university of groningen

faculty of science and engineering

Liquid Spiking Networks with Adaptive Structural Dynamism

Harshith Sai Gavi Matam

university of groningen / faculty of science and engineering

University of Groningen

Liquid Spiking Networks Gradient With Adaptive Structural Dynamism

Master's Thesis

To fulfill the requirements for the degree of
Master of Science in Artificial Intelligence
at University of Groningen under the supervision of
Dr. Herbert Jaeger (Artificial Intelligence, University of Groningen)
and
Steven Abreu (Artificial Intelligence, University of Groningen)

Harshith Sai Gavi Matam (s4949447)

November 27, 2023

Contents

Acknowledgments

This project has proven tougher than initially expected and I would like to thank all those who have supported me throughout the process. I would like to dedicate this thesis to my grandmother who always believed in me. May she rest in peace.

I would like to offer my sincerest thanks to Steven Abreu for our brainstorming discussions and the guidance that has been instrumental in bringing this project to completion.

I am forever grateful to Herbert Jaeger for the exceptional lectures, unique knowledge sharing, and meticulous grading of my thesis. Your guidance has been instrumental in my academic growth.

I want to express my heartfelt gratitude to my parents for their unwavering support and love throughout my academic journey.

I'm indebted to Formula Y for providing me with the resources and support that played a crucial role in seeing this project through to completion.

I want to extend my thanks to my friends for their unwavering moral support and encouragement, which propelled me towards successful completion.

Abstract

Spiking Neural Networks (SNNs) represent an advanced class of neural networks, characterised by their ability to more closely mimic the temporal dynamics of biological neural systems. These networks were integrated with Liquid Time-Constant Networks to enhance the model's capability to retain information over extended time horizons, thereby performing well on time-series tasks. In this thesis, this Liquid Spiking Network model was evaluated on an event-based audio classification task to establish an initial benchmark. A series of experiments were designed and conducted to assess the model and its hyperparameters, leading to the conclusion that the model achieves a test accuracy of 84.7%. Notably, when efficiency is considered, the model's global ranking improves, underscoring its real-world applicability. Post-evaluation, neurons were provided with the ability to dynamically choose their connections to examine the impact on performance and efficiency. Our findings suggest that this capability leads to a network compression of 10.5%. Although this compression marginally reduces accuracy by 3%, it offers a balanced trade-off between performance and efficiency compared to other state-of-the-art models.

1 Introduction

The evolution of the field of AI, from the first mathematical model of a biological neuron to the well popular Large Language Models, has not been linear. Instead, it has been shaped by the challenges encountered along the way. One of the early milestones in AI research was the introduction of the McCulloch-Pitts Neuron or MP-Neuron [1], which demonstrated the ability to approximate simple functions such as the AND gate and the OR gate. However, as the field progressed, it became apparent that the MP-Neuron lacked the ability to approximate more complex functions. This realisation led to the development of the Perceptron [2], a breakthrough model that addressed some of the limitations of the MP-Neuron.

The Perceptron model was capable of solving linearly separable problems, where the data points can be separated by a simple linear boundary; but faced challenges when dealing with non-linearly separable problems [3]. To overcome this limitation, Multi-Layer Perceptron (MLP) [3] was designed, where the neurons are interconnected across multiple layers. This enabled it to learn and model complex relationships within the data. The development of the MLP marked a significant milestone in AI research, as it laid the foundation for deep learning and paved the way for the subsequent advancements in the field. Harnessing the power of MLPs, neural networks were used to tackle more complicated tasks such as pattern recognition, image classification, and natural language processing. Several types of neural networks were subsequently developed to improve performance in different tasks. Convolutional Neural Networks (CNN) [4] extended the power of neural networks by excelling in image recognition and processing. Recurrent neural networks (RNN) [5] were also introduced, specialising in sequential data analysis and enabling applications such as natural language processing and speech recognition. Generative Adversarial Networks (GAN) [6] revolutionised unsupervised learning and data generation tasks. These developments culminated in the rise of Transformers [7], especially as Large Language Models (LLMs), which pushed the boundaries of language processing and text generation.

The challenges that all the aforementioned neural networks have overcome were goal-oriented or performance-oriented, that is, they persisted until satisfactory performance was achieved. These challenges are often identified during the development process of the model and are therefore recognized as limitations that are usually addressed later. It is important to understand and acknowledge that these are challenges often identified and addressed, but not the only ones that exist. There are other challenges that are often overlooked because they are either naturally concealed or deemed unimportant in the service of the primary goal. These challenges pertain to aspects of efficiency, such as efficient training methods, efficient architectures, energy efficiency, and others. These are often termed as *Optimization challenges*, and are of current concern as we head towards a sustainable future.

Most of the research that has been conducted up until late has been on overcoming the performance-oriented challenges. Aiming to overcome the optimisation challenges, the field of AI continues to evolve, and researchers are constantly exploring new avenues to enhance the capabilities of artificial intelligent systems. One promising area of research is Spiking Neural Networks (SNNs) [8]. Spiking Neural Networks (SNN) are new class of neural networks inspired by the biological structure of the brain, and through simulation of behaviour of biological neurons, it aims to capture the temporal dynamics of information processing. SNNs have proved to be on par or better than ANNs in terms of performance, while being significantly more energy efficient in some domains [9, 10, 11]. Its ability

to improve energy efficiency without compromising performance paved a way to solve other optimisation challenges such as training methods. For instance, the training method Back Propagation Through Time (BPTT) [12] is an approach that yields high performance, however it is not efficient in terms of memory and time. As an alternative, Forward Propagation Through Time (FPTT) [13] was proposed to improve the efficiency.

Similarly, focusing on improving the architecture of SNNs for tasks that require intricate time-dependent processing, Liquid Time-constant Networks (LTC) [14] were coupled with SNNs. LTCs are continuous-time neural networks determined by Ordinary Differential Equations (ODEs) that are effective for modeling time series data. They are defined by an input-dependent, varying internal time-constant. This parameter aids in improving performance at at complex tasks [14], and offers better degrees of expressivity in terms of *trajectory length* [15]; it measures how the output of a network changes as the input sweeps along a one-dimensional path. This defining parameter, Liquid Time Constant τ, is integrated into SNNs to give rise to Liquid Spiking Networks (LSN) [16] which excels in modelling time series data while being energy efficient. This was further improved by Bojian Yin et al. through implementation of relatively novel training method known as Forward Propagation Through Time (FPTT) [16]. This method of training offers a significant memory benefits by reducing memory complexity and provides efficiency by reducing the time taken to train.

Invention of LSNs was one of the significant achievements in this area of research, and requires a holistic evaluation before we proceed to improve the architecture. While LSNs have been assessed using visual event stream data, such as S-MNIST and CIFAR10-DVS [17], their performance on other types of event stream data, particularly audio, remains unexplored. Therefore, through this research, the primary aim is to evaluate the performance of these networks on audio event stream data to gain a more complete understanding of their overall capabilities. Following this evaluation, the goal is to enhance the energy efficiency of the LSN architecture. We aim to achieve this by modifying the model architecture and making the network dynamic. Though there are different types of dynamism that could be installed to the network, our primary focus involves making the size of the network dynamic through pruning and regeneration. Through this end, we aim to provide an answer to our second research question: How does the ability to choose connections of neurons influence task performance and efficiency? By answering these questions, we aim to provide directions on how LSNs can be improved.

1.1 Research Questions

To summarize, this thesis focuses on the following problems:

Q1. How does the Liquid Spiking Neural Network model perform on audio event stream classification tasks?

Q2. How does the ability to choose connections of neurons influence task performance and efficiency?

1.2 Thesis Outline

In the following section, through Background Literature, we discuss history of neural networks that lead to the development of SNNs, in all its different aspects. In the following section, we provide a comprehensive overview of the methods to answer the research questions. Based on the methods,

in the consecutive section Experimental Setup, we provide the details on how the experiments are conducted. In the penultimate section, we display the results and provide our analysis of it. In the final section, we summarise the research, present a discussion of the results and provide directions for future work.

2 Background Literature

The history of neural networks goes back decades, beginning with a simple mathematical representation of an information-processing neuronal unit. As research and practical applications have expanded, our understanding of neural networks has grown significantly. As time has passed, some concepts have been modified, while others have been redefined; it has become a necessary condition to define the framework within which we are working. In this section, we discuss the essential elements of this framework to provide the reader with a basic understanding of the topic at hand.

This section is divided into four parts: Information representation, Architectures, Training methods, and Dynamic Networks. In the first quarter, we delve into the details of how information is represented, processed, and the influence it holds on the performance of the network. In the subsequent subsection, we discuss architectural aspects, their evolution over time, and the differences in their skeleton and implementation. Following the discussion on architectures, we describe the methods to train them. In the final subsection, a brief history of how to make networks dynamic is summarised.

2.1 Information Representation

It is a well-known fact that neural network models are often considered black boxes, implying that understanding the intricate workings of the system is difficult. These black boxes vary in size depending on the complexity of the task. As the task's complexity increases, so do the number of hidden units, the number of hidden layers, and the number of operations performed, adding to the difficulty in understanding the processes. Despite this difficulty, plotting the learning curve provides us with insight into the intricacies of the learning process. This insight enables us to fine-tune the model, making it possible to capture the underlying data distribution and learn the parameters that influence performance. Through this process, influential parameters can be identified and used to improve the models. However, the degree of improvement in performance depends on the model's intrinsic parameters and is limited. In other words, fine-tuning a model improves performance and the ability to generalize only to a certain extent, beyond which it becomes highly challenging.

With a lack of understanding of how parameters influence performance, optimizing the model can become a daunting task. Alternatively, optimization can be performed on the data to facilitate easier learning. While optimization aims to achieve better performance with minimal error, there are various pathways to achieve the same goal. Some commonly used techniques include data augmentation and feature engineering. Through data augmentation, data is distorted or manipulated to a certain extent with noise so that the model trained on this data will be robust to noise. Feature engineering, on the other hand, enables model optimization by utilizing extracted features from the data. Although these approaches may seem to differ in their algorithms, they share a common purpose: manipulating data representation to facilitate learning. In the former technique, noise that represents different data aspects is introduced into the actual data, and leading to a new representation of the data. In the latter technique, data representation is modified by extracting features. In this fashion, these techniques, including a few others, manipulate data representation to optimize learning.

While these techniques modify the data representation to a certain extent that enables the model to establish better performance, they can be further enhanced. Further enhancement requires the data representation to be modified at a deeper level, which is achieved through the manipulation of

the information representation. Before proceeding further, there is a need to differentiate, define, and provide a description of the representations. Firstly, despite the usage of the terms 'Data' and 'Information' interchangeably, there is a difference between them. Information is the essential and utility-filled content, while data is the mere organization of the same information in a manner that allows easier interpretability. Simply put, data refers to the fabric, and information refers to the nature of the fabric.

For instance, consider the number zero. Its face value is zero, which is the information it carries. The data representation of zero is its written form, *0*. When we modify the data representation, we change how *0* is visually represented, such as by representing it as *00* in binary. This modification is at the surface level and doesn't change the fundamental meaning of zero. However, when we manipulate the information representation, we operate at a deeper level. For example, instead of representing *0* as zero face value, it could be represented as *lack of value*, which changes the fundamental interpretation of '0' itself. Consider another yet better example: an image from the MNIST dataset that represents the digit '0'. Through augmentation, or pooling, or other techniques, we transform the data representation of this image, making it appear slightly different while still representing the digit '0'. But if we change the information representation of this image, we could transform it into integers, completely altering its meaning from a handwritten digit to a numerical value.

Human brain is known as the most optimal neural network due to its efficiency at various tasks, and its efficiency is attributed to how the information is represented and processed. Neurons in the brain process and communicate via action potentials, or *Spikes*, which are electrical impulses. Taking inspiration from this, Spiking Neural Networks (SNNs) [18] were developed. In SNNs, a *Spike* is binary value indicating information transmission through synapse between two neurons. These are processed along the time dimension, with varying intensity and sparsity, resulting in information procession. As the spikes are produced along time scale, intensity describes the frequency of the spikes at particular point in time, and as the spikes are not emitted at all times, it induces sparsity. A more detailed explanation on the working nature of SNNs can be found in the subsection 2.2. Before the network of neurons process the information, it is first necessary to modify the information representation. The modification can take place from any format including numerical values, images, or audio, to spikes. There are online resources that make the transformed datasets readily available. In our case, we use the *tonic* platform that readily provides converted data to a range of tasks. An alternative to *tonic*, to encode information as Spikes, various spike conversion software can be used depending on the input format, and for audio data, *Lauscher* [19] has been used in the past. The converted spike data has the same informational value as it had before, but the representation has been modified. This data however does not have a structural integrity, i.e., it contains spikes at random minuscule time points. Therefore, there is a need to give the data a structured format so it can be processed by SNNs sequentially.

While structuring the data is necessary for processing the information, it is also necessary to structure the outputs as to interpret the information. To aid with these procedures, there are three information encoding and three information decoding schemes reviewed by J. K. Eshraghian et al [20].

2.1.1 Input Encoding

There are three popular encoding techniques that are used to encode spike data:

1. **Rate coding**: It represents number of spikes that occur per time interval. Through this encoding, any event in time is split into to a series of multiple time intervals, and the number of spikes per interval are calculated.

2. **Latency coding**: It encodes the data as spikes occurring at discrete points in time.

3. **Delta modulation**: The change in the information is encoded as a spike. To elaborate, as the neurons process the information in time, there exists a user-defined threshold for change, and when the change in information exceeds the threshold, it is encoded as a spike.

2.1.2 Output Decoding

For the tasks involving classification, it is necessary to decode the output spikes to allow for an easier interpretation. There are three ways to achieve this:

1. **Rate coding**: The neuron with the highest firing rate or spike count is considered to be the output class.

2. **Latency coding**: The first neuron to produce a spike is the predicted class.

3. **Population coding**: Multiple neurons per class applying the aforementioned schemes.

Apart from the aforementioned encoding mechanisms, there are less researched methods of encoding that include using spikes to represent a prediction or reconstruction error [21]. While these are the encoding methods that were proposed to enable easier interpretability, there is no evidence that the human brain uses them. Disregarding the bio-plausibility, each type of encoding has its own advantages: While rate codes have better error tolerance and promote more learning through stronger gradients (due to additional spikes), latency codes are energy efficient as they consume less power.

It has been proven that at certain tasks, SNNs exhibit higher performance while being energy efficient [9, 10, 11, 22]. A natural deduction of reason for such superiority reveals either information representation or information propagation through the architecture. Through this subsection, we have discussed the importance that information representation holds, and in the following section, information propagation will be discussed, covering the one of the aspects that make SNNs superior.

2.2 Architectures

The functioning of Artificial Neural Networks (ANNs) has been well-established. To recap, when an input is passed to a neuron or a system of neurons, it is multiplied by the weight matrix of the connections from the current layer to the next layer. The resulting output is then summed with the bias of the current layer to obtain the output of that layer. This output is subsequently passed through an activation function, such as *ReLU*, *Sigmoid*, or others. The activation function produces the final output, which is then passed to the next layer. This process is repeated across multiple hidden layers until the final output layer produces a result that can be compared with the true outputs. The calculation of error and its subsequent backpropagation to facilitate model learning is discussed in the next subsection 2.3. It's important to note that the activation function plays a pivotal role in processing information, as it determines whether and to what extent a neuron 'fires'. While the neuron itself holds the weight of the connection, the activation function dictates the amount of information to be

propagated to the next neuron by controlling it.

While ANNs have been traditional in usage for decades now, there is a new generation of neural networks that operate differently, focusing on biological plausibility and energy efficiency. These neural networks are known as Spiking Neural Networks (SNNs) and they are inspired from the working nature of the brain. They are similar to ANNs in that, there are a certain number of neurons per layer, a few layers that are interconnected, and they combinedly work to result in information procession. However, the definition of the neuron for these networks has been redefined to achieve energy efficiency and bio-plausibility. Through this subsection, we delve deeper into the different type of neurons and how they evolved since their inception.

Spiking Neural Networks emerged from the concept of Leaky Integrate and Fire (LIF) Neurons. LIF is the simplest and most widely used single neuron model that was initially developed by Lapicque [23]. The dynamics of the LIF neurons is straightforward; a neuron is defined by the membrane potential u that changes with time t depending on the input current I and it's mebrane resistance R. As the membrane potential accumulates over time, it is bound to cross a pre-defined threshold θ at time t_s, which produces an output spike. Post emission of the spike, the membrane potential of the neuron is reset to a baseline value known as resting potential u_r. This whole concept can be summed up as the equations 1 and 2 below.

$$T_u u(t+1) = \begin{cases} u_r, & \text{if } u(t=t_s) > \theta \\ -(u(t)-u_r) + RI(t), & \text{otherwise} \end{cases} \tag{1}$$

$$s(t) = \begin{cases} 1, & \text{if } u(t=t_s) > \theta \\ 0, & \text{otherwise} \end{cases} \tag{2}$$

where T_u represents the time constant whose value is 10ms, and the second equation represents the working mechanism for the spike as function of time; a spike is produced when the membrane potential crosses the threshold at time t_s.

LIF is the first one in this new generation of neurons that lie close to the biological neurons in terms of their function. Despite the similarity between LIF and an actual biological neuron, the former fails to produce various firing patterns observed in the latter, such as adaptive firing, delayed firing and others [24]. To close the gap between them and to improve LIF in its function, research was initiated by [25]. This research laid the foundations for Adaptive LIF (adLIF) neuron, and with further research from [26, 27, 28], the formulation became more precise. The improvement offered through the research was that Adaptive LIF (adLIF) has a threshold adjusting mechanism that is triggered after a spike is emitted. The mechanism implements a recovery variable *w(t)* that is linearly coupled with the membrane potential, which aids in modifying the threshold. The resulting dynamics of *adLIF* neuron follow the equations 3 and 4 below:

$$T_u u(t) = -(u(t)-u_r) + RI(t) - Rw(t) - T_u(\theta - u_r)\Sigma\delta(t-t_s) \tag{3}$$

$$T_w w(t) = -w(t) + a(u(t)-u_r) - T_w b\Sigma\delta(t-t_s) \tag{4}$$

where T_w is a longer time constant of value 100ms, and b is a variable that represents the jump size of the recovery variable following $w(t_s) = w(t_s) + b$.

An improved dynamics can be observed through *adLIF* as the threshold is calibrated and adjusted dynamically. However, since the relation between the recovery variable w and membrane potential u is linear, the range of the dynamics can be further improved by modifying the equations to have a non-linear relationship. This allows the threshold adaption mechanism to be smoother, and is a step towards biological plausibility. Proposed by Brette and Gerstner [29], Adaptive Exponential Integrate and Fire (AdEx) uses a combination of linear and exponential functions, followed by the equation,

$$f(u) = -(u(t) - u_r) + \Delta exp[-\frac{\theta - u(t))}{\Delta}] \tag{5}$$

From all the neuron models described above, *AdEx* is the most physiologically plausible in terms of fitting with naturalistic pyramidal-neuron voltage traces [30].

Regardless of the type of neuron model, all the aforementioned neuron models have distinguishing features that make them spiking neurons rather than artificial neurons. These distinguishing features include the integration of the activation function directly into the neuron's design, the nature of the information they utilise, and the distinct manner in which the information is processed.

To elaborate, firstly, the activation functions in ANNs such as *Sigmoid* are declared separately. At first glance, this might not seem to make a significant difference, as the output from the hidden layer goes through the activation layer immediately. However, declaring the activation layer separately results in a uniform activation function for all the neurons in the layer. This can be disadvantageous, as it can be restrictive for neurons to exhibit the same activation behavior. In SNNs, there exists an internal hidden state, *membrane potential* whose function is similar to that of activation function. Since the membrane potential is a function of input spikes, neurons receiving different input spikes affect their respective membrane potential differently. As a result, they exhibit different activation behaviors. Apart from differences in activation dynamics, the outputs post-activation also vary. While ANNs produce a range of values from -1 to 1, or 0 to infinity in the case of the ReLU (Rectified Linear Unit) activation function, SNNs output binary values, i.e., either 0 or 1, depending on their firing state. The difference in these outputs marks the difference in the type of information. It is also important to notice that the information type in ANNs lie in a range of values at input layer ranging from negatives to positives, where the values can be either integers or floats. However, in SNNs its strictly binary regardless of the layer.

In regards to the information processing, SNNs operate in time domain, i.e., the processing of input spikes or the production of output spikes is non-linear and can occur at different time points. Conversely, in non-recurrent ANNs, information is continuously produced and propagated across all layers simultaneously. These distinguishing factors makes SNNs different than ANNs and are suitable for event driven information processing. However, the spike function used by SNNs is non-differentiable, making the usual training methods for ANNs inapplicable to SNNs. They either require a modification in the existing training methods or need a novel training methods, which will be discussed in the next subsection 2.3.

2.3 Training

The training of Spiking Neural Networks (SNNs) involves a different approach compared to Artificial Neural Networks (ANNs). The training methods traditionally used for ANNs, excluding those

for RNNs like Back Propagation Through Time (BPTT), do not apply to SNNs due to their distinct modus operandi. The reason for the usage of novel approaches can be traced back to two elements of training: Information processing and loss optimisation. In traditional ANNs, the information is processed along the spatial dimension, however in SNNs, the information is processed in the temporal dimension. In addition to this, the information propagation in ANNs and SNNs is dissimilar, as the information is of continuous values in the former and discrete in the latter. The problem is realised during loss optimisation due to the non-differentiable nature of the spike function in SNNs. Multiple approaches have been developed to overcome the aforementioned problems, which will be discussed in this section.

One of the earliest and foundational training methods developed to overcome these problems was Spike-Timing-Dependent Plasticity (STDP). The rule was first put forth by Dan and Poo [31], and is rooted in neurological principles. It is an unsupervised learning algorithm that relies on the timing of presynaptic and postsynaptic spikes. For any given synapse, if a presynaptic spike elicits a postsynaptic spike within a time window, then the strength of the synaptic weight is enhanced [32]. Conversely, if a postsynaptic spike is produced before a presynaptic spike, then the synaptic weight is weakened. The adjustment in the strength of the synaptic weight is a function of time between the presynaptic and postsynaptic spike events. However, when observed in the human brain, STDP is combined with reward-based learning that involves neuromodulators in the brain. To enable the usage of the algorithm for supervised learning tasks, Bengio et al [33] introduced a formula for weight update that is expressed only in terms of firing rates and their derivatives. This reward-modulated STDP method was used in supervised learning tasks, and the working nature of the algorithm is similar to the Stochastic Gradient Descent and Back Propagation.

Recently, several series of experiments were conducted aiming to evaluate the performance of SNNs trained using STDP, both supervised and unsupervised, on a range of tasks such as FashionMNIST and CIFAR10 [34]. This research led to a conclusion that the SNNs trained with STDP alone are inefficient and hardly achieve a high performance of SNNs [34]. It was also declared that the performance does not improve even upon integration of other techniques. Based on this, it can be stated that better methods need to be implemented to improve the performance. Ignoring the performance, despite the extension to supervised techniques, there are certain limitations of the algorithm, including the need for complex hardware implementations and difficulties in training deep SNN architectures.

Training SNNs with Stochastic Gradient Descent is not an alternative because of the non-differentiable nature of the spikes, however, there are a variety of approaches that ease the implementation through modifications. These approaches can be classified under three titles: Hybrid ANN-SNN models, Differentiable models, and Surrogate Gradient models.

The primary category, Hybrid ANN-SNN models, bypasses the training problem by using conventional ANN models. These models exhibit spiking behaviour during forward pass, but the timing of spikes are ignored and are not included in the learning rules. Recently, by coupling SNN with ANN through layer-wise weight sharing, Wu et al [35] have successfully found a way to efficiently backpropagate the gradients. The forward pass in SNN and ANN goes in parallel, where the SNN computes the spikes while the ANN calculates the corresponding spike count or its approximations. During the backward pass, the error is backpropagated only in the ANN, and after its weights are updated, the weights are transferred to the SNN. Despite the approach proving to be successful in performing speech recognition tasks [36], it has certain shortcomings; it is only applicable to SNNs

with non-adaptive neurons. This translates its non applicability to adaptive neuron models. In terms of the training method, the spike times are ignored by being reduced to approximations as spike count.

Differentiable models have a characteristic that is defined by their formulation which ensures well-behaved gradients, that are directly suitable for optimization. They can be categorised into four categories: soft nonlinearity models, probabilistic models, rate coded models, and single-spike temporal coded models.

Soft nonlinearity models, featuring a smooth spike-generating process, are applicable across all neuron models discussed in the previous subsection 2.2. This is achieved by employing a continuous function to estimate the firing probability of a neuron at any given point. This approach has been successfully implemented once by Huh and Sejnowski [37], where the binary spiking non-linearity was replaced by a continuous-valued gating function. As a result, the network had the ability to be optimised by ANN methods such as BPTT. These models cannot be classified as Spiking Neural Networks as they compromise the characteristics by replacing the binary spikes with continuous-valued duplicates.

An alternative to the soft non-linearity models is the probabilistic models, where the discontinuous binary non-linearity is smoothed out by stochasticity. These binary probabilistic models have been under research for many years in the field of machine learning, in the context of restricted Boltzmann machines [38]. The propagation of gradients in the same context has also been ongoing [39]. These models are useful because of the log-likelihood function that smooths out the spike train. The log-likelihood function transforms the discontinuous and binary nature of spikes into a probabilistic framework, where the likelihood of firing is a smooth, continuous function of the input. While using these models to study the biologically plausible ways of propagating error, it was found that variational learning approaches were capable of learning useful hidden-layer representations in SNNs [40, 41]. Variational learning is a method that focuses on estimating probability distributions and model parameters in a way that's computationally tractable and efficient. It's often used in the context of complex models where direct computation of probabilities and likelihoods is challenging. However, the research also highlighted a significant challenge: the noise introduced to smooth out the process often complicates optimisation efforts [41].

Another approach to overcome the non-differentiable spike function problem is to obtain gradients through rate-based coding scheme. The concept involved can be summarised as the usage of spike rate as the information carrier rather than the individual spikes. For most neuron models, the threshold and the firing rate is dependent on the neuron input. This input-output dependence is captured by the f-I curve of a neuron, which enables usage of gradient-based optimisations [42]. Using this rate-coded scheme, Neftci et al. demonstrated competitive performance on various datsets such as CIFAR-10 and MNIST [43]. Despite the performance, this approaches may be inefficient as it requires estimation of firing rate which is an average of the number of spikes. Furthermore, to average out the discretisation noise, it is necessary that the average to have relatively high firing rates or long averaging times.

Optimising SNNs without injection of noise or without reverting to rate-based schemes is a challenging task, and there are a few studies who approached the problem by considering the outputs of neurons to be a set of firing times. This temporal coding is a better approach than the rate-based coding as instead of the count of spikes, individual spikes are utilised for processing, making it more

meaningful than the average. The idea behind the working of temporal coding networks was put forth by Bohte et al. [44]. The algorithm tracks the exact times when spikes occur in the network, as it is crucial for encoding information in spiking neurons. Error is calculated using the actual spike time and the expected spike time. Based on the calculated error, the network's weights are adjusted to minimise the temporal difference. Although the spike timing formulation did yield well-defined gradients and a good performance, it suffers from certain limitations. For instance, each hidden unit was required to produce only one spike per trial. It was necessary because it is difficult to define the firing times for inactive units. Therefore, this algorithm might suffer from not being energy efficient because of such formulation.

Having discussed the different differentiable models, it can be observed that each of the training method has a limitation of a different kind despite approaching the problem through approximations in training. Further research into the training methods, trying to overcome the problems mentioned before, led to the invention of Surrogate Gradient methods. Surrogate Gradient methods not only provide an alternative approach for overcoming the difficulties associated with the discontinuous nonlinearity, but also provide a path to to reduce the high algorithmic complexity usually involved in training SNNs.

The differentiating factor of these methods is that rather than modifying the model definition as observed before in smoothed approaches, here, a surrogate gradient is introduced that is a smooth, continuous transformation of the spiking nonlinearity, used for optimization purposes. Surrogate Gradients can be defined in two ways: the primary one which constitutes a continuous relaxation of the binary spiking nonlinearity for optimisation purposes. Another definition introduces surrogate gradients that explicitly influence the underlying optimization algorithms themselves, improving the computational efficiency of the learning process. Regardless of the type of Surrogate Gradient method, they enable training of SNNs in an efficient manner, without needing to specify the coding scheme.

Surrogate gradients aim to reduce computational overload by affecting the locality of update rules. The research involved with these methods is very limited as it is difficult to implement local modifications on neuromorphic hardware, and therefore has been implemented only in simulation. However, Surrogate gradients under the primary definition do not explicitly change the optimisation algorithm and can be used with methods used to train RNNs, for instance, in combination with Back Propagation Through Time (BPTT). The traditional backpropagation algorithm [12] is one of the most powerful technique that was developed which is now widely used. At the core of backpropagation is a method for calculating derivatives efficiently in large systems made up of elementary subsystems that are represented by differentiable functions. Applying the same technique to a system of RNNs would entail Back Propagation Through Time (BPTT) [12]. The procedure involved gets modified slightly as the computational graphs are required to unroll one step at a time. To elaborate, gradients are computed by tracing the error signal backward through time, considering the impact of each network's previous states and weights on the current prediction. While it was initially developed to deal with temporal dynamics of RNNs, it can be applied to other neural networks where there are temporal dependencies are involved. Thus, the BPTT algorithm was applied to SNNs and it was found that the performance closely approaches ANNs in classification tasks on various datasets [45, 46, 47, 48]. Despite the performance achieved, it was also revealed that training using BPTT algorithm suffers from large memory footprint [45].

All the aforementioned training techniques compromise the efficiency of SNNs in one form or another. Therefore, to overcome the problem of training SNNs without compromising efficiency, a novel training method known as Forward Propagation Through Time (FPTT) was adapted for SNNs [16]. The original FPTT algorithm was developed to train ANNs with temporal dependencies such as RNNs and LSTMs, and it showed significant performance on various datasets such as CIFAR-10, and S-MNIST [13]. Based on the performance and the promising energy efficiency, it was adapted to SNNs by Bojian et al [16]. FPTT involves two steps: optimising instantaneous risk function, and updating weights. To elaborate, the forward pass is performed and the loss l_t is calculated using the equations 6, 7 and 8.

$$l_t = \beta l_t^{CE}(\hat{y}, y) + (1-\beta) l_t^{div} \tag{6}$$

$$l_t^{CE} = -\sum_{i=1}^{N} y_i \log(\hat{y}_i) \tag{7}$$

$$l_t^{div} = -\sum_{\bar{y} \in y} Q(\bar{y}) log \hat{P}(\bar{y}) \tag{8}$$

where $\beta \in [0,1]$, l_t^{CE} is the classical cross-entropy loss, and l_t^{div} is a divergence term used as auxiliary loss to reduce the distance between the prediction distribution $\hat{P}$ and target label distribution Q. $\hat{y}$ and y represent the predicted and true values, respectively. N is the total number of classes.

Using the losses recorded over epochs, a dynamic regularisation penalty R_t is added to the loss function to get the instantanrous risk function l_t^{dyn}. The regulariser R_t is calculated using the equations 9, 10 and 11 below.

$$R(\bar{\Phi}_t) = \frac{\alpha}{2} ||\Phi - \bar{\Phi}_t - \frac{1}{2\alpha} \nabla l_{t-1}(\Phi_t)|| \tag{9}$$

$$\Phi_{t+1} = \Phi_t - \eta \nabla_\Phi l_t(\Phi)|_{\Phi=\Phi_t} \tag{10}$$

$$\bar{\Phi_{t+1}} = \frac{1}{2}(\bar{\Phi}_t + \Phi_{t+1}) - \frac{1}{2\alpha} \nabla l_t(\Phi_{t+1}) \tag{11}$$

where α is a regularisation term, η is the learning rate, and $\bar{\Phi}_t$ is a state vector that summarises previous losses. In other words, Φ_t is the weight vector, and $\bar{\Phi}_t$ is the average of the weight vector, calculated using the equations above.

Upon calculation of the instantaneous risk, it is used to update the weights using the equations 10 and 11. After the weights are updated, a new epoch begins and the whole process repeats. In this way, FPTT was employed in SNNs and was evaluated on various datasets such as S-MNIST, PS-MNIST, R-MNIST and DVS-CIFAR10 [17]. The results indicated an improvement in performance compared to SNNs trained with BPTT, while also demonstrating lower memory and time complexity [16]. While this method is efficient, there are ways in which SNNs can be made even more efficient. One of the ways to improve efficiency is reducing the parameters of the network, that is achieved through application of dynamism to the networks. In the final subsection, we discuss the history of dynamism, and the different kinds of dynamism that exist.

2.4 Dynamic Networks

As evident in the previous subsections, all the different elements of Neural Networks have been refined over the years, and their definition redefined. However, all these networks follow a common blueprint: the structure and the parameters are static and unchanging. At first glance, this might appear inherent, but there are limitations to such static models. Research has revealed that as the architecture remains fixed, it limits their representation power, efficiency and interpretability [49, 50, 51]. The limitations can be elaborated as follows: Fixed architectures restrict a network's ability to adapt and evolve in response to diverse or complex patterns of the dataset. This rigidity can hinder the network's capacity to accurately model and learn from intricate patterns inherent in the data, thereby limiting the scope of its learning. Similarly, a rigid architecture results in equal allocation of computational resources across all data points, regardless of their individual needs. This approach leads to an inefficient utilization of resources. The lack of flexibility in the architecture makes it challenging to understand how the network processes and derives insights from the input data, turning the model into a more opaque, less interpretable system. To overcome these limitations, and claim the benefits that the neural networks has to offer, there is a need to make them dynamic. Research was performed to some extent in the past to achieve this, and different methods were proposed to achieve dynamism to different aspects of the system. In this subsection, starting with the definition, we discuss a brief history of research in achieving dynamism.

Dynamic Neural Networks (DyNN) refer to the neural networks that are dynamic in nature, in any of its different working elements; such as training method, structure and parameters. The dynamic nature arises when the network modifies itself based on a particular parameter that is either intrinsic and extrinsic. While the mechanism differs, the goal remains the same: to achieve dynamism. In this manner, we classify the methods into three different categories:

1. Spatial-wise Dynamic Networks
2. Sample-wise Dynamic Networks
3. Temporal-wise Dynamic Networks

2.4.1 Spatial-wise Dynamic Networks

Computer Vision (CV) tasks are some of the most compute-intensive tasks in the world. The computational costs can be attributed to the input image data, as it contains redundant information, and to the design choices that are made when designing the network. Supporting the former statement, it has been found that not all pixels of the input image contribute to the final output, which translates to redundant computation for the same performance [52]. Another observation that supports the findings: a low-level representation is sufficient enough for a decent performance [53]. The core issue is that the redundant information varies from one input to another. To process it efficiently would require a different processing pipeline for each input. Spatial-wise Dynamic Networks (SpDN) are designed to address this by replacing static processing with input-dependent information processing.

Used in combination with Convolutional Neural Networks (CNNs), Spatial-wise Dynamic Networks (SpDNs) exploit the spatial information of input images to perform adaptive inference. These networks offer two major benefits: they adjust the model's architecture to reduce computation and increase efficiency, and they modify the network parameters to enhance representation power while decreasing computational costs. The degree of dynamism categorizes the methods into further groups:

Pixel-wise Dynamism, Region-level Dynamism, and Resolution-level Dynamism.

Pixel-wise Dynamism dynamically adjusts computations at the pixel level, focusing on the unique attributes of each pixel. Region-level Dynamism, on the other hand, adapts computational processes to the specific attributes of different image regions, allowing for varied focus across distinct areas. Lastly, Resolution-level Dynamism modifies the computational intensity based on the resolution of various parts of the image, optimising resource use for areas with differing levels of detail and complexity. As the current research does not pertain to CNNs or image data, we will not delve into the various details and proposed methods related to *SpDN*.

2.4.2 Sample-wise Dynamic Networks

Deriving from the same concept as *SpDN*, Sample-wise Dynamic Networks (SaDN) extend the methodology to include other types of ANNs beyond CNNs. *SaDN* process different inputs differently, either by changing the network's architecture or its parameters. There are two primary types of dynamism that are exhibited by these networks: Dynamic Architectures and Dynamic Parameters.

2.4.2.1 Dynamic Architectures

Dynamic Architecture refers to the ability of a network to adapt its architecture to maintain task performance while lowering computational redundancy. With this as the common goal, numerous techniques were proposed that achieve dynamism by influencing the architecture along its various dimensions. The intuition behind the working of techniques is that it requires different computational capacity to process canonical (”easy”) input samples than it takes to process non-canonical (”hard”) input samples. Therefore, these techniques involve performing inference and adjusting the network architecture accordingly. Exerting an influence on the network architecture is often achieved through two primary ways: Modifying it's depth and width.

Dynamic Depth techniques aim to reduce computation by identifying ”easy” and ”hard” samples at inference and adjusting the network depth accordingly. Dynamic Width techniques follow a similar procedure but the modification is performed on the width of the layers. There are two renowned concepts - Early Exiting and Layer Skipping - through which Dynamic Depth can be realised. The working principle involved in Early Exiting is that ”easy” inputs require less computation as compared to ”hard” inputs, and therefore, ”easy” inputs are processed through shallow networks, while the ”hard” inputs are forwarded to process through more deeper layers [54, 55, 56]. Following a different approach, Layer Skipping employs algorithms that selectively skips the layers based on the input [57]. The concept of skipping is extended into other algorithms that make the network width dynamic. Dynamic width algorithms work by bypassing or skipping the network elements such as neurons, branches or channels. Skipping Neurons is a straight-forward approach where certain neurons are not activated with the belief that these neurons might represent relatively insignificant information. This can be achieved by adaptively controlling the neuronal activations through auxiliary branches [58] or low-rank approximations [59]. Skipping Branches, also known as Mixture-of-Experts (MoE) [60, 61], follows a relatively bigger scheme, where multiple network branches are trained in parallel and a selective of them are executed at run time. The outputs are fused with data-dependent weights to obtain the final outcome. Skipping Channels, as the name implies, skips a number of channels aiming to reduce channel redundancy in CNNs.

2.4.2.2 Dynamic Parameters

Dynamic Parameters refer to the adaptation of the model's parameters based on the input. This is achieved while keeping the architecture of the model rigid, which in turn improves the representation power. Popular methods of implementation include adjusting the trained parameters, and generating and scaling the parameters.

One key method of parameter adjustment is the application of soft attention, which involves selectively focusing on certain parts of the input data and adjusting the weights accordingly. Soft attention mechanisms assign varying levels of importance to different input features, allowing the model to prioritize more relevant information. For instance, by applying locally masked convolution, segmentation-aware convolutional network [62] integrates information with larger weights from similar pixels, i.e, the network selectively emphasizes certain parts of the input data based on their relevance. In this process, the model assigns more significant weights to pixels that are similar to each other, enhancing the focus on these areas.

2.4.3 Temporal-wise Dynamic Networks

Apart from the dynamism that exploits the spatial information to adapt the network parameters, there exists another dynamism which exploits the temporal redundancies in sequential data and adapts the network parameters depending on it. Temporal-wise Dynamic Networks are usually built with RNNs and applied to all kinds of sequential data such as text and video.

The dynamism involved in processing text data differs from that in video data, partly due to the nature of the data. In dealing with textual data, traditional RNNs process tokens sequentially and update the hidden states at every time step. However, not all tokens contribute equally to learning and performance, leading to a degree of redundancy. Temporal-wise Dynamic Networks aim to minimize this redundancy through adaptive computation algorithms.

Skimming is one of the methods that is employed to reduce redundancy, and it works by skimming less informative tokens and updating the hidden states with cheaper computation. It involves selectively focusing on more informative tokens within the data while bypassing or "skimming" over those considered less informative. This method has been implemented a few times such as in Skim-RNN[63] and others [64], where a subset of dimensions of hidden states are calculated and the remaining are copied from the previous step. To achieve the partial update, selective rows in weight matrix are dynamically activated, or a choice is made between two independent RNNs. The concept of Halting score [65] can also integrated into these networks to further limit the redundancy and perform limited updates. As skimming only reduces the amount of computation by certain extent, an alternative to it is proposed where the computation is eliminated for certain cases.

Through Skipping, all the updates for unimportant inputs at certain points in time are skipped and thereby limiting the number of computations performed. Skip-RNN [66] is the first to employ this where a controlling signal is updated in every step to determine whether to update or avoid the hidden state. Structural-Jump-LSTM [67] is another neural network which deploys an additional agent that makes the skipping decision conditioned on the previous state and the current input. Another line of work [68], follows a different approach by training a predictor to estimate whether each input will contribute to the change in hidden state, and thereby eliminating the usage of controllers.

The nature of textual data is hierarchical and exploiting its intrinsic structure, hierarchical RNNs were developed [69]. These models use a dynamic update mechanism and encode the temporal dependencies that lie within the data. There are high level RNNs whose procession of data is dependent on the outcome of the low level RNNs; in this manner, high level RNNs do not process information all the time but rather selectively. Through this, a significant amount of processing can be avoided leading to the reduction in the overall computation involved. Other mechanisms such as Early Exiting can be integrated into these techniques to further reduce the redundancy in computation. However, Early exiting can also induce some computation as all the tokens have to be fed to the model. This is avoided in LSTM-Jump [70] which learns to decide *where to read* by skipping some tokens strategically without reading them, and directly jumping to next point in time where the token is important. It is implemented by deploying another agent which predicts the size of the jump within declared range and the mechanism runs until a prediction of zero is obtained, signifying the end of the sequence. Extending on this, LSTM-Shuttle [71] allows for backward jump to supplement for the missed history in the sequence. Through all of the above mechanisms, a dynamism in temporal procession of data can be induced with a varying degree of reduced redundancy.

Concerning the visual data, video input consists of sequence of frames where each frame is processed through same procedure, resulting in a computational overload. To avoid this, Temporal-wise Dynamic Networks can be deployed which allocates varied computation for various frames of the video.

The working of Temporal-wise Dynamic Networks for visual data occurs through two major pathways, either through dynamic updates or pre-sampling. The concept of Dynamic updates mentioned above is implemented for visual data with a minor modification to process visual data rather than textual data. *Skimming* is replaced with *Glimpse*, where unimportant frames are allocated lower computational cost. *Jumping* technique is modified to learn *where to see* from the original *where to read*.

The original implementations of the aforementioned techniques can be observed in a range of research work. For instance, LiteEval [72], ActionSpotter [73], and AdaFuse [74] implement the same techniques with minor variations in order to attain the same goal. Through LiteEval, a decision has to be made between two LSTMs that vary in computational costs. Whereas the decisions in ActionSpotter are input-dependent, based on which the hidden states updates are made. AdaFuse, with the aim to efficiently make use of historical information, reuses selective feature channels from previous step. As an extension to this line of work, further research has proposed to adaptively determine the numerical precision or modalities while processing the sequential frames [75, 76]. In regards to Jumping technique, it is similar to the Jumping seen in processing the textual data, where the model learns to predict the location where the jump should be made to, at each time step.

2.4.4 Pruning

Pruning is one of the oldest and well-known techniques used for achieving dynamism in neural networks. Through pruning, an initially large network is reduced to a relatively smaller network with little or no drop in performance.

In the paper 'Dynamic Network Surgery' [77], the authors propose a method to reduce network complexity by making on-the-fly connection pruning. Unlike the previous pruning methods, the

proposed techniques also allows for recovery of connections for the case where the connections are incorrectly pruned.

To learn a sparse model from a dense reference by abandoning unimportant parameters and keeping the important ones, it is crucial to measure parameter importance. However, it can be difficult due to mutual activations and the influence they exert. This underscores the importance in maintaining the network structure continually through the learning process. This cyclic process of maintenance is akin to the synthesis of excitatory and inhibitory neurotransmitters observed in the brain.

Firstly, the importance manifests as state of the connections with a binary matrix T. There are two possible states 1 and 0, referring to the important and unimportant connections, respectively. The matrix T is initialised with all ones, and through the update scheme of the weights, this matrix is also updated. Since this does not involve actual removal of connections but is a mere imitation of the connection removal, T is also referred to as the mask matrix. Updating the mask matrix is done in parallel with the weight updates such a way that the combination of the weight matrix and the mask matrix enables minimization in loss. The actual update of the mask matrix happens through a discriminative function h_k applied to the network parameters. The function operates in three intervals, i.e., if the network parameter is below threshold a_k, then the respective entry is made 0 (unimportant hence connection pruned), else if it is above threshold b_k, the respective entry is made 1 (important hence connection spliced), and if it lies between a_k and b_k, then no change is made to the connection. In this way, the connections are continuously pruned, spliced and maintained, making the network dynamic. Implementation of this method resulted in a compression of 108x and 17.7x on LeNet-5 and AlexNet, without any drop in performance. The prediction error of LeNet-5 was 0.91% while it was 19.99% for AlexNet.

Analogous to this method, another method was developed which is similar to the concept of plasticity in the brain and is applicable to various kinds of Spiking Neural Networks [78]. The implementation takes place through three individual mechanisms, that together enable plasticity to the networks. The primary mechanism is the synaptic constraint algorithm that creates boundaries for each synapse. The boundary is formed by establishing an positive boundary and negative boundary. These boundaries are defined by the synaptic activity of the synapse such that higher synaptic activity expands the boundary and lower synaptic activity contracts the boundary. Through this mechanism, active synapses are promoted, while decayed synapses are limited. Synaptic activity is measured for synapses between neurons in the current layer and neurons in the previous layers.

To elaborate, when a synaptic weight is higher than the positive boundary consecutively for T number of times, then it signifies that there is a higher activity for the synapse, and as a result, synaptic boundary increases. The increase in synaptic boundary is an average of difference between synaptic boundary and synaptic weight. Similarly, when a synaptic weight is lower than the negative boundary through multiple epochs, then it implies that the synapse is of low activity, and the boundary shrinks. The decreased value is calculated in the same manner as for the positive boundary.

Using the activity level of the synapses, the second mechanism prunes neurons that have lower synaptic activity. To prune the neurons, all the synaptic activities for a neuron are summed, and the final value provides an overall synaptic activity for a neuron. Depending on the predefined pruning rate ρ_p, the weights of lowest synaptic activity neurons are made zero. In this way, the connections

are pruned for the neurons that are used relatively fewer times. The third mechanism implements regeneration of the connections, and it is dependent on the gradients with respect to the loss. In every epoch, we calculate the gradients for each synapse, and for the pruned connections, the weights with highest gradient values are restored. The amount of connections regenerated is predefined through the variable regeneration rate ρ_g. In this manner, all the pruned connections which are of higher importance are restored, and others are dissolved.

Apart from these mechanisms, there are minor ways in which dynamism can be induced to the overall algorithm. It is achieved by making the pruning rate ρ_p and regeneration rate ρ_g dynamic. After pruning is performed, pruning rate is adjusted based on the number of neurons in the current and next layer, as follows:

$$\rho_p = \rho_p + \delta \frac{N^l}{N^{l+1}} \tag{12}$$

$$\delta = \begin{cases} \alpha exp^{-(epoch-START)} & epoch \leq MID \\ \beta & epoch > MID \end{cases} \tag{13}$$

where N^l and N^{l+1} are neurons in the current layer and the next layer, respectively. α and β are constants whose values lie in the range [0, 1], and epoch, START and MID are current epoch, epoch where pruning starts and epoch where pruning slows down, respectively.

The regeneration rate is updated differently, not dependent on the number of active neurons. The update is a non-linear and occurs after regeneration of connections takes place. The updates for the regeneration rate follows the equation below:

$$\rho_g = \rho_g + \gamma^{epoch-START} \tag{14}$$

where γ is a rate constant with a value of 1.1. The variation of synaptic activity at neuron level affects the connections in such a way that the less activated connections are pruned and when the loss of the pruned connections is high, the connections are regenerated to avoid loss. This enables the architectural dynamism in networks which is akin to the neurogenesis observed in the brain.

3 Methods

The research conducted as a part of this project is experimental and exploratory, and intent is to provide a conclusion on the aforementioned research questions. To this end, in this chapter, we provide the methodology for the experiments that are to be conducted, and the intuitive and research-based reasoning to the methods.

3.1 Data

The patterns observed in the data of video event stream classification tasks such as S-MNIST and CIFAR10-DVS [17], do not extend to audio event stream classification tasks. Thus, as the next step towards getting a holistic benchmark of the LSN architecture, we choose a audio event stream classification dataset and evaluate the model in all its aspects. While there are two publicly available datasets for audio event stream classification tasks, namely Spiking Speech Commands (SSC) and Spiking Heidelberg Digits (SHD) [19], our dataset of choice is the latter.

The SHD dataset is an audio dataset that is later converted to Spike trains using third-party software known as *Lauscher* [19], an artificial cochlea model. The initial audio dataset contains over 10,000 high-quality audio recordings of 12 distinct speakers speaking the digits 0 to 9 in both English and German. The training set contains over 8000 samples of 10 speakers, while the testing set contains over 2000 samples of the remaining 2 speakers. Each of these samples lies in the range of 0.24 seconds to 1.37 seconds. Audio data was transformed into spikes of 700 input channels through Lauscher. A sample spike train is visualised in the figure 1 below.

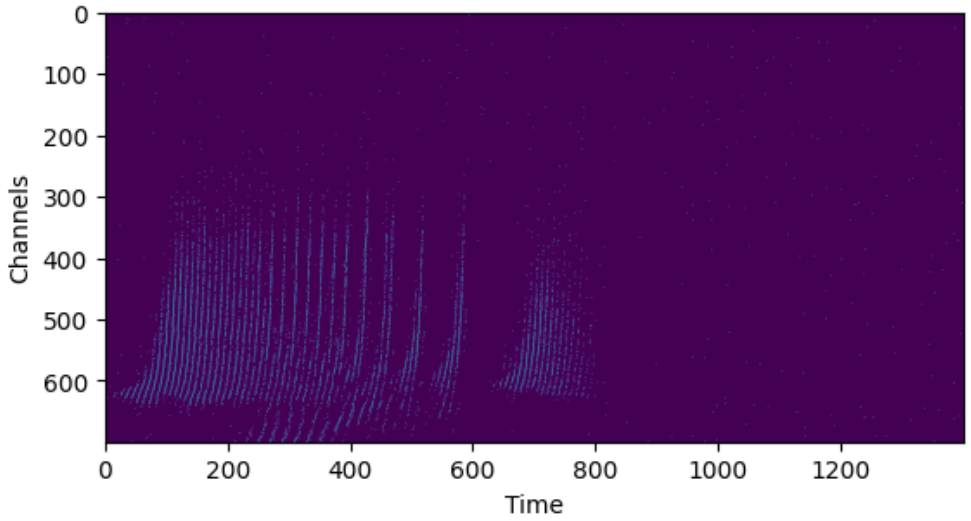


Figure 1: Sample Spike Train from SHD dataset

3.2 Model

3.2.1 Baseline

Liquid Spiking Networks (LSN) is built upon Spiking Recurrent Neural Network (SRNN) that is coupled with Liquid Time Constant (LTC) parameter. SRNNs are used for processing time-series data,

and the LTC parameter aids in processing time-series data better as it dissects the time into little slices which are then processed by the network. As the data is processed in little slices in time, the model is said to improve its performance by improving memory retention. However, the model was evaluated on Video event stream data and memory retention tasks, but the different underlying patterns in data makes it unsure if the same holds for audio event stream data. Therefore, we aim to use the model and establish a benchmark on audio event stream classification task using the SHD dataset.

To elaborate the model from the scratch, the building block of the model architecture is the spiking neuron that has recurrent connections. As the task being dealt with is an audio event stream classification task, usage of RNNs will be beneficial. The usage of recurrent connections here serves two purposes: to retain memory of history and enable loss optimisation for the spiking neurons. The recurrent spiking neuron is coupled with two important parameters known as Liquid Time Constants (LTCs). The parameters are membrane time constant τ_m and adaptation time constant τ_{adp}, that are used to update the membrane potential u_t and adaptive threshold θ_t, respectively. The importance of integration of Liquid Time Constants into the model is that it enables to slice the data in time into minute parts and help the model learn the intricacies underlying the data. So, in the implementation of the model, this entails as small and adaptive updates of membrane potential and the threshold of the neurons.

To summarise, the implementation of LSN follows the following equations:

$$\rho = \tau_{adp}^{-1} = \sigma([x_t, b_{t-1}]||W_{T_{adp}}) \tag{15}$$

$$\tau_m^{-1} = \sigma([x_t, u_{t-1}]||W_{T_m}) \tag{16}$$

$$b_t = \rho b_{t-1} + (1-\rho)s_{t-1} \tag{17}$$

$$\theta_t = 0.1 + 1.8b_t \tag{18}$$

$$du = (-u_{t-1} + x_t)/\tau_m \tag{19}$$

$$u_t = u_{t-1} + du \tag{20}$$

$$s_t = f_s(u_t, \theta) \tag{21}$$

$$u_t = u_t(1-s_t) + u_r s_t \tag{22}$$

While the dynamics of the hidden layers function as described above, the output layer functions differently. The output layer is leaky integrator without spike. Here, the membrane potentials of all the neurons are integrated and are subjected to log-softmax function to obtain probability distribution of the outputs. The outputs are then compared against the true values and a classical cross entropy loss is generated. Using the loss, instantaneous loss is calculated by adding regularisation penalty to the overall loss, as described in the section 2.3 under the FPTT algorithm. Following this, the weight updates take place following the methods of FPTT.

3.2.2 Dynamism

Enabling the dynamism feature to the model follows the same procedure as the baseline model, however, following an extensive adaptive structural dynamism procedure afterwards.

The procedure for adaptive structural dynamism takes place in three components: synaptic constraints, pruning, and regeneration. While synaptic constraints work throughout training, pruning and regeneration start at a certain epoch and slow down after a specific number of epochs. Synaptic constraint is the primary algorithm where the activity level of synapses is calculated. Pruning and regeneration are the other algorithms aiming to prune the connections or synapses with the least activity and regenerate the synapses with high error, respectively. The aim of implementing such a dynamic algorithm in the model is to measure the extent to which a model can be made efficient and to check if there is any trade-off in performance when measured against efficiency. Achieving dynamism is another goal that is to be accomplished through this implementation.
The overall model including dynamism can be visualised as the flow graph, which is showcased in figure 2 below.

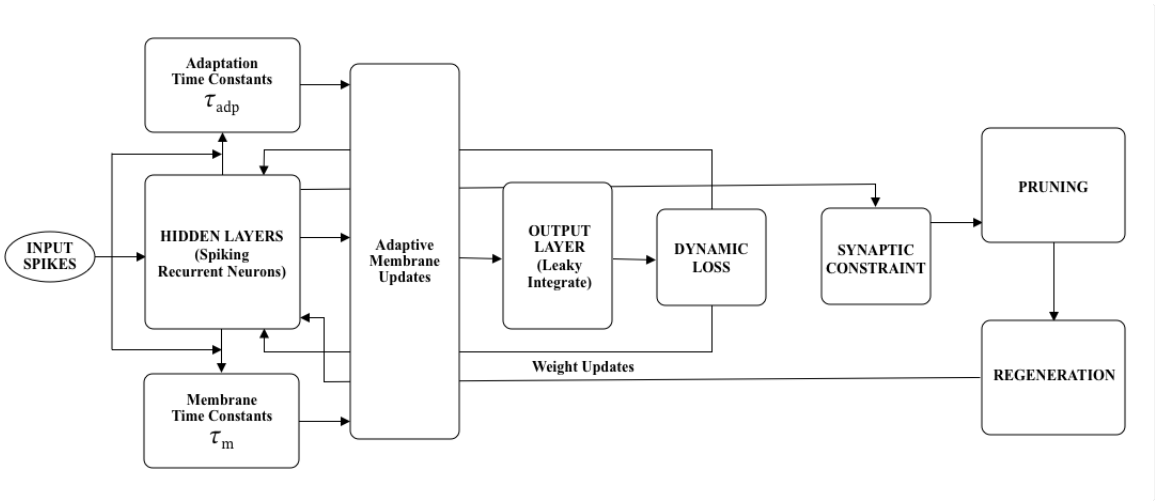


Figure 2: Flow Graph

4 Experimental Setup

Using the model and data described in section 3, the research questions from Section 1.1 will be answered using several experiments. To allow for reproduction of results all the specifications, steps of pre-processing, hyper parameter optimisation and experimental settings will be described in this chapter.

4.1 Specifications

4.1.1 Hardware

The hardware used to run the initial set of experiments was provided by the supervisor Steven Abreu. It was an Nvidia 2080 GPU that is sufficient to run the initial set of experiments where the computation is not highly demanding. The experiments that were conducted later on were computationally demanding, which required us to migrate the code and use more powerful hardware. At this stage, the biotech company Formula Y provided support by giving access to Nvidia T4 GPU, where the rest of the experiments were conducted. Using this hardware, the experiments ran for approximately 3 days per experimental setting, for different variations in the parameter values.

4.1.2 Software

The models and all data processing is implemented using Python 3.X and PyTorch 2.X [79]. Code is hosted on github and available on request.

4.2 Preprocessing

Configuration for the experiments start with the data. The data provided by the authors is of a different format than that of the one required by the SNNs. The provided format represents data using two variables: Times and Units, which is the time at which the neuron fired and it's id, respectively. The format of the data has to be modified in order for the SNNs to process.

Upon inspection of data, we found that the maximum duration of data point in the training dataset was 1370 milliseconds, while the shortest duration was 240 milliseconds. We process each datapoint, and pad them with null values to ensure that each data point has the same duration. Upon padding, we proceed to rate encode the inputs by determining the number of time bins in which it will be processed. Digitizing the firing times in the time bins and assigning them the respective id of the unit, we proceed to batch them. In summary, we convert the data into 3d sparse tensor of size *batch_size * time_bins * ids*, which is fed to the model.

Authors mention the number of inputs to be 700 that either produce a spike or they do not, at each point in time. For our preliminary test, we split the data into 100 time bins with a batch size of 128. Post modification, the data will have a dimension of 128 * 200 * 700.

4.3 Model and Hyper Parameter Optimisation

Regarding the model, since the data has 700 functioning units, this will be the input layer size. Following the input layer, we implement two hidden layers of varying number of hidden units, and an

output layer with 20 units. While the units in the input layer have spiking neurons with just feed forward connections, hidden layers also have recurrent connections, and the output layer is leaky integrate layer where there are no spikes.

All the outputs of the hidden layers undergo batch normalisation at each time step, where the running mean of the weights is also calculated that aids in parameter update later. Following the batch normalisation, the outputs are summed and utilised to calculate the values of *T_m* and *T_adp*. The output is concatenated with the membrane potential of the layer and is passed to a dense network. Similarly, the output is concatenated with the intermediate threshold parameter of the layer and is passed to another dense network. Finally, the output of these dense networks are then subjected to Sigmoid activation, which produces the final values of *T_m* and *T_adp*. Using the outputs, parameters calculated and the previous state of the neuronal parameters, we update the state of neuronal parameters. During the parameter update, whether a spike is produced or not, and the output spikes is passed on to the next layer and the process repeats until the end of final hidden layer.

The spikes from the final hidden layer is passed to the output layer, which is a leaky integrate layer. Here, there are no recurrent connections and no update takes place to the parameters other than the membrane potential. The membrane potential is updated, which then is subjected to Log_softmax activation, which will yield the classification result. Based on the results, loss is calculated in three parts; firstly, classification loss is calculated which is the Negative Log Likelihood loss between the prediction and the true value. Secondly, an auxiliary loss is calculated which is determined based on the predicted label distribution and true label distribution. Finally, A regularisation term is calculated using the parameters calculated in the batch normalisation layer. Summing up these values, the final value of loss is obtained.

While this is the general setup of the maodel used, there are hyper-parameters that require optimisation. Hyper-tuning these parameters will enable us to measure the extent to which the model can perform. While in general, the hyper-parameters such as learning rate, loss function and activation function are fine tuned, we copy these from the original implementation. Focusing on the efficiency of the model, we aim to conduct experiments in a manner that produces an optimal model. To this end, we aim to conduct experiments on other parameters such as number of hidden units and Time bins (parts). More on this is discussed in the next subsection.

4.4 Experiments

The primary aim of the thesis is to establish a benchmark of LSNs on audio event stream classification tasks. The experiments were designed to improve the model systematically by evaluating the influence of every individual parameter. We employ a greedy algorithm strategy, where the performance of one parameter is optimized to its fullest before proceeding to the next. In this manner, we perform a range of experiments to truly test the capabilities and limitations of the model.

Firstly, LSNs are known for their ability to show superior performance in memory tasks, and basing on this fact, we make the network shallow and not deep. In this manner, we keep the number of hidden layers as constant. We chose to use two hidden layers, and designed the primary series of experiments by varying the size of hidden layers. Through the first series of experiments, the variation in the number of hidden units per layer is as follows: 128, 256 and 512. This aids in finding the optimal network size, using which we proceed to perform further experiments.

Second set of experiments were crafted to analyse the influence of the size of time step/ number of time bins on the performance. Here, since time is made discrete to achieve a step along the time dimension, we range of exploration is quite huge and variable. However, we minimise the number of experiments by selectively choosing the values. Therefore, we perform the experiments on the following different number of time bins: 5, 10, 20, 30, 50, 100, 200, which can be translated to step size in time of 280ms, 140ms, 70ms, 47ms, 28ms, 14ms, and 7ms, respectively. Exceeding 500 time bins makes the training inefficient as training lasts longer than a day on an average GPU, and hence, those values have been avoided. Through these series of experiments, we find the optimal processing speed of the network.

Following these experiments, we perform fine-tuning of the model hyper-parameters such as the rate of regularisation, gradient clipping and weight decay, which will reveal their influence on the performance. These experiments enable us to evaluate the model, establish a benchmark, and provides an answer to the primary research question. The next series of experiments are designed to measure the extent of compression that can be exerted on a model that will result in best accuracy-size trade-off.

Since the model compression is achieved through a dynamic tension between pruning rate and regeneration rate, we aim to measure the influence of the same. To elaborate, we fix the initial pruning and regeneration rate, and measure how these parameters evolve over time. Along with these parameters, we also measure how the number of pruned, regenerated and total connections evolve over time. These experiments provide an overall analysis on how the connections are modelled throughout training and this insight can help in designing better networks.

4.5 Experimental Setting

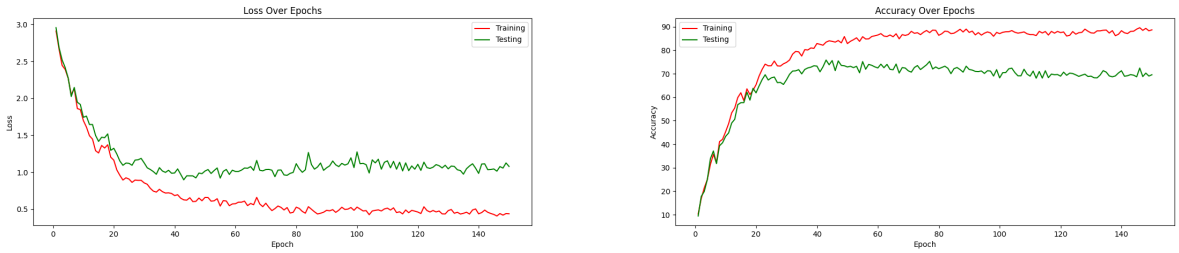
The initial setup of the model is as follows: Training for 150 epochs with an initial learning rate of 0.0005. Based on the original implementation, Adam optimiser with $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\varepsilon = 1e-8$ is used. There is no weight decay, but learning rate decays by 90% of its original value after 20 epochs.

The training of baseline models is done for 150 epochs, while the models with adaptive structural dynamism are trained for 300 epochs. This is due to because of pruning and regeneration, which affects the learning process of the model.

5 Results

5.1 Task 1: Network Size

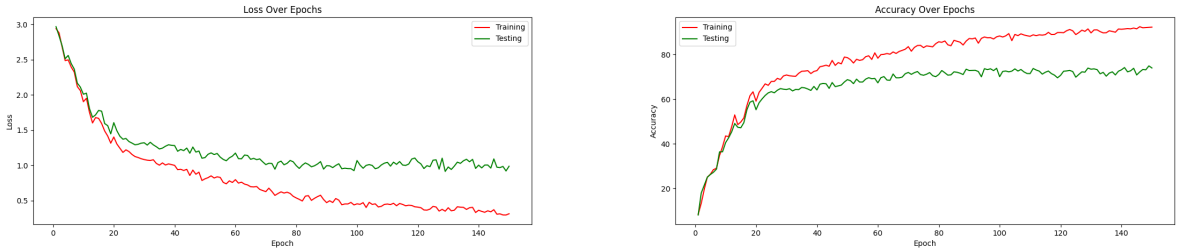
As we strive to assess the effectiveness of the model, our goal is to identify the optimal size that delivers consistently high performance. We choose 512 hidden units per layer in the neural network for our initial experiment. It achieves convergence with an accuracy of 97.89% on the training set and a significantly lower test accuracy of 68.88%. The model's generalization capabilities appear to have suffered; this indicates that the current model size is sub-optimal, emphasizing the need to determine an ideal network size for improved performance.



(a) Loss over epochs (b) Accuracy over epochs

Figure 3: Loss and Accuracy of the model for 512 hidden units per layer

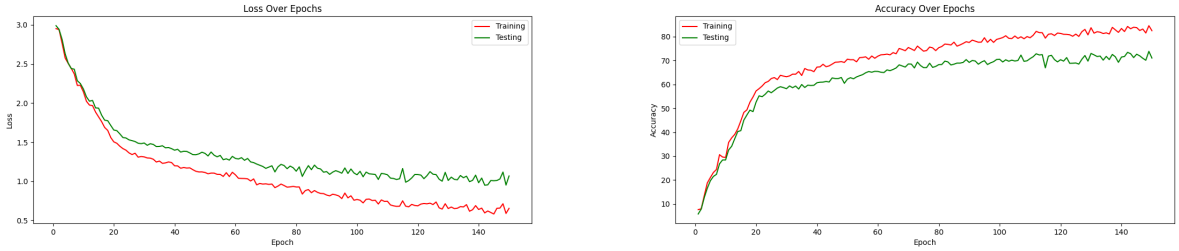
In our subsequent experiment, we opted for a network size of 256 hidden units per layer, and the outcomes are plotted in the figure 4 below. Examination of the plots reveals an improvement in the model's generalization capabilities compared to the initial experiment. The reduced divergence between the learning curves indicates an improvement in the model's ability to generalize. Notably, the accuracy achieved in this experiment is 72.75%, signifying an improvement from the previous model. This network size appears to be more optimal than the previous one. Our subsequent experiment is designed to assess whether we can identify an even more optimal network size.



(a) Loss over epochs (b) Accuracy over epochs

Figure 4: Loss and Accuracy of the model for 256 hidden units per layer

For our third experiment, we selected 128 hidden units per layer, and the results are depicted in the figure 5 below. The findings suggest an overall improvement compared to previous experiments. The notable observation is that the model does not achieve full convergence on the training set, in contrast to prior experiments. Despite this, a substantial reduction in the divergence between the learning curves is evident. This observation implies that while a smaller network may not fully optimise learning capabilities, it demonstrates an improvement in generalisation capabilities.



(a) Loss over epochs

(b) Accuracy over epochs

Figure 5: Loss and Accuracy of the model for 128 hidden units per layer

The experiments conducted are summarised in the plot shown in Figure 6. Analysis of these experiments indicates a trade-off in the model's performance based on the number of hidden units. Specifically, models with a higher number of hidden units demonstrate enhanced learning capabilities but exhibit reduced generalisation performance. Conversely, models with fewer hidden units show better generalization but are limited in their learning capacity. Importantly, a configuration with a moderate number of hidden units emerges as an optimal solution, ensuring proficient learning while maintaining robust generalisation ability.

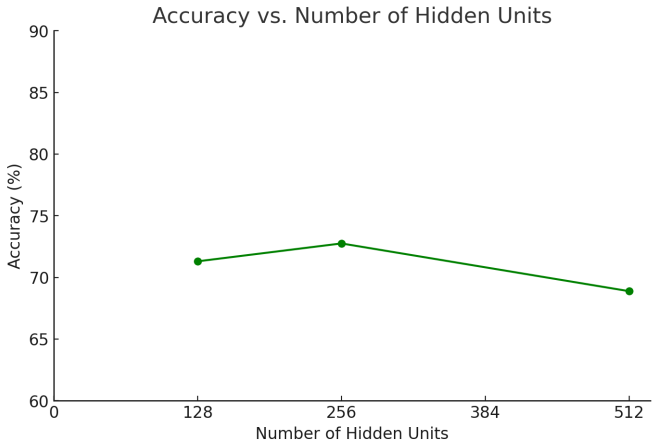
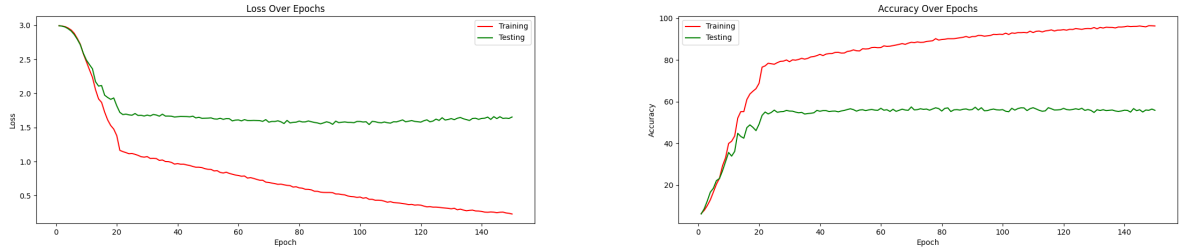


Figure 6: Influence of number of hidden units on performance

Based on the findings from this initial series of experiments, a conclusion can be drawn: the optimal network size appears to be 256 hidden units per layer. This configuration demonstrates a balanced trade-off between generalisation and learning capabilities. By achieving improved generalization while maintaining a reasonable level of learning capacity, the model at this network size represents a better trade-off for the given task. These insights provide guidance for refining the model architecture in pursuit of optimal performance.

5.2 Task 2: Information Procession

SNNs function by processing information in series of several time steps, also known as parts. In the second series of experiments, our objective was to assess the impact of the number of parts on the model's performance. To explore this, we conducted seven experiments, each involving a different quantity of parts. In the initial experiment, the data points were divided into two parts for processing. The results of this experiment are illustrated in Figure 7 below.

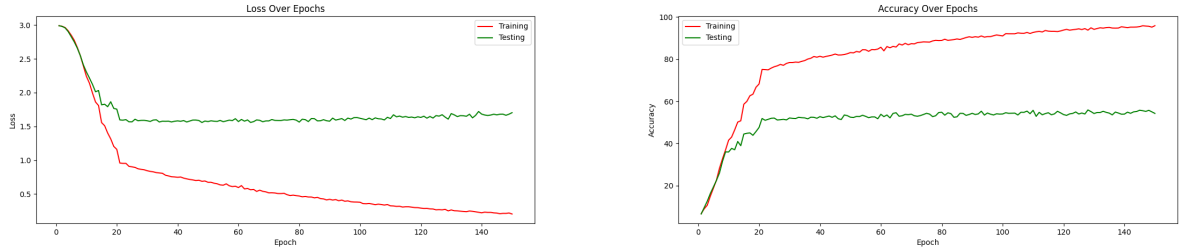


(a) Loss over epochs (b) Accuracy over epochs

Figure 7: Loss and Accuracy of the model when data is processed in 2 parts.

From the figures, it is evident that the learning progresses steadily for the first ten epochs, followed by an oscillation in the learning curve, and a subsequent divergence between the training and testing curves. However, stability is achieved in the learning curve pattern by epoch 20. In terms of the training set, performance steadily advances until reaching a loss of 0.06 and an accuracy of 96%. Conversely, the performance on the test set exhibits a different pattern, plateauing with slight variations at each epoch. Specifically, the model achieves an accuracy of 55.83% with a final loss of 1.65 on the test set. In summary, for this particular experiment, it can be concluded that while the model effectively learns the patterns present in the training set, it struggles to generalize these learned patterns to new and unseen data, leading to sub-optimal performance on the test set.

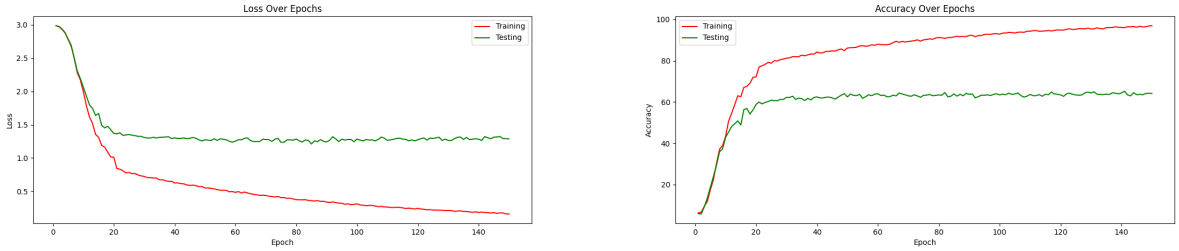
We increase the number of parts to 3 to see if it can affect the performance. The accuracy and loss of the model across epochs for this experiment are illustrated in Figure 8 below.



(a) Loss over epochs (b) Accuracy over epochs

Figure 8: Loss and Accuracy of the model when data is processed in 3 parts.

At first glance, the plots appear to be similar to the plots when the number of parts are 2, but upon close inspection, it can be found that is not the case. The convergence patterns during training are remarkably similar, resulting in only a marginal difference in the final accuracy. The model achieves a loss of 0.058, maintaining the same accuracy at 96%. When evaluating on the testing set, the performance shows neither improvement nor deterioration with an accuracy of 55.85%. However, slight variations in performance are observed at each epoch. To verify if the trend follows, we performed experiments with the number of parts as 5, and the results are visualised below in figure 9.

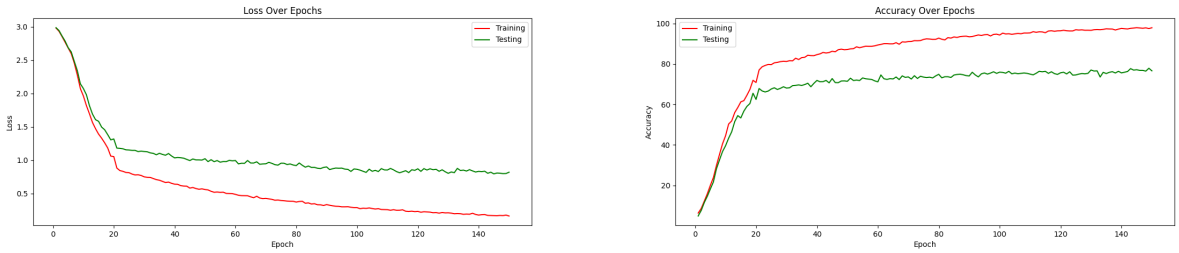


(a) Loss over epochs (b) Accuracy over epochs

Figure 9: Loss and Accuracy of the model when data is processed in 5 parts.

As the data is processed in five parts, there is a slight improvement in the model's generalization capability, as evident in the plots above. On the training set, the model attains a similar level of convergence as observed in the previous experiments. Notably, on the testing set, the instability observed in the initial epochs is now reduced. Furthermore, the divergence between the curves of the testing set and training set is notably narrowed down. In this experiment, the model's performance on the testing set increased to 64.15%, accompanied by a decrease in loss to 1.28.

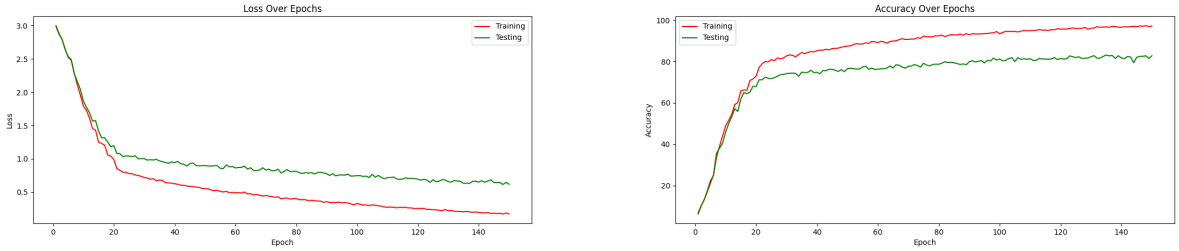
Continuing our exploration, we increased the number of parts to 10 to assess its impact on the model's performance. The observed trend persisted, with the model's generalization capability showing a further improvement by approximately 12%, achieving an accuracy of 76.56% on the test set, and a loss of 0.81. The loss curve also reveals a further reduction in the divergence between the training and testing curves. However, the instability in the accuracy at each epoch is still apparent.



(a) Loss over epochs (b) Accuracy over epochs

Figure 10: Loss and Accuracy of the model when data is processed in 10 parts.

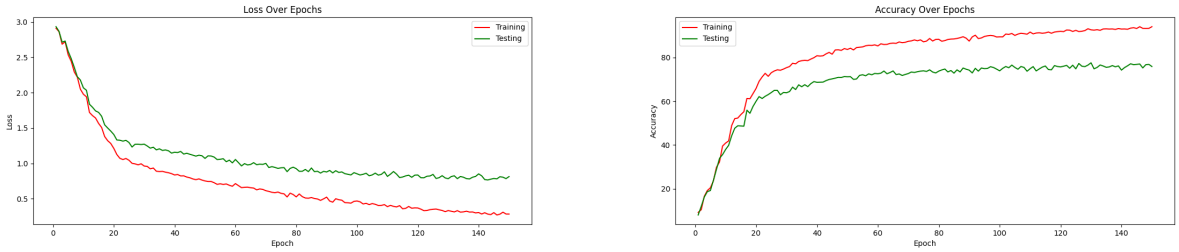
In the subsequent experiment, the model's performance demonstrated further improvement, achieving an accuracy of 84.76% with a loss of 0.59. The corresponding plots over epochs are presented in figure 11 below. Notably, consecutive experiments revealed a consistent variance in performance of approximately $\pm 2.1\%$. While there is no observable difference in the model's convergence or learning, its generalization capabilities have notably improved. The trend that the performance of the model improves with the number of parts which was evident till now does not hold always; the results from the next few experiments provide evidence to this.



(a) Loss over epochs (b) Accuracy over epochs

Figure 11: Loss and Accuracy of the model when data is processed in 20 parts.

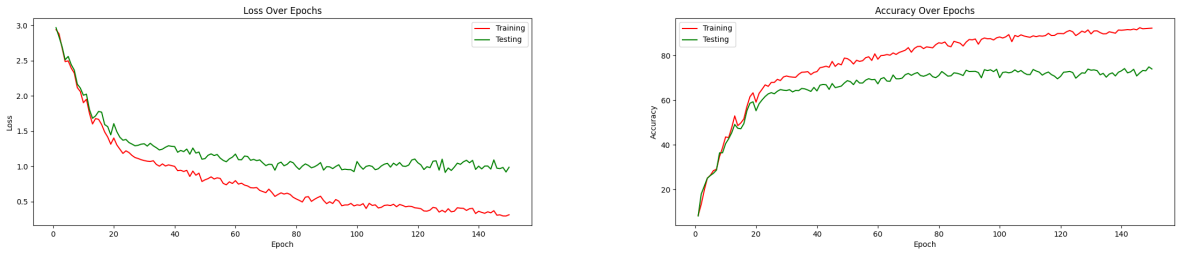
The plots presented in figure 12 undermine the previously observed trend. Despite achieving a certain level of convergence in the learning process, the accuracy in predictions at each epoch exhibits noticeable variance. This variability is also evident in the evaluation on the test set. The overall accuracy on the test set experiences a drop of approximately 9% compared to the preceding experiment, resulting in an accuracy of 75.87% with a loss of 0.81. There is potential for this case to be an outlier; therefore, we proceed to the next experiment to assess if this deviation persists.



(a) Loss over epochs (b) Accuracy over epochs

Figure 12: Loss and Accuracy of the model when data is processed in 50 parts.

With a further increase in the number of parts to 100, the disturbance in the learning process becomes more pronounced, as depicted in Figure 13. The learning curves for both the training and testing sets exhibit an unclear pattern. The performance on the test set drops even further, reaching 73.92% accuracy with a loss of 0.96.

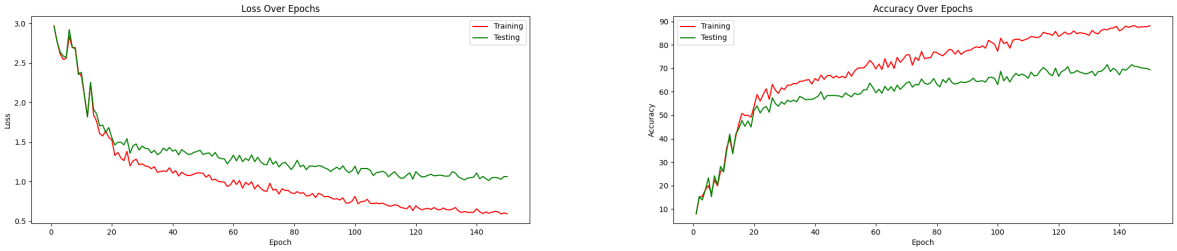


(a) Loss over epochs (b) Accuracy over epochs

Figure 13: Loss and Accuracy of the model when data is processed in 100 parts.

In the concluding experiment of this series, we evaluated the model processing the data in 200 parts. As illustrated in Figure 14, the disturbances in the learning curve have intensified, and

the model's performance has deteriorated, resulting in an accuracy of 69.39% and a loss of 1.06. This observation underscores the complexity of the relationship between the number of parts and the model's ability to learn and generalize effectively.



(a) Loss over epochs

(b) Accuracy over epochs

Figure 14: Loss and Accuracy of the model when data is processed in 200 parts.

The impact of varying parts on model performance is succinctly depicted in Figure 15 below. Examination of the figure reveals a clear pattern: the model initially underperforms with a lower count of parts, but its performance improves with an increase in the number of parts. However, this improvement eventually declines beyond a certain point. Further analysis attributes this decline in performance to learning instability. As the number of parts continues to rise, the model struggles with integrating information from the various parts, leading to diminished performance. Notably, the model achieves its optimal performance with a relatively modest number of parts, specifically at 20 parts in this study, where it effectively integrates information from different parts.

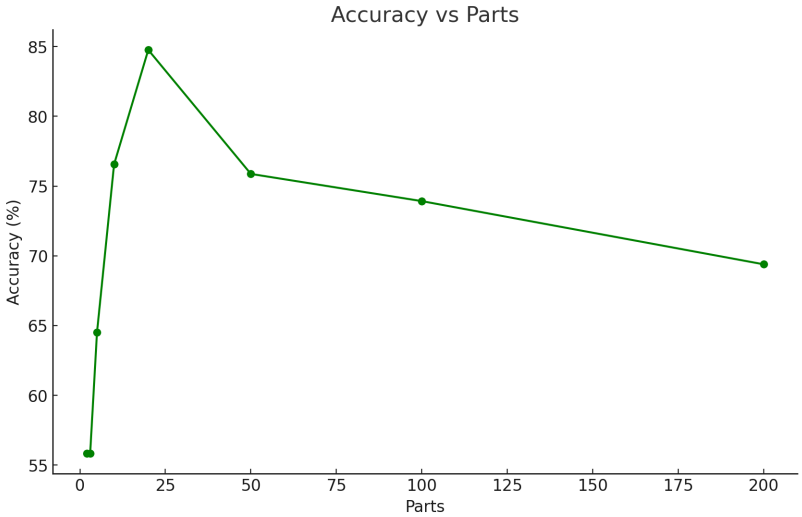


Figure 15: Influence of number of parts on performance

In this series of experiments, we evaluated the influence the parts parameter holds on the performance of the model. To summarise, the influence was directly proportional until a certain point i.e., increasing the parts increased the performance, beyond which, the influence was inversely proportional i.e., increasing the parts reduced the performance. The peak performance was observed when the data was processed in 20 parts, with an accuracy of 84.76%.

In concluding the experiments on varying the number of parts, we further explored performance enhancement through the fine-tuning of various hyper-parameters. The summarized results are presented in the table 1 below.

For better interpretability, the influence of dropout on performance is plotted below in figure 16. The data indicates that implementing dropout in the first hidden layer yields superior results compared to its application in the second hidden layer. This suggests that certain non-essential features learned by the model may hinder generalization, and the application of dropout helps in discarding these features, thus enhancing performance. Experimentation with varying dropout rates across different layers led to further improvements, culminating in a peak test accuracy of 84.7%. This finding implies that both low-level and high-level representations in the model contain unimportant features. Removing these features through dropout enables the model to generalize more effectively and achieve better performance.

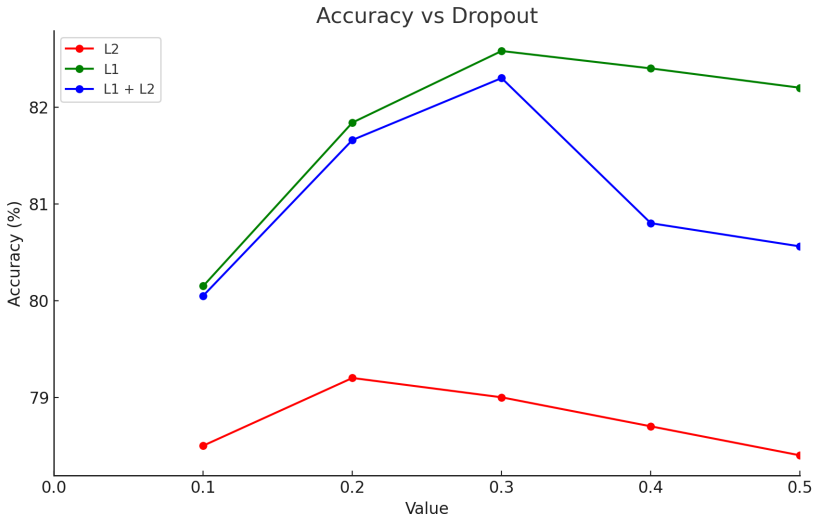


Figure 16: Influence of dropout on performance

The regulariser described in the equation 9, has the parameter α, which was varied to observe the influence it holds on performance. Along with α, there are other parameters ρ and λ, which affect the extent of regularisation. These were also varied and the influence of these parameters on performance is plotted below in figure 17.

5.3 Task 3: Algorithmic Flaws

In the following series of experiments, we explore the implementation of the Structural Dynamism algorithm. For the initial series of experiments, our aim was to study the dynamics of the algorithm over the epochs. However, certain problems were revealed during implementation, leading us to conduct additional experiments to find a deployable solution.

The first problem occurs in the formulation of the dynamic pruning rate, where this rate depends on the number of neurons in the current layer and in the target layer. Two issues arise from this dependency: Static pruning rate and Exploding pruning rate. The Static pruning rate issue occurs due to a lack of a clear definition of what constitutes the number of neurons. As the connections are pruned and regenerated, a scenario emerged in all experiments where no neurons had all connections. If we

Experiment	Layer	Value	Accuracy
Dropout	L2	0.1	78.5%
Dropout	L2	0.2	79.2%
Dropout	L2	0.3	79%
Dropout	L2	0.4	78.7%
Dropout	L2	0.5	78.4%
Dropout	L1	0.1	80.15%
Dropout	L1	0.2	81.84%
Dropout	L1	0.3	82.58%
Dropout	L1	0.4	82.4%
Dropout	L1	0.5	82.2%
Dropout	L1 + L2	0.1 + 0.1	80.05%
Dropout	L1 + L2	0.2 + 0.2	81.66%
Dropout	L1 + L2	0.3 + 0.3	82.3%
Dropout	L1 + L2	0.4 + 0.4	80.8%
Dropout	L1 + L2	0.5 + 0.5	80.56%
Dropout	L1 + L2	0.2 + 0.2	81.66%
Dropout	L1 + L2	0.3 + 0.2	83.77%
Dropout	L1 + L2	0.4 + 0.2	83.13%
Dropout	**L1 + L2**	**0.5 + 0.2**	**84.76%**
Regularisation (α)	-	0.005	83.1%
Regularisation (α)	-	0.01	83.3%
Regularisation (α)	-	0.05	82.3%
Regularisation (α)	-	0.5	81.7%
Regularisation (ρ)	-	0.01	83%
Regularisation (ρ)	-	0.1	83.6%
Regularisation (ρ)	-	0.2	83.5%
Regularisation (ρ)	-	0.5	82.07%
Regularisation (λ)	-	0	82.3%
Regularisation (λ)	-	1	84.76%
Weight Decay	-	0.01	72.75%
Weight Decay	-	0.1	72.91%
Gradient Clipping	-	5	72.75%

Table 1: Performance comparison for different experiments with varied Hyper-parameters

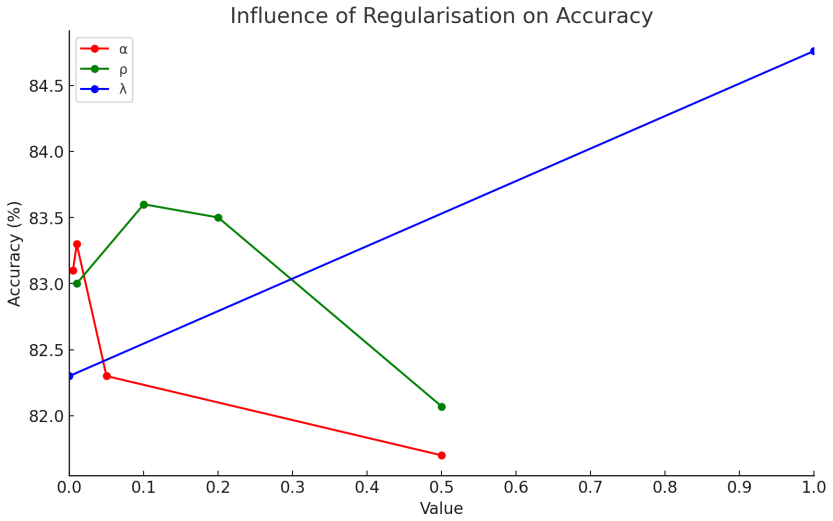


Figure 17: Influence of regularisation parameters on performance

assume a neuron is active only if it has all its connections, then there are zero neurons in the layer, resulting in a fixed pruning rate and no observable dynamics in terms of pruning rate. Alternatively, if we consider a neuron active if it has both an input and an output, then, as per our observations, all neurons are present. However, since all the neurons are always present, there is no change in their number, leading to a static pruning rate. Conversely, the Exploding pruning rate was also observed in our experiments, where the pruning rate keeps increasing. This dynamic was observed in the final hidden layer. Since the number of neurons in the final hidden layer is generally greater than in the output layer, the fraction involved in the dynamic pruning rate keeps increasing, resulting in all neurons getting pruned. Consequently, an entire layer of hidden units is deleted, making it impossible to train the network.

The proposed method aims to address these problems by eliminating the dependency on neurons and instead using the number of connections as a replacement. To elaborate, using neurons in a particular layer necessitates defining the neuron, as mentioned in the previous paragraph. However, by using the number of connections, we can avoid defining the neuron and still observe similar dynamics to what would have been with neurons. The new formula for updating the pruning rate is as follows:

$$\rho_p = \rho_p + \delta \frac{S^l}{S^{l+1}} \tag{23}$$

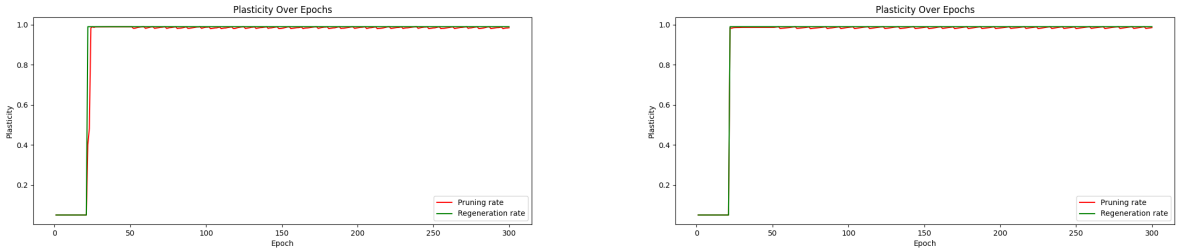
$$S^l = \frac{C^l}{N^{l+1}} \tag{24}$$

where C^l and N^{l+1} represent the number of connections between the layers and the size of the target layer, respectively. This new formulation helps avoid a static pruning rate by creating a dependency on the number of connections.

However, the exploding pruning rate is a problem that still persists. In regards to exploding regeneration rate, the original implementation had a solution where if the regeneration rate exploded, it was set 99% as to restore the incorrectly pruned synapses. However, the dynamics of the pruning rate is not the same, as pruning is performed on neurons with least activity level; and the selection of

certain number of neurons with low activity has to be dynamic. Or the definition of dynamic pruning rate does not hold. Therefore, to overcome this issue of explosive pruning rate, it is necessary to regularize it. To this end, we conduct experiments with a series of regularization values on the dynamic pruning rate to explore its dynamics.

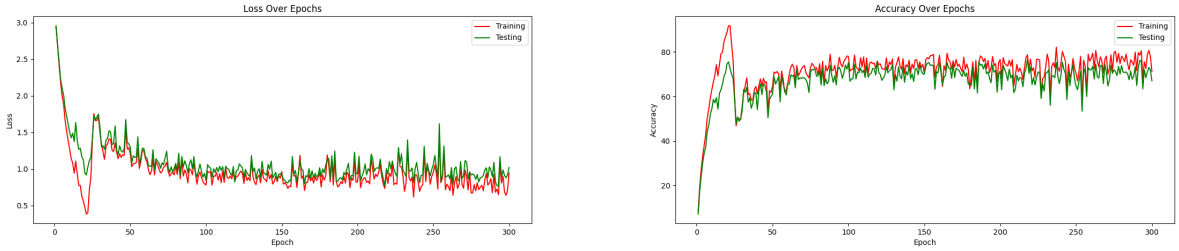
For the first experiment, we fix the pruning rate to a maximum of 99% if it exceeds the same.



(a) Plasticity of the first hidden layer (b) Plasticity of the second hidden layer

Figure 18: Plasticity over epochs for the hidden layers where the pruning rate is reset to 99%.

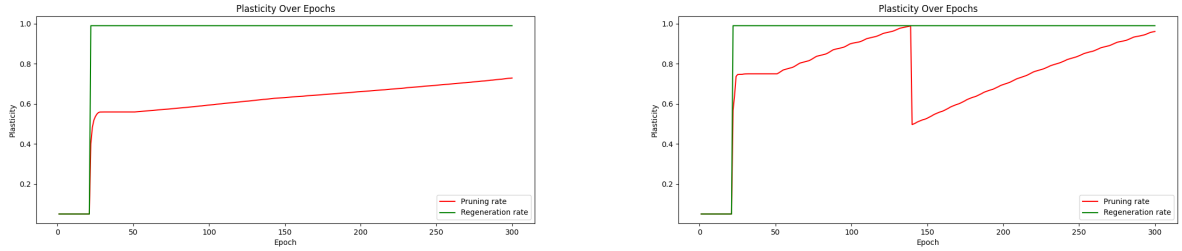
From the figure 18 above, it can be observed that there are improved dynamics in terms of pruning rate; it is not stationary anymore. However, the pruning rate increases to a maximum in the beginning epochs and does not reduce to any significant value. The regularisation value acts as a lower bound in the new dynamics. In terms of performance, the constant regeneration and pruning affects the training process intensely, as the network trained was for twice the duration as the baseline model and yet, convergence was not achieved.The performance of the model throughout training can be observed below in figure 19.



(a) Loss over epochs (b) Accuracy over epochs

Figure 19: Loss and Accuracy of the model over the epochs.

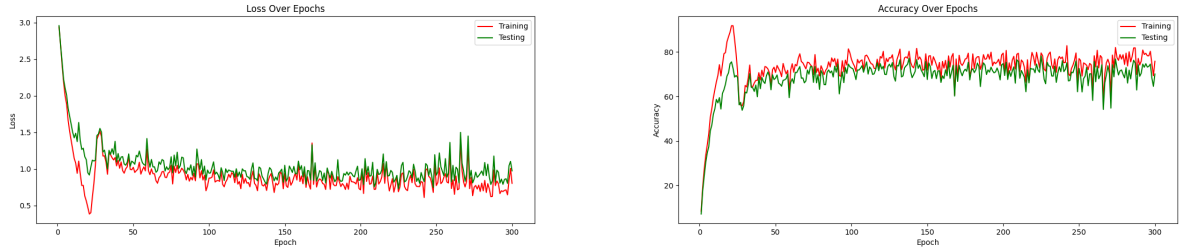
It can be observed that pruning and regeneration causes high instability during the training process. The fluctuation is due to pruning of large number of synapses, which are regenerated later. However, through regularisation of pruning rate, we can reduce the variance in performance at every epoch. Therefore, we conduct another experiment where the pruning rate term is reset to 50% of the value when it exceeds 99% pruning rate. A observation that is crucial is that the divergence of the curve at each epoch is smaller than the values observed in the experiments related to the baseline model; It signifies that the algorithm improves the network's ability to generalise.



(a) Plasticity of the first hidden layer (b) Plasticity of the second hidden layer

Figure 20: Plasticity over epochs for the hidden layers where the pruning rate is reset to 50%.

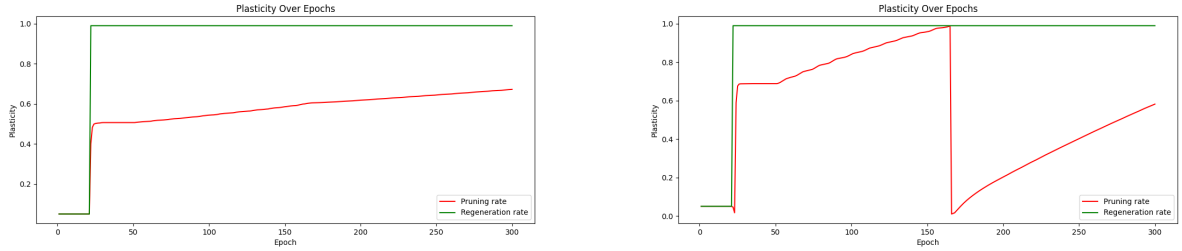
As compared to the first experiment, the pruning rate increases gradually over epochs for the first hidden layer. For the second hidden layer however, the dynamics are similar to as observed in the previous experiment; the regularisation value acts as the lower bound and the pruning rate slowly increases only to be reset again. In regards to performance, there is not observable difference. However, the variance in performance at each epoch has reduced slightly, but the overall performance is quite similar. The similarity of performance can be compared to the figure 21 below.



(a) Loss over epochs (b) Accuracy over epochs

Figure 21: Loss and Accuracy of the model over the epochs.

In the next experiment, we reduce the lower bound even further by reducing the reset value to 1% of the actual value. The evolution of pruning rate over the epochs for each layer is plotted below in figure 22.

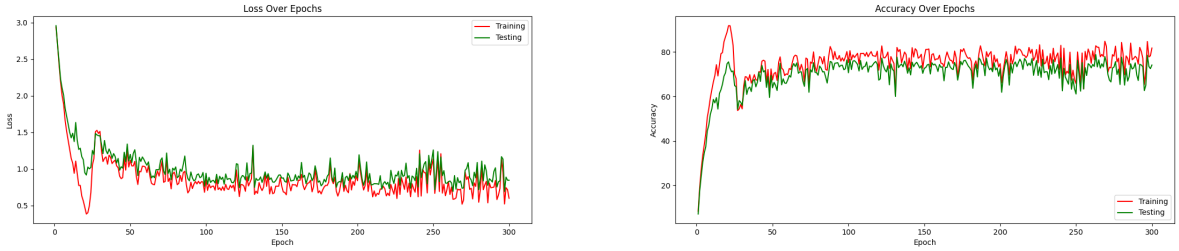


(a) Plasticity of the first hidden layer (b) Plasticity of the second hidden layer

Figure 22: Plasticity over epochs for the hidden layers where the pruning rate is reset to 1%.

While the curves evolve almost similarly to the previous experiment, the instability in performance is reduced, and the overall performance has improved. The improvement in performance is however minimal with an increase of 2% leading to total accuracy of 78.2% on test set, and 83.4%

on the training set. The performance plots of this experiment are visualised in the figure 23. In this series of experiments, we observed how the regularisation term acts as lower bound for the pruning rate. Another crucial observation is that the instability in the performance reduced as we decreased the regularisation value. The implication of such behaviour is that as the regularisation decreases the overall pruning rate, less number of incorrect connections are pruned; as a result, the instability in the performance reduces.



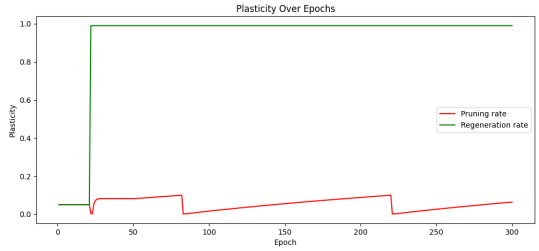
(a) Loss over epochs

(b) Accuracy over epochs

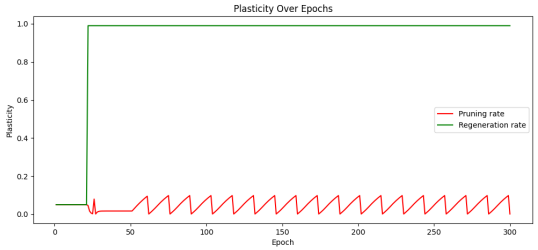
Figure 23: Loss and Accuracy of the model over the epochs.

We conducted the same experiments by reformulating the regeneration rate using a regulariser, similar to the experiments performed to the pruning rate above. The observed dynamics for regeneration rate were similar to the dynamics observed for the pruning rate, however the performance had increased divergence between the training and testing set. There is a deeper problem in the implementation that occurs at formulation: High pruning rate leads to pruning incorrect synapses leading to instability in performance. While regularisation value regulates the pruning rate, as the pruning rate is dynamic, it increases eventually to the maximum value. The solution to overcome this problem is to create an upper bound for the pruning rate, in order to avoid explosive pruning rate.

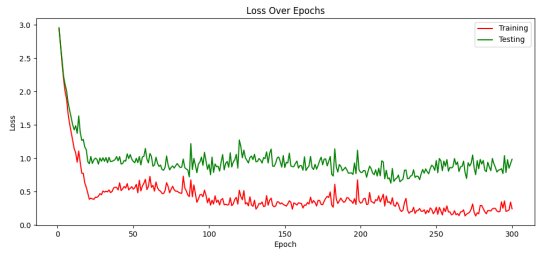
Therefore, we conduct experiments to observe the dynamics implied by applying the upper bound to pruning rate. To this end, we perform our experiment with an upper bound to pruning rate as 10%. The results obtained were as follows:



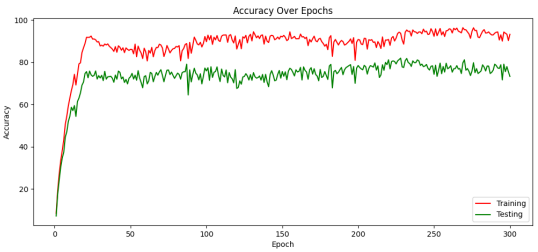
(a) Plasticity of the first hidden layer



(b) Plasticity of the second hidden layer



(c) Loss over epochs

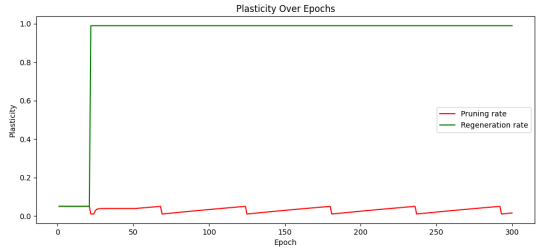


(d) Accuracy over epochs

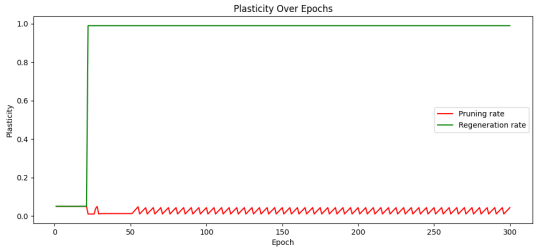
Figure 24: Plasticity with upper bound and the model performance.

It can be observed that the performance improved over the training and testing set. However, the divergence between the learning curves increased as compared to the previous experiments. Despite the instability in performance, the model is able to converge on the training set. Its performance on testing set improved, however the model's overall generalising capability reduced. In regards to the pruning rate, the rise in the pruning rate in the first hidden layer is slower than the pruning rate in the second hidden layer. We conduct our next experiment by reducing the upper bound on pruning rate to 5% to see if the trend continues.

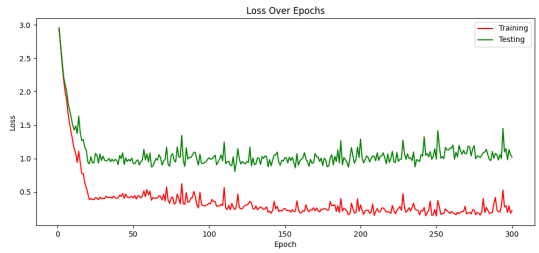
There is no observable difference in the performance, however the instability has reduced as compared to the previous experiment. This signifies that the reducing the upper bound reduces the instability in performance. Despite reducing the upper bound, there is no difference in the divergence between the learning curves.



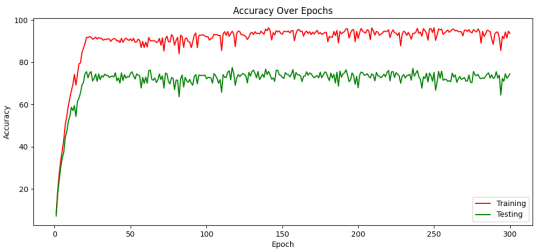
(a) Plasticity of the first hidden layer



(b) Plasticity of the second hidden layer



(c) Loss over epochs



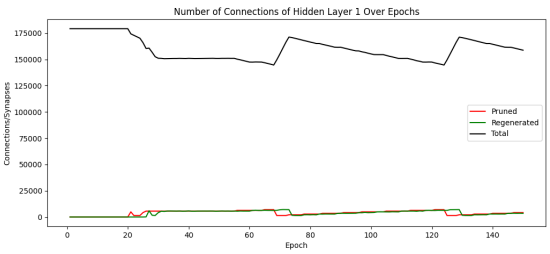
(d) Accuracy over epochs

Figure 25: Plasticity with upper bound and the model performance.

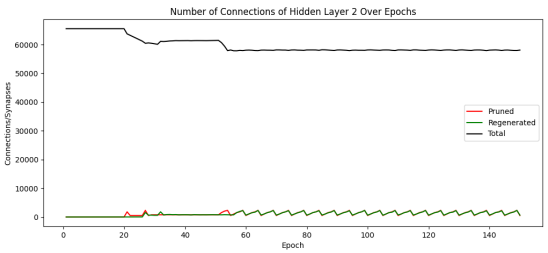
5.4 Task 4: Pruning Rate

Having delved into the intricate dynamics of pruning rate and regeneration rate across epochs and their consequential impact on performance, we consider this approach as a remedy for the previously identified issues. With this solution in place, we proceed to deploy the algorithm to assess its effects on performance under varying algorithmic parameters.

As the dynamics of pruning rate evolve over training, the initial pruning rate plays a key role in influencing the number of connections at each epoch, and the overall performance of the model. Therefore, we conduct a few experiments to evaluate the influence of the initial pruning rate on performance. For the first experiment, we choose the initial pruning rate as 3% and how it evolves over the training period can be seen in the figure 26 below.



(a) First hidden layer

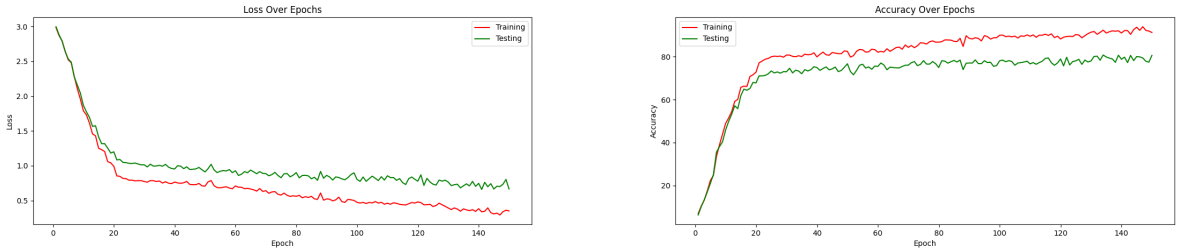


(b) Second hidden layer

Figure 26: Number of connections evolving through training

In the initial stage of the first hidden layer, there is a noticeable decrease in the total number of connections. However, as the epochs progress, this trend reverses, with the number of connections gradually increasing. This phase is characterized by a lower rate of connection pruning compared to

the rate of regeneration. Following this, the total connections oscillate, alternating between reduction and increase, indicating a continuous cycle of connection removal and reformation without reaching a stable state. In contrast, the second hidden layer exhibits a different pattern: the total number of connections declines and eventually stabilises. While there is a difference in the amount of neurons pruned and regenerated initially, the difference eventually reduces to a negligible value. The algorithm achieves compression rates of approximately 10% in the first hidden layer and around 12% in the second. The overall compression of the model post training is at 10.5%. In terms of performance, the model demonstrates comparable results with a training accuracy of 93% and a testing accuracy of 80.65%. Figure 27 reports the performance through the training period. Notably, the performance instability noted in prior experiments has been markedly reduced in this instance.



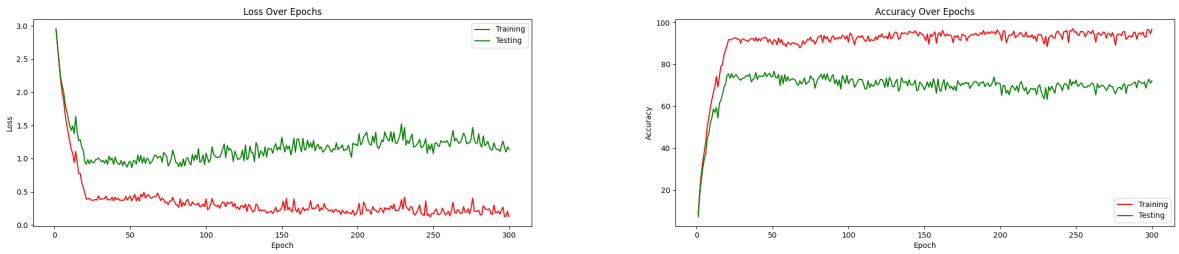
(a) Loss over epochs (b) Accuracy over epochs

Figure 27: Loss and Accuracy of the model over the epochs.

In our next experiment, we raised the initial pruning rate to 5%. Despite this adjustment, the dynamics of the connections remained largely unchanged compared to our previous trial. However, we did observe a enhancement in performance, with the final accuracy improving by 1.05%, culminating in a total accuracy of 81.70%.

5.5 Task 5: Regeneration Threshold

In this investigation, our objective is to measure the impact of the parameter T_{num} on both performance metrics and the number of connections within the model. Notably, all preceding experiments were executed with $T_{num} = 5$. To initiate this exploration, our first experiment involves selecting a smaller value of $T_{num} = 3$. The ensuing results are graphically represented in Figure 28.

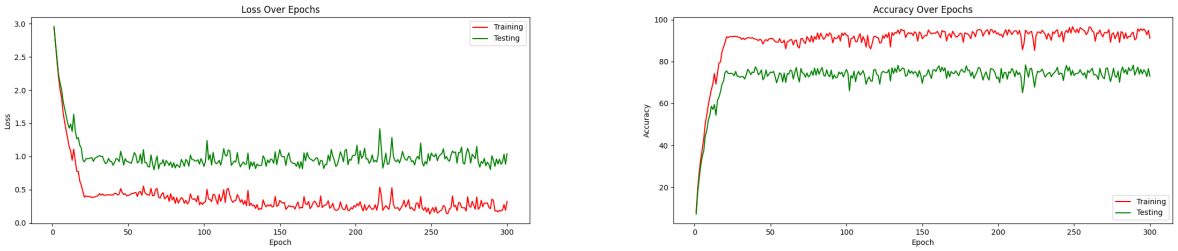


(a) Loss over epochs (b) Accuracy over epochs

Figure 28: Model performance when T_{num} is 3.

It can be observed that the model's performance on training reaches convergence. On testing set however, the performance declines as compared to the previous experiments. This decline suggests

a reduction in the model's generalization capability as the value of T_{num} is decreased. To further investigate this relationship, a subsequent experiment is conducted by increasing the value of T_{num} to 7, to verify if the value of T_num is proportional to the model performance. The results are published below in figure 29.



(a) Loss over epochs

(b) Accuracy over epochs

Figure 29: Model performance when T_{num} is 7.

The results appear to be similar to the previous experiment where $T_{num} = 3$, however the divergence between the learning curve reduced. It implies that the generalisation capability of the model improved. This result supports the conclusion from the previous experiment that model's generalisation capability is proportional to the value of T_{num}. These experimental findings are consistent with the conclusions made in the original implementation.

5.6 Task 6: Comparison

After estabishing the model's performance with and without the dynamism aspect, we proceed to underscore its strengths. The comparative analysis is presented in the table 2 below.

Model	Parts	Parameters	Accuracy
SNN with delays [80]	10	∼250k	95.1%
RSNN with Adaptation [81]	-	∼4M	94.6%
SNN with Attention [82]	15	∼100k	92.4%
RSNN with Temporal Attention [83]	16	∼100k	91.1%
RSNN [84]	1400	∼300k	84.8%
LSN (Ours)	20	∼250k	84.7%
LSN with Dynamism (Ours)	20	∼220k	81.7%

Table 2: Performance comparison against state-of-the-art models.

Firstly, the quantity of parameters is crucial in crafting an efficient network. Our model excels in this regard, possessing 16 times fewer parameters than the current state-of-the-art models. This efficiency, however, comes at a cost: an approximate 10% decrease in accuracy.

Furthermore, the number of processing parts is crucial for operational efficiency, with a direct impact on power consumption. Our model significantly outperforms in this metric as well, requiring only 20 parts for processing compared to the 1400 parts used by the traditional RSNN model with

over 300k parameters. This substantial reduction underscores the efficiency of our approach.

Additionally, it's noteworthy that some models employ techniques such as Data Augmentation [83], Delays (in information transmission) [80], and Attention Mechanisms [82, 83], which have been shown to enhance performance. Our LSN model, while not utilising these methods, has still achieved respectable performance in comparison. For future research, we aim to conduct additional experiments by incorporating these techniques into our model, as the LSN could potentially benefit from this.

6 Discussion and Conclusion

In this research, our primary aim was to evaluate the LSN model for the task of audio event stream classification, specifically using the Spiking Heidelberg Digits (SHD) dataset. To achieve this, we conducted a series of experiments focusing on the model's parameters to determine their influence on performance. The initial experiments concentrated on the number of hidden units in each layer. Our findings suggest that 256 hidden units per layer are optimal. A lower count compromises the model's learning capability, although it improves its generalising capacity. While a higher number of hidden units enhances the learning capability, it adversely affects generalising capability. However, at the midpoint of 256 hidden units per layer, the model offers a balanced trade-off between learning capability and generalising capability.

Having determined an optimal network size, we then focused on the network's processing speed, specifically the number of time bins it can process information in, and its impact on performance. Our results indicate that with a smaller number of bins, the model performs poorly. However, performance improves as the number of bins increases. This trend does not hold indefinitely; as the number of bins continues to rise, we have observed increased instability in performance, leading to a relatively lower overall end performance. These findings imply that performance is not directly proportional to the number of time bins. Instead, optimal performance is achieved at an intermediate value, where the LSN can best utilise its memory capabilities. In benchmarking, our model ranks sixth in terms of performance. However, when considering efficiency, our model stands out as the most efficient at this level of performance. These results suggest that LSNs are useful in tasks that involve memory, where the model can deliver excellent performance while maintaining computational efficiency.

Starting with the baseline model, we implemented the dynamism algorithm, which initially revealed inherent issues during its deployment. These problems primarily manifested in the form of an 'exploding pruning rate' — a scenario where the pruning rate escalated to its maximum, leading to the complete removal of connections and, consequently, the deletion of an entire layer. To address this, we introduced a dual strategy: setting an upper limit and establishing a reset mechanism. These modifications effectively mitigated the issue. Nonetheless, the model still experiences performance instability due to the continuous cycle of pruning and regenerating connections. This necessitates extended training periods. Although the instability remains to some extent, the model's overall performance shows noticeable improvement following prolonged training.

Employing the implemented fix, we proceeded to evaluate the impact of varying the initial pruning rate, ranging from a minimal value to the established upper bound. This sequence of experiments shed light on the dynamism algorithm's dual role: it not only facilitates adaptability during training but also effectively compresses the model. However, this compression is accompanied by a compromise in accuracy. The most significant compression achieved was a reduction to 10.5% of the base model's size; however, it resulted in a 3% decrease in accuracy.

Afterwards, we carried out experiments to assess the impact of the regeneration threshold on the model's performance. Our findings indicate a proportional relationship between the regeneration threshold and performance, aligning with the results from the original implementation. Additionally, we observed that the model's generalisation capabilities improve as the regeneration threshold increases. However, this improvement requires extended training periods.

In summary, the LSN model effectively captures the complex geometry of the SHD dataset and generalises well, offering an excellent balance between performance and efficiency. The Dynamism algorithm serves as a complementary component to the LSN model, enabling model compression through ongoing cycles of pruning and regeneration. However, it's important to note that this compression process does result in a reduction of accuracy. Thus, the Dynamism algorithm functions as an effective compression mechanism, enhancing the LSN model's ability to achieve an optimal blend of performance and efficiency.

7 Future Work

We open up multiple new avenues for future work, each focused on different elements of the network. The first is to achieve dynamism through temporal dynamics, inspired by the functioning of the brain. We first discuss the biological origins and then proceed to explain how we can implement this concept.

Refractory Period (RP) refers to the phase where the neurons rebel in their function. In other words, neurons will either not function or will resort to limited functioning. There are two RPs that can be observed in a functioning human brain: Absolute Refractory Period (ARP) and Relative Refractory Period (RRP). ARP is the primary phase of a neuron that is observed post-emission of a spike (Firing Period) (FP), during which it's unresponsive to any new stimulus, no matter how intense it may be. Following the ARP, the neuron enters the RRP, where it would require a stimulus of higher intensity to generate a response.

In the current version of SRNNs, the function of RRP is implemented as Adaptive threshold. While the implementation influences the dynamics of the network in terms of neuronal communication, the range of dynamics could be further enhanced with the introduction of ARP. The membrane potential of a neuron will either increase gradually as it processes the input spikes or decrease instantaneously when it is reset after crossing the threshold. After the reset mechanism, we introduce ARP, where the membrane potential does not increase despite the input spike, and as a result, the membrane potential is constant throughout the ARP. An example of this concept is presented below in figure 30.

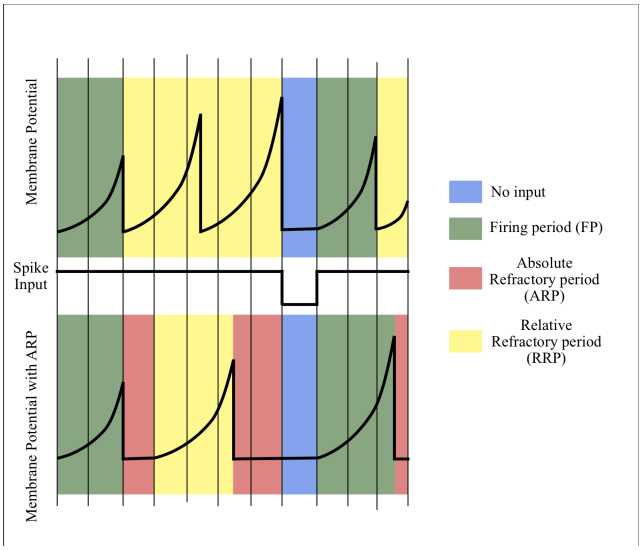


Figure 30: Sample Spike Train from SHD dataset

The dynamics of a neuron without ARP are as follows: The first time a spike is received, it enters FP, where membrane potential starts building up. After each time unit, the membrane potential increases, and the buildup continues until it reaches the threshold value. At the threshold value, it is reset to the baseline value, and the neuron remains in an idle state. After the reset, the threshold value is adjusted depending on the intensity of the input spikes, changing the state of the neuron to RRP. This input-dependent state transition is a key feature of the spiking neural networks that aid in modelling temporal data with great confidence.

To improve the network's ability to model the data, we introduce a new state to the neuron, and its dynamics can be observed in the picture above. Until the reset, the behaviour is the same. After the reset, however, the neuron enters ARP, where it adjusts the threshold value based on the input, but there is no buildup of membrane potential. After certain time steps, the neuron will exit ARP and enter RRP, and the process continues.

Bibliography

[1] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The bulletin of mathematical biophysics*, pp. 115–133, 1943.

[2] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in the brain.," *Psychological Review*, vol. 65, pp. 386–408, 1958.

[3] M. Minsky and S. Papert, *Perceptrons: An Introduction to Computational Geometry*. MIT Press, 1969.

[4] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[5] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning internal representations by error propagation," in *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Volume 1: Foundations*, pp. 318–362, 1986.

[6] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," in *Advances in Neural Information Processing Systems*, vol. 27, 2014.

[7] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," 2023.

[8] W. Maass, "Networks of spiking neurons: The third generation of neural network models," *Neural Networks*, vol. 10, no. 9, pp. 1659–1671, 1997.

[9] M. R. Azghadi, C. Lammie, J. K. Eshraghian, M. Payvand, E. Donati, B. Linares-Barranco, and G. Indiveri, "Hardware implementation of deep network accelerators towards healthcare and biomedical applications," *IEEE Trans. Biomed. Circuits Syst.*, vol. 14, no. 6, pp. 1138–1159, 2020.

[10] E. Ceolini, C. Frenkel, S. B. Shrestha, G. Taverni, L. Khacef, M. Payvand, and E. Donati, "Hand-gesture recognition based on emg and event-based camera sensor fusion: A benchmark in neuromorphic computing," *Frontiers in Neuroscience*, vol. 14, 2020.

[11] R. Primorac, R. Togneri, M. Bennamoun, and F. Sohel, "Generalized joint sparse representation for multimodal biometric fusion of heterogeneous features," pp. 1–6, 12 2018.

[12] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, 1986.

[13] A. Kag and V. Saligrama, "Training recurrent neural networks via forward propagation through time," in *Proceedings of the 38th International Conference on Machine Learning* (M. Meila and T. Zhang, eds.), vol. 139 of *Proceedings of Machine Learning Research*, pp. 5189–5200, PMLR, 2021.

[14] R. M. Hasani, M. Lechner, A. Amini, D. Rus, and R. Grosu, "Liquid time-constant networks," in *AAAI Conference on Artificial Intelligence*, 2020.

[15] M. Raghu, B. Poole, J. Kleinberg, S. Ganguli, and J. Sohl-Dickstein, "On the expressive power of deep neural networks," in *Proceedings of the 34th International Conference on Machine Learning*, vol. 70 of *Proceedings of Machine Learning Research*, pp. 2847–2854, PMLR, 06–11 Aug 2017.

[16] B. Yin, B. Yin, F. Corradi, and S. M. Bohté, "Accurate online training of dynamical spiking neural networks through forward propagation through time," *Nature Machine Intelligence*, 2023.

[17] H. Li, H. Liu, X. Ji, G. Li, and L. Shi, "Cifar10-dvs: an event-stream dataset for object classification," *Frontiers in neuroscience*, vol. 11, p. 309, 2017.

[18] A. L. Hodgkin and A. F. Huxley, "A quantitative description of membrane current and its application to conduction and excitation in nerve," *The Journal of Physiology*, vol. 117, no. 4, pp. 500–544, 1952.

[19] B. Cramer, Y. Stradmann, J. Schemmel, and F. Zenke, "The heidelberg spiking data sets for the systematic evaluation of spiking neural networks," *IEEE Transactions on Neural Networks and Learning Systems*, 2020.

[20] J. K. Eshraghian, M. Ward, E. O. Neftci, X. Wang, G. Lenz, G. Dwivedi, M. Bennamoun, D. S. Jeong, and W. D. Lu, "Training spiking neural networks using lessons from deep learning," *Proceedings of the IEEE*, vol. 111, no. 9, pp. 1016–1054, 2023.

[21] Y. Bengio, "How auto-encoders could provide credit assignment in deep networks via target propagation," *CoRR*, vol. abs/1407.7906, 2014.

[22] W. He, Y. Wu, L. Deng, G. Li, H. Wang, Y. Tian, W. Ding, W. Wang, and Y. Xie, "Comparing snns and rnns on neuromorphic vision datasets: Similarities and differences," *Neural Networks*, vol. 132, pp. 108–120, 2020.

[23] L. Lapicque, "Recherches quantitatives sur l'excitation électrique des nerfs traitée comme une polarisation," *J. Physiol. Pathol. Gen.*, vol. 9, pp. 620–635, 1907.

[24] W. Gerstner and W. M. Kistler, *Spiking Neuron Models: Single Neurons, Populations, Plasticity*. Cambridge University Press, 2002.

[25] A. V. Hill, "Excitation and accommodation in nerve," *Proceedings of the Royal Society of London. Series B - Biological Sciences*, vol. 119, no. 814, pp. 305–355, 1936.

[26] A. Treves, "Mean-field analysis of neuronal spike dynamics," *Network: Computation in Neural Systems*, vol. 4, p. 259, aug 1993.

[27] E. M. Izhikevich, "Resonate-and-fire neurons," *Neural Networks*, vol. 14, no. 6, pp. 883–894, 2001.

[28] N. Brunel, V. Hakim, and M. J. E. Richardson, "Firing-rate resonance in a generalized integrate-and-fire neuron with subthreshold resonance," *Phys. Rev. E*, vol. 67, p. 051916, May 2003.

[29] R. Brette and W. Gerstner, "Adaptive exponential integrate-and-fire model as an effective description of neuronal activity," *Journal of Neurophysiology*, vol. 94, no. 5, pp. 3637–3642, 2005.

[30] L. Badel, S. Lefort, T. K. Berger, C. C. H. Petersen, W. Gerstner, and M. J. E. Richardson, "Extracting non-linear integrate-and-fire models from experimental data using dynamic i–v curves," *Biological Cybernetics*, 2008.

[31] Y. Dan and M. M. Poo, "Spike timing-dependent plasticity: from synapse to perception," *Physiological reviews*, vol. 86, no. 3, pp. 1033–1048, 2006.

[32] G. Q. Bi and M. M. Poo, "Synaptic modification by correlated activity: Hebb's postulate revisited," *Annual Review of Neuroscience*, vol. 24, no. 1, pp. 139–166, 2001.

[33] Y. Bengio, T. Mesnard, A. Fischer, S. Zhang, and Y. Wu, "An objective function for STDP," *CoRR*, vol. abs/1509.05936, 2015.

[34] Y. Dong, D. Zhao, Y. Li, and Y. Zeng, "An unsupervised stdp-based spiking neural network inspired by biologically plausible learning rules and connections," *Neural Networks*, vol. 165, pp. 799–808, 2023.

[35] J. Wu, Y. Chua, M. Zhang, G. Li, H. Li, and K. C. Tan, "A tandem learning rule for effective training and rapid inference of deep spiking neural networks," *IEEE Transactions on Neural Networks and Learning Systems*, 2021.

[36] J. Wu, E. Yılmaz, M. Zhang, H. Li, and K. C. Tan, "Deep spiking neural networks for large vocabulary automatic speech recognition," *Frontiers in neuroscience*, vol. 14, p. 199, 2020.

[37] D. Huh and T. J. Sejnowski, "Gradient descent for spiking neural networks," in *Advances in Neural Information Processing Systems*, vol. 31, 2018.

[38] D. H. Ackley, G. E. Hinton, and T. J. Sejnowski, "A learning algorithm for boltzmann machines," *Cogn. Sci.*, vol. 9, pp. 147–169, 1985.

[39] Y. Bengio, N. Léonard, and A. C. Courville, "Estimating or propagating gradients through stochastic neurons for conditional computation," *CoRR*, vol. abs/1308.3432, 2013.

[40] D. Jimenez Rezende and W. Gerstner, "Stochastic variational learning in recurrent spiking networks," *Frontiers in computational neuroscience*, vol. 8, p. 38, 2014.

[41] H. Mostafa and G. Cauwenberghs, "A learning framework for winner-take-all networks with stochastic synapses," *Neural Computation*, vol. 30, no. 6, pp. 1542–1572, 2018.

[42] E. O. Neftci, H. Mostafa, and F. Zenke, "Surrogate gradient learning in spiking neural networks: Bringing the power of gradient-based optimization to spiking neural networks," *IEEE Signal Processing Magazine*, vol. 36, no. 6, pp. 51–63, 2019.

[43] E. O. Neftci, C. Augustine, S. Paul, and G. Detorakis, "Event-driven random back-propagation: Enabling neuromorphic deep learning machines," *Frontiers in neuroscience*, vol. 11, p. 324, 2017.

[44] S. M. Bohte, J. N. Kok, and H. La Poutré, "Error-backpropagation in temporally encoded networks of spiking neurons," *Neurocomputing*, vol. 48, no. 1, pp. 17–37, 2002.

[45] W. Guo, M. E. Fouda, A. M. Eltawil, and K. N. Salama, "Efficient training of spiking neural networks with temporally-truncated local backpropagation through time," *Frontiers in Neuroscience*, vol. 17, p. 1047008, 2023.

[46] L. Deng, Y. Wu, X. Hu, L. Liang, Y. Ding, G. Li, G. Zhao, P. Li, and Y. Xie, "Rethinking the performance comparison between snns and anns," *Neural networks*, vol. 121, pp. 294–307, 2020.

[47] Y. Wu, L. Deng, G. Li, J. Zhu, and L. Shi, "Spatio-temporal backpropagation for training high-performance spiking neural networks," *Frontiers in neuroscience*, vol. 12, p. 331, 2018.

[48] Y. Wu, L. Deng, G. Li, J. Zhu, Y. Xie, and L. Shi, "Direct training for spiking neural networks: Faster, larger, better," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 33, pp. 1311–1318, 2019.

[49] A. Graves, "Adaptive computation time for recurrent neural networks," *CoRR*, vol. abs/1603.08983, 2016.

[50] G. Huang, D. Chen, T. Li, F. Wu, L. van der Maaten, and K. Q. Weinberger, "Multi-scale dense convolutional networks for efficient prediction," *CoRR*, vol. abs/1703.09844, 2017.

[51] B. Yang, G. Bender, Q. V. Le, and J. Ngiam, "Condconv: Conditionally parameterized convolutions for efficient inference," in *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[52] B. Zhou, A. Khosla, À. Lapedriza, A. Oliva, and A. Torralba, "Learning deep features for discriminative localization," *CoRR*, vol. abs/1512.04150, 2015.

[53] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *CoRR*, vol. abs/1704.04861, 2017.

[54] G. Huang, D. Chen, T. Li, F. Wu, L. van der Maaten, and K. Q. Weinberger, "Multi-scale dense convolutional networks for efficient prediction," *CoRR*, vol. abs/1703.09844, 2017.

[55] S. Teerapittayanon, B. McDanel, and H. T. Kung, "Branchynet: Fast inference via early exiting from deep neural networks," *CoRR*, vol. abs/1709.01686, 2017.

[56] T. Bolukbasi, J. Wang, O. Dekel, and V. Saligrama, "Adaptive neural networks for fast test-time prediction," *CoRR*, vol. abs/1702.07811, 2017.

[57] A. Graves, "Adaptive computation time for recurrent neural networks," *CoRR*, vol. abs/1603.08983, 2016.

[58] Y. Bengio, N. Léonard, and A. C. Courville, "Estimating or propagating gradients through stochastic neurons for conditional computation," *CoRR*, vol. abs/1308.3432, 2013.

[59] A. S. Davis and I. Arel, "Low-rank approximations for conditional feedforward computation in deep neural networks," *CoRR*, vol. abs/1312.4461, 2013.

[60] R. A. Jacobs, M. I. Jordan, S. J. Nowlan, and G. E. Hinton, "Adaptive mixtures of local experts," *Neural Computation*, vol. 3, no. 1, pp. 79–87, 1991.

[61] D. Eigen, M. Ranzato, and I. Sutskever, "Learning factored representations in a deep mixture of experts," *CoRR*, vol. abs/1312.4314, 2013.

[62] A. W. Harley, K. G. Derpanis, and I. Kokkinos, "Segmentation-aware convolutional networks using local attention masks," *CoRR*, vol. abs/1708.04607, 2017.

[63] M. J. Seo, S. Min, A. Farhadi, and H. Hajishirzi, "Neural speed reading via skim-rnn," *CoRR*, vol. abs/1711.02085, 2017.

[64] Y. Jernite, E. Grave, A. Joulin, and T. Mikolov, "Variable computation in recurrent neural networks," *arXiv preprint arXiv:1611.06188*, 2016.

[65] A. Graves, "Adaptive computation time for recurrent neural networks," *CoRR*, vol. abs/1603.08983, 2016.

[66] V. Campos, B. Jou, X. Giró-i-Nieto, J. Torres, and S. Chang, "Skip RNN: learning to skip state updates in recurrent neural networks," *CoRR*, vol. abs/1708.06834, 2017.

[67] C. Hansen, C. Hansen, S. Alstrup, J. G. Simonsen, and C. Lioma, "Neural speed reading with structural-jump-lstm," *CoRR*, vol. abs/1904.00761, 2019.

[68] J. Tao, U. Thakker, G. Dasika, and J. Beu, "Skipping rnn state updates without retraining the original model," in *Proceedings of the 1st Workshop on Machine Learning on Edge in Sensor Systems*, pp. 31–36, 2019.

[69] J. Chung, S. Ahn, and Y. Bengio, "Hierarchical multiscale recurrent neural networks," *arXiv preprint arXiv:1609.01704*, 2016.

[70] A. W. Yu, H. Lee, and Q. V. Le, "Learning to skim text," *arXiv preprint arXiv:1704.06877*, 2017.

[71] T.-J. Fu and W.-Y. Ma, "Speed reading: Learning to read forbackward via shuttle," in *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pp. 4439–4448, 2018.

[72] Z. Wu, C. Xiong, Y.-G. Jiang, and L. S. Davis, "Liteeval: A coarse-to-fine framework for resource efficient video recognition," *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[73] G. Vaudaux-Ruth, A. Chan-Hon-Tong, and C. Achard, "Actionspotter: Deep reinforcement learning framework for temporal action spotting in videos," in *2020 25th International Conference on Pattern Recognition (ICPR)*, pp. 631–638, IEEE, 2021.

[74] Y. Meng, R. Panda, C.-C. Lin, P. Sattigeri, L. Karlinsky, K. Saenko, A. Oliva, and R. Feris, "Adafuse: Adaptive temporal fusion network for efficient action recognition," *arXiv preprint arXiv:2102.05775*, 2021.

[75] X. Sun, R. Panda, C.-F. R. Chen, A. Oliva, R. Feris, and K. Saenko, "Dynamic network quantization for efficient video inference," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 7375–7385, 2021.

[76] Z. Weng, Z. Wu, H. Li, J. Chen, and Y.-G. Jiang, "Hcms: Hierarchical and conditional modality selection for efficient video recognition," *ACM Transactions on Multimedia Computing, Communications and Applications*, vol. 20, no. 2, pp. 1–18, 2023.

[77] Y. Guo, A. Yao, and Y. Chen, "Dynamic network surgery for efficient dnns," *Advances in neural information processing systems*, vol. 29, 2016.

[78] B. Han, F. Zhao, Y. Zeng, and W. Pan, "Adaptive sparse structure development with pruning and regeneration for spiking neural networks," 2023.

[79] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, pp. 8024–8035, Curran Associates, Inc., 2019.

[80] I. Hammouamri, I. Khalfaoui-Hassani, and T. Masquelier, "Learning delays in spiking neural networks using dilated convolutions with learnable spacings," 2023.

[81] A. Bittar and P. N. Garner, "A surrogate gradient spiking baseline for speech command recognition," *Frontiers in Neuroscience*, vol. 16, 2022.

[82] C. Yu, Z. Gu, D. Li, G. Wang, A. Wang, and E. Li, "Stsc-snn: Spatio-temporal synaptic connection with temporal convolution and attention for spiking neural networks," *Frontiers in Neuroscience*, vol. 16, 2022.

[83] M. Yao, H. Gao, G. Zhao, D. Wang, Y. Lin, Z. Yang, and G. Li, "Temporal-wise attention spiking neural networks for event streams classification," *CoRR*, vol. abs/2107.11711, 2021.

[84] T. Nowotny, J. P. Turner, and J. C. Knight, "Loss shaping enhances exact gradient learning with eventprop in spiking neural networks," 2022.