and the impact of the input current, which is adjusted by the synaptic weight. When the membrane potential surpasses the threshold, the neuron will fire a spike, and the membrane potential will fall towards reset.

```
def leaky_integrate_and_fire(mem, x, w, beta, threshold=1):
    spk = (mem > threshold)
    mem = beta * mem + w*x - spk*threshold
    return spk, mem

delta_t = torch.tensor(1e-3)
tau = torch.tensor(5e-3)
beta = torch.exp(-delta_t/tau)

x = torch.cat((torch.zeros(10), torch.ones(190)*0.5), 0)
mem = torch.zeros(1)
mem_rec = []
spk_rec = []

w = 0.4
beta = 0.819

num_steps = 200
for step in range(num_steps):
    spk, mem = leaky_integrate_and_fire(mem, x[step], w=w, beta=beta)
    mem_rec.append(mem)
    spk_rec.append(spk)

mem_rec = torch.stack(mem_rec)
spk_rec = torch.stack(spk_rec)
```

Figure 3.5

The procedure begins with initial parameters defining the time step (delta_t) and the membrane's time constant (tau). Initially the input current(x) has been defined zero, followed by a sustained value representing a constant stimulus. The membrane potential (mem) begins at zero, and arrays

to track the membrane potential (mem_rec) and spikes (spk_rec) are set up for the duration of the simulation time steps. The function named 'leaky_integrate_and_fire' is repeatedly called to simulate the passage of time in discrete steps. Lastly, the plots showcases the neuron's behaviour throughout the process.

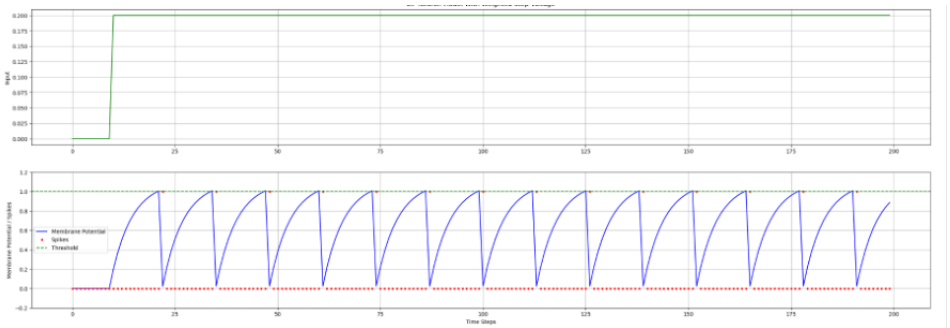

Figure 3.6

The top graph shows the input current over time, and the bottom graph displays the membrane potential. A horizontal line indicates the threshold level, helping to visualize when spikes are triggered. The plot offers showcases the dynamics of the Leaky integrate and fire neuron model, demonstrating how neurons can process inputs and generate an output in the form of spikes.

3.2.2 Adaptive integrate and fire neuron:

The adaptive integrate and fire(AdEx) model is sophisticated version of integrate and fire neuron model that incorporates to more accurately simulate the firing behaviour of biological neurons. This model is useful to demonstrate computational neuroscience for studying how neurons adjust their firing patterns in response to their input history.

The Adaptive integrate and fire model builds on the classic integrate and fire approach by adding a mechanism for spike frequency adaptation. This process demonstrate the observed biological behaviour where the response of a neuron to a constant input current diminishes over time, making the neuron less likely to fire spikes as frequently as it initially did.

Components of the AdEx Model:

Membrane Potential (V): Represents the electrical charge across the neuron's membrane, influenced by incoming spikes and the adapt leakiness of the neuron. Input Current (I): External currents applied to the neuron from synaptic inputs or external sources, driving changes in the membrane potential.

Adaptation Variable (w): A crucial feature of the AdEx model. This variable increases with each spike and decays exponentially otherwise. It represents the accumulation of ion channel changes or other cellular mechanisms that contribute to reducing the neuron's excitability over time.

Threshold and Reset Dynamics: Similar to other integrate-and-fire models, the AdEx model has a threshold (V_th) that, when crossed by the membrane potential, triggers a spike. After firing, the membrane potential is reset to a specific value (V_reset), and the adaptation variable is increased by a set amount (b), enhancing the neuron's adaptability.

The dynamics of the AdEx model can be described by these two equations.

$$\frac{dV}{dt} = -\frac{1}{\tau_m}(V - E_{leak}) + \frac{I}{C_m} - w$$

$$\frac{dw}{dt} = a(V - E_{leak}) - \frac{w}{\tau_w}$$

Figure 3.7: Mathematical Equation of AdEx Model

Where τ_m is the membrane time constant, which influences the rate of exponential decay of the membrane potential. E_{leak} is the leak potential, akin to the resting potential. C_m is the membrane capacitance. a, b, and τ_w are parameters defining the strength of adaptation. The parameter a scales the subthreshold adaptation, b determines the spike-triggered increase in w, and τ_m is the adaptation time constant. w is the adaptation constant. As the neuron fires, w can increase or decrease depending on the neuron's activity. In these equations, V represents the membrane potential of the neuron. The term $\frac{I}{C_m}$ reflects the influence of external input current I to the membrane capacitance C_m.

The membrane potential reaches the threshold voltage V_{th}, and when it exceeds V_{th}, the neuron generates a spike, and the membrane potential again reaches V_{reset}. Along with that, the adaptation variable w, is increased by b, which affects the membrane potential dynamics by making it harder for the neuron to reach the firing threshold again soon.

The AdEx model, or Adaptive Exponential Integrate-and-Fire model, is used to study various neuronal behaviors and brain functions, including how neurons adapt to continuous stimulation and how this affects learning and memory.

Demonstration of AdEx model using a python code: To showcase the adaptation of the adaptive integrate and fire model, I utilized a Python script. Initially, I defined the input parameters such as the membrane time constant τ_m as 20 ms, membrane capacitance C_m as 200 pF, leak potential E_{leak} as -70 mV, input current I as 300 pA, adaptation increment b as 5 nS (spike-triggered increment), subthreshold adaptation a as 4 nS, threshold V_{th} as -50 mV, and reset potential V_{reset} as -65 mV. The script sets the stage by defining a set of parameters: dt as the time step for numerical integration, and time as the simulation duration from 0 to 500 milliseconds.

The simulation starts with the membrane potential V set to the leak potential E_{leak}, and the adaptation variable w set to 0, indicating no initial adaptation. Two lists, V_{trace} and w_{trace}, are also created to store the values of the membrane potential and the adaptation variable at each time step.

While performing inside the loop, it calculates the changes in the membrane potential $\frac{dV}{dt}$ and the adaptation variable $\frac{dw}{dt}$ based on the AdEx model equations. The membrane potential V is then updated by the amount $\frac{dV}{dt}$, and w is updated by $\frac{dw}{dt}$. If V goes above the threshold V_{th}, generating

```
dt = 0.1
time = np.arange(0, 500, dt)
tau_m = 20.0
C_m = 200.0
E_leak = -70.0
I = 300.0
b = 5.0
a = 4.0
tau_w = 100.0
V_th = -50.0
V_reset = -65.0
V = E_leak
w = 0.0
V_trace = []
w_trace = []
for t in time:

    dV_dt = (E_leak - V + I/C_m - w) / tau_m
    V += dV_dt * dt
    dw_dt = (a * (V - E_leak) - w) / tau_w
    w += dw_dt * dt
    if V >= V_th:
        V = V_reset
        w += b
    V_trace.append(V)
    w_trace.append(w)
```

Figure 3.8: AdEx Model to simulate patterns

a spike, it is reset to V_{reset}, and the adaptation variable w is increased by b to represent the spike-triggered adaptation response.

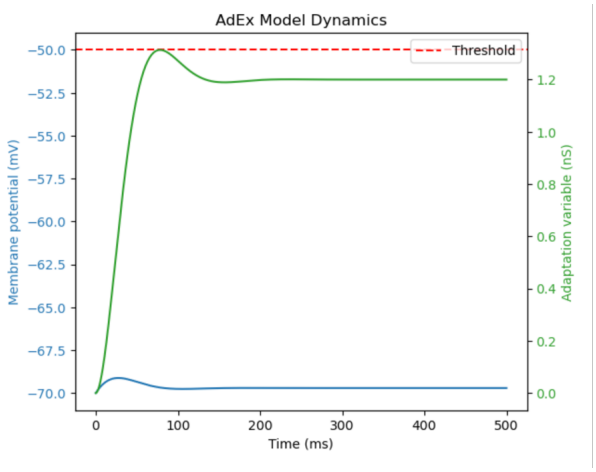

Figure 3.9: Simulation of Adex model

In the plot, the membrane potential is shown in blue and the adaptation variable in green, each on their own respective y-axis to clearly distinguish their scales and behaviours. A red dashed line represents the threshold potential. This plot demonstrates the dynamic behaviour of the neuron, showcasing how it responds to constant input and adapts over time, showcasing the characteristic spike patterns.

3.2.3 Hodgkin – Huxley model:

The Hodgkin-Huxley model is a seminal computational framework in neuroscience, developed by Alan Hodgkin and Andrew Huxley in 1952. This model describes how action potentials in neurons are initiated and propagated, based on the biophysical properties of neuronal membranes. It is particularly useful to demonstrate the detailed representation of ionic flows through the neuronal membrane, and it has laid the foundational understanding of the electrical behaviour of nerve cells. The model is very much useful in the study of spiking neural networks, demonstrating insights of precise mechanism that underlie neural excitability and signal transmission.

The Hodgkin-Huxley model focuses on the squid giant axon (a highly specialized biological device whose sole purpose is to rapidly and reliably activate the muscles of the squid's mantle to generate its jet propulsion mechanism) to describe the basic principal of electrical signalling in neurons. It accounts the dynamics of the neuron's action potential through equations that represents the conductance of sodium (Na+) and potassium (K+) ions, which are critical for the generation and propagation of action potentials The Hodgkin – Huxley model introduces the concept that action potentials are driven by the movement of ions across the neuron's membrane through specific ion channels. It mathematically formulates the flow of potassium and sodium ions as being key to this process. The conductance of ion channels are not constant but depend on the membrane voltage and the history of channel states (open or closed). This dependency is described through differential equations relating to the gating variables.

$$C_m \frac{dV}{dt} = I - I_{Na} - I_K - I_L$$

Figure 3.10: Mathematical Equation of Hodgkin – Huxley Model

where:

- C_m is the membrane capacitance per unit area.
- V is the membrane potential.
- I is the total current supplied to the cell.
- $I_{\text{Na}}, I_{\text{K}}$, and I_{L} are the sodium, potassium, and leakage ionic currents, respectively.

where:

- g_{Na}, g_{K}, and g_{L} are the maximum conductance for the sodium, potassium, and leakage channels, respectively.
- $E_{\text{Na}}, E_{\text{K}}$, and E_{L} are the equilibrium potentials for the sodium, potassium, and leakage channels, respectively.

$$I_{Na} = g_{Na} m^3 h (V - E_{Na})$$
$$I_K = g_K n^4 (V - E_K)$$
$$I_L = g_L (V - E_L)$$

Figure 3.11

- m, h, and n are variables that represent the gating particles of the sodium and potassium channels and are governed by their own differential equations that describe their time and voltage-dependent changes.

The dynamics of these gating variables are described by:

$$\frac{dm}{dt} = \alpha_m(1-m) - \beta_m m$$
$$\frac{dh}{dt} = \alpha_h(1-h) - \beta_h h$$
$$\frac{dn}{dt} = \alpha_n(1-n) - \beta_n n$$

In the context of spiking neural networks, the Hodgkin-Huxley model provides a detailed method to demonstrate how neurons interact through their action potentials. The Hodgkin – Huxley model shows neural activity under various conditions and is essential for understanding the impact of ionic dynamics on neural computation. By integrating Hodgkin-Huxley neurons into larger network models, we can explore complex behaviours like synchronization, oscillations, and wave propagation in neural tissues for further mimicking of biological neurons in future use of spiking neural networks.

To effectively describe the Hodgkin-Huxley model using graphs, we can simulate and visualize the behaviour of the neuron's membrane potential and the variables that control the flow of ions through sodium and potassium channels. These visualizations will highlight how changes in these variables correlate with action potentials, demonstrating the dynamic nature of neural signalling as described by the model.

For that, I have created a Python code to demonstrate the Hodgkin-Huxley model. Initially, I have fixed the parameters for setting the neuron's biophysical properties, including membrane capacitance (C_m), maximum conductance for sodium (g_{Na}), potassium (g_{K}), and leakage (g_{L}), and the respective equilibrium potentials for these ions (V_{Na}, V_{K}, V_{L}). The initial membrane potential is set to -65 mV, and the initial values for the variables m, h, and n have been defined. A time array has also been created with increments of 0.01 ms to simulate 50 milliseconds of neuronal activity. An external current (I_{ext}) is applied between 10 and 20 milliseconds to stimulate the neuron. The membrane potential V has been updated by integrating the net ionic current, which is the difference between the external current and the sum of all ionic currents, over the membrane capacitance.

```
Cm = 1.0
gNa = 120.0
gK = 36.0
gL = 0.3
VNa = 50.0
VK = -77.0
VL = -54.4
V = -65.0
m = 0.05
h = 0.6
n = 0.32
dt = 0.01
time = np.arange(0, 50, dt)
I_ext = np.zeros_like(time)
I_ext[(time >= 10) & (time <= 20)] = 10
V_trace = np.zeros_like(time)
m_trace = np.zeros_like(time)
h_trace = np.zeros_like(time)
n_trace = np.zeros_like(time)

def alpha_m(V): return 0.1 * (25 - V) / (np.exp((25 - V) / 10) - 1)
def beta_m(V): return 4 * np.exp(-V / 18)
def alpha_h(V): return 0.07 * np.exp(-V / 20)
def beta_h(V): return 1 / (np.exp((30 - V) / 10) + 1)
def alpha_n(V): return 0.01 * (10 - V) / (np.exp((10 - V) / 10) - 1)
def beta_n(V): return 0.125 * np.exp(-V / 80)
```

Figure 3.12

```
for i in range(1, len(time)):
    m += dt * (alpha_m(V) * (1 - m) - beta_m(V) * m)
    h += dt * (alpha_h(V) * (1 - h) - beta_h(V) * h)
    n += dt * (alpha_n(V) * (1 - n) - beta_n(V) * n)

    INa = gNa * (m**3) * h * (V - VNa)
    IK = gK * (n**4) * (V - VK)
    IL = gL * (V - VL)
    dV = (I_ext[i] - INa - IK - IL) / Cm
    V += dV * dt
    V_trace[i] = V
    m_trace[i] = m
    h_trace[i] = h
    n_trace[i] = n
```

Figure 3.13

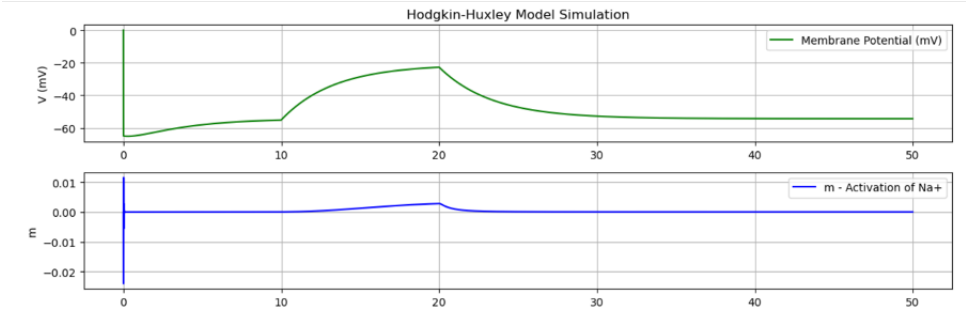

Figure 3.14

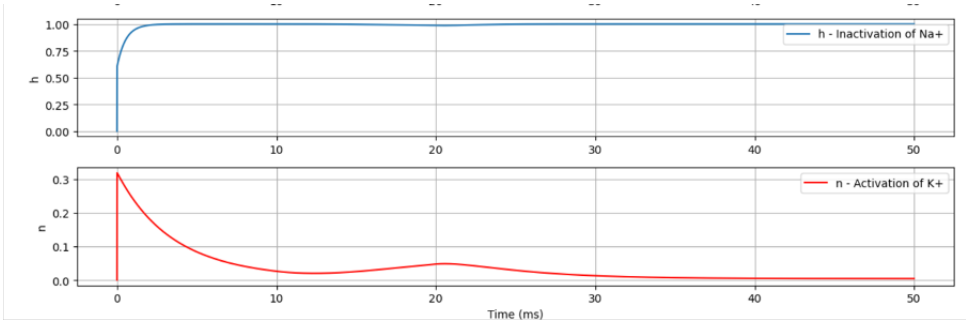

Figure 3.15

The 1st graph shows the membrane potential (V) of the neuron. We can observe the action potential spike when the external current is applied between 10 ms and 20 ms. The sharp rise and subsequent fall in potential demonstrate the typical behaviour of an action potential.

The 2nd Panel, 'm' represents the activation state of sodium channels, which increases rapidly as the membrane potential approaches the threshold for an action potential, facilitating the rapid influx of Na^{++} ions.

The 3rd Panel, 'h' represents the inactivation state of sodium channels, which starts high and decreases as the action potential peaks, contributing to the termination of the Na^{++} influx.

The 4th Panel, 'n' shows the activation state of potassium channels, which increases more slowly. This delayed response allows K^{++} ions to exit the neuron more prominently after the peak of the

action potential, helping to repolarize and reset the membrane potential.

3.2.4 Izhikevich Neuron model:

The Izhikevich neuron model is a computational model of neuronal dynamics that combines the biological plausibility of biophysical models with the computational efficiency of simple integrate-and-fire models. Developed by Eugene Izhikevich in 2003, this model is particularly noted for its ability to reproduce diverse patterns of neural activity seen in biological neurons with a minimal set of equations and parameters.

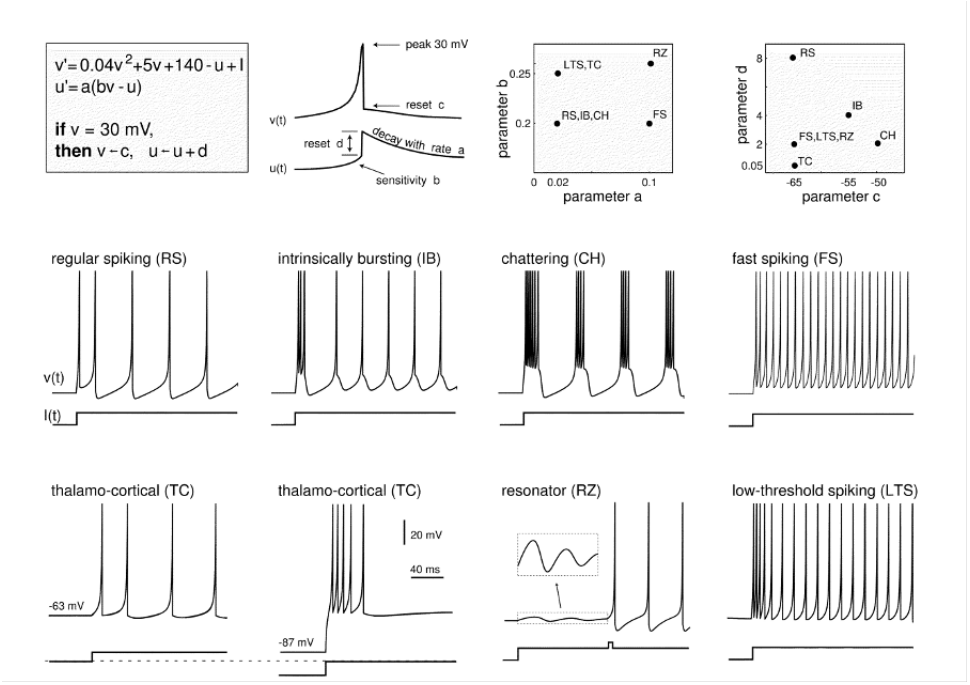

Figure 3.16

Mathematical description: The dynamics of the Izhikevich model are described by the following equations

$$\frac{dV}{dt} = 0.04V^2 + 5V + 140 - u + I$$
$$\frac{du}{dt} = a(bV - u)$$

Figure 3.17: mathematical equation of Izhikevich model

The neuronal dynamics are described by two main equations. The first equation determines the change in membrane potential (V) over time, incorporating the influence of a recovery variable (u), and I, the external input current. The second equation describes the evolution of the recovery variable, which models the neuron's refractory period and spike frequency adaptation. Here, I represents the synaptic or injected current, and a, b, c, and d are parameters that can be adjusted to replicate different types of neuronal behavior.

When the membrane potential reaches a certain threshold (usually set at 30 mV), the neuron is considered to have fired, and the membrane potential and recovery variable are reset according

to the following conditions: If $V \geq 30$ mV, then set:

$$V \leftarrow c$$

$$u \leftarrow u + d$$

These conditions ensure that the neuron responds to high membrane potentials by resetting, which simulates the firing of a real neuron and its subsequent refractory period. This particular model might not be necessity for our work, but to make more sophisticated neuron model, we should consider this kind of model in future.

3.3 Synaptic model

Synapses are the connections between neurons through which they communicate. In SNNs, synapses not only transmit the strength of the connection (weight) but also the timing of spikes, which is crucial for the network's performance. In a spiking neural networks, synaptic model plays a crucial role in defining how neurons interact and process information. Synaptic inputs are integrated by the neuronal model, such as the leaky integrate-and-fire or Hodgkin-Huxley models. These models describe how incoming spikes influence the membrane potential of the postsynaptic neuron, determining whether it will fire a spike in response. Synapses in SNN are the point of communication between neurons. Unlike the traditional neural networks that use continuous values, bio inspired spiking neural networks operates with the discrete events(spikes). Each synapses has an associated weight that determines the strength of the connection between two neurons. Synaptic models in spiking neural networks are also account for the timing of spikes. The exact time when a spike generates, can encode information. Synaptic efficiency might depend on the timing difference between spikes from pre-synaptic and postsynaptic neurons, known as spike-timing-dependent plasticity(STDP). STDP varies the synaptic weight based on the relative timing of the spikes, enhancing the network's ability to perform temporal pattern recognition. There are two primary type of synaptic models, Static Synaptic Models and Dynamic Synaptic Models. Each category has their individual identity and applications in modelling a bio inspired spiking neural networks.

3.3.1 Static Synaptic Models

Static synaptic models use fixed synaptic weights that do not change in response to neural activity. These models are simpler and typically used in situations where learning or synaptic adaptation is not required. Synaptic weights between pre and post neurons are pre-set and remain constant does not change for the activity of neurons. Because the synaptic weights do not change, the computational complexity is reduced, making these models easier to analyse and implement. Static synaptic models are often used in theoretical studies where the focus is on the network typologies and firing pattern without influencing the dynamics of leanings where the models can be trained offline and deployed without further adaptation.

3.3.2 Dynamic Synaptic Models

Dynamic synaptic models are more complex and are designed to mimic the adaptive nature of the biological synapses. These models allow synaptic weights to change based on the neurons' activity, facilitating learning and memory in neural networks. Dynamic models can be further

differentiate based on the type of plasticity.

Rate-Based Plasticity: Synaptic weights change according to the average firing rates of the connected neurons, emphasising the principle that ”neurons that fire together, wire together”, quoted by Donald Hebb in 1949. This type of plasticity does not require precise timing of spikes, making it suitable for models where overall rate of activity is more critical than the specific timing of spikes.

Mathematical Model The synaptic weight adjustment can be modeled as:

$$\Delta w = \eta \cdot r_{\text{pre}} \cdot r_{\text{post}}$$

where η is the learning rate parameter.

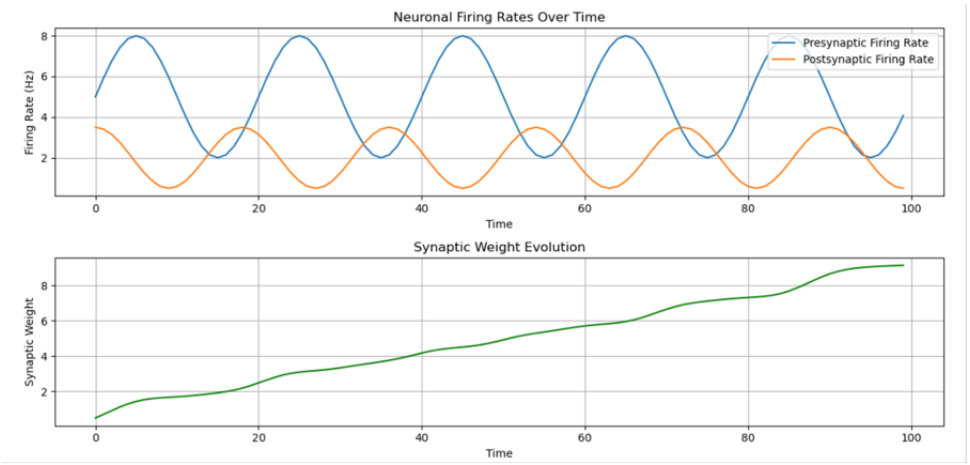

Figure 3.18

plot shows how the synaptic weights evolves over time in response to the interaction between the firing rates of two neurons. Here the weight increases indicating that the synaptic strength has been increased, where the rates of two neurons are well aligned.

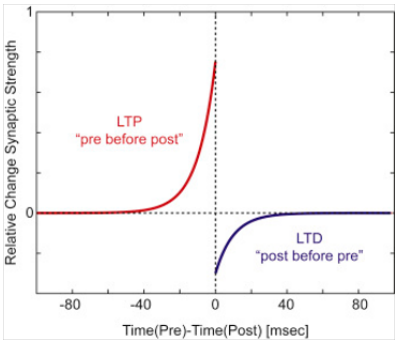

Figure 3.19

Spike-Timing-Dependent Plasticity (STDP): changes based on the relative timing of the arrival of spikes from the pre-synaptic neuron and the generation of spikes in the postsynaptic neuron. The mechanism of synapse strengthens (potentiating) when pre-synaptic spikes precede postsynaptic spikes and weakens (depression) when postsynaptic spikes precede pre-synaptic spikes.

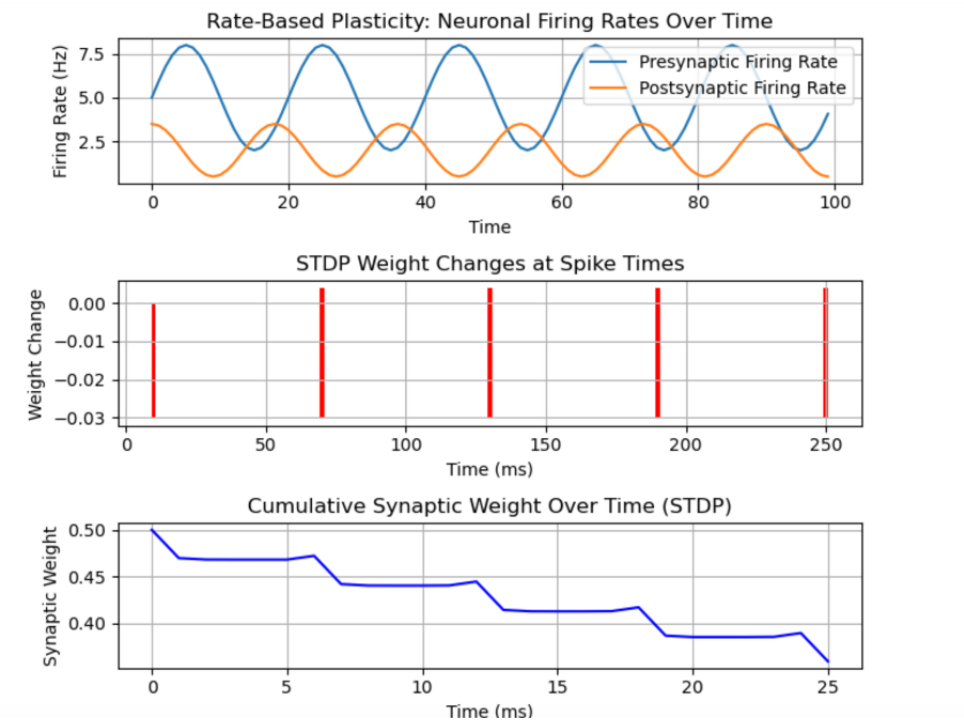

Figure 3.20

The second plot shows that the change of synaptic weight due to individual spike timing difference between pre-synaptic and post synaptic spikes. It also demonstrate the amount and direction of weight change: positive for potentiation (when presynaptic spikes precede postsynaptic spikes) and negative for depression (when pre-synaptic spikes follow postsynaptic spikes).

Cumulative Synaptic Weight Over Time (STDP) plot shows the cumulative effect of STDP on synaptic weight. It enhances the weight changes over time, demonstrating the dynamic evolution of synaptic strength based on precise spike timings. This plot shows the net effect of multiple spike interactions over the simulation period, highlighting how synaptic efficacy evolves due to temporal patterns in neuronal firing.

Short-Term and Long-Term Plasticity: Short-Term plasticity includes mechanisms like facilitation and depression that temporarily increase or decrease synaptic efficacy based on recent activity

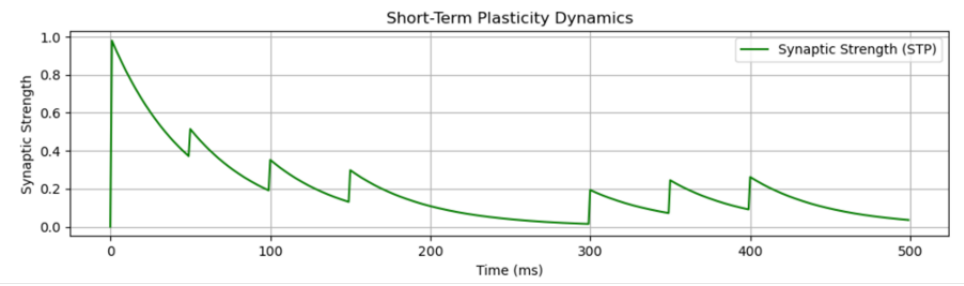

Figure 3.21

The plot demonstrates how synaptic strength changes in response to individual spikes under the Short-Term Plasticity model. The increases in synaptic strength immediately following each spike,

reflecting facilitation. This is followed by a decay back towards a baseline, simulating the temporary nature of short-term plastic changes like facilitation and depression.

Long-Term Plasticity includes long-term potentiation (LTP) and depression (LTD), which make more permanent changes to synaptic strength based on sustained patterns of activity.

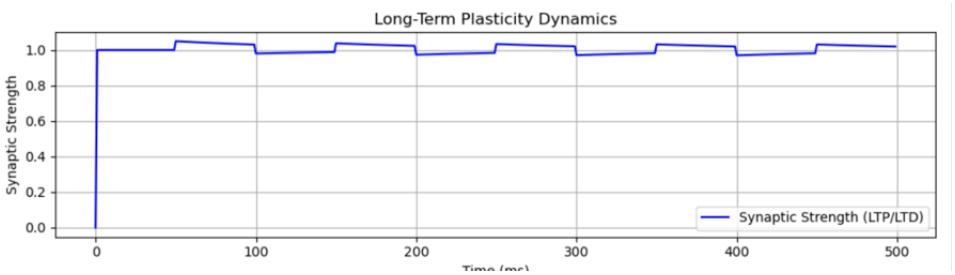

Figure 3.22

The plot demonstrates the effects of Long-Term Plasticity, including both Long-Term Potentiation (LTP) and Long-Term Depression (LTD). Synaptic strength increases with paired spikes that are frequent enough to trigger LTP, while less frequent or misaligned spikes lead to decreases due to LTD. Over time, these changes piles up, leading to a more permanent adjustment in synaptic strength.

Static and dynamic synaptic models provide the framework for simulating different aspects of neuronal behaviour in SNNs. While static models offer simplicity and stability, dynamic models bring adaptability and realism, making them suitable for a wide range of applications from basic research to complex computational tasks.

Chapter 4

Literature Review

4.1 Neuromorphic computing

In the field of biologically inspired machine learning, there are so much works going on. One of those fields are neuromorphic computing. Event-based simulation in neuromorphic computing, particularly when applied to Spiking Neural Networks (SNNs), refers to a computational approach that relies on the discrete, asynchronous occurrence of events (spikes) for processing information. This simulation style closely mirrors the biological processes found in the human brain, where neurons communicate by sending and receiving spikes or impulses only when certain conditions are met. Unlike traditional computers that use a continuous flow of binary data (0s and 1s), neuromorphic systems operate using spikes—a form of digital signal that mimics the binary "pulses" neurons use to communicate. These spikes are generated only when a particular threshold of stimulation is reached, reflecting an event-driven process.

One of the such research has been conducted in Peking University, Beijing, China and published by 'Nature Communications', in the year 2020. The research paper named "**Spiking neurons with spatiotemporal dynamics and gain modulation for monolithically integrated memristive neural networks**" affiliated by Qingxi Duan, Zhaokun Jing, Xiaolong Zou, Yanghao Wang , Ke Yang1, Teng Zhang, Si Wu, Ru Huang and Yuchao Yang discusses various aspects of spiking neurons with a focus on neuromorphic computing using memristive neural networks, especially address the concept of event-driven simulation of spiking neural networks.
The research presents an advancement in neuromorphic computing through the development of an artificial neuron based on NbOx volatile memristor. This neuron showcases capabilities beyond the traditional all-or-nothing spiking, including threshold-driven spiking, spatiotemporal integration, and gain modulation. A 4x4 fully memristive neural network, integrating these neurons with nonvolatile TaOx memristor-based synapses, demonstrates enhanced functionalities like pattern recognition and coincidence detection, pushing forward the potential for intelligent, neuromorphic systems.

The study has conducted for the need to enhance the capabilities of artificial neurons to match the complex functionalities of biological neurons. Traditional hardware implementations do not fully capture the nonlinear and dynamic nature of biological processing, often limited by energy inefficiency and scalability. The research explores the use of memristive devices, which offer promising

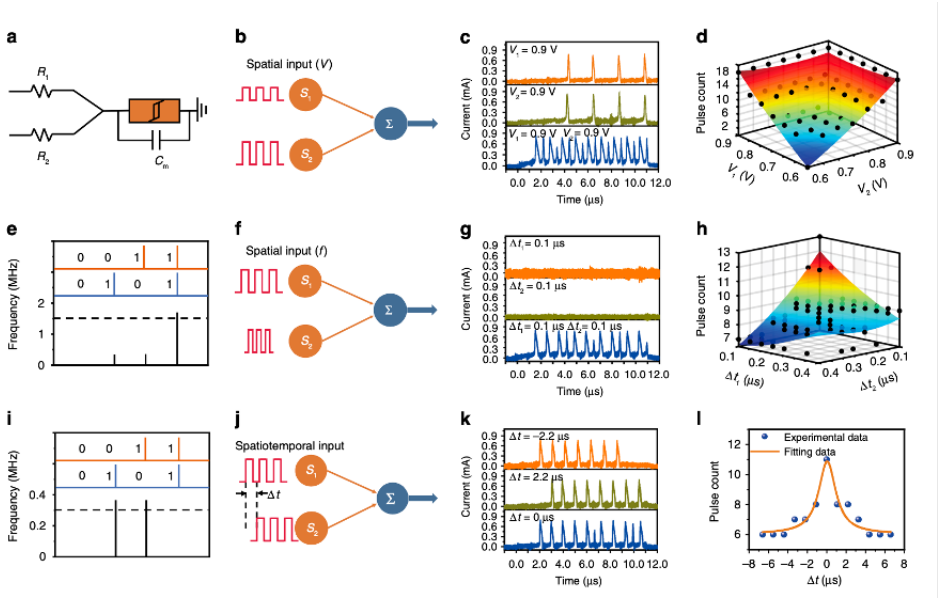

Figure 4.1

pathways due to their compactness and functional versatility. In the study, the research team developed an NbOx-based artificial neuron that utilizes NbOx memristors to enable threshold-driven spiking and spatiotemporal integration. This innovation extends the neuron's capabilities by introducing dynamic logic functionalities, including XOR operations that are essential for handling non-linear separability in computational tasks. A key feature of this development is the implementation of gain modulation, which allows for dynamic adjustment of the neuron's response based on the strength and type of synaptic inputs. This enhances the adaptability and efficiency of the neuron in processing complex information.

In our experiment we are using the SNN and try to simulate the Spiking neural networks. But the environment for that SNN can execute itself properly will be dependent on the structure of our hardware. These kind of neuromorphic experiment is important to execute the spiking neural network models.

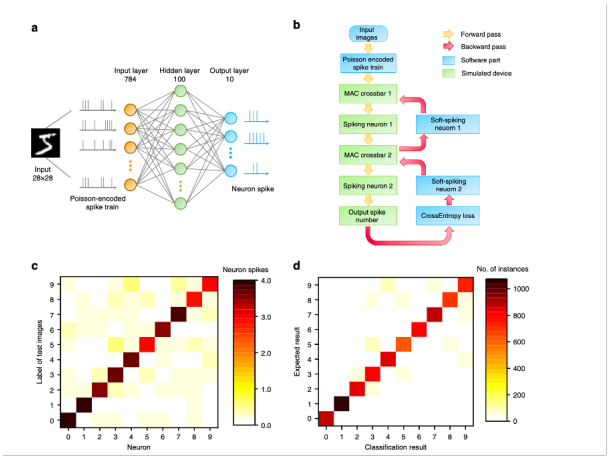

Figure 4.2

Further advancing the practical application of these innovations, the researchers successfully demonstrated a monolithically integrated 4x4 spiking neural network that incorporates both volatile and non-volatile memristive components. The network enhances the online learning and pattern recognition tasks, demonstrating the potential of these advanced neurons in real-world neuromorphic systems. The network's performance is marked by high efficiency and low power consumption, which are critical factors for scaling this technology for real world applications. the network's ability to perform spatiotemporal summation plays an important role in processing inputs across both space and time, demonstrating more sophisticated decision-making capabilities.

This study helps us to understand the efficiency, real time processing capabilities of SNN and if these kind of project gets executed in nere future, we will get different hardware architecture that will open new horizons for spiking neural networks.

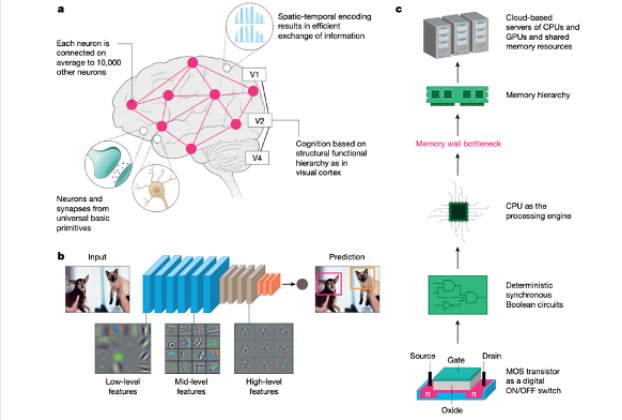

Figure 4.3

In Spiking neural network models, neurons communicate by sending discrete, sparse, and temporally precise spikes. The timing and frequency of these spikes, rather than continuous numerical values, carry information. This inherently discrete and event-driven nature of SNNs makes them suitable candidates for Approximate Computing, as small errors introduced in timing or frequency of spikes can still result in functionally correct behaviour overall. In the article "Towards spike-based machine intelligence with neuromorphic computing " written by Kaushik Roy, Akhilesh Jaiswal and Priyadarshini Panda, affiliated with Purdue University, located in West Lafayette, IN, USA, gives more prospective of this field. The paper gives us the insight about neuro- synaptic frameworks, spike based encoding and integration of algorithms and hardwires. The paper demonstrate the architecture of neuromorphic systems, particularly focusing on how neurons and synapses are replicated using silicon technology. This includes the design of circuits that can mimic the spiking behaviour of neurons and the dynamic connectivity of synapses. The research underscores the significance of temporal processing in SNNs, significantly different from traditional neural networks that rely on continuous data streams. This temporal aspect allows for more dynamic and responsive computing that better mimics human cognitive processes. The research explores various learning mechanisms for spiking neural networks (SNNs), including supervised and unsupervised learning methods that gives an upper hand on the temporal dynamics of spikes

for better performance in tasks such as pattern recognition and decision-making. One of the primary outcomes is the potential reduction in energy consumption, with neuromorphic systems designed to use energy only when processing information, similar to how neurons fire spikes only when necessary. By mimicking the brain's architecture and functionality, the research aims to advance machine intelligence, making it capable of performing complex cognitive tasks more efficiently than traditional AI systems. Kumarasinghe et al. [2021]

4.2 Advances in Spiking Neural Networks for Sensory Data Classification

In today's ever changing era, the pattern recognition is an important aspect, on which the world has already been focused on. These kind of task has been overly performed using traditional machine learning and CNN. There are a few work on pattern recognition has been performed using Spiking neural networks. One of the pattern recognition/classification research project has been conducted by the gropu of Anup Vanarse , Josafath Israel Espinosa-Ramos , Adam Osseiran , Alexander Rassau and Nikola Kasabov. In the research paper named "**Application of a Brain-Inspired Spiking Neural Network Architecture to Odor Data Classification**", researchers proposed a model using the NeuCube architecture, a 3D brain-inspired evolving connectionist system (ECOS), designed to process spatio-temporal data through spike-based representations.

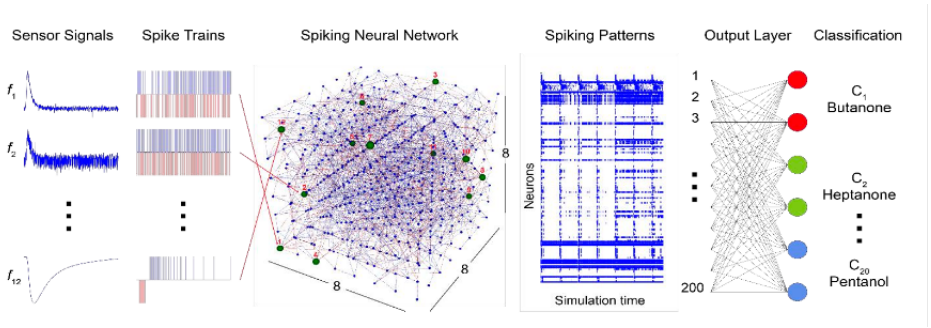

Figure 4.4

The study focuses on neuromorphic olfaction, which involves simulating the olfactory pathways of the brain to process and classify odor data. The primary objective is to introduce and validate a spiking neural network method and associated deep machine learning system for classifying odors. This approach is inspired by the brain's method of processing sensory information.

The proposed model uses the NeuCube architecture, a 3D brain-inspired evolving connectionist system (ECOS), designed to process spatio-temporal data through spike-based representations. In this model, the data-to-Spike Encoder Transforms raw sensor responses into spiking data. The Spiking Neural Network Reservoir (SNNr) learns input spike patterns using methods like spike-timing-dependent plasticity (STDP). After all the procedure, the classification module classifies patterns and evolves new output neurons to adapt to new data.

The system shows high classification accuracy and demonstrates the capability for incremental

Classification performance of the 3D SNN classifier.

No. of Classes.	Feature Set	Accuracy (SF Encoding)	Accuracy (BSA Encoding)
20	Original Signals	94.5%	79%
	Exponential Moving Averages	93%	80%
	Normalized Relative Resistance	87.5%	77%
4	Original Signals	80%	66%
	Exponential Moving Averages	84%	68%
	Normalized Relative Resistance	74%	60%

Figure 4.5

learning, adapting to new odors without retraining from scratch. In this research, different spike encoding algorithms were tested, with step-wise encoding found most effective for this particular application. From this research, Spiking neural networks gives us a promising approach for future development and gives us opportunity to explore more biologically mimic-able algorithms and implementation of these systems on neuromorphic hardware for real world applications.

Another research has been conducted by a group of researchers named Dexuan Huo, Jilin Zhang, Xinyu Dai, Pingping Zhang, Shumin Zhang, Xiao Yang, Jiachuang Wang, Mengwei Liu, Xuhui Sun, Hong Chen, on the topic of speech recognition system, detecting the pattern using Spiking neural networks. The paper "**A Bio-Inspired Spiking Neural Network with Few-Shot Class-Incremental Learning for Gas Recognition** " discusses a study on a bio-inspired spiking neural network (SNN) designed for the recognition of various gases using few-shot class-incremental learning. The study introduces a spiking neural network to recognize nine different flammable and toxic gases, showing a high accuracy rate in five-fold cross-validation. This network can be quickly retrained with new gases at a minimal accuracy cost. The research demonstrate the need to improve the accuracy and reliability of gas recognition systems, often compromised by sensor drifting and environmental factors.

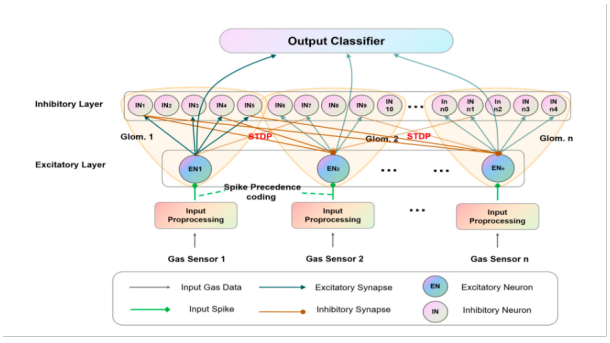

Figure 4.6

In the experiment, the researchers adopted a integrate-and-fire (IF) model to simulate the basic functions of biological neurons. The formula they used, has been shown below, which is most commonly used formula for these type of spiking neural network's biologically mimicking neurons. This research work is important as these research work is important as we will also use the leaky-integrate-and-fire model in our experiment

$$\tau \frac{dV(t)}{dt} = (V_{rest} - V(t)) + \tau_{exc} V_{in}(t) - \tau_{inh} V_{in}(t)$$

Figure 4.7

where $V(t)$ is the membrane voltage, V_{rest} is the resting membrane voltage, $V_{\text{in}}(t)$ is the changing voltage caused by input spikes, τ_{exc} and τ_{inh} are conductance constants of excitatory and inhibitory synapses, respectively, and τ is the timing constant. When the spike S_{exc} from excitatory synapses arrives at the neuron model, $\tau_{\text{exc}} V_{\text{in}}(t)$ causes an increase in the membrane voltage $V(t)$. On the contrary, when S_{inh} from inhibitory synapses arrives, $V(t)$ decreases. Once $V(t)$ reaches a membrane threshold V_{th}, the neuron model fires and $V(t)$ is set to V_{rest} immediately. Then, the neuron model is in a refractory period, in which it cannot receive any spikes from any neurons or itself spike.

This experiment demonstrates superior performance compared to traditional gas recognition algorithms such as SVM (Support Vector Machines), KNN (K-Nearest Neighbors), and conventional ANNs (Artificial Neural Networks) in terms of accuracy and adaptability. The spiking neural network achieved an accuracy of 98.75% in identifying nine types of gases each at five different concentrations. This was validated through a rigorous five-fold cross-validation process. These comparison is very much important for us as we are also going to compare our SNN model's results with the CNN model's result.

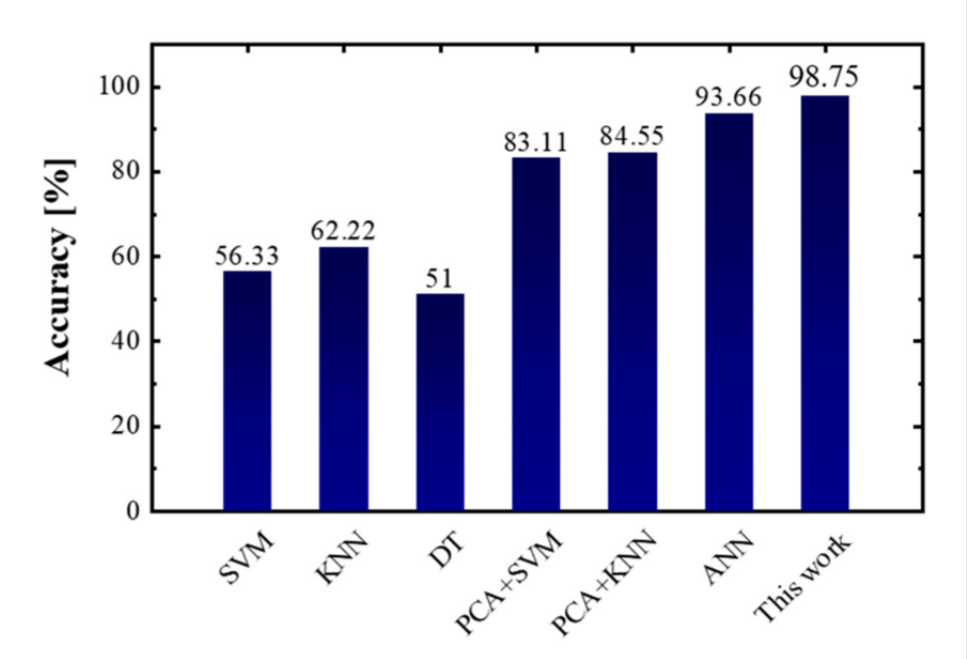

Figure 4.8

One of the standout features of the proposed SNN is its ability to incorporate new gas types with minimal degradation in performance. This is critical for applications where the gas composition

can vary unpredictably. The SNN showed a significantly reduced rate of catastrophic forgetting compared to other networks. When trained on new gases, the accuracy for previously trained gases remained stable, demonstrating the network's strength and adaptability.

The paper mentions specific improvements over other methods, citing an accuracy improvement of 5.09% compared to the next best performing traditional model. In practical tests, even when new gases were introduced into the system one by one, the network maintained a high accuracy rate, only showing minor losses (less than 2% in most cases).

The success of the SNN in this study demonstrates the potential of bio-inspired neural networks in practical applications. The ability to learn from a few samples of new gases without retraining the entire network from scratch addresses a significant limitation in current sensing technologies. Moreover, the high accuracy and adaptability of the SNN make it suitable for deployment in environments where gas compositions are complex and changing, such as industrial safety and environmental monitoring. The results not only met but exceeded the initial expectations of the study, demonstrating the effectiveness of spiking neural networks in overcoming challenges associated with traditional gas sensing technologies. The network's design, inspired by biological neural processes, offers a promising pathway toward developing more intelligent, efficient, and reliable gas detection systems in the future.

While discussing about pattern recognition, the paper named "Radar-Based Hand Gesture Recognition Using Spiking Neural Networks" shows the approach of hand gesture recognition using biologically inspired spiking neural networks. While conducting this study, researchers used frequency modulated continuous wave (FMCW) millimetre-wave radar, with a focus on a signal-to-spike conversion process that transforms radar Doppler maps into spike trains. These spike trains are then processed through a spiking recurrent neural network, specifically a liquid state machine (LSM), to achieve high accuracy in gesture recognition

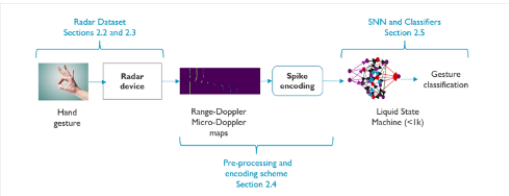

Figure 4.9

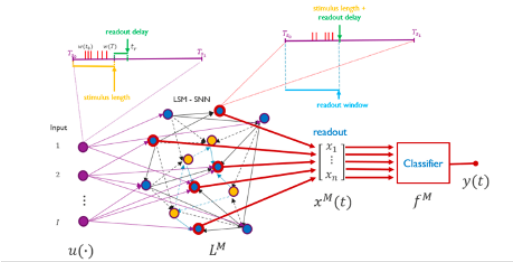

Figure 4.10: Second Image

The study emphasises a specific type of Spiking neural networks known as a Liquid State Machine (LSM), which is advantageous for its simplicity and potential for real-time processing capabilities. The primary expectation was that the LSM, with its spiking mechanism, would be able to achieve high accuracy in recognizing hand gestures from radar signals. This was based on the ability of SNNs to process time-dependent data effectively, capturing the dynamics inherent in hand movements. The LSM architecture consists of a large reservoir or "liquid" of randomly connected neurons that can dynamically respond to incoming spike trains. The neurons in the LSM are predominantly excitatory but include a substantial proportion of inhibitory neurons to regulate the network's dynamics. Unlike conventional neural networks that require extensive training across

all layers, the LSM only requires training at the readout layer. This simplifies the training process significantly and focuses on adjusting the weights that map from the high-dimensional space of the liquid's state to the output.Tsang et al. [2021]

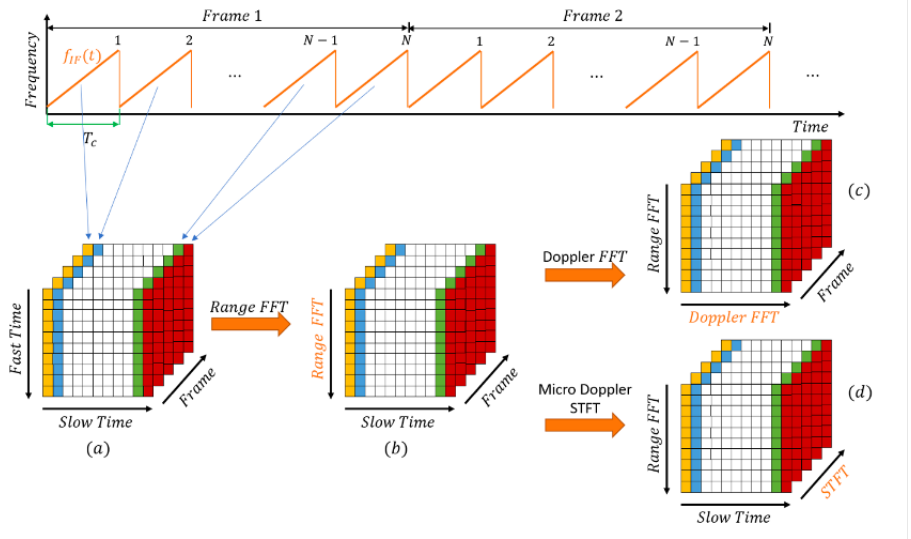

Figure 4.11

The performance of the SNN-based approach has been tested using two datasets. The Soli dataset provided by Google uses a 60 GHz radar system and includes various hand gestures. Secondly the Dop-NET Dataset focuses on different types of radar systems and also captures a variety of hand gestures.

Network/Algorithm	Best Accuracy	Dataset	Reference
Spiking LSM	**98.02**%	Soli	this work
CNN + LSTM	87.17%	Soli	[12]
Random Forest	92.1%	Soli	[3]
Spiking LSM	**98.91**%	Dop-NET	this work
SVM Quadratic	74.2%	Dop-NET	[16]
K Nearest Neighbor	87.0%	Dop-NET	[2]

Figure 4.12

The LSM achieved an accuracy of over 98% on 10-fold cross-validation for both datasets used in the study. This high accuracy demonstrates the effectiveness of the spike-based neural processing in capturing the nuances of radar-based hand gestures. The study compared the performance of the LSM with several classifiers. Logistic Regression classifier showed lower performance compared to more complex models. Random Forest provided good accuracy but still below that of the LSM with SVM. The SVM classifier, when used at the readout layer of the LSM, consistently provided the best results, enhancing the LSM's performance in classifying gesture data effectively. On the Soli dataset, the LSM achieved an accuracy of up to 98.6% in the best configurations. This was particularly notable because the Soli radar data is complex due to the high-resolution capture of

hand gestures. Similarly, on the Dop-NET dataset, which included different types of gestures and radar settings, the LSM achieved similar high accuracies, demonstrating its capabilities across different types of radar data and gesture dynamics. The study demonstrates the potential of Spiking neural networks and LSMs not just as a theoretical model but as a practical solution for real-world applications in gesture recognition technologies.

The NeuCube takes an important role to make learning structure more efficient in the case of SNN. In this case, the paper "Brain-inspired spiking neural networks for decoding and understanding muscle activity and kinematics from electroencephalography signals during hand movements" shows us some very interesting results.The study investigates Brain-Inspired Spiking Neural Networks (BI-SNN) for interpreting and predicting muscle activity and kinematics from EEG signals during hand movements. This approach offers an advanced model for decoding neural activities non-invasively and aims to provide a more interpretable and incrementally learning neural decoder for Brain-Computer Interfaces (BCIs). The primary objective of the study was to evaluate the capability of BI-SNN in predicting muscle activity and movement kinematics from EEG signals. The research integrated previous computational models with the NeuCube architecture to predict continuous muscle activity and movement kinematics during grasp-and-lift tasks.

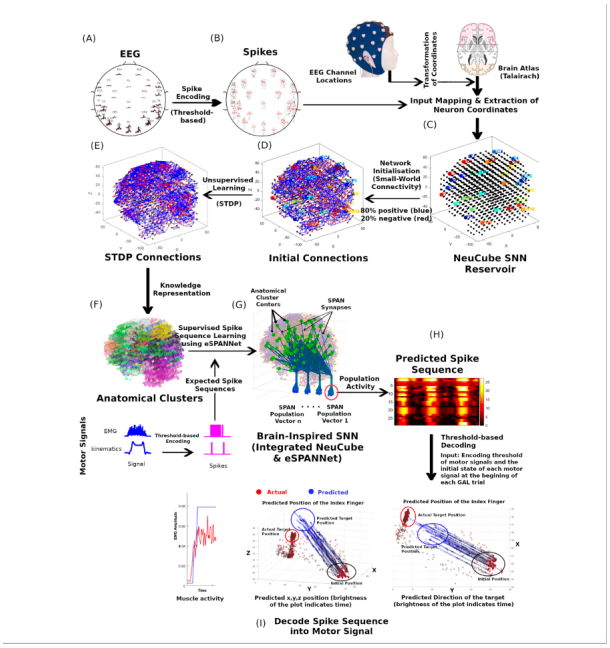

Figure 4.13

The BI-SNN model demonstrated high accuracy in predicting muscle activities and hand kinematics, outperforming existing models like the Generalized Linear Model (GLM) and eSPANNet in several metrics. BI-SNN proved its capability for real-time predictions, a critical feature for effective BCI applications. The latency measurements indicated that BI-SNN could process input data and provide output in a time frame suitable for real-time applications, suggesting its practical usability in dynamic environments. One of the significant achievements of BI-SNN was its high degree of interpretability compared to traditional methods. The model provided visual and

quantitative feedback about the brain activity related to specific movements, enabling a deeper understanding of the underlying neural processes.

The study used cross-correlation to measure the accuracy of predictions. BI-SNN frequently achieved 'high' cross-correlation coefficients ($r \geq 0.7$), suggesting a strong similarity between the predicted and actual signals. This was particularly evident in the muscle activities of the anterior deltoid, brachioradial, flexor digitorum, common extensor digitorum, and first dorsal interosseous muscles. In a comparative analysis with other models (eSPANNet and GLM), BI-SNN consistently showed superior performance. For instance, BI-SNN achieved the highest average correlation coefficients in predicting the activities of 23 out of 29 muscle signals tested, indicating its capabilities in capturing complex motor behaviours.

References:
Kumarasinghe et al. [2021]
Tsang et al. [2021]
Huo et al. [2023]
Vanarse et al. [2020]
Wang et al. [2018]
Roy et al. [2019b]
Duan et al. [2020]
Van De Burgt and Gkoupidenis [2020]
Fang et al. [2021]
Roy et al. [2019a]
Subbulakshmi Radhakrishnan et al. [2021]
Doborjeh et al. [2020]
Tee et al. [2023]
Lee et al. [2020]
Vanarse et al. [2020]
Kang et al. [2015]
Izhikevich [2003]

Chapter 5

Neural Network Models:

5.1 SNN Models

A spiking neural network is a neural network that closely mimics the biological neurons. The SNN model should have the spiking neurons, synapses and membrane potentials. To create a model that can achieve capabilities of a neuron 1st we have created a neuron model that can generate spikes.

Our SNN model consists a neuron class named as RNN_Spiking_Neuron. SNN model performs on a discrete time dependent input mechanism, not on a continuous input that works on CNN. That's why we have created an input encoder and a output decoder, to convert the training and testing data in a specific format required for performing the task.

5.1.1 Spiking neuron method(RNN_Spiking_Neuron)

In the entire model the RNN_Spiking_neuron class has adapted as the spiking neural model within the recurrent neural networks having a recurrent dynamics. The primary focus of this class is to adapt the behaviour of a spiking neuron which has dynamic properties easily distinguishable from artificial neurons.
The main parameters that this class is taking as input are computation_device, n_inputs, n_hidden, decay_multiplier, threshold, penalty_threshold. The n_imput gives a pre-defined dimensions for all the input images that are coming to the model. Secondly, the n_hidden parameter defines the number of neurons in a hidden layer. The threshold and the penalty_threshold is the mimicking factor of the membrane potential that generates the spikes when the voltage reaches the potential threshold in a spiking neural network.

```python
class RNN_Spiking_Neuron(nn.Module):

    def __init__(self, computation_device, n_inputs=64*64, n_hidden=100, decay_multiplier=0.9,
                 threshold=2.0, penalty_threshold=2.5):
        super(RNN_Spiking_Neuron, self).__init__()
        self.computation_device = computation_device
        self.n_inputs = n_inputs
        self.n_hidden = n_hidden
        self.decay_multiplier = decay_multiplier
        self.threshold = threshold
        self.penalty_threshold = penalty_threshold

        self.fc = nn.Linear(n_inputs, n_hidden)

        self.init_parameters()
        self.reset_state()
        self.to(self.computation_device)

    def init_parameters(self):
        for param in self.parameters():
            if param.dim() >= 2:
                nn.init.xavier_uniform_(param)

    def reset_state(self):
        self.prev_inner = torch.zeros([self.n_hidden]).to(self.computation_device)
        self.prev_outer = torch.zeros([self.n_hidden]).to(self.computation_device)
```

Figure 5.1

```python
def forward(self, x):

    if self.prev_inner.dim() == 1:
        # Adding batch_size dimension directly after doing a `self.reset_state()`:
        batch_size = x.shape[0]
        self.prev_inner = torch.stack(batch_size * [self.prev_inner])
        self.prev_outer = torch.stack(batch_size * [self.prev_outer])

    input_excitation = self.fc(x)
    inner_excitation = input_excitation + self.prev_inner * self.decay_multiplier
    outer_excitation = F.relu(inner_excitation - self.threshold)
    do_penalize_gate = (outer_excitation > 0).float()
    inner_excitation = inner_excitation - (self.penalty_threshold/self.threshold *
                                           inner_excitation) * do_penalize_gate
    outer_excitation = outer_excitation
    delayed_return_state = self.prev_inner
    delayed_return_output = self.prev_outer
    self.prev_inner = inner_excitation
    self.prev_outer = outer_excitation
    return delayed_return_state, delayed_return_output
```

Figure 5.2

class: RNN_Spiking_neuron

A fully connected layer (`self.fc`) has been defined to map the input features to the hidden layer. The method `init_parameters()` initializes the weights of this layer using the Xavier uniform initialization, which is commonly used for maintaining a constant variance in activation across layers. The `reset_state()` method initializes the neuron states (`prev_inner` and `prev_outer`) to zeros, and the entire module is moved to the computation device specified.

In this class, a `forward(self, x)` function has been defined. This method outlines how the data (`x`) passes through neurons and how the neuron's states are updated.

The input passes through the fully connected layer to compute initial excitation based on the current inputs. The decay multiplier has been applied to the previous internal state (`'prev_inner'`), and this decayed state is combined with the new excitation from the input. The ReLU function is applied to the difference between the inner excitation and the threshold, simulating the neuron's firing response only if the excitation exceeds the threshold. If the neuron fires (i.e., produces a positive output in `outer_excitation`), a penalty proportional to the inner excitation is subtracted from the inner state to prevent immediate subsequent firings, simulating refractory dynamics. The delayed return function returns the state and output from the previous time-step, ensuring that there is a delay, which mimics the processing time in real neuron responses.

Decay Mechanism: In this class, we have implemented a decay mechanism via the `'decay_multiplier'`. This parameter is crucial to mimic how biological neurons gradually forget their previous state (get reset over time) rather than an instant drop. This decay mechanism affects how the previous internal state of the neuron influences the current state. Inside the forward pass, the "inner_excitation" is calculated by adding the input excitation (`'input_excitation'` from `'self.fc(x)'`) to the product of `'self.prev_inner'` and `'self.decay_multiplier'`. This affects the neuron's behavior and leaves an effect of the previous state on the current state to decay.

Spiking mechanism: The spiking mechanism is the core of the spiking neural network model, and it has been implemented in the neuron model. The neuron uses a threshold (`'self.threshold'`) to determine when to fire a spike. The `'inner_excitation'` is calculated and passed through a ReLU function. We have determined the `'outer_excitation'` using the formula

`F.relu(inner_excitation - self.threshold)`. Whenever the `'inner_excitation'` exceeds `'self.threshold'`, the neuron fires; otherwise, it does not fire. Once the neuron fires, the spiking mechanism introduces a penalty threshold for decaying and reducing the `'inner_excitation'` by a factor of `'self.penalty_threshold'`. This function mimics the concept of the refractory period that takes place in a biological neuron.

This class effectively models a spiking neuron within an RNN framework, successfully demonstrating memory decay, threshold-driven firing, and refractory periods, which are true to the nature of biological neurons.

5.1.2 Input Encoder(Input_layer)

This class has been designed to process inputs from the neural network and then transform them for the subsequent layers, adding a degree of randomness to the activations. The constructor `__init__` initializes the module by calling `reset_state()` to reset the state variables. The command `self.to(self.computation_device)` moves the entire module's parameters and buffers to the GPU. After implementing the `reset_state()` method, it maintains the current state and invokes the `forward` method.

The `forward(x, is_2D=True)` method defines how input data is processed through this layer. If the incoming data (in this case, the image) is a 2D image, it flattens the 2D image in the process of input reshaping into a 1D vector using the operation `x = x.view(x.size(0), -1)`. This operation is critical for processing image data in neural networks that use fully connected layers downstream, which require 1D input vectors.

Along with that, a tensor of random values, `random_activation_perceptron`, is generated with the same shape as the flattened input. This tensor is created on the GPU, ensuring all operations stay on the same hardware. The element-wise multiplication of this random tensor with the input vector introduces variability into the activations, enhancing the agility of the data and preventing the network from overfitting in the training data.

```python
class Input_layer(nn.Module):

    def __init__(self, computation_device):
        super(Input_layer, self).__init__()
        self.computation_device = computation_device

        self.reset_state()
        self.to(self.computation_device)

    def reset_state(self):
        #     self.prev_state = torch.zeros([self.n_hidden]).to(self.computation_device)
        pass

    def forward(self, x, is_2D=True):
        x = x.view(x.size(0), -1)  # Flatten 2D image to 1D for FC
        random_activation_perceptron = torch.rand(x.shape).to(self.computation_device)
        return random_activation_perceptron * x
```

Figure 5.3: Input Layer

By returning the product of the random activations and the input data, the Input_layer effectively adds stochasticity to the neural network's input stage.

5.1.3 Output Decoder(Output_layer)

```
class Output_layer(nn.Module):

    def __init__(self, average_output=True):

        super(Output_layer, self).__init__()
        if average_output:
            self.reducer = lambda x, dim: x.sum(dim=dim)
        else:
            self.reducer = lambda x, dim: x.mean(dim=dim)

    def forward(self, x):
        if type(x) == list:
            x = torch.stack(x)
        return self.reducer(x, 0)
```

Figure 5.4: Output Layer

This class has been designed to process outputs from the neural network and then transform them for the subsequent outputs needed. The constructor `__Init__` initializes the output layer with the parameter named 'average-output'. This parameter determines how the outputs passed through the layer will be aggregated. If `average_output` is set to `True`, the outputs are summed across a specified dimension, which can be useful for models where the total activation is more informative than the average. If `average_output` is `False`, it computes the mean across the specified dimension, which can help in maintaining numerical stability of the gradients, especially when integrating with regular neural networks that might be sensitive to the scale of output values. Based on the `average_output` flag, a lambda function `self.reducer` has been assigned to perform the appropriate reduction operation.

5.1.4 The Spiking Neural Network Module(Pulse_Network)

The `Pulse_network` class is a simulation of a connected layer of neural networks with spiking neuron dynamics over discrete time steps. This class contains the connected neuron layers that mimics the behaviour of the neurons in a biological system, using recurrent connections and threshold based activation. The `__Init__` constructor initialises the network with these computation_device, n_time_steps, begin_eval components. The n_time_steps defines the number of time steps over which the network operates, reflecting the temporal dynamics of input processing. The begin_eval, indicates the start time step from which outputs are considered for final processing, allowing the network to ignore initial transient states. We get the value of the n_time_steps and begin_eval values when we define the model.

The network architecture includes the Input_layer and Output_layer that has been described above. Initially we will discuss about two layers which has been defined in this model. But later on based on this model, another models has also been created that contains 3 and 4 layers. The layer1 and

```
class Pulse_Network(nn.Module):
    def __init__(self, computation_device, n_time_steps, begin_eval):
        super(Pulse_Network, self).__init__()
        assert (0 <= begin_eval and begin_eval < n_time_steps)
        self.computation_device = computation_device
        self.n_time_steps = n_time_steps
        self.begin_eval = begin_eval

        self.input_conversion = Input_layer(computation_device)

        self.layer1 = RNN_Spiking_Neuron(
            computation_device, n_inputs=64*64*3, n_hidden=100,
            decay_multiplier=0.9, threshold=1.0, penalty_threshold=1.5
        )

        self.layer2 = RNN_Spiking_Neuron(
            computation_device, n_inputs=100, n_hidden=10,
            decay_multiplier=0.9, threshold=1.0, penalty_threshold=1.5
        )

        self.output_conversion = Output_layer(average_output=False)

        self.to(self.computation_device)
```

Figure 5.5: SNN module

layer2 calls the neuron model that has been created named 'RNN_Spiking_Neuron' that inherits the working procedure of a biological neuron. Along with that the layers gives proper configurations that are required for the neuron model. Computation_device(GPU), n_inputs=64*64*3, n_hidden=100, decay_multiplier=0.9, threshold=1.0, penalty_threshold=1.5 has been defined in this layer while calling the RNN_Spiking_Neuron' model.

```
def forward_through_time(self, x):

    self.input_conversion.reset_state()
    self.layer1.reset_state()
    self.layer2.reset_state()

    out = []

    all_layer1_states = []
    all_layer1_outputs = []
    all_layer2_states = []
    all_layer2_outputs = []
    for _ in range(self.n_time_steps):
        xi = self.input_conversion(x)

        layer1_state, layer1_output = self.layer1(xi)

        layer2_state, layer2_output = self.layer2(layer1_output)

        all_layer1_states.append(layer1_state)
        all_layer1_outputs.append(layer1_output)
        all_layer2_states.append(layer2_state)
        all_layer2_outputs.append(layer2_output)
        out.append(layer2_state)

    out = self.output_conversion(out[self.begin_eval:])
    return out, [[all_layer1_states, all_layer1_outputs], [all_layer2_states, all_layer2_outputs]]
```

Figure 5.6: Processing data through a neural network

The 'forward_through_time(x)' method simulates the network's response over the specified number of time steps. It resets the state of the input layer and each spiking neuron layer before looping through each time step. The input is processed by the 'input_conversion'. Each spiking neuron

layer process the output from the previous layer. The internal state and the output of each layer has been stored for visualisation and analysis. From the ”begin_eval” time step onwards, outputs are compiled using the output_conversion process.

Using this method we return the aggregated output along with the states and outputs of each layer for all time steps, providing comprehensive internal dynamics.

```
def forward(self, x):
    out, _ = self.forward_through_time(x)
    return F.log_softmax(out, dim=-1)
```

Figure 5.7: forward function

The forward(self, x) method is the standard forward pass method that only returns the final output of the network. It uses forward_through_time to compute the outputs across all time steps and applies the log SoftMax function for classification purposes.

5.1.5 Visualization Module

The `visualize_neuron(x, layer_idx, neuron_idx)` method focuses on the temporal activity of a specific neuron, showing how its internal state and output change over time. Both methods use the functions `plot_layer` and `plot_neuron` to visualize. Using these functions, we have generated the plots that demonstrate the computational processes within the network. The spikes shows the intensity of the information has been transmitted form input layer to output layer. This spiking patterns will help us to understand the characteristics of the model and different complexities of different models where we will play along with spiking neuron layers.

```
def visualize_neuron(self, x, layer_idx, neuron_idx):
    assert x.shape[0] == 1 and len(x.shape) == 4, (
        "Pass only 1 example to Pulse_Network.visualize(x) with outer dimension shape of 1.")
    _, layers_state = self.forward_through_time(x)

    all_layer_states, all_layer_outputs = layers_state[layer_idx]

    layer_state = torch.stack(all_layer_states).data.cpu().numpy().squeeze().transpose()
    layer_output = torch.stack(all_layer_outputs).data.cpu().numpy().squeeze().transpose()

    self.plot_neuron(layer_state[neuron_idx], title=f"Inner state values of neuron {neuron_idx} of layer {layer_idx+1}")
    self.plot_neuron(layer_output[neuron_idx], title=f"Output spikes (activation) values of neuron {neuron_idx} of layer {layer_idx+1}")
```

Figure 5.8: Visualization Module

5.2 Model Complexity of SNN

The complexity of the spiking neural networks model lies in the layers of the model. We have already defined the neuron architecture in the `RNN_Spiking_Neuron` module. Along with that, we have also defined the Spiking neural network model as the `Pulse_Network` module. To understand the complexity of the neural network, we need to evaluate it on the basis of interconnected

layers.

5.2.1 2-layer model

The two layer model is one on the simplest but way efficient model in terms of complexity. The model consists an input layer, followed by RNN Spiking neuron layer, and the output layer.

```
self.input_conversion = Input_layer(computation_device)

self.layer1 = RNN_Spiking_Neuron(
    computation_device, n_inputs=64*64*3, n_hidden=100,
    decay_multiplier=0.9, threshold=1.0, penalty_threshold=1.5
)

self.layer2 = RNN_Spiking_Neuron(
    computation_device, n_inputs=100, n_hidden=10,
    decay_multiplier=0.9, threshold=1.0, penalty_threshold=1.5
)

self.output_conversion = Output_layer(average_output=False)
```

Figure 5.9: 2-layer model complexity

The input x to the network is first processed by the `self.input_conversion` layer. This layer is responsible for converting the raw input data into a form suitable for the spiking neurons. After processing, the converted input is then passed directly to the first layer of spiking neurons (`self.layer1`). This layer receives the pre-processed input data as its input at each timestep of the network's operation. In `self.layer1`, each neuron processes the input it receives based on its current state and parameters like decay multiplier and threshold. The output from this layer is twofold: the updated internal states of the neurons and their output spikes. The output spikes are those neuron responses where the input strength surpasses the neuron's firing threshold.

```
def forward_through_time(self, x):

    self.input_conversion.reset_state()
    self.layer1.reset_state()
    self.layer2.reset_state()

    out = []

    all_layer1_states = []
    all_layer1_outputs = []
    all_layer2_states = []
    all_layer2_outputs = []
    for _ in range(self.n_time_steps):
        xi = self.input_conversion(x)
        layer1_state, layer1_output = self.layer1(xi)
        layer2_state, layer2_output = self.layer2(layer1_output)
        all_layer1_states.append(layer1_state)
        all_layer1_outputs.append(layer1_output)
        all_layer2_states.append(layer2_state)
        all_layer2_outputs.append(layer2_output)
        out.append(layer2_state)
    out = self.output_conversion(out[self.begin_eval:])
    return out, [[all_layer1_states, all_layer1_outputs], [all_layer2_states, all_layer2_outputs]]
```

Figure 5.10: inter connectivity of 2 spiking neuron layers

The outputs (spikes) from the first spiking neuron layer are then forwarded as the input to the second spiking neuron layer (`self.layer2`). Thus, `self.layer2` uses the spike outputs of `self.layer1` as its input. Similar to `self.layer1`, `self.layer2` processes the spike signals from the first layer under its own set of neuronal dynamics parameters. It too generates an output consisting of the neuron states and their respective spikes. The outputs selected from `self.layer2` are then passed to `self.output_conversion`. This layer converts the series of output spikes into a final output format.

5.2.2 3-layer model

We have created a 3-layer model named `Pulse_Network_3layer`. Similar to the 2-layer model, the 3-layer model also receives the converted spike input from the input layer, and this processed data from the input conversion layer is then passed directly to `self.layer1`. Each time-step's processed output from the input layer becomes the input for `self.layer1`. The spiking output from `self.layer1` (not the internal states) is then fed into `self.layer2` as its input. This progression allows the network to build a deeper understanding of the input data as it moves through the layers, where each layer's output can be seen as a more refined or abstract representation of the input.

```
self.input_conversion = Input_layer(computation_device)

self.layer1 = RNN_Spiking_Neuron(
    computation_device, n_inputs=64*64*3, n_hidden=100,
    decay_multiplier=0.9, threshold=1.0, penalty_threshold=1.5
)

self.layer2 = RNN_Spiking_Neuron(
    computation_device, n_inputs=100, n_hidden=50,
    decay_multiplier=0.9, threshold=1.0, penalty_threshold=1.5
)


self.layer3 = RNN_Spiking_Neuron(
    computation_device, n_inputs=50, n_hidden=10,
    decay_multiplier=0.9, threshold=1.0, penalty_threshold=1.5
)

self.output_conversion = Output_layer(average_output=False)

self.to(self.computation_device)
```

Figure 5.11: 3-layer model complexity

```
all_layer1_states = []
all_layer1_outputs = []
all_layer2_states = []
all_layer2_outputs = []
all_layer3_states = []
all_layer3_outputs = []
for _ in range(self.n_time_steps):
    xi = self.input_conversion(x)
    layer1_state, layer1_output = self.layer1(xi)
    layer2_state, layer2_output = self.layer2(layer1_output)
    all_layer1_states.append(layer1_state)
    all_layer1_outputs.append(layer1_output)
    all_layer2_states.append(layer2_state)
    all_layer2_outputs.append(layer2_output)
    all_layer3_states.append(layer3_state)
    all_layer3_outputs.append(layer3_output)
    out.append(layer3_state)

out = self.output_conversion(out[self.begin_eval:])
return out, [[all_layer1_states, all_layer1_outputs], [all_layer2_states, all_layer2_outputs],
             [all_layer3_states, all_layer3_outputs]]
```

Figure 5.12: inter connectivity of 3 spiking neuron layers

Similar to `self.layer1`, `self.layer2` processes its received inputs, affecting its internal states and producing spiking outputs based on its thresholds and decay settings. The spiking output from `self.layer2` serves as the input to `self.layer3`. `self.layer3` processes the inputs from `self.layer2`, updating its neuron states and generating spikes according to its configuration. This layer, being closer to the output, plays a crucial role in determining the final decision or classification performed by the network.

After all time-steps are processed, only the outputs from `self.layer3` starting from the time-step `self.begin_eval` are considered for final output processing. The collected outputs from `self.layer3` are then passed to `self.output_conversion`, which formats them into the final output structure (`Output_layer`).

5.2.3 4-layer model

Similar to the above 2 and 3 layer models, we have created a 4-layer model. The working procedure is similar to the previous models. Like the above networks, this model also receives the input data from the `input_layer` in the form of spikes. After passing through layers 1, 2, and 3, the spikes ultimately reach layer 4. In this layer, the incoming data (spikes from `self.layer3`) is processed to update neuron states and generate outputs, which then serve as inputs to `self.layer4`.

`self.layer4` processes the incoming spikes and updates its states accordingly, producing the final set of output spikes for the network.

The outputs from `self.layer4`, specifically the spike data produced after `self.begin_eval`

```
self.input_conversion = Input_layer(computation_device)

self.layer1 = RNN_Spiking_Neuron(
    computation_device, n_inputs=64*64*3, n_hidden=100,
    decay_multiplier=0.9, threshold=1.0, penalty_threshold=1.5
)

self.layer2 = RNN_Spiking_Neuron(
    computation_device, n_inputs=100, n_hidden=50,
    decay_multiplier=0.9, threshold=1.0, penalty_threshold=1.5
)

self.layer3 = RNN_Spiking_Neuron(
    computation_device, n_inputs=50, n_hidden=30,
    decay_multiplier=0.9, threshold=1.0, penalty_threshold=1.5
)

self.layer4 = RNN_Spiking_Neuron(
    computation_device, n_inputs=30, n_hidden=10,
    decay_multiplier=0.9, threshold=1.0, penalty_threshold=1.5
)

self.output_conversion = Output_layer(average_output=False)

self.to(self.computation_device)
```

Figure 5.13: 4-layer model complexity

```
self.input_conversion.reset_state()
self.layer1.reset_state()
self.layer2.reset_state()
self.layer3.reset_state()
self.layer4.reset_state()
out = []
all_layer1_states = []
all_layer1_outputs = []
all_layer2_states = []
all_layer2_outputs = []
all_layer3_states = []
all_layer3_outputs = []
all_layer4_states = []
all_layer4_outputs = []
for _ in range(self.n_time_steps):
    xi = self.input_conversion(x)
    layer1_state, layer1_output = self.layer1(xi)
    layer2_state, layer2_output = self.layer2(layer1_output)
    layer3_state, layer3_output = self.layer3(layer2_output)
    layer4_state, layer4_output = self.layer4(layer3_output)
    all_layer1_states.append(layer1_state)
    all_layer1_outputs.append(layer1_output)
    all_layer2_states.append(layer2_state)
    all_layer2_outputs.append(layer2_output)
    all_layer3_states.append(layer3_state)
    all_layer3_outputs.append(layer3_output)
    all_layer4_states.append(layer4_state)
    all_layer4_outputs.append(layer4_output)
    out.append(layer4_output)
out = self.output_conversion(out[self.begin_eval:])
return out, [[all_layer1_states, all_layer1_outputs], [all_layer2_states, all_layer2_outputs],
             [all_layer3_states, all_layer3_outputs], [all_layer4_states, all_layer4_outputs]]
```

Figure 5.14: inter connectivity of 4 spiking neuron layers

(allowing for a warm-up period), are sent to the `self.output_conversion` layer. This layer converts these spike patterns into a form suitable for output, such as by applying a softmax function for classification tasks.

5.3 The Work Flow of SNN module

To train and evaluate the model I have created different functions.

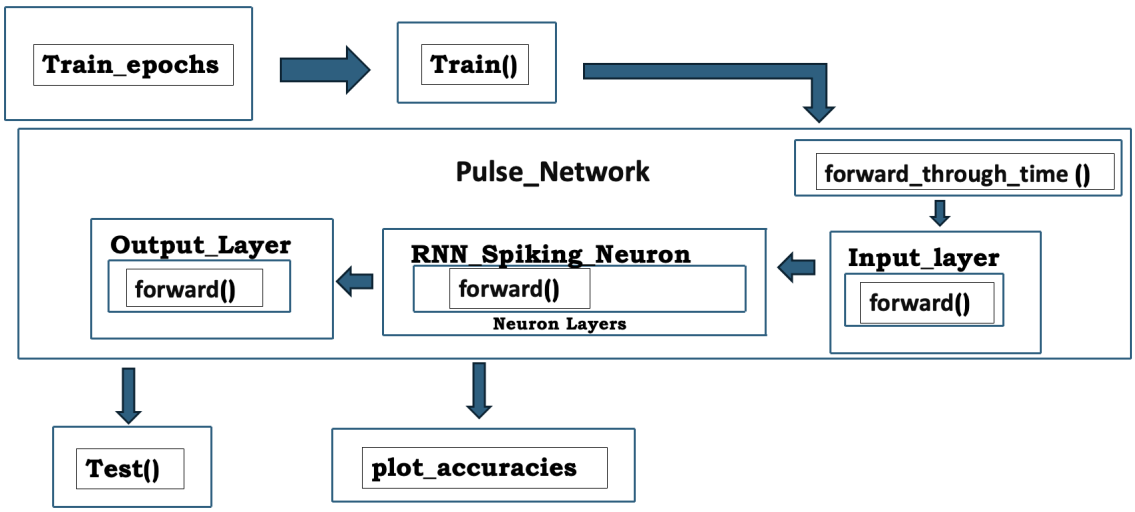

Figure 5.15: Workflow of the SNN

Train Neural Network for One Epoch: The function handles the training process for one epoch at a time. The function accepts several parameters, like the model, the computing device (I have connected a GPU), a data loader for the training set, an optimizer (an algorithm for updating

```python
def train(model, device, train_set_loader, optimizer, epoch, logging_interval=100):
    model.train()
    accuracies = []

    for batch_idx, (data, target) in enumerate(train_set_loader):
        data, target = data.to(device), target.to(device)
        optimizer.zero_grad()
        output = model(data)
        loss = F.nll_loss(output, target)
        loss.backward()
        optimizer.step()

        if batch_idx % logging_interval == 0:
            pred = output.max(1, keepdim=True)[1]
            correct = pred.eq(target.view_as(pred)).float().mean().item()
            accuracies.append(correct)
            print('Train Epoch: {} [{}/{} ({:.0f}%)] Loss: {:.6f} Accuracy: {:.2f}%'.format(
                epoch, batch_idx * len(data), len(train_set_loader.dataset),
                100. * batch_idx / len(train_set_loader), loss.item(),
                100. * correct))

    return accuracies
```

Figure 5.16: The Train module

synaptic weights), the current epoch number, and the logging interval for outputting progress.

The `train` function manages the training of the neural network model for a given epoch. It begins by setting the model and starting over the training dataset using the data loader. Along with that, this function also collects accuracy for each step of the training for further use. The `train` function repeats the process until all the epochs are performed. After completing all the epochs, it aggregates the collected accuracy metrics and displays them using the `plot accuracies` function, which demonstrates the model's learning process over time.

```python
def train_epochs(model):
    epoch = 1
    optimizer = optim.SGD(model.parameters(), lr=0.1, momentum=0.5)
    accuracies = train(model, device, train_set_loader, optimizer, epoch, logging_interval=10)
    test(model, device, test_set_loader)

    epoch = 2
    optimizer = optim.SGD(model.parameters(), lr=0.05, momentum=0.5)
    accuracies += train(model, device, train_set_loader, optimizer, epoch, logging_interval=10)
    test(model, device, test_set_loader)

    epoch = 3
    optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.5)
    accuracies += train(model, device, train_set_loader, optimizer, epoch, logging_interval=10)
    test(model, device, test_set_loader)

    plot_accuracies(accuracies)
```

Figure 5.17: The Train$_e$*pochsmodule*

Multi-Epoch Training with Adaptive Learning Rates: This function initiates the whole training process and organizes the entire procedure by performing multiple epochs, altering learning rates, and testing after every epoch. The function begins with setting up the optimizer with an initial learning rate. We call the `train` function for every epoch and collect the accuracy metrics. After training, we evaluate the model using the `test` function. This cycle is repeated for every epoch with adjusted learning rates, to achieve a good training process and to improve the model performance.

Testing function: The purpose of the `test` function is to evaluate the spiking neural networks on the testing data. The function accepts the model, the computing device (GPU), and a data loader for the testing dataset to initiate the process. During the model evaluation mode, a function is

initialized by using `model.eval()`. This function specifically turns off behaviors typical to training, such as dropout and batch normalization, ensuring that the model's performance is measured accurately and consistently.
After which, the model initialises variables to accumulate the total loss(test_loss), and give the count of correctly predicted samples.

```
def test(model, device, test_set_loader):
    model.eval()
    test_loss = 0
    correct = 0
    all_preds = []
    all_targets = []

    with torch.no_grad():
        for data, target in test_set_loader:
            data, target = data.to(device), target.to(device)
            output = model(data)
            test_loss += F.nll_loss(output, target, reduction='sum').item()
            pred = output.argmax(dim=1, keepdim=True)
            correct += pred.eq(target.view_as(pred)).sum().item()

            all_preds.extend(pred.view(-1).cpu().numpy())
            all_targets.extend(target.cpu().numpy())

    test_loss /= len(test_set_loader.dataset)

    conf_matrix = confusion_matrix(all_targets, all_preds)
    accuracy = accuracy_score(all_targets, all_preds)

    print('\nTest set: Average loss: {:.4f}, Accuracy: {}/{} ({:.2f}%)\n'.format(
        test_loss, correct, len(test_set_loader.dataset),
        100. * correct / len(test_set_loader.dataset)))

    return conf_matrix, accuracy
```

Figure 5.18: The Test module

The next step is to store all the prediction and the corresponding targets as variables in all_preds and all_targets. Within the model, the gradient computational is disabled, which helps in reducing the memory usage and speeds up the process, as these gradients are not needed for model evaluation.

After configuring the model based on the above details, it will go into the inference part where in the test dataset will be in inferred over a for loop. The output of the inferences, that is data and target, will be stored for loss calculation, prediction and accuracy. There are separate functions built to calculate the evaluation metrics. At the end, the final calculations will include the average loss, accuracy, confusion matrix and accuracy score.

5.3.1 The Data Flow

To understand the working procedure of the spiking neural networks model, it is also very much important to understand the shape and structure of the data in each layer.

Here's a breakdown of the data shapes at each step as the data flows through the network and related functions:

Pulse_Network (forward_through_time): This function represents the initial step of the spiking neural network module, which receives x (the data) with shape $[\text{batch\_size}, 3, 64, 64]$.

Input layer: The input tensor x with shape $[\text{batch\_size}, 3, 64, 64]$ is fed into the input layer. Here, the data consists of images with 3 color channels, each of 64x64 pixels resolution. The input layer transforms this data into the shape $[\text{batch\_size}, 64 \times 64 \times 3]$ by flattening the data. This flattening process reshapes the multi-dimensional array into a single-dimensional vector per batch instance, which simplifies the structure for processing in subsequent neural network layers.

RNN_Spiking_Neuron (Layer 1 Forward): In this layer, the data coming from the input layer will go in and then it will produce two outputs (internal state and output state), both having a shape of $[\text{batch\_size}, n_{\text{hidden}}]$ where n_{hidden} is the number of hidden units (100 for layer1).

RNN_Spiking_Neuron (Layer 2 Forward): After the first layer, the data approaches the next layer (internal neuron layer or output neuron layer depending on the complexity). Two outputs will again be generated by this layer, both with the shape $[\text{batch\_size}, n_{\text{hidden}}]$ where n_{hidden} is the number of hidden units (10 for layer 2).

Output Layer (Forward): After coming out from the neuron layer, the data will approach the output layer (decoder). In this layer, the input will be a list of states from layer2 starting from the `begin_eval` index, each with shape $[\text{batch\_size}, 10]$. The decoder will convert it into a single tensor combined from the list, reduced along the specified dimension, resulting in a shape $[10]$ using sum reduction.

Output Layer (Forward): After coming out from the neuron layer, the data will approach the output layer (decoder). In this layer, the input will be a list of states from layer2 starting from the `begin_eval` index, each with shape $[\text{batch\_size}, 10]$. The decoder will convert it into a single tensor combined from the list, reduced along the specified dimension, resulting in a shape $[10]$ using sum reduction.

After that the data will be utilised after converting into proper order by the visualization module.

5.4 The CNN module

:

We have created a CNN model to perform the same binary and multi class classification task to compare the results in between the CNN and SNN. This will give us a better prospective.

In our CNN model, initially we have a convolution layer having 12 filters each with a kernel size of 5 X 5. We have implemented this layer to extract initial features from the input image. The ReLu activation function is applied to introduce non-linearities into the model, helping it learn complex pattern. Following that, we have a maxpolling layer to reduce the spatial dimensions of

```
model = Sequential()
model.add(Conv2D(filters=12, kernel_size=(5, 5), activation='relu', input_shape=Input_shape))
model.add(MaxPooling2D())

model.add(Conv2D(filters=8, kernel_size=(5, 5), activation='relu'))
model.add(Conv2D(filters=4, kernel_size=(5, 5), activation='relu'))
model.add(Dropout(0.3))

model.add(AveragePooling2D())
model.add(Flatten())
model.add(Dense(units=16, activation='relu'))
model.add(Dense(units=4, activation='softmax'))
```

Figure 5.19: The CNN model

the output of the previous layer.

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 220, 220, 12)      912

 max_pooling2d (MaxPooling2  (None, 110, 110, 12)      0
 D)

 conv2d_1 (Conv2D)           (None, 106, 106, 8)       2408

 conv2d_2 (Conv2D)           (None, 102, 102, 4)       804

 dropout (Dropout)           (None, 102, 102, 4)       0

 average_pooling2d (Average  (None, 51, 51, 4)         0
 Pooling2D)

 flatten (Flatten)           (None, 10404)             0

 dense (Dense)               (None, 16)                166480

 dense_1 (Dense)             (None, 4)                 68

=================================================================
Total params: 170672 (666.69 KB)
Trainable params: 170672 (666.69 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Figure 5.20: Sequential Model Architecture Summary

Next we have a second convolution layer having 4 filters with each 5 X 5 kernel size and uses ReLu activation function. Along with that we have a Dropout Layer to resolve the issue of over-fitting. Similarly like maxpolling layer, we have also used an average pooling layer. The output from the pooling layer having a multi dimension tensor goes to the flattening layer and transformed to the one dimension array. Then we can see our first Dense layer. This fully connected layer has 16 units and uses ReLu activation. It interprets and flattens the features extracted pooled from the images. The final layer is a dense layer and the number of units in this layer will be dependent on what type of classification we are performing. In the case of binary classification, it will have 2 units and in the case of multi class classifications, it will have 4 units.

Chapter 6

Evaluation

6.1 The Experiment

We have performed different types of experiments to understand and analyse the characteristics of spiking neural networks. To understand how the spiking neural networks model works in different situations, we have performed binary classification and multi class classifications. One of the main reason to use spiking neural networks is to check its adaptability in situations where the data is limited. To check that, we have performed two different types of binary classifications. We have used the same type of model once to classify a huge amount of data and next time to classify a very small amount of data. Along with that we will analyse the performance of the models in two cases, by using accuracy in training and testing cases. Similarly we will perform the multi class classification and check the accuracies of training and testing. Along with that we have the confusion matrix that will give use more insights about the model and the accuracies. We will perform a detailed comparison using accuracy graphs and confusion matrixes in between the binary classifications and multi class classifications. We will also compare between the accuracies and the results when we use the small and large datasets. Then we will perform the binary and multi class classifications using CNN and compare the results between and CNN and SNN. These comparisons will take us to an conclusion and draw a understanding, in which environment a the SNN works better.

6.2 The Dataset

In this project we have used two different type of dataset for three different type of experiment using Spiking neural network model. For Binary classification, a dataset has been used which has two different classes. This dataset has 10000 images for each class and segregated properly for train the model.

For visualisation, I have created a function name show_images to display the training and testing data. We can see the images labeled properly in this graph.

```python
import matplotlib.pyplot as plt

def show_images(dataloader):

    for images, labels in dataloader:
        plt.figure(figsize=(10, 10))
        for i in range(10, 20):
            plt.subplot(1, 10, i - 9)
            plt.imshow(images[i].permute(1, 2, 0).cpu().numpy())
            plt.title(labels[i].item())
            plt.axis('off')
        plt.show()
        break

# Display the first 10 images from the training set
print("Training Data:")
show_images(train_set_loader)

# Display the first 10 images from the testing set
print("Testing Data:")
show_images(test_set_loader)
```

Figure 6.1

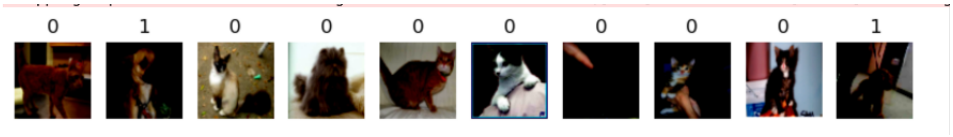

Figure 6.2: Training Data for Binary classification

Figure 6.3: Testing Data for Binary classification

In this above images we can see the training and testing data of the binary classification model has been displayed.

For our experimental purpose we are performing 2 different type of binary classification. A binary classification with a large dataset and a binary classification with a small dataset.

Binary Classification with Large Dataset: In this experiment we are using the a dataset where we have data categorised in Dogs and Cats. The we kept the dataset inside a folder named ‘Data’. Inside the ‘Data’ folder, we can find two categories, ‘Dogs’ and ‘Cats’. Each folder contains 10000 images of each category, segregated in a proper manner.

Binary Classification with small dataset: In this task we are using a modified dataset that we have used in the above task. In this task we are using the dataset that has images segregated properly in between ‘Cats’ and ‘Dogs’ having 1/10th data from the above dataset. The data has been stored inside a folder names ‘S_data’. Inside that we can find the two folders named ‘Cats’ and ‘Dogs’. Inside each folder, we will able to find 1000 images from each category.

Secondly I am using the same dataset but this time there are very limited number of images for the

binary classification. The purpose of using this small dataset is to train the spiking neural network model in this small dataset and try to demonstrate the adaptability of the model and compare the performance of the network.

There is another dataset that I am also using for multi class classification. The dataset has been taken from Kaggle, known as Satellite images for classification. The dataset consists 4 different type of images. The 4 different folder named “cloudy”, “desert”, “green_area”, and “water”. The dataset is not in large in numbers but it is still good enough to make a classification project work.

Figure 6.4: Training Data for multi class classification

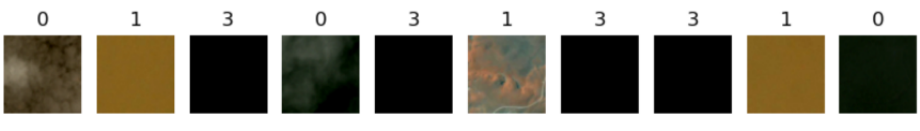

Figure 6.5: Testing Data for multi class classification

In the above images we can see the training and testing dataset has been displayed. In the above images, we can see the data set has been segregated properly from 0 to 3 having 4 different classes. 0 has been allocated to the “Cloudy”, 1 has been assigned to the “dessert”, 2 has been assigned to the “green_area” and the 3 has been assigned to the “water”.

6.3 Experiment Results

6.3.1 Binary Classification using SNN

In the binary classification we are using a Dog vs Cat dataset from Kaggle. For comparison and check one of the fundamental advantage of the spiking neural network, we have used this dataset twice. 1st we have trained the model with the large dataset, and check how the model is performing. Later on we have also trained the models with 1/10th amount data of that dataset and compared the result with the results of the model where we have used the large dataset.

Binary Classification using large dataset:

For this experiment, we have used three different models with varying intensities: a 2-layer model, a 3-layer model, and a 4-layer model, which are instantiated as spiking_model_2layers, spiking_model_3layers, and spiking_model_4layers, respectively.”

2-Layer Model: 1st we train the model by calling the train_epochs function and passing the spiking_model_2layers as an argument.”

```
spiking_model_2layers = Pulse_Network(device, n_time_steps=128, begin_eval=0)
train_epochs(spiking_model_2layers)
```

Figure 6.6: Training the data using 2-layered SNN model

The model trains on the training data, with multiple epochs and gives us the accuracy of the model.

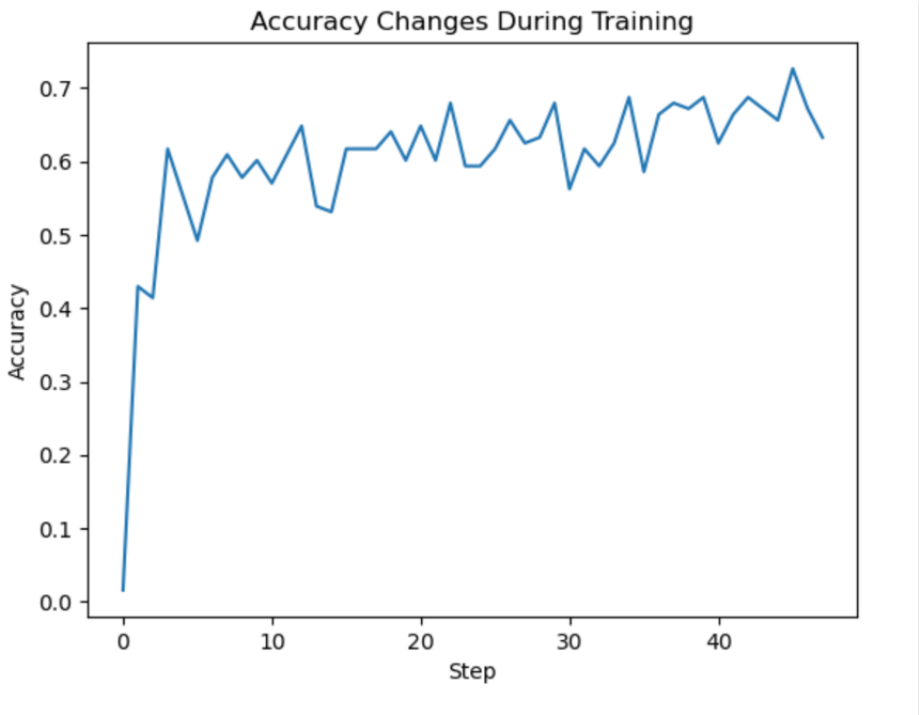

Figure 6.7

The above graph shows how the accuracy changes on each step of the training. From this graph we can see the accuracy changes through training, and gets a saturation at some point of 65%. A better visualisation of the accuracy will be determined when we will see the confusion metrics. To demonstrate the spiking patterns of the model while training the data, we have initialised the visualize_neuron function.

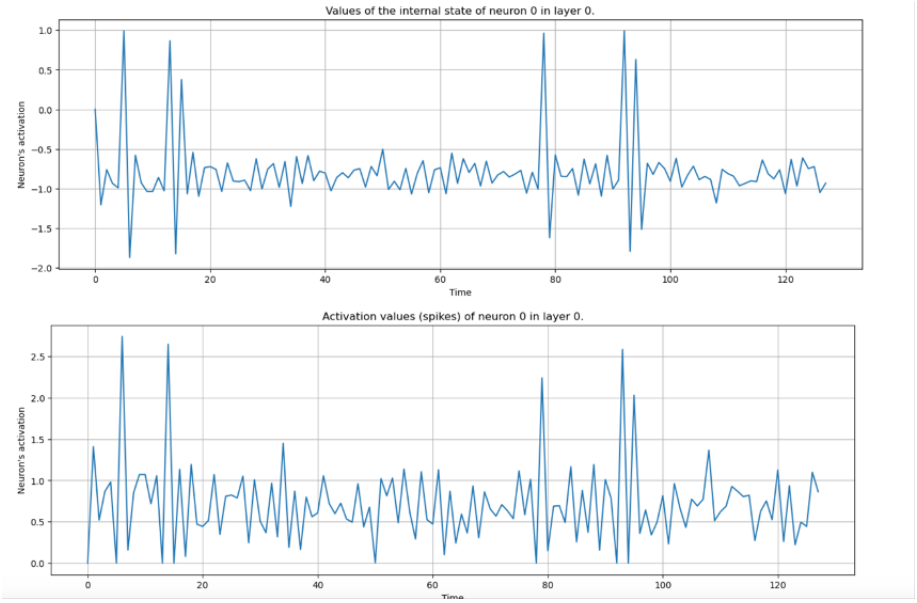

Figure 6.8

This graph shows the huge amount of information passing through the neuron's internal state over time. It's a trace of the neuron's membrane potential, which fluctuates based on incoming spikes and the neuron's own leaky properties (decay). Sharp peaks indicate moments when the potential reached a threshold, typically leading to the neuron firing. This shows the characteristics of a leaky integrate and fire model.

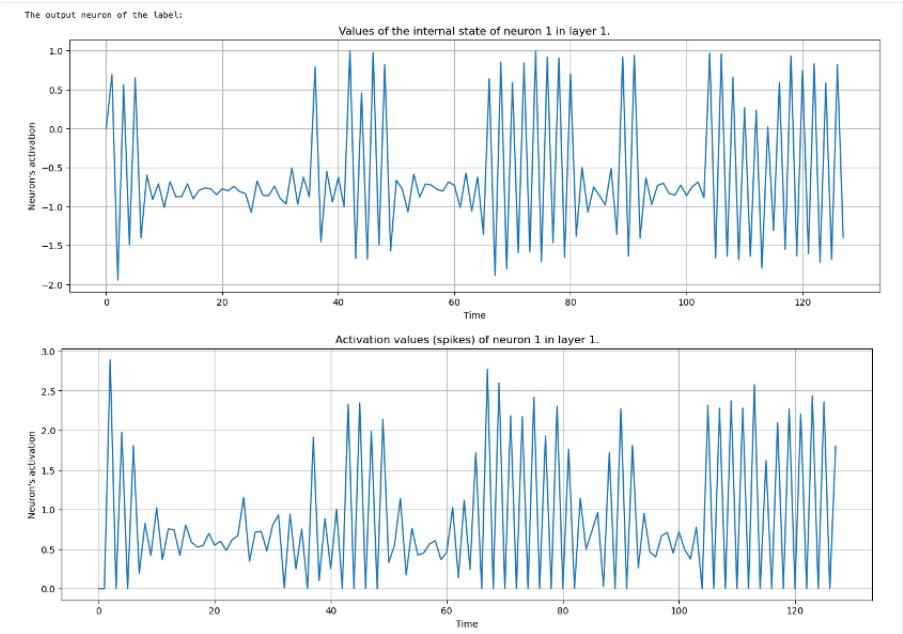

Figure 6.9

This output neurons spiking shows the neuron's activation values, represented as spikes. Each spike represents a moment when the neuron's internal state exceeded a certain threshold, causing it to "fire." These spikes are binary events in time, indicating the neuron's response to its input.

3-Layer Model: To work and evaluate the 3 layer network, we have executed the code below. This model has 3 connected neuron layers.

```
spiking_model_3layers = Pulse_Network_3layer(device, n_time_steps=128, begin_eval=0)
train_epochs(spiking_model_3layers)
```

Figure 6.10

The training process starts with training the epochs. While training, the model's training accuracy gets stored on every step and form the graph below, we can interpret insights of the model.

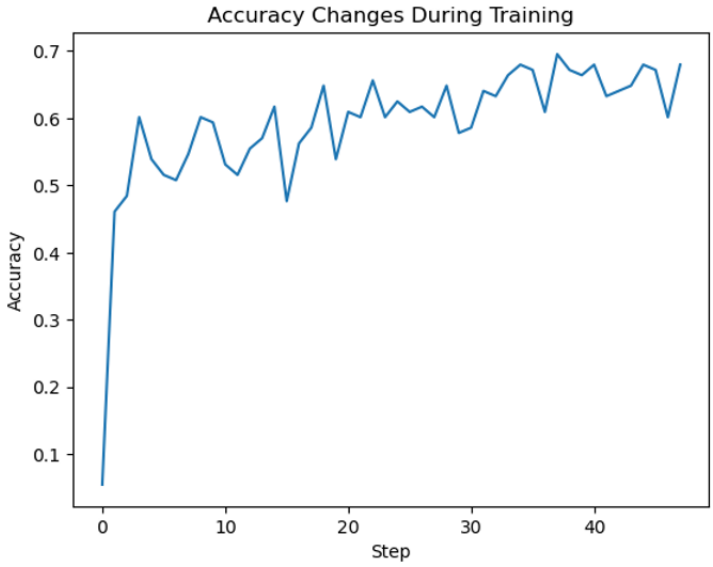

Figure 6.11: Accuracy of 3-Layer model

The graph shows how good the model is getting at making correct predictions as it learns from the data. As training goes on, we see the model's performance improve quickly at first, but then it levels off and doesn't get much better after reaching about 65 to 70% accuracy.
To understand the model's performance in more detail, we'll look at some confusion matrices later on. We've also set up a way to see the activity of the model's neurons during training to give us more insight into how it's working.

The firing patterns of the model has shown below, how the information in the data triggers the neuron layers and then the spikes are getting generated

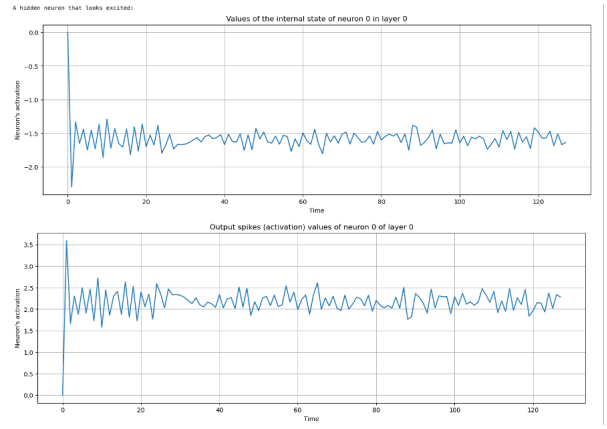

Figure 6.12

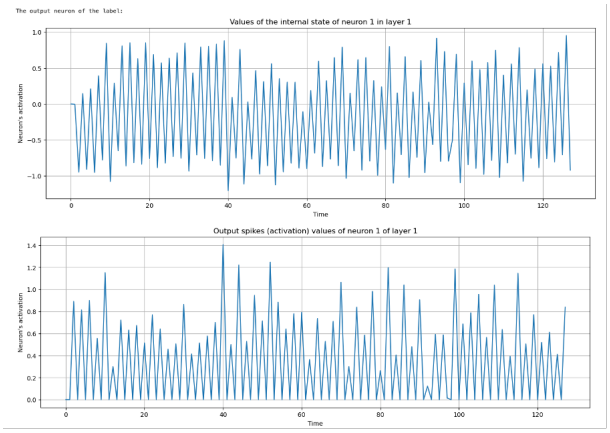

Figure 6.13

From the above spiking patterns, we can see the smooth transmission of the information through the layers. This graphs along with the confusion matrices will help us later on to get insights this model's characteristics.

```
spiking_model_4layers = Pulse_Network_4layer(device, n_time_steps=128, begin_eval=0)
train_epochs(spiking_model_4layers)
```

Figure 6.14

4-Layer Model: To work and evaluate the 4-layer network, we have executed the code. This 4 layer model is more complex model than the other models as the number of layers can create a make or break situation for the model. The complexity can significantly improve the model's performance but with that kind of complexity, the process of data transmission can get effected, which may lead to a lower accuracy.

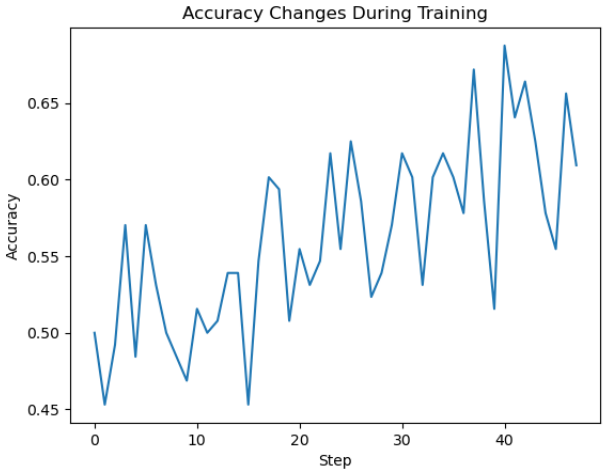

Figure 6.15

This graph shows that the 4 layer model is not one of the accurate model rather than 3 or 2 layer models. But still this model has achieved a moderate accuracy. We can visualise the reasons of this fluctuating accuracy more clearly when we see the spiking layers and the accuracy on the testing dataset.

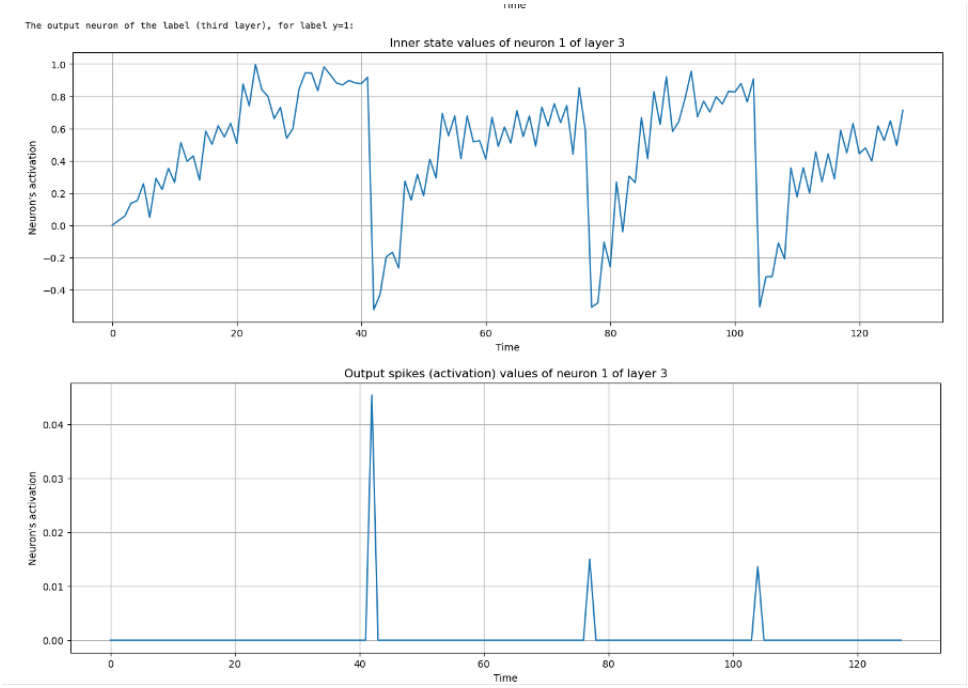

Figure 6.16

The firing patterns of the graphs shown, how the information in the data triggers the neuron layers and then the spikes are getting generated. We can see that the 3rd layer of the neuron in the above graph. In this graph, we can see that the model very closely mimics the firing pattern of a leaky integrate and fire model of the spiking neural networks. Each and every spikes are generated in the output when the input currents are become the reason of the membrane potential to reach the threshold voltage.

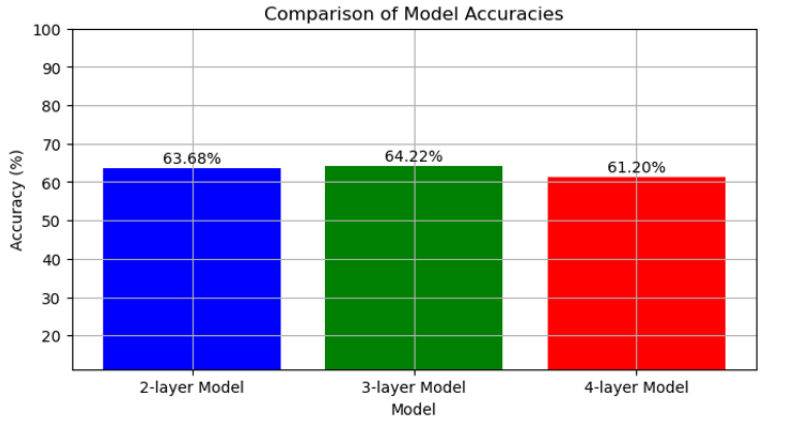

Figure 6.17: Performance comparison of 2, 3, and 4 layer spiking neural network models

Comparison between 2, 3 and 4 layers of Spiking Neural networks: We have compared the accuracy of the models using a bar chart. The bar chart provided illustrates the performance comparisons of the spiking neural network models with varying complexities, as measured by their accuracy percentages. The simplest model with 2 layers did a good job and got 63.68% right, which we see as a blue bar. A slightly more complex model with 3 layers did a little bit better and got 64.22% right, shown by a green column. But then, when we have tried an even more complex model with 4 layers, it didn't performed as good as the less complex models.

Binary Classification using small dataset:

To compare the fundamental advantage of the spiking neural network's ability to adapt on a minimum amount of data over traditional neural networks, we have trained the same models and compared the accuracy of these models

```
spiking_model_2layers_minimum_data_more_epochs = Pulse_Network(device, n_time_steps=128, begin_eval=0)
train_epochs(spiking_model_2layers_minimum_data_more_epochs)
```

Figure 6.18: Implementation of 2-layer model for small dataset

```
spiking_model_3layers_minimum_data_more_epochs = Pulse_Network_3layer(device, n_time_steps=128, begin_eval=0)
train_epochs(spiking_model_3layers_minimum_data_more_epochs)
```

Figure 6.19: Implementation of 3-layer model for small dataset

```
spiking_model_4layers_minimum_data_more_epochs = Pulse_Network_4layer(device, n_time_steps=128, begin_eval=0)
train_epochs(spiking_model_4layers_minimum_data_more_epochs)
```

Figure 6.20: Implementation of 4-layer model for small dataset

In the above images, we can see that we have created a spiking neural network model with two, three and four layers, designed to handle a minimum amount of data over a greater number of training epochs. This model is being instantiated with specific parameters and the accuracy has been stored to compare later on.

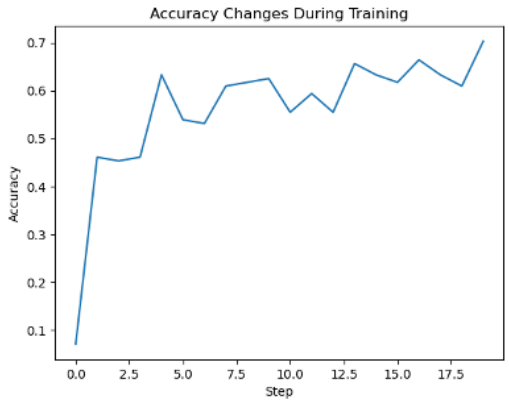

Figure 6.21: Accuracy of 2 layer model

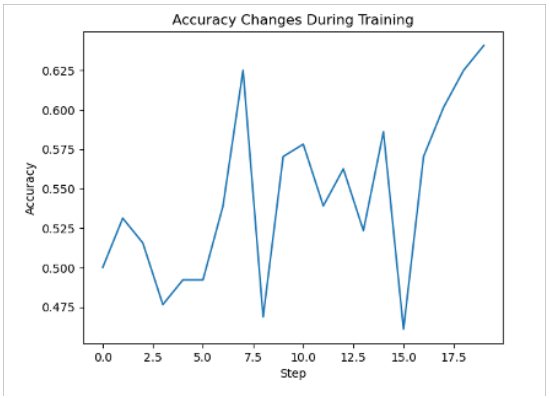

Figure 6.22: Accuracy of 3 layer model

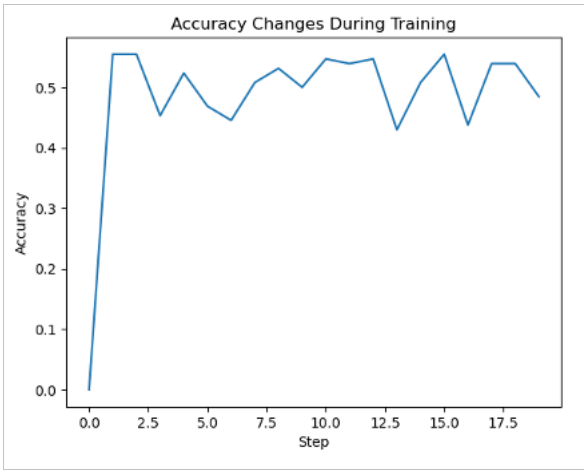

Figure 6.23: Accuracy of 4 layer model

In the above figures, we can see the accuracy graphs for the three different models with 2, 3 and 4 internally connected neuron layers. The first graph(model with 2 layers) shows an accuracy trajectory that starts quite low but steadily improves, indicating that the model may be learning effectively over time. The second graph(model with 3 layers) shows a more fluctuating accuracy while training epochs, which suggests that this model's learning process might be less stable or encountering more difficulty in converging. The third graph(model with 4 layers) also starts with a low accuracy but reaches a more consistent level, albeit with some fluctuation. We can get more insights from the spiking pattern of each models.

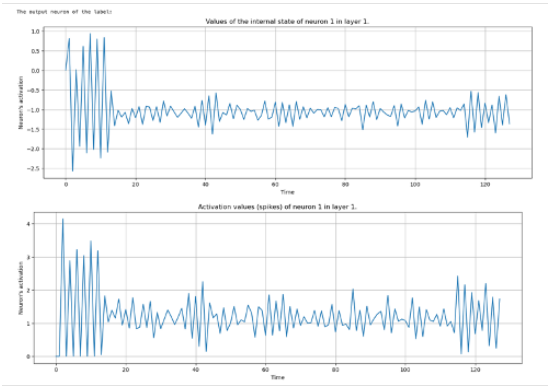

Figure 6.24: Spiking patterns in input layer

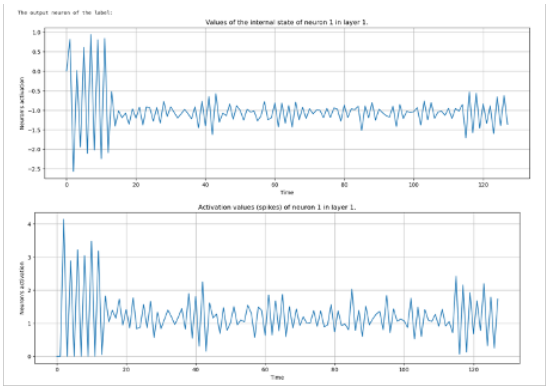

Figure 6.25: Spiking patterns in output layer

The above spiking graphs are the two layers of two layer model. If we see carefully, there are enough spikes available at the output layer, which means the model has recognise a good pattern to classify the data.

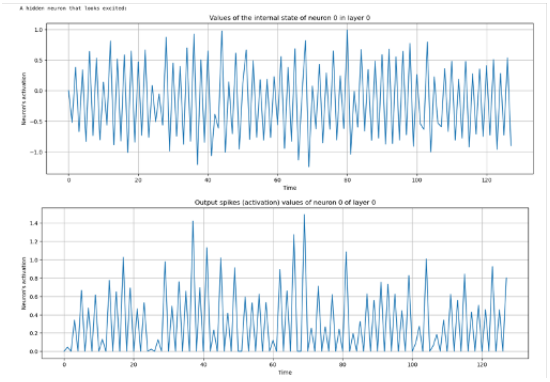

Figure 6.26: Spiking patterns in output layer for 3-layer model

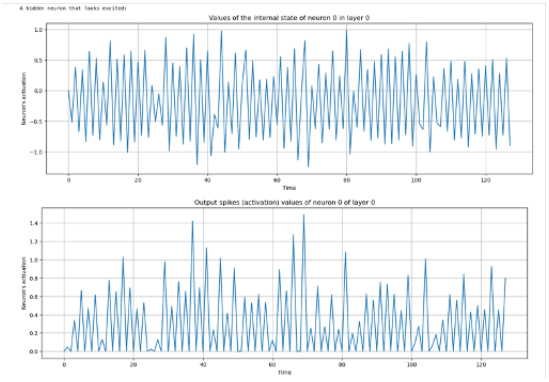

Figure 6.27: Spiking patterns in output layer for 4-layer model

The left graph shows the output layer of the 3 layer model and the right graph shows the output layer of the 4 layer model. We can see the difference of the spikes for the same image in the output layer. That shows that the 4 layer model is not able to recognise the classifying pattern. This explains the difference in accuracy demonstrated in the accuracy graphs. The difference will be very much clear when we will plot all the accuracy graphs together.

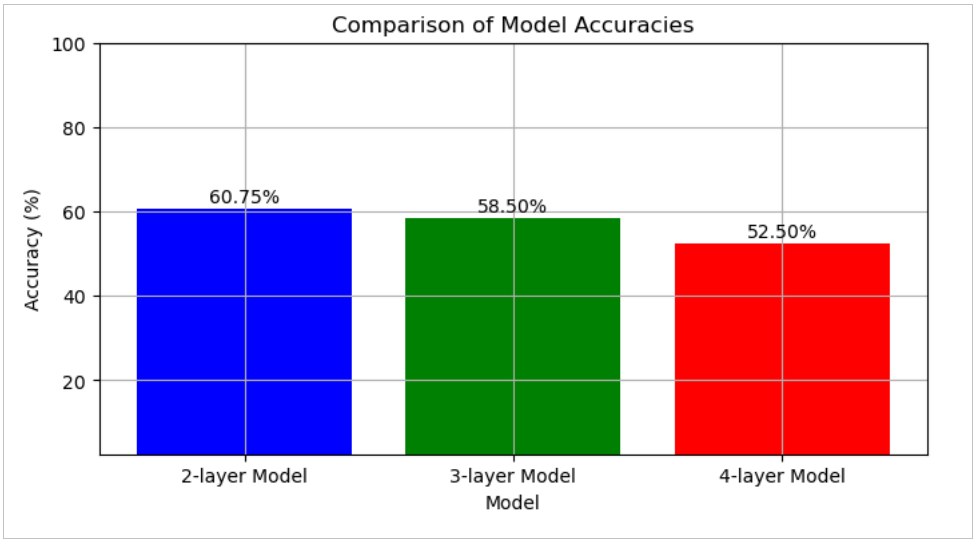

Figure 6.28: Performance comparison of 2, 3, and 4 layer spiking neural network models

Comparison between 2, 3 and 4 layer models of Spiking Neural networks while using small dataset: We have compared the accuracy of the models using a bar chart. The bar chart provided illustrates the performance comparisons of the spiking neural network models with varying complexities, as measured by their accuracy percentages. The two-layer model is represented by a blue bar and has achieved an accuracy of 60.75%. The three-layer model, shown in green, has a slightly lower accuracy of 58.50%. The four layer model depicted in red, performs the least effectively, with an accuracy of only 52.50%.

6.3.2 Multi class Classification using SNN:

As we have explained in the dataset module, we are using a Kaggel dataset of satellite images to perform the multi class classification.The dataset consists 4 different type of images. The 4 different folder named “cloudy”, “desert”, “green_area”, and “water”. Again we have performed the experiment with 3 different models having 2, 3 and 4 layers of interconnected neurons.

```
spiking_model_2layer_multiclass = Pulse_Network_2layer(device, n_time_steps=128, begin_eval=0)
train_epochs(spiking_model_2layer_multiclass)
```

Figure 6.29: Implementation of 2-layer model for multi class classification

```
spiking_model_3layer_multiclass = Pulse_Network_3layer_multiclass(device, n_time_steps=128, begin_eval=0)
train_epochs(spiking_model_3layer_multiclass)
```

Figure 6.30: Implementation of 3-layer model for multi class classification

```
spiking_model_4layer_multiclass = Pulse_Network_4layer_multiclass(device, n_time_steps=128, begin_eval=0)
train_epochs(spiking_model_4layer_multiclass)
```

Figure 6.31: Implementation of 4-layer model for multi class classification

In the above images, we can see that we have created a spiking neural network model with two, three and four layers, designed to handle a data having four classes over a three training epochs to perform a multi class classification. This model is being instantiated with specific parameters and the accuracy has been stored to compare later on.

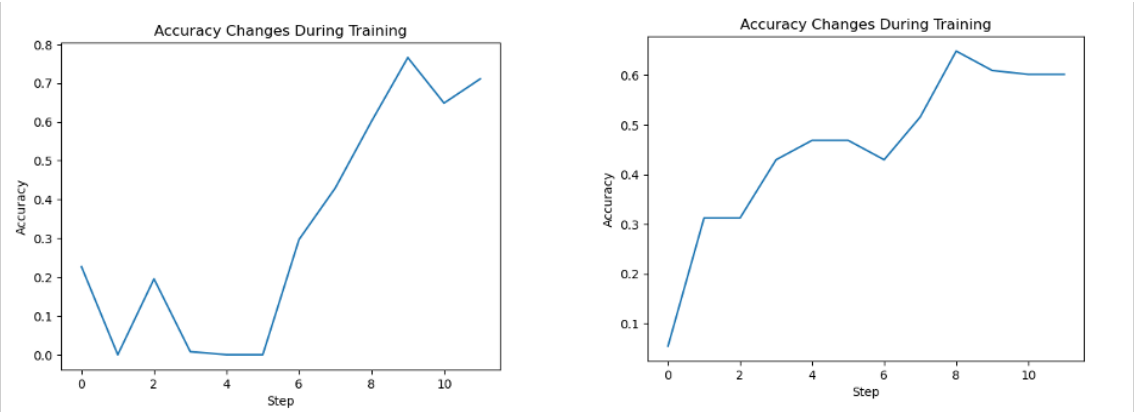

Figure 6.32: Accuracy of 2 layer model

Figure 6.33: Accuracy of 3 layer model

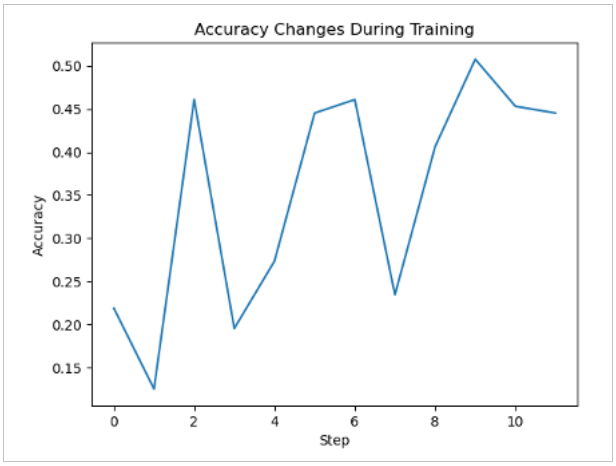

Figure 6.34: Accuracy of 4 layer model

In the above figures, we can see the accuracy graphs for three different computational models during their training phases. These models has 2, 3, and 4 layers of interconnected neuron models. From these graphs, we can compare the accuracy of the three different models and also can make an opinion on the relations between performance of the models based on their complexity.
The first graph shows the model with 2 layers of inter connected neurons, generates significant fluctuation in accuracy but ends with a high accuracy, suggesting that despite some volatility, the model could be capturing the underlying patterns in the data effectively.

The second graph shows the model with 3 layers of inter connected neurons, generates a more steady and incremental improvement in accuracy, which might indicate a more stable learning process, possibly due to better generalization.

The third graph is the model with 4 layers of inter connected neurons, shows a more fluctuating accuracy while training epochs. It showcases the challenges in learning, potential over-fitting to noise in the training data.

We can get more insights from the spiking pattern of each models.

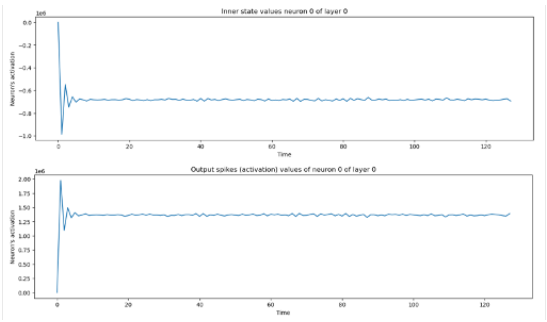

Figure 6.35: Spiking patterns for input layer

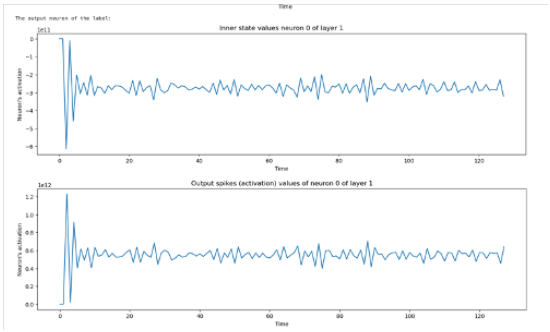

Figure 6.36: Spiking patterns for output layer

Comparative analysis of spiking patterns between input and output layers of a 2-layer model

In the above figures, we can see the spiking patterns for both the layers of the 2 layer model. . If we see carefully, there are similarities in both the patterns and the timings of the patterns. The spiking pattern also indicates the segments of the data where the density of available information was higher.

Now, we will compare the output spikes of the three-layer and four-layer models, which will provide deeper insights into the models. By examining these plots, we will try to understand the reasons behind the differences in accuracy.

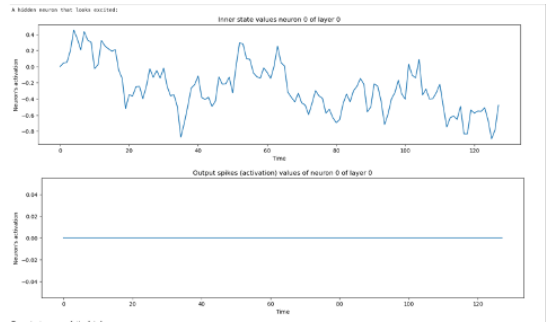

Figure 6.37: Spiking patterns of output layer of 3-layer model

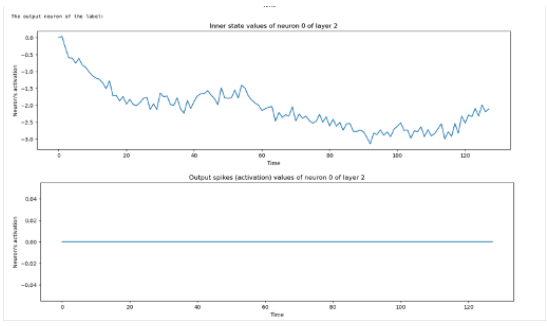

Figure 6.38: Spiking patterns of output layer of 4-layer model

Comparative analysis of spiking patterns of output layers of 3-layer and 4-layer models

In the above plots the left plot (plot no()) shows the output spikes of 3 layer model and the right plot is showing the output spikes of the 4 layer model. From both the plots, we can see that when the information is coming from the previous layer, there are a small amount of spikes left. But when the information passes through the last RNN neuron layer, the neuron are not able to generate any significant spike for this particular example. The reason might be, in that particular image, there are not such information available. But the scenario describes the overall behaviour of the 3 and 4 layer spiking neural network model. The lack of spikes is the reason of low accuracy of these models. This will became more evident when will compare on the based on confusion

matrices.

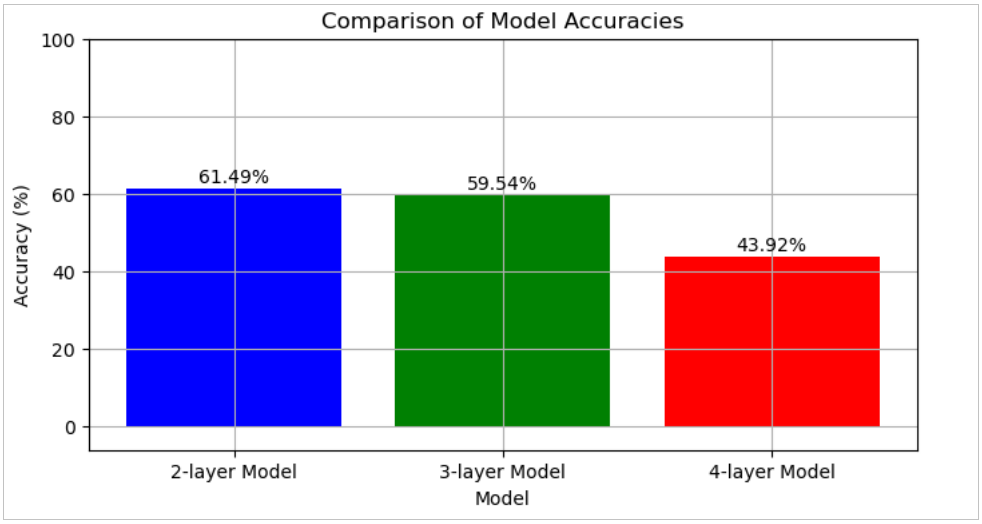

Figure 6.39: Performance comparison of 2, 3, and 4 layer spiking neural network models in multi class classification

Comparison between 2, 3 and 4 layers of Neural networks: We have compared the accuracy of the models using a bar chart. The bar chart provided illustrates the performance comparisons of the spiking neural network models with varying complexities, as measured by their accuracy percentages. The two-layer model shows a 61.49% accuracy and is represented by the blue bar. The green bar represents the three-layer model with a slightly lower accuracy of 59.54%. Surprisingly, the four-layer model, depicted with the red bar, has a significantly lower accuracy of 43.92%. This visual data suggests that additional complexity, as seen in the four-layer model, does not necessarily equate to better performance and may, in fact, lead to a reduction in accuracy.

6.3.3 Binary and Multi class classification Using CNN

We have performed the binary and multi class classifications using the convolution neural networks for the purpose of comparison the accuracy with the SNN

Binary Classification using CNN We have performed the binary classification using CNN twice. Initially we have performed using the large dataset and second time using the small dataset.

```
model1 = Sequential()
model1.add(Conv2D(filters=12, kernel_size=(5, 5), activation='relu', input_shape=Input_shape))
model1.add(MaxPooling2D())

model1.add(Conv2D(filters=8, kernel_size=(5, 5), activation='relu'))
model1.add(Conv2D(filters=4, kernel_size=(5, 5), activation='relu'))
model1.add(Dropout(0.3))

model1.add(AveragePooling2D())
model1.add(Flatten())
model1.add(Dense(units=16, activation='relu'))
model1.add(Dense(units=2, activation='softmax'))
```

Figure 6.40: Model used for Binary classification for large dataset

The model showed a good accuracy while training but while testing the accuracy dropped significantly.

```
binary_large_dataset_score = model1.evaluate(x_test_image, y_test,verbose=0)
print('Test Accuracy : {:.4f}'.format(binary_large_dataset_score[1]))
print('test Loss : {:.4f}'.format(binary_large_dataset_score[0]))
```

```
Test Accuracy : 0.5987
test Loss : 3.9719
```

Figure 6.41: Accuracy for Binary classification using large dataset

In this figure we can see the accuracy or the binary classification using large dataset. The accuracy is significantly low comparing with training shows a overfitting effect.

Similarly we have also performed the binary classification using the small dataset.

```
Model: "sequential_3"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_9 (Conv2D)           (None, 220, 220, 12)      912

 max_pooling2d_3 (MaxPoolin  (None, 110, 110, 12)      0
 g2D)

 conv2d_10 (Conv2D)          (None, 106, 106, 8)       2408

 conv2d_11 (Conv2D)          (None, 102, 102, 4)       804

 dropout_3 (Dropout)         (None, 102, 102, 4)       0

 average_pooling2d_3 (Avera  (None, 51, 51, 4)         0
 gePooling2D)

 flatten_3 (Flatten)         (None, 10404)             0

 dense_6 (Dense)             (None, 16)                166480

 dense_7 (Dense)             (None, 2)                 34

=================================================================
Total params: 170638 (666.55 KB)
Trainable params: 170638 (666.55 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Figure 6.42: Model used for Binary classification for small dataset

The model also showed a good accuracy while training but while testing the accuracy dropped significantly as we haven't changed the model to set up the comparison parameter.

We will evaluate the model's accuracy later on while comparing with Snn model's accuracy.

```
binary_small_dataset_score = model2.evaluate(x_test_image, y_test,verbose=0)
print('Test Accuracy : {:.4f}'.format(binary_small_dataset_score[1]))
print('test Loss : {:.4f}'.format(binary_small_dataset_score[0]))
```

```
Test Accuracy : 0.6100
test Loss : 3.2152
```

Figure 6.43: Accuracy for Binary classification using small dataset

```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 220, 220, 12)      912

 max_pooling2d (MaxPooling2  (None, 110, 110, 12)      0
 D)

 conv2d_1 (Conv2D)           (None, 106, 106, 8)       2408

 conv2d_2 (Conv2D)           (None, 102, 102, 4)       804

 dropout (Dropout)           (None, 102, 102, 4)       0

 average_pooling2d (Average  (None, 51, 51, 4)         0
 Pooling2D)

 flatten (Flatten)           (None, 10404)             0

 dense (Dense)               (None, 16)                166480

 dense_1 (Dense)             (None, 4)                 68

=================================================================
Total params: 170672 (666.69 KB)
Trainable params: 170672 (666.69 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Figure 6.44: Model used for multi class classification

Multi class Classification using CNN For the multi class classification, we have used this model shown in the figure. After the training, we have calculated the accuracy of the model using the testing dataset. This model showed effect of over-fitting while calculating the test accuracy.

```
multiclass_score = model.evaluate(x_test_image, y_test,verbose=0)
print('Test Accuracy : {:.4f}'.format(multiclass_score[1]))
print('test Loss : {:.4f}'.format(multiclass_score[0]))
```

```
Test Accuracy : 0.6876
test Loss : 0.4709
```

Figure 6.45: Accuracy for multi class classification

6.4 Comparisons

6.4.1 Compare between binary and multi-class classification using Spiking Neural Networks:

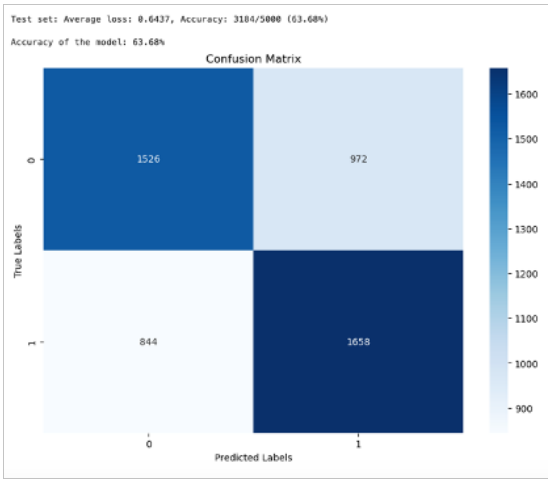

Figure 6.46: confusion matrix of Binary classification

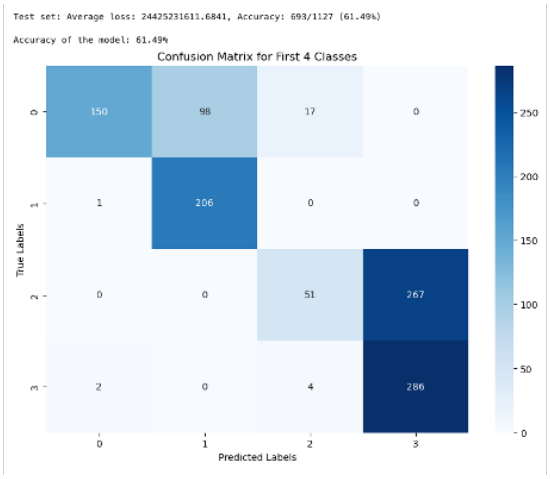

Figure 6.47: confusion matrix of multi class classification

confusion matrix of 2 layer models

The plots in the above shows the result for the 2 layer models. The left side plot shows the confusion matrix for a binary classification and in the right plot, we have the results, when we performed the multi class classifications. The true positive and true negatives are doing justice with the accuracy showed by the models previously.

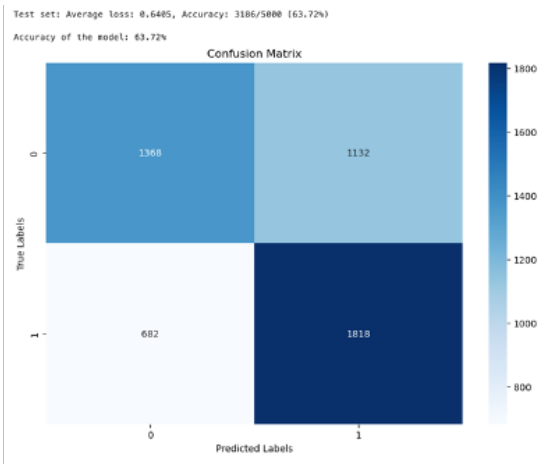

Figure 6.48: confusion matrix of Binary classification

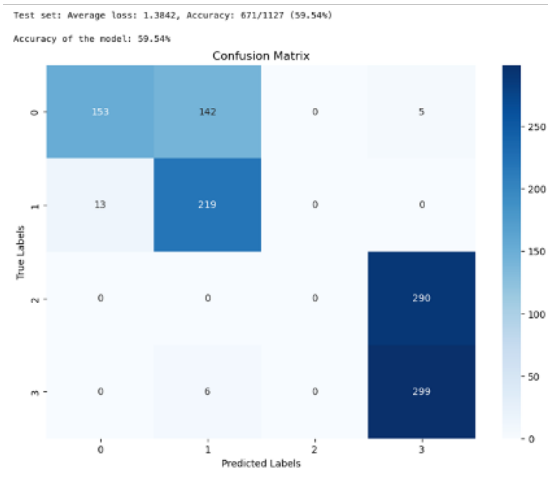

Figure 6.49: confusion matrix of multi class classification

confusion matrix of 3 layer models

The charts above shows the performance outcomes for models with three internal layers. The plot on the left shows the confusion matrix derived from binary classification, which captures the model's predictive accuracy and the relationship between true and predicted classes. The plot on the right displays the results from a multi class classification showing different predictive patterns.

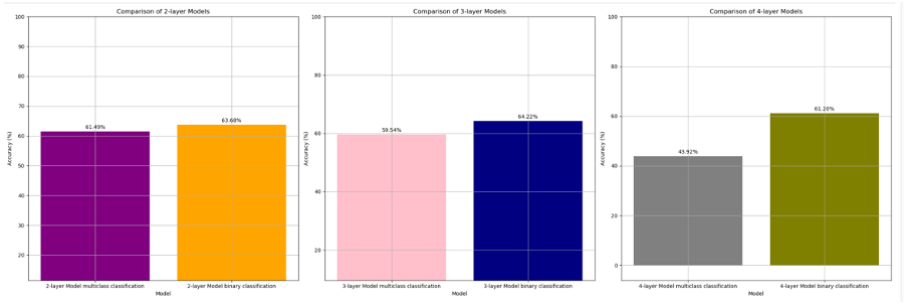

Figure 6.50: Performance Comparison of Spiking Neural Network Models for binary and multi class

These patterns in the confusion matrices align with the trends observed in the accuracy graphs that we have discussed previously, gives us the insights of the model's performance for two different classes.

For a better comprehension, these graphs have been plotted to compare accuracy. The first graph illustrates the performance of two-layer models, with the purple bar at 61.49% accuracy for the model trained for a multi class classification and the yellow bar showing a 63.68% accuracy for the binary classification. The middle graph contrasts three-layer models, where the multi-class classification, shown in pink, has an accuracy of 59.54%, and the blue bar indicates a 64.22% accuracy for the binary classification. Finally, the third graph compares four-layer models where the multi class classification has been shown in the grey graph indicates 43.92% accuracy and the plot shown in olive indicates the accuracy of 61.20% for the binary classifications. This gives us some insights that with the increases of complexity, the data transmission from the input layer to output layer is getting difficult. This is one of the potential reasons to show a lower accuracy in the more complex models.

6.4.2 Compare between performance of Spiking neural network models while using small dataset and large dataset

We have compared the performance of the both situation based on the accuracy. For that we have already generated the confusion matrix to showcase the accuracy difference in both the cases while using the model on testing datasets.

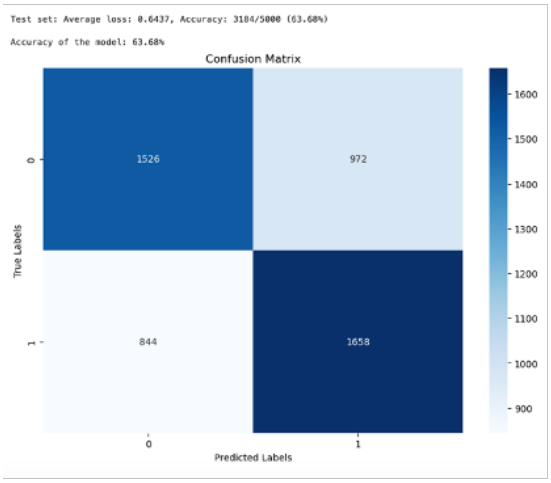

Figure 6.51: confusion matrix while using large dataset

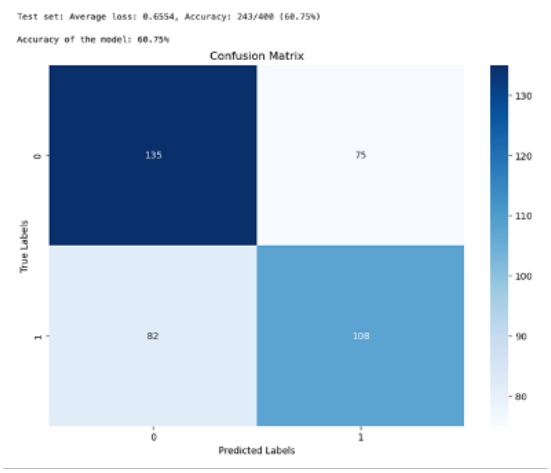

Figure 6.52: confusion matrix while using small dataset

confusion matrix of 2 layer models

The plots in the above shows the result for the 2 layer models. The left side plot shows the confusion matrices when we have used the large dataset and in the right plot, we have the results when we have used the small dataset.

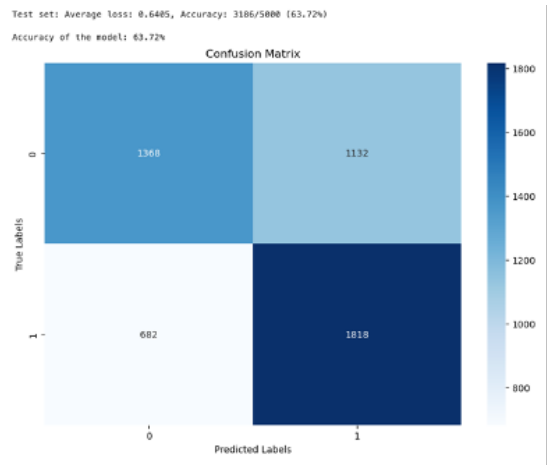

Figure 6.53: confusion matrix while using large dataset

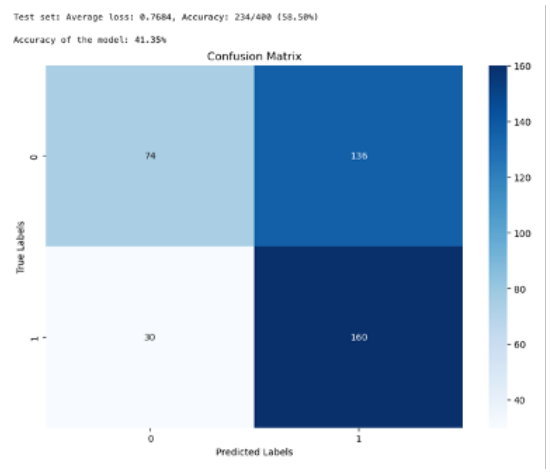

Figure 6.54: confusion matrix while using small dataset

confusion matrix of 3 layer models

The charts above display the outcomes for the three-layer models. The plot on the left presents the confusion matrix generated using a large dataset, while the plot on the right exhibits the results from employing a small dataset. The patterns observed in the confusion matrices are consistent with the previously shown accuracy graphs.

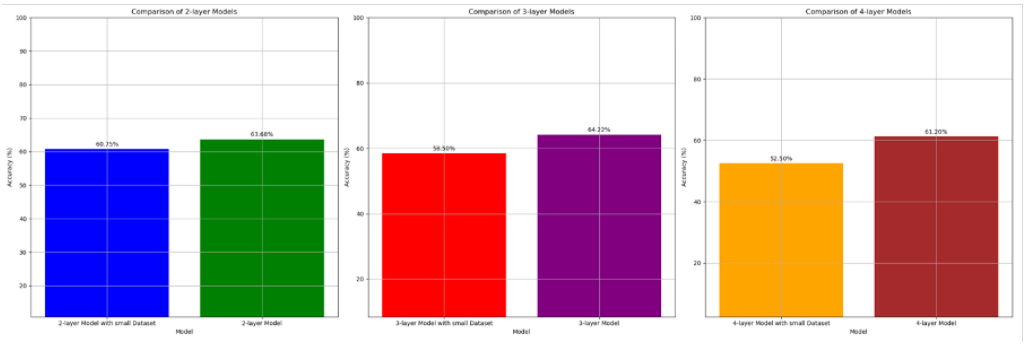

Figure 6.55: Performance Comparison of Spiking Neural Network Models for binary classification of different models

For a better comprehension, these graphs have been plotted to compare accuracy. The first graph illustrates the performance of two-layer models, with the blue bars showing a 60.75% accuracy for the model trained on a small dataset and the green bar showing a 63.68% accuracy for the standard model. The middle graph contrasts three-layer models, where the model with a small dataset, shown in red, has an accuracy of 58.50%, and the purple bar indicates a 64.22% accuracy for the standard model. Finally, the third graph compares four-layer models, with the yellow bar representing the small dataset model at 52.50% accuracy, while the standard model is depicted in dark red with an accuracy of 61.20%. These graphs demonstrate that the amount of data influences the accuracy of models with different layers, but the impact is not significant. This highlights the benefits of using spiking neural networks.

6.4.3 Compare between performance of CNN models

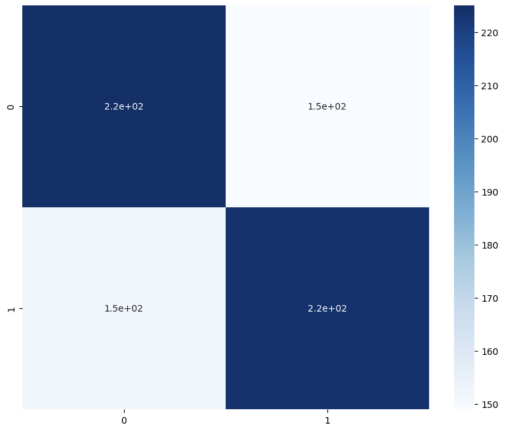

Figure 6.56: confusion matrix while using large dataset

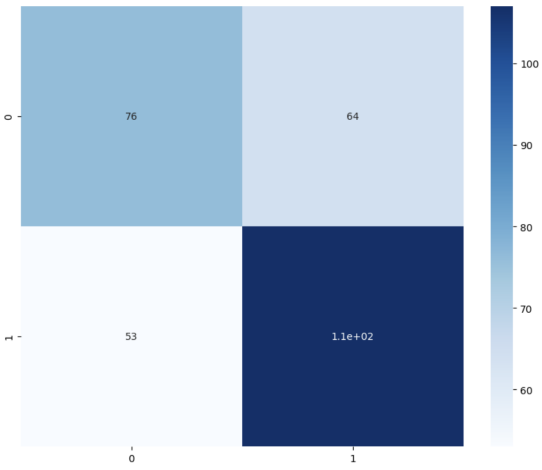

Figure 6.57: confusion matrix while using small dataset

confusion matrix of CNN models performing binary classification

The charts above display the outcomes for the CNN models. The plot on the left presents the confusion matrix generated using a large dataset, while the plot on the right exhibits the results from employing a small dataset. The patterns observed in the confusion matrices are consistent with the previously accuracy calculation performed upon test data.

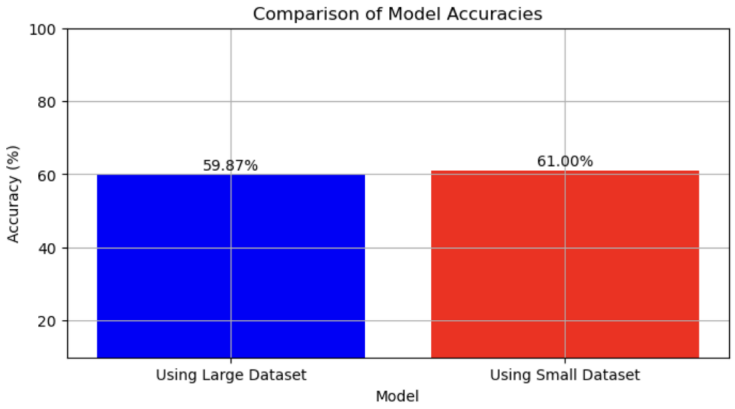

Figure 6.58: Performance Comparison of CNN using for using small and large dataset

To visualise the accuracy we have displayed the graph. The blue bar represents a model trained using a large dataset, achieving an accuracy of 59.87%. The red bar, on the other hand, represents a model trained using a smaller dataset, which surprisingly achieves a slightly higher accuracy of 61.00%. This visual comparison highlights the differences in model performance relative to the dataset size, possibly indicating the impact of data quality, overfitting, or the model's ability to generalize from smaller datasets.

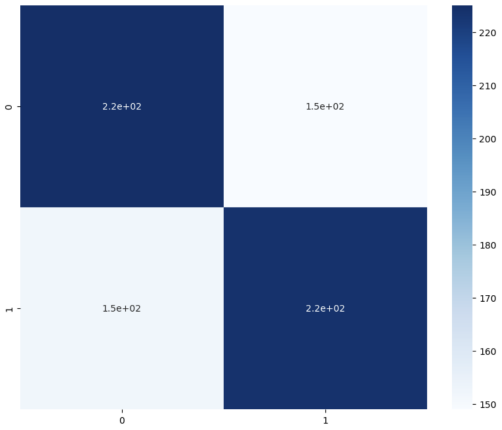

Figure 6.59: confusion matrix while performing binary classification using CNN

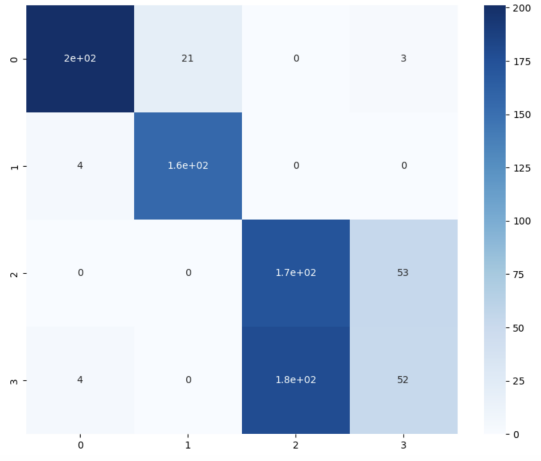

Figure 6.60: confusion matrix while performing multi class classification using CNN

confusion matrix of CNN models performing binary and multi class classification

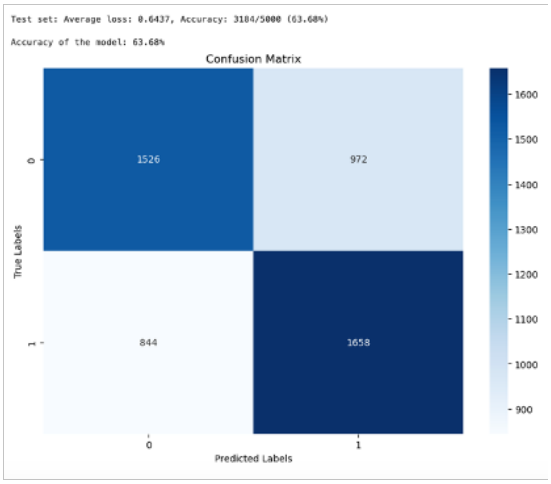

Figure 6.61: confusion matrix of Binary classification using SNN

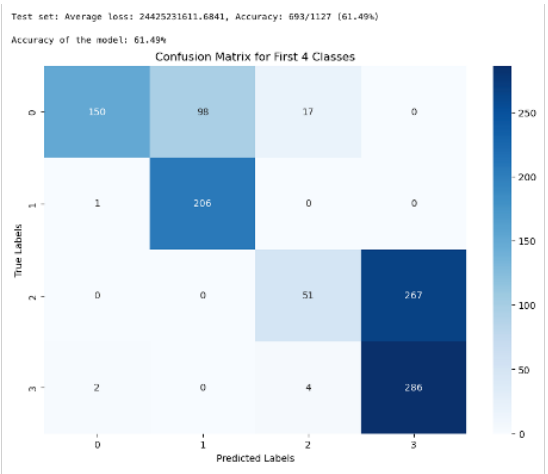

Figure 6.62: confusion matrix of multi class classification using SNN

confusion matrix of SNN models performing binary and multi class classification

From all the above comparison, we can come to the conclusion that the SNN slightly outperformed the CNN in terms of accuracy, but in terms or time consumption and energy efficiency and avoiding overfitting, the SNN totally out performed CNN.

Chapter 7

Conclusion

7.1 Discussion

In this dissertation we have highlighted the precise blessings and programs of spiking neural networks This have a look at highlighted the right benefits and programs of spiking neural networks (SNNs), which examined the capacity to imitate human mind typical overall performance and provide most advantageous energy-efficient answers for real-time programs This potential to sleep fantastic from conventional synthetic neural networks for reasons they're uniquely proper for programs in which era inputs are used usually.

Throughout our studies, we examined several SNN models and evaluated their effectiveness via a comparative evaluation of their accuracy metrics. This look at not simplest highlighted the strengths of SNNs, which consist of low strength intake and real-time information overall performance, but additionally their robustness in schooling and the complexity of cutting-edge learning algorithms the shortage of tailored designs for their precise systems have been moreover highlighted Some barriers have been also highlighted Despite those demanding situations , the effects spotlight the ability of SNNs to push the limits of what may be performed with neural engineering.

Additionally, the have a look at highlights the sizable improvements in computing abilities that SNNs carry to the table, specially in situations where traditional networks would require appreciably more computing resources This overall performance is due to their nature driven via records a builds on it, with computation handiest responding to incremental output in choice to being a non-forestall information glide, which appreciably reduces the required computing energy.

In precis, our comparative observe not great demonstrates the inherent abilities of SNNs but additionally clarifies crucial areas where in addition enhancements are wished.

In this study, we have highlighted the distinctive advantages and potential applications of Spiking Neural Networks (SNNs) over the traditional neural networks(mostly against CNN), showcasing their capacity to mimic the human brain's functionality and provide energy-efficient solutions for real-time processing. We have also compared different models of spiking neural networks and compared the accuracies to demonstrate the capabilities and the limitations of the spiking neural networks. This prospective will help us to work intensely on the spiking neural networks and

opens the wide path for the future works.

7.2 Limitation

Spiking neural networks (SNNs), although promising, have several major obstacles that evade widespread adoption and improvement.

One of the first challenges in the development of SNN technology is the easy availability of comprehensive research papers and sources Compared to traditional neurons, the SNN task is new for it's a wonderful thing. This lack of literature and established research makes it difficult for new researchers to enter the field and for existing researchers to advance their research, slowing innovation and meaning-making

Current computer systems are mainly designed to support traditional neural networks that rely on continuous statistical models This design is not ideal for discrete SNNs, with opportunity and utilizes it, with the data-driven approach primarily based on neuronal maturation time. Effective implementation requires considerable flexibility, making it far less likely to be widely used.

Along with that the SNN requires a different type of input and output shape to access the information from the data, as it doesn't use the continuous form of data, as it rather use discrete data. The input encoder and the output decoder is one of the key limitation for the SNN as we have to built it each time befor using different types of data.

7.3 Future Work

The capability of Spiking Neural Networks (SNNs) to revolutionize numerous sectors, mainly those requiring real-time information processing and excessive energy overall performance, is tremendous. As this test has demonstrated, even as SNNs hold promising blessings, there may be massive art work to be completed in terms of studies and development to absolutely harness those benefits.

Object Detection Integration: Future research should explore the integration of SNNs with object detection technologies. SNNs have the potential to enhance the efficiency and adaptability of object detection systems, particularly in environments that require real-time processing and high levels of accuracy, such as in autonomous vehicles or advanced surveillance systems.

Development of Specialized Learning Algorithms: Currently, one of the giant traumatic situations going via SNNs is the dearth of effective mastering algorithms which are particularly designed for their form. Future studies should goal at growing and refining mastering algorithms which encompass Spike-Timing-Dependent Plasticity (STDP) and awesome plasticity policies which could optimize the training technique of SNNs. These upgrades may additionally want to decorate the networks' functionality to generalize in the direction of particular obligations,

making them extra flexible and relevant to a broader sort of problems.

Research in hybrid models: Another promising learning approach is to build hybrid models that combine the temporal accuracy of SNNs with traditional neural networks with spatial processing capabilities should be directed towards utilitarian knowledge and give us tremendously good result in the field of machine learning. Such fashions ought to doubtlessly result in breakthroughs in complicated sensory processing duties in regions together with vision and speech reputation. Research need to be directed in the direction of expertise how those types of network architectures may be correctly combined to leverage both their strengths.

Scalability and Commercial Viability: As SNN technology advances, another important area of recognition can be to deal with problems of scalability and business viability. Ensuring that SNNs can be deployed at a large scale and included seamlessly into current technological infrastructures can be vital for their good sized adoption.

By addressing those areas, destiny studies can substantially develop the sector of neural networks, particularly improving the applicability, performance, and effectiveness of SNNs in real-life situations. This could now not only contribute to the theoretical knowledge of neural networks but also to the practical skills of technology in numerous industries.

In conclusion, while challenges remain, the future for Spiking Neural Networks in advancing the frontiers of artificial intelligence and computational neuroscience are promising and bright. Continued research and development in this field of spiking neural networks as a part of bio inspired machine learning models are not only warranted but essential for utilise the full potential of bio-inspired computing paradigms. The journey of exploring and harnessing the capabilities of SNNs is just beginning, and it shows significant potentials for advancement in technologies in the field of machine learning and artificial intelligence.

Bibliography

Zohreh Doborjeh, Maryam Doborjeh, Mark Crook-Rumsey, Tamasin Taylor, Grace Y Wang, David Moreau, Christian Krägeloh, Wendy Wrapson, Richard J Siegert, Nikola Kasabov, et al. Interpretability of spatiotemporal dynamics of the brain processes followed by mindfulness intervention in a brain-inspired spiking neural network architecture. *Sensors*, 20(24):7354, 2020.

Qingxi Duan, Zhaokun Jing, Xiaolong Zou, Yanghao Wang, Ke Yang, Teng Zhang, Si Wu, Ru Huang, and Yuchao Yang. Spiking neurons with spatiotemporal dynamics and gain modulation for monolithically integrated memristive neural networks. *Nature communications*, 11(1): 3399, 2020.

Xiaoyan Fang, Shukai Duan, and Lidan Wang. Memristive hodgkin-huxley spiking neuron model for reproducing neuron behaviors. *Frontiers in Neuroscience*, 15:730566, 2021.

Dexuan Huo, Jilin Zhang, Xinyu Dai, Pingping Zhang, Shumin Zhang, Xiao Yang, Jiachuang Wang, Mengwei Liu, Xuhui Sun, and Hong Chen. A bio-inspired spiking neural network with few-shot class-incremental learning for gas recognition. *Sensors*, 23(5):2433, 2023.

Eugene M Izhikevich. Simple model of spiking neurons. *IEEE Transactions on neural networks*, 14(6):1569–1572, 2003.

Dae-Hwan Kang, Hyun-Goo Jun, Kyung-Chang Ryoo, Hongsik Jeong, and Hyunchul Sohn. Emulation of spike-timing dependent plasticity in nano-scale phase change memory. *Neurocomputing*, 155:153–158, 2015.

Kaushalya Kumarasinghe, Nikola Kasabov, and Denise Taylor. Brain-inspired spiking neural networks for decoding and understanding muscle activity and kinematics from electroencephalography signals during hand movements. *Scientific reports*, 11(1):2486, 2021.

Chankyu Lee, Syed Shakib Sarwar, Priyadarshini Panda, Gopalakrishnan Srinivasan, and Kaushik Roy. Enabling spike-based backpropagation for training deep neural network architectures. *Frontiers in neuroscience*, 14:497482, 2020.

Deboleena Roy, Priyadarshini Panda, and Kaushik Roy. Synthesizing images from spatio-temporal representations using spike-based backpropagation. *Frontiers in neuroscience*, 13:437493, 2019a.

Kaushik Roy, Akhilesh Jaiswal, and Priyadarshini Panda. Towards spike-based machine intelligence with neuromorphic computing. *Nature*, 575(7784):607–617, 2019b.

Shiva Subbulakshmi Radhakrishnan, Amritanand Sebastian, Aaryan Oberoi, Sarbashis Das, and Saptarshi Das. A biomimetic neural encoder for spiking neural network. *Nature communications*, 12(1):2143, 2021.

Chee Keong Tee, Hian Hian See, Brian Lim, Soon Hong Harold Soh, Tasbolat Taunyazov, SNG Weicong, Sheng Yuan Jethro Kuan, and Abdul Fatir Ansari. Event-driven visual-tactile sensing and learning for robots, October 19 2023. US Patent App. 18/010,656.

Ing Jyh Tsang, Federico Corradi, Manolis Sifalakis, Werner Van Leekwijck, and Steven Latré. Radar-based hand gesture recognition using spiking neural networks. *Electronics*, 10(12):1405, 2021.

Yoeri Van De Burgt and Paschalis Gkoupidenis. Organic materials and devices for brain-inspired computing: From artificial implementation to biophysical realism. *MRS Bulletin*, 45(8):631–640, 2020.

Anup Vanarse, Josafath Israel Espinosa-Ramos, Adam Osseiran, Alexander Rassau, and Nikola Kasabov. Application of a brain-inspired spiking neural network architecture to odor data classification. *Sensors*, 20(10):2756, 2020.

JJ Wang, SG Hu, XT Zhan, Q Yu, Z Liu, Tu Pei Chen, Y Yin, Sumio Hosaka, and Y Liu. Handwritten-digit recognition by hybrid convolutional neural network based on hfo2 memristive spiking-neuron. *Scientific reports*, 8(1):12546, 2018.