Biologically Inspired Machine learning: Spiked Neural Network

Domantas Sakalys, Simon Nordensten, Jonny Igeh
(Dated: June 11, 2024)

This report presents the development and evaluation of a Spiking Neural Network (SNN) for classifying the Iris dataset. Implementing a Leaky Integrate-and-Fire (LIF) model with Spike-Timing Dependent Plasticity (STDP) for synaptic weight updates, we achieved a mean accuracy of 96.9% with STDP compared to 49.4% with non-STDP for batch size 2. Analysis of the weight matrix revealed neuron clustering similar to biological neural processes. Our model demonstrates robustness with limited data, highlighting the potential of SNNs for efficient and accurate data classification, mimicking biological neural network behaviors.

I. INTRODUCTION

In recent years, machine learning algorithms have revolutionized various domains, enabling computers to excel at tasks ranging from image recognition to natural language processing. As we delve deeper into the realm of artificial intelligence, researchers constantly strive to develop more powerful and efficient algorithms. One promising advancement that has emerged is the Spiking Neural Network (SNN), representing the next generation of machine learning models.

To understand the significance of SNNs, it is crucial to acknowledge the three generations of machine learning algorithms that led to their creation. The first-generation algorithms were primarily based on symbolic logic and expert systems. These early models relied heavily on manual rule-based approaches and faced significant limitations in handling complex and noisy data[1].

The second generation introduced the concept of artificial neural networks (ANN), inspired by the structure and functioning of the human brain. ANNs utilized interconnected layers of nodes, called neurons, to process information and learn from data. With the advent of backpropagation, these models could be trained to approximate any function, significantly enhancing their capabilities. However, traditional ANNs faced challenges in processing temporal data effectively and lacked the ability to incorporate time dynamics.

This prompted the need for the third generation of machine learning algorithms that could better handle time-varying information. Enter SNNs, an innovative approach that closely mimics the behavior of biological neural networks[2]. Unlike traditional ANNs, SNNs operate on a completely different principle - they communicate through discrete electrical impulses known as spikes, analogous to action potentials in the brain.

In this report we will investigate a simple SNN implementation in Python, using the widely studied Iris dataset. Our objective is to construct an effective SNN classifier capable of accurately predicting species based on the dataset's attributes.

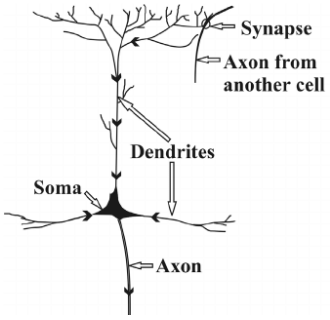

FIG. 1: An overview of the structure of a neuron. (source)

The theoretical foundations for our project will be presented neatly in section II, before showing our method of implementation and some code in section III. Then, we will present the results from running our algorithms on the dataset in section IV, and provide some insights and analysis of the network in the section V. Lastly, we will conclude our findings in section VI.

II. THEORY

A. Neuronal Dynamics

A neuron can be divided into three main parts with different functions, the dendrites, soma, and axon.

Simply put, the dendrites can be thought of as the "neuronal input" collecting signals from other neurons and transmitting them to the soma. The soma is responsible for processing the transmitted input from the dendrites. If the total input exceeds some threshold, an output signal is generated. The axon is responsible for transmitting the output signal to other neurons, an overview is given in FIG.1. We will refer to the transmitting neurons as the *presynaptic* neurons, and the receiving neurons as the *postsynaptic* neurons. The axon of the presynaptic neuron transmit signals by connecting to the dendrites (or soma) of the postsynaptic neuron. The point of connection between the two neurons are what we call a *synapse*. Our focus lies on modeling what we call a chem-

ical synapse, where the axon of the presynaptic neuron comes very close to the postsynaptic neuron leaving a gap between the two neuronal cell membranes called the *synaptic cleft*. Here the synaptic inputs are transmitted to the postsynaptic neuron through complex biochemical processes. For further reading we refer you to [3], [4].

Neuronal dynamics can be thought of as an summation (or integration) process of input signals, which we will call action potentials or spikes. In the absence of spikes, the postsynaptic neuron is at rest corresponding to a constant membrane potential u_{rest}. Suppose now that the presynaptic neuron j transmits a spike across the synapse to the postsynaptic neuron i. The effect of an incoming spike on postsynaptic neuron i will induce a change to the membrane potential. Formally, at $t = 0$ the presynaptic neuron j fires a spike, such that for $t > 0$ it induces a response for neuron i,

$$u_i(t) - u_{\text{rest}} \equiv \epsilon_{ij}(t), \tag{1}$$

where $u_i(t)$ is the momentary membrane potential for neuron i, $\epsilon_{ij}(t)$ defines the postsynaptic potential (PSP). If the change in the PSP is positive we have an excitatory PSP (EPSP), and if the change is negative an inhibitory PSP (IPSP).

Now consider two presynaptic neurons $j = 1, 2$, which both transmit spikes to the postsynaptic neuron i. Let the presynaptic neurons fire spikes at $t_j^{(1)}, t_j^{(2)}, \ldots$, where each spike induces a PSP ϵ_{i1} and ϵ_{i2} respectively. In the case of several inputs from neuron j, we call this a spike train ($u_i(t)$ now contains the summed effect of all inputs). The total change in the potential for neuron i is given (as an approximation) as the sum of the individual PSPs,

$$u_i(t) = \sum_j \sum_k \epsilon_{ij}\left(t - t_j^{(k)}\right) + u_{\text{rest}}, \tag{2}$$

i.e., the potential change is linear w.r.t the input spikes. Assuming all incoming spikes are excitatory, the membrane potential will continue to increase. If the membrane potential reaches some critical threshold value ϑ from below, the postsynaptic neuron i fires its own spike. When neuron i fires, the potential will not immediately reduce to the resting potential u_{rest} but exhibit what we call hyperpolarization, where the potential will reduce below the resting value to some value u_{reset}. This process breaks the linearity of eq. (2). Thus eq. (2) can only explain the sub-threshold behaviour of the membrane potential.

1. LIF Model

The Leaky Integrate-and-Fire (LIF) model is one of the simplest models used to neurons. This model is based on the integration of synaptic input currents and the firing of an action potential/spike when the membrane potential reaches a certain threshold. The model simplifies the complex dynamics of real neurons into an electrical circuit composed of a capacitor and a resistor. The neuron's membrane potential is treated like the voltage across the capacitor. Synaptic inputs to the neuron are modeled as currents that charge the capacitor. The 'leak' part of the model refers to the resistance that allows charge to leak out, analogous to the conductance of the neuron's membrane returning the membrane potential to its resting potential. A derivation is given in appendix A. The LIF model can be summarized in the following way:

1. **Integration**: The membrane potential $u(t)$ of the postsynaptic neuron integrates the incoming current $I(t)$ over time, which can be represented by the differential equation:

$$\tau_m \frac{\mathrm{d}u}{\mathrm{d}t} = -[u(t) - u_{\text{rest}}] + R_m I(t), \tag{3}$$

   where τ_m is the membrane time constant, u_{rest} is the resting membrane potential, R_m is the membrane leak resistance, and $I(t)$ is the synaptic input current.

2. **Leakage**: $u(t) - u_{\text{rest}}$ represents the leakage across the cell membrane, which causes the potential to decay over time back to u_{rest}.

3. **Firing**: If $I(t)$ is sufficiently strong such that the membrane potential $u(t)$ reaches the threshold value ϑ, the neuron fires an action potential / spike, and $u(t)$ is reset to a reset potential $u_{\text{reset}} < \vartheta$. Usually the reset potential is set to u_{rest} or a hyperpolarized value $u_{\text{reset}} < u_{\text{rest}}$.

4. **Refractory Period**: After firing, the neuron may enter into a refractory period during which it cannot fire another spike regardless of the input current. To model this behaviour, we define the reset potential u_{reset} such that if $u(t^f) \geq \vartheta$, then $u(t) = u_{\text{reset}}$ for $t \in (t^f, t^f + \tau_{ref}]$. Where t^f is the firing time when $u(t)$ just reached/exceeded the threshold ϑ, and τ_{ref} is the refractory time.

The model's limitations mean that it is most applicable for exploring general principles of neuronal dynamics rather than for creating realistic biophysical simulations. However, due to its simplicity and computational efficiency, the LIF model is particularly useful for studying large networks of neurons where detailed biophysical models would be computationally infeasible.

2. Receptive Fields

In the study of neuronal dynamics, the concept of a receptive field is to understand how individual neurons process sensory information. A receptive field is the specific

region of sensory space that triggers a response from a neuron. In biological systems, this might be an area of the retina sensitive to light or a portion of the skin responsive to touch.

Spiking neural networks are designed to emulate this aspect of biological stimuli response. When a stimulus is presented that falls within the neuron's receptive field, the corresponding input (presynaptic) neurons generate spikes. The target (postsynaptic) neuron integrates these spikes, by the principles of the LIF model. These inputs could be directly from sensors in a sensory layer or from other neurons in preceding network layers. If the neuron's receptive field matches the features of the stimulus well, and the timing of the incoming spikes is such that the accumulated excitation reaches the threshold before significant decay, the neuron will fire. This is a selective process, as neurons will primarily respond to stimuli that strongly activate their receptive field.

The characteristics of receptive fields within an SNN can be altered through mechanisms such as spike-timing-dependent plasticity, which adjusts the synaptic weights based on the precise timing of spikes between neurons. Through learning and adaptation, receptive fields can be shaped, allowing neurons to become more finely tuned to specific and relevant patterns of sensory input.

3. *Spike-Timing-Dependent Plasticity*

Spike-Timing-Dependent Plasticity (STDP) is a biological learning rule that adjusts the strength of connections, or synaptic weights, between neurons based on the precise timing of their spikes. This rule stems from Hebbian theory, it is observed in various types of neural tissue and is thought to be a crucial mechanism for synaptic plasticity — the process by which the connections between neurons change to reflect new experiences and facilitate learning and memory.

STDP is grounded on the principle that if a presynaptic neuron fires and then, within a short time frame, the postsynaptic neuron also fires, the synaptic weights is usually increased. This happens because the presynaptic spike could have contributed to the postsynaptic neuron reaching the threshold for firing. With repeated occurrences, the synaptic weights will be strengthened, making it more likely that the presynaptic neuron will influence the postsynaptic neuron in the future. This mechanism is called *long-term potentiation* (LTP). On the other hand, if the postsynaptic neuron fires just before the presynaptic neuron, the synaptic weights is typically decreased. The logic behind this is that if the postsynaptic neuron is already firing, then the spike from the presynaptic neuron arriving shortly afterward is not contributing to the generation of the action potential / spike. When repeated, this weakens the synaptic weights, making it less likely the presynaptic neuron will influence the postsynaptic neuron. This is called *long-term depression* (LTD).

To formalize this, assume that a presynaptic neuron fires at time t_{pre}, and the postsynaptic neuron fires at time t_{post}. We define the *latency* between the presynaptic and postsynapic spikes as $t = t_{\text{post}} - t_{\text{pre}}$. The change in synaptic weight ΔW is given by

$$\Delta W = \begin{cases} P \exp\left(\dfrac{-t}{\tau_P}\right) & \text{if } t > 0, \\ -D \exp\left(\dfrac{t}{\tau_D}\right) & \text{if } t < 0, \end{cases} \tag{4}$$

where P is the maximal synaptic potentiation, D is the maximal synaptic depression, τ_P and τ_D determines the intervals where potentiation or depression occurs, respectively. from this we see that the STDP mechanism is highly dependent on the exact timing of spikes. The potentiation or depression of synaptic weights occurs maximally when the latency between spikes goes to zero (For the case where $t = 0$ the two spikes occur at the same moment in time, thus the neurons cannot have influenced one another, as it would violate casuality.), and diminishes as the latency increases.

STDP has been used to develop learning algorithms for SNNs. Through STDP, SNNs can adapt autonomously and learn temporal patterns within data streams, making them powerful tools for time-series analysis, pattern recognition, and other tasks that are reliant on temporal dynamics.

B. Network architecture

To begin constructing our SNN, we want to combine the elements of the previous section. We begin by setting up our network with one input layer, and one output layer. The input layer will use the encoding scheme presented in section II B 1. The encoded input layer acts as the presynaptic neurons, while the output layer modeled as LIF neurons, will act as the postsynaptic neurons. The synaptic weights between presynaptic and postsynaptic neurons will yield different potential-increases for different input samples, giving us a scheme we can optimize to make more accurate predictions. An architectural overview of such a network can be seen in FIG.2.

Each sample will trigger different synaptic neurons (nodes), causing activations throughout the network infrastructure. More activation in the presynaptic neurons will cause an increased potential in subsequent postsynaptic neurons - where they reach a threshold, a prediction will be made.

1. *Encoding*

Following the encoding structure presented in [5], we set up the input layer in our SNN. By the theory of receptive fields, this layer attempts to emulate the way the neurons

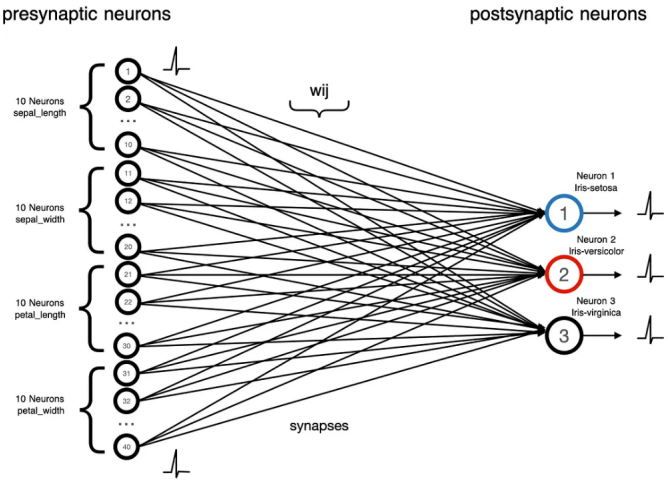

FIG. 2: Spiking Neural Network architecture for the classification of the IRIS dataset. With 10 presynaptic neurons responsible for each feature, and 3 postsynaptic neurons responsible for classification.

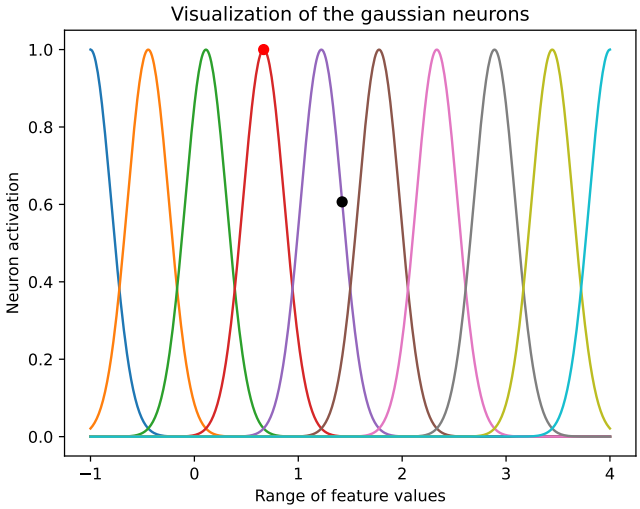

FIG. 3: Visualization of the Gaussian curves of eq. (5) representing our neurons. Here we have $K = 10$ postsynaptic neurons evenly distributed between the extreme points of one feature in the dataset.

in the brain, that are linked with the iris in the eye, gets stimulated in certain areas when observing certain shapes, colors etc. Assume now that we have a $m \times n$ input matrix A, where m is the number of observations, and n is the number of features. To extract the temporal dynamics of our input data, we will first encode it using Gaussian receptive fields. We will assign K-neurons to every feature, such that for every feature $j = 1, 2, \ldots, n$, we get an interval $I_j = [\min A_{ij}, \max A_{ij}]$, where $i = 1, \ldots, m$. Now divide this interval into K equal length sub-intervals with their respective midpoint x_k. Now for every neuron,

$$X_{j,k}(x) = A_j \exp\left(-\frac{(x - x_k)^2}{2\sigma_j^2}\right), \quad k = 1, \ldots, K, \quad (5)$$

where σ_k^2 is the variance of the k-th receptive field, and A_j is the normalization.

To simplify the notation assume now we are looking at one feature. To extract the firing times $t_{\text{pre}}^{(k)}$ and latencies of each neuron, we want to find the intersection points of every observation with the corresponding Gaussian receptive fields. From the FIG. 3 we can see two datapoints, red and black. The red dot represents a spike in neuron number 4, yielding a firing time $t_{\text{pre}}^{(4)} \approx 0.99$, while the black dot represents a spike in neuron 5 with firing time $t_{\text{pre}}^{(5)} \approx 0.5$. We can assign latencies to these values, setting $t_{\text{post}} = 1$. Meaning, that neuron 4 has a latency of $t = 1.0 - 0.99 = 0.01$ms, and neuron 5 has a latency of $t = 1.0 - 0.5 = 0.5$ms.

2. *Classification*

Classification is a supervised learning task, where we have a dataset with input samples and corresponding target values. The goal of the classifier is to learn the patterns in the dataset, and make predictions on new, unseen data. In our case, we will use the LIF-model structure that closely mimics the natural processes inside the eye and brain, as explained in the previous sections, to make predictions (classify) various target given some sample data.

3. *Iris dataset*

The Iris dataset is a well-known dataset in the field of machine learning, and is often used to test classification algorithms. The dataset consists of 150 samples of iris flowers, with four features for each sample: sepal length, sepal width, petal length, and petal width. The target values are three species of the iris flowers: setosa, versicolor, and virginica. The Iris dataset is a simple dataset, and is well-suited for building, and testing, new classification algorithms. First 50 elements in this dataset is the observations for Setosa, next 50 elements (50 - 100) are versicolor, and the last elements (100 - 150) are virginica.

III. METHODS & IMPLEMENTATION

In the theory section, we introduced the idea of the LIF model, and how it simulates the dynamics of neurons

by integrating input currents, generating spikes (fires), and incorporating leak mechanicsms. In this section, we provide a detailed explanation of the implementation of each aspect of the LIF model.

We begin with the encoding process, where we transform the raw Iris dataset into a format suitable for SNN processing. This is followed by a detailed discussion of the LIF model itself, divided into subparts covering the firing mechanism, the integration of potentials, and the implementation of the leak mechanism.

Next, we discuss the training process, highlighting our approach to updating the synaptic weights using both unmodified and modified learning rules based on STDP. Finally, we address a challenge we encountered: sometimes the model failed to generate spikes for each observation. To overcome this, we implemented batching, a technique that allows the model to process multiple input samples simultaneously, thereby enhancing its performance and stability. This approach is detailed in the final subpart of this section.

A. Encoding

As mentioned in the theory section, the first step involves encoding the Iris dataset for our model. The following algorithm outlines the logic behind the encoding process. The function is implemented in Python and operates on the design matrix of the Iris dataset, which consists of four columns representing the features and 150 rows representing the observations. The goal of this encoding method is to return a new matrix with 150 columns (one for each observation) and 40 rows (one for each presynaptic neuron) where each entry represents a latency.

1. **Initialization:**
   - Set the standard deviation for the Gaussian activation function to 0.1.
   - Initialize an empty list `latencies` to store the encoded latencies for each feature.
2. **Processing Each Feature:**
   - For each feature (column) in the design matrix, extract the feature values.
   - Compute a set of means evenly distributed between the minimum and maximum feature values. These means represent the centers of the Gaussian receptive fields for each neuron.
3. **Calculating Latencies:**
   - For each value in the feature:
     - Initialize an empty list to store latencies for each neuron.
     - For each mean:
       * Calculate the Gaussian activation of the value with respect to the mean.
       * If the activation is greater than 0.1, compute the latency as $1 - \text{activation}$. Otherwise, set the latency to 'NaN'.
4. **Storing Latencies:**
   - Append the latencies for each value to the list for the current feature.
   - Append the list of latencies for the current feature to the main `latencies` list.
   - Return the latency matrix

By following this algorithm, we ensure that each feature is accurately represented as latencies, making the data suitable for processing by the SNN.

Algorithm 1 Encoding Algorithm

Set σ^2 to 0.1
Initialize an empty list `latencies`
for each column in the dataset **do**
    Extract `feature_values` from the column
    Compute `means` as an array of N values evenly spaced between the min and max of `feature_values`
    Initialize an empty list `feature_latencies_list`
    for each value in `feature_values` **do**
        Initialize an empty list `neuron_latencies`
        for each mean in `means` **do**
            Calculate activation using Gaussian function: $\exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$
            if activation > 0.1 **then**
                Set latency to $1 - \text{activation}$
            else
                Set latency to 'NaN'
            end if
            Append latency to `neuron_latencies`
        end for
        Append `neuron_latencies` to `feature_latencies_list`
    end for
    Append `feature_latencies_list` to `latencies`
end for
Return `latencies`

B. The LIF-model

This subsection is divided into three parts. First, we explain how the potential is accumulated in the postsynaptic neurons (Integration). Next, we describe how the postsynaptic neurons fire a spike when a sufficient potential is reached (Fire). Finally, we detail how the leakage of potential is modeled (Leak).

1. Integration

In our model, we start by examining the encoded matrix *latencies*, as described in the encoding sec-

tion. This matrix consists of 150 columns (observations) and 40 rows (presynaptic neurons). Due to our encoding implementation, most elements in each column/observation are 'NaN' values, indicating that those specific neurons have large latencies and are considered inactive.

For simplicity reason, we chose to set resting potential u_{rest} to be zero.

We process each observation individually by iterating over each column vector in the latency matrix. For each observation, we calculate the contribution of each latency from the presynaptic neurons to the postsynaptic neurons. If a specific presynaptic neuron is "excited" (i.e., it has a latency value and is not 'NaN'), we take that value as t_{pre}. We simplify the contribution expression to $C = 1 - t_{pre}$. This means that if the latency is low (the neuron fires quickly after receiving visual information about the flower), the contribution will be high, providing a significant value to the postsynaptic neuron.

However, this contribution is sent to all three postsynaptic neurons, so we assign a specific weight to each contribution: $C = \omega(1 - t_{pre})$. After training the model, these weights will adjust so that specific neurons provide larger contributions to particular postsynaptic neurons. Consequently, these neurons will become more "excited" upon recognizing features of the iris flower that indicate a specific species.

After calculating all the contributions for the i-th observation, we accumulate the contributions to update the potential of the postsynaptic neurons.

The methodical steps for integration is as follows

(a) **Initialization:**
- Initialize the membrane potential P to the resting potential u_{rest}.

(b) **Processing Each Observation:**
- For each observation (column) in the latency matrix:
  - Extract the latency values for the current observation.

(c) **Calculating Contributions:**
- For each presynaptic neuron in the observation:
  - If the neuron is "excited" (i.e., the latency value is not 'NaN'):
    - Take the latency value as t_{pre}.
    - Compute the contribution $C = 1 - t_{pre}$.

(d) **Assigning Weights:**
- Assign a specific weight ω to the contribution for each postsynaptic neuron:
  - $C = \omega(1 - t_{pre})$.
  - Adjust the weights through training so specific neurons provide larger contributions to particular postsynaptic neurons.

(e) **Updating Potential:**
- Accumulate the contributions to update the potential of the postsynaptic neurons.

2. Firing

While accumulating the contributions to the potential for each of the three postsynaptic neurons, we need to ensure that when a specific threshold is reached, the postsynaptic neuron fires a pulse. This firing event is crucial as it indicates a classification in our case.

In the same iteration when we are going through the observations, we wish to check the value of potential of every postsynaptic neuron, if it exceeded the threshold value, we initiate the firing event.

(a) **Threshold Check:**
- During the accumulation of contributions, continuously monitor the membrane potential P of each postsynaptic neuron.
- If the membrane potential P exceeds the predefined threshold θ, the neuron fires.

(b) **Firing Event:**
- Upon firing, the neuron emits a pulse, corresponding to a classification decision in our model.
- We also store the firing event inside a list or a dictionary, where we keep track of the firing events to their corresponding observations.

(c) **Reset Mechanism:**
- After firing, reset the membrane potential P to the resting potential u_{rest} or another predefined reset potential u_{reset}. In our case, the resting potential is 0. Note, that we are not incorporating any hyperpolarization principles here (see theory section II A), this is done purely for simplicity reasons.

- This reset ensures the neuron is ready to process new input without being influenced by the previous high potential.

The methodical steps for firing are as follows:

3. Leak

In the same iteration where we process each observation, we also apply the leakage mechanism. There are two ways we have implemented leakage:

(a) **Constant Leak Rate:**

- Initially, we applied a simplified version where a constant leak rate was subtracted from all three potentials in every iteration.

(b) **Dynamic Leak Rate:**

- We discovered that using a dynamic leakage mechanism improved the model's performance. In this approach, the leak rate increases with the value of the potential.
- The dynamic leak rate is defined as follows:

$$\text{Dynamic leak rate} = \text{leak rate} \times \left(1 + \left|\frac{P}{\tau}\right|\right)$$

where P is the current potential value, and τ is the dynamic leak constant.

We then subtract this dynamic leak rate from the current potential value in every iteration of observation. Additionally, we ensure that the membrane potential P never drops below zero by taking the maximum of P and 0 in each iteration.

C. Training

As mentioned in the integration part of this method section, the contributions are calculated from the presynaptic neurons to all three postsynaptic neurons. During training, our task is to assign correct values to those synaptic weights. We implemented two methods for this:

(a) **Simple Weight Update Algorithm:**

- This algorithm updates the weight positively for the synapse connecting to the correct postsynaptic neuron and negatively for the other two synapses, with predifined constants which would be the hyperparameters.

(b) **Modified Learning Rule Inspired by STDP:**

- This method uses a modified learning rule inspired by (STDP), as explained in the theory section. It adjusts weights based on the timing of spikes between presynaptic and postsynaptic neurons.

  In our model however, we only track the times of presynaptic neurons from the latency matrix. We therefore only use the timings presynaptic neurons firings.

The training algorithm is supervised since we utilize the known target labels to guide the weight adjustments.

The methodical steps for the training process are as follows:

(a) **Initialization:**

- Since we are not using the whole data for training, determine the number of training observations from the shape of the latency matrix. In our case, we use 70% of the whole data as training data.

(b) **Processing Each Observation:**

- For each training observation, identify valid indices where latency values are not 'NaN'. Just as we did in the integration part.
- Extract the valid latency values for the current observation.

(c) **Calculating Weight Changes:**

- Compute the positive weight change ($\Delta W_{\text{positive}}$) as:

$$\Delta W_{\text{positive}} = \mu_{\text{positive}} \times \exp(1 - t_{pre})$$

- Compute the negative weight change ($\Delta W_{\text{negative}}$) as:

$$\Delta W_{\text{negative}} = \mu_{\text{negative}} \times \exp(1 - t_{pre})$$

- These steps are mainly for the modified learning rule inspired by STDP. If we would use the simple weight update, we would just not scale the μ.

(d) **Updating Weights:**

- Note that the weight-matrix size is [number of presynaptic neurons X number of postsynaptic neurons]
- Determine the correct target neuron (s) for the current observation.
- Update the weights for the valid indices:

$$W[\text{valid\_indices}, s] += \Delta W_{\text{positive}}$$

$$W[\text{valid\_indices}, :] += \Delta W_{\text{negative}}$$

D. Batching

We introduce a batching system because our model does not always produce a prediction from a single observation. Therefore, we need to feed the network with multiple observations to obtain a classification. This is achieved by dividing the entire test dataset into smaller, manageable subsets (batches). Specifically, we group n observations with the same target into one batch, ensuring that each batch corresponds to a single target.

(a) **Creating Batches:**
- We divide the latencies and target arrays into smaller batches of a specified size n.
- For each batch, we yield a subset of latencies and the corresponding target value.

(b) **Predicting target to each batch:**
- We create a function that takes a single batch as an input
- For that batch, we predict the class by processing the batch of latencies through the network.
- The membrane potential is reset to ensure independence between batches.

(c) **Batch Prediction:**
- We create batches from the entire dataset and predict the class for each batch.
- We call the function that we created in last step that predicts target for a single batch
- run that function for all batches
- The accuracy is computed based on the predicted and true target values.

IV. RESULTS

A. Hyper-Parameter Tuning

After implementing the entire model in Python, we aimed to optimize its performance by tuning the hyperparameters. This optimization was achieved through a grid search, systematically comparing different combinations of hyperparameter values to identify the configuration that yields the highest accuracy.

The hyperparameters considered in this study include:

- **Threshold:** The potential level that must be reached for a neuron to fire a spike.
- **Leak Rate:** The rate at which the membrane potential decays over time.
- **Mu Positive (μ_+):** The reward value used during the training process, which strengthens the synaptic connection between the presynaptic and postsynaptic neurons.
- **Mu Negative (μ_-):** The penalty value applied during the training process, which weakens the synaptic connection.
- **Tau (τ):** The parameter for the dynamic leak rate, influencing how the leak rate changes with respect to the current potential.

The specific ranges and values of the hyperparameters explored in the grid search were as follows:

- **Threshold:** 0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 5.0
- **Leak Rate:** 0.01 to 0.09 in increments of 0.01
- **Mu Positive (μ_+):** 0.01 to 0.09 in increments of 0.01
- **Mu Negative (μ_-):** -0.01 to -0.09 in increments of -0.01
- **Tau (τ):** 0.1 to 1.0 in increments of 0.1

Each combination of these parameters was evaluated to determine the optimal set of hyperparameters that maximizes the model's accuracy.

TABLE I shows a snippet of some of the hyper-parameter combinations that achieved an accuracy of 0.9 or better on the test dataset. The test dataset consisted of 30% of the entire dataset, and a batch size of 2 observations was used. The weights were trained using the modified training method, incorporating STDP.

TABLE I: Hyperparameter Combinations with High Accuracy

Threshold	**Leak Rate**	$\mu_{positive}$	$\mu_{negative}$	τ	**Accuracy**
2.0	0.01	0.03	-0.05	1.0	0.9
2.5	0.01	0.03	-0.02	0.1	0.933
3.0	0.01	0.03	-0.01	0.9	0.9
3.0	0.03	0.05	-0.01	0.8	0.933
3.5	0.01	0.07	-0.01	1.0	0.9
3.5	0.03	0.07	-0.01	1.0	0.933
4.0	0.01	0.07	-0.02	1.0	0.967
4.5	0.01	0.09	-0.02	1.0	0.967

We chose to use the following hyperparameters for the remainder of the project: `threshold` = 2.0, `leak_rate` = 0.03, $\mu_{positive}$ = 0.05, $\mu_{negative}$ = -0.01, and τ = 1.0.

B. Accuracy statistics

Since the model is initialized with random values for the weights, each training run can result in a

different set of trained weights. Consequently, the model's performance can vary, sometimes achieving higher accuracy and sometimes lower. To assess the variability in accuracy, we ran the model's training process multiple times. Specifically, we implemented a function that trains the model 80 independent times. This function calculates the mean accuracy, standard deviation, and standard error of the accuracy over these runs, providing a comprehensive view of the model's performance consistency.

Running our algorithm using STDP training, with a batch size of 2 observations, and evaluating our model on the test data yields the following average scores in TABLE II

Accuracy	**Std.dev**	**Std.err**
0.969	0.041	0.004

TABLE II: Averaged statistical values for 80 runs with optimal parameters, and a batch size of 2, and dynamic leak

FIG. 4 shows the histagram of all the accuracy values achieved over the 80 runs.

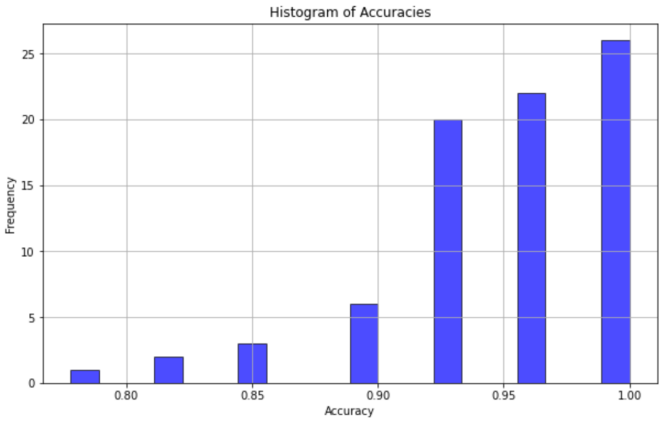

FIG. 4: Histogram of accuracies from 80 runs, showing the variability in model performance.

C. Visualization of potential evolution and postsynaptic neuron spikes

In FIG. 5, 6 and 7 we can see plotted potential for all three postsynaptic neurons. In FIG. 8 we have plotted i clearer spike representation, where every line indicates a spike event (classification) with is corresponding colour. Red for class 0, blue for class 1 and green for class 3. Note that no batching were used to produce these plots. We produced them simply by feed forwarding the whole dataset through the model. It is also clear that the potentials go below the value 0, which would be an error in our code.

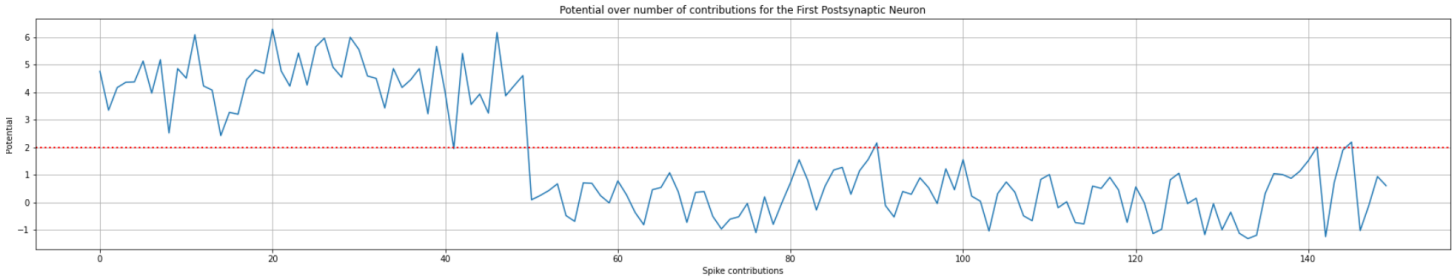

FIG. 5: Potential evolution over the number of spike contributions for the first postsynaptic neuron. The red dotted line represents the threshold value. The plot shows how the potential changes over time.

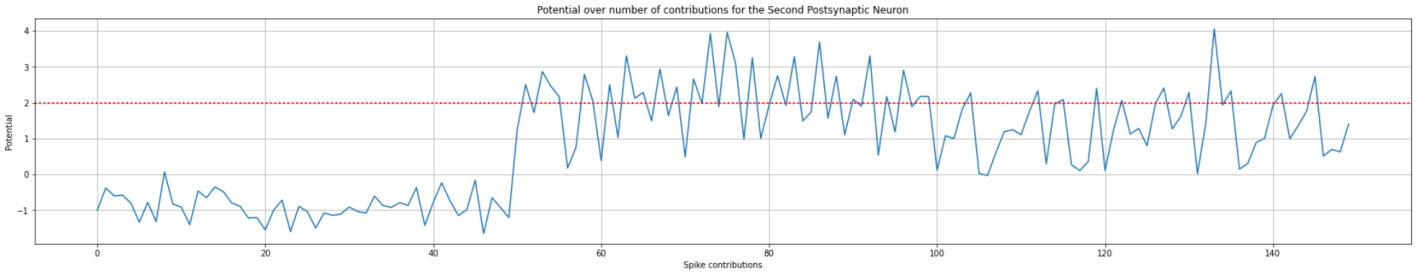

FIG. 6: Equivalant plot to FIG. 5, but for the evolution of potential for the second postsynaptic neuron.

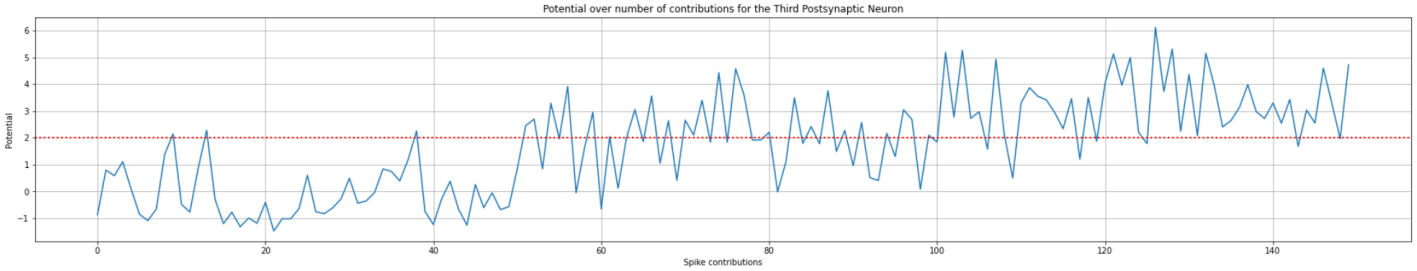

FIG. 7: Equivalant plot to FIG. 5, but for the evolution of potential for the third postsynaptic neuron.

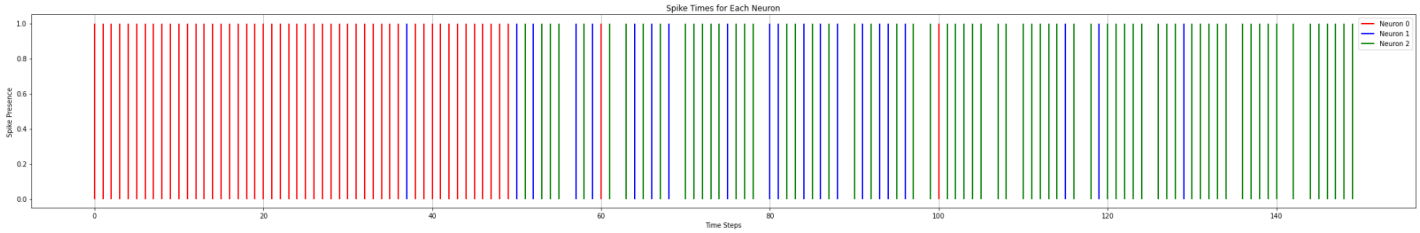

FIG. 8: Spike times for each neuron over time steps. Each vertical line represents a spike event for a specific neuron, with different colors indicating different neurons (red for Neuron 0, blue for Neuron 1, and green for Neuron 2).

D. Weight matrix evolution

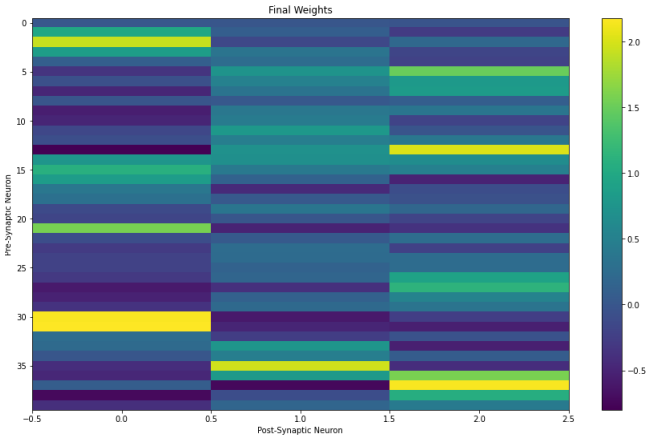

FIG. 9: Final weight matrix after training the SNN. The heatmap shows the weights between presynaptic neurons (y-axis) and postsynaptic neurons (x-axis). The color scale indicates the weight values, with yellow representing higher positive weights and purple representing lower or negative weights.

FIG. 9 presents a heatmap of the weight matrix. The displayed weight matrix was obtained after training the model using the modified training method, incorporating STDP.

E. Comparison with STDP training and non-STDP training

Batch Size	Method	Mean Acc.	Std. Dev.	Std. Error
2	Non-STDP	0.4944	0.1455	0.0163
2	STDP	0.9685	0.0410	0.0046
3	Non-STDP	0.6833	0.1640	0.0183
3	STDP	0.9694	0.0361	0.0040
4	Non-STDP	0.8375	0.1234	0.0138
4	STDP	0.9769	0.0411	0.0046

TABLE III: Comparison of training methods (Non-STDP and STDP) for different batch sizes.

To evaluate the performance of the SNN model, we conducted experiments using two different training methods: non-STDP and STDP. The experiments were performed with batch sizes of 2, 3, and 4, each repeated 80 times to ensure statistical reliability. The results of these experiments are summarized in TABLE III. From the results, it is evident that the STDP training method significantly outperforms the non-STDP method across all batch sizes. The mean accuracy is consistently higher with the STDP method, and the lower standard deviation and standard error indicate more reliable and stable performance. These results highlight the effectiveness of incorporating STDP into the training process, resulting in superior accuracy and model consistency.

F. Comparison with other popular classification algorithms

To evaluate the performance of our SNN model, we compared its accuracy on the IRIS dataset with several well-known classification algorithms: Support Vector Machine (SVM), Naive Bayes, Multi-Layer Perceptron (MLP), k-Nearest Neighbors (k-NN), and Linear Discriminant Analysis (LDA). The results are summarized in the table below.

Algorithm	Accuracy on Test Data (%)
SNN (Proposed Method)	0.969
SVM	0.96
Naive Bayes	0.96
MLP	0.927
k-NN (k=3)	0.92
LDA	0.96

TABLE IV: Comparison of classification accuracies on the IRIS dataset.

While our SNN model achieves an accuracy of 96.9%, it is important to note that this result was obtained using a batching system with a batch size of only 2 observations. Using such a small batch size still demonstrates the model's effectiveness with minimal data, suggesting that even better results could be achieved with larger batch sizes.

The classification accuracies for the other algorithms are sourced from the work by Wang et al. [6].

V. DISCUSSION

A. Hyper-parameter tuning

As presented in TABLE I, we see some of the best results gained during our grid-search for the optimal parameters. There seems to be a nice correlation between larger threshold values for our neurons to activate, and the resulting accuracy scores. This is something we may expect to see, as a higher threshold means more "time" for all the neurons to interchange information, before a prediction is made - effectively giving "more" flexibility in our model, making better use of the variational parameters.

A smaller leak-rate also seems to yield better results, and this can also be linked to similar behaviour as a higher threshold. A higher leakage

rate will make it so that activation's in the neuron potentials at different activation times will not be as strongly affected by each-other, due to much of the activation disappearing due to the high leakage. Meaning, a lower leak-rate could potentially make the neurons more sensitive to the later spikes in the model.

B. Accuracy

As seen in TABLE II, the optimal accuracy score for the classification algorithm, averaged over 80 runs, tops out at $\approx 96\%$. A common benchmark for classification algorithms is a 98% accuracy rate - meaning our SNN implementation performs at a high level. This shows the potential in our third-gen algorithm, where even our simple implementation shows prowess in learning the patterns in our IRIS-dataset.
A potential improvement to this averaged accuracy-score, would be to properly "reset" the model to the same starting point in parameter space, by seeding our randomizer. This does pose a different problem however, if the initial guess is "always" situated near a local minima, our model may perform significantly worse for "optimal" parameters due to it getting stuck in parameter space. Randomizing the initial starting point for the various runs will minimize this issue somewhat, but it does give rise to situations where poor parameters may start near the global minima, showing improved performance for effectively "worse" hyperparameters.

C. Spike times and classification

An interesting analysis of our networks performance is to look at the spike-times in the various neurons, given the different sample inputs. With our ordered Iris-dataset (samples are ordered with their target in chronological order, see II B 3, we can highlight the spike-times as seen in FIG. (8). Here we see that the first neuron fires solely (in red) throughout the first phase of our simulation, where only samples with target 0 are fed into the network. This shows that the SNN accurately predicts on this target. The two subsequent neurons, target 1 and 2, has somewhat uniform firing, but with a mixture - highlighting some inaccuracies in our model, where even some samples evaluated in red - far into the run-time of our SNN.

In FIG. 5, 6 and 7 we can clearly see that the potential goes below zero, which would indicate an error in our code. This could be the result of floating point precision errors. Because of the leak, the value of the potential could sometimes "jump" down below the 0, and get stuck there in away. Meaning the implementation of $P \leftarrow \max(P, 0)$ could be faulty.

D. Weight matrix and neuron clustering

In FIG. 9 we present the final weight matrix trained using the STDP principle, represented as a heatmap. The plot reveals that certain neurons, which are close to each other, receive higher weights than others. For instance, neurons 30 and 31 exhibit significantly large weight values, indicating that the algorithm strengthens their synaptic weights towards the first postsynaptic neuron. Simultaneously, the synaptic weights between these presynaptic neurons (30 and 31) and the second and third postsynaptic neurons are notably weakened.

This pattern suggests that the algorithm forms clusters of presynaptic neurons, dedicating them to specific postsynaptic neurons. These clusters are responsible for detecting particular features and become excited when these features correspond to a specific classification. A similar clustering pattern is observed with presynaptic neurons 36, 37, and 38, which show strengthened synaptic connections towards the third postsynaptic neuron.

This observed clustering is consistent with established principles of neural organization in biological systems. The neocortex, for example, is composed of local neural circuits that are iteratively repeated within each area, forming modular structures. According to V B Mountcastle [7], these modules are often grouped into entities by sets of dominating external connections, forming distributed systems with nested properties and long-range intracortical connections linking columns with common properties . This modular and columnar organization supports the idea that neuron clusters in our model could mimic the coordinated activity observed in biological neural networks.

In addition, the potential plots reveal that the model classifies the first class significantly better than the others. The weight matrix heatmap shows that the weights are much more scaled for the first class compared to the others. This successful weight initialization for the first class is clearly reflected in the model's high accuracy for predicting this class.

E. Data Efficiency of the model

One notable advantage of our model is its low data dependency. The Iris dataset, consisting of 150 data points, is relatively small by machine learning standards. We utilize 70% of the dataset for training, which is 105 data points. This is lower than the typical 80% used in many machine learning applications. Despite this, our model achieves high accuracy, demonstrating that it is not data-greedy and performs well even with a limited amount of training data. This efficiency highlights the robustness of our model and its suitability for applications where data is scarce.

VI. CONCLUSION

In this study, we successfully developed and evaluated a Spiking Neural Network for the classification of the Iris dataset, utilizing a Leaky Integrate-and-Fire model with Spike-Timing Dependent Plasticity. The implementation of STDP significantly enhanced the model's accuracy and consistency compared to non-STDP methods. Specifically, the model achieved a mean accuracy of 96.9% with a standard deviation of 0.0410 and a standard error of 0.004 when using a batch size of 2.

Our results demonstrate that the STDP-based training method effectively creates clusters of presynaptic neurons dedicated to recognizing specific features, mimicking the modular and columnar organization observed in biological neural networks. This clustering enhances the network's ability to accurately classify inputs by strengthening relevant synaptic connections.

Additionally, our model showed robustness in data efficiency, achieving high accuracy with a relatively small training set of 105 samples, which is lower than the typical 80% used in many machine learning applications. This indicates that the SNN with STDP is not data-greedy and performs well even with limited training data.

For the future scope of this model, it would be interesting to incorporate more of the time dynamics in the model architecture, allowing us to define the timings of the spikes for post-synaptic neurons to incorporate the full theory without simplifying the expressions. Additionally, experimenting with the number of neurons per feature could provide insights into how clustering emerges. Testing the model on larger datasets would be beneficial to evaluate how increased data volume affects the model's performance and to determine the potential improvements in accuracy and stability with more extensive training data.

In summary, our implementation of an SNN with STDP for the Iris dataset classification task provides promising results, demonstrating the potential of third-generation neural networks in achieving high accuracy and data efficiency. Future work can focus on optimizing hyperparameters further and exploring more complex datasets to validate and enhance the model's capabilities.

Appendix A: Derivation of the LIF model

Every neuron is surrounded by a cell membrane. By receiving a synaptic input current $I(t)$, the cell membrane will be charged, acting as a capacitor with capacitance C_m. Since the cell membrane is not a perfect insulator, and to account for the passive flow of ions across the membrane, we introduce a leak resistance R_m. We can then model the neuron's electrical properties as an RC-circuit, where the membrane potential $u(t)$ is subject to change based on the input current and the properties of the circuit.

We can split the input current $I(t) = I_R + I_C$, where I_R is the current through the leak resistor, and I_C is the current charging the capacitor. By Ohm's law, $I_R = u_R/R_m$, where u_R is the voltage over the leaky resistor,

$$I_R(t) = \frac{u(t) - u_{\text{rest}}}{R_m}.$$

For a capacitor, the relationship between the capacitance C_m, the charge q, and the voltage across the capacitor $u(t)$ are $C_m = q/u(t)$. Rearranging, taking the time derivative where $\frac{dq}{dt} = I_C(t)$, and assuming C_m is constant, we get

$$I_C(t) = C_m \frac{du}{dt}$$

Thus, we get that the total input current $I(t)$ is related to the membrane potential and its time derivative as follows:

$$I(t) = \frac{[u(t) - u_{\text{rest}}]}{R_m} + C_m \frac{du}{dt}$$

If we define the time constant $\tau_m = C_m R_m$ and rearrange the terms, we derive a differential equation that describes the time evolution of the membrane potential:

$$\tau_m \frac{du}{dt} = -[u(t) - u_{\text{rest}}] + R_m I(t) \tag{A1}$$

To represent the evolution of the membrane potential difference from resting potential over time after an impulse input at time t_0, the solution to eq.(A1) is,

$$u(t) - u_{\text{rest}} = \Delta u \exp\left(-\frac{t - t_0}{\tau_m}\right) \tag{A2}$$

where Δu is the change in membrane potential from u_{rest} at time t_0.

[1] Nils J Nilsson. *Principles of artificial intelligence*. Springer Science & Business Media, 1982.
[2] Wolfgang Maass. Networks of spiking neurons: the third generation of neural network models. *Neural networks*, 10(9):1659–1671, 1997.
[3] Laurence F. Abbott and Peter Dylan. *Theoretical Neuroscience: Computational and Mathematical Modeling of Neural Systems*. The MIT Press, 2001.
[4] Wulfram Gerstner, Werner M. Kistler, Richard Naud, and Liam Paninski. *Neuronal Dynamics: From single neurons to networks and models of cognition and beyond*. Cambridge University Press, 2014.
[5] Andrey Urusov. First steps in spiking neural networks, 2023. https://medium.com/@tapwi93/first-steps-in-spiking-neural-networks-da3c82f538ad.
[6] Jinling Wang, Ammar Belatreche, Liam Maguire, and Thomas Martin McGinnity. An online supervised learning method for spiking neural networks with adaptive structure. *Neurocomputing*, 144:526–536, 2014.
[7] Vernon B. Mountcastle. The columnar organization of the neocortex. *Brain*, 120(4):701–722, 1997.