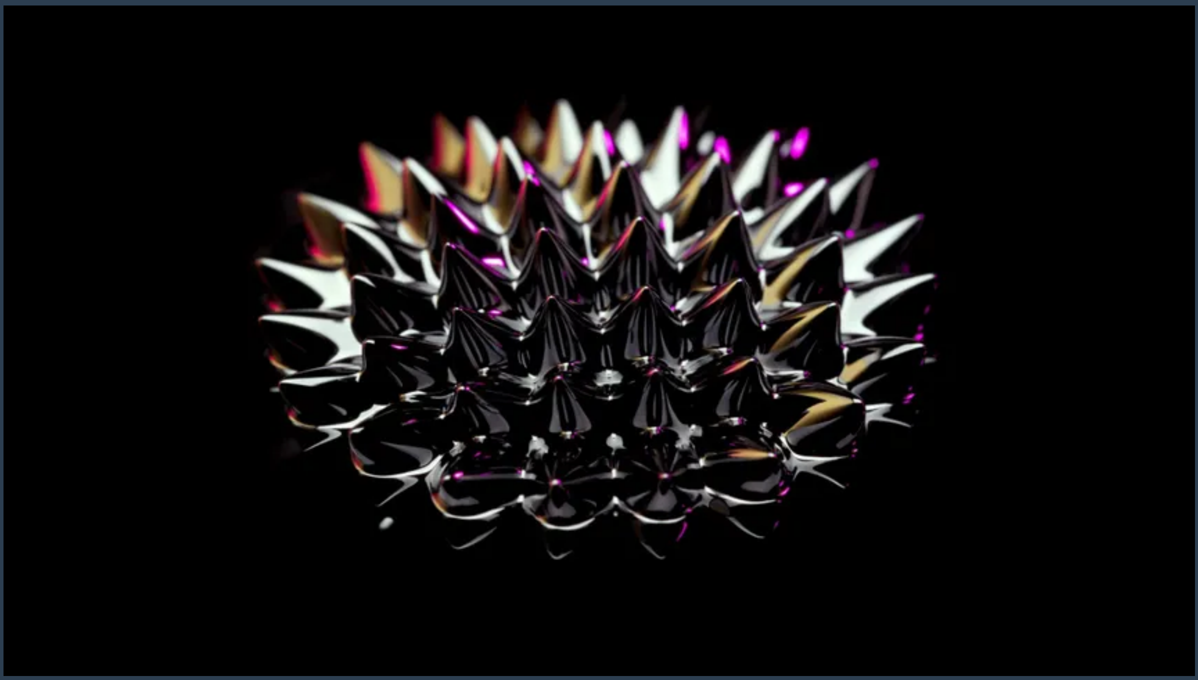
Final Report

Spiking Neural Networks
Sound Detection and Classification



COURREGE Téo
GANDEEL Lo'aï

Date: February 26, 2024

Contents

1 Introduction 4

2 Reminder on Spikes and Spiking Neural Networks 5
2.1 Spikes encoding . . . 5
2.1.1 Rate encoding . . . 5
2.1.2 Latency encoding . . . 6
2.1.3 Decoding . . . 6
2.2 Spiking neuron models . . . 7
2.2.1 Comparison of the different neuron models . . . 8
2.2.2 The choice of the Leaky Integrate-and-Fire Neuron Model . . . 8
2.2.3 Leaky Integrate-and-Fire Neuron Model . . . 9
2.3 Spiking Neural Networks . . . 12
2.3.1 Loss . . . 12
2.4 Convolutional Spiking Neural Networks . . . 13

3 SNN - Behind the scenes 14
3.1 How to train a SNN . . . 14
3.1.1 Dead Neuron Problem . . . 14
3.2 Shadow training . . . 14
3.3 Surrogate Gradient . . . 14
3.3.1 Loss function treatment . . . 15

4 Practical implementation 15

List of Figures

1 Mathematical description of a Spike[1] . . . 4
2 Spike encodings of an image [1] . . . 5
3 MNIST[1] $t = 0$ . . . 6
4 MNIST[1] $t = 15$ . . . 6
5 MNIST[1] $t = 32$ . . . 6
6 Average of the encoding of the MNIST dataset . . . 6
7 Neuron models with schemes . . . 7
8 Comparison of the different neuron models . . . 8
9 A neuron, which is enclosed by the cell membrane (big circle), receives a (positive) input current $I(t)$ which increases the electrical charge inside the cell. The cell membrane acts like a capacitor in parallel with a resistor which is in line with a battery of potential u_{rest}[2] . . . 9
10 The cell membrane reacts to a step current (top) with a smooth voltage trace (bottom)[2] . . . 9
11 Neuron model basis[3] . . . 10
12 Generating spikes[3] . . . 10
13 Definition of the variables used in the LIF model . . . 10
14 Simulation of the LIF model . . . 11
15 Result of the simulation of the LIF model . . . 12
16 Output layer with rate coding, target vector is a one-hot encoded vector . . . 13
17 (a) CSNN model, (b) CNN model . . . 13
18 Dead neuron problem[1] . . . 14
19 Visualization of the Heaviside step function approximation[3] . . . 15

List of Tables

1 Introduction

In the last report, we explored the foundational aspects of Spiking Neural Networks (SNNs) and their potential applications, focusing on audio classification tasks. We delved into the theoretical underpinnings of SNNs, studying various neuron models and encoding methods such as rate, latency, and frequency coding. Additionally, we addressed challenges related to data preprocessing, verification, and the reconstruction of audio signals using Mel-frequency cepstral coefficients (MFCCs).

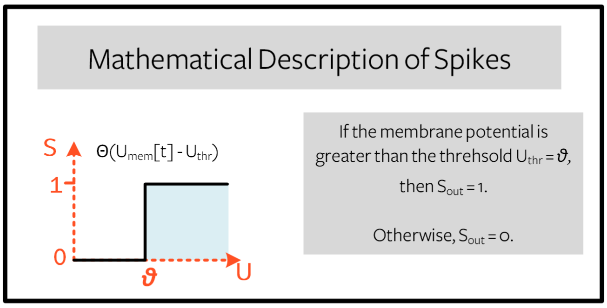


Figure 1: Mathematical description of a Spike[1]

Building upon this foundation, the current report will delve deeper into the intricacies of SNNs, specifically addressing the non-differentiability issue inherent in these networks. Non-differentiability poses a challenge in applying traditional gradient-based optimization techniques commonly used in training Artificial Neural Networks (ANNs). We will explore strategies to tackle this issue and optimize SNNs effectively.

Furthermore, we will introduce the concept of Convolutional Spiking Neural Networks (CSNNs), extending the discussion beyond simple SNN architectures. CSNNs leverage the spatial hierarchies present in convolutional neural networks (CNNs) and integrate them with the temporal dynamics of SNNs. This fusion holds promise for tasks like image recognition, where both spatial and temporal features play crucial roles.

In the practical implementation section, we will present a small program showcasing our results. This program will include the application of CSNNs in a specific task, demonstrating the capabilities and potential advantages of this hybrid architecture.

Throughout this report, our aim is to provide a comprehensive understanding of the advancements and challenges in the realm of Spiking Neural Networks, offering insights into their unique characteristics and applications.

2 Reminder on Spikes and Spiking Neural Networks

Through this section, we aim to provide a comprehensive and visual view of the concepts and mechanisms behind Spiking Neural Networks (SNNs). We will start by revisiting the fundamental aspects of spikes and their encoding, followed by an overview of neuron models (more specifically the Leaky Integrate-and-Fire (LIF) neuron model). We will then delve into the architecture of SNNs and their convolutional variant, Convolutional Spiking Neural Networks (CSNNs).

2.1 Spikes encoding

The idea behind spikes is simple: we need to encode some data (let's say an image) into some other kind of data that has a temporal dependency. To do so, we will need to define time steps, a number of steps, and a threshold. In a rate encoding scheme, we take the values of the pixels of the image and make them pass through a function that will output or not a spike (for example, a Bernoulli function).

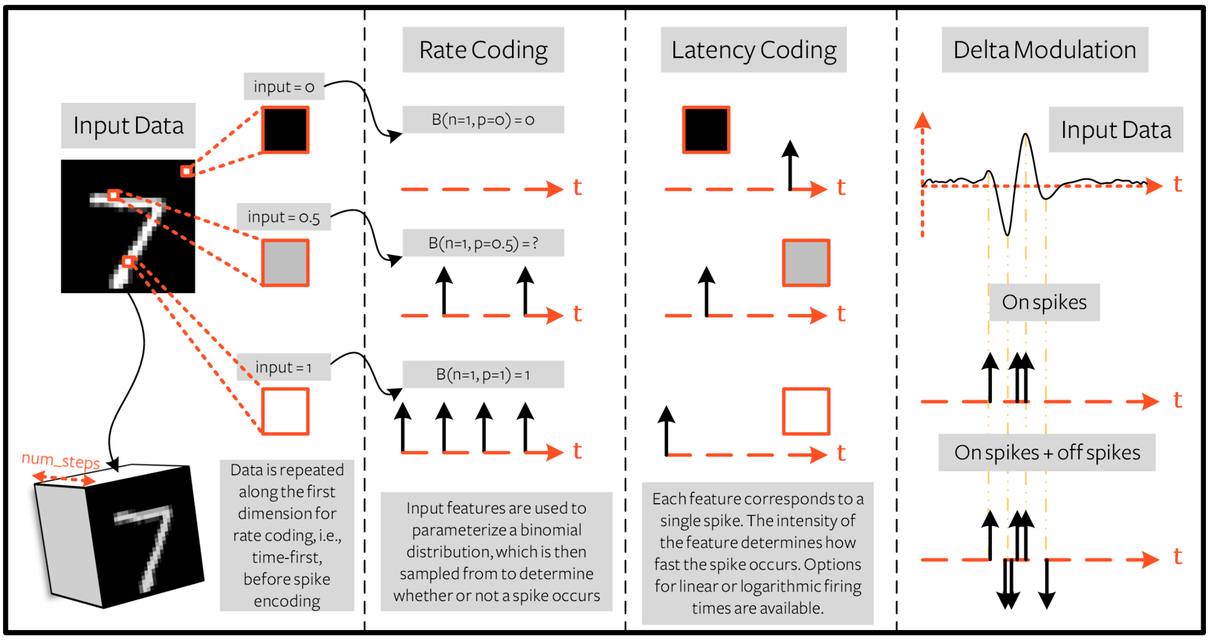


Figure 2: Spike encodings of an image [1]
Source : https://snntorch.readthedocs.io/en/latest/tutorials/tutorial_1.html

2.1.1 Rate encoding

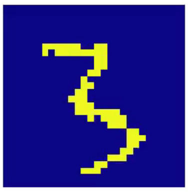
Example

The rate encoding will for each neuron at each time step, have a probability to fire a spike (1) or not (0) given by a Bernoulli distribution, where p is proportional to the intensity of the pixel associated with the neuron,.

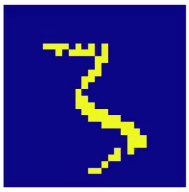
$$P(X = x) = p^x(1-p)^{1-x} \quad \text{for } x \in \{0, 1\}$$

Let's say we have a white pixel (value close to 1) and a black pixel (value close to 0). The white pixel always emits a spike while the black pixel never fires. In the case of a grey pixel, let's say with intensity of 0.5, when the number of time steps increases, we see that the neuron fires one out of two time steps on average.

This way, we end up with an approximation of the image in the form of spikes (binary values !). In order for it to be efficient, we will need encode the image multiple times. So, we will encode the image using the same method but with different time steps:



$\longrightarrow$



$\longrightarrow$

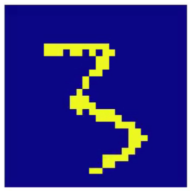


Figure 3: MNIST[1] $t = 0$

Figure 4: MNIST[1] $t = 15$

Figure 5: MNIST[1] $t = 32$

We would define a number of time steps n_{steps} and encode the image n_{steps} times. The more time steps we have, the more precise the encoding will be :

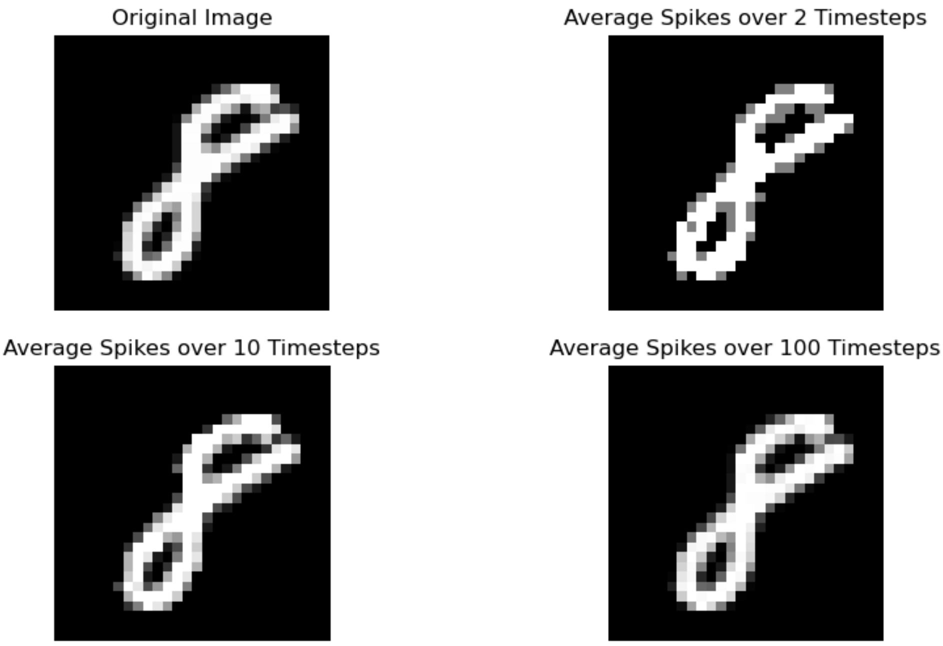


Figure 6: Average of the encoding of the MNIST dataset

2.1.2 Latency encoding

Here the intensity of the pixel is related to the spike timing with a logarithmic dependency, a brighter pixel will spike earlier than a darker one.

This encoding induces more sparsity, but is more prone to have perturbation related errors, as each spike carries way more information than with the rate encoding.

2.1.3 Decoding

For the output we need to convert spikes into something the network can interpret. Outputs neurons take input spikes and depending on the encoding chosen will have different behavior.

- Rate coding: the predicted class corresponds to the output neuron with the most spikes fired.

- Latency coding: the predicted class correspond to the output neuron that fire the first.

2.2 Spiking neuron models

Another important aspect of Spiking neural network would concern the transmission of spikes and their associated information to the next layer.

We can basically distinguish three phases in a spiking neural network, 2 of them will be explicit later on in the report :

- Encoding the data into spikes :

  Converting the input data into spikes (binary values that have approximatly the same "meaning" as the input data, but in a different form), using a specific encoding method (rate, latency, delta modulation, etc.)

- Transmitting the spikes (and their associated information) to the next layer :

  Allowing for an information to go further in the network iff the neuron fires a spike (ie. if the feature is important enough to be transmitted to the next layer), kind of imitating the role of an activation function in an ANN.

- Decoding the spikes into a meaningful output :

  For a classification task, it would involve firing or not a spike in the output layer (multiple neurons of this final layer can fire at the same time). The neuron that would fire the most over time would be the one associated to the predicted class.

To do so, we would need a type of neuron that "fires" spikes when it receives enough input (whatever the inputs are). One might find different kinds of neurons, each with its own characteristics and properties.

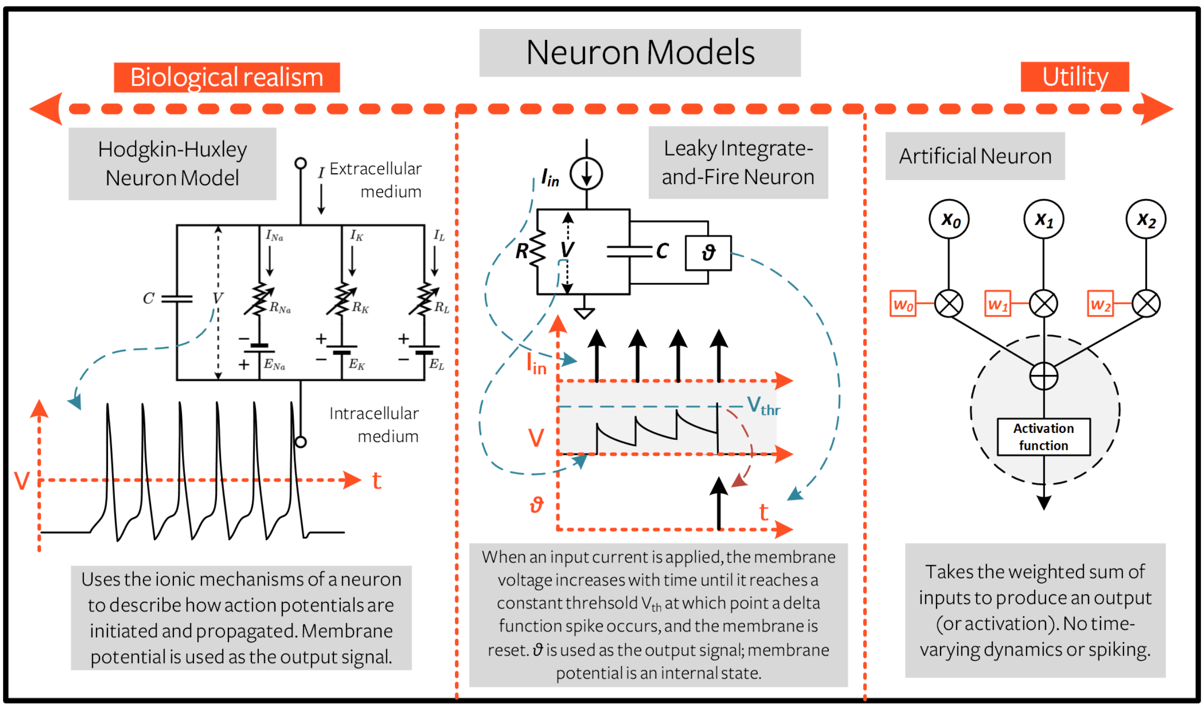


Figure 7: Neuron models with schemes

2.2.1 Comparison of the different neuron models

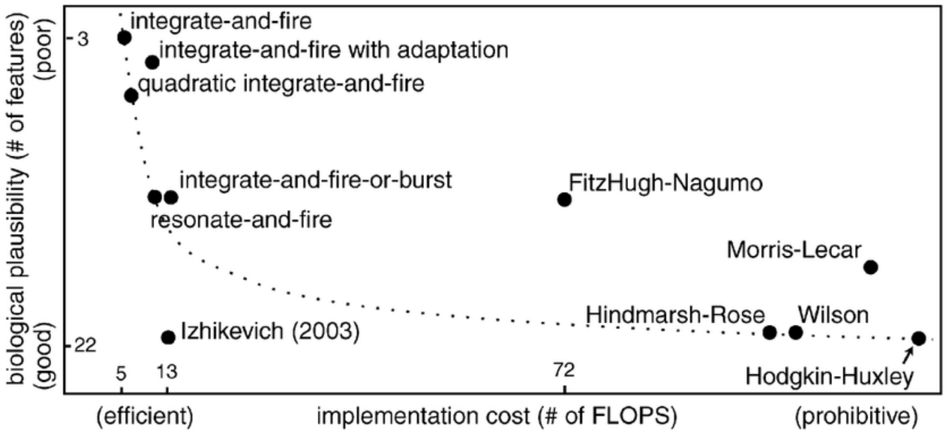


Figure 8: Comparison of the different neuron models
Source: https://www.researchgate.net/figure/Trade-off-between-biological-plausibility-with-respect-to-a-number-of-features-known-to_fig3_320409442

2.2.2 The choice of the Leaky Integrate-and-Fire Neuron Model

Initially, the LiF model was made as a simple model to mimic the behavior of a neuron.

> "In order to arrive at an equation that links the momentary voltage $u_i(t) - u_{rest}$ to the input current $I(t)$, we use elementary laws from the theory of electricity. A neuron is surrounded by a cell membrane, which is a rather good insulator. If a short current pulse $I(t)$ is injected into the neuron, the additional electrical charge $q = \int I(t')dt'$ has to go somewhere: it will charge the cell membrane. The cell membrane therefore acts like a capacitor of capacity C. Because the insulator is not perfect, the charge will, over time, slowly leak through the cell membrane. The cell membrane can therefore be characterized by a finite leak resistance R.
>
> The basic electrical circuit representing a leaky integrate-and-fire model consists of a capacitor C in parallel with a resistor R driven by a current $I(t)$ (cf the LiF "Neuron models with schemes")" [2]

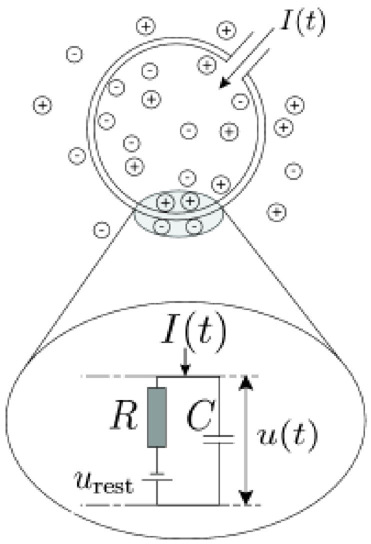


Figure 9: A neuron, which is enclosed by the cell membrane (big circle), receives a (positive) input current $I(t)$ which increases the electrical charge inside the cell. The cell membrane acts like a capacitor in parallel with a resistor which is in line with a battery of potential u_{rest}[2]

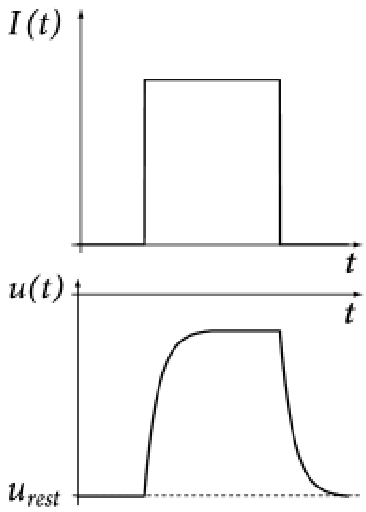


Figure 10: The cell membrane reacts to a step current (top) with a smooth voltage trace (bottom)[2]

From the Ohm's law and the capacitance equation, we can derive the following equation :

$$\Longrightarrow \; RC\frac{du(t)}{dt} = -\left[u(t) - u_{rest}\right] + RI(t) \tag{1}$$

Say the neuron starts at some value u_0 with no further input, i.e., $I(t) = 0$. The solution of the linear differential equation is:

$$u(t) = u_0 e^{-\frac{t}{\tau}}$$

Where $\tau = RC$ is the time constant of the neuron. This equation shows that the membrane potential decays exponentially to zero with a time constant τ.

2.2.3 Leaky Integrate-and-Fire Neuron Model

With the Leaky Integrate-and-Fire (LIF) model been the most used in practice. It can be seen as follows :

- We define a membrane potential U_m that will be updated at each time step

$$U_m[t+1] = \underbrace{\beta U_m[t]}_{\text{decay}} + \underbrace{WX[t+1]}_{\text{input}} - \underbrace{S[t]U_{\text{thr}}}_{\text{reset}} \quad [1] \tag{2}$$

- We define a threshold U_{thr} that will be used to reset the membrane potential when it is reached
- The output spike will be emitted following the following equation :

$$S(t) = \begin{cases} 1 & \text{if } U_m(t) \geq U_{\text{thr}} \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

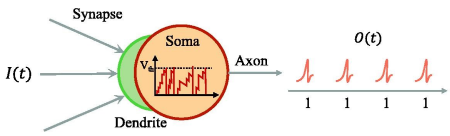


Figure 11: Neuron model basis[3]

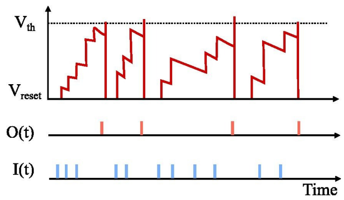


Figure 12: Generating spikes[3]

Small code example

The algorithm defining the Lif neuron model in Snntorch have the following structure :

Algorithm 1 Snntorch : Leaky Integrate-and-Fire Neuron Model

Require: $input$, mem_0

Initialize $beta$, $threshold$, $spike\_grad$, $surrogate\_disable$, $init\_hidden$, $inhibition$, $learn\_beta$, $learn\_threshold$, $reset\_mechanism$, $state\_quant$, $output$
Initialize $Leaky.beta$ and $Leaky.threshold$ if they are learnable
for each $input$ in $batch$ **do**
    if $reset\_mechanism = "subtract"$ **then**
    end if
    Compute $U[t+1] = \beta U[t] + I_{in}[t+1] - RU_{thr}$ if $reset\_mechanism = "subtract"$
    Compute $U[t+1] = \beta U[t] + I_{syn}[t+1] - R(\beta U[t] + I_{in}[t+1])$ if $reset\_mechanism = "zero"$
    Compute spk and mem_1 for each element in the batch
    Return spk and mem_1
end for

Here, the 'reset_mechanism' parameter is used to define the way the membrane potential is reset, for instead :

- If 'reset_mechanism' is set to "subtract", the membrane potential will be reset to 0 when the threshold is reached
- If 'reset_mechanism' is set to "zero", the membrane potential will be reset to $\beta U[t] + I_{in}[t+1]$ when the threshold is reached

In practice, when using Snntorch we would use 3 arrays to define this neuron model :

```python
x = torch.cat((torch.zeros(10),
               (torch.randint(0, 100, (190,)) > 80).float()),
               0)
mem_rec = []
spk_rec = []
```

Figure 13: Definition of the variables used in the LIF model

Where x is the input, a tensor made (here) of 100 elements, followed by an alternative of 0 and 1 (to mimic the spikes) randomly generated (here max of 80). The membrane potential record mem_{rec} is a list that will record the membrane potential at each time step and the spikes record spk_{rec} is a list that will record the spikes at each time step.

Once the initialization is done (including some other parameters $\beta = \exp -\frac{\Delta t}{\tau} = 0.819$ and $W = 0.4 \in \mathbb{R}$), we can run the simulation for every time step.

```python
# neuron simulation
for step in range(num_steps):
  spk_out_temp, mem_temp = leaky_integrate_and_fire(mem_temp, x[step], w=w, beta=beta)
  mem_rec.append(mem_temp)
  spk_rec.append(spk_out_temp)

# convert lists to tensors
mem_rec = torch.stack(mem_rec)
spk_rec = torch.stack(spk_rec)
```

Figure 14: Simulation of the LIF model

And we get the following results :

- At the top of the image the input x is shown
- In the middle of the image the membrane potential mem_{rec} is shown
- At the bottom of the image the spikes spk_{rec} are shown

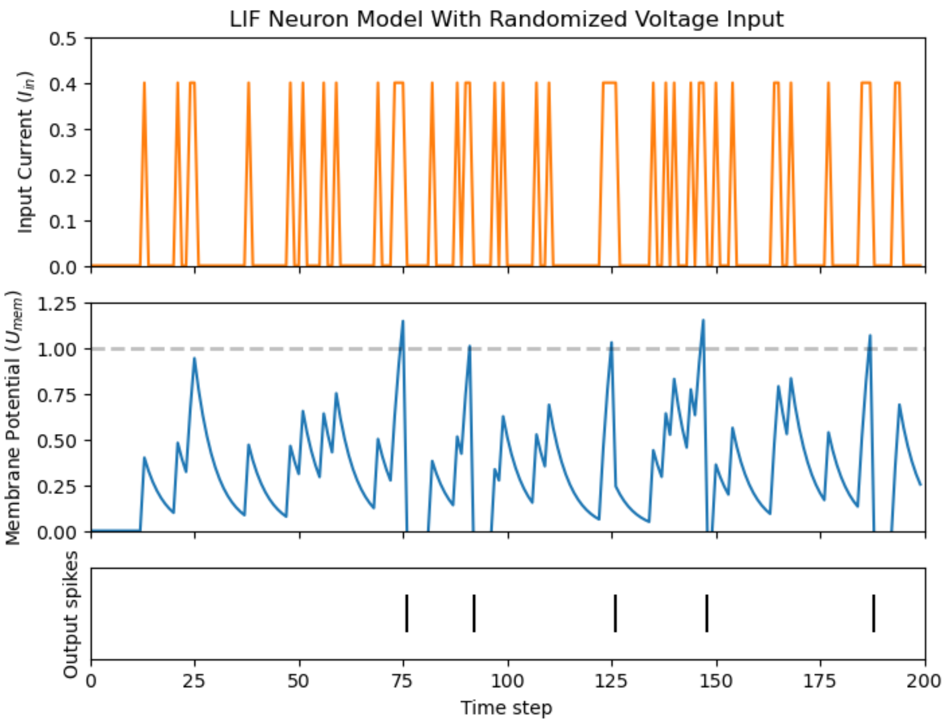


Figure 15: Result of the simulation of the LIF model

2.3 Spiking Neural Networks

Now that the spiking neuron model is defined, we can create a network. The network is a mix between classic layers (for example fully connected layer) and spiking layers. Firstly a spiking layer needs to encode the data into spikes, which are then processed by the other layer. In some way, spiking layers will "take the place" of activation functions in CNN.

2.3.1 Loss

To train the model, we use the cross entropy loss. With rate encoding the spike for the correct class is encouraged to produce a spike at each time step where the other are suppressed.

To compute the loss with N_c classes, first the number of spikes is counted for each output neuron. With a spike count of output layer $\vec{c}$, we compute the softmax of spike count for each neuron.

$$p_i = \frac{e^{c_i}}{\sum_{i=1}^{N_c} e^{c_i}}$$

The cross entropy between p_i and the target $y_i \in \{0, 1\}$, which is a one-hot target vector, is obtained using:

$$LCE = \sum_{i=0}^{N} y_i \log(p_i)$$

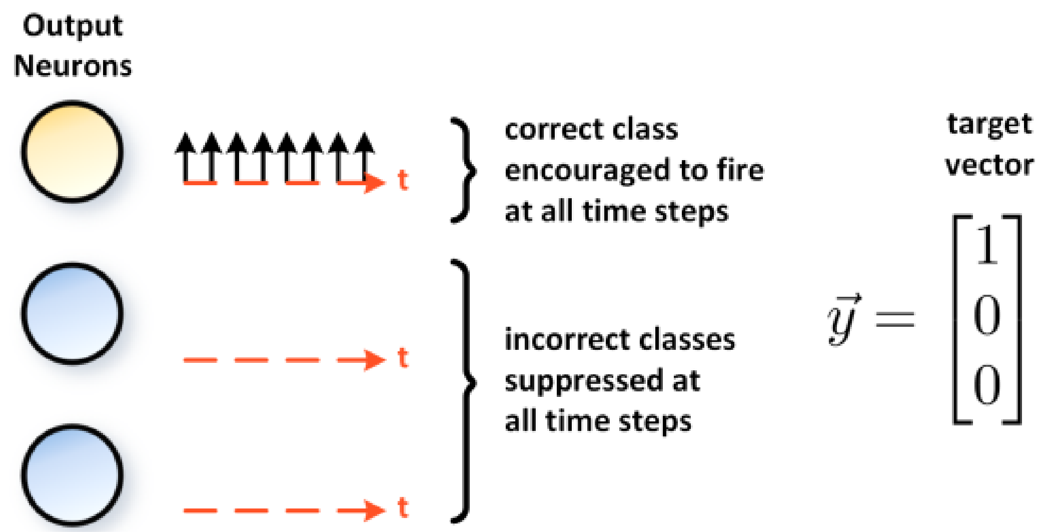


Figure 16: Output layer with rate coding, target vector is a one-hot encoded vector

2.4 Convolutional Spiking Neural Networks

A convolutional Spiking neural network, or CSNN, is a particular case of SNN where the data pass into convolutional spiking layers to extract meaningful features.

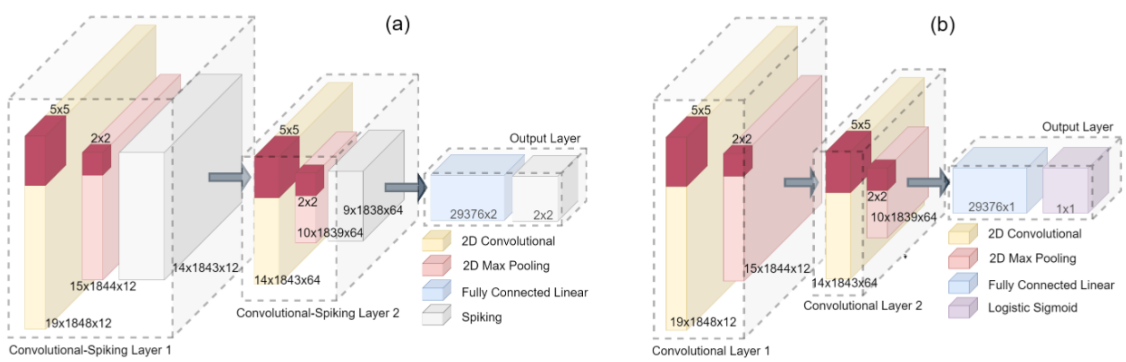


Figure 17: (a) CSNN model, (b) CNN model

3 SNN - Behind the scenes

3.1 How to train a SNN

3.1.1 Dead Neuron Problem

To train a spiking neural network, we aim to adjust the weights based on the loss gradient, minimizing the overall loss. Backpropagation achieves this through a chain of derivatives (for one single time step):

$$\frac{\partial \mathcal{L}}{\partial W} = \frac{\partial \mathcal{L}}{\partial S} \underbrace{\frac{\partial S}{\partial U}}_{\{0,\infty\}} \frac{\partial U}{\partial I} \frac{\partial I}{\partial W} \tag{4}$$

Here, $\mathcal{L}$ is the loss, W represents weights, S is the output, U is the activation function, and I is the input.

The challenge lies in the term $\frac{\partial S}{\partial U}$, which takes values between 0 and ∞. The derivative of the Heaviside step function from the input (U) is the Dirac Delta function. This function is 0 everywhere except at the threshold θ, where it tends to infinity. Consequently, the gradient is often nullified to zero (or saturated if θ precisely aligns with the threshold), hindering learning. This issue is commonly known as the **dead neuron problem**. There are multiple ways to address this issue.

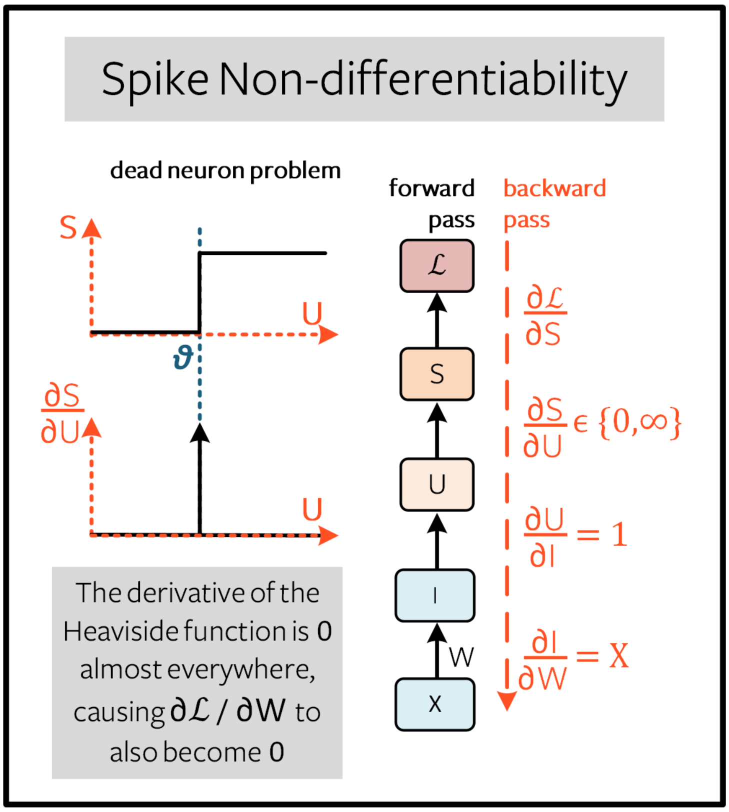


Figure 18: Dead neuron problem[1]

3.2 Shadow training

In shadow training, a classic ANN is trained and converted into an SNN by interpreting the "excitation" of the activation function as a firing rate (rate coding) or spike time (latency coding). The goal is to achieve the same input-output mapping with a deep SNN as the original ANN. This is very practical because it doesn't need particular knowledge about SNNs for the training. However, it has some flaws, for example one major obstacle is that in ANNs it does not matter if activations are negative, whereas firing rates in SNNs are always positive.[4]

3.3 Surrogate Gradient

One way to address this non-differentiability issue would be to compute a gradient on a "relaxed" version of the non-differentiable function. This would mean approximating the Heaviside step function $S(U)$ with a differentiable function $f(U)$, such as the sigmoid function or the $\arctan$ function. For example:

$$S(U) = \begin{cases} 0 & \text{if } U < U_{\text{thr}} \\ 1 & \text{if } U \geq U_{\text{thr}} \end{cases}$$

$$S(U) \approx \frac{1}{1 + e^{-(U - U_{\text{thr}})}}$$
$$S(U) \approx \frac{1}{\pi} \arctan((U - U_{\text{thr}})) + \frac{1}{2}$$
$$S_k(U) \approx \frac{1}{2} \frac{U - U_{\text{thr}}}{\sqrt{1 + k|U - U_{\text{thr}}|}} + \frac{1}{2}$$

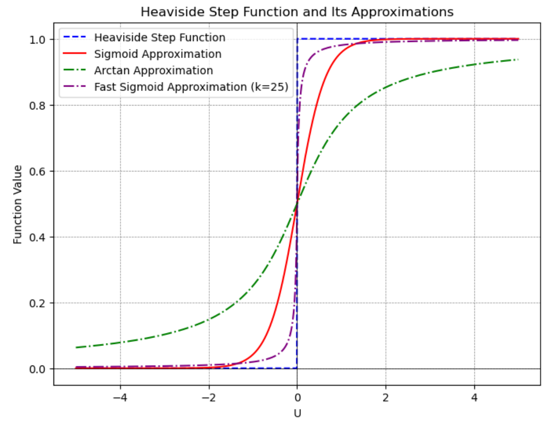


Figure 19: Visualization of the Heaviside step function approximation[3]

This solution should only account for the backpropagration process. Indeed, we only need to approximate the derivative of the Heaviside step function, not the function itself. The approximation of the function is only used to compute the gradient, and the original function is used to compute the output of the neuron.

3.3.1 Loss function treatment

In order for this to be taken into account in the loss function (and because we now have to take the time as a parameter), we have to perform an operation on the loss function for all its time steps.

$$\frac{\partial \mathcal{L}}{\partial W} = \sum_t \frac{\partial \mathcal{L}[t]}{\partial W} = \sum_t \sum_{s \leq t} \frac{\partial \mathcal{L}[t]}{\partial W[s]} \frac{\partial W[s]}{\partial W} \tag{5}$$

In the above equation, we ensure causality by summing over all time steps t and all previous time steps s. We thereby make the assumption that the weights at time $s \in [||0, t||]$ influence the weights at time t.

- We define as *prior influence* the influence of the weights at time $s < t$ on the weights at time t.
- We define as *immediate influence* the influence of the weights at time t on the weights at time t.

4 Practical implementation

References

[1] J. K. Eshraghian, M. Ward, E. Neftci, X. Wang, G. Lenz, G. Dwivedi, M. Bennamoun, D. S. Jeong, and W. D. Lu, *Training spiking neural networks using lessons from deep learning*, Proceedings of the IEEE, 111 (2023), pp. 1016–1054.

[2] W. Gerstner, W. M. Kistler, R. Naud, and L. Paninski, *Neuronal Dynamics: From single neurons to networks and models of cognition*, Cambridge University Press, 2014.

[3] X. Liao, Y. Wu, Z. Wang, D. Wang, and H. Zhang, *A convolutional spiking neural network with adaptive coding for motor imagery classification*, Neurocomputing, 549 (2023), p. 126470.

[4] M. Pfeiffer and T. Pfeil, *Deep learning with spiking neurons: Opportunities and challenges*, frontiers, 12 (2018).