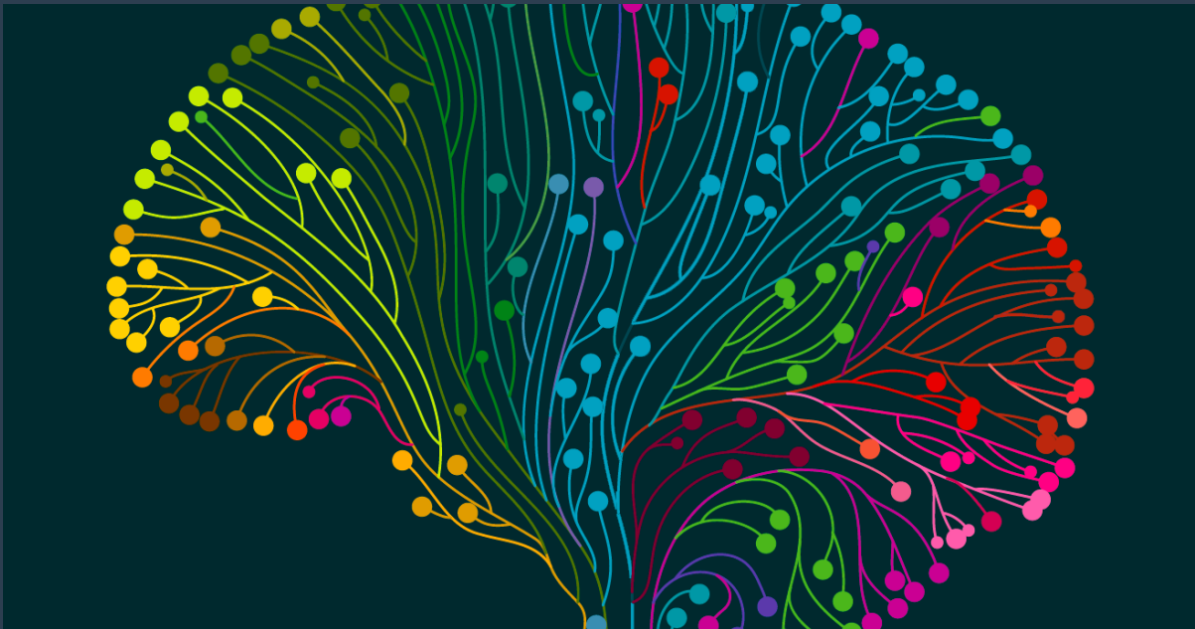# Final Report

Spiking Neural Networks
Sound Detection and Classification

COURREGE Téo
GANDEEL Lo'aï

Date: February 26, 2024

Contents

List of Figures

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).
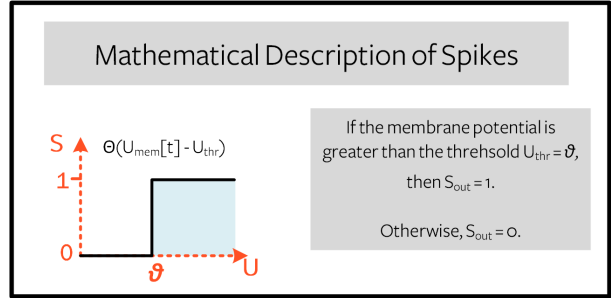
Figure 1: Mathematical description of a Spike[1]

Building upon this foundation, the current report will delve deeper into the intricacies of SNNs, 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.

Furthermore, we will address 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.

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.

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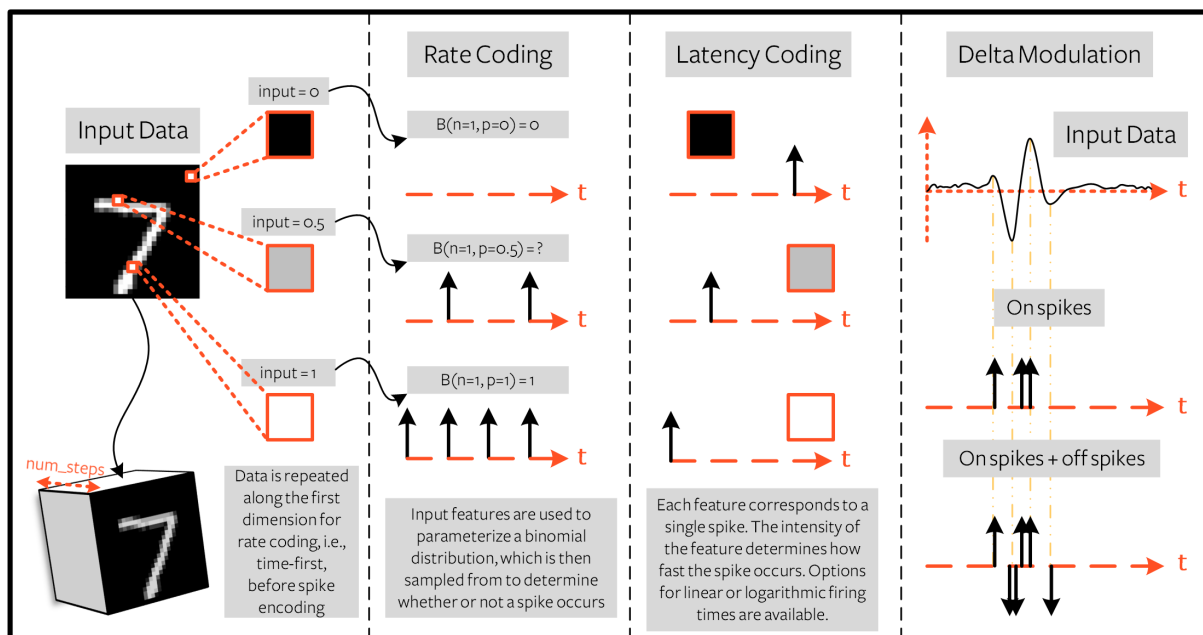
Figure 2: Spike encodings of an image [1]
Source : https://snntorch.readthedocs.io/en/latest/tutorials/tutorial_1.html

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:

 $\longrightarrow$ $\longrightarrow$
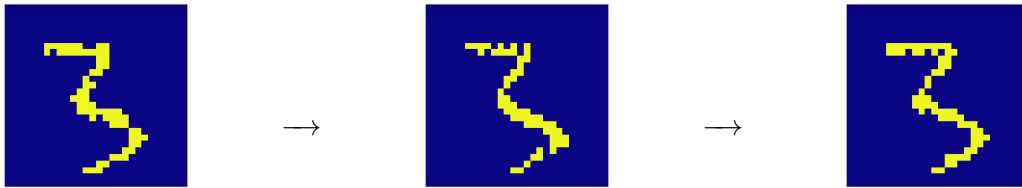
Figure 3: MNIST[1] $t = 0$ Figure 4: MNIST[1] $t = 15$ Figure 5: MNIST[1] $t = 32$

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 :
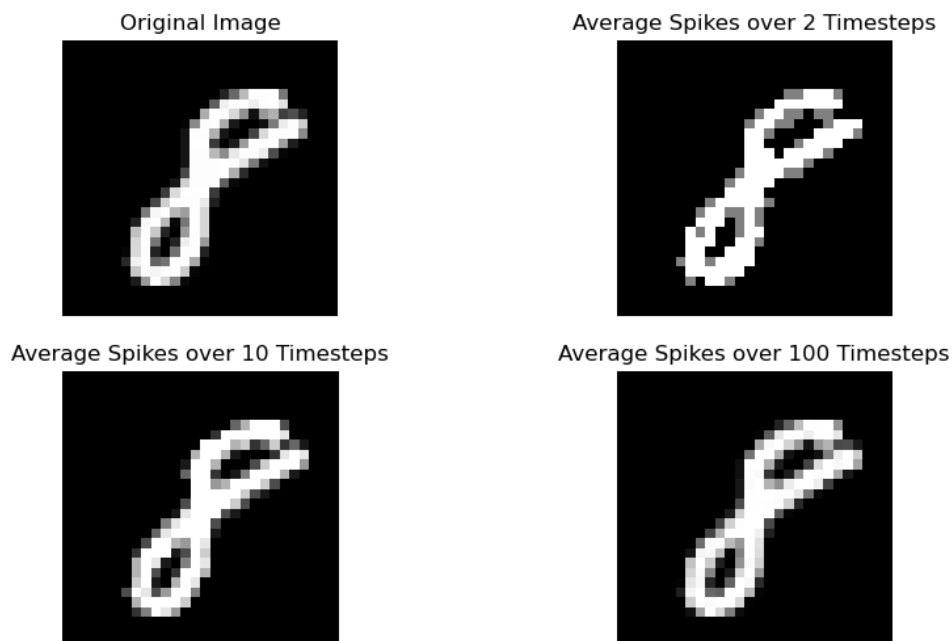
Figure 6: Average of the encoding of the MNIST dataset

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.

5

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.
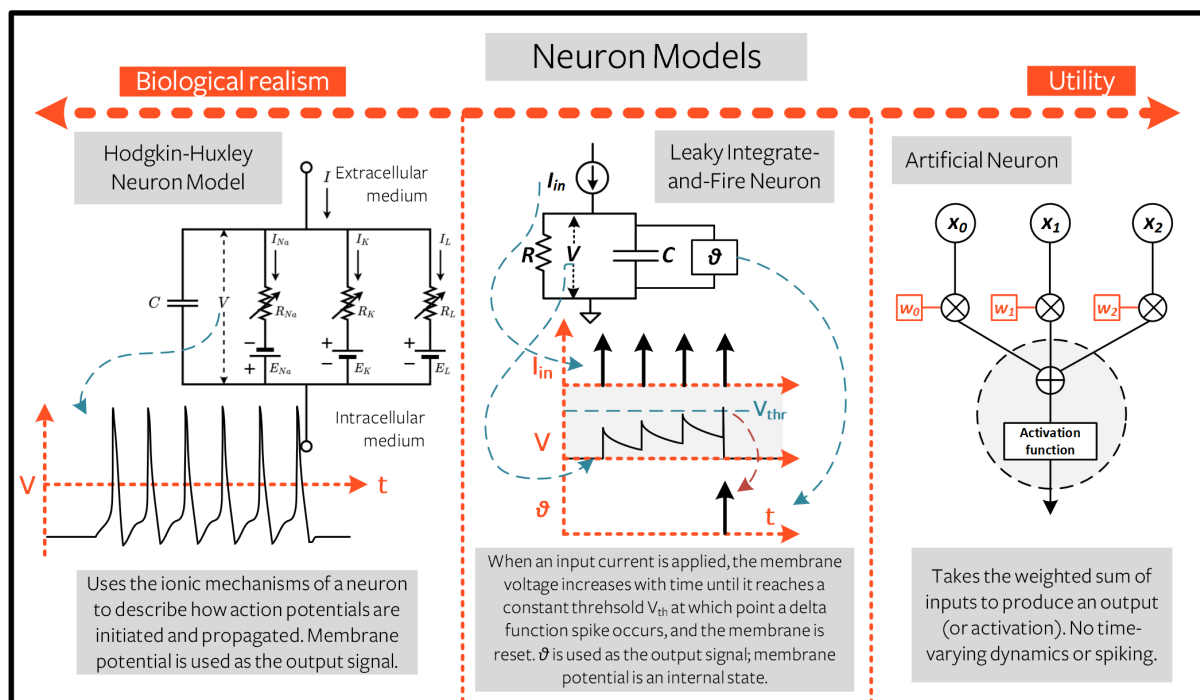

Figure 7: Neuron models with schemes

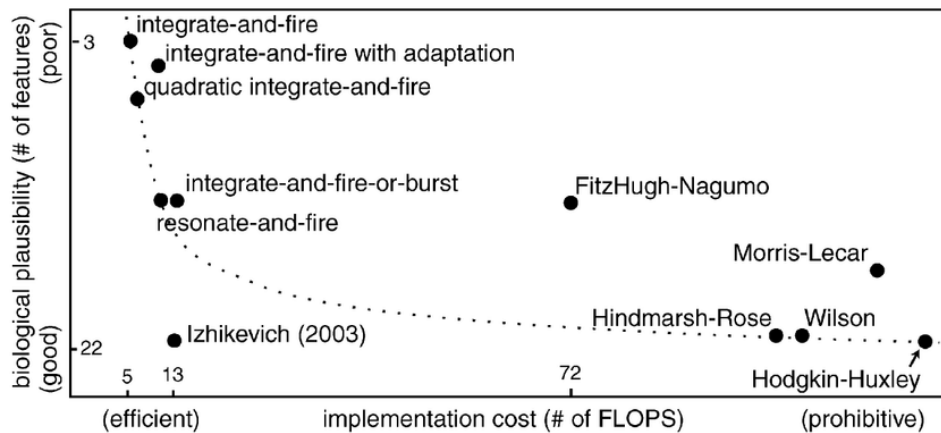### 2.2.1 Comparison of the different neuron models

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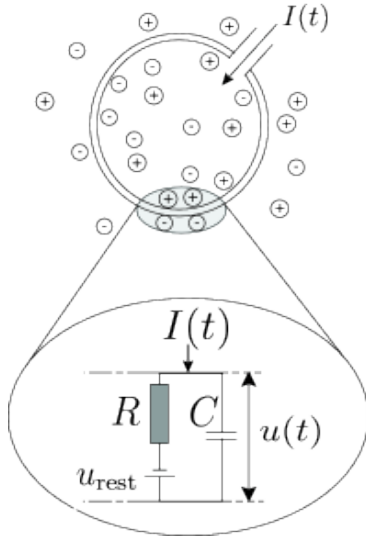
Figure 9: A neuron, which is enclosed by the cell membrane (big circle), receives a (positive) input current $I(t)$ which increases the electrical charge inside the cell. The cell membrane acts like a capacitor in parallel with a resistor which is in line with a battery of potential u_{rest}[2]
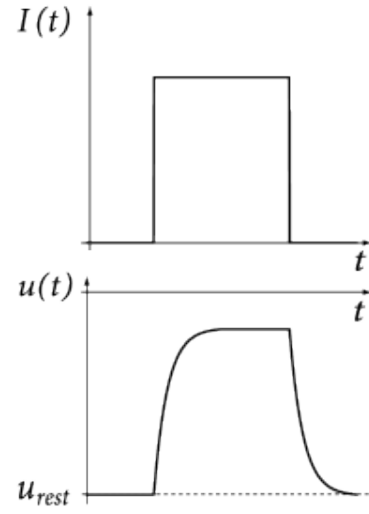

Figure 10: The cell membrane reacts to a step current (top) with a smooth voltage trace (bottom)[2]

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

$$\implies RC\frac{du(t)}{dt} = -u(t) + RI(t) \tag{1}$$

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

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

- 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 :

$$S(t) = \begin{cases} 1 & \text{if } U_m(t) \geq U_{\text{thr}} \\ 0 & \text{otherwise} \end{cases} \tag{3}$$
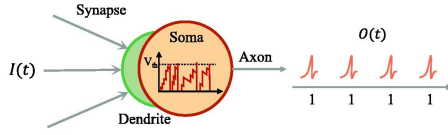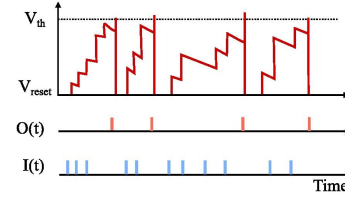
Figure 11: Neuron model basis[3]

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
 1: Initialize $beta$, $threshold$, $spike\_grad$, $surrogate\_disable$, $init\_hidden$, $inhibition$, $learn\_beta$, $learn\_threshold$, $reset\_mechanism$, $state\_quant$, $output$
 2: Initialize $Leaky.beta$ and $Leaky.threshold$ if they are learnable
 3: **for** each $input$ in $batch$ **do**
 4:     **if** $reset\_mechanism = "subtract"$ **then**
 5:     **end if**
 6:     Compute $U[t+1] = \beta U[t] + I_{in}[t+1] - RU_{thr}$ if $reset\_mechanism = "subtract"$
 7:     Compute $U[t+1] = \beta U[t] + I_{syn}[t+1] - R(\beta U[t] + I_{in}[t+1])$ if $reset\_mechanism = "zero"$
 8:     Compute spk and mem_1 for each element in the batch
 9:     Return spk and mem_1
10: **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 :

```
1  x = torch.cat((torch.zeros(10),
2                 (torch.randint(0, 100, (190,)) > 80).float()),
3                 0)
4  mem_rec = []
5  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

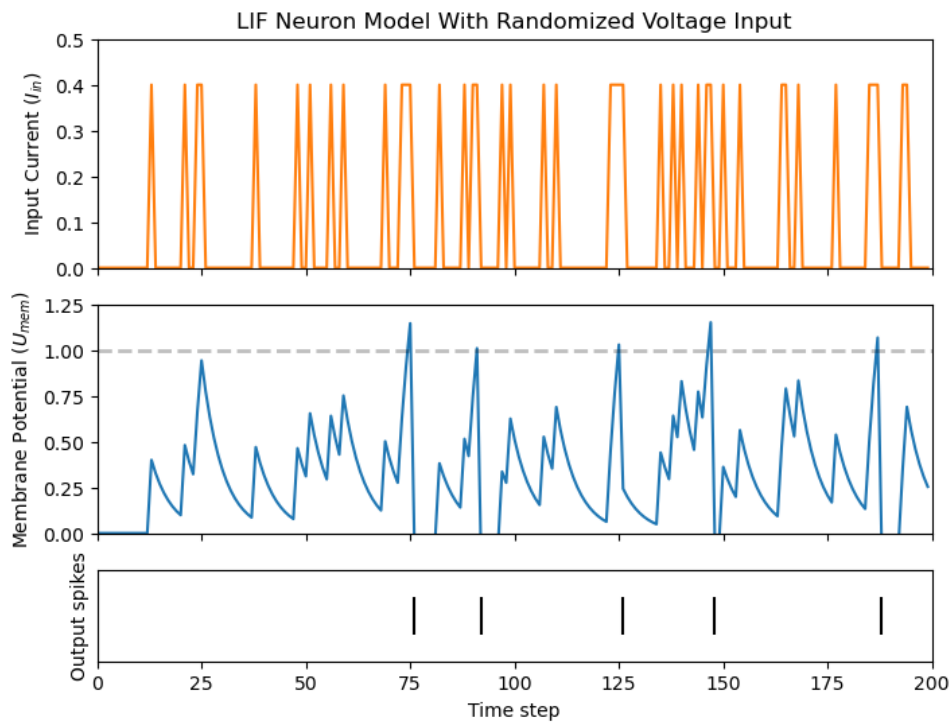- At the bottom of the image the spikes spk_{rec} are shown

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. The input is encoded in spikes, then processed by the network at each time step.
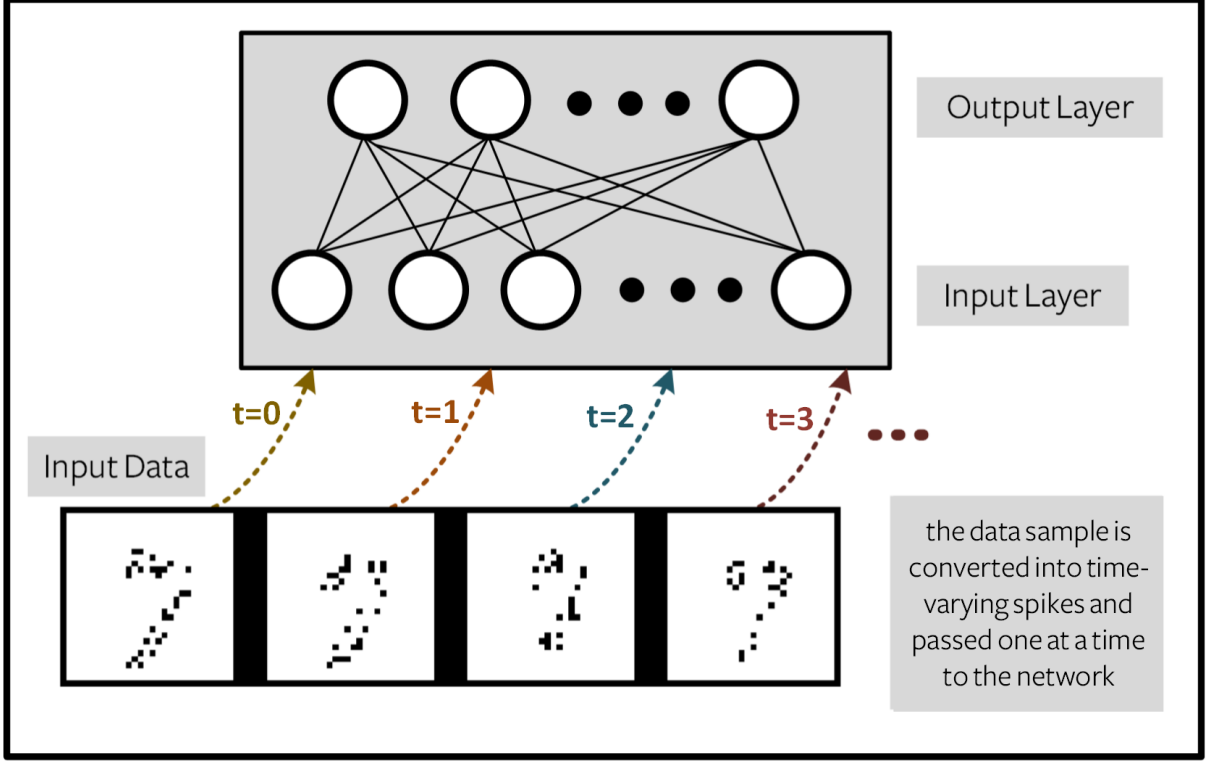

Figure 16: Network with encoded inputs

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 (loss is computed for the sum of the time steps: $\mathcal{L} = \sum_t \mathcal{L}[t]$). 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:

$$\mathcal{L}CE = \sum_{i=0}^{N} y_i \log(p_i)$$

In the backpropagation, parameters are updated for each time step. Virtually, given a number of time steps n, we replicate the number of parameters of networks n times.
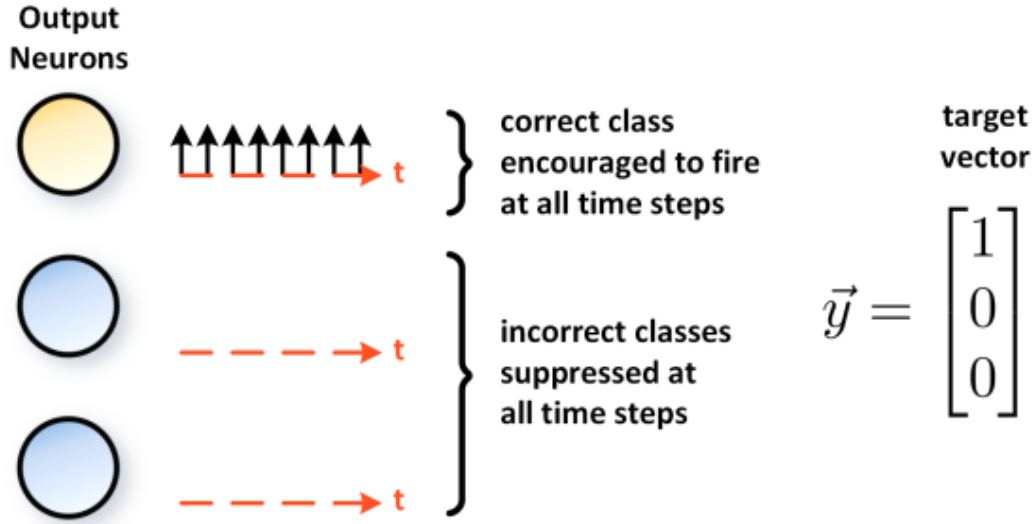
Figure 17: Output layer with rate coding, target vector is a one-hot encoded vector

In the general scenario, the more spikes an output neuron is firing when compared to the others, the more likely it is to be the predicted class.

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.

At each time step, the image is converted, into spike, then passed in he convolutional spiking layer. For each time step, weights are updated.
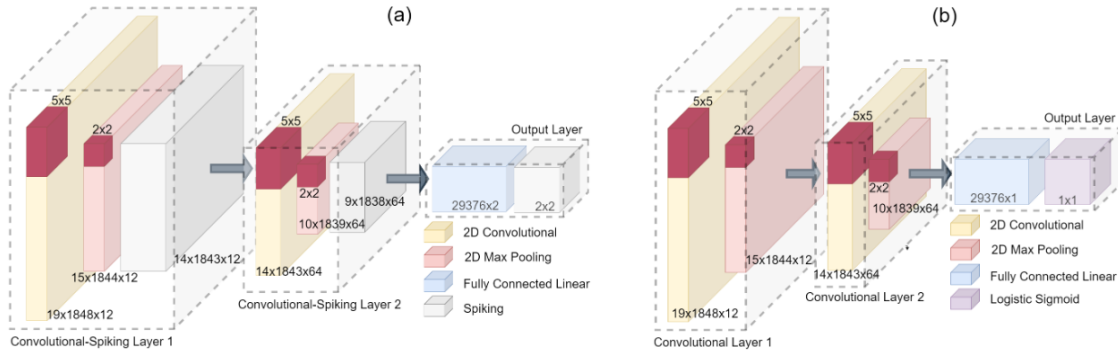

Figure 18: (a) CSNN model, (b) CNN model

3 SNN - Behind the scenes

3.1 How to train a SNN

3.2 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):

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

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.
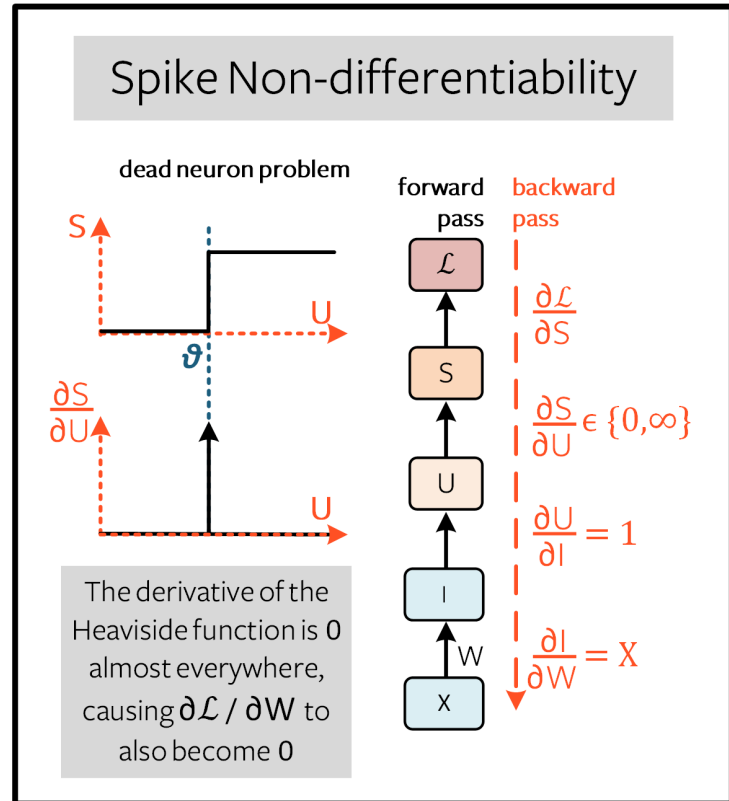

Figure 19: Dead neuron problem[1]

3.3 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]

3.4 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:

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

$$S(U) \approx \frac{1}{1 + e^{-(U - U_{\text{thr}})}}$$

$$S(U) \approx \frac{1}{\pi} \arctan((U - U_{\text{thr}})) + \frac{1}{2}$$

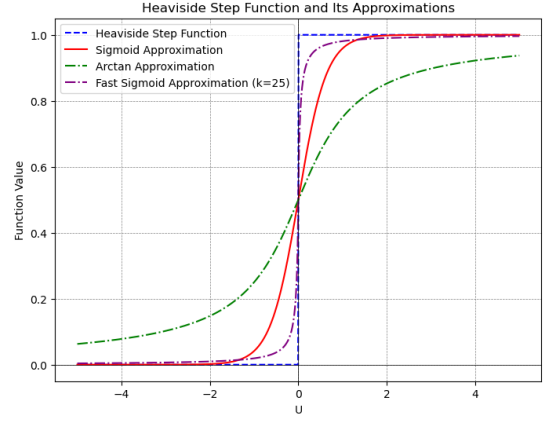$$S_k(U) \approx \frac{1}{2} \frac{U - U_{\text{thr}}}{\sqrt{1 + k|U - U_{\text{thr}}|}} + \frac{1}{2}$$

Figure 20: 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.

3.4.1 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.

$$\frac{\partial \mathcal{L}}{\partial W} = \sum_t \frac{\partial \mathcal{L}[t]}{\partial W} = \sum_t \sum_{s \leq t} \frac{\partial \mathcal{L}[t]}{\partial W[s]} \frac{\partial W[s]}{\partial W} \tag{4}$$

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.

A recurrent system constrains the weight to be shared across all steps: $W[0] = W[1] = \ldots = W$. Therefore, a change in $W[s]$ will have the same effect on all W, which implies that $\partial W[s]/\partial W = 1$:

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

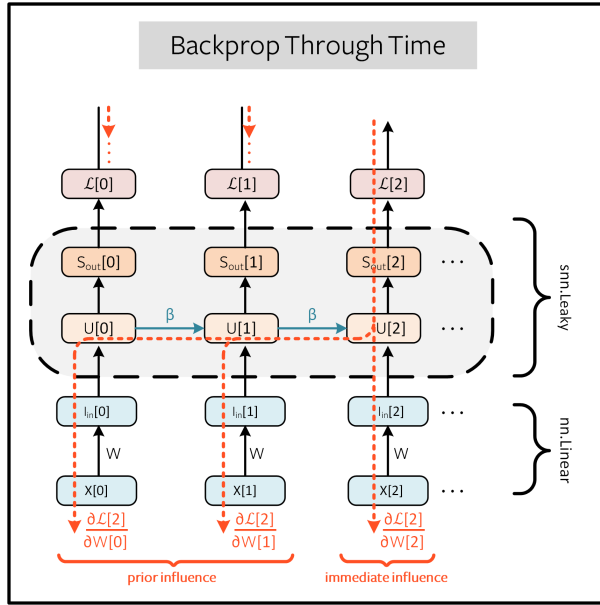By performing a chain rule, we can express the derivative of the loss with respect to the weights at time s as:

14

Figure 21: Backpropagation through time

$$\frac{\partial \mathcal{L}[t]}{\partial W[t-1]} = \frac{\partial \mathcal{L}[t]}{\partial S[t]} \underbrace{\frac{\partial \tilde{S}[t]}{\partial U[t]}}_{Eq.\ (5)} \underbrace{\frac{\partial U[t]}{\partial U[t-1]}}_{\beta} \underbrace{\frac{\partial U[t-1]}{\partial I[t-1]}}_{1} \underbrace{\frac{\partial I[t-1]}{\partial W[t-1]}}_{X[t-1]}$$

$$\frac{\partial \mathcal{L}[t]}{\partial W[t-2]} = \frac{\partial \mathcal{L}[t]}{\partial S[t]} \underbrace{\frac{\partial \tilde{S}[t]}{\partial U[t]}}_{Eq.\ (5)} \underbrace{\frac{\partial U[t]}{\partial U[t-2]}}_{\beta^2} \underbrace{\frac{\partial U[t-2]}{\partial I[t-2]}}_{1} \underbrace{\frac{\partial I[t-2]}{\partial W[t-2]}}_{X[t-2]}$$

$$\vdots$$

$$\frac{\partial \mathcal{L}[t]}{\partial W[0]} = \frac{\partial \mathcal{L}[t]}{\partial S[t]} \underbrace{\frac{\partial \tilde{S}[t]}{\partial U[t]}}_{Eq.\ (5)} \underbrace{\frac{\partial U[t]}{\partial U[0]}}_{\beta^t} \underbrace{\frac{\partial U[0]}{\partial I[0]}}_{1} \underbrace{\frac{\partial I[0]}{\partial W[0]}}_{X[0]}$$

[1]

4 Practical implementation

4.1 Encountered difficulties

We encountered several difficulties during the project, some of which are listed below:

- The dataset was quite large and unbalanced, which made it difficult to train the models.

- Training a spiking neural requieres (virtually) to train a "new model" for each time step, which is quite ressource and time consuming. For example, we observe that training our CSNN model is equivalent in terms of time consumption to training the equivalent CNN + RESNET50 + RESNET101 + EFFICIENTNET-B0 (40 to 60 epochs each)

- The memory consumption issue was also a problem, as we had to reduce the number of MFCC features or the number of time steps to make the model trainable (using the memory of 2 GPUs at the same time).

- Kaggle's limitations on the number of hours to train with the GPU that we had access to (combined with the previous point, we ended up finding that the program crashed after n epochs only). Also we were not able to save all the models we trained due to Kaggle's limitations accessing the notebook.

4.2 Defining studied models

Let us now present a small program showcasing our results. In this section, we aim to show some of the results we had and their meaning in the context of the project.

We defined 2 SNN models during the last full time period of the project. The first one is a simple feedforward SNN, and the second one is a convolutional SNN. We also defined the same models but with the ANN architecture, and also tried some other kownn architectures for comparison. Respectively:

15

4.2.1 SNN

Simple SNN for looking at spikes

A feedforward SNN with the following struture :
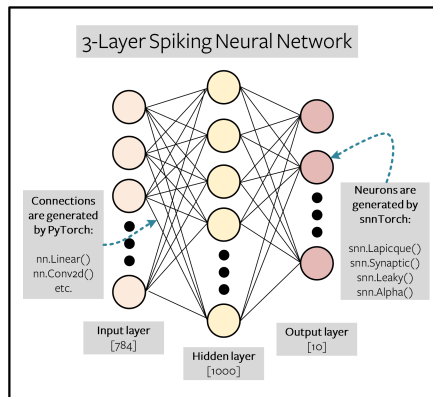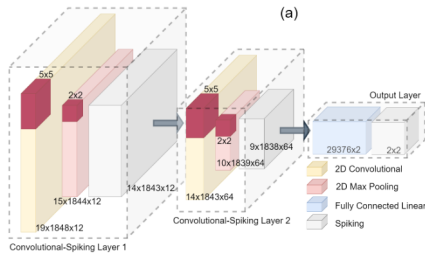

Figure 22: Simple SNN

```python
class SpikingNet(nn.Module):
    def __init__(self, num_inputs=784, num_hidden=1000, num_outputs=10, beta=0.99):
        super(SpikingNet, self).__init__()
        # Define the first fully connected layer
        self.fc1 = nn.Linear(num_inputs, num_hidden)
        # Define the first leaky integrate-and-fire neuron
        self.lif1 = snn.Leaky(beta=beta, spike_grad=spike_grad)
        # Define the second fully connected layer
        self.fc2 = nn.Linear(num_hidden, num_outputs)
        # Define the second leaky integrate-and-fire neuron
        self.lif2 = snn.Leaky(beta=beta, spike_grad=spike_grad)
        # Initialize the membrane potentials for the neurons
        self.mem1 = self.lif1.init_leaky()
        self.mem2 = self.lif2.init_leaky()

    def forward(self, spk_in):
        # Initialize lists to record membrane potentials and spikes
        mem2_rec = []
        spk1_rec = []
        spk2_rec = []
        # Loop over the input spikes
        for step in range(spk_in.size(0)):
            # Compute the post-synaptic current (input I) for the first layer
            cur1 = self.fc1(spk_in[step])
            # Compute the spikes and membrane potentials for the first layer
            spk1, self.mem1 = self.lif1(cur1, self.mem1)
            # Compute the post-synaptic current for the second layer
            cur2 = self.fc2(spk1)
            # Compute the spikes and membrane potentials for the second layer
            spk2, self.mem2 = self.lif2(cur2, self.mem2)
            # Record the membrane potentials and spikes
            mem2_rec.append(self.mem2)
            spk1_rec.append(spk1)
            spk2_rec.append(spk2)
        # Return the recorded membrane potentials and spikes
        return torch.stack(spk2_rec), torch.stack(mem2_rec), torch.stack(spk1_rec)
```

Figure 23: Simple SNN

CSNN

The above convolutionnal spiking neural network :

Figure 24: CSNN code

4.2.2 ANNs

To make some comparison, we also defined the same models but with the ANN architecture, and also tried some other kownn architectures for comparison. Respectively:

- The equivalent ANN for our CSNN model.

- RESNET50, RESNET101

- EFFICIENTNET-B0

- VGG16

- Densenet 121

- MobileNetV2

Please note that some parameters like the number of MFCC features may vary between the ANNs and the CSNN usage, due to memory usage issues.

17

4.3 Results on MNIST

When we worked with the MNIST dataset, we only worked with spiking neural networks.

Feedforward SNN

Firstly we issued some illustrations on what proportion of the input data (converted into spiking data) was transmitted, layer by layer. This gave us the bellow results, the fist line corresponds to the rate encoding of the input data, the second line corresponds to the spikes emitted by the first layer, and the last spike corresponds to the spikes emitted by the output layer (all stacked in an image).
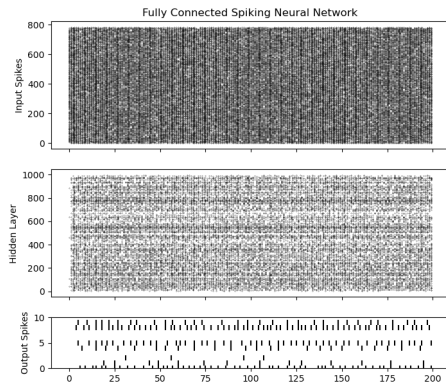

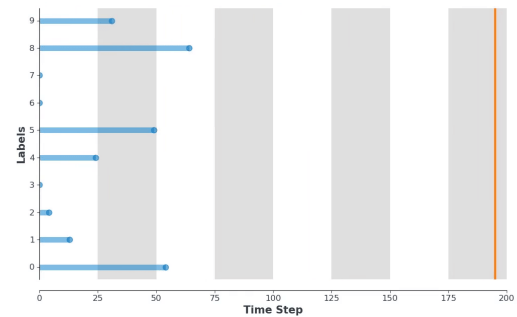Figure 25: Observation of spikes after rate encoding and after each LiF layers

Figure 26: Last image of a prediction of the SNN model on the MNIST dataset (not trained)

From top to bottom of figure 24, we observe the clear reduction of the number of spikes emitted. Also note that this model was not trained and was only used to observe the spikes.

Convolutional SNN

When training the CSNN model (having far less parameters than the previous one), we obtained the following results.
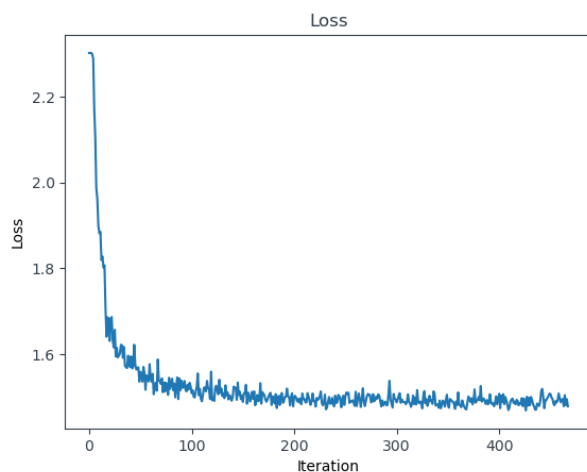

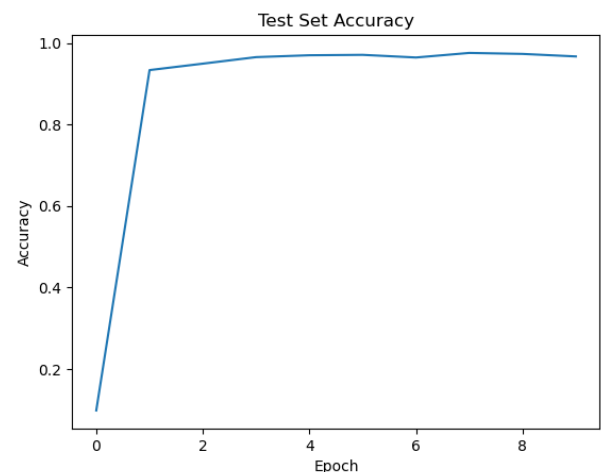Figure 27: Loss of the CSNN model on the MNIST dataset

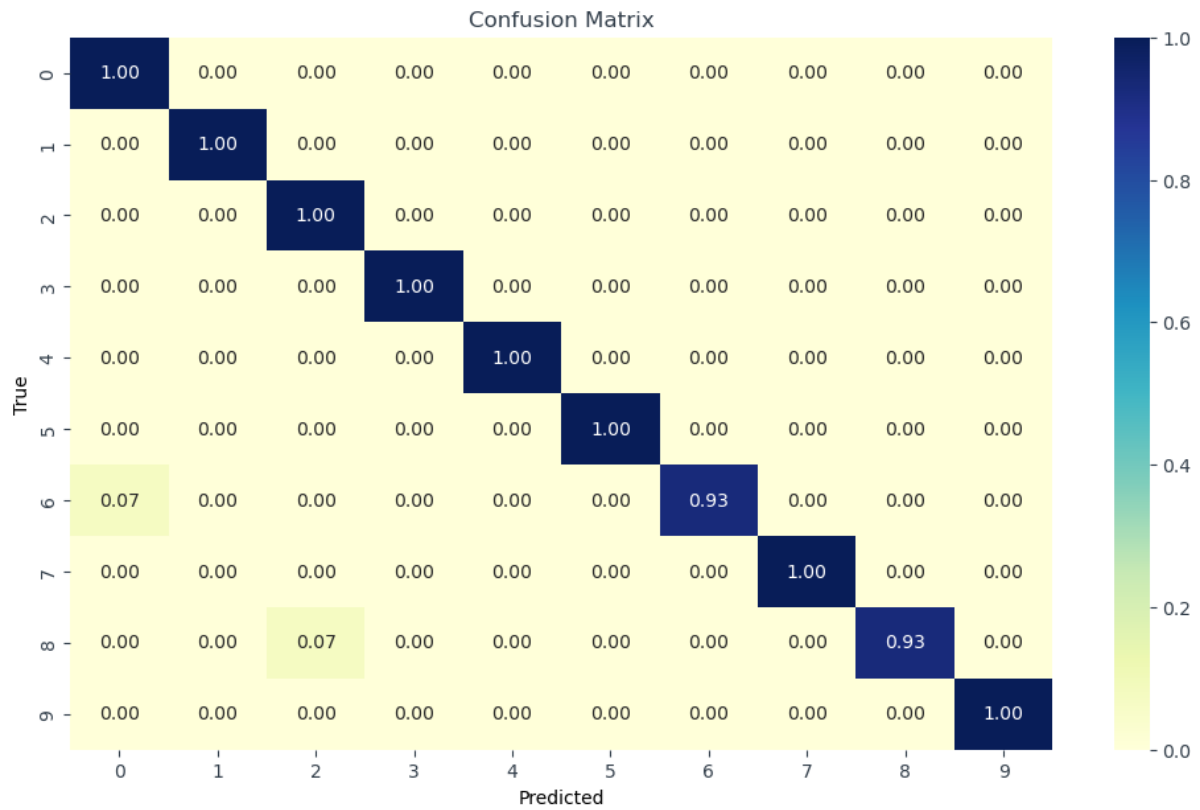Figure 28: Accuracy of the CSNN model on the MNIST dataset

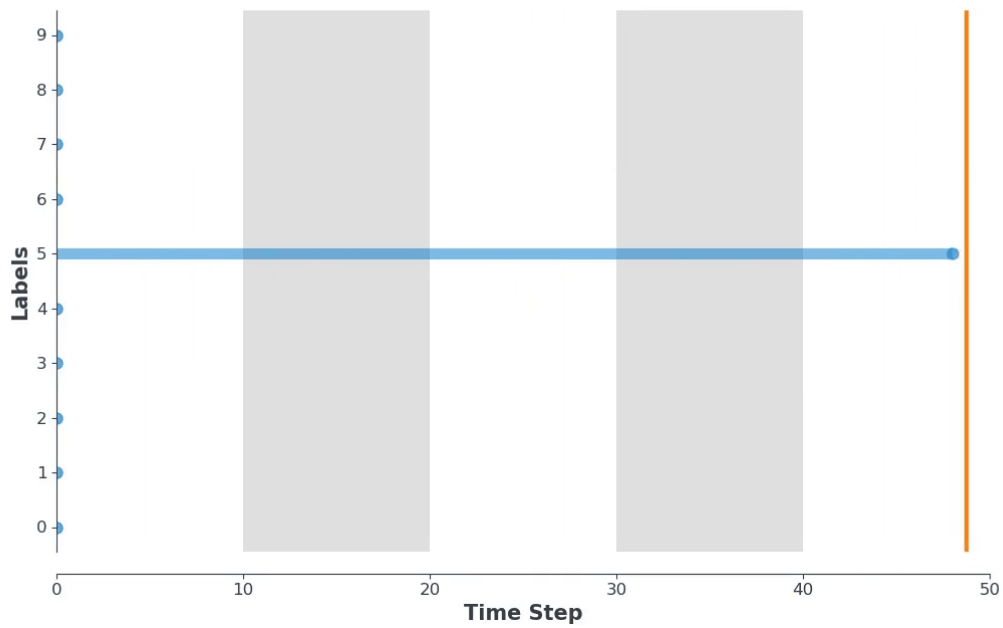Figure 29: Confusion matrix of the CSNN model on the MNIST dataset

Figure 30: Last image of a prediction of the CSNN model on the MNIST dataset

In this last image, we observe that the only neuron that fired for all time steps is the one

associated to the predicted class 5. We aim now to test the model on our test dataset and as similar results as possible (in term of efficiency).

4.4   Our Dataset

We used the Google AudioSet dataset as the basis for our audio classification task. The dataset is divided into 10-second audio clips, each labeled with one or multiple classes. The classes are organized in a hierarchical structure, with the top-level classes being broad categories like "Music" or "Speech", and the lower-level classes being more specific, like "Piano" or "Saxophone".

It is composed of csv files, each one containing some time codes and a link to a youtube video. In order to create an exploitable dataset out of this one, we used the following steps:

- Divide the dataset into 3 csv files, each one associated to a top-level class (Music, Animals and Sound of things)

- Make sure there are no repetitions in between those classes

- Filter the dataset so that we keep "recognizable" sounds

- Perform some post-processing to make the dataset exploitable (outdated links, duration smaller than 10 seconds, etc.)

So we end up with 3 class files, each one containing a list of youtube links and their associated class (with most of the time other "noise" classes that are not to be taken into account for the classification task on high level classes). We then used the youtube-dl library to download the videos (parallel operations) and the soc library to extract the audio from the videos with the correct duration and format.

We decided to use this dataset to perform some audio classification because, we though that, as the dataset is quite large and, as it is a hierarchical dataset and its categories are quite diverse, it would be a good dataset to test the capabilities of spiking convolutional neural networks. Furthermore, if the model we create performs well on this dataset, which is quite unbalanced and diverse, it would be a good sign that it could perform well on embedded systems, where the data has often this kind of characteristics.

4.4.1   Mel-frequency cepstral coefficients (MFCC)

After having imported the dataset, we needed to make it usable by the networks. We chose to convert it to MFCCs.

This approach is used in audio analysis to capture information specific to human auditory characteristics while reducing data redundancy. MFCCs thus encapsulate frequency variations over time in a compact way, producing a set of cepstral coefficients that are widely used for automatic speech recognition and other audio signal processing tasks.

Two of the main advantages of MFCCs are compactness and information discrimination. MFCCs condense information while preserving the signal's distinctive characteristics. Compact representation facilitates storage, transmission, and processing of large amounts of data. By focusing on perceptual features rather than raw frequency, MFCCs are less sensitive to pitch variations, improving the robustness of sound recognition.

Some python libraries like librosa have functions to convert directly audio in MFCC. These are the steps to convert an audio signal in MFCC:

- Take the signal's Short Term Fourier Transform (STFT), which divides the audio signal into multiple parts and compute the Fourier Transform for each one. In practice, we use the Discrete Fourier Transform.

$$STFT : \{x[n]\} \equiv X(m,\omega) = \sum_{n=-\infty}^{\infty} x[n]w[n-m]\mathrm{e}^{-j\omega n} \qquad (6)$$

- Map the powers of the resulting spectrum to the Mel scale, multiplying by overlapping window functions (triangular or cosine), then take the logarithm of the amplitudes at each mel frequency. The Mel scale is semi-logarithmic, to better represent actual human perception of frequencies.

$$Mel\ scale : \ m = 2595 \log_{10}\left(1 + \frac{f}{700}\right) \qquad (7)$$

- Take the discrete cosine transform (DCT) of the list of logarithmic powers of the mel frequencies, as if it were a signal. The MFCCs are the amplitudes of the resulting spectrum.

$$DCT : \ S_{u,v} = \sum_{m=0}^{M-1}\sum_{n=0}^{N-1} s(m,n)\mathrm{e}^{-2\mathrm{i}\pi\left(\frac{um}{M}+\frac{vn}{N}\right)} \qquad (8)$$
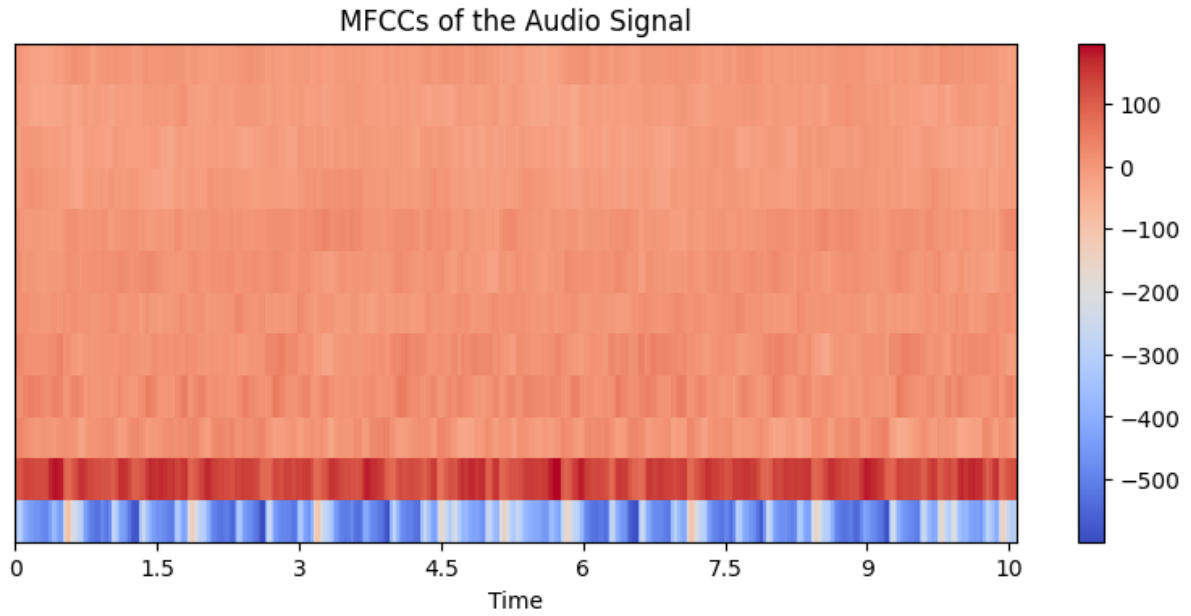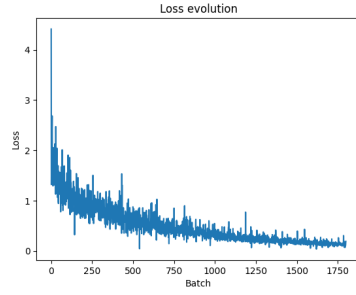

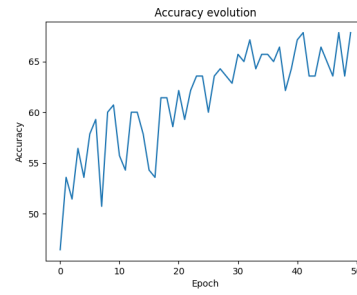Figure 31: MFCC matrix, each line corresponding to a coefficient
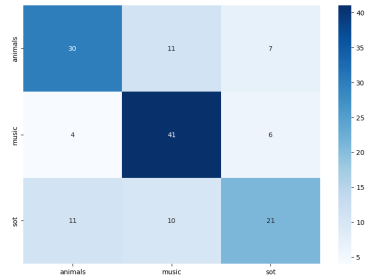
4.5   Results and discussion

ANNs

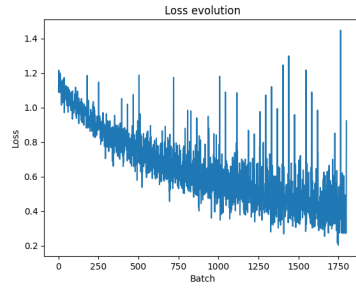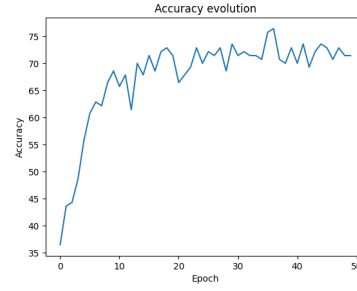We trained the ANNs on the dataset, and obtained the following results:
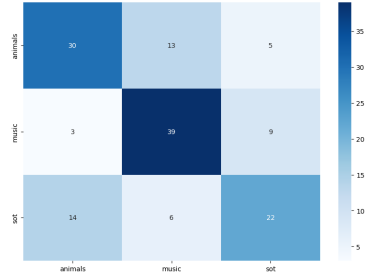
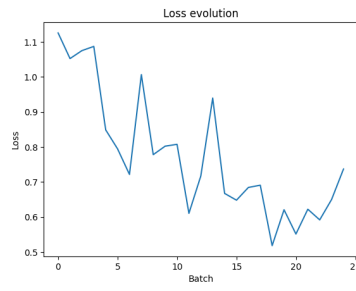(a) Loss CNN　　　(b) Accuracy CNN　　　(c) Confusion matrix CNN
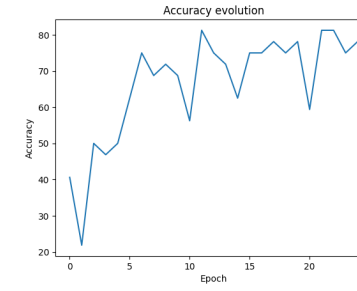
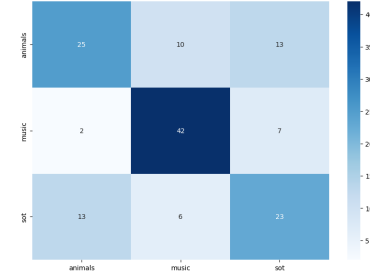(d) Loss RESNET50　　　(e) Accuracy RESNET50　　　(f) Confusion matrix RESNET50

(g) loss MobileNetV2　　　(h) Accuracy MobileNetV2　　　(i) Confusion matrix MobileNetV2

Figure 32: Results of the ANNs on the dataset (no data augmentation)

SNNs

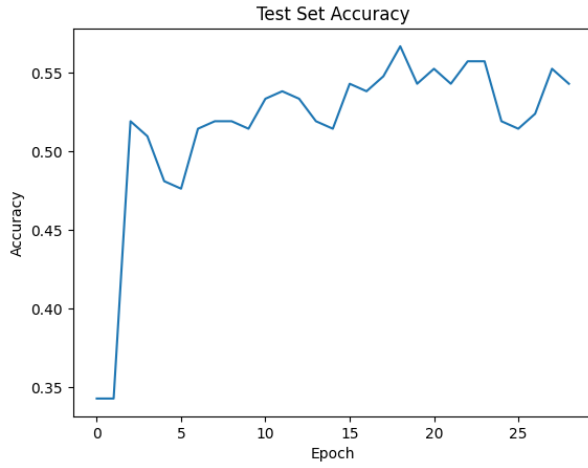After training the CSNN on the dataset, we obtained the following results:

Figure 33: Accuracy of the CSNN model on the dataset for 50 time step and 20 n_{mfcc}
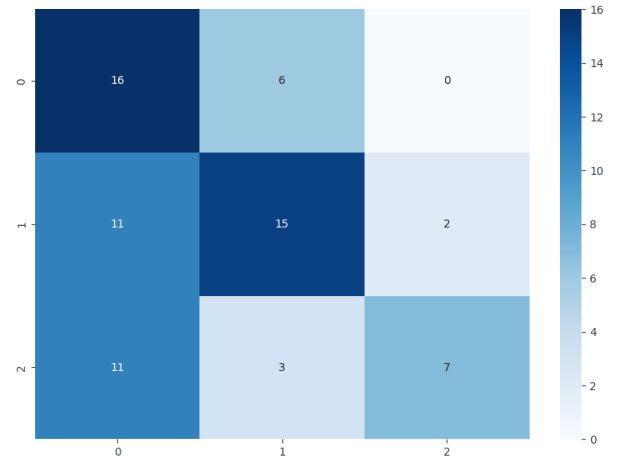

Figure 34: Confusion matrix of the CSNN model on the dataset for 50 time step and 20 n_{mfcc}

We observe a deacrease in the accuracy of the model (which was expected), and a more hesitant confusion matrix. This could be due to a lack of training. However, the stagnation of the accuracy over the epochs would suggest that or the model is not learning, or the model is learning but the dataset is too unbalanced to be able to learn anything.

By performing some data augmentation, we obtained an accuracy going from 60% to 72% (due to technical issues, we are not able to provide the corresponding confusion matrix).

We then tried working on elements such as the number of mfcc or the number of time steps as these parameters are the one that drains the most memory. It turns out that reducing the number of mfcc made the model significantly less efficient, and compensating by increasing the number of time steps did not improve the results as much as expected ($\approx 40 - 45\%$ accuracy).

5   Conclusion

For the audio dataset, results are slightly better with ANNs, we made some inferences on the explanation. It may be more difficult to estimate real gradient accurately with surrogate gradient, we may need more time steps to effectively predict, the networks may not be resilient enough to perturbations. Finally, the architecture of our network might not be good enough for this task.

Despite a long training and worse results than with the ANNs tested, SNNs proved themselves as an effective tool for a classifying task. Moreover, their main strength resides in the sparsity of the information passed through the network which can promote a continued use on embedded devices for example, but can be extended to every part of deep learning because SNN a way less computationally expensive to run. For example, the open-sourced code for the largest language-generating SNN ever, named SpikeGPT, which uses 22 times less energy than a similar model using typical deep learning, was released in 2023. [5]

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).

[5] R.-J. Zhu, Q. Zhao, G. Li, and J. K. Eshraghian, *Spikegpt: Generative pre-trained language model with spiking neural networks*, arXiv preprint arXiv:2302.13939, (2023).