Training Neuromorphic Neural Networks for Speech Recognition

Rihards Klotiņš

Supervisor: Dorian Florescu

May 23, 2025

Abstract

FILLER ABSTRACT IGNORE - Spiking Neural Networks (SNNs), the third generation of neural networks, offer a promising alternative to traditional Artificial Neural Networks (ANNs) due to their energy efficiency, temporal processing capabilities, and potential for neuromorphic hardware implementation. Unlike ANNs, which rely on continuous-valued activations, SNNs process information through discrete spike events, closely mimicking biological neural systems. This characteristic enables them to efficiently handle sequential data, making them suitable for applications such as speech recognition, event-based vision, and edge computing. Training SNNs, however, remains a significant challenge due to the non-differentiability of spike events. While ANN-to-SNN conversion provides a workaround, it imposes computational costs and limits architectural flexibility. Direct training methods, such as Backpropagation-Through-Time (BPTT) with surrogate gradients, local learning rules, and biologically inspired approaches like e-prop and EventProp, have emerged as viable alternatives. Each method presents trade-offs in terms of computational complexity, biological plausibility, and performance. Notably, EventProp has demonstrated state-of-the-art results while reducing memory and computational overhead compared to BPTT. This work explores the theoretical advantages of SNNs, their energy-efficient processing, and recent advancements in training methodologies. It also highlights their potential impact on low-power computing applications, particularly in audio processing, where temporal encoding is crucial. By addressing current limitations and leveraging novel training strategies, SNNs could play a pivotal role in next-generation AI systems.

Contents

1 Introduction **4**

1.1 The Problem to Solve 5

1.2 Objectives 5

2 Context and Past Work **5**

2.1 Spiking Neuron Model 5

2.1.1 Hodgkin–Huxley 5

2.1.2 Leaky Integrate-and-Fire Neuron 7

2.1.3 Izhikevich 9

2.1.4 Neural Networks 10

2.2 Training Neural Networks 10

2.2.1 Backpropagation 11

2.2.2 ANN-to-SNN Conversion 14

2.2.3 Backpropagation-Through-Time 14

2.2.4 Eligibility Propagation 15

2.2.5 Spike-Time-Dependent-Plasticity 16

2.2.6 Eventprop 17

3 Methods **20**

3.1 Dataset used 20

3.2 Software Libraries 21

3.3 Training the Network 21

3.4 Deriving an New and Improved Loss Function and Eventprop Scheme 21

3.5 Implementing the Novel Eventprop Scheme in Software 26

3.6 Profiling the Power Usage of GPU vs CPU 26

4 Results **26**

4.1 Reproducing State-Of-The-Art Accuracy 26

4.2 New Loss Function Improves Training 26

4.3 Power Usage of GPU and CPU 30

5 Discussion **30**

6 Conclusion **31**

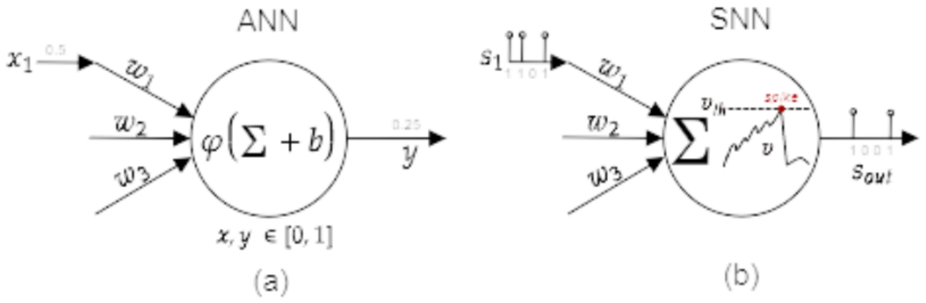


Figure 1: (a) Shows a 2nd generation neuron. (b) Shows a 3rd generation neuron.

1 Introduction

Artificial Neural Networks (ANN) are computational models inspired by the brain; made up of layers of interconnected "neurons" which can learn patterns when given large amounts of data. After learning patterns, ANNs can be used to make inferences, predicting an output based on some given input. ANNs have long been used for application ranging from computer vision to financial forecasting [citation]. However the use of ANNs has surged recently, increasing fourfold since 2017, driven by the development of generative AI models like ChatGPT [1]. To compound this, models are growing in size and becoming more complex, causing the global AI power consumption to increase at an exponential rate [2]. Not only does this make models expensive to operate, but it also raises sustainability concerns.

A graphics processing unit (GPU) running a large language model (LLM) consumes hundreds of watts of power. On the other hand, the human brain - which perceives the world through visual recognition, guides actions through motor control, keeps involuntary biological systems functioning, weaves together natural language to convey meaning, and gives rise to the phenomenon of consciousness - consumes only about 20 watts. Such efficient information processing has motivated research into neuromorphic computing - a field aiming to emulate the brain's neural architecture. This goal sparked the creation of Spiking Neural Networks (SNNs) which are considered the third generation of neural network models [3], following the second generation of neural networks commonly referred to as artificial neural networks (ANNs). Compared to neurons in ANNs, SNN neurons more closely model how biological neurons behave in the brain. For a second generation neuron, the output is a weighted sum of the inputs passed through an activation function (see figure 1 (a)). For a third generation neuron, the output is a spike, sometimes referred to as an "event", which occurs when the voltage of a neuron surpasses a threshold (see figure 1 (b)), this results in neurons communicating with each other via trains of spikes, which vary in timing and frequency. Since information is encoded in the timing of outputs, SNNs can inherently capture temporal patterns, making them capable of processing tasks involving time-dependent signals, such as speech recognition, sensory processing, and event-based data streams. Moreover, it has been shown theoretically that SNNs posses higher computational power than the previous generation of ANNs [4]. Spiking neurons do not require a clock signal, moreover they are inactive most of the time, only activating when they receive a spike; these features allow for significant power savings. For instance, the leading neuromorphic platform TrueNorth is capable of simulating a million spiking neurons in real-time while consuming 63 mW. The equivalent network executed on a high-performance computing platform was 100–200× slower than real-time and consumed 100,000 to 300,000× more energy per synaptic event [5]. Such power savings are highly sought after for edge computing devices that run on battery or have limited power budgets - like drones, smartphones, wearables. Moreover, their low-latency could see them applied to autonomous vehicles and robotics.

1.1 The Problem to Solve

Large Language Models (LLMs) - such as ChatGPT - have ushered in a paradigm shift in how we interact with computers. We can now interface with machines using natural language, and vice versa. Having reached 1 billion users in 3 years - compared to Google's 13 years - and 200 million weekly active users highlights the utility and impact ChatGPT and other LLMs are already having on the global population [6]. While text remains the most common way people interact with LLMs, speech-based interfaces are rapidly gaining traction, with companies investing heavily in voice-driven LLM services. Last year, Meta spent £14.8 billion on its Reality Labs division, which develops smart glasses that primarily use voice as their interface [7]. Apple has been uncharacteristically scrambling, and failing, to release an AI integrated Siri voice assistant [8]. And the company behind a speech-based AI device called Rabbit R1, reached a valuation of £100 million before it had a functional product [9]. SNNs could unlock greater accuracy and longer battery life for these in-demand technologies, reducing latency by allowing speech to be processed on-device.

1.2 Objectives

This project focuses on training spiking neural networks (SNNs) for speech recognition. Section 2 sets the stage by exploring how neurons are modeled, outlining the core principles of neural network training, and reviewing prior work—comparing competing approaches to SNN training and selecting one for replication. Section 3 details the methods used to train networks with the chosen algorithm. It also presents a mathematical derivation and software implementation of a novel variation, developed to address a flaw identified in the original approach. The results and their implications are presented and discussed in Sections 4 and 5.

2 Context and Past Work

2.1 Spiking Neuron Model

In computational neuroscience, several neuron models have been developed to simulate how neurons behave, each offering a different trade-off between biological detail and computational efficiency. This section introduces three widely used models: the Hodgkin–Huxley model, the Leaky Integrate-and-Fire (LIF) model, and the Izhikevich model.

We will begin with the most influential model in neuroscience, the Hodgkin–Huxley model. Introduced in 1952 and awarded the Nobel Prize in Physiology or Medicine in 1963, this model provides a detailed mathematical description of how electrical signals - action potentials - are generated and propagated in neurons. While this model is highly accurate and biologically grounded, its complexity makes it computationally demanding, which limits its use in large-scale or real-time simulations.

Next we will discuss the Leaky Integrate-and-Fire model. To improve efficiency, the Leaky Integrate-and-Fire model simplifies neuron behaviour while retaining essential features. It treats the neuron like an electrical circuit that accumulates incoming signals over time. When the membrane voltage crosses a set threshold, the neuron emits a spike and resets. Though simplified, this model captures key spiking behavior and is highly efficient to compute, making it the most commonly used neuron model in neuromorphic engineering and machine learning applications.

Finally, we will discuss the Izhikevich model which offers a compromise between realism and efficiency. Using just two simple differential equations, it can reproduce a wide range of spiking and bursting patterns seen in real neurons.

2.1.1 Hodgkin–Huxley

The Hodgkin-Huxley is a detailed neuron model developed in 1952 by Alan Hodgkin and Andrew Huxley. It was the first model to quantitatively explain how neurons generate and propagate electrical impulses by describing the

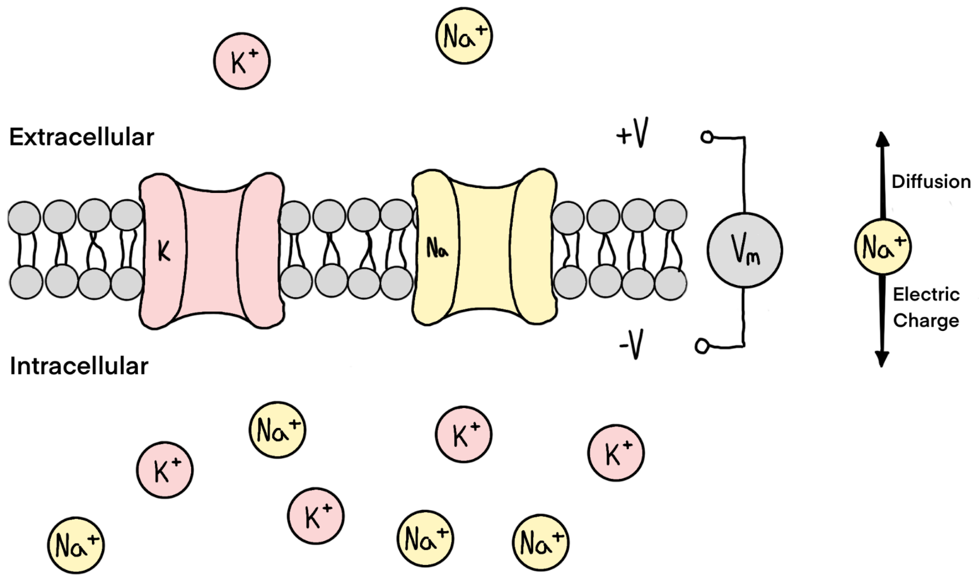


Figure 2: Diagram of neuron membrane.

flow of sodium and potassium ions through voltage-gated channels. Its success in reproducing the full waveform of electrical impulses earned Hodgkin and Huxley the 1963 Nobel Prize in Physiology or Medicine.

A neuron is enclosed in a cell membrane, with gates for ion flow. We will consider positively charged potassium (K+) and sodium (Na+) ions as they are considered as the primary contributors to the electrical behaviour of neurons. The neuron membrane has channels which allow ions to flow between the inside and the outside of the neuron; these channels are selective, allowing only specific ions to pass through, e.g. only allowing Na+ to flow. The amount of flow they allow depends on the membrane voltage V_m, which is the measure of voltage difference between the inside and outside of the neuron.

Two factors govern the tendency of ions to flow between the inside and outside; diffusion and electric charge. Diffusion is the tendency of ions to flow from areas of high concentrations to low concentrations; in figure 2 Na+ ions will tend to flow to the extracellular side of the membrane as the concentration of Na+ ions is lower there. Particles which have the same electric charge repel, particles with opposite electrical charges attract. Therefore, Na+ ions will be attracted towards the intracellular side of the membrane due to the inside of the neuron having a negative electric charge. The directions of these opposing forces are displayed in figure 2. Naturally the cell reaches an equilibrium where the diffusion and electric charge forces are equal. When an external current is applied to the system, the equilibrium is lost and the system experiences transient behaviour defined by the mathematics of the Hodgkin–Huxley model.

The cell membrane acts as a dielectric separating two charged mediums, in other words it acts as a capacitor. According to the capacitor current-voltage equation, we know that $C\frac{dV}{dt} = I$. Applying this to the context of the neuron membrane, we get equation (8). Where V_m is the potential difference across the cell membrane, C_m is a constant representing the characteristic capacitance of the cell membrane, I_{Na} and I_K are the net charge flows of sodium and potassium ions respectively, I_L is the leakage current caused by the movement of other ions, and I_{ext} is the external current applied to the neuron. So we see that the sum of currents causes the voltage to change.

$$C_m \frac{dV_m}{dt} = -(I_{Na} + I_K + I_L - I_{ext}) \tag{1}$$

The currents - I_{Na}, I_K, and I_L - are determined by equations (2), (3) and (4). They can be examined like the classic $I = V/R = Vg$ equation (g being conductance). Where the left hand side is the current, the part in the brackets is the voltage, and the part outside the brackets is the conductance of the gates which the ions flow through.

$$I_{Na} = \bar{g}_{Na} m^3 h (V_m - E_{Na}) \tag{2}$$

$$I_K = \bar{g}_K n^4 (V_m - E_K) \tag{3}$$

$$I_L = \bar{g}_L (V_m - E_L) \tag{4}$$

When there is an imbalance in the force of *electric attraction* and *diffusion*, there is a *net flow of ions*, i.e. a current. This represented in equation (2) by the term $(Vm - E_{Na})$. Where E_{Na} is the equilibrium potential due to diffusion forces. When the two variables are in balance their difference is 0, thus the current is also zero. The same also applies for equations (3) and (4).

As per equation (2) the current also depends on the *conductance* of the channels which the ions move through. There are specialised channels for Na+ ions and for K+ ions, which have different conductances. The conductance of sodium channels is given by $\bar{g}_{Na} m^3 h$. The $\bar{g}_{Na}$ term is the conductance of the channel when it is fully open, i.e. the channels maximum conductance. The $m^3 h$ term is the probability - 0 to 1 - that a sodium channel is open. This probability comes from the fact that each channel has 3 "m" gates and 1 "h" gate, ions can only flow through the channel when all gates are open. Due to the large number of channels in a neuron, the value of the probability will correspond to what proportion of the channels are open. The probabilities of $m,\ h,\ n$ are given by equation (5), where α_x and β_x are rate constants at which speed the gates open and close respectively, they are voltage dependent.

$$\frac{dx}{dt} = \alpha_x(V_m)(1 - x) - \beta_x(V_m)(x),\ where\ x \in \{m, h, n\} \tag{5}$$

Simple first order ion flow equations characterise how the neurons in our brain function to a high degree of accuracy. Figures 3, 4, and 5 show how a neuron would react to a 5ms, 15ms, and 20ms pulse of current. While the HH neuron model has propelled our understanding of neurons due to it's biophysical precision, it's complexity makes it computationally expensive to simulate. As a result researchers had to look elsewhere to find a neuron model which could be used to efficiently train machines.

2.1.2 Leaky Integrate-and-Fire Neuron

The leaky integrate-and-fire (LIF) neuron model is the most widely used neuronal model in SNNs due to its simplicity and efficiency which helps it scale for large networks. The neuron receives spikes from "pre-synaptic" neurons (see figure 1 (b)) - e.g. neurons which feed into the neuron in question. These spikes increase the "membrane potential" of the neuron - e.g. the voltage between the inside and outside of the neuron. When the membrane potential reaches a threshold voltage, the neuron in question fires a spike to the neurons that it is connected to, and it's membrane voltage resets to a baseline value.

In order to be modelled computationally, this behaviour is expressed in mathematical terms. Spikes received by a neuron at time t induce a current $X[t]$. This input current increases the voltage between the inside and outside of the neuron, H[t] is the intermediate value for the voltage. The voltage decays over time, the speed of the decay depends on the membrane time constant, τ. This is what equation (6) describes. This spiking logic is defined in equation (7), where $\Theta(x)$ is a function that is 0 unless $x \geq 0$. Equation (8) describes that when the threshold voltage is reached, $V[t]$ is set to V_{reset} and a spike is released; if the threshold voltage is not reached then $V[t] = H[t]$.

$$H[t] = V[t-1] + \frac{1}{\tau}(X[t] - (V[t-1] - V_{reset})), \tag{6}$$

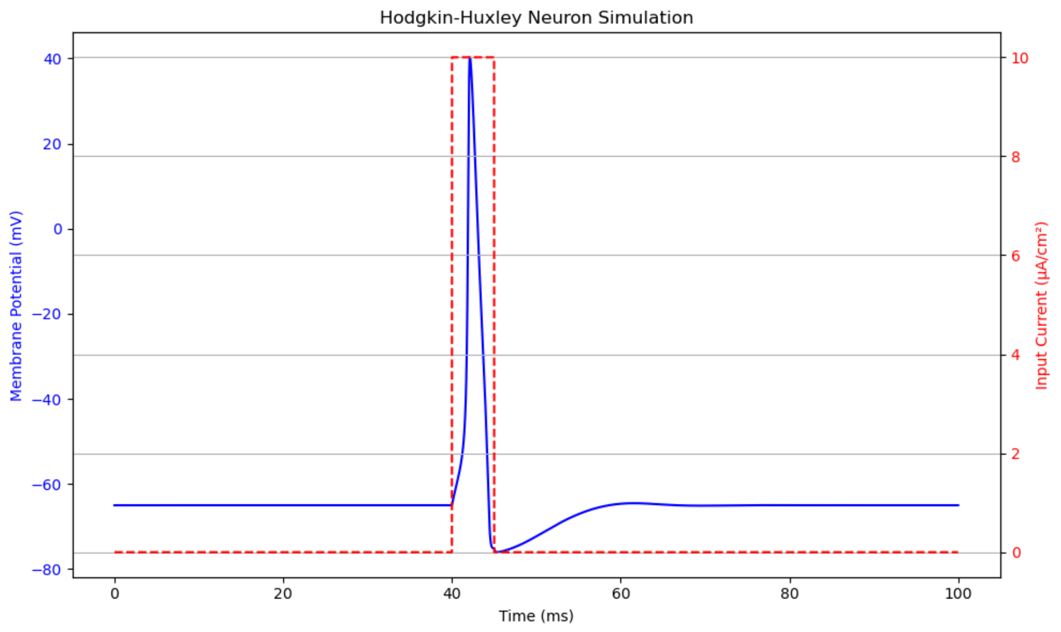


Figure 3: Voltage response of the Hodgkin–Huxley model (blue) to a 5ms step function input current (red).

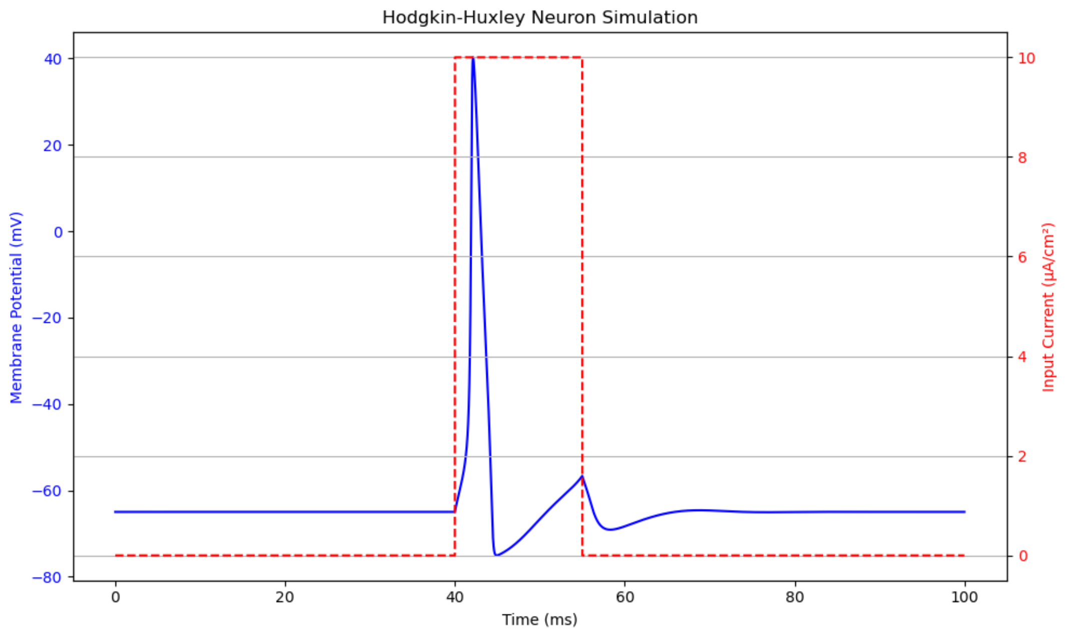


Figure 4: Voltage response of the Hodgkin–Huxley model (blue) to a 10ms step function input current (red).

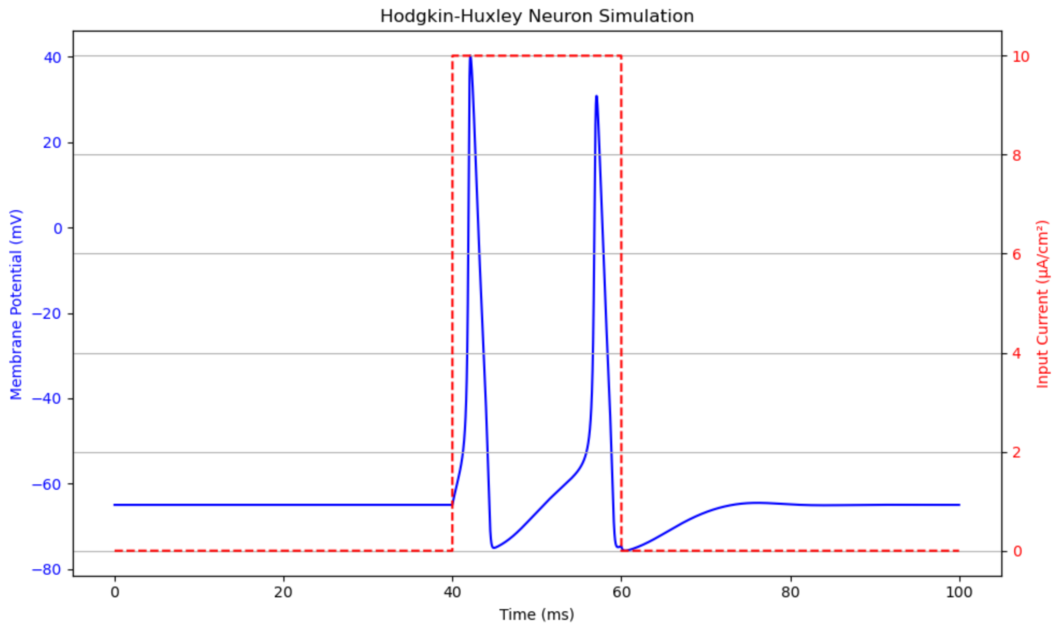


Figure 5: Voltage response of the Hodgkin–Huxley model (blue) to a 15ms step function input current (red).

$$S[t] = \Theta(H[t] - V_{th}), \tag{7}$$

$$V[t] = H[t](1 - S[t]) + V_{reset}S[t] \tag{8}$$

Here is what the spiking and decaying behaviour of a leaky integrate-and-fire neuron looks like. Spikes are released at the output of the neuron at,

2.1.3 Izhikevich

Building on the simplicity of the leaky integrate-and-fire (LIF) neuron, which uses a single differential equation and a fixed threshold to generate spikes, the Izhikevich model strikes a balance between biological realism and computational efficiency [10]. Eugene Izhikevich observed that Hodgkin–Huxley-type models, while highly detailed, require integrating four equations per neuron, which make large-scale simulation infeasible, and that simple integrate-and-fire models cannot reproduce many cortical firing patterns [10]. To address this, he derived a minimal two-variable model that can emulate diverse neuron behaviours by tuning just four parameters [11].

Mathematically, the model comprises these coupled ordinary differential equations:

$$\frac{dt}{dv} = 0.04v^2 + 5v + 140u + I(t), \tag{9}$$

$$\frac{du}{dt} = a(b * v - u), \tag{10}$$

Where v is the membrane potential (in mV), u is a membrane recovery variable accounting for K+ activation and Na+ inactivation, and $I(t)$ is an external input current [10]. When v reaches the peak (typically 30 mV), it is reset alongside u :

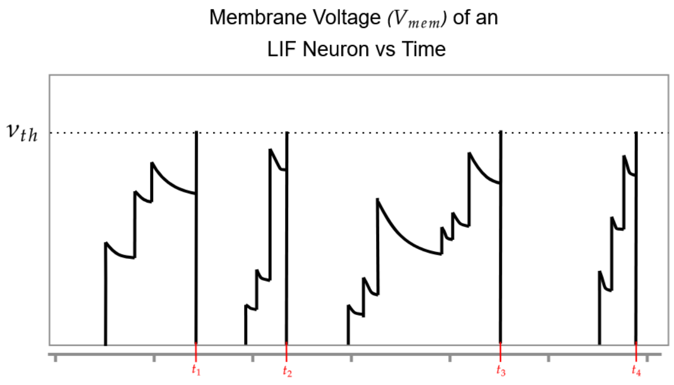


Figure 6: Display of the the spiking and decaying behaviour of a leaky integrate-and-fire neuron. Spikes are released at the output of the neuron at t_1, t_2, t_3, and t_4.

$$\text{if } v \geq 30\,\mathrm{mV}, \quad \begin{cases} v \leftarrow c \\ u \leftarrow u + d \end{cases} \tag{11}$$

with (a, b, c, d) chosen to match specific neuron types. Despite its simplicity, this formulation reproduces over twenty cortical firing patterns by varying these four parameters, far surpassing the expressive capacity of the LIF model [11]. Moreover, Izhikevich demonstrated that this model runs in real time for tens of thousands of neurons on a single CPU core - comparable to LIF performance.

2.1.4 Neural Networks

However, alone, a neuron does not do anything very impressive; in a network, intelligent behaviour can emerge. When many neurons are connected together, forming a neural network, they can collectively process information, detect patterns, and make decisions. Each neuron receives inputs from other neurons through connections called synapses (figure 7). These connections have strengths, or "weights", which determine how much influence one neuron has on another; these strengths are visualised by the darkness of the connections in figure 7. By carefully adjusting these weights, the network can learn to perform complex tasks such as image recognition, speech processing, or decision-making. This idea underpins both biological neural circuits and artificial neural networks used in machine learning.

Recurrent Neural Networks (RNNs) are a class of neural networks which incorporate feedback connections. Unlike typical feedforward networks, where information moves in one direction from input to output, RNNs allow connections that loop back to previous layers or neurons. This architecture enables the network to retain a memory of past inputs. In spiking neural networks this allows the network to encode temporal patterns and maintain memory.

2.2 Training Neural Networks

The objective of training a neural network for spoken word recognition is to construct a model that accurately maps audio input to the correct lexical output. Supervised learning remains the dominant training paradigm in this domain, consistently achieving state-of-the-art results in speech recognition tasks [12, 13]. Supervised learning relies on labelled datasets—typically composed of audio clips annotated with their corresponding transcriptions—providing explicit guidance for the model to learn the mapping between acoustic features and linguistic units. In contrast, unsupervised learning operates on unlabelled data, requiring the model to uncover inherent structure or patterns within the input without external annotation. In this section I will introduce backpropagation and explore different ways that spiking neural networks can be trained.

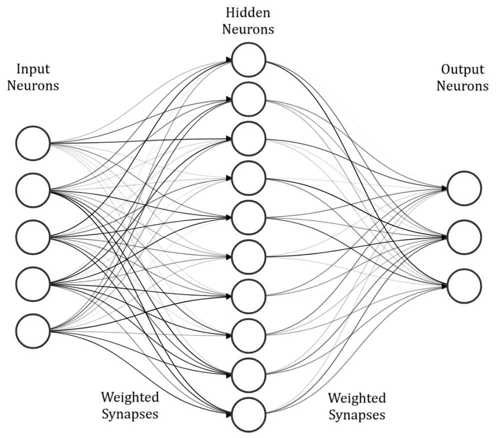


Figure 7: Diagram of a 3 layer neural network. The weights of the synapses are visualised by the darkness of the connections.

2.2.1 Backpropagation

Backpropagation - an efficient implementation of gradient descent - is the most effective and widely used method for training second generation artificial neural networks. It provides a systematic way to adjust the internal parameters - or weights - of a network to minimize the discrepancy between the network's predictions and the desired outputs. An overview of the backpropagation process is as follows:

1. Forward Pass: The model receives an input and processes it through its layers, generating an output.
2. Error Calculation: The output is compared to the desired target, and a loss function quantifies the error.
3. Gradient Computation: The derivative of the loss function is computed with respect to each weight in the network using the chain rule of calculus.
4. Weight Update: The weights are adjusted by subtracting a fraction of the corresponding gradient, typically scaled by a learning rate. This step moves the model toward a configuration that reduces error.

In supervised learning, the desired output is the label of the data. When a model predicts an output, the "loss" is calculated to quantify the error between the actual output and the desired output. For instance, a common way to calculate the loss is by calculating the mean squared error (MSE), where you find the sum of the squared differences between the actual output neuron values and the desired output neuron values (Equation (12).

$$L = \frac{1}{n}\sum_{i=1}^{n}(\mathbf{y}_i - \hat{\mathbf{y}}_i)^2 \tag{12}$$

To visualise how the loss of a network is minimised, let us assume that a network has 2 weights, θ_1 and θ_2. Figure 8 shows the loss plotted against the weights θ_1 and θ_2. The optimal weight configuration occurs at the lowest point on the surface. If the weights of the neural network get initialised to the point P, finding the gradient of the surface at that point which will indicate the direction of the steepest ascent. Therefore, going along the surface in the opposite direction of the gradient will follow the fastest path down the surface, minimising the loss function until the local minimum is reached where the gradient becomes 0. This process is known as *gradient descent*, backpropagation is an efficient implementation of gradient descent where the gradient of the network is calculated backwards layer by layer.

To demonstrate how backpropagation works, let's consider a simplified, fully-connected network consisting of a layer of input neurons, hidden neurons, and output neurons (Figure 9). Fully-connected means each neuron in a layer is connected to all neurons in the following layer.

The highlighted neuron, $\mathbf{h}_1$, is connected to all input neurons, $\mathbf{x}_1$, $\mathbf{x}_2$, and $\mathbf{x}_3$. Therefore $\mathbf{h}_1$ is a weighted sum of

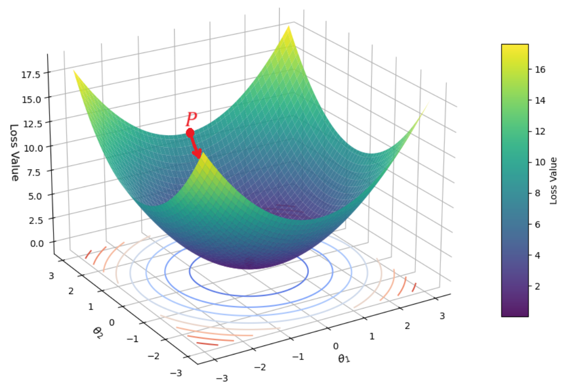


Figure 8: The loss as a function of the parameters θ_1 and θ_2 plotted.

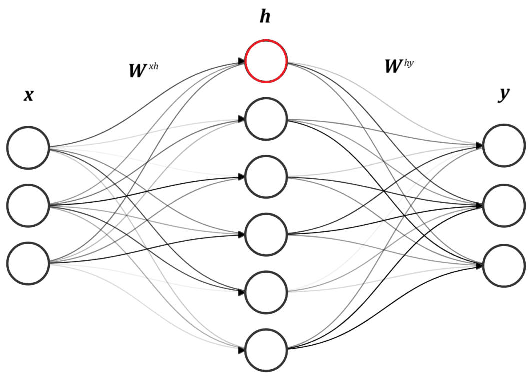


Figure 9: The input. $\mathbf{x}$, is a vector of length 3. $\mathbf{h}$ is a hidden layer of length 6. $\mathbf{W}^{xh}$ is the matrix containing the values of each of the weights connecting $\mathbf{x}$ and $\mathbf{h}$, thus it is of size 3 by 6. $\mathbf{y}$ is the output vector of length 3. $\mathbf{W}^{hy}$ is the matrix containing the values of each of the weights connecting $\mathbf{h}$ and $\mathbf{y}$. thus it is of size 6 by 3.

$$\left[\boxed{x_1 \quad x_2 \quad x_3}\right] \times \left[\begin{array}{cccccc} \boxed{\begin{matrix} w_{11} \\ w_{21} \\ w_{31} \end{matrix}} & \begin{matrix} w_{12} \\ w_{22} \\ w_{32} \end{matrix} & \begin{matrix} w_{13} \\ w_{23} \\ w_{33} \end{matrix} & \begin{matrix} w_{14} \\ w_{24} \\ w_{34} \end{matrix} & \begin{matrix} w_{15} \\ w_{25} \\ w_{35} \end{matrix} & \begin{matrix} w_{16} \\ w_{26} \\ w_{36} \end{matrix} \end{array}\right] = \left[\boxed{h_1} \quad h_2 \quad h_3 \quad h_4 \quad h_5 \quad h_6\right]$$

Figure 10: Matrix calculation of neural network values. The highlighted values of vector $\mathbf{x}$ are multiplied by highlighted values of the $\mathbf{W}$ matrix and summed to get $\mathbf{h}_1$.

all the input neurons (Equation 13).

$$\mathbf{h}_1 = \sum_{i=1}^{3} \mathbf{x}_i \times \mathbf{W}_i^{xh} \tag{13}$$

An efficient way to write these weighted summations for all of the neurons in a layer is by using matrix algebra.

$$\mathbf{h} = \mathbf{x} \times \mathbf{W}^{xh} \tag{14}$$

$$\mathbf{y} = \mathbf{h} \times \mathbf{W}^{hy} \tag{15}$$

These two equations encompass all of the weighted summations that define the network. The relationship between the weighted summation equation (equation **??**) and the matrix multiplication representation of the network is highlighted in figure 10. You can see that each input neuron x_i is multiplied by the corresponding weight $w_{i,1}$ and summed to give h_1.

Matrix multiplications perform the *forward pass* of the network - i.e. the inference. This inferred output value, y, is compared with the true "label" value, $\mathbf{y}_{label}$. The comparison between the actual output and the desired output is dome by a loss function, $l()$. There are many ways of implementing the loss function, one way by finding the mean squared error (MSE):

$$L = l(\mathbf{y}, \mathbf{y}_{label}) \tag{16}$$

The goal is to minimize this loss by updating the network's weights in the direction of the negative gradient. This requires computing the gradient of the loss with respect to each weight w_i:

$$\frac{\partial L}{\partial \mathbf{W}_{ij}} \tag{17}$$

We first calculate the gradient of the layer closest to the output, W^{hy}.

$$\frac{\partial L}{\partial \mathbf{W}^{hy}} = \frac{\partial L}{\partial \mathbf{y}} \cdot \frac{\partial \mathbf{y}}{\partial \mathbf{W}^{hy}} = \frac{\partial L}{\partial \mathbf{y}} \cdot \mathbf{h} \tag{18}$$

We then use the chain rule again to calculate the gradient of L with respect to W^{xh}.

$$\frac{\partial L}{\partial \mathbf{W}^{xh}} = \frac{\partial L}{\partial \mathbf{y}} \cdot \frac{\partial \mathbf{y}}{\partial \mathbf{h}} \cdot \frac{\partial \mathbf{h}}{\partial \mathbf{W}^{xh}} = \frac{\partial L}{\partial \mathbf{y}} \cdot \mathbf{W}^{hy} \cdot \mathbf{x} \tag{19}$$

This step-by-step application of the chain rule allows the error to be propagated backward through the network, enabling efficient computation of gradients at each layer - this is the essence of backpropagation.

Once the gradients are known, the weights are updated using the gradient descent rule:

$$\mathbf{w}_{ij} \leftarrow \mathbf{w}_{ij} - \eta \frac{\partial L}{\partial \mathbf{w}_{ij}} \tag{20}$$

where η is the learning rate. A higher η would mean faster learning but risks instability. A lower η would ensure stable convergence to the minima, but may make the training very slow.

Repeating this backpropagation process across many training examples allows the network to gradually learn the desired input-output mapping by descending the loss function surface. A problem with this approach to be aware of is that the gradient descent algorithm may find a local minimum of the loss function instead of the global minimum. As a result the performance of the model may be significantly lower than it potentially could be. A popular way of dealing with this problem is by training the network several times with different random weight initialisations; if one network initialisation gets stuck in a local minima, others may not. If there are too many local minima, this approach may not be enough.

While backpropagation has proven to be a powerful algorithm for training artificial neural networks, there have been doubt about weather it can be applied to spiking neural networks (SNNs), due to the non-differentiable nature of spike events, which prevents the straightforward calculation of gradients required for weight updates. In the following section, we introduce several methods for training spiking neural networks, highlighting how they address the challenges of spike-based computation and enable effective learning in spiking systems.

2.2.2 ANN-to-SNN Conversion

Due to the popularity of ANNs, literature and methods for training them is advanced. This has lead people to train ANNs using state-of-the-art methods, and converting the ANNs to SNNs. Usually a trained artificial neural network is transformed into a spiking neural network by substituting activation functions between neurons with spiking neuron models. This process often involves additional adjustments such as weight normalization and threshold balancing to maintain performance and stability. These ways of training have had good results for some tests [14, 15]. However, such a method incurs large computational costs during conversion and is limited by the architecture of ANNs which are less adaptable to dynamic data like audio [16]. Thus, to fully harness the benefits of SNNs — from energy efficiency to novel architectures — effective direct training methods are essential.

2.2.3 Backpropagation-Through-Time

Similar to the ANN-to-SNN method, Backpropagation-Through-Time (BPTT) attempts to carry over the effectiveness of the backpropagation algorithm to SNNs. However, this time the SNNs get trained directly which allows them to learn dynamic patterns from temporal data. Familiar gradient-based methods are applied in tandem with "surrogate gradients" (SG) which are spike gradient approximations used to overcome the fact that spikes are non-differentiable. Thinking of the network's activity over time as a very deep chain of simple processing steps; BPTT "unrolls" this chain so that the error at the end can be traced back step by step to adjust every connection [17]. Because a spike is a discontinuous event (it either happens or it doesn't), we replace its true derivative - which is zero almost everywhere and jumps to infinity at spike times - with a smooth "surrogate" function during training. This surrogate lets us compute approximate gradients so that standard optimisers like gradient descent can still work [16].

When applied to speech-recognition benchmarks - such as the Spiking Speech Commands (SSC) and Spiking Heidelberg Digits (SHD) datasets - this method achieves accuracy on par with conventional neural networks while operating in a sparse, event-driven fashion that can be more energy-efficient at inference time [15, 18]. Researchers have swapped out recurrent layers in end-to-end speech models for surrogate gradient trained spiking modules, showing only small drops in word-error rate and offering a path toward low-power, real-time processing [15].

However, BPTT comes with two major downsides. First, it requires storing every intermediate state over the entire duration of an input—meaning memory usage grows with the length of the audio clip, which can quickly exceed hardware limits for long recordings [18]. Second, because the learning rule relies on a global error signal propagated across many time steps and layers, it differs starkly from the local, synapse-by-synapse learning observed in biological brains, this means training would not be feasible directly on neuromorphic platforms.

Parallelisable LIF A major bottleneck in training spiking neural networks (SNNs) using BPTT is the strictly sequential nature of classic Leaky Integrate-and-Fire (LIF) neurons, which update their membrane potential step by step in time. The Parallelizable LIF (ParaLIF) model overcomes this by decoupling the linear integration of inputs from the spiking (thresholding) operation and executing both across all time steps in parallel. This reorganization leverages highly optimized matrix operations on modern accelerators to deliver orders of magnitudes of speed-ups in training [19, 20].

In a standard LIF neuron, the membrane potential $V(t)$ at time t depends on its previous value $V(t1)$ plus any new inputs, and a spike is emitted once V crosses a threshold. ParaLIF rewrites this process as two separate GPU kernels. The first kernel computes, for every neuron, the entire sequence of membrane-potential updates in one batched matrix multiplication; the second applies the threshold-and-reset rule simultaneously at all time points. By removing the need for "time-step loops," ParaLIF converts a fundamentally serial simulation into a fully vectorized parallel computation [19].

When benchmarked on neuromorphic speech (Spiking Heidelberg Digits), image and gesture datasets, ParaLIF achieves up to 200× faster training than conventional LIF models, while matching or exceeding their accuracy with comparable levels of sparsity [19, 20]. Compared to other parallel schemes ParaLIF maintains similar speed-ups on short sequences and far greater scalability on very long inputs [20].

However, this method comes with notable disadvantages. By reorganizing the time dimension, ParaLIF departs from the continuous, step-by-step integration that real neurons exhibit, reducing its biological plausibility. This parallel update can also undermine the network's ability to capture fine temporal dependencies, since precise spike timing and sequential context are approximated rather than explicitly modelled [21, 22]. Moreover, the specialized GPU kernels and data-layout transformations needed for ParaLIF introduce implementation complexity and may not map efficiently to more constrained neuromorphic hardware, limiting its applicability in low-power edge scenarios, a key potential application of SNNs.

2.2.4 Eligibility Propagation

In contrast, eligibility propagation, or e-prop, is a method for training spiking neural networks (SNNs) that aligns more closely with how learning is believed to occur in the brain while still taking inspiration from the clearly effective backpropagation algorithm. Unlike traditional training methods like backpropagation-through-time (BPTT), which require storing the entire history of neuron activities and propagating errors backward through time, e-prop simplifies this process by using eligibility traces and a learning signal. Eligibility traces act like short-term memories at each synapse, recording recent activity patterns. They capture how the timing of spikes affects the potential for learning. The learning signal is a global factor that represents the overall error or feedback from the network's output. Instead of sending detailed error information back through every layer and time step, as in BPTT, e-prop uses this single signal to modulate the eligibility traces. When the network makes a mistake, the learning signal adjusts the synapses with high eligibility traces, effectively correcting the connections that contributed most to the error.

By updating synaptic weights immediately based on recent pre- and post-synaptic activity, e-prop reduces memory requirements compared to BPTT [23] and can dramatically lower energy consumption on event-driven hardware [24, 25]. For instance, spiking recurrent networks were trained on the neuromorphic SpiNNaker 2 system for the Google Speech Commands dataset, achieving over 91% accuracy with only 680KB of training memory over 12× lower energy consumption than GPU-based BPTT solutions [26].

Though it can be argued that the potential of e-prop is limited due to it not scaling well. It requires maintaining multiple eligibility traces per synapse (e.g., for membrane potential and adaptive threshold), as well as optimizer state such as moment vectors, resulting in significant memory overhead for large networks [25**?**]. Moreover, because

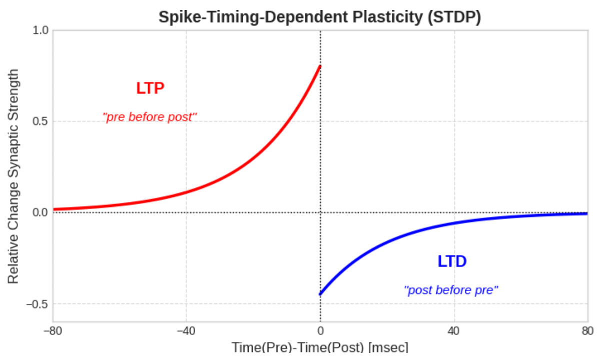


Figure 11: The change of synaptic weight plotted against the difference in timing of the pre- and post-synaptic neuron spikes, showing the long term potential and long term depression phenomena.

it employs approximate surrogate gradients rather than true backpropagation, e-prop–trained models typically achieve lower accuracy than their BPTT-trained counterparts [23]. Third, although e-prop avoids backward error propagation through time, it still depends on a global learning signal to modulate local eligibility traces, introducing communication overhead and deviating from strictly local synaptic updates—factors that can limit its energy efficiency on distributed neuromorphic hardware [24].

2.2.5 Spike-Time-Dependent-Plasticity

Spike-timing-dependent plasticity (STDP) represents a training approach for SNNs that draws even greater inspiration from the learning rules observed in biological neurons. STDP is a biological learning rule that adjusts the strength of synaptic connections based on the precise timing of pre-synaptic and post-synaptic spikes. If a pre-synaptic neuron fires just before a post-synaptic neuron, the connection between them is strengthened. Conversely, if the order is reversed, the connection is weakened. This temporally sensitive form of Hebbian learning, often summarized by the maxim “cells that fire together, wire together,” operates locally at each synapse and does not require global error signals. This makes it inherently biologically plausible, asynchronous, and capable of unsupervised learning.

Reward-modulated STDP (R-STDP) extends the basic STDP rule by incorporating a global reward or punishment signal that modulates the synaptic weight changes. This allows the network to learn task-specific features by reinforcing connections that contribute to correct outputs and weakening those associated with errors. R-STDP bridges the gap between unsupervised STDP and supervised learning, enabling SNNs to tackle more complex, goal-oriented tasks like speech recognition. The reward signal in R-STDP can potentially be implemented using neuromodulators, further enhancing the biological plausibility of SNN training.

STDP has been employed to train SNNs for speech recognition, often in conjunction with temporal coding schemes such as time-to-first-spike for efficient processing. It has been used for unsupervised feature extraction from speech signals, with subsequent classification using methods like Hidden Markov Models (HMMs) or tempotrons. STDP-based convolutional SNNs have shown promise in achieving accurate and efficient speech recognition, with performance levels approaching those of traditional ANNs in certain scenarios.

Memristors are two-terminal devices whose conductance changes based on the history of voltage or current, closely mimicking how biological synapses adjust their efficacy [27]. In memristor-based STDP, each memristor stores a synaptic weight in its conductance, and weight updates occur directly on-chip whenever spikes arrive, following the device’s own switching dynamics [28]. By collocating memory and computation, this approach avoids the von Neumann bottleneck and enables energy-efficient, on-device learning in neuromorphic hardware [29].

Vlasov et al. (2022) demonstrated this concept on a spoken-digit recognition task by training spiking neural networks with memristor-based STDP using two memristor types - poly-p-xylylene (PPX) and CoFeB–LiNbO nanocomposite [30]. Their networks, deployed entirely on neuromorphic hardware, achieved classification accuracies between 80 % and 94 % depending on network topology and decoding strategy, rivalling more complex off-chip learning algorithms while consuming minimal power and memory [31].

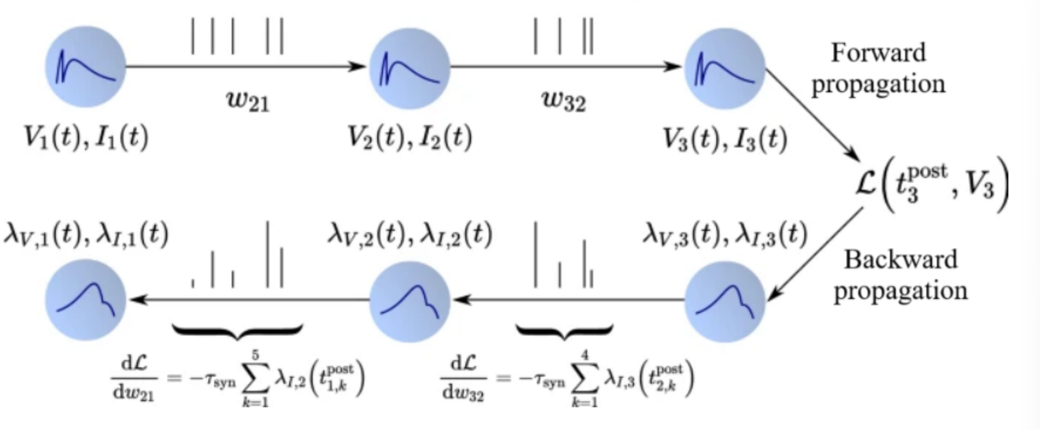

w_{21}
w_{32}
$V_1(t), I_1(t)$
$V_2(t), I_2(t)$
$V_3(t), I_3(t)$
Forward propagation
$\mathcal{L}\left(t_3^{\text{post}}, V_3\right)$
Backward propagation
$\lambda_{V,1}(t), \lambda_{I,1}(t)$
$\lambda_{V,2}(t), \lambda_{I,2}(t)$
$\lambda_{V,3}(t), \lambda_{I,3}(t)$
$\frac{d\mathcal{L}}{dw_{21}} = -\tau_{\text{syn}} \sum_{k=1}^{5} \lambda_{I,2}\left(t_{1,k}^{\text{post}}\right)$
$\frac{d\mathcal{L}}{dw_{32}} = -\tau_{\text{syn}} \sum_{k=1}^{4} \lambda_{I,3}\left(t_{2,k}^{\text{post}}\right)$

Figure 12: Spikes propagated forward according to LIF neuron dynamics. Error signals propagated backward via computing adjoint variables backwards in time. [35]

While promising, training deep spiking neural networks to achieve state-of-the-art accuracy on complex tasks using STDP-based approaches presents significant challenges compared to conventional backpropagation methods [32]. Specifically, STDP itself can be sensitive to hyperparameter choices and network architecture [33]. Moreover, it may prioritize frequently occurring features over those most discriminative for a given task [34].

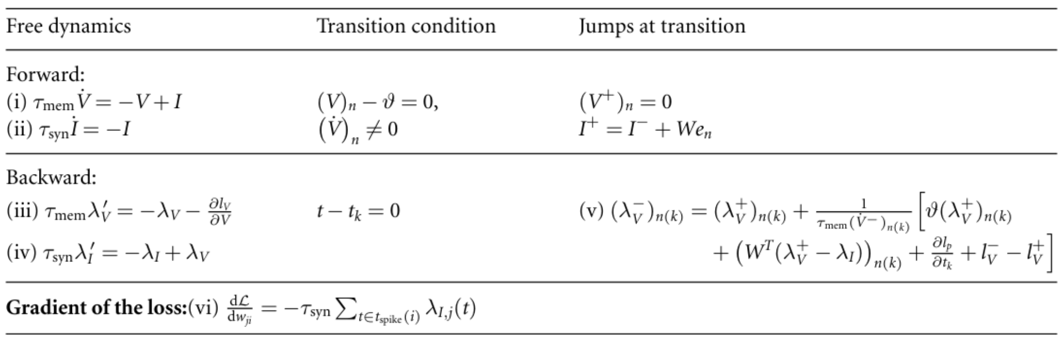
2.2.6 Eventprop

A novel, accurate, and efficient training algorithm. Eventprop uses the adjoint method from optimisation theory to implement backpropagation in an efficient manner for spiking neural networks. Unlike other implementations of backpropagation for SNNs, Eventporp computes *exact* gradients to train the network, as opposed to approximate surrogate gradients typically used in BPTT. Moreover, it does so while using less compute and memory resources, reaching state-of-the-art (SOTA) performance for the SHD dataset while using 4x less memory and running 3x faster [35].

What allows Eventprop to be so efficient is its event-driven nature. Similar to the forward propagation of information in an SNN, the backward propagation of error signals in Eventprop occurs only at the precise times of recorded spikes [36], see how in figure 12 the backpropagated error signals occur at the same time in the backward propagation as the spike signals do in the forward propagation. This results in significant temporal and spatial sparsity in the computation, leading to reduced computational cost and improved efficiency, especially for parallel computing architectures such as the ones in GPUs. Furthermore, Eventprop offers favourable memory requirements compared to methods like BPTT. It only necessitates the storage of neuron states at the times of spike events, rather than at every time step, which can drastically reduce memory usage, particularly for long input sequences common in speech recognition tasks. This memory efficiency makes Eventprop well-suited for implementation on resource-constrained neuromorphic hardware, this was recently explored, with successful deployments on platforms like Intel's Loihi 2, showcasing its potential for low-power, event-based machine learning on specialized hardware [37].

Eventprop has demonstrated its effectiveness on challenging speech recognition benchmarks. It has achieved state-of-the-art performance on the Spiking Heidelberg Digits (SHD) dataset and shown good accuracy on the Spiking Speech Commands (SSC) dataset when training recurrent spiking neural networks. Notably, when compared to leading surrogate gradient-based SNN training methods, implementations of Eventprop have been shown to be significantly faster (up to 3x) and require considerably less memory (up to 4x) [35]. This efficiency allows for scaling SNN training to more complex tasks and larger models, which have yet to be explored and could potentially increase the accuracy of models further.

Eventprop's flexibility extends to its compatibility with a wider class of loss functions [35], including those commonly used with BPTT, allowing researchers to tailor the training process to specific task requirements. Furthermore, the Eventprop formalism can be extended to incorporate learnable delays within SNNs, which can be crucial for capturing temporal dependencies in sequential data like speech. It has been shown that training neuron model



Free dynamics	Transition condition	Jumps at transition
Forward:		
(i) $\tau_{\text{mem}}\dot{V} = -V + I$	$(V)_n - \vartheta = 0,$	$(V^+)_n = 0$
(ii) $\tau_{\text{syn}}\dot{I} = -I$	$(\dot{V})_n \neq 0$	$I^+ = I^- + We_n$
Backward:		
(iii) $\tau_{\text{mem}}\lambda_V' = -\lambda_V - \frac{\partial l_V}{\partial V}$	$t - t_k = 0$	(v) $(\lambda_V^-)_{n(k)} = (\lambda_V^+)_{n(k)} + \frac{1}{\tau_{\text{mem}}(\dot{V}^-)_{n(k)}}\Big[\vartheta(\lambda_V^+)_{n(k)}$ $+ \big(W^T(\lambda_V^+ - \lambda_I)\big)_{n(k)} + \frac{\partial l_p}{\partial t_k} + l_V^- - l_V^+\Big]$
(iv) $\tau_{\text{syn}}\lambda_I' = -\lambda_I + \lambda_V$		
Gradient of the loss:(vi) $\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}w_{ji}} = -\tau_{\text{syn}}\sum_{t\in t_{\text{spike}}(i)}\lambda_{I,j}(t)$		

Figure 13: Spikes propagated forward according to LIF neuron dynamics. Error signals are propagated backword according to the adjoint system. [35]

parameters like the membrane time constant or the synapse time constants - which control how fast the neuron voltage and current decay - can achieve good results.

In summary, Eventprop is a direct training method - unlike ANN-to-SNN conversion - which enables it to train networks of architectures more suited to SNNs without requiring the costly conversion step. Eventprop performs more efficiently - both computationally and memory wise - than BPTT implementations, scaling better for longer sample durations. In addition, unlike the Parallelisable-LIF implementation of BPTT - which does come with performance increases - Eventprop doesn't increase the complexity of neuronal model implementations which allows it do be implemented in neuromorphic hardware, unlocking the great potential of efficiency for SNNs. Not only does Eventprop scale better than eligibility propagation (e-prop) in terms of compute resources, it also doesn't use approximate gradients and thus achieves better performance. And finally, Eventprop is easier to train for complex tasks than spike-timing-dependent-plasticity (STDP) based approaches, requiring less computational resources and achieving better performance. For these reasons I have chosen to train spiking neural networks using Eventprop for speech recognition; it shows great potential for efficiently and accurately training larger models which can be used for efficient edge processing.

Mathematics of Eventprop At the core of Eventprop is the "adjoint method" from optimisation theory. The adjoint method is a highly efficient mathematical technique for calculating the sensitivity of a function's output to the functions parameters. In the case of optimising neural networks, the adjoint method would calculate the sensitivity of the network's loss function to changes in the synaptic weights of the network; this sensitivity information is used to optimise the algorithm. The adjoint method is very computationally efficient, the cost of computing the gradient is nearly independent of the number of parameters; unlike in methods such as finite differences, where the computational cost scales linearly with the number of parameters. This linear scaling would become prohibitive in large networks.

An adjoint system to the leaky integrate-and-fire (LIF) network has been derived by [36] and is shown in figure 13. The adjoint variables for the input current and membrane voltage are λ_I and λ_V respectively. They represent how sensitive the loss function is to changes to the input current or membrane voltage for a given neuron at a given time. By solving the adjoint equations backwards in time, going from $t = T$ to $t = 0$, you are left with all the values of λ_I. Summing the values of λ_I at spike events - received by neuron j and transmitted by neuron i - and multiplying this sum by $-\tau_{syn}$ you get the gradient of the loss function with respect to weight w_{ji}. This is described in equation (21).

$$\frac{d\mathcal{L}}{dw_{ji}} = -\tau_{syn} \sum_{spikes\ from\ i} (\lambda_I)_j \tag{21}$$

Figure 13 shows the maths for general loss functions; the spike time dependent loss, l_p, and the output neuron voltage dependent loss, l_V. This allows flexibility in how loss functions are defined, enabling the shaping of loss functions to suit the task. On the SHD dataset, Eventprop has achieved state-of-the-art accuracy by using a loss function based on the membrane voltage of the output neurons [35].

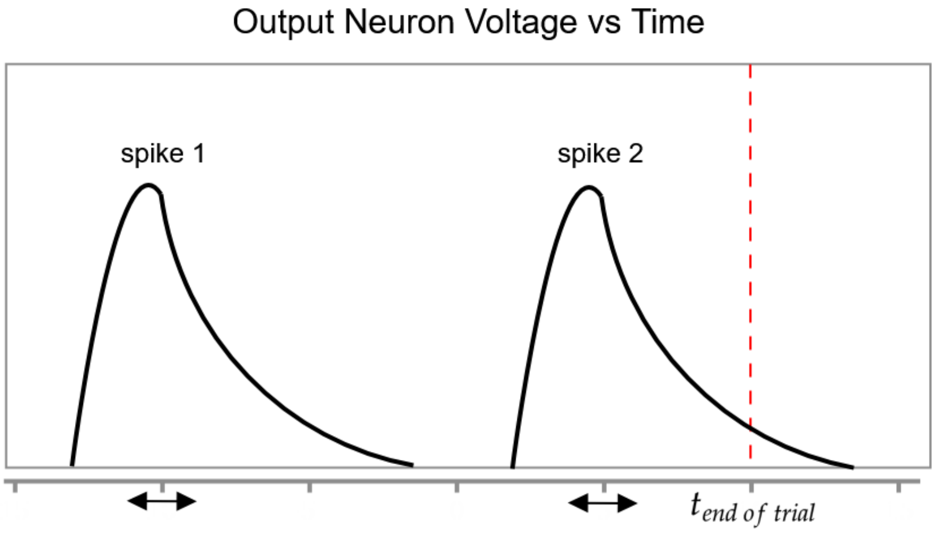


Figure 14: The membrane voltage of an output neuron which receives 2 spikes at the end of the trial. Spike 1 causes the voltage to increase and then decay to 0. Spike 2 increases the voltage, but before the voltage can decay to 0, the trial ends, so the tail end of the decay is cut off.

Initially, researchers tried using a simple loss function that integrated the voltage at an output neuron over the whole recording, $\int_0^T V(t)dt$. Each of the 20 output neurons correspond to a spoken digit, 0 - 9 in English and German. The output neuron with the highest integral is what the network infers was said in the recording. The loss function took this integral for each output neuron, and calculated the loss using log-softmax function, as in equation (22).

$$\mathcal{L}_{sum} = -\frac{1}{N_{batch}} \sum_{m=1}^{N_{batch}} log(\frac{exp(\int_0^T V_{l(m)}^m(t)dt)}{\sum_{k=1}^{N_{out}} exp(\int_0^T V_k^m(t)dt)}) \tag{22}$$

Where V_k is the voltage at neuron k, and $V_{l(m)}$ is the voltage at the correct neuron (the one the corresponds with the recording label). This loss function has shown better accuracy than alternatives. One alternative is based on the timing of the first spike - the first spike to arrive at an output neuron determines how the recording is classified. Another alternative classifies the recording based on the max membrane voltage reached in the output neurons.

While it had the best accuracy, it was found that the learning rate of this sum based loss function, $\mathcal{L}_{sum}$, was very slow. When spikes reach the output neurons, the membrane voltage of the output neuron increases, and then decays based on the membrane time constant, τ_{mem}. However, if spikes reach the output neurons near the end of the trial, the tail-end of the decaying voltage will be cut off, see figure 14. Changing the weights between neurons affects the timing of the spikes they generate; a stronger synaptic connection will increase the voltage in the post-synaptic neuron faster, and thus it will reach the threshold voltage quicker. Changing the timing of the spikes occurring near the end of the recording will cause more or less of their area to be cut-off. Therefore, their timing will affect the loss - which is the result of the output neuron membrane voltage integrated over the duration of the trial - far more than earlier spikes. This disproportionate effect that later spikes have on the loss is an arbitrary artifact of the trial duration end, and causes learning to be slower and less efficient.

To improve gradient flow, a weighted sum based loss was proposed and tested, $\mathcal{L}_{sum\_exp}$ in equation (23). This weights earlier spikes exponentially more than the later spikes according to the $e^{-t/T}$ term. This loss function ended up achieving state-of-the-art accuracy.

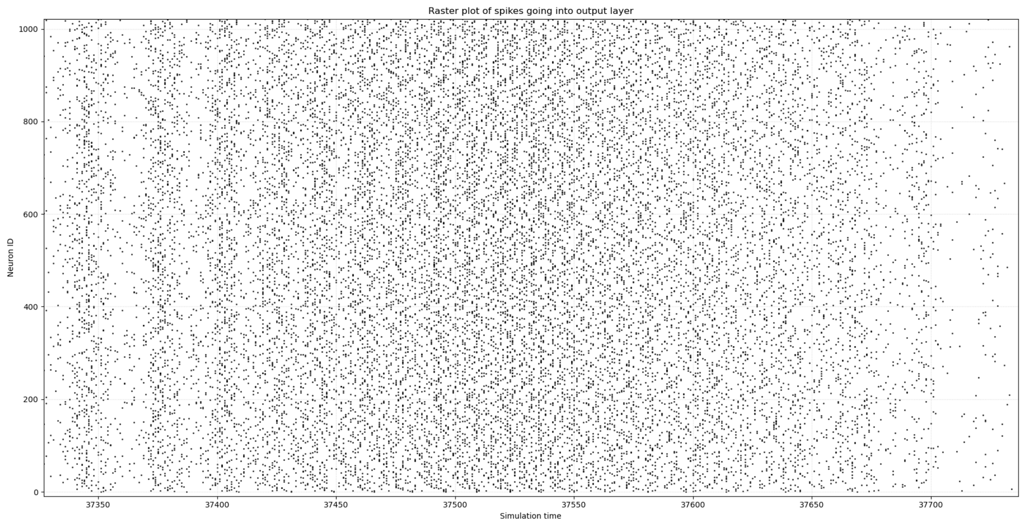


Figure 15: A raster plot of the spikes transmitted to the output neurons of the network from the hidden neurons. There is a large number if spikes very early on.

$$\mathcal{L}_{sum\_exp} = -\frac{1}{N_{batch}} \sum_{m=1}^{N_{batch}} log \frac{exp(\int_0^T e^{-t/T} V_{l(m)}^m(t)dt)}{\sum_{k=1}^{N_{out}} exp(\int_0^T e^{-t/T} V_k^m(t)dt)} \tag{23}$$

However, I hypothesise that this loss function is still sub-optimal. While it deals with the edge case of the final spikes, it now over-emphasises early spikes. In particular, I am referring the spikes that happen instantaneously to the recording beginning, since at that time there is still no way to know what the word is, therefore they are just a result of noise. Moreover, they occur before the recurrent network's rich temporal dynamics have had a chance to play-out. I predict that this causes the rate of learning of the network to be slowed because random noise is emphasised and the actual signal is attenuated. It can be seen in figure 15 that there are spikes which reach the output neurons almost instantly.

This is why I will explore deriving and implementing a novel loss function and Eventprop scheme. More on this in the methods section.

3 Methods

3.1 Dataset used

The Spiking Heidelberg digits dataset [38] is derived from the Heidelberg Digits dataset, comprising approximately 10,000 high-quality recordings of spoken digits (0 to 9) in English and German. These recordings feature a balanced group of 12 speakers (6 males and 6 females) aged between 21 and 56 years, with a mean age of 29. The audio data is converted into spike trains using the Lauscher artificial cochlea model, resulting in 700 input channels that represent different frequency bands detected by the human ear. This transformation captures the temporal dynamics of speech, making the dataset suitable for training and evaluating SNNs that process time-dependent information. The data is split between roughly 8,000 recording for the training (and validation) and 2,000 test recordings which the model will not have seen before.

The SSC dataset is a spiking version of the Google Speech Commands dataset, containing a large number of utterances across 35 word categories. These recordings are processed using the same Lauscher artificial cochlea model as in SHD, producing spike trains over 700 input channels [39]. The SSC dataset encompasses a broader vocabulary and a larger speaker base, providing a more extensive benchmark for SNNs in speech recognition tasks.

The TIMIT Acoustic-Phonetic Continuous Speech Corpus is a dataset containing a diverse group of English speakers reading phonetically-rich sentences with detailed transcription [40]. A downside of TIMIT is that it is not free, needing a license to be used. LibriSpeech ASR Corpus is a very large-scale dataset containing 1,000 hours of speech derived from audio books [41]. While it is free, the fact that it is read out loud means it may not generalise well to spontaneous conversational speech, such as might be used when engaging with a large language model through your smartphone or smart-glasses. Another prevelant speech dataset is the TIDIGITS dataset, containing recordings of digits spoken in sequence [**?**], making it more complex than the SHD dataset which contains recordings of single digits being spoken. However, a major issue with all of these alternatives to SHD and SSC is that they aren't inherently spike-based, this would cause less fair comparisons between competing training algorithms as researchers might have differing ways of converting the datasets to spiking datasets. This is why the SHD and SSC datasets are the most prevalent in the research of applying spiking neural networks to speech recognition. I will use the SHD dataset.

3.2 Software Libraries

There are several libraries which implement core Eventprop functionality. The first library I considered [42] was based on PyTorch, which is a very widely use machine learning framework that I am familiar with, with significant GPU optimisations. However, PyTorch is mainly used for ANNs, therefore it is not as optimised for SNNs; this fact could cause long training times, delaying development.

Next I evaluated a C++ based library implemented by one of the authors of the paper which first introduced Eventprop [**?**]. Being implemented in C++ - a low level programming language - means it has very little computational overhead, enabling it to run very fast. However, though the code can run fast on CPUs, it is not optimised to run on GPUs. Since neural networks calculate many things in parallel, using a CPU to run training would be very slow compared to running training on a GPU. To be optimised for GPUs, the C++ code would need to utilise a software extension like CUDA - an industry standard platform for accelerating computation using Nvidia GPUs.

The library that I chose to use for implementing and training neural networks [43] uses the GeNN - GPU enhanced Neuronal Network - framework to accelerate spiking neural networks using CUDA. The GeNN framework, developed by a professor at University of Surrey, is highly optimised for GPUs, allowing much faster development and testing of models and algorithms. Moreover, it is the library that was used to achieve state-of-the-art accuracy on the SHD dataset, and has functionality to use the previously discussed $\mathcal{L}_{sum\_exp}$ function.

Having chosen the library I will use to implement Eventprop, I installed a Linux environment in which I will develop and run code. I chose to use Linux due to the availability of software tools, and previous professional experience in developing artificial intelligence software on Linux. Next I read the documentation of the Eventprop library to understand the required packages, such as CUDA, compatible C++ compilers, and more. Having installed the necessary packages, I set up other environmental setting such as the Linux $PATH$.

3.3 Training the Network

First I implemented the model used in [35], using optimal hyperparameters. The model uses Leaky Integrate-and-Fire neuron model with exponential synapses. The model has a single hidden layer with 512 neurons. The hidden layer is recurrent, being fully connected into itself. The model is trained for 300 epochs, the best model out of 300 is saved. It is then tested against the completely new test data. I did this for the current state-of-the-art loss, $\mathcal{L}_{exp\_sum}$.

3.4 Deriving an New and Improved Loss Function and Eventprop Scheme

The current best loss function for Eventprop in the literature has a decreasing exponential weighting for the voltage at the output neurons. The mathematical representation of the exponential weighting can be observed in equation (24). This kind of weighting reduces the arbitrarily exaggerated impact that the spikes near the end of the recording

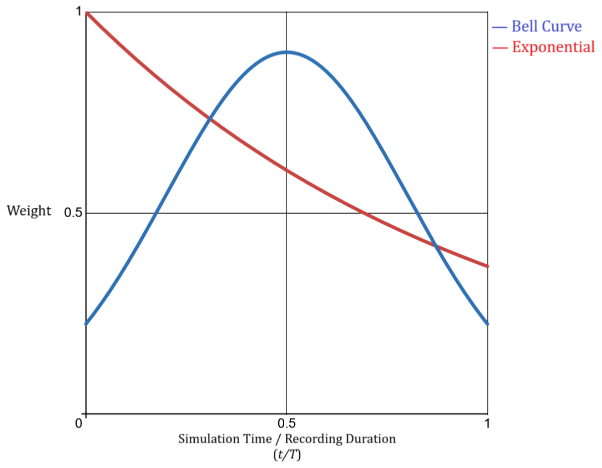


Figure 16: Plot of the weightings of the exponential function (red) and the bell curve function (blue) against -t/T from 0 to 1.

have on the loss, improving gradient descent. However, a side-effect of this function is that it emphasises the spikes which occur instantaneously. These instantaneous spikes occur while the word in question has barely begun to be said, a judgement on the word at that point would simply be a guess which would not correlate with the ground truth, hence these instantaneous spikes are simply noise. As electronic engineers know from signal processing, *noise is something that should be attenuated not amplified.*

$$V_{exp\_sum}(t) = e^{-t/T}\ V(t) \tag{24}$$

An improved loss function would take what the exponential loss function did well - fixing the issue of spikes near the end being arbitrarily emphasised - while also attenuating spikes occurring instantaneously. It should be a continuous function, so that it can be differentiated for backpropagation. I first considered the Gaussian function, famously known as the "bell curve". It will weight early and late spikes lower, while keeping the spikes in the middle weighted high.

$$V_{bell}(t) = a \exp\left(\frac{(-t/T) - b)^2}{2c^2}\right) V(t) \tag{25}$$

The exponential weighting and bell curve weightings from equations (24) and (**??**) are plotted in figure 16. The term $-t/T$ is a ratio of the current simulation time and the total simulation duration, therefore it exists between 0 (the beginning of the simulation) and 1 (the end of the simulation).

The inefficiencies of training caused by very early and very late spikes are of a different nature, therefore it would be beneficial to enable the weighting function to be asymmetrical. This would allow scaling early spikes differently to late spikes. To do this I modified the standard Gaussian function, scaling it by $-t/T$, this gave it a distinct shape, rising quickly earlier in the recording, and fading slowly near the end (figure 17).

$$V_{gausian\_mod}(t) = a - b\frac{t}{T} exp\left(\frac{-(t/T)^2}{2c^2}\right)\ V(t) \tag{26}$$

Implementing this modified Gaussian function in the log-softmax form give the new loss function, $\mathcal{L}_{sum\_gaus}$ in equation (8).

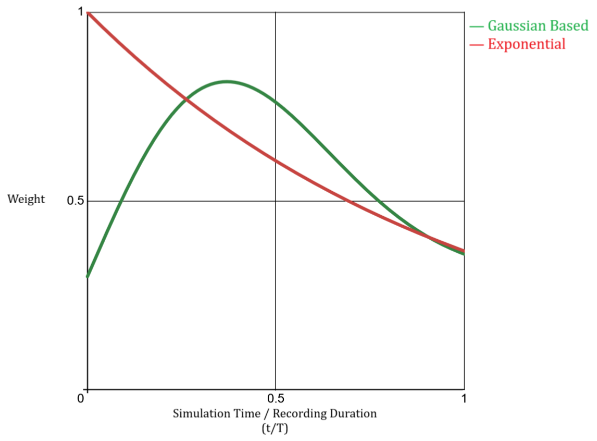


Figure 17: Plot of the weightings of the exponential function (red) and the Gaussian based function (green) against -t/T from 0 to 1.

$$\mathcal{L}_{sum\_gaus} = -\frac{1}{N_{batch}} \sum_{m=1}^{N_{batch}} log \frac{exp(\int_0^T a - b\frac{t}{T}exp\left(\frac{-(t/T)^2}{2c^2}\right) V^m_{l(m)}(t)dt)}{\sum_{k=1}^{N_{out}} exp(\int_0^T a - b\frac{t}{T}exp\left(\frac{-(t/T)^2}{2c^2}\right) V^m_k(t)dt)} \tag{27}$$

As discussed before, the gradient for each weight is backpropagates by solving for the adjoint variables - λ_V and λ_I - backwards in time. These adjoint variables represent how sensitive the loss is to changes in the state variables - V and I - at a given neuron at a given time. The dynamic properties of these adjoint variables are given by a system of ordinary differential equations (ODEs) - equations (28) and (29). Solving for $\tilde{\lambda}_{I,j}(t)$ - which is the adjoint variable for the current summed over an entire batch (equation (31)) - allows for finding the gradient of the loss function with respect to the weight w_{ji} - which connect neurons j and i - as is described in equation (30). These equations are for a general loss function $\mathcal{L}_F = F\left(\int_0^T l_V dt\right)$, where l_V is a function of output neuron voltages.

$$\tau_{\text{syn}}\tilde{\lambda}'_{I,j} = -\tilde{\lambda}_{I,j} + \tilde{\lambda}_{V,j}, \tag{28}$$

$$\tau_{\text{mem}}\tilde{\lambda}'_{V,j} = -\tilde{\lambda}_{V,j} - \sum_n \frac{\partial F}{\partial\left(\int l^n_V dt\right)} \frac{\partial l^n_V}{\partial V_j} \tag{29}$$

$$\frac{d\mathcal{L}_F}{dw_{ji}} = -\tau_{\text{syn}} \sum_{t\in t_{\text{spike}}(i)} \tilde{\lambda}_{I,j}(t) \tag{30}$$

Where:

$$\tilde{\lambda}_{I,j} := \sum_n \frac{\partial F}{\partial\left(\int l^n_V dt\right)} \lambda^n_{I,j}(t) \tag{31}$$

and

$$\tilde{\lambda}_{V,j} := \sum_n \frac{\partial F}{\partial\left(\int l^n_V dt\right)} \lambda^n_{V,j} \tag{32}$$

To integrate the new, Gaussian based loss function, $\mathcal{L}_{sum\_gaus}$, into the Eventprop scheme we formulate the gradient of the loss as:

$$\begin{aligned}\frac{d\mathcal{L}}{dw_{ji}} &= \frac{1}{N_{\text{batch}}}\sum_{m=1}^{N_{\text{batch}}}\frac{d}{dw_{ji}}\left(-\log\frac{\exp\left(\int_0^T a-b\frac{t}{T}exp\left(\frac{-(t/T)^2}{2c^2}\right)V_{l(m)}^m(t)dt\right)}{\sum_{k=1}^{N_{\text{out}}}\exp\left(\int_0^T a-b\frac{t}{T}exp\left(\frac{-(t/T)^2}{2c^2}\right)V_k^m(t)dt\right)}\right)\\ &= \frac{1}{N_{\text{batch}}}\sum_{m=1}^{N_{\text{batch}}}\frac{dF^m}{dw_{ji}},\end{aligned} \tag{33}$$

Where F^m is the loss of a single trial, m, within a batch of trials, N_{batch}. Written as a function of the integral of the voltage, F^m becomes:

$$F^m\left(\int_0^T a-b\frac{t}{T}exp\left(\frac{-(t/T)^2}{2c^2}\right)V(t)dt\right) = -\log\frac{\exp\left(\int_0^T a-b\frac{t}{T}exp\left(\frac{-(t/T)^2}{2c^2}\right)V_{l(m)}^m(t)dt\right)}{\sum_{k=1}^{N_{\text{out}}}\exp\left(\int_0^T a-b\frac{t}{T}exp\left(\frac{-(t/T)^2}{2c^2}\right)V_k^m(t)dt\right)} \tag{34}$$

So that we can use this loss function within the Eventprop adjoint ODE (equations (28) and (29)), we need to find the derivative of F^m. We first simplify the input to F^m as:

$$X_k = \int_0^T a-b\frac{t}{T}exp\left(\frac{-(t/T)^2}{2c^2}\right)\ V_k^m(t)\ dt \tag{35}$$

Where the integrated output neuron voltage score is X_k. Therefore, $X_{l(m)}$ would specifically be the score of the correct output neuron, $l(m)$ for trial m. Now we can rewrite F^m as:

$$F^m = -\log\frac{\exp\left(X_{l(m)}\right)}{\sum_{k=1}^{N_{\text{out}}}\exp\left(X_k\right)} \tag{36}$$

Using the property of logarithms $log(a/b) = log(a) - log(b)$, we get:

$$F^m = -\log\left(\exp\left(X_{l(m)}\right)\right)\ +\log\left(exp\sum_{k=1}^{N_{\text{out}}}\exp\left(X_k\right)\right) \tag{37}$$

Simplifying:

$$F^m = \log\left(exp\sum_{k=1}^{N_{\text{out}}}\exp\left(X_k\right)\right) - X_{l(m)} \tag{38}$$

Now we differentiate with respect to X_n, where n is an output neuron.

$$\frac{\partial F^m}{\partial X_n} = \frac{\partial}{\partial X_n}\left(\log\left(\sum_{k=1}^{N_{\text{out}}}\exp(X_k)\right) - X_{l(m)}\right) \tag{39}$$

This can be split into two differentials:

$$\begin{aligned}&1.\quad \frac{\partial}{\partial X_n}\left(\log\left(\sum_{k=1}^{N_{\text{out}}}\exp(X_k)\right)\right)\\ &2.\quad \frac{\partial}{\partial X_n}(-X_{l(m)})\end{aligned} \tag{40}$$

To differentiate the first part, let $S = \sum_{k=1}^{N_{\text{out}}} \exp(X_k)$. So the first part becomes $\frac{\partial}{\partial X_n}(\log S)$, and using chain rule:

$$\frac{\partial(\log S)}{\partial X_n} = \frac{1}{S} \cdot \frac{\partial S}{\partial X_n} = \frac{1}{S} \cdot \frac{\partial}{\partial X_n} \left(\sum_{k=1}^{N_{\text{out}}} \exp(X_k) \right) \tag{41}$$

The partial derivative of the sum with respect to X_n would be non-zero only when $k = n$. So treating the values where $k \neq n$ as constants we simplify $\frac{\partial S}{\partial X_n}$ as:

$$\frac{\partial S}{\partial X_n} = \frac{\partial}{\partial X_n}(\exp(X_n)) = \exp(X_n) \tag{42}$$

Therefore the first part is:

$$\frac{1}{\sum_{k=1}^{N_{\text{out}}} \exp(X_k)} \cdot \exp(X_n) = \frac{\exp(X_n)}{\sum_{k=1}^{N_{\text{out}}} \exp(X_k)} \tag{43}$$

The second part, $\frac{\partial}{\partial X_n}(-X_{l(m)})$, depends on weather n is the same as $l(m)$.

- If $n = l(m)$, then we are differentiating $-X_n$ with respect to X_n, which is -1.
- If $n \neq l(m)$, then $X_l m$ is treated as a constant with respect to X_n, so the derivative is 0.

This can be written concisely using the Kronecker delta, $\delta_{n,l(m)}$, which is 1 if $n = l(m)$ and 0 if $n \neq l(m)$. So, the second part is:

$$\frac{\partial}{\partial X_n}(-\mathbf{X}_{l(m)}) = -\delta_{n,l(m)}. \tag{44}$$

Combining these parts we get:

$$\frac{\partial F^m}{\partial X_n} = \frac{\exp(X_n)}{\sum_{k=1}^{N_{\text{out}}} \exp(X_k)} - \delta_{n,l(m)} = -\delta_{n,l(m)} + \frac{\exp(X_n)}{\sum_{k=1}^{N_{\text{out}}} \exp(X_k)} \tag{45}$$

Substituting back the original integral notation:

$$\frac{\partial F^m}{\partial \left(\int_0^T a - b\frac{t}{T} exp\left(\frac{-(t/T)^2}{2c^2}\right) V_n^m dt \right)} = -\delta_{n,l(m)} + \frac{\exp\left(\int_0^T a - b\frac{t}{T} exp\left(\frac{-(t/T)^2}{2c^2}\right) V_n^m dt \right)}{\sum_{k=1}^{N_{\text{out}}} \exp\left(\int_0^T a - b\frac{t}{T} exp\left(\frac{-(t/T)^2}{2c^2}\right) V_k^m dt \right)} \tag{46}$$

Therefore, using equations 28), 29), and 30), we can formulate the new Eventprop scheme of the Gaussian based loss function:

$$\tau_{\text{syn}} \tilde{\lambda}_{I,j}^{m'} = -\tilde{\lambda}_{I,j}^m + \tilde{\lambda}_{V,j}^m \tag{47}$$

$$\tau_{\text{mem}} \tilde{\lambda}_{V,j}^{m'} = -\tilde{\lambda}_{V,j}^m + \delta_{j,l(m)} - \frac{\exp\left(\int a - b\frac{t}{T} exp\left(\frac{-(t/T)^2}{2c^2}\right) V_j^m dt \right)}{\sum_{k=1}^{N_{\text{class}}} \exp\left(\int_0^T a - b\frac{t}{T} exp\left(\frac{-(t/T)^2}{2c^2}\right) V_k^m dt \right)} \tag{48}$$

$$\frac{dF^m}{dw_{ji}} = -\tau_{\text{syn}} \sum_{\{t_{\text{spike}}(i)\}} \tilde{\lambda}_{I,j}^m(t_{\text{spike}}) \tag{49}$$

This defines how the adjoint variables, λ_V and λ_I, dynamically evolve, and how the gradient of the loss function, F^m, with respect to a weight, w_{ji}, is calculated.

3.5 Implementing the Novel Eventprop Scheme in Software

The derived mathematics needed to be actualised in code. I examined the Eventprop library to find where the loss function and the dynamics of the output neurons are defined. I used C++ to implement a new function defining the behaviour of the network for the Gaussian based loss. Using C++ enables greater performance over higher-level languages like Python [44].

The time variable `local_t` is a fraction from 0 to 1 of how far into the recording the simulation is, seen before as t/T. The section marked "Backward Pass" steps backwards from the end of the recording, solving the differential equations for the adjoint variables using the simple Euler method. The "Forward Pass" section again uses the Simple Euler method to solve for the voltage at the output neuron over small time steps. For readability the Gaussian weighting function is split into smaller chunks, lines 17 to 28 and 51 to 60 in figure 18)

To test this Eventprop scheme, a model is trained for 300 epochs, and the best version of the model is saved. Then the saved model is tested against completely new data. I also ran it 5 times for a reduced amount of epochs, 100, along with the old loss function. This is to compare the learning rate averaged out over 5 runs for statistical significance.

3.6 Profiling the Power Usage of GPU vs CPU

To compare the performance of a machine learning workflow on a GPU versus a CPU, I conducted a series of tests. The CPU (AMD Ryzen 9 5950X) run was executed with a batch size of 1. For the GPU (Nvidia RTX 4090), training was performed using seven different batch sizes (1, 2, 4, 8, 16, 32, 64) to evaluate how performance scaled with increased parallelism. During each run, I recorded the power consumption and measured the duration of each epoch. By averaging the power usage and epoch times, I calculated the energy consumed per epoch as the product of average power and average epoch duration. This analysis aimed not only to compare speed, but also to determine whether the GPU offers greater energy efficiency.

4 Results

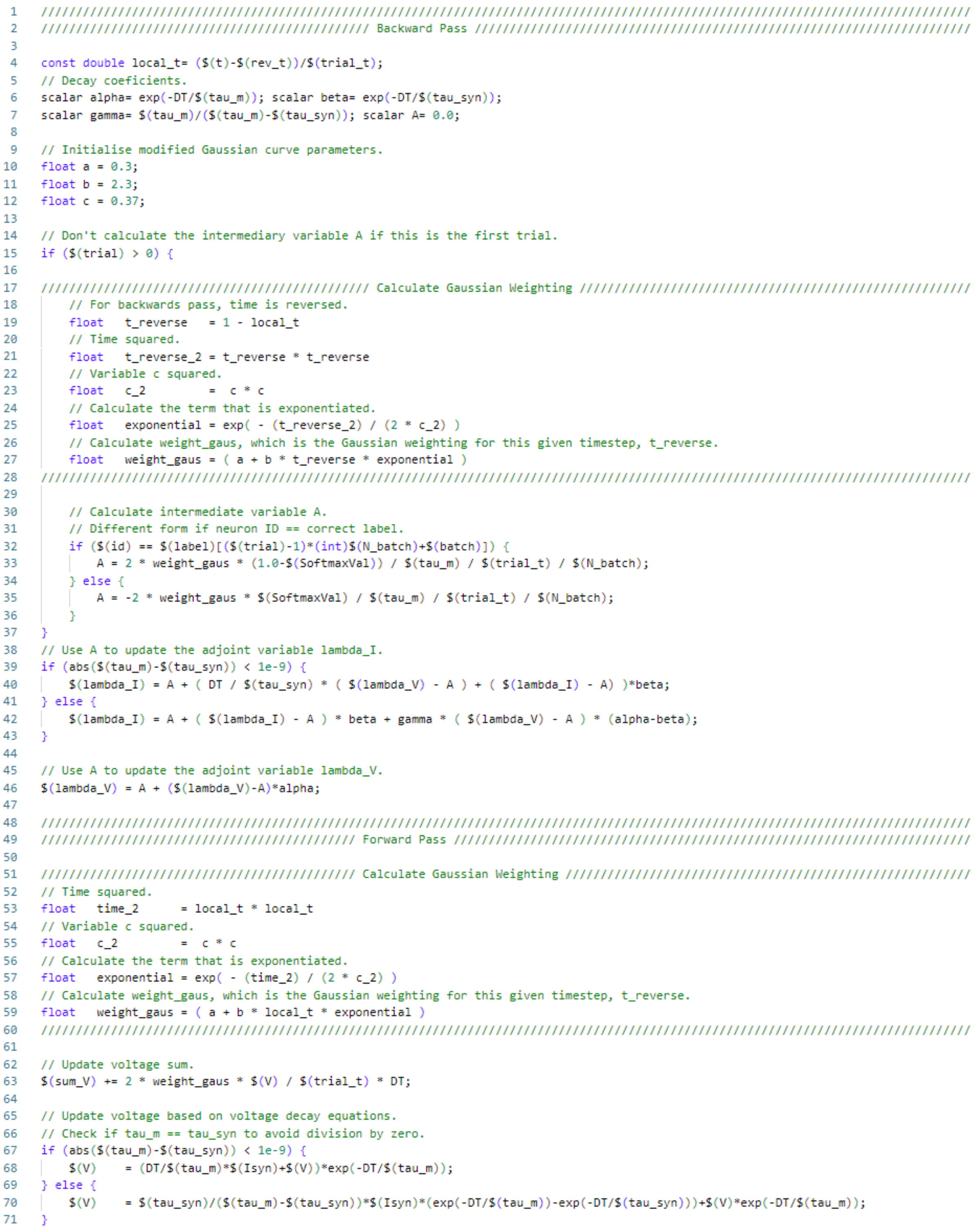
4.1 Reproducing State-Of-The-Art Accuracy

Figrue shows the evaluation correct rate for training of the network running for 300 epochs. The maximum evaluation correct of 91.5% is reached during the 281st epoch. This value matches the value reported in literature [35]. Then I tested the model which achieved the best evaluation during training on the test data, it achieved 93.3%. This again matched the value reported in literature [35].

4.2 New Loss Function Improves Training

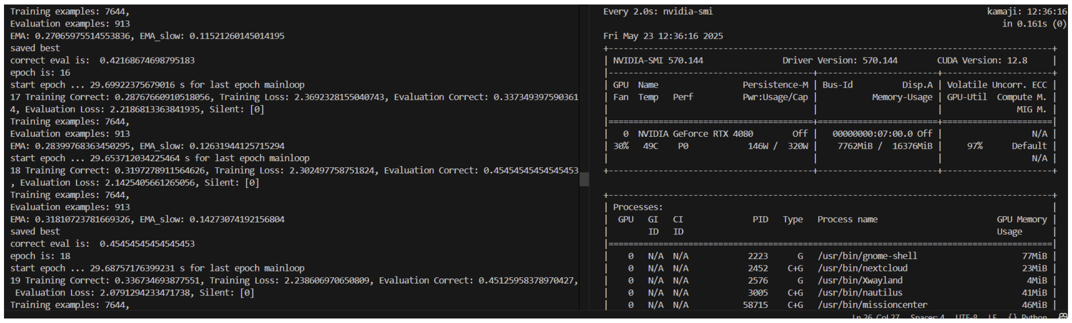
Figure 21) shows the training of the new loss function, $\mathcal{L}_{sum\_gaus}$. It too reaches a maximum value of 91.5% during the 300 epochs of training. During the test it reached a slightly lower 92.9%.

Figure 22) shows the graph of mean evaluation correct for the first 100 epochs for the Gaussian based loss function end the exponential based loss function, ran 5 times each. It can be seen that the Gaussian learning rate is much faster in the beginning.



```
/////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////// Backward Pass /////////////////////////////////////////////////////////////////////

const double local_t= ($(t)-$(rev_t))/$(trial_t);
// Decay coeficients.
scalar alpha= exp(-DT/$(tau_m)); scalar beta= exp(-DT/$(tau_syn));
scalar gamma= $(tau_m)/($(tau_m)-$(tau_syn)); scalar A= 0.0;

// Initialise modified Gaussian curve parameters.
float a = 0.3;
float b = 2.3;
float c = 0.37;

// Don't calculate the intermediary variable A if this is the first trial.
if ($(trial) > 0) {

//////////////////////////////////////////////// Calculate Gaussian Weighting /////////////////////////////////////////////////////
    // For backwards pass, time is reversed.
    float   t_reverse   = 1 - local_t
    // Time squared.
    float   t_reverse_2 = t_reverse * t_reverse
    // Variable c squared.
    float   c_2         =  c * c
    // Calculate the term that is exponentiated.
    float   exponential = exp( - (t_reverse_2) / (2 * c_2) )
    // Calculate weight_gaus, which is the Gaussian weighting for this given timestep, t_reverse.
    float   weight_gaus = ( a + b * t_reverse * exponential )
/////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

    // Calculate intermediate variable A.
    // Different form if neuron ID == correct label.
    if ($(id) == $(label)[($(trial)-1)*(int)$(N_batch)+$(batch)]) {
        A = 2 * weight_gaus * (1.0-$(SoftmaxVal)) / $(tau_m) / $(trial_t) / $(N_batch);
    } else {
        A = -2 * weight_gaus * $(SoftmaxVal) / $(tau_m) / $(trial_t) / $(N_batch);
    }
}
// Use A to update the adjoint variable lambda_I.
if (abs($(tau_m)-$(tau_syn)) < 1e-9) {
    $(lambda_I) = A + ( DT / $(tau_syn) * ( $(lambda_V) - A ) + ( $(lambda_I) - A) )*beta;
} else {
    $(lambda_I) = A + ( $(lambda_I) - A ) * beta + gamma * ( $(lambda_V) - A ) * (alpha-beta);
}

// Use A to update the adjoint variable lambda_V.
$(lambda_V) = A + ($(lambda_V)-A)*alpha;

/////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
/////////////////////////////////////////////// Forward Pass ///////////////////////////////////////////////////////////////////////

/////////////////////////////////////////////// Calculate Gaussian Weighting ///////////////////////////////////////////////////////
// Time squared.
float   time_2      = local_t * local_t
// Variable c squared.
float   c_2         =  c * c
// Calculate the term that is exponentiated.
float   exponential = exp( - (time_2) / (2 * c_2) )
// Calculate weight_gaus, which is the Gaussian weighting for this given timestep, t_reverse.
float   weight_gaus = ( a + b * local_t * exponential )
/////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

// Update voltage sum.
$(sum_V) += 2 * weight_gaus * $(V) / $(trial_t) * DT;

// Update voltage based on voltage decay equations.
// Check if tau_m == tau_syn to avoid division by zero.
if (abs($(tau_m)-$(tau_syn)) < 1e-9) {
    $(V)     = (DT/$(tau_m)*$(Isyn)+$(V))*exp(-DT/$(tau_m));
} else {
    $(V)     = $(tau_syn)/($(tau_m)-$(tau_syn))*$(Isyn)*(exp(-DT/$(tau_m))-exp(-DT/$(tau_syn)))+$(V)*exp(-DT/$(tau_m));
}
```

Figure 18: Code to implement the mathematics of the Eventprop scheme for a Gaussian based loss.



```
Training examples: 7644,
Evaluation examples: 913
EMA: 0.27065975514553836, EMA_slow: 0.11521260145014195
saved best
correct eval is:  0.42168674698795183
epoch is: 16
start epoch ... 29.69922375679016 s for last epoch mainloop
17 Training Correct: 0.28767660910518056, Training Loss: 2.3692328155040743, Evaluation Correct: 0.337349397590361
4, Evaluation Loss: 2.2186813363841935, Silent: [0]
Training examples: 7644,
Evaluation examples: 913
EMA: 0.28399768363450295, EMA_slow: 0.12631944125715294
start epoch ... 29.653712034225464 s for last epoch mainloop
18 Training Correct: 0.3197278911564626, Training Loss: 2.302497758751824, Evaluation Correct: 0.45454545454545453
, Evaluation Loss: 2.1425405661265056, Silent: [0]
Training examples: 7644,
Evaluation examples: 913
EMA: 0.31810723781669326, EMA_slow: 0.14273074192156804
saved best
correct eval is:  0.45454545454545453
epoch is: 18
start epoch ... 29.68757176399231 s for last epoch mainloop
19 Training Correct: 0.336734693877551, Training Loss: 2.238606970650809, Evaluation Correct: 0.45125958378970427,
 Evaluation Loss: 2.0791294233471738, Silent: [0]
Training examples: 7644,
```

```
Every 2.0s: nvidia-smi                                                    kamaji: 12:36:16
                                                                            in 0.161s (0)
Fri May 23 12:36:16 2025
+-----------------------------------------------------------------------------------------+
| NVIDIA-SMI 570.144                Driver Version: 570.144        CUDA Version: 12.8     |
|-----------------------------------------+------------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id          Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util  Compute M. |
|                                         |                        |               MIG M. |
|=========================================+========================+======================|
|   0  NVIDIA GeForce RTX 4080        Off |   00000000:07:00.0 Off |                  N/A |
| 30%   49C    P0            146W /  320W |    7762MiB /  16376MiB |     97%      Default |
|                                         |                        |                  N/A |
+-----------------------------------------+------------------------+----------------------+

+-----------------------------------------------------------------------------------------+
| Processes:                                                                              |
|  GPU   GI   CI              PID   Type   Process name                        GPU Memory |
|        ID   ID                                                               Usage      |
|=========================================================================================|
|    0   N/A  N/A            2223      G   /usr/bin/gnome-shell                     77MiB |
|    0   N/A  N/A            2452    C+G   /usr/bin/nextcloud                       23MiB |
|    0   N/A  N/A            2576      G   /usr/bin/Xwayland                         4MiB |
|    0   N/A  N/A            3005    C+G   /usr/bin/nautilus                        41MiB |
|    0   N/A  N/A           58715    C+G   /usr/bin/missioncenter                   46MiB |
```

Figure 19: A view of the training in progress on the left, and the GPU being profiled on the right.

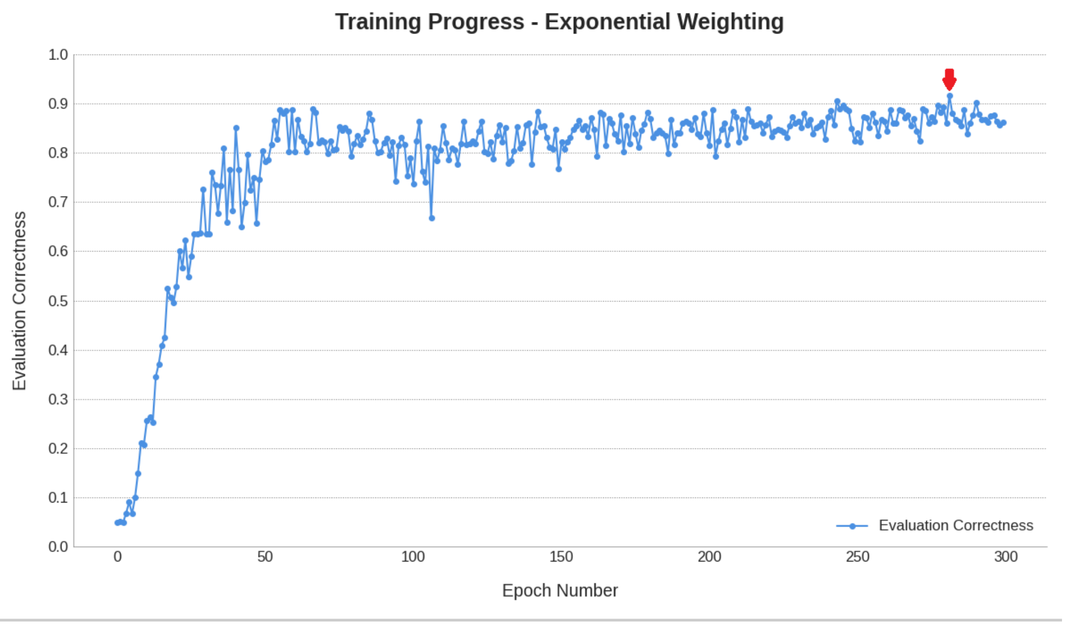


Figure 20: Evaluation correctness vs epoch number, exponential weighting loss function. Red arrow shows maximum value reached.

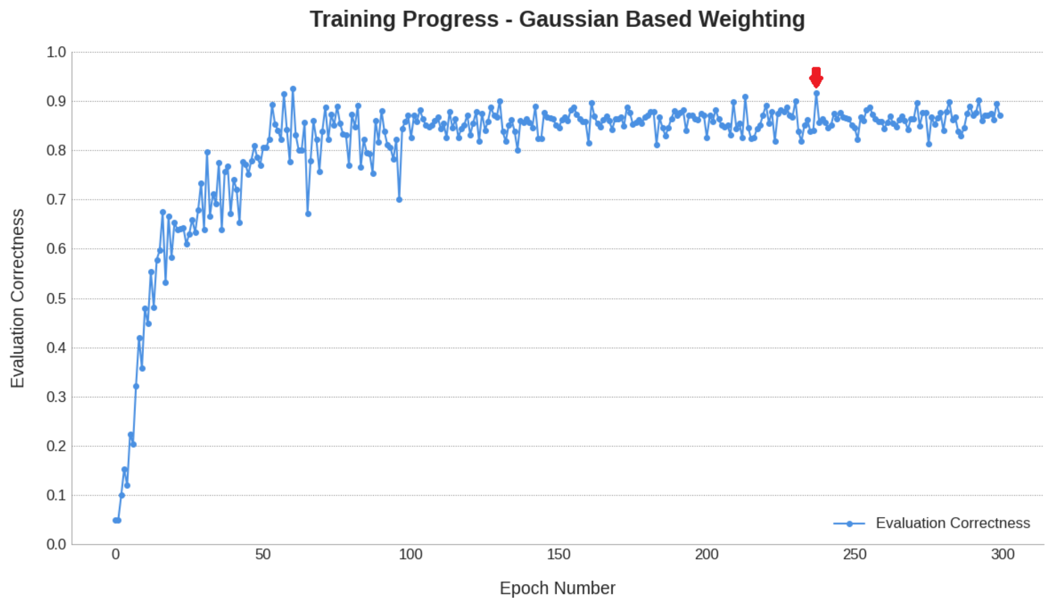


Figure 21: Evaluation correctness vs epoch number, Gaussian weighting loss function. Red arrow shows maximum value reached.

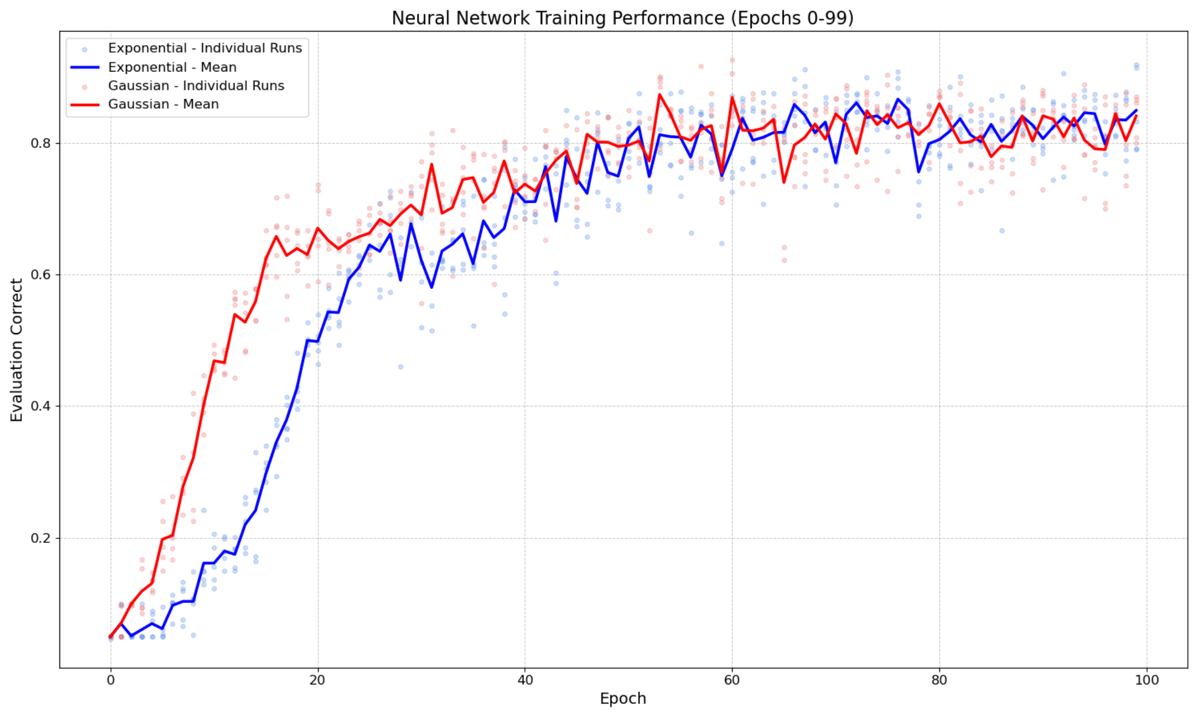


Figure 22: Plot of the mean correct evaluation accuracy per epoch of the Gaussian based sum loss (red) and the exponential sum loss (blue), over 5 runs each.

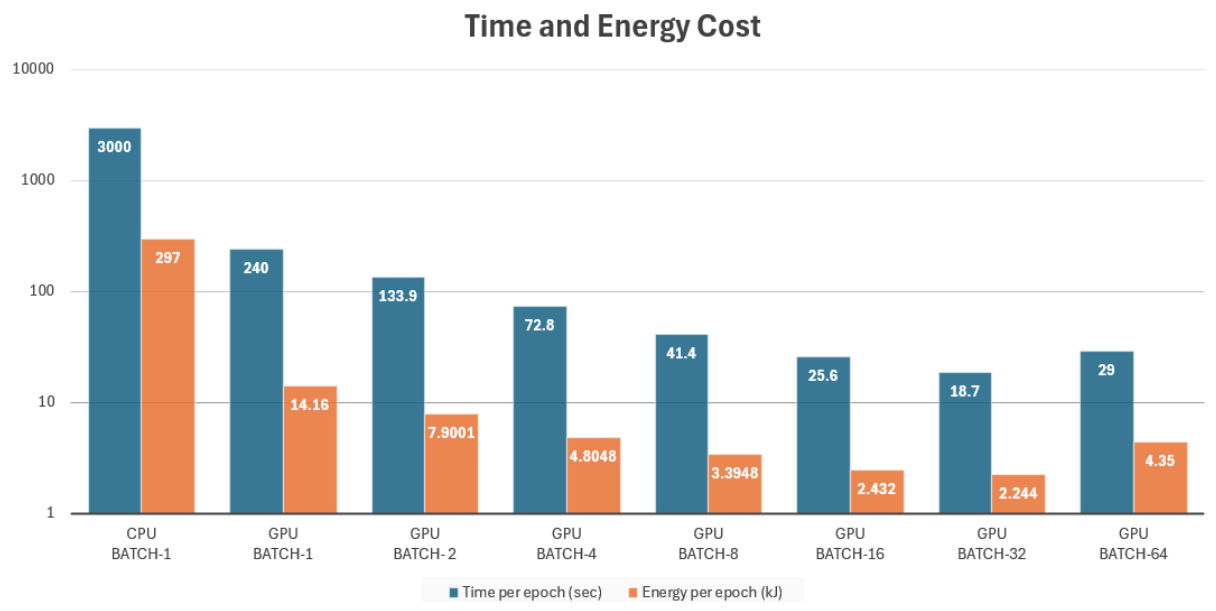


Figure 23: Bar graph of the time cost and energy cost for CPU and GPU using different batch sizes.

4.3 Power Usage of GPU and CPU

The figure 23) presents the time and energy consumption per epoch for each setup on a logarithmic scale. The CPU, processing a single batch, was over 10 times slower than the GPU. While the CPU consumed less power, its substantially longer runtime led to greater overall energy usage. Increasing the batch size improved GPU speed up to a batch size of 32, beyond which performance declined.

5 Discussion

This project addressed the challenge of training Spiking Neural Networks (SNNs) for speech recognition, with a focus on the EventProp algorithm - a promising approach for learning in temporal spiking domains. In the initial phase, I successfully reproduced state-of-the-art performance on the Spiking Heidelberg Digits (SHD) dataset, achieving a test accuracy of 93.3% using the established exponentially weighted loss function (L_{sum_exp}). This result serves to validate both the experimental framework and the implementation, establishing a reliable benchmark for assessing subsequent methodological contributions.

The central focus of this investigation was to address a hypothesized sub-optimality in the $\mathcal{L}_{sum\_exp}$ loss function. Specifically, it was posited that the exponential weighting scheme disproportionately amplified noise arising from early, potentially uninformative spikes at the beginning of input sequences. To mitigate this effect, a novel Gaussian-based loss function, L_{sum_gaus}, was mathematically derived and implemented. This formulation attenuates the influence of both early and late spikes, thereby concentrating the learning signal on temporally central segments of the input, which are assumed to be more informative for speech recognition.

The application of the $\mathcal{L}_{sum\_gaus}$ loss function resulted in a peak test accuracy of 92.8%. Although this is marginally lower than the 93.3% achieved using $\mathcal{L}_{sum\_exp}$, it remains within the error bounds of the state-of-the-art benchmark. Notably, $\mathcal{L}_{sum\_gaus}$ exhibited a substantially faster initial learning rate, as shown in Figure 22. This rapid early convergence suggests that the Gaussian weighting effectively directed the network toward salient features more efficiently—supporting the hypothesis that it reduces the influence of early, noise-driven transients. In practical terms, the improved learning rate offers the potential for shorter training durations and decreased computational cost, providing a meaningful advantage in iterative model development and hyperparameter tuning.

Though slight, the deficit in the final accuracy warrants careful consideration. The factors, including uncertainties and incompleteness in the current work, which could contribute to this observation:

- Hyperparameter optimisation. The hyperparameters, including those defining the network learning rate, and neuron dynamics, were systematically optimised for the $\mathcal{L}_{sum\_gaus}$ loss function [35]. For the training of $\mathcal{L}_{sum\_gaus}$ this established hyperparameter set was adopted without modification. Therefore the full potential of $\mathcal{L}_{sum\_gaus}$ may only be realised after a dedicated and systematic hyperparameter optimisation. This was beyond the scope of this project, but offers routes for future research.
- Inherent stochasticity. Neural network training is subject to inherent randomness. While the learning rate comparison was averaged over five runs, the final SOTA comparison was based on a single extended run for each loss function. The 0.5% difference in peak accuracy is relatively small and could, to some extent, be influenced by this stochasticity. More extensive runs would be needed to establish the statistical significance of this minor difference with greater confidence.

The effect of these uncertainties on project outcomes is that while the novel loss function has demonstrated a clear advantage in learning speed, its capacity to achieve or surpass existing SOTA accuracy on the SHD dataset remains an incompletely answered question, primarily due to the limited hyperparameter exploration

Another avenue for exploration is the application of Eventprop and these loss functions to significantly larger SNN models. The current study utilized a model with a single hidden layer of 512 neurons. Eventprop's inherent computational and memory efficiency is documented to allow for scaling to more complex tasks and larger models, a domain that holds potential for further accuracy improvements but has not yet been fully investigated. Testing how both the $\mathcal{L}_{sum\_exp}$ and the novel $\mathcal{L}_{sum\_gaus}$ perform with deeper or wider architectures, or those incorporating more complex connectivity patterns, is a critical next step.

Beyond the core investigation into loss functions, the power usage profiling conducted in Section 4.3 (Figure 23) offers practical implications for SNN research and development. The benchmarks comparing CPU (AMD Ryzen 9 5950X) and GPU (Nvidia RTX 4090) performance revealed not only that the GPU was vastly faster but also pinpointed an energy "sweet spot" for the specific GPU used. For the Nvidia RTX 4090, a batch size of 32 yielded the lowest energy consumption per epoch (2.02 kJ), demonstrating that optimal batch sizing is crucial for maximizing energy efficiency. Even though GPUs are often considered power-hungry in terms of their instantaneous draw, these results show they are, in fact, the most energy-efficient means of training these neural networks when considering the total energy required to complete a given amount of training. This superior efficiency and speed dramatically reduce the turnaround time for experiments, making iterative research, complex model development, and extensive hyperparameter searches far more feasible.

Despite the uncertainties, the engineering achievement of this project is significant. It successfully progressed from a theoretical critique of an existing SOTA method to the mathematical derivation (Section 3.4), software implementation (Section 3.5, Figure 18), and empirical validation of a novel alternative. The demonstration that thoughtful loss function design can directly and positively influence SNN training dynamics is a valuable contribution. This reinforces the principle that model performance is an interplay of architecture, optimisation, and the precise formulation of the learning objective. The power profiling results (Section 4.3, Figure 23) also provided a practical perspective, underscoring the substantial energy and time efficiencies achievable with GPU acceleration in SNN training workflows.

6 Conclusion

References

[1] "The state of AI in 2025: Global survey | McKinsey," https://www.mckinsey.com/capabilities/quantumblack/our-insights/the-state-of-ai.

[2] B. Kindig, "AI Power Consumption: Rapidly Becoming Mission-Critical," https://www.forbes.com/sites/bethkindig/2024/06/20/ai-power-consumption-rapidly-becoming-mission-critical/.

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

[4] W. Maass and H. Markram, “On the computational power of circuits of spiking neurons,” *Journal of Computer and System Sciences*, vol. 69, no. 4, pp. 593–616, Dec. 2004.

[5] P. A. Merolla, J. V. Arthur, R. Alvarez-Icaza, A. S. Cassidy, J. Sawada, F. Akopyan, B. L. Jackson, N. Imam, C. Guo, Y. Nakamura, B. Brezzo, I. Vo, S. K. Esser, R. Appuswamy, B. Taba, A. Amir, M. D. Flickner, W. P. Risk, R. Manohar, and D. S. Modha, “A million spiking-neuron integrated circuit with a scalable communication network and interface,” *Science*, vol. 345, no. 6197, pp. 668–673, 2014.

[6] R. Waters, “OpenAI’s mind-boggling growth masks challenges,” *Financial Times*, Apr. 2025.

[7] T. Bradshaw and H. Murphy, “Meta’s investment in VR and smart glasses on track to top $100bn,” *Financial Times*, Feb. 2025.

[8] M. Acton, “Apple delays iPhone AI features as it stumbles in race with rivals,” *Financial Times*, Mar. 2025.

[9] K. Wiggers, “Rabbit is building an AI model that understands how software works,” Oct. 2023.

[10] E. Izhikevich, “Simple model of spiking neurons,” *IEEE Transactions on Neural Networks*, vol. 14, no. 6, pp. 1569–1572, Nov. 2003.

[11] ——, “Which model to use for cortical spiking neurons?” *IEEE Transactions on Neural Networks*, vol. 15, no. 5, pp. 1063–1070, Sep. 2004.

[12] L. Deng and X. Li, “Machine Learning Paradigms for Speech Recognition: An Overview,” *IEEE Transactions on Audio, Speech, and Language Processing*, vol. 21, no. 5, pp. 1060–1089, May 2013.

[13] G. Hinton, L. Deng, D. Yu, G. Dahl, A.-r. Mohamed, N. Jaitly, V. Vanhoucke, P. Nguyen, T. Sainath, and B. Kingsbury, “Deep Neural Networks for Acoustic Modeling in Speech Recognition,” 2012.

[14] 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, Mar. 2020.

[15] A. Bittar and P. N. Garner, “A surrogate gradient spiking baseline for speech command recognition,” *Frontiers in Neuroscience*, vol. 16, p. 865897, Aug. 2022.

[16] G. Bellec, F. Scherr, E. Hajek, D. Salaj, R. Legenstein, and W. Maass, “Biologically inspired alternatives to backpropagation through time for learning in recurrent neural nets,” *arXiv.org*, Feb. 2019.

[17] E. O. Neftci, M. Hesham, and Z. Friedemann, “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, Nov. 2019.

[18] C. Zhou, H. Zhang, L. Yu, Y. Ye, Z. Zhou, L. Huang, Z. Ma, X. Fan, H. Zhou, and Y. Tian, “Direct training high-performance deep spiking neural networks: A review of theories and methods,” *Frontiers in Neuroscience*, vol. 18, Jul. 2024.

[19] S. Y. Arnaud Yarga and S. U. N. Wood, “Accelerating spiking neural networks with parallelizable leaky integrate-and-fire neurons*,” *Neuromorphic Computing and Engineering*, vol. 5, no. 014012, Mar. 2025.

[20] S. Y. A. Yarga and S. U. N. Wood, “Accelerating SNN Training with Stochastic Parallelizable Spiking Neurons,” in *2023 International Joint Conference on Neural Networks (IJCNN)*, Jun. 2023, pp. 1–8.

[21] R. Koopman, A. Yousefzadeh, M. Shahsavari, G. Tang, and M. Sifalakis, “Overcoming the Limitations of Layer Synchronization in Spiking Neural Networks,” Aug. 2024.

[22] Y. Zhong, R. Zhao, C. Wang, Q. Guo, J. Zhang, Z. Lu, and L. Leng, “SPikE-SSM: A Sparse, Precise, and Efficient Spiking State Space Model for Long Sequences Learning,” Oct. 2024.

[23] G. Bellec, F. Scherr, E. Hajek, D. Salaj, A. Subramoney, R. Legenstein, and W. Maass, “Eligibility traces provide a data-inspired alternative to backpropagation through time,” in *Real Neurons {\&} Hidden Units: Future Directions at the Intersection of Neuroscience and Artificial Intelligence @ NeurIPS 2019*, Oct. 2019.

[24] ——, "Eligibility traces provide a data-inspired alternative to backpropagation through time," in *Real Neurons {\&} Hidden Units: Future Directions at the Intersection of Neuroscience and Artificial Intelligence @ NeurIPS 2019*, Oct. 2019.

[25] A. Rostami, B. Vogginger, Y. Yan, and C. G. Mayr, "E-prop on SpiNNaker 2: Exploring online learning in spiking RNNs on neuromorphic hardware," *Frontiers in Neuroscience*, vol. 16, p. 1018006, Nov. 2022.

[26] ——, "E-prop on SpiNNaker 2: Exploring online learning in spiking RNNs on neuromorphic hardware," *Frontiers in Neuroscience*, vol. 16, Nov. 2022.

[27] W. Chen, L. Song, S. Wang, Z. Zhang, G. Wang, G. Hu, and S. Gao, "Essential Characteristics of Memristors for Neuromorphic Computing," *Advanced Electronic Materials*, vol. 9, no. 2, p. 2200833, 2023.

[28] Y. Li, K. Su, H. Chen, X. Zou, C. Wang, H. Man, K. Liu, X. Xi, and T. Li, "Research Progress of Neural Synapses Based on Memristors," *Electronics*, vol. 12, no. 15, p. 3298, Jan. 2023.

[29] C. Weilenmann, A. N. Ziogas, T. Zellweger, K. Portner, M. Mladenović, M. Kaniselvan, T. Moraitis, M. Luisier, and A. Emboras, "Single neuromorphic memristor closely emulates multiple synaptic mechanisms for energy efficient neural networks," *Nature Communications*, vol. 15, no. 1, p. 6898, Aug. 2024.

[30] D. Vlasov, Y. Davydov, A. Serenko, R. Rybka, and A. Sboev, "Spoken Digits Classification Based on Spiking Neural Networks with Memristor-Based STDP," in *2022 International Conference on Computational Science and Computational Intelligence (CSCI)*, Dec. 2022, pp. 330–335.

[31] A. Sboev, M. Balykov, D. Kunitsyn, and A. Serenko, "Spoken Digits Classification Using a Spiking Neural Network with Fixed Synaptic Weights," in *Biologically Inspired Cognitive Architectures 2023*, A. V. Samsonovich and T. Liu, Eds. Cham: Springer Nature Switzerland, 2024, pp. 767–774.

[32] Y. Guo, X. Huang, and Z. Ma, "Direct learning-based deep spiking neural networks: A review," *Frontiers in Neuroscience*, vol. 17, p. 1209795, Jun. 2023.

[33] C. Lee, P. Panda, G. Srinivasan, and K. Roy, "Training Deep Spiking Convolutional Neural Networks With STDP-Based Unsupervised Pre-training Followed by Supervised Fine-Tuning," *Frontiers in Neuroscience*, vol. 12, Aug. 2018.

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

[35] T. Nowotny, J. P. Turner, and J. C. Knight, "Loss shaping enhances exact gradient learning with Eventprop in spiking neural networks," *Neuromorphic Computing and Engineering*, vol. 5, no. 1, p. 014001, Jan. 2025.

[36] T. C. Wunderlich and C. Pehle, "Event-based backpropagation can compute exact gradients for spiking neural networks," *Scientific Reports*, vol. 11, no. 1, p. 12829, Jun. 2021.

[37] T. Shoesmith, J. C. Knight, B. Mészáros, J. Timcheck, and T. Nowotny, "Eventprop training for efficient neuromorphic applications," Mar. 2025.

[38] "Spiking Heidelberg Digits and Spiking Speech Commands – Zenke Lab."

[39] "Papers with Code - SSC Dataset," https://paperswithcode.com/dataset/ssc.

[40] J. S. Garofolo, L. F. Lamel, W. M. Fisher, D. S. Pallett, N. L. Dahlgren, V. Zue, and J. G. Fiscus, "TIMIT Acoustic-Phonetic Continuous Speech Corpus," Jan. 1993.

[41] V. Panayotov, G. Chen, D. Povey, and S. Khudanpur, "Librispeech: An ASR corpus based on public domain audio books," in *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, Apr. 2015, pp. 5206–5210.

[42] P. Savarese, "Lolemacs/pytorch-eventprop," Oct. 2024.

[43] T. Nowotny, "Tnowotny/genn_eventprop," Jan. 2025.

[44] F. Zehra, M. Javed, D. Khan, and M. Pasha, "Comparative Analysis of C++ and Python in Terms of Memory and Tim," 2020.