Projet de fin d'etudes : Video coding using Spiking Neural Networks

Alois TURUANI, Morgan Briancon, MAM5

January 2020

Chapter 1

Introduction

As technology improves, the need for finding better ways for the transmission and storage of information augments dramatically, During the last years the progress of video compression algorithms has become very challenging since the improvement of the already existing standards seems to be very difficult and time consuming. All of the current compression algorithms follow almost the same paradigm. At the moment they consider video as a sequence of pictures with high temporal redundancy, and that is why current solution are using motion estimation as an important feature. Those coding scheme does not account for the actual biological visual system behavior. It is our belief that the mammalian's visual system developed efficient coding strategies that could be used as a source of inspiration to imagine a different kind of compression algorithms. A great example of the neural coding activity is the one performed by the ganglion cells for the encoding of the visual information. The Leaky Intergrate and Fire(LIF) neural model seems to be one of the most promising neural model for quantization. The LIF model is based on the exact time each neuron emits its spike. This time carries all the necessary information about the intensity of the input. The higher the input intensity is, the more spike will be emited during a time period. If we assume that a neuron is inhibited just after the release of its spike, then for a given observation window, a high intensity signal will generate far more spike than a low intensity signal. In this project, we aim to introduce a new python library to help further research about the LIF quantizer.
This research project was made from November 2019 to February 2020 in collaboration with Marc Antonini and Jean Martinet.

Chapter 2

Overview of the state-of-the-art in compression :

2.1 The coding Principle

The coding principle is the most common architecture in lossy compression. It has been adapted by all the current lossy compression algorithms and it describes the encoding and decoding process of an input signal. The input signal f could be either an audio, image or video signal. This signal will be transformed into a more compressible format. To achieve this goal, compression algorithm have been using different transforms such as the Direct Wavelet Transform(DWT), Direct Cosine Transform(DCT) and Fourier. The transformed data is then identified by a quantizer in order to decide which information is to be removed. The quantization step is solely responsible to introduce distortion. That is why in lossless compression there is no quantization step. The last step is the Entropy coding, it's an lossless function which translates the quantized intensity of the signal into codewords, whose length vary inversely to the frequency of occurrence. Once the input signal has been through all those step, it can be saved or sent through the communication channel to the receiver who need to use this code in order to reconstruct the input signal. This is the decoding process which consists of the entropy decoding, the de-quantization and the inverse transform.

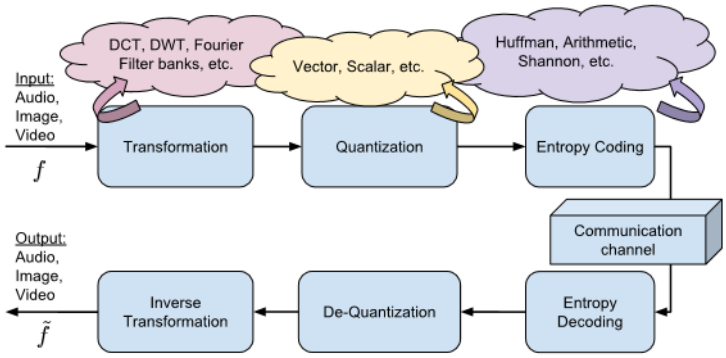


Figure 2.1: The Coding pinciple

2.2 Qualitative Metrics

There are many metrics that are used to quantified the quality of the reconstructed signal with respect to the original one. In compression, it is desired to achieve the lowest distortion as possible while discarding the most information as possible. The distortion is usually defined as $D(f, \tilde{f})$ where f is the input signal and f is the reconstructed signal as described in fig 2.1

2.2.1 Mean squared error (MSE)

The most common metric is the mean squared error(MSE) which is defined by :

$$MSE(f, \tilde{f}) = \frac{1}{n} \sum_{i=1}^{n} (f_i - f_i)^2 \tag{2.1}$$

where n is the size of the input signal, $f = (f_1, \ldots, f_n)$ and $\tilde{f} = (\tilde{f}_1, \ldots, \tilde{f}_n)$. The Distortion is minimized when the MSE approaches 0. The MSE is poorly correlated to human perception. Nonetheless models can be optimized to minimize the MSE while getting excellent results in terms of perceptual quality as well.

2.2.2 Peak Signal to Noise Ratio (PSNR)

PSNR (Peak Signal to Noise Ratio) is defined by:

$$PSNR(f, \tilde{f}) = \log_{10} \frac{(2^b - 1)^2}{MSE(f, \tilde{f})} \tag{2.2}$$

where b is the number of bpp (bits per pixel). The PSNR is exprimed in decibels(dB). The PSNR is inversly proportional to the MSE, it means that if the MSE is close to zero, the PSNR will be approache infinity. It means that high value of PSNR provide a better image quality. A small value of PSNR means that there is a high numerical difference between the input and the output.

2.2.3 Structure SIMilarities

SSIM is used for measuring the similarity between two images. It is defined by:

$$SSIM(f, \tilde{f}) = l(f, \tilde{f})c(f, \tilde{f})s(f, \tilde{f}) \tag{2.3}$$

where

$$l(f, f) = \frac{2\mu_f \mu_{\tilde{f}} + c_1}{\mu_f^2 + \mu_{\tilde{f}}^2 + c_1} \tag{2.4}$$

$$c(f, \tilde{f}) = \frac{2\sigma_f \sigma_{\tilde{f}} + c_2}{\sigma_f^2 + \sigma_{\tilde{f}}^2 + c_2} \tag{2.5}$$

$$s(f, \tilde{f}) = \frac{\sigma_{f,\tilde{f}} + c_3}{\sigma_f \sigma_{\tilde{f}} + c_3} \tag{2.6}$$

where μ_f is the average of f, $\mu_{\tilde{f}}$ the average of $\tilde{f}$, σ_f^2 is the variance of f, $\sigma_{\tilde{f}}^2$ the variance of $\tilde{f}$, $\sigma_{f,\tilde{f}}^2$ the covariance of f and $\tilde{f}$,$c_1 = k_1 L^2$,$c_2 = k_2 L^2$ and $c_3 = c_2/2$ are three positive variables to stabilize the division with weak denomitor, L is the dynamic range of the pixel values and $k_1 = 0.01$, $k_2 = 0.03$ by default. The output of the SSIM is range between 0 and 1. The difference with respect to other techniques mentioned previously such as MSE or PSNR is that these approaches estimate absolute errors; on the other hand, SSIM is a perception-based model that considers image degradation as perceived change in structural information, while also incorporating important perceptual phenomena, including both luminance masking and contrast masking terms [7].

2.3 Shannon Entropy

A quantity known as "entropy" is defined in terms of the statistical properties of the information source. In other words the entropy is the mesure of the unpredictability of the state, or equivalently, of its average information content. The entropy rate of a data source means the average number of bits per symbol needed to encode it. Given an input S there are random symbols $s_1, s_2, \ldots, s_n$. Each one of those symbols i has a probability p_i to occur, then the shannon entropy of its source S is defined by :

$$H(S) = -\sum_{i=1}^{n} p_i \log_b p_i \tag{2.7}$$

where b is the base of the logarithm used. Common values of b are 2, Euler's number exp and 10. The corresponding units of entrpy are the bits for $b = 2$, nats for $b = \exp$ and bans for $b = 10$

Chapter 3

Presentation of the solutions

A lot of different approach exists when it comes to designing biological neuron model In this section we will describe different types of neuron model.

3.1 Neuron Models

3.1.1 Leaky Integrate and Fire Model :

In the LIF model [2], we assume that the information is hidden in each individual spike(the arrival time andor the interspike interval) is sufficient to describe the input stimulus. The fastest a spike is emitted, the strongest the input signal.

The LIF is a neural model which can be described by the following relation :

$$I(t) = \frac{V(t)}{R} + C\frac{dV}{dt} \tag{3.1}$$

where I(t) is the input current, C the membrane capacitor of a neuron which is in parallel with the resistor R and V(t) is the voltage across the resistor. By multipying (1) by R and by introducing a time constant $\tau_m = RC$ the equation becomes :

$$\tau_m \frac{du}{dt} = -u(t) + RI(t) \tag{3.2}$$

In the integrate-and-fire model, spikes are generated at a firing time $t^{(f)}$. The firing time is defined by the following threshold criterion:

$$t^f : u(t^{(f)}) = \theta. \tag{3.3}$$

Immediately after the spike, the potential is set to a given value $u_r < \theta$. Since spikes are stereotyped events, i.e. with nearly identical shapes, they are fully characterized by their firing time. Let's assume that LIF is applied to a given constant input current $I(t) = I_0$. We will assume that the reset potential $u_r = 0$. The LIF will set a threshold θ which is the criterion to decide if the neuron will spike or not. If the membrane potential exits the threshold, the neuron will spike otherwise, it will remain inactive. The LIF generates a spike train that encodes the input signal $I(t)$. Assuming that the first spike arrives at time t^1, the trajectory of the membrane potential can

be found by integrating eq (1.2) with the initial condition $u(t^1) = u_r = 0$. The solution is given by the relation::

$$u(t) = RI_0[1 - exp(-\frac{t - t^1}{\tau_m})] \tag{3.4}$$

The asymptotic value RI_0 in eq (1.5) determines the generation of spikes. If $RI_0 \leq \theta$ there is no spike, otherwise a spike will be fired. After each spike, the potential is reset to the value $u_r = 0$ and the integration process starts again. The condition $u(t^2) = \theta$ is satisfied when the next spike occurs.

$$\theta = RI_0[1 - exp(-\frac{t^2 - t^1}{\tau_m})] \tag{3.5}$$

Now we define $d(u)$ the delay between two spikes. There is two cases for $d(u)$, we are in the case of a LIF with no refractory period we have the following definition for $d(u) = t^2 - t^1$, we can use the equation (2.5) to find:

$$d(u) = \begin{cases} \infty & u < 0 \\ h(u;\theta) = \tau \log \frac{u}{u-\theta} & u \geq 0 \end{cases} \tag{3.6}$$

In the case of a LIF with refractory period we have $d'(u) = t^2 - t^1 + \Delta^{abs}$ where Δ^{abs} is the duration of the refractory period

3.1.2 QIF Model:

The QIF (Quadratic-Integrate-and-Fire) Model is a variant of the LIF Model where:

$$\tau_m \frac{du}{dt} = -u(t)^2 + RI(t) \tag{3.7}$$

This model has the avantage to be accurate and still computationally efficient. The QIF neurons can show some dynamic properties like delayed spiking.

To implement this model we need the threshold crossing value who give us the moments of the spike and also a reset value so that when the solution reach the threshold and spike, the solution is immediately reset to the reset value.

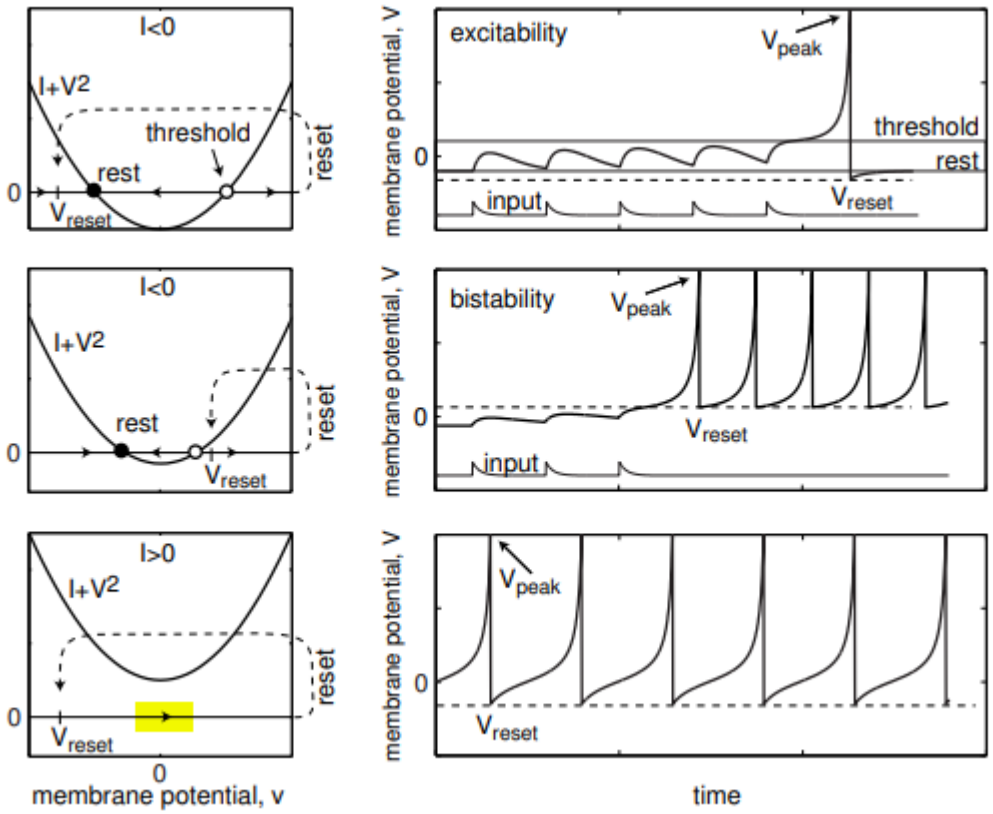


Figure 3.1: Simulation of the QIF for some of his features with time-dependent input

3.1.3 Theta Neuron Model

The theta neuron model is a canonical model. We obtain the theta neuron model by the following transformation of the potential u to phase θ

$$u(t) = \tan(\frac{\theta(t)}{2}) \tag{3.8}$$

we obtain a spike when the phase passes π. In the following figure we see the differents phase of the Theta Neuron Model.

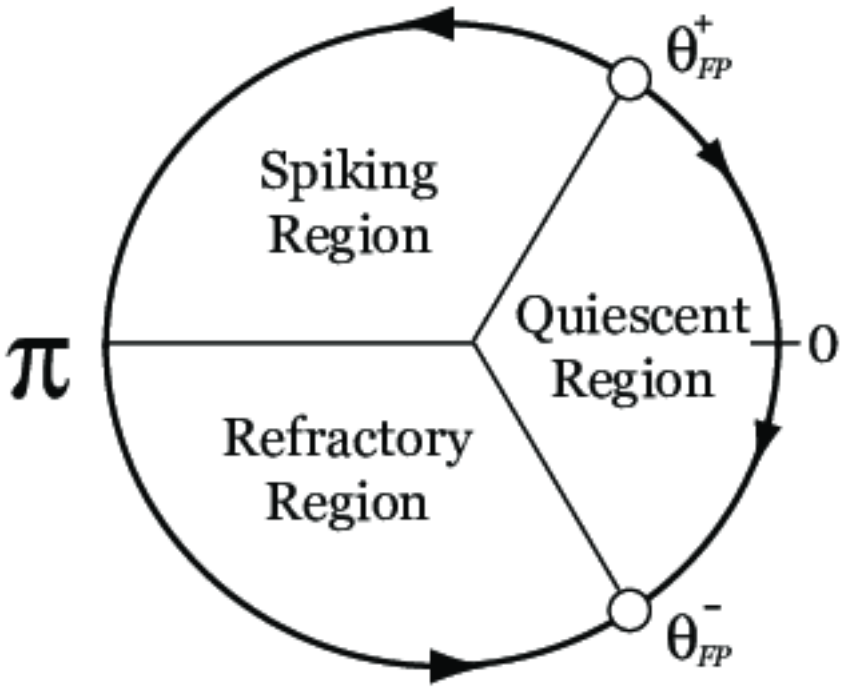


Figure 3.2: Phase circle of the Theta neuron model where the baseline current $I(t) < 0$

For the trajectory of the phase we have the following relation :

$$\tau \frac{d\theta}{dt} = (1 - \cos\theta) + \alpha I(t)(1 + \cos\theta) \tag{3.9}$$

where τ is the neuron phase constant, α is a scaling constant and $I(t)$ is the input current.This model allow us to do some more advanced gradient approaches because the spikes are described in a continuous manner.

3.1.4 How to interpret the spikes?

Rate Codes

The output of a spiking neuron is its firing rate. However, the time each neuron produce it's spike train for a given input signal is irregular. This irregularity might arise from some stochastic forces [3]. One way to compute the firing rate for a given input stimulus is to use the Michaelis-Menten function.

3.2 Quantizer

3.2.1 The LIF Quantizer

The encoder

As described in the LIF section, the LIF neuron produces a number of spike for a given input signal. This means we can map each number of spike to a input intensity. For each input sample, the membrane potential is computed according to the Ohm's law. The integration delay $d(u)$ is calculated by the equation (3.4) according the the threshold value θ. If there is a refractory period Δ^{abs} it is added to the interspike delay. Then using the relation $N_s = \left\lfloor \frac{t_{obs}}{d'(u)} \right\rfloor$ we get the number of spikes produced. The output of our model will be a number of spikes for each input sample. This is the data produced by the encoder.

The decoder

When we have our encoded signal, the data is processed by the decoder, in order to reconstruct the original signal. In [1] it has been proven that an estimation of the firing period $\tilde{d}$ can be found using$\tilde{d}(u) = \frac{t_{obs}}{N_s}$. and then we use the inverse function of the equation 2.9:

$$\bar{u} = \begin{cases} 0 & \bar{d}(u) = \infty \\ h^{-1}(\bar{d}(u); \theta) = \frac{\theta}{1 - \exp(-\frac{\bar{d}}{\tau}} & \bar{d}(u) < \infty \end{cases} \tag{3.10}$$

We can find an approximation of the input current value with the equation:

$$\bar{I} = \frac{\bar{u}}{R} \tag{3.11}$$

where R is the resistance.

Chapter 4

Work Done:

During this project we've been able to develop a tool to compute the LIF quantizer in both 1D, 2D, 3D. All code is available at https://github.com/aloaah/LIFQ/ it will be updated in the next few days to be more user friendly are more modular.

4.0.1 LIF Quantizer in 1D

```
import brian2 as b2
import matplotlib.pyplot as plt
import numpy as np

# Value used to define the parameters of the LIF
V_REST = 0 * b2.mV
V_RESET = 0 * b2.mV
FIRING_THRESHOLD = 0.09 * b2.mV
MEMBRANE_RESISTANCE = 550 * b2.mohm
MEMBRANE_TIME_SCALE = 7 * b2.ms
ABSOLUTE_REFRACTORY_PERIOD = 0 * b2.ms
SIMULATION_TIME = 66* b2.ms



def simulate_LIF_neuron(input_current,
                        N,
                        simulation_time=SIMULATION_TIME,
                        v_rest=V_REST,
                        v_reset=V_RESET,
                        firing_threshold=FIRING_THRESHOLD,
                        membrane_resistance=MEMBRANE_RESISTANCE,
                        membrane_time_scale=MEMBRANE_TIME_SCALE,
                        abs_refractory_period=ABSOLUTE_REFRACTORY_PERIOD):
    # differential equation of Leaky Integrate-and-Fire model
    eqs = """
    dv/dt =
    ( -(v-v_rest) + membrane_resistance * input_current(t, i) ) /
                                            membrane_time_scale : volt (unless
                                            refractory)"""
```

```
    # LIF neuron using Brian2 library
    neuron = b2.NeuronGroup(
        N, model=eqs, reset="v=v_reset", threshold="v>firing_threshold",
        refractory=abs_refractory_period, method="euler")
    neuron.v = v_rest  # set initial value

    # monitoring membrane potential of neuron and injecting current
    state_monitor = b2.StateMonitor(neuron, ["v"], record=True)
    spike_monitor = b2.SpikeMonitor(neuron)
    # run the simulation
    b2.run(simulation_time)
    return state_monitor, spike_monitor


#Rewrite the input signal  into the type for brian2 neuron models
def create_time_matrix1D(matrix, time, simulation_time):
    big_matrix = np.empty(np.int64(np.ceil(simulation_time/time)), )
    for i in range(len(matrix)):
        temp = np.hstack(c for c in np.full((np.int64(np.ceil(simulation_time/time))
                                                , 1), matrix[i]))
        big_matrix = np.vstack((big_matrix, temp))
    big_matrix = np.delete(big_matrix, 0, 0)
    return b2.TimedArray(np.transpose(big_matrix) * b2.mA, dt = time)



# Reconstruction of the input from the output of the model
def reconstr_array(spike_count, FIRING_THRESHOLD, MEMBRANE_TIME_SCALE,
                                        MEMBRANE_RESISTANCE, SIMULATION_TIME) :
  dict_u_hat = dict()
    for values in np.unique(spike_count):
        if values == 0:
            dict_u_hat[values] = 0
        else :
            d_u_hat = SIMULATION_TIME/values
            dict_u_hat[values] = (((FIRING_THRESHOLD/(1-np.exp(-(d_u_hat/
                                                    MEMBRANE_TIME_SCALE))))*(1/
                                                    MEMBRANE_RESISTANCE)) /b2.mA)
    reconstr_array = np.ndarray((n,1))
    for i in range(len(spike_count)):
        reconstr_array[i] = dict_u_hat[spike_count[i]]



def compute_for_our_signal():


    signal = our_signal()
    n = len(signal)
    b2.BrianLogger.log_level_debug()
    matrix = create_time_matrix1D(signal, SIMULATION_TIME, SIMULATION_TIME)
    state, spike = simulate_LIF_neuron(matrix, n)
    reconstr_array = reconstr_array(spike.count, FIRING_THRESHOLD,
                                            MEMBRANE_TIME_SCALE,
                                            MEMBRANE_RESISTANCE, SIMULATION_TIME)
```

```
    plt.title("Caracteristic function of  quantizer")
    plt.plot(signal, reconstr_array)
    plt.show()
```

This code allows you to compute the LIF quantizer for a given input signal in 1D, and reconstruct the input signal as explained in 3.2.1.As of now, it will plot the characteristic function of the quantizer based on the different model state variables. It is based on the brian2 simulator for spiking neural network [6].

4.0.2 LIF Quantizer in 2D

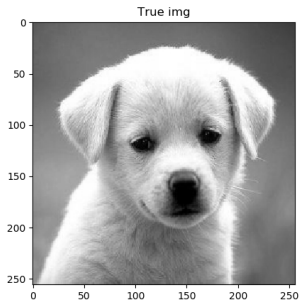
```
import brian2 as b2
import matplotlib.pyplot as plt
from IPython.display import display
import numpy as np
from skimage.measure import compare_psnr, compare_ssim
import cv2

V_REST = 0 * b2.mV
V_RESET = 0 * b2.mV

FIRING_THRESHOLD = 0.09 * b2.mV

MEMBRANE_RESISTANCE = 550 * b2.mohm

MEMBRANE_TIME_SCALE = 7 * b2.ms

ABSOLUTE_REFRACTORY_PERIOD = 0 * b2.ms

SIMULATION_TIME = 66* b2.ms


def simulate_LIF_neuron(input_current,
                        N,
                        simulation_time=SIMULATION_TIME,
                        v_rest=V_REST,
                        v_reset=V_RESET,
                        firing_threshold=FIRING_THRESHOLD,
                        membrane_resistance=MEMBRANE_RESISTANCE,
                        membrane_time_scale=MEMBRANE_TIME_SCALE,
                        abs_refractory_period=ABSOLUTE_REFRACTORY_PERIOD):
    # differential equation of Leaky Integrate-and-Fire model
    eqs = """
    dv/dt =
    ( -(v-v_rest) + membrane_resistance * input_current(t, i) ) /
                                            membrane_time_scale : volt (unless
                                            refractory)"""


    # LIF neuron using Brian2 library
    neuron = b2.NeuronGroup(
        N, model=eqs, reset="v=v_reset", threshold="v>firing_threshold",
        refractory=abs_refractory_period, method="euler")
    neuron.v = v_rest  # set initial value

    # monitoring membrane potential of neuron and injecting current
```

```
    state_monitor = b2.StateMonitor(neuron, ["v"], record=True)
    spike_monitor = b2.SpikeMonitor(neuron)
    # run the simulation
    b2.run(simulation_time)
    return state_monitor, spike_monitor

def create_time_matrix(matrix, time, simulation_time):
    big_matrix = np.empty(np.int64(np.ceil(simulation_time/time)), )
    for i in range(len(matrix)):
        for j in range(len(matrix[i])):
            temp = np.hstack(c for c in np.full((np.int64(np.ceil(simulation_time/
                                                  time)), 1), matrix[i][j]/255 )
                                                  )
            big_matrix = np.vstack((big_matrix, temp))
    big_matrix = np.delete(big_matrix, 0, 0)
    return b2.TimedArray(np.transpose(big_matrix) * b2.mA, dt = time)

def create_timed_array(value, time, simulation_time):
    return b2.TimedArray(np.hstack(c for c in np.full((np.int64(np.ceil(
                                          simulation_time/time)), 1), value ))*
                                          b2.mA, dt = time)




def plot_multi_spike(spike_mon, nb_neuron, time):
    b2.plot(spike_mon.t/b2.ms, spike_mon.i, '.', ms=2)
    b2.xlim(0, time)
    b2.ylim(0, nb_neuron)
    b2.xlabel('Time (ms)')
    b2.ylabel('Neuron index');
    b2.show()

def compute_entropy(spike_count, test):
    entropy = 0
    for i in np.unique(spike_count, return_index = True)[1]:
        entropy += proba(i,  spike_count)*np.log(proba(i, spike_count))
    return -entropy

def proba(i, spike_count):
    unique, counts = np.unique(spike_count, return_counts=True)
    dict_temp = dict(zip(unique, counts))
    return dict_temp[spike_count[i]]/len(spike_count)


def decode2D(spike_count, test):
  dict_u_hat = dict()
  for values in np.unique(spike_count):
    if values == 0:
      dict_u_hat[values] = 0
    else :
      d_u_hat = SIMULATION_TIME/values
      dict_u_hat[values] = (((FIRING_THRESHOLD/(1-np.exp(-(d_u_hat/
                                          MEMBRANE_TIME_SCALE))))*(1/
                                          MEMBRANE_RESISTANCE)) /b2.mA)*255
  reconstr_array = np.ndarray((len(test),len(test[0])))
  for i in range(len(spike_count)):
```

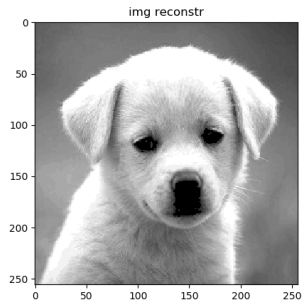
```
    reconstr_array[np.int64(np.floor(i/len(test)))][i%len(test[0])] = np.int64(np.
                                        floor(dict_u_hat[spike_count[i]]))

  return reconstr_array



def compute_and_print_mesure(true_img, reconstr_img):
    psnr_value = compare_psnr(true_img,  reconstr_img)
    print("PSNR : {}".format(psnr_value))
    mssim, grad, S = compare_ssim(true_img, reconstr_img, gradient=True, full=True)
    print("SSIM : {}".format(mssim))
    plt.title("Gradient SSIM")
    plt.imshow(grad)
    plt.show()
    plt.title("SSIM Image")
    plt.imshow(S, cmap='gray')
    plt.show()

def show_img(img, title, map_):
    plt.title(title)
    plt.imshow(img, cmap=map_)
    plt.show()


def compute_quantization():
    img_dog = plt.imread("dogg.jpeg")
    img_1_chan = img_dog[:, :, 0]

    N = len(img_1_chan[0])*len(img_1_chan)

    b2.BrianLogger.log_level_debug()

    ta_matrix = create_time_matrix(img_1_chan, SIMULATION_TIME, SIMULATION_TIME)

    state, spike = simulate_LIF_neuron(ta_matrix, N)


    print("Entropy : {}".format(compute_entropy(spike.count,img_1_chan)))

    reconstr_IMG = decode2D(spike.count, img_1_chan)

    show_img(img_1_chan,  "True img", "gray")
    show_img(reconstr_IMG, "img reconstr", "gray")

    compute_and_print_mesure(img_1_chan, reconstr_IMG)

compute_quantization()
```

This code encode and decode a given image based on the LIF quantizer, it also compute the SSIM, PSNR and entropy for this image. It will start by scaling the input intensity to $[0, 1]$ a

An example of results we get is :



(a) Input image



(b) Reconstructed image

(c) SSIM Image

Figure 4.1: SSIM = 0.9643, PNSR = 32.06 dB

For this image we used as parameters for the LIF neuron : $\theta = 0.09mV$, $R = 550m\Omega$, $\tau = 7ms$, $\Delta = 0$ and $t_{obs} = 66ms$

4.0.3 LIF Quantizer in 3D

Our implementation of the LIF Quantizer in 3D is simply to take every frame as an image and apply a the 2D. For now, it will resize video to a (n*n) size where $n = \min(height * width)$ and it will convert the image to grayscale, for the grayscale parti it will be change soon, since we only have to consider each channel of the image as a grayscale image and then reunite them.

```
import brian2 as b2
import matplotlib.pyplot as plt
from IPython.display import display
import numpy as np
from skimage.measure import compare_psnr, compare_ssim
import cv2
import sys

def read_video_and_get_frame(path_to_video):
    vid = cv2.VideoCapture(path_to_video)
```

```
    video_frame = []

    i = 0
    #tant que la vid[U+FFFD]n'est pas fini
    while(vid.isOpened()):
        ret, frame = vid.read()
        if ret:
            min_dim = min(frame.shape[0], frame.shape[1])
            resize_frame = cv2.resize(frame , (min_dim, min_dim), interpolation =
                                                        cv2.INTER_LINEAR)
            video_frame.append(cv2.cvtColor(resize_frame, cv2.COLOR_BGR2GRAY))
        else:
            break

    vid.release()
    cv2.destroyAllWindows()

    video_frame = np.asarray(video_frame, dtype = np.int64)
    return video_frame

def reformat_video_frame_matrix(video_frame, time):
    reformated_matrix = []
    for frame in video_frame:
        reformated_matrix.append(create_time_matrix(frame, time/2, time))
    np.save('video_frame.npy', np.asarray(reformated_matrix, dtype= np.float64))
    return reformated_matrix

def create_time_matrix(matrix, time, simulation_time):
    big_matrix = np.empty(np.int64(np.ceil(simulation_time/time)), )
    for i in range(len(matrix)):
        for j in range(len(matrix[i])):
            temp = np.hstack(c for c in np.full((np.int64(np.ceil(simulation_time/
                                                        time)), 1), matrix[i][j]/255 )
                                                        )
            big_matrix = np.vstack((big_matrix, temp))
    big_matrix = np.delete(big_matrix, 0, 0)
    return big_matrix




video_frame = read_video_and_get_frame("xylophone.mp4")

reformated_matrix = reformat_video_frame_matrix(video_frame, 66* b2.ms)
reformated_matrix = np.load('video_frame.npy')



N = len(video_frame[0])*len(video_frame[0][0])


SIMULATION_TIME= 66 *b2.ms
v_rest = 0 * b2.mV
v_reset = 0 * b2.mV
firing_threshold = 0.09 * b2.mV
membrane_resistance = 550 * b2.mohm
```

```
membrane_time_scale = 7 * b2.ms
abs_refractory_period = 0 * b2.ms
n = len(video_frame[0]) * len(video_frame[0][0])


output_rates = []
time_range = []

for i in range(0, 100):
    time_range.append((1 + i * 66) * b2.ms)


# Construct the network just once
# LIF neuron using Brian2 library
eqs = """
    dv/dt =
    ( -(v-v_rest) + membrane_resistance * input_current(t, i) ) /
                                            membrane_time_scale : volt (unless
                                            refractory)"""


neuron = b2.NeuronGroup(
    N, model=eqs, reset="v=v_reset", threshold="v>firing_threshold",
    refractory=abs_refractory_period, method="euler")
neuron.v = v_rest  # set initial value

spike_monitor = b2.SpikeMonitor(neuron)
# Store the current state of the network
b2.store()
test = b2.TimedArray(np.transpose(reformated_matrix[0]) * b2.mA, dt =
                                          SIMULATION_TIME/2)
for frame in reformated_matrix:
    #Load the new frame
    input_current = b2.TimedArray(np.transpose(frame) * b2.mA, dt = SIMULATION_TIME/
                                            2)
    #print(spike_monitor.count)

    # Restore the original state of the network
    b2.restore()
    # Run it with the new frame
    b2.run(SIMULATION_TIME)
    output_rates.append(np.asarray(spike_monitor.count))


print(output_rates[0])
print(output_rates[1])
print(output_rates[2])


def decode_all(output_spike_count, video_frame):
    video_recons = []
    for spike_count in output_spike_count:

        video_recons.append(decode_single_frame(spike_count, video_frame))

    return video_recons
```

```
def decode_single_frame(spike_count, video_frame):

    dict_u_hat = dict()
    for values in np.unique(spike_count):
        if values == 0:
            dict_u_hat[values] = 0
        else :
            d_u_hat = SIMULATION_TIME/values
            dict_u_hat[values] = (((firing_threshold/(1-np.exp(-(d_u_hat/
                                                    membrane_time_scale))))*(1/
                                                    membrane_resistance)) /b2.mA)*
                                                    255
    reconstr_array = np.ndarray((len(video_frame[0]),len(video_frame[0][0])))
    for i in range(len(spike_count)):
        reconstr_array[np.int64(np.floor(i/len(video_frame[0])))][i%len(video_frame[
                                                0][0])] = np.int64(np.floor(
                                                dict_u_hat[spike_count[i]]))
    return reconstr_array
video_recons = decode_all(output_rates, video_frame)

def create_video(video_recons, fps, path_to_video):
    fourcc = cv2.VideoWriter_fourcc(*'MP42')
    video_writer = cv2.VideoWriter(path_to_video + '.avi', fourcc, float(fps), (len(
                                            video_recons[0]), len(video_recons[0][
                                            0])))

    for frame in video_recons:
        frame = np.asarray(frame, dtype = np.uint8)
        frame_3c = cv2.merge([frame, frame, frame])
        video_writer.write(frame_3c)

    video_writer.release()



plt.title("Decoded frame")
plt.imshow(video_recons[0], cmap  = 'gray')
plt.show()


plt.title("True frame")
plt.imshow(video_frame[0], cmap  = 'gray')
plt.show()

create_video(video_recons, 15, "video_reconstructed")
```

Chapter 5

Positioning of the solution

Spiking Neural Networks have some advantages compared to other neural networks, they are universal for digital computation, they introduce an explicit notion of time by comparing rates instead of fires and the interconnections transmit bits instead of numbers and they are computationnally powerful. We can also have a first approximate outpal of the final layer just after we have the first spike even in multi-layer networks due to the fact that spikes are propagated immediately to higher layers when we have sufficient activity. Spiking neural networks also improve their performances at each spikes.

There is also some limitations, they do not perform as well as the others neural networks on existing database, this could come from the conversion of the images into spike trains that can lose some information. Also the training algorithms are limited so we can apply spiking neural networks only on specifics cases. The fact that we compute in an asynchronous and a discontinuous way make the use of back-propagation difficult.

5.0.1 Use of spiking neural networks

Spiking neural networks can be use in many fields like robotics control based, trajectory tracking, decision making with application to financial market or in our case image recognition.

Actually there is no unified framework for the spiking neural networks. We can found applications for different models each of one is a trade-off between the biological plausibility and the computational cost. The LIF model is known to be simple and efficient where the Hodgkin-Huxley model is sophisticated but slow.

One of the big challenges for spiking neural networks is how to do a good learning algorithm because the goal of the methods is to approach the biological learning method of the brain which we don't understand well yet. There is also fact that spikes are discontinuous.

5.1 Further Work

It is our belief that using a architecture close to the one used in [5] is potentialy a way to extract visual information though Sipke based neural network.

5.1.1 NatCSNN Architecture :

NatCSNN is a Natural Convolutional Spiking Neural Network. It is based on a variant of the LIF neurons.Equation (3.2) can be modified using the Multi-Timescale Adaptive Threshold Model (MAT) presented in [4]. MAT provides an adaptive threshold that prevents neuron from over-spiking when exposed to high continuous currents. The adaptive threshold is described by the equations below.

$$V_{th}(t) = \sum_{k} H(t - t_k) + E_L \tag{5.1}$$

where

$$H(t) = \sum_{j=1}^{L} w_j^{-t\tau_{mj}} \tag{5.2}$$

with $V_{th}(t)$ is the threshold voltage in time t, t_k is the k^{th} spike time, L is the number of threshold time constants, $\tau_{mj}(j = 1, 1, \ldots, L)$ are the j^{th} time constants, $w_j(j = 1, \ldots, L)$ are the weights of the j^{th} time constants, and E_L is the reset membrane potential value [4].

The input layer of the NatCSNN receive an image. The first layer transform the input value of each pixels into spike trains. The second Layer 2a, extract feature from the input image, and the Layer 2b provides a lateral inhibition, The last layer is used to classify the image. The architecture is shown below:

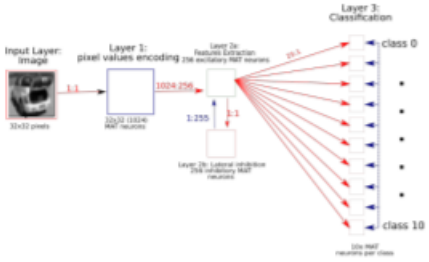


Figure 5.1: NatCSNN architecture

The neuron in Layer 1 are fully connected to the neurons in Layers 2a. During the training phase, STDP synapses are used to adjust the weights of the connection between the pre-neurons(Layer 1) and the post-neuron (Layer 2a).

Each neuron in the Layer 2a present it's output to a single neuron of the Layer 2b via an excitatory synapse and receives incoming spike events from all the other neurons in the lateral inhibition group.

In our case, the last layer does not interest us, the interesting layers are 2a and 2b, where the potential visual information are stored.

This whole network has achieve pretty good results on the CIFAR-10 dataset. Even if the first goal of the this neural model is to classify images, in order to do so, the network has to extract perceptual visual information, which can potentialy be encoded and decoded.

Chapter 6

Conclusion

The first goal of this project was to try to use spike neural networks (SNN) and the LIF quantizer to extract and encode visual information from videos. It turned out that it was too complicated to do in the short time period I actually worked, and the low amount of paper on this subject made it even harder and the fact that I almost worked completely alone. That is why I decided instead of trying endlessly to build a network when i had no idea how to build it, i tried to build a python "library" to encode data using the LIFQ, which did not existed before. As of now what I produced is not yet able to fit that role well enough, but the core components are present. I'm able to encode and decode images and videos, I can compute the most common qualitative metrics. The code just need a bit of structure. During this project, I learned a lot about how the retina works and what can we try to learn from the biological system to make our systems better. It also helped me understand better the whole coding principle which was very vague to me at the start of the project. I think trying to mimic biological behavior is a good approach to build complex systems that are efficient, and i believe that in the future, progress will be made in this regard.

Bibliography

[1] Effrosyni Doutsi et al. "Retinal-inspired filtering for dynamic image coding". In: Sept. 2015. DOI: 10.1109/ICIP.2015.7351456.

[2] Wulfram Gerstner and Werner Kistler. *Spiking Neuron Models: An Introduction*. USA: Cambridge University Press, 2002. ISBN: 0521890799.

[3] David Heeger. "Poisson Model of Spike Generation". In: (Oct. 2000).

[4] Ryota Kobayashi, Yasuhiro Tsubo, and Shigeru Shinomoto. "Made-to-order spiking neuron model equipped with a multi-timescale adaptive threshold". In: *Frontiers in Computational Neuroscience* 3 (2009), p. 9. ISSN: 1662-5188. DOI: 10.3389/neuro.10.009.2009. URL: https://www.frontiersin.org/article/10.3389/neuro.10.009.2009.

[5] Pedro Machado, Georgina Cosma, and T. McGinnity. "NatCSNN: A Convolutional Spiking Neural Network for recognition of objects extracted from natural images". In: (Sept. 2019).

[6] Marcel Stimberg, Romain Brette, and Dan FM Goodman. "Brian 2, an intuitive and efficient neural simulator". In: *eLife* 8 (Aug. 2019). Ed. by Frances K Skinner et al., e47314. ISSN: 2050-084X. DOI: 10.7554/eLife.47314. URL: https://doi.org/10.7554/eLife.47314.

[7] Zhou Wang et al. "Image Quality Assessment: From Error Visibility to Structural Similarity". In: *Image Processing, IEEE Transactions on* 13 (May 2004), pp. 600–612. DOI: 10.1109/TIP.2003.819861.