POLITECNICO DI TORINO

Master's Degree in Electronic Engineering

Master's Degree Thesis

Design of an hardware accelerator for a Spiking Neural Network

Supervisors

Prof. Stefano DI CARLO

Prof. Alessandro SAVINO

Candidate

Alessio CARPEGNA

October 2021

Summary

Spiking neural networks (SNN) are a new paradigm of artificial neural networks that aims to emulate the human brain's behaviour more accurately. First of all, differently from their classical counterparts, as the name itself suggests, they propagate information through discrete spikes. Secondly the model tries to mimic the membrane potential dynamics of a biological neuron. Finally the learning process itself can be chosen in order to be as similar as possible to what today seems to happen in a human brain.
The biological plausibility of the model however is not the only advantage. Working with simple spikes, which are representable in numeric terms using a single bit, avoids the need for complex multiplications and does not involve the computation of non-linear output functions. This leads to an algorithm that is lighter and more suitable for the creation of a hardware accelerator.

This thesis aims to the complete development of a model that can be efficiently translated into an hardware architecture. The main challenge is to minimize the resources required by the algorithm to maximize the area utilization and obtain the maximum performances with a fixed hardware.
In this sense the mathematical model that describes the behaviour of the artificial neuron has a dominant role. For this reason the model chosen is the *Leaky Integrate and Fire* (LIF) which treats the neuron as the parallel of a capacitor and a resistor. It is sufficiently simple but at the same time allows a quite biologically accurate description of the neuron.
For the same reasons the chosen training algorithm is the *Spike-Timing Dependent Plasticity* (STDP). It consists in increasing the weight associated to a specific synapse if the time difference between when an input spike arrives through it and when the target neuron itself fires is sufficiently low. On the contrary if the role of the synapse is negligible, meaning that the input spike traveling through it reaches the target neuron after it has generated an output spike, its weight is decreased.

The model has been realized from scratch, taking the state of the art solutions as a reference and trying to make it as light as possible in order to target an hardware

accelerator. The model has been tested using *Brian 2*, a simulator for spiking neural networks written in python. In this phase the parameters of the developed model have been finely tuned in order to guarantee its correct behaviour and to obtain acceptable accuracy results. Once the network was ready and completely defined in terms of the characteristic equations describing the neurons' behaviour, the model has been explicitly implemented using python language. Here the first choices that aimed at the minimization of the required resources have been taken. First of all the way in which each neuron is updated, for example choosing a step-based solution instead of an event-based one. Once described the network and the inner computations performed by it in detail the obtained structure was simulated, verifying that the results were comparable with the *Brain 2* version. The network has been then further simplified to reduce the complexity to a minimum without considerably affecting the accuracy. Finally a possible architecture is presented, considering an offline learning system, and a first estimation of the performance is provided.

Spiking neural networks represent a relatively new field of research and present a huge variety of possible choices in terms of the model to describe the behaviour of the neurons, the learning method, the structure of the network itself, so the way in which neurons are interconnected one to the other. Different models are suitable for different applications, so there are versions that allow a very detailed and faithful description of a real biological neuron, but are not suitable for an hardware accelerator due to their complexity. Along this thesis work different alternatives are presented, in particular for what concerns the available models for the membrane potential, and the reasons for which one solution is preferred over the others is specified. The goal is to provide an overview of SNN in general, their behaviour and the possible choices that can be taken along the project. The hope is that this work could serve as a possible starting point for the ones that approach spiking neural network for the first time.

Acknowledgements

Table of Contents

List of Tables

List of Figures

Acronyms

ANN
Artificial Neural Network

ASIC
Application Specific Integrated Circuit

FPGA
Field Programmable Gate Array

IF
Integrate and Fire

LFSR
Linear feedback shift register

LIF
Leaky Integrate and Fire

LSB
Least Significant Bit/Byte

LTD
Long Term Depression

LTP
Long Term Potentiation

LUT
Look-Up Table

MSB

Most Significant Bit/Byte

SNN

Spiking Neural Network

STDP

Spike Timing Dependent Plasticity

VHDL

VHSIC Hardware Description Language

VHSIC

Very High Speed Integrated Circuits

Chapter 1

Introduction

Artificial neural networks represent a first human attempt to emulate his own brain. In many cases the models take inspiration from the biological organ but then separate themselves from it, trying to optimize their applicability in real world situations. Spiking neural networks represent a possible solution to fill this gap between biological plausibility and effective usefulness of the model in practical applications, so as pattern recognition, classification and similar tasks.
This chapter provides a brief overview of the reasons that have led to the choice of spiking neural networks as the target of an hardware accelerator and allows to better understand the decisions that have been taken along the project.
First of all, a simple model of a biological neuron is presented. This is fundamental to understand the main characteristics of artificial neural networks and even more so of spiking neural networks. After this a brief overview on the first ANNs is presented. Finally, spiking neural networks are introduced and their main characteristics are explained. They will be examined in more detail in the following chapters.

1.1 Biological neural network model

Human brain is composed by tens of billions of neural cells interconnected among themselves. In order to better understand how the information is elaborated inside the brain let's start by analyzing the structure of these cells, the neurons. Figure 1.1 shows a simple model of a biological neuron. Its main elements are:

1. **Body** of the neuron, also called **soma**: this is characterized by a membrane potential which can be increased by excitatory inputs or decreased by inhibitory inputs. Since the neuron is not perfectly isolated it presents a certain amount of leakage that in turns decreases the membrane potential. This means that, in absence of excitatory inputs and within a time interval that depends from the discharging time, the membrane potential tends to stabilize itself to a rest

value. Whenever the membrane potential exceeds a threshold the neuron itself becomes active, generating an output signal, known as the *action potential*, and its membrane potential is reset to the rest value.

2. **Dendrites**: input extremities of the neuron. These allow the neuron itself to receive signals from other neurons. As a consequence each neuron presents many of these dendrites.

3. **Axon**: output extremity of the neuron. It allows to transmit signals from the neuron towards other cells. Each neuron presents a single axon.

4. **Axon terminals**: terminal ramification of the axon that allows to connect it to multiple neurons.

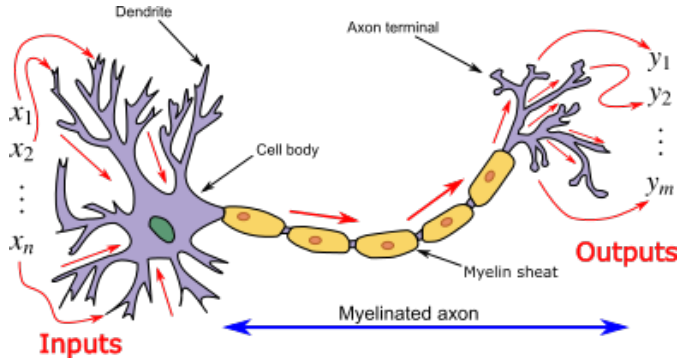


Figure 1.1: Biological neuron model from Egm4313.s12 at English Wikipedia, CC BY-SA 3.0, via Wikimedia Commons

It has been observed that the *action potential*, propagated through the axons and received by the dendrites takes the shape of short current spikes.
This spike is transferred from the *axon terminals* to the *dendrites* through **synapses**. Whenever a spike arrives through a specific synapse the membrane potential is increased or decreased, depending on if the corresponding input is excitatory or inhibitory. If the membrane potential exceeds a threshold it in turns emits a spike. In technical jargon it is said that the neuron *fires*.

Each synapse has its own weight in increasing or decreasing the membrane potential, meaning that some inputs have a stronger impact on modifying the state of the neuron. The *learning* consists in the fine tuning of the weights of different synapses in order to adapt the neuron response to the received inputs. If for example a specific input is more likely to make the current neuron fire its weight is increased. Vice versa if it doesn't have an active role in making the membrane potential

exceed the threshold its weight is decreased. This creates paths inside the brain that interconnect neurons that are frequently active together and this is one of the main characteristics that allows the human being to learn and to remember. This capability of strengthen or weaken the synapse is called **plasticity**.

1.2 Artificial neural networks

Artificial neural networks are in general mathematical models inspired by the human brain. They are composed by interconnected artificial neurons, each of which applies a specific mathematical function to its inputs and then propagates the result towards all the neurons connected to its output. Figure 1.2 shows a graph representation of a generic ANN. It can be seen that the network is organized in columns, called *layers*. Each node represents a neuron, except for the nodes in the first layer. Each neuron is connected to all the nodes of the previous and following layer, leading to what is called a *fully connected graph*.

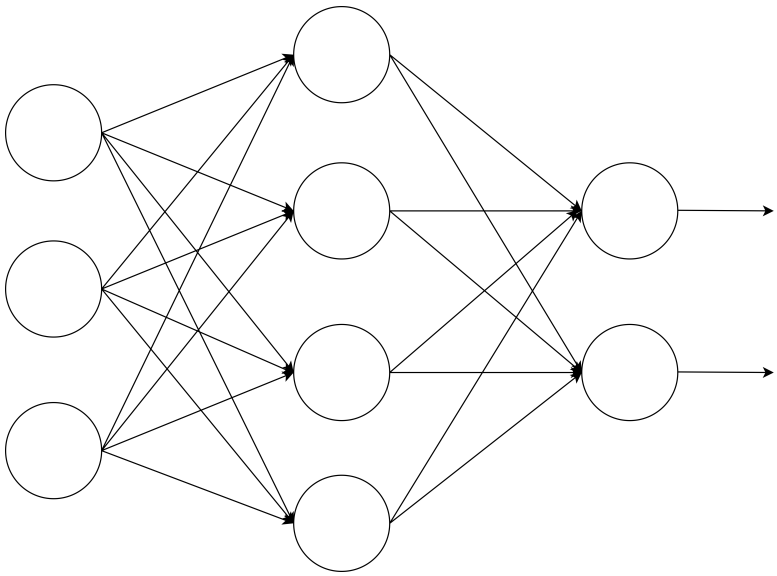


Figure 1.2: Generic artificial neural network

The network is organised in:

1. **Input layer**: first group of nodes. The nodes are not associated to physical neurons, but to the inputs of the network. For example if the network has been designed to work with real numbers, each node represents the numerical value of a specific input. If instead it is a spiking neural network each node is associated to a spike train of a predetermined duration.

2. **Output layer**: last group of nodes. It is not connected to any other layer of neurons and provides the result of the elaboration performed by the network as an output.

3. **Hidden layers**: all the layers that stay between the input and the output layers.

For classical ANN the similarity with the human brain is only vague: the information exchanged between the neurons is generally represented in form of real numbers and the neurons themselves apply functions that have nothing to do with the real behaviour of a biological neuron. Figure 1.3 shows a generic model of an artificial neuron, that is generally composed by:

1. Weighted sum of the inputs. The neuron is treated as an ideal integrator, that simply add together the inputs without decreasing the obtained value in any way with the passage of time. In reality there will always be some leakage and so a more biologically plausible model should behave as a leaky integrator.

2. Application of a non-linear function, called the *activation function*, to the result.

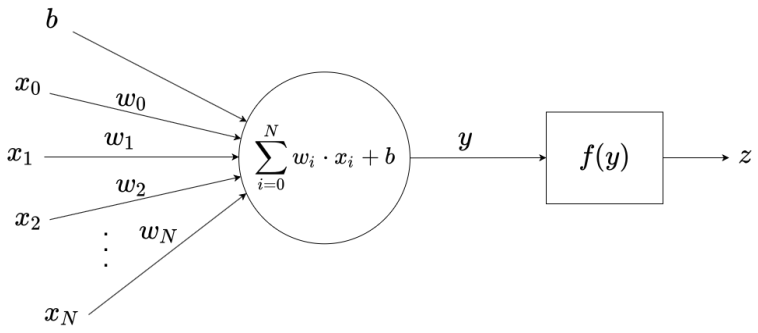


Figure 1.3: Generic artificial neuron

Examples of non-linear functions used in neural network models are the *binary step*, the *sigmoid function*, the *hyperbolic tangent*, the *Rectified Linear Unit* (ReLU).

A possible approach to compare this kind of models with the biological structure presented in the previous section is the following one:

- As said before the type of information exchanged by the neurons is completely different from the biological case. Here each neuron receives real numbers in

input and returns a real number as an output, while in the biological model both inputs and output are represented using trains of spikes. The only way to correlate the two kind of data encoding is to consider the real number as a parameter which describes a specific characteristic of the train of spikes, for example the frequency of the spikes, also called *firing rate*. In this case the higher the real value, the higher the number of spikes received from the specific input within a predetermined time interval.

- The neuron first of all performs a weighted sum of its inputs. The result can be interpreted as the value of the membrane potential.

- Secondly a non-linear function is applied to the result. In the simplest case in which this activation function is a binary step it can be seen as the application of a threshold: if the weighted sum is higher than a threshold the neuron fires, otherwise it doesn't. If instead the function is more complex its output can be again associated with the firing rate of the neuron.

However this is an artifice to find a similarity between the models and has no practical relevance.

From the learning point of view the most used method in classical ANN is the **backpropagation**: the network is provided with both the input and the expected output and a *cost function* is computed as the difference between the expected value and the real output of the network. The learning consists in minimizing this cost function by changing the weights of the synapses of all the neurons starting from the output layer and moving towards the inputs, propagating the error in the backward direction. In this way the network gradually adapts itself to the received inputs and becomes able to distinguish and classify them.

Even if with the current knowledge of the way in which the human brain learns it is not possible to exclude a priori a possible role of the backpropagation in the learning process, it doesn't seem the most plausible way. Today the most promising method seems to be the *spike timing dependent plasticity*, an unsupervised method based on the temporal difference between the arrival of input and output spikes. The difference between supervised and unsupervised methods is presented in chapter 6, while the STDP method is briefly explained in the next section and then investigated in detail in the next chapters.

Even if ANN are in general quite different from a biological brain they have shown themselves very useful in many fields, from the data classification to computer vision, to pattern recognition and so on.

1.3 Spiking neural networks

Spiking neural networks try to mimic the biological brain more accurately. They have been called "the third generation of neural network models"[1] thanks to their biological plausibility. As in more classical ANN the network can be seen as a group of interconnected neurons. Figure 1.4 shows a schematic model of a spiking neuron. It can be seen that both the inputs and the output are encoded in form of spike trains. This implies a first crucial difference with respect to the previously presented models, that is the introduction of the concept of time.

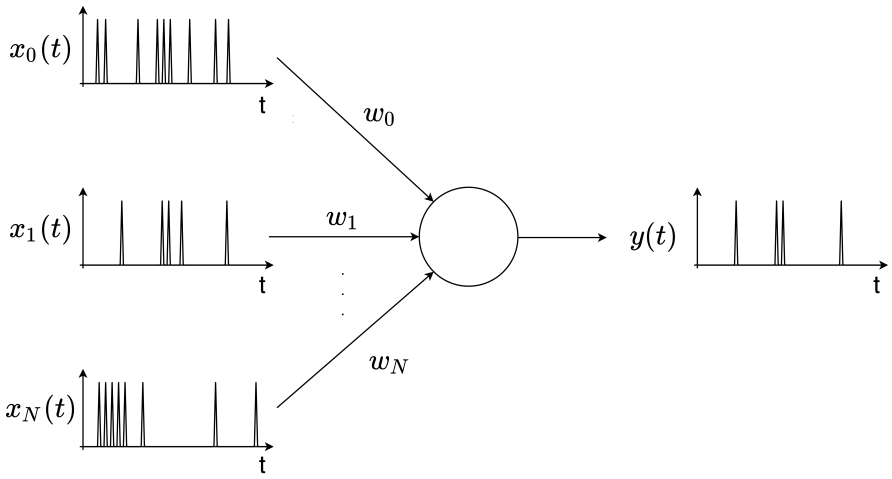


Figure 1.4: Simple spiking neuron

Spikes intrinsically bring a binary information:

1. A spike arrives through the specific synapse during the current time step.
2. No spike is received from the synapse during the current time step.

SNN use an extra dimension, the time, to encode more complex information. This implies that the moment in which a spike arrives, the number of spikes that reach a neuron within a certain time interval and the time distance between two subsequent spikes all become important parameters that affect the output of the neuron.

Many mathematical models have been created to describe the temporal evolution of the membrane potential of a biological neuron. They will be analyzed in chapter 4. The common principles on which they are based are:

1. Every time the neuron receives a spike from an excitatory synapse its membrane potential is increased by a quantity proportional to the synapse weight.

2. Every time the neuron receives a spike from an inhibitory synapse its membrane potential is decreased by a quantity proportional to the synapse weight.

3. The membrane potential decreases with the passage of time and, in absence of excitatory inputs, it tends towards a rest value.

4. Every time the membrane potential exceeds a threshold the neuron fires and the membrane potential is reset to the rest value.

So to sum up the membrane potential model behaves as a leaky integrator with a firing threshold that once exceeded causes its reset.

As said before the chosen learning method is the *spike timing dependent plasticity*, which seems to be the best model of the biological process of learning. It consists in modifying the weight of a synapse on the base of its relevance in making the neuron fire. In particular:

- If an input spike arrives through the synapse shortly before the neuron fires it probably has a primary role in the generation of the output spike. In this case the synapse weight is increased.

- If instead the input spike coming from the specific synapse arrives after the neuron has fired it is probably irrelevant for the generation of the output spike. In this case the synapse weight is decreased.

So spiking neural networks provide a simplified model of a biological brain, involving its main features but at the same time allowing to reach an accuracy that is near to the one of more abstract artificial neural network models. A right balance between biological plausibility and practical applicability is fundamental to advance in the understanding of how a human brain works.

Chapter 2

Brief history of SNN: model of a biological neuron

This chapter provides a brief overview of the main historical steps that have led to the models and methods that will be used along the project. Its aim is not to be a comprehensive historical reference but to analyze the main historical figures that have allowed to advance the understandings in how the human brain works and how to mathematically model and emulate it.

2.1 First recorded reference to the brain

The first documented reference to the human brain dates back to the 17^{th} century BC. It corresponds to a hierogliph, repeated eight times inside an Egyptian manual of military surgery.

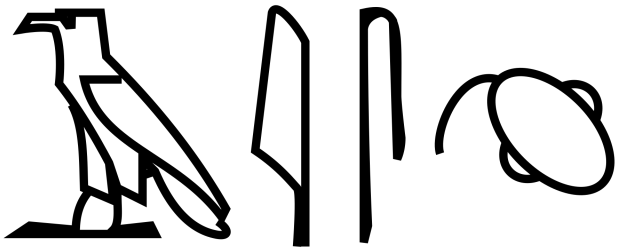


Figure 2.1: Brain hierogliph from Riccardo.metere, CC BY-SA 4.0, via Wikimedia Commons

The manual is considered the first medical treatise on trauma and reports 48 types of injuries, fractures, wounds, dislocations and tumors, with a total length of the document of 4.68m. Between the various injuries also head wounds are cited: the papyrus refers to the external surface of the brain, together with the meninges and the cerebrospinal fluid. The interesting aspect of the document is that it shows a very rational approach towards medicine, resorting to magic only to explain mysterious illnesses still not understandable with the science of the time.

The papyrus was named *Edwin Smith Papyrus*[2] after Edwin Smith, the man that acquired it in Luxor in 1862 from the Egyptian seller Mustafa Agah.

2.2 Galvani and the animal electricity

Along the path towards the modeling of the behaviour of a biological neuron one fundamental step has been the discovery of the transmission of electricity in biological organisms. This opened the way to many studies which demonstrated that the information is propagated inside the brain, and in the same way from the brain to the rest of the body, through electricity.

The first who hypothesized the role of the electricity in the movements of animal muscles was an Italian physician, physicist, biologist and philosopher, *Luigi Galvani*[3], together with his wife *Lucia Galeazzi Galvani*.

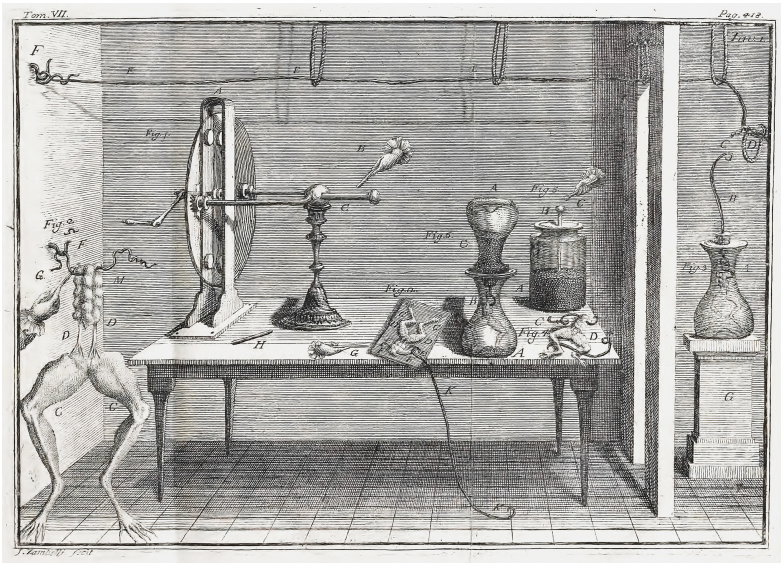


Figure 2.2: Experiment De viribus electricitatis in motu musculari, Luigi Galvani, Public domain, via Wikimedia Commons

The details of his experiment are reported in *De viribus electricitatis in motu musculari*, written by Galvani himself in 1791, in which the main steps that have led to the discovery of the *animal electricity* are explained. At the time the most quoted theory about the mechanism through which muscles were able to contract and relax was the *balloonist theory*. The idea was that muscle contraction was caused by the inflation of air or fluid. The theory had been already questioned by the discovery that neurons where not hollow and so would not be able to conduct the fluid towards the muscles, but Galvani's discoveries finally demolished it.

There is an interesting popular legend about how the Galvani discovered the role of the electricity in the contraction of the muscles: at the time he and his wife were working on experiments on the electricity and some of these experiments involved the skin of the frog, which they rubbed to obtain static electricity. It seems that Galvani's assistant, passing near the frog that he was skinning, accidentally touched an exposed sciatic nerve with a scalpel that has picked up charge. The reaction was that the legs of the dead frog instantly kicked as if the frog came back to life. In any way it has gone Galvani start to study the response of frog legs to the application of a current through two electrodes and discovered that was the electricity and not an inflated fluid the cause for the muscle contraction. The theory that he and his wife derived stated that the electricity was brought to the muscle through a fluid of ions, leading to what they called the *animal electricity*. Their idea was that the motion was allowed by a current intrinsic in the legs, so as in other part of the body. The main opposer of this thesis was the Italian physicist and chemist *Alessandro Volta*, that rightfully claimed that the electric current through the legs was the same that went from one metal electrode to the other. Volta himself coined the term *galvanism* to refer to the generation of electric current by chemical action. It is interesting to note that the debate with Galvani led Volta to develop his famous battery, today known as the *Volta's battery*.

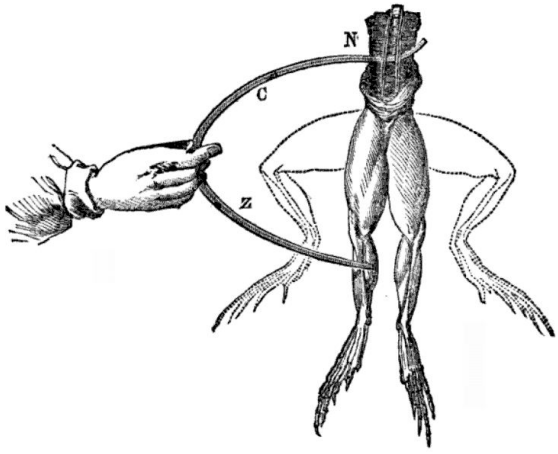


Figure 2.3: Experiments on the frog legs, Luigi Galvani, Public domain, via Wikimedia Commons

Galvani's work directly inspired Mary Wollstonecraft Shelley in the writing of Frankenstein, in which Victor Frankenstein uses electricity to bring back to life the

corpse of his monster.

2.3 Birth of the neuroscience

In the 19^{th} centuries scientists begin to study the propagation of electric current along the nervous cells. In particular the Italian physicist and neurophysiologist *Carlo Matteucci*, following the work of Galvani, demonstrated that the membrane of these cells have a voltage across them and can generate a direct current.

The work of Matteucci inspired *Emil Du Bois Raimond*, who discovered the action potential in 1843. Few years later his friend *Hermann Von Helmoltz* measured for the first time the propagation velocity of this action potential. The work of the two German scientists sets off the birth of the neuroscience.

2.4 The neuron

At the end of the 19^{th} century the diffused conviction, postulated by the German anatomist *Joseph von Gerlach* in 1871, was that everything in the nervous system, first of all the brain, was composed by a single continuous network[4]. The thesis was confirmed by the work of the Italian biologist and pathologist *Camillo Golgi*.

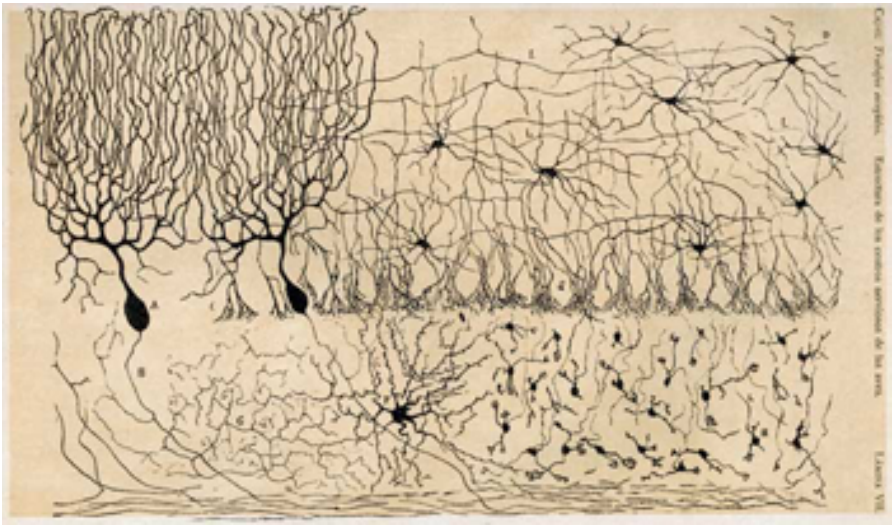


Figure 2.4: Cajal cerebellum, Santiago Ramón, Public domain, via Wikimedia Commons

In 1873 Golgi was doing research on the human brain structure, sectioning and analyzing pieces of it. One evening a servant accidentally threw away a piece of

brain that still needed to be analyzed. As a coincidence Golgi had threw away some silver nitrate in the same waste basket few hours before. The following day, once retrieved the piece of brain, Golgi analyzed it and noticed that it had uniformly absorbed the silver nitrate and that the material highlighted the internal structure of the brain if watched at the microscope. Starting from there he developed a method to color the brain cells in order to precisely visualize them, which he initially named the *black reaction*, since it caused the brain cells to color in black. The method was then later named the *Golgi stain* or *Golgi method*[5]. Anyway Golgi's observations didn't go against Gerlach's reticular theory and strengthened the conviction that the nervous system was composed by a single big network.

It was only fourteen years later, in 1887, that the Spanish neuroscientist, pathologist, and histologist *Santiago Ramón Y Cajal*, discovering the method developed by Golgi and perfecting it, was able to highlight the nervous cells as separated entities, demonstrating that also the brain, as all the other human organs, can be divided into elementary units. During this period he made extensive and detailed drawings of the neural material, directly observing it at the microscope. Figure 2.4 shows one of these drawings. Cajal and Golgi shared the Nobel prize for the medicine in 1906.

Finally four years later, in 1891,the German anatomist *Heinrich Wilhelm von Waldeyer-Hartz* wrote a complete review of the doctrine for which the brain was composed by separated cells, in which he firstly introduced the term *neuron*.

2.5 Julius Bernstein and the membrane theory of electrical potentials

Julius Bernstein was a German physiologist and belonged to the Berlin school of organic physicists, who played a central role in creating modern physiology and biophysics during the second half of the 19th century. He trained under Emil Du Bois Raimond and worked as a researcher with Hermann Von Helmoltz. He his mainly known for two results, that are the natural continuation of the work of his two mentors:

1. The first accurate description of the action potential, obtained in 1868.
2. The first plausible psycho-chemical model of bioelectric events, published in 1902 in his *Membrane Theory of Electrical Potentials*.

During his work on the action potential Emil Du Bois Raimond introduced the concept of *negative variation* to indicate a temporary reduction in the current

propagated towards the muscle under analysis. Few years later Hermann Von Helmoltz, measuring the propagation time of the action potential concentrated on the speed of the so called *excitatory process* along the nerves. However it was not clear if the two processes were the same thing and whether they propagated at the same speed. To solve the problem Bernstein developed a new instrument, called the differential reothome, reported in figure 2.5. The *Reothome*, that can be literally translated in "current cutter", was a primitive version of a switch. It can be seen that it has a plate with a cam on its edge.

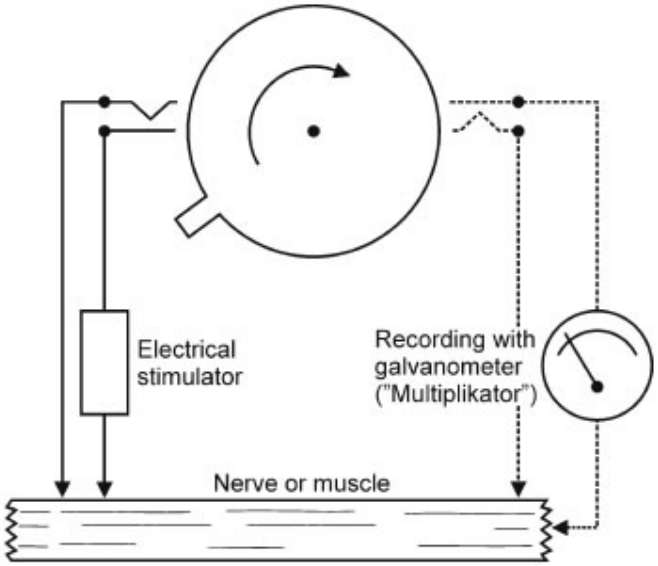


Figure 2.5: Bernstein's differential reothome, from "Julius Bernstein (1839–1917): pioneer neurobiologist and biophysicist", Ernst-August Seyfarth

By rotating, this plate can close the electrical stimulator loop with a time resolution below the ms and then close the measurement loop, allowing the recording of the current with a classical galvanometer. In this way Bernstein demonstrated that Du Bois' negative variation and Von Helmoltz' excitatory process are two different process that however propagate at the same speed. The experiment allowed him to obtain what is considered the first accurate description of the action potential, reported in figure 2.6. Here an excitation is delivered in point p and then the propagation of the two waves in the two opposite directions is plotted.

Finally Bernstein introduced the first plausible psycho-chemical model of bioelectric events: following the discoveries of Walter Nernst and Wilhelm Ostwald he developed a chemical model of the membrane of muscles and nerves. Applying the principles of semi-permeable membranes to ions he discovered that the selective

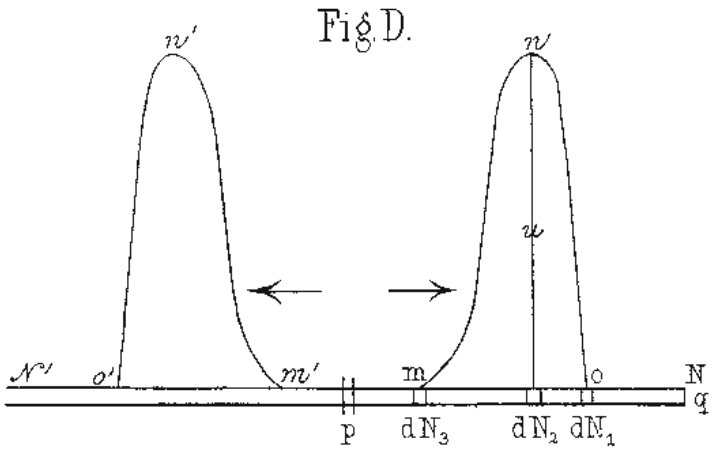


Figure 2.6: Bernstein's action potential, from "Julius Bernstein (1839–1917): pioneer neurobiologist and biophysicist", Ernst-August Seyfarth

permeability to some specific ions allows the accumulation of charge on one side of the membrane and so the creation of the potential. [6]

2.6 Louis Lapique and his model of the membrane potential

The work of Luigi Galvani and his wife demonstrated that the nerves could be excited electrically. Some aspects of the phenomenon however, such as the required duration and intensity of the stimulation that allowed to effectively reach the excitation, together with the link between this excitability and the biophysics of the nerve, was not covered by their study.

Here came *Louis Lapique*, a French physiologist who was the first to develop a physical model of the membrane excitability in 1907. The idea was that, in order to excite the nerve, and so to obtain a movement of the frog's leg, the membrane voltage should cross a certain threshold. The research of Lapique was focused on the characteristics that the input excitation should have had to cause the exceeding of this threshold. He compared his results with the measurements performed, also in this case, on a frog leg. The frequent use of frogs for biological experiments should not surprise since they were largely diffused and practical to use in laboratory.

The model developed by Lapique threated the membrane of the neuron as a *leaky capacitor*, that is a capacitor with a resistance in parallel, such as the one reported

in figure 2.7, where K represents the capacitance and ρ the resistance.

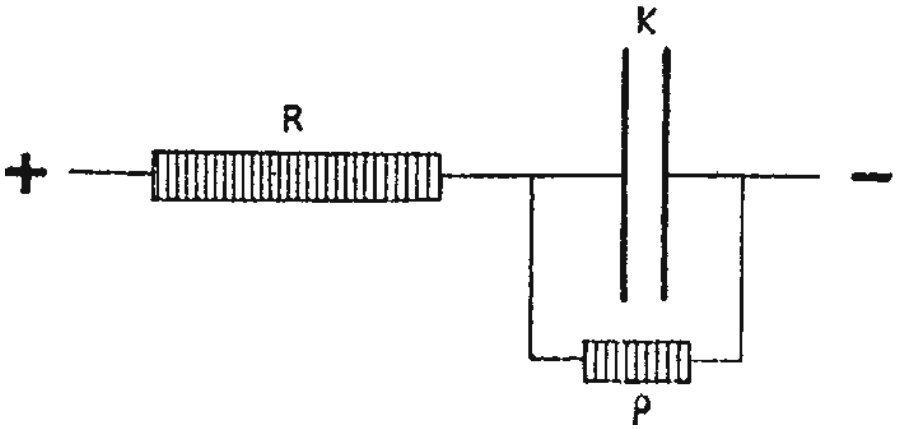


Figure 2.7: Lapique membrane model, from Quantitative investigations of electrical nerve excitation treated as polarization, Mark C. W. Van Rossum

In those years a constant current generator was difficult to obtain. For this reason Lapique used a constant voltage source, a battery. The resistor R reported in figure 2.7 has a resistance value much higher than ρ and is used to stabilize the current to an almost constant value, in order to obtain something similar to a constant current source. This is obtained with a one meter long wire with a slider which allows to select the resistance value. For this reason the voltages in Lapique's paper are reported in centimeters.

In order to analyze the required duration of the input excitation a time resolution of few ms was required. The instrument that allowed to obtain a current pulse of the desired duration with such a resolution was the *reothome*. There were many versions of it, such as reothomes that used pendulums or rotating discs. Lapique used an ingenious alternative, called the *ballistic reothome*. It consisted in a gun-like equipment that shot towards two subsequent wires. The first wire, when cutted by the incident bullet, started the current conduction, while the second interrupted it. By tuning the distance of the two wires the time needed for the bullet to go from the first wire to the second one, and so the current impulse duration, could be modified as well. Lapique in his paper says that all the components of the experiment was put together in a big incubator, *"except the ballistic interrupter with its fulminate rifle, which emits unpleasant smoke in a restricted space"*. Since the pulse duration is selected through the distance of the two wires also the time values analyzed along the paper are reported in centimeters.

The results obtained by Lapique was not satisfying and didn't correspond to the experimental one. Part of the fault can be attributed to the rudimentary

instruments used for the experiments and part to the oversimplicity of the model. However this is a fundamental step to understand how the neural excitability works. [7]

Even if in the paper there is no reference to the spiking nature of the current received and generated by the neurons[8], the link between the neuron excitation and the generation of such spikes or the reset of the membrane that follows the spike, this is commonly considered the fundamental research work on which the *leaky integrate and fire* model has its roots. It is a simple model of a biological neuron that describes it simply as the parallel between a capacitor and a resistor, exactly as Lapique's model did. The first documented integrate and fire models started to appear in 1960s. Thanks to its simplicity alongside with its biological plausibility the model is much diffused nowaday. It will be analyzed in detail in the next chapters.

2.7 Hodgkin, Huxley and Katz

In 1949 Alan Hodgkin and Sir Bernard Katz refined Bernstein's ideas on the semi-permeability of the axonal membrane, considering that it could have different permeability to different ions, and demonstrated the fundamental role of sodium permeability in the action potential. During their research they successfully measured, together with Andrew Huxley, the dependence of the axonal membrane's permeability to sodium with respect to the voltage and the time, from which they obtained a quantitative model of the action potential. The model, known as *Hodgkin-Huxley model*[9], has become probably the most important and accurate model of the action potential and will be described more in detail in chapter 4

Chapter 3

Brief history of SNN: learning in the human brain

The other side of the coin in the development of a model of the human brain is how it is able to learn. In fact a sheer model of biological neurons interconnected one to the other and able to exchange information by integrating their inputs and discharging their action potential towards other neurons, however accurate it is,is not sufficient to explain how it allows humans to perform complex tasks and to improve their capabilities through the learning. This sections provides a brief overview of the evolution of the concept of learning[10], while the technical solutions used to practically implement a working *spiking neural network* will be discussed in the next chapters.

3.1 The first philosophers

Human learning has been topic of philosophical discussion for centuries. *Plato* for example argued that the human mind was molded before its birth. The learning process in his opinion is nothing else than *anamnesis*, a recollection of concepts already known before the birth and stimulated by a sensory experience.This kind of view is called *innatism* and asserts that the brain structure is predefined when an individual is born.

Aristotle, student under Plato, went against his master's idea and in 350BC formulated the famous concept of *tabula rasa*: the idea was the exact contrary of what Plato argued, and asserted that the human mind, when an individual is born, is a *tabula rasa*, a white board without any previous knowledge in it.

The concept has been revisited in different ways in the following centuries: the

Persian philosopher *Avicenna* in 1000 AD developed it, arguing that cerebral structures are molded by experiences and education; *John Locke* in 1689 AD recast the concept comparing the human mind at the birth to a *clean blackboard*, shaped along the life by the experience.

3.2 Synapses and plasticity

Alexander Bain in his *Mind and Body*[11] of 1873 was probably the first to hypothesize a link between the way in which the information flows in the brain and the learning and memory. He talks about changes in the connections between cells when the concept of *synapse* was still unknown.

The American psychologist *William James* introduced some fundamental concepts on the strengthening of the junctions between the neurons as a way to learn. In 1890 in *The principle of psychology*[12] he hypothesized that the association of objects and events, and so the learning, was associated with a facilitation of the current flow along paths that it has already followed, thanks to the plasticity of the organic material that composed the nerves. In his opinion the key elements for the learning was three:

1. *Repetition*: the activity of a specific region depends on how many times the excitation of other points have accompanied the activation of that specific region.
2. *Intensity* of excitation of such afferent points.
3. *Competition*: the activation of a specific region depends on the presence of other points, disjointed by the region, in which the excitation of the afferent points can discharge itself.

He highlighted the key role of the *association*: a stimulus that on its own is not able to excite a nervous center, working together with other stimuli can cause the excitation. As a consequence if two processes has happened one near to the other for many times, the activation of the first causes the excitation of the second and vice versa. This concept is quite similar to the postulates that *Donald Hebb* would formulate about one century later.

The concept of *synapse* has been introduced some years later, by *Sir Charles Sherrington* in 1897[13]. He initially described the structure, referring to it with the name *synapse* only in 1909[14]. Working in parallel with Ramon Y Cajal he argued that the fundamental structural unit of the nervous system was a specialized cell with unidirectional polarization transmission, the neuron, not continuously

connected, but barely in contact. However he doesn't link the modifications of the synapses to the learning process.

3.3 The Hebbian plasticity

Donald Hebb was a Canadian psychologist, known for his work on the synaptic plasticity. As he stated many times he didn't introduced new concepts, the ideas that he expressed were quite diffused at the time. However he had the merit to put together the existing theories on the plasticity and to create a unified model, which for this reason is generally known as *Hebbian plasticity*. In *The organization of behavior*[15], written in 1949 he introduced the key concepts of his theory:

1. The memory is stored in the case in which a repeated joint activity of neuron groups causes the connection between them to be strengthen.

2. The persistence and repetition of *reverberation activity*, which he called *traces*, induces changes that last in time. The concept of *reverberation* was not new. The key idea was that the presence of closed loops inside the brain caused a stimulus, once entered, to propagate inside it for an amount of time of the order of some minutes.

3. The memory consists in the creation of *phase sequences*, that is the ordered activation of specific neuron groups associated to a given concept or memory. The same cell can be associated with multiple sequences. The activation of such sequences does not require the direct presence of the stimulus. As Hebb himself stated: "You need not have an elephant present to think about elephants".

Hebb claimed the possibility to form new connections as a consequence of a repeated stimulus. In this case, if the activation of a neurons group repeatedly activates an unconnected cell, through the chain propagation through interconnected groups, a new connection is created.

It is interesting to note that Hebb considered only a enhancement of the connection between groups if they are active together, not its depression in the opposite case in which their activation is uncorrelated. This kind of *synaptic potentiation* from this work on became known as *Hebbian plasticity*, while the connection of neuron groups took the name of *Hebbian assemblies*.

A famous, even if a bit simplistic, summation of Hebbian's postulate, created by *Carla Shatz* in 1992 is: "*cells that fire together wire together*".

3.4 Jerzy Konorski and his concept of plasticity

Jerzy Konorski was a Polish neurophysiologist that deserves a quote. He introduced a theory on synaptic plasticity that was quite similar to the Hebb's one, one year before it[16]. His ideas, originated from the work of *Ivan Pavlov*, a Russian physiologist mainly known for his work on the *conditioning of behaviour*, soon diverged from it and Konorski went against some of the key concepts expressed by Pavlov. The Russian scientist at the time benefited of a discrete religious and political consideration and so Konorski's work went suppressed for many years. As a consequence his theories had much less resonance than deserved in the West, even if his impact on the work of some of the main researchers of the time, such as Hebb, is unquestionable. For these reasons the *Hebbian plasticity* is sometimes called *Hebb-Konorski plasticity*.

In spite of the similarities with Hebb's work, Konorski's theory presents some differences: he went against the idea of the creation of new connections, arguing that learning and memories derived from the strengthening of already existing connections only. He also introduced the concept of *inhibitory connections*, making the receiving neuronal center less active after the activation of the transmitting one.

3.5 First applications of Hebbian plasticity

In 1956 an IBM group including *Rochester, Holland, Haibt* and *Duda* tested the creation of *Hebbian assemblies*, through the strengthening of their connection, on one of the biggest computers of the time[17]. They realized that in practice it didn't work and developed an alternative formulation. The problems in particular were:

1. There was no mechanism to weaken the connection between cells that are rarely active together.

2. The weights of the connection, that in practical applications correspond to its strength, could grow indefinitely, there was no mechanism that took them under control.

For this reason they introduced some modifications to the original formulation. In particular their study postulated the existence of a weakening of the inactive synapses through a competitive mechanism: the idea is that the total sum of the weights of the synapses must stay constant. This means that, after every learning interval, in which active synapses are strengthened, that is their weight is increased,

a normalization step is performed. In this way inactive synapses are indirectly weakened in order to keep the total sum constant.

An early lasting mathematical formulation of the synaptic plasticity was developed in 1958 by *Frank Rosenblatt*[18], inspired by the work of Hebb and his fellows. He was the first to introduce the concept of *bivalent system*, with the possibility of "*rewarding*" the active synapses or "*punishing*" the inactive ones. In his famous description of the brain as a *multilayer perceptron* the weight of a synapse can be incremented or decremented depending on its effect on the output neuron.

In those years many studies on the depression of the synapses, being it a direct mathematical decrease or an indirect reduction due to a normalization were performed. An other research hot topic was the presence of both *excitatory* and inhibitory synapses, a concept expressed by Konorski about twenty years earlier[16].

3.6 BCM learning rule

A fundamental step in the development of models for the synaptic plasticity was carried out in 1970 with the development of the BCM learning rule[19], which takes its name from the last names of its creators, *Eli Bienstock*, *Leon Cooper* and *Paul Munro*. The rule unified the existing theories in a mathematical model based on the *frequency* of the presynaptic pulses:

1. An high frequency input excitation leads to a strengthening of the connection, which at the time was already called *long term potentiaton* (LTP).

2. Vice versa a low frequency stimulation implies a weakening of the connection, called *long term depression* (LTD).

Bienstock, Cooper and Munro introduced in this way the concept of *homosynaptic depression*. The weakening of the connection in this case does not depend on a normalization which follows a strengthening of the active synapses, nor it depends by the inactivity of the connection during a competitive input. The plasticity depends on a well defined frequency requirement.

The drawback of the method is that the dependence from *time* is completely eliminated, so there is no dependence of the strengthening or weakening of the weights from the arrival time of an input excitation or the output firing.

3.7 Backward propagation of the action potential

The BCM rule, together with later studies on the plasticity, didn't consider the importance of the reciprocal timing of input and output spikes. This disagreed with both the original Hebb's postulate and the observations that more detailed analysis provided in the following years.

Between 1980 and 1990 it was clear that the strengthening or weakening of the connection between different cells happened at the synapses level, that is in correspondence of the dendrites. As a consequence *William B. Levy* and *Oswald Steward* hypothesized that there was a way in which a postsynaptic pulse could propagate backward towards the dendrites to prepare the synapse for the plasticity[20].
In 1994 *Greg Stuart* unequivocally demonstrated the possibility for the action potential to actively propagate in the backward direction[21].

3.8 Spike timing dependent plasticity

The first experimental study on the importance of the precise relative timing of the spikes emitted by pre and postsynaptic neurons was presented by *Henry Markram* at the *Annual Society for Neuroscience Meeting* in 1995[22]. In his analysis the back-propagating spike could be seen as the integrated sum of all the presynaptic inputs. In the experiment the postsynaptic spike was obtained through a direct current injection and so the plasticity demonstrated to be non-heterosynaptic, that is, it depends only on the specific neuron and not from a normalization which involves all the synapses in the network.

The study demonstrated that the LTP was induced in presence of a causal pre-before-post spike timing, while LTD occurred in case of acausal post-before-pre spike timing with a temporal displacement of 10ms in both cases. A time difference grater than 100ms showed to be ineffective for the connection plasticity and this rejects Shatz' expression: neurons that fire together not always wire together because also the relative timing is important.[23]

This kind of strengthening/weakening mechanism was then called *spike timing dependent plasticity* by *Sen Song et al.* in 2000 and to this day it seems to be the most biologically plausible mechanism through which the human brain is able to learn.

Chapter 4

Mathematical models of the membrane potential

4.1 Hodgkin-Huxley model

This is a mathematical model which accurately describes the electrical behaviour of the neuron's membrane. It was initially developed to describe the action potential within a *squid giant axon*[9] and earned Alan Hodgkin and Andrew Huxley the Nobel prize in 1963.

The electrical potential is described through the equivalent circuit reported in figure 4.1.

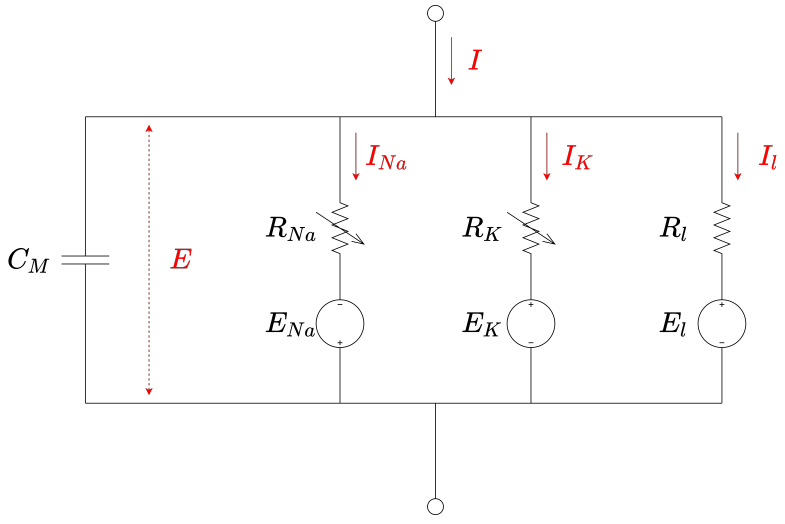


Figure 4.1: Hodgkin-Huxley equivalent circuit

There are two main mechanisms through which the membrane voltage can be modified:

1. Charging of the membrane capacitance.
2. Flow of ions through the membrane resistance.

The ionic current that can cross the membrane is divided into different contributes, two carried by *sodium* and *potassium* ions, respectively I_{Na} and I_K, and one small *leakage current*, composed by chloride and other ions, indicated as I_l. The *sodium* and *potassium* resistance is variable and depends on the membrane voltage. All the other components, so R_l, E_{Na}, E_K, E_l and C_M, are constant. For the mathematical analysis is more convenient to work with the conductance instead of the resistance:

$$g_{Na} = \frac{1}{R_{Na}} \tag{4.1}$$

$$g_K = \frac{1}{R_K} \tag{4.2}$$

$$g_l = \frac{1}{R_l} \tag{4.3}$$

Let's now introduce some notation to simplify the equations. E_r identifies the membrane rest potential.

$$V = E - E_r \tag{4.4}$$

$$V_{Na} = E_{Na} - E_r \tag{4.5}$$

$$V_K = E_K - E_r \tag{4.6}$$

$$V_l = E_l - E_r \tag{4.7}$$

V represents the displacement if the membrane potential from its rest value. The total current entering the membrane can be described as:

$$I = C_M \cdot \frac{dV}{dt} + g_{Na} \cdot (V - V_{Na}) + g_K \cdot (V - V_K) + g_l \cdot (V - V_l) \tag{4.8}$$

Along the following sections the physical interpretation provided by Hodgkin and Huxley is provided to explain the conductance variations. However, as they say in the original paper, this should not be interpreted as a reliable model of how the membrane really works, it only gives a physical meaning to the developed mathematical formulas.

4.1.1 Potassium conductance

In order to interpolate the measured dependence of the *potassium* conductance from the displacement voltage a fourth-order exponential equation is needed. To simplify the notation g_K is computed as the fourth power of a variable which obeys a first-order relation.

$$g_K = \overline{g_K} \cdot n^4 \tag{4.9}$$

$$\frac{dn}{dt} = \alpha_n \cdot (1 - n) - \beta_n \cdot n \tag{4.10}$$

where $\overline{g_K}$ is a constant value and n is a dimensionless value varying between 0 and 1. α_n and β_n are instead rate constants that depend on the membrane voltage, but not on time. They have dimension $[s^{-1}]$.

The physical meaning of the expression can be interpreted as follows: the transition of a *potassium* ion is allowed only if there are four similar molecules, called *activating molecules*, in the same position, for example inside the membrane. The quantity n represents the probability to find one of these molecules inside the membrane and (1-n) becomes the probability to find it outside the membrane. So the variation of the probability to find one molecule inside the membrane, with respect to time, is given by the probability to have it outside the membrane, multiplied by the rate of transfer between outside and inside, α_n, minus the probability to have it inside the membrane, multiplied by the rate of transfer from inside to outside. To consider the presence of four molecules the fourth power of n is considered.

4.1.2 Sodium conductance

A similar result can be obtained for the *sodium*. Also in this case the conductance is considered to be proportional to the quantity of ions contained inside the membrane. The condition for a *sodium* ion to cross the membrane is that there are three *activating* molecules inside the membrane. Differently from before however there is a second mechanism, called *inactivation*, that is caused by the presence of a specific molecule inside the membrane and blocks the transport of *sodium*. In this case the modification of the conductance, given by the transition of a *sodium* ion from inside to outside, is given by:

$$g_{Na} = \overline{g_{Na}} \cdot m^3 \cdot h \tag{4.11}$$

where m and h can assume values between 0 and 1 and respectively represent the probability to have an activating molecule inside the membrane and the probability

to have an inactivating molecule outside the membrane. As before the temporal evolution of m and h is described by:

$$\frac{dm}{dt} = \alpha_m \cdot (1 - m) - \beta_m \cdot m \tag{4.12}$$

$$\frac{dh}{dt} = \alpha_h \cdot (1 - h) - \beta_h \cdot h \tag{4.13}$$

4.2 Integrate and fire

The *integrate and fire* model is a very simple model of the temporal evolution of the membrane potential. It can be traced back to *Louis Lapique*[8], though he considered a resistive term in his analysis, so it would be better to say that he worked on a *leaky integrate and fire model* (see section 4.3). The first documented true integrate and fire model dates back to 1936[24]. Since then many variants and implementations have been studied, mainly thanks to the simplicity of the model.

The IF model treats the neuron's membrane as an ideal capacitor, perfectly isolated, as shown in figure 4.2. The current source represents the input from all the synapses. It will be better analyzed in chapter ... For now let's consider it as a constant current source for simplicity.

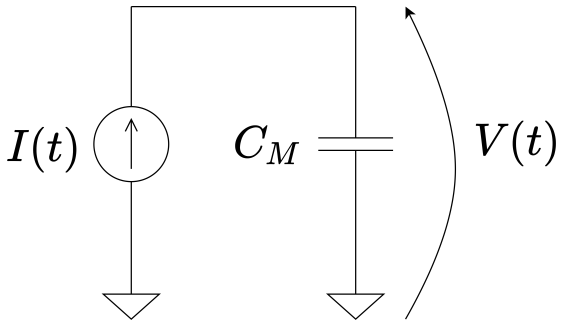


Figure 4.2: Integrate and fire equivalent circuit

The name of the model directly comes from its behaviour: equation 4.14 shows the relation between the input current and the voltage across the capacitor. First of all, inverting the expression in order to find the temporal evolution of the voltage leads to equation 4.15. It can be seen that the input current is integrated by the capacitor.

$$C_M \cdot \frac{dV}{dt} = I(t) \tag{4.14}$$

Secondary, whenever the voltage exceeds a fixed threshold it is reset to a rest value and an output current spike is generated.

$$V(t) = \frac{1}{C_M} \int_{t_0}^{t} I(t)dt \tag{4.15}$$

Figure 4.3 shows the temporal evolution of the membrane potential in presence of two different inputs:

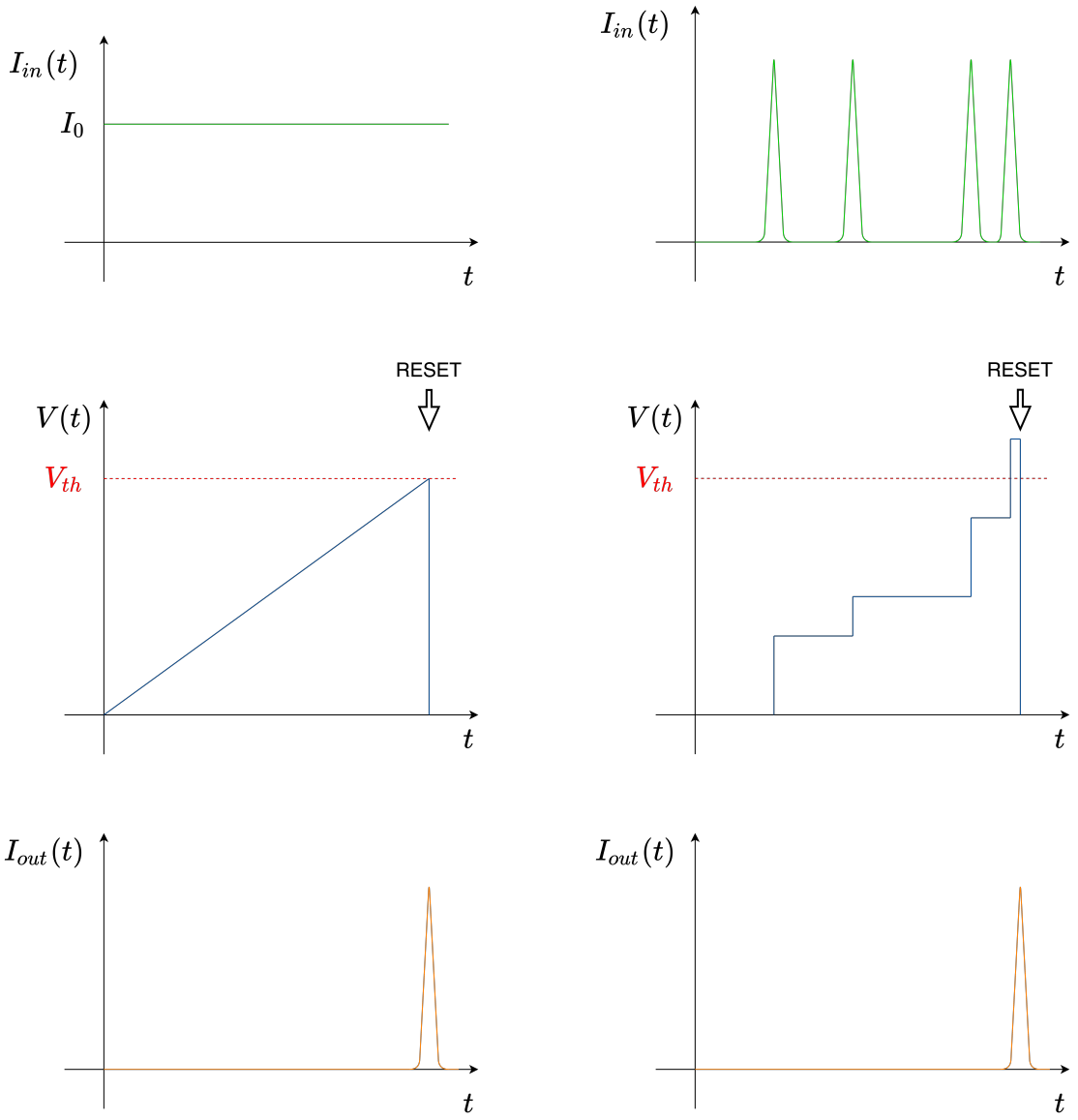
1. Constant current source, figure 4.3a.
2. Current spikes, figure 4.3b.

Let's start analysing figure 4.3a. With a constant input current the voltage increases linearly, being it the integral of a constant value. When it reaches the threshold voltage, reported in red it is reset to its rest value, in this case zero. In correspondence of this event an output current spike is generated by the neuron.

This is useful to understand how the model works, but in real applications the input source, corresponding to the input synapses, brings current spikes. The behaviour of the membrane potential in this case is reported in figure 4.3b. $I_{in}(t)$ reports the spikes coming from all the input neurons. It would be better to separate the inputs in different plots but being this a simple example, for the sake of visualization, they are condensed into a single plot. Whenever a spike arrives the membrane potential is incremented by a quantity that corresponds to the weight of the specific synapse. It can be seen that the reported spikes belong to different input neurons because the increments in the membrane potential in correspondence of them are not all equal. In particular it is sure that the second spike comes from a neuron different from the ones that have generated the other three. For what concerns these last elements, they can either come from a single neuron or from different neurons with the same synaptic weight. However this is not important for the current example.

The reset in figure 4.3b is delayed with respect to the excess of the threshold. The reason is that in practice there will be a control system that checks the value of the membrane potential at every computational cycle. In this case the control system checks the potential after it has exceeded the threshold due to the arrival of an input spike.

One feature that can be added to the model is the *refractory period*. In this case whenever the threshold is exceeded and an output spike is generated the neuron remains in a quiet state for a certain amount of time, called the *refractory period*. All the input spikes received within this time window are ignored and the neuron starts to be reactive to its inputs again only at the end of the time interval.



(a) Constant current input **(b)** Spiking current input

Figure 4.3: Membrane potential temporal evolution

4.3 Leaky integrate and fire

The integrate and fire model provides a first approximation of how the membrane potential evolves. Its simplicity is its main advantage, but it is also its main

weakness and makes the IF too rough as a model. In particular the ideal isolation hypothesized for the membrane is not coherent with its physical counterpart. As for every real system in fact, the membrane experiences a certain amount of leakage that tends to discharge its capacitance if it is not stimulated with any input. The *leaky integrate and fire* model includes this feature, modeling it with a resistor in parallel to the membrane capacitance, as shown in figure 4.4. The first studies on such a model can be traced back to Louis Lapique and his work on the frog legs[8].

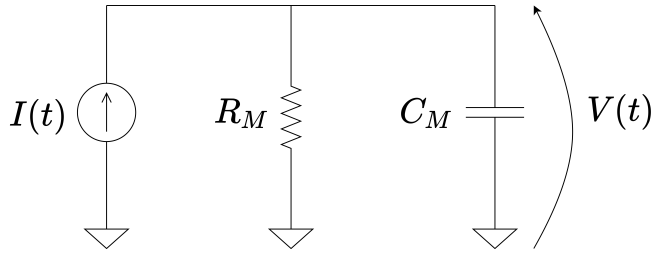


Figure 4.4: Leaky integrate and fire equivalent circuit

Figure 4.5 shows the temporal evolution of the membrane potential when it is stimulated by input current spikes. In this case the behaviour of the membrane has been simulated using a python script and the two inputs are represented on two separated plots. It can be seen that the potential decreases exponentially after input spike arrival, due to the finite resistance of the membrane.

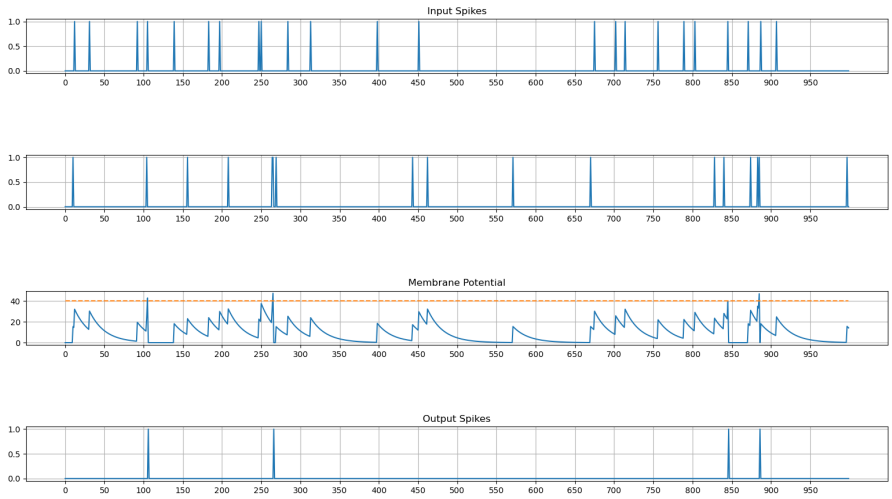


Figure 4.5: Membrane potential temporal evolution

Finally the rest potential has been supposed to be 0V. This means that, in absence of any input the membrane potential exponentially tends towards 0V. It is interesting to consider instead a rest potential with an arbitrary value because this allows to consider more realistic systems with voltage values that are more coherent with the real brain. Figure 4.6 shows the equivalent circuit in this specific case.

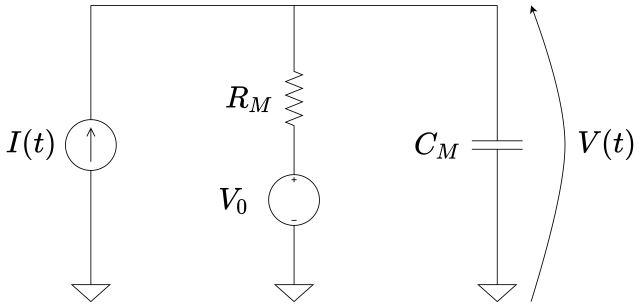


Figure 4.6: Leaky integrate and fire equivalent circuit with arbitrary rest potential

4.4 Izhikevich

Finally let's present an alternative model developed by *Eugene M. Izhikevich* in 2003[25]. It tries to combine the biological plausibility of the Hodgkin-Huxley model and the simplicity of the integrate and fire model. Here the model is briefly presented, but the complete mathematical derivation is reported in a dedicated book written by *Izhikevich* himself[26].

Using bifurcation methodologies the biologically accurate Hodgkin-Huxley model can be reduced to a two-dimensional system of ordinary differential equations:

$$\frac{dv}{dt} = 0.04 \cdot v^2 + 5 \cdot v + 140 - u + I \tag{4.16}$$

$$\frac{du}{dt} = a \cdot (b \cdot v - u) \tag{4.17}$$

All the variables in the equation are dimensionless and represent:

- v: membrane potential of the neuron
- u: recovery variable that accounts for the activation of *potassium* K^+ and the inactivation of *sodium* Na^+, providing a negative feedback to v.
- a: modifies the time scale of the recovery variable u. A smaller value of a corresponds to a slower recovery.

- b: sensitivity of the recovery to the membrane potential. Grater values of b couple v and u more strongly, possibly leading to subthreshold fluctuations of the potential and low threshold firing dynamics.
- I: contains all the input contributes, being them spikes brought by the synapses or a simple dc-current.

The quadratic part of equation 4.16, that is $0.04 \cdot v^2 + 5 \cdot v + 140$, has been derived in order to cover the evolution of different types of biological cells for large groups of neurons.

The part $\frac{dv}{dt} = 0.04 \cdot v^2 + I$ is sometimes called the quadratic integrate and fire part.

The after-spike reset instead is performed as follows:

$$v = c \tag{4.18}$$

$$u = u + d \tag{4.19}$$

where c and d represent the after-spike reset value of respectively the membrane potential and the recovery variable.

Chapter 5

Synapses

One of the most important characteristics of the mathematical models of the membrane potential of the neuron is the synapse, that directly affects the way in which the potential is updated when an input spike is received. There are two main types of synapses:

1. Current-based

2. Conductance-based

Let's analyze the two models separately. A simple *leaky integrate and fire* model with a rest potential V_{rest} will be used in the following sections to describe the membrane of the neuron.

5.1 Current-based synapse

In the current-based model each of the input neurons can be represented as a current generator. Each neuron generates the correspondent current in form of a train of spikes, that can be mathematically modeled as ideal Dirac deltas. The impact that these spikes have on the target neuron's membrane potential depends on the synapse that connect each source neuron to one of its inputs. To better visualize such a situation figure 5.1 can be considered. It represent each input neuron as a current source which generates a current in form of spikes, with a specific temporal pattern. The common characteristic between the different generators is the shape of the spikes, all identical in terms of their area.
The area of the spikes is given by the integral of the current with respect to time, as shown in equation 5.1, where t_0 and t_1 are two generic time instants such that

the current pulse is located within the two.

$$A = \int_{t_0}^{t_1} I(t)dt \tag{5.1}$$

What is interesting is that 5.1 corresponds to the charge carried by the current. So in other words each current spike brings the same charge to the membrane capacitance. As said before, what modifies the amount of charge that finally reaches the membrane is the synapse. In this sense each synapse can be seen as a charge amplifier with a gain corresponding to the weight of the synapse.

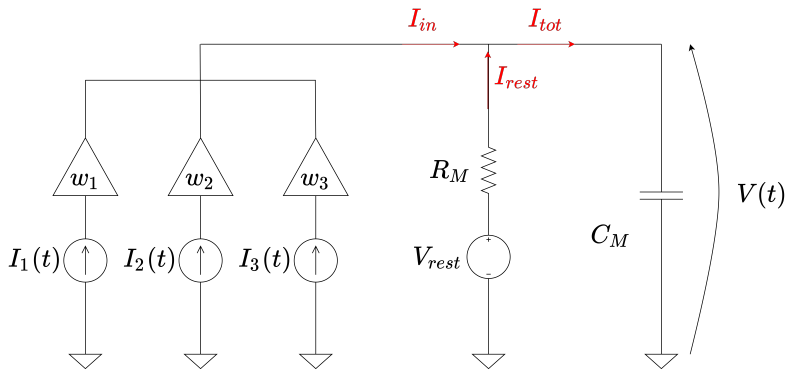


Figure 5.1: Equivalent circuit of the current based model

Appendix A reports a complete mathematical analysis of the model. The final result allows to describe the evolution of the membrane potential after the arrival of the input spikes, is reported in equation A.30. This suppose that the starting value of the membrane potential is equal to V_{rest}. More in general:

$$V(t) = V(t_0) + \left(\sum_{i=1}^{N} s_i \cdot w_i - V_{rest}\right) \cdot e^{-\frac{t-t_0}{\tau}} \tag{5.2}$$

where t_0 is the time instant in which one or more input spikes are received. Equation 5.2 leads to the temporal evolution of the membrane potential reported in figure 4.5.

5.2 Conductance-based synapse

The equivalent circuit for the conductance-based synapse model is shown in figure 5.2. In this case each synapse is modeled as a variable conductance. Whenever a spikes arrives to the synapse it increases its conductance. All the inputs are

connected to the same input voltage V_{in}, and so the amount of current that reaches the membrane capacitance depends on the conductance of the specific synapse. Increasing it increases the total current I_{in} and so the membrane potential $V(t)$.

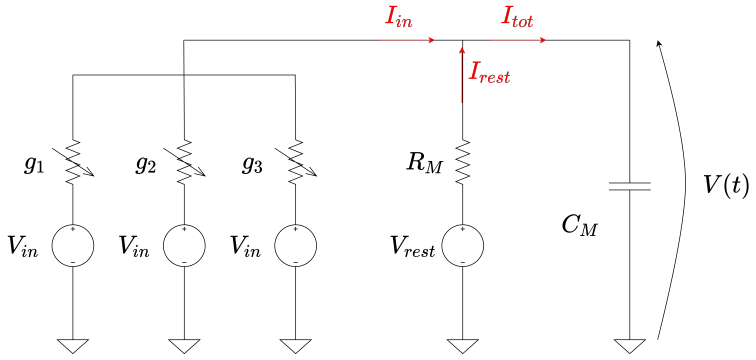


Figure 5.2: Equivalent circuit of the conductance based model

Generally the equation that describe the temporal evolution of the generic conductance g_i is:

$$\frac{dg_i}{dt} = -\frac{g_i}{\tau_i} \tag{5.3}$$

which simply represents a decreasing exponential function with steady state value equal to 0 and time constant τ_i. In other words, in absence of spikes the conductance of each synapse is null and so it brings no current to the membrane capacitance. As a consequence at the equilibrium this tends to stabilize itself at the value V_{rest}. When a spike is received the conductance is increased as:

$$g_i = g_i + w_i \tag{5.4}$$

For simplicity all the conductances are considered to have the same time constant τ. This allows to easily compute the parallel between them and to put them all together in a single conductance of value:

$$g_{in} = \sum_{i=0}^{N} g_i \tag{5.5}$$

where N is the total amount of input synapses. The equivalent circuit becomes the one reported in figure 5.3.

Adding together the temporal evolutions of the different synapses gives:

$$\sum_{i=0}^{N} \frac{dg_i}{dt} = -\sum_{i=0}^{N} \frac{g_i}{\tau} \tag{5.6}$$

Thanks to the linearity of the derivative and to the unique value of τ the expression can be rewritten as:

$$\frac{d}{dt}\sum_{i=0}^{N} g_i = -\frac{1}{\tau}\cdot\sum_{i=0}^{N} g_i \tag{5.7}$$

$$\Longrightarrow \quad \frac{dg_{in}}{dt} = -\frac{g_{in}}{\tau} \tag{5.8}$$

Also the contribute of the synapses' weights is linear and so the increment of the conductance in correspondence of the arrival of an arbitrary amount of spikes is:

$$g_{in} = g_{in} + \sum_{i=0}^{N} s_i \cdot w_i \tag{5.9}$$

where s_i represents the presence or absence of an input spike on the synapse i. See appendix A for more details.

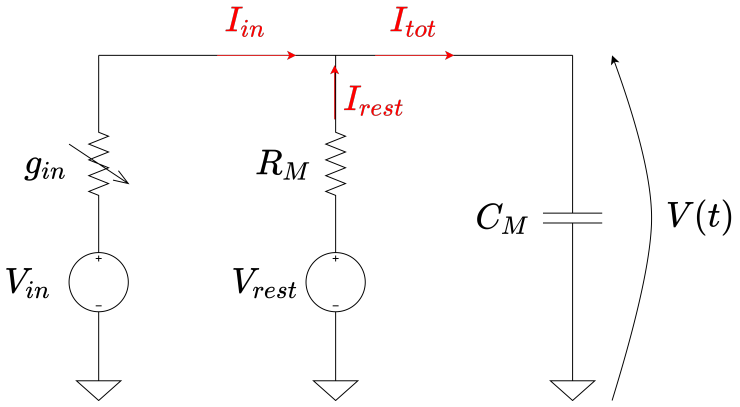


Figure 5.3: Equivalent circuit of the conductance based model with condensed synapses

The model of the conductance is very similar to the one obtained for the membrane potential and can be written as:

$$g_{in}(t) = g_{in}(t_0) + \left(\sum_{i=1}^{N} s_i \cdot w_i\right) \cdot e^{-\frac{t-t_0}{\tau}} \tag{5.10}$$

where t_0 represents the time instants in which one or more spike arrive on the input of the neuron.

Computing the temporal evolution of the membrane potential:

$$C_M \cdot \frac{dV(t)}{dt} = g_M \cdot V_{rest} + g_{in}(t) \cdot V_{in} \tag{5.11}$$

and substituting the expression of $g_i(t)$ reported in equation 5.10 it is clear that the model becomes non-linear and much more complex if compared to the current based synapse one.

The Hodgkin-Huxley model, reported in section 4.1 is an example of conductance-based model, in which an additional degree of complexity is included by considering the dependence of the conductance from the concentration of *sodium* and *potassium* activating and inactivating molecules inside and outside the membrane of the neuron.

Finally another example of conductance based model is the one developed by Peter U. Diehl and Matthew Cook[27] and used as a starting point and as a reference for this work. Appendix C reports the details of the model.

Chapter 6

Learning through Spike Timing Dependent Plasticity

6.1 Supervised and unsupervised learning

Now that both the membrane potential and the synapses models have been explained in detail, the last problem to consider is how to make the network learn to recognize an input pattern and to separate it from different ones. Nowadays the most diffused method to train a neural network on such a task is the backpropagation.

Backpropagation is an algorithm made famous by an historic article written by David E. Rumelhart, Geoffrey E. Hinton and Ronald J. Williams in 1986[27]. The techniques described in the article were already known and present in the scientific literature, but this specific article had the power to unify and diffuse the methods in form of a single elegant algorithm.

The method consists in providing the network with the input data, reading its output and, knowing in advance the expected result, trying to minimize the difference between it and the obtained one by back-propagating the errors computed on the output. This means to provide the network with both the input and the output data and make the network fit the output as faithfully as possible.

This kind of learning is called supervised because the training consists in imposing from outside a correction of the network hyperparameters, in particular the synaptic weights, in order to make its output similar to the desired one. At the moment

it doesn't seem to be the way in which the human brain learns. Not completely at least. To cite Geoffrey Hinton himself, if the backpropagation is such a good method to make a neural network learn, the natural evolution of the human brain should have developed it independently. So backpropagation, or a slightly modified version of it, should not be excluded a priori as a possible way in which the human brain learns. Nowadays the knowledge of the human brain is still too rough to make sure assumptions on the topic.

However the most promising and biologically plausible method at the moment seems to be the spike timing dependent plasticity[21][28][29].This is an unsupervised method in which therefore the network is provided with the input data and is left free to adapt to them, which is intuitively more similar to what happens in the human brain. When a visual stimulus, a dog for example, is received, the eyes translate it into proper pattern of electrical pulses that are then provided to the brain. The brain for its part analyzes the spikes and learns to associate them to a concept, the concept of dog in this case, in an autonomous way. There is not an external supervisor which tries to minimize the difference between the concept elaborated by the brain and the label "dog" by directly modifying the synaptic weights. This at least is the more diffused idea at the moment.

Spike timing dependent plasticity is only one of the many existing unsupervised methods. Section 3.6 for example presented a method based on the frequency with which the spikes are received. However STDP seems to be the most faithful model of the human brain learning process and is sufficiently simple to be practically used in an efficient way.

6.2 Spike timing dependent plasticity

As the name itself says the method is based on the arrival time of the spikes and in particular on the difference between the instant in which a spike is received on the input of the neuron and the instant in which a new spike is generated by the neuron itself. In particular:

1. If an input spike arrives a little before the generation of a new output spike the synapse weight is increased. The shorter the time difference between the two the higher the increment. In other words the synapse is rewarded if it is likely to make the neuron fire. This is called *long term potentiation*.

2. If vice versa the input spike arrives after the generation of the output one it means that the synapse had no role in making the neuron fire and so it is punished with a reduction of its weight. This is called *long term depression*.

The mathematical function that describes the two operations is reported in equation 6.1 and plotted in figure 6.1.

$$\Delta w = \begin{cases} A_{LTP} \cdot e^{-\frac{\Delta t}{\tau_{ltp}}} if\ \Delta t > 0 \\ -A_{LTD} \cdot e^{\frac{\Delta t}{\tau_{ltd}}} if\ \Delta t < 0 \end{cases} \tag{6.1}$$

where

$$\Delta t = t_{out} - t_{in} \tag{6.2}$$

So for positive time differences the synapse weight is increased. The decreasing exponential in this case imposes lower increments for higher time differences. This is because the farther in time the input spike is, the lower is the probability of its direct role in making the membrane potential exceed the threshold. For negative time differences instead the input has no role in the generation of the output spike.In this case the smaller is the time difference the higher is the reduction of the synapse weight.

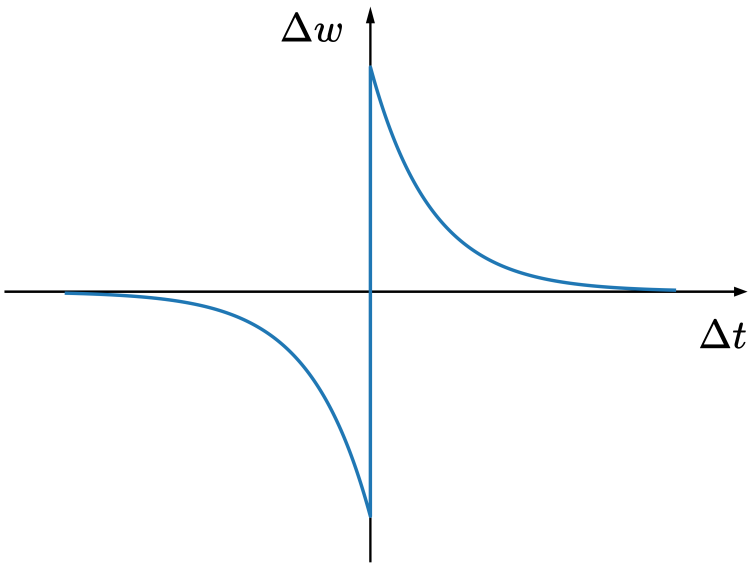


Figure 6.1: Plot of the STDP equation

Figure 6.1 shows a symmetric characteristic, that is the *long term potentiation* and the *long term depression* have exactly the same impact. In mathematical terms:

$$A_{LTP} = A_{LTD} \tag{6.3}$$

This is generally not true in practice. The characteristic can be asymmetric both in terms of amplitude and exponential time constants.

6.3 Synaptic traces

The STDP method is quite light as an algorithm but a direct implementation of it is not so efficient, in particular working with an hardware accelerator. In particular the algorithm require to know, for each neuron, the time instants corresponding to each input spike and to the output spike and it requires the the computation of an exponential function.

In order to simplify the computation the synaptic traces can be used. In particular for each input synapse there is a variable, the pre-synaptic trace a_{pre}, and for each output, so for each neuron, a second variable, the post-synaptic trace a_{post}, that track the arrival of the spikes. The update of the weights is performed as follows.

1. When an input spike is received:

$$\begin{cases} a_{pre} = A_{LTP} \\ w = w - a_{post} \end{cases} \tag{6.4}$$

2. When an output spike is generated:

$$\begin{cases} a_{post} = A_{LTD} \\ w = w + a_{pre} \end{cases} \tag{6.5}$$

The *pre-* and *post-synaptic traces* are then updated following an exponential trend,as shown in equations 6.6 and 6.7.

$$\frac{da_{pre}}{dt} = -\frac{a_{pre}}{\tau_{pre}} \tag{6.6}$$

$$\frac{da_{post}}{dt} = -\frac{a_{post}}{\tau_{post}} \tag{6.7}$$

As appendix B shows the method is mathematically equivalent to the more classical STDP equations. With such an alternative formulation however the exponential functions, characteristics of the STDP updating rule, can be computed in an iterative way, following the same method used for the membrane potential. Section ... shows the mathematical details. This makes the algorithm more suitable for a practical implementation, in particular with an hardware accelerator.

Chapter 7

Chosen model and structure of the neural network

Now that all the various parts of the model have been presented in their main different variants, the choices oriented to the realization of an hardware accelerator can be explained more in detail. All the parameters of the model are reported in section D.

7.1 Structure of the network

Let's start with the analysis of the general structure of the network. Figure 7.1 shows a simple graph representation of it. The main components of the network are:

1. Input layer: this doesn't correspond to a layer of physical neurons, but represents the input data provided to the network.
2. Excitatory layer: this is a layer of neurons, each implementing the chosen model. The "excitatory" adjective comes from the fact that the connections that start from these neurons all have a positive weight, leading to an increment in the target neuron's membrane potential.
3. Inhibitory layer: again this is a layer of neurons as described by the chosen model. The connections starting from them have a negative weight, leading to a reduction in the target neuron's membrane potential, and so to the "inhibitory" adjective. The layer has exactly the same amount of neurons of the excitatory one.
4. Synapses that connect the input layer to the excitatory one, or more in general two consecutive excitatory layers: here is where the learning takes place. The

weights of the network are initialized to random values and then increased or decreased following the *spike-timing dependent plasticity* method. The two layers in this case are fully connected. The connections are shown in green.

5. Synapses that connect the excitatory layer to the inhibitory one. Each excitatory neuron is connected to a single inhibitory neuron. The weight of the synapse is set to a constant positive value sufficiently high to make the neuron immediately fire. The connections are reported in red.

6. Synapses that connect the inhibitory layer to the excitatory one. In this case each neuron is connected to all the neurons of the excitatory layer, except to the one for which there is a connection in the opposite direction. The weight of the synapse is set to a constant negative value. The connections are reported in blue.

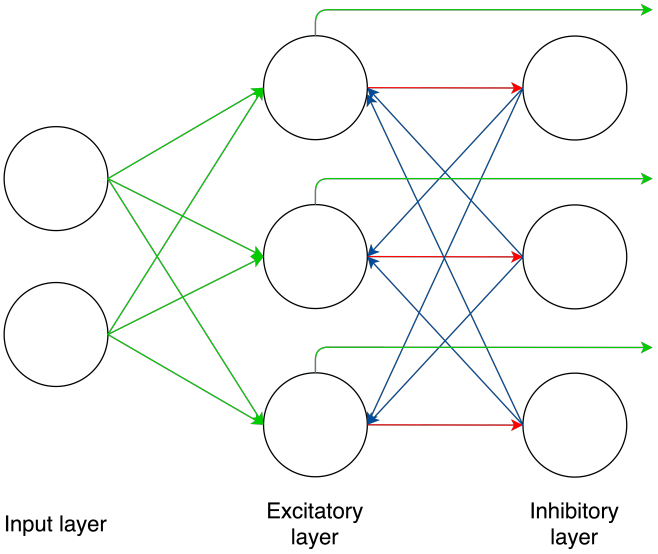


Figure 7.1: Structure of the network

So to sum-up the models consists of an arbitrary amount of excitatory layers, fully connected one to the other and to the input layer. This would lead to the general structure presented in figure 1.2. In addition however each neuron is connected to all the other excitatory neurons of the same layer through an inhibitory connection. The practical implementation of this leads to an additional layer of neurons, each connected to the corresponding excitatory element with a weight sufficiently high to make it fire whenever an excitatory spike is generated. In this case the spike leads to the reduction of the potential of all the other neurons in the excitatory layer through the inhibitory connections.

Note that this reference structure, with this specific number of neurons will be used in the following sections as a reference for the descriptions of the data structures and for the plot of the input and output of the various neurons.

The chosen structure has a single layer of excitatory and inhibitory neurons. The grater is the number of such neurons, the higher is the accuracy of the network. The goal is to find a total amount of neurons which allows to reach a sufficiently high accuracy, being at the same time small enough to fit into the target hardware platform.

7.2 Model of the neuron and synapses

The target of the whole project is an hardware accelerator able to reach a degree of parallelism sufficient to outperform its software counterpart. The main challenges in this sense are:

1. The limited hardware resources available for the design. Being the target of the accelerator an FPGA or an ASIC the amount of components is inevitably not infinite.

2. The computations required by a single neuron are quite complex, as shown in the previous chapter, depending on the chosen model. The expectation is to need a quite large amount of neurons to obtain an acceptable accuracy, so the overall resources requirements will surely be high.

As a consequence, the simpler is the neuron model the larger can be the network itself. Supposing then the requirements of the network, in terms of the total amount of neurons needed, to be sufficiently low to use only part of the available hardware components, the remaining resources can be used to further increase the parallelism of the neurons themselves, leading to faster computations. So in general a lighter algorithm is preferable.

For this reason the chosen model is the *leaky integrate and fire* with *current-based synapses*. The model is explained in detail in sections 4.3, 5.1 and A. Its equivalent circuit is reported in figure 4.6. I(t) in this case represents the weighted sum of the current generated by all the input neurons.

Now that the model is defined the temporal evolution of the membrane potential and of the output spikes can be analyzed more in detail. The reference structure for the following plots is the one reported in figure 7.1. The network has been provided

with random input spikes and its evolution has been monitored for 150 milliseconds. The parameters used for the simulation are the ones reported in section D. The neurons are numbered starting from the upper ones and going downward, starting from zero. Figure 7.2 shows the membrane potential and output spikes evolution of the first excitatory neuron, together with the input spikes, both from the input and the inhibitory layers. The threshold is plotted in orange.

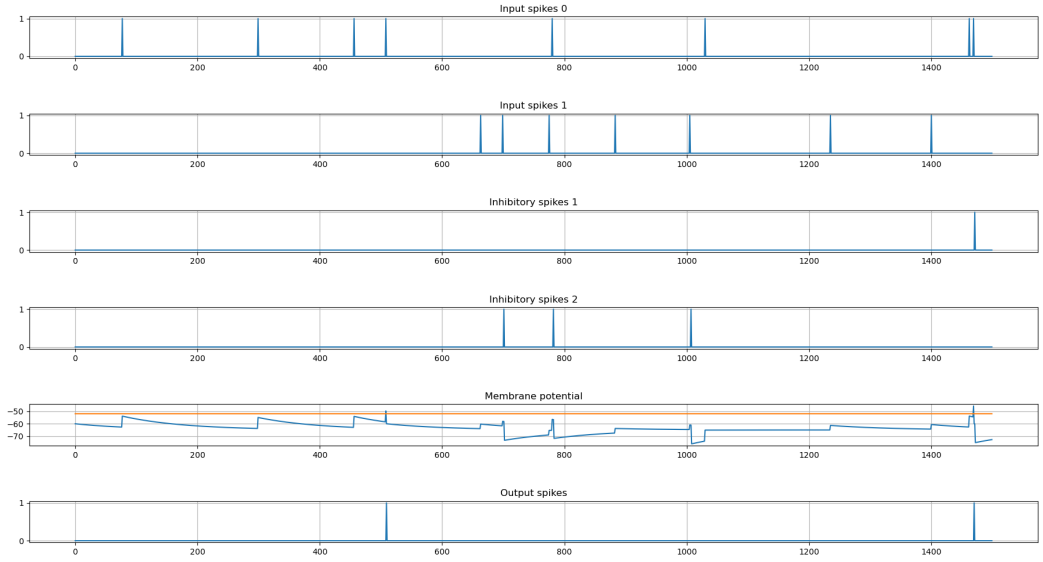


Figure 7.2: Input spikes with membrane potential and output spikes

To better visualize the evolution of the membrane potential figure 7.3 shows it isolated from the input and output spikes.

The membrane potential has been initialized to the reset potential, that is grater than the rest one. So the first thing to observe is that, in the first part of the evolution, in absence of input spikes the mmebrane tends towards its rest value. Whenever a spikes arrives from one of the two inputs the potential is increased and then starts again to decrease exponentially. If instead it exceeds the threshold it is reset to the reset value and then tends again toward the rest voltage. Finally when a spike arrives from one of the two inhibitory neurons the potential is decreased and then exponentially tends towards the rest value. If the potential goes below the rest value after the inhibitory spike exponential trend is inverted, as shown in figure 7.3.

Finally figure 7.4 shows the same kind of plot for the inhibitory neuron. In this case there is a single input, corresponding to the single excitatory neuron connected.

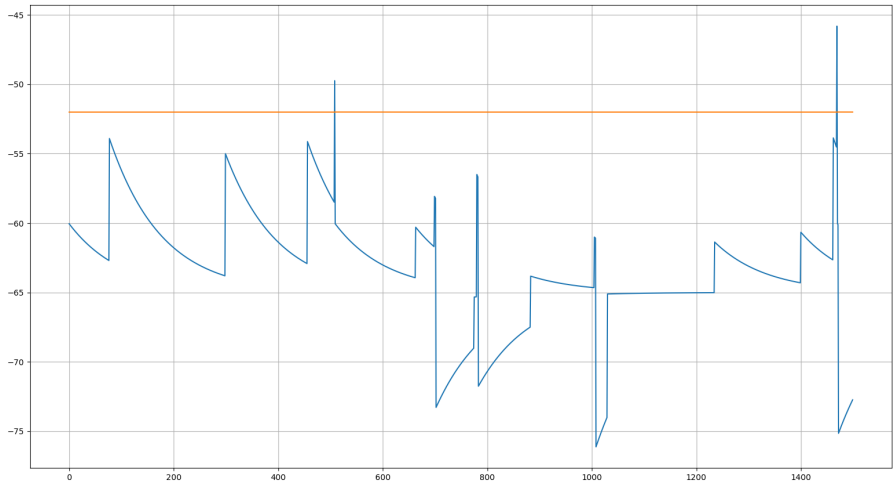


Figure 7.3: Membrane potential of the first excitatory neuron

It can be observed that a single spike is sufficient to make the potential exceed the threshold.

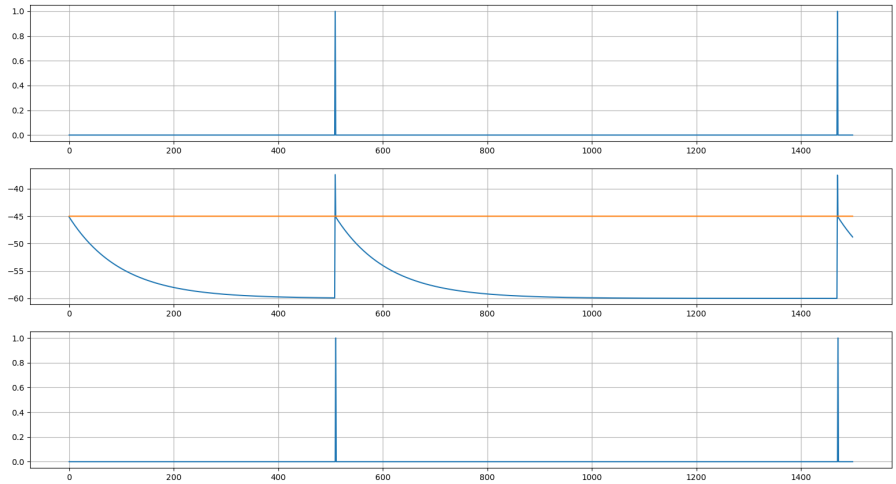


Figure 7.4: Membrane potential of the first inhibitory neuron

As a consequence the output spikes pattern faithfully follows the input one, sending

an inhibitory spike to neurons one and two whenever neuron zero generates an output spike. Again the reset voltage is higher that the rest one and this explains the decreasing exponential both at the beginning and after the reset.

7.3 Homeostasis

The risk in working with inhibitory spikes between the neurons of the same layer is that if a specific neuron is frequently active it can completely mask the others, impeding them to generate output spikes. To avoid this an additional characteristic is added to the model: the homeostasis.

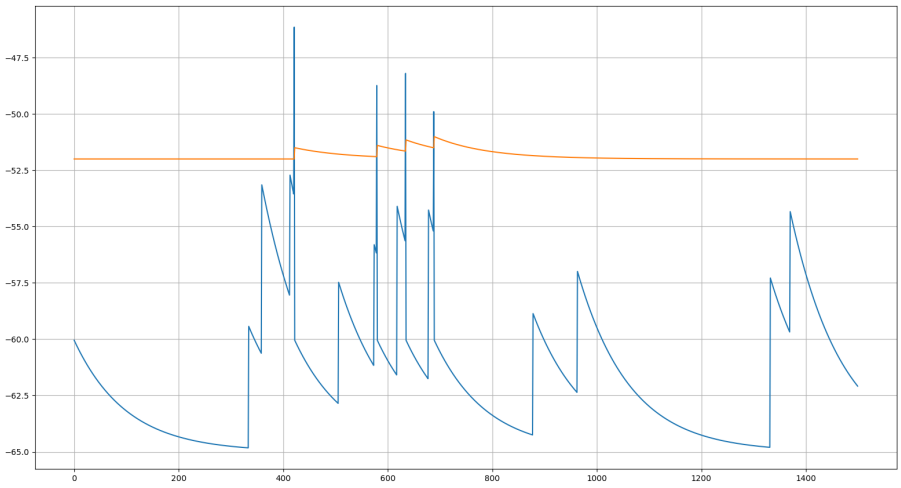


Figure 7.5: Homeostasis

The homeostasis is the natural tendency, which characterize all the living organisms, to reach a relatively stable state independently from the output environment. In this specific case the stability is represented by a uniform spiking activity for all the neurons within a layer, avoiding dominating nodes that block all the others. The parameter through which such a uniformity can be achieved is the firing threshold. The more frequently a neuron fires the more its threshold is increased. If however it becomes too high the neuron is no more able to generate new output spikes and so the problem persists. For this reason the threshold is exponentially decreased up to a rest value in absence of output spikes. The temporal evolution of the threshold

is reported in equations 7.1 and 7.2.

$$\frac{d\theta}{dt} = -\frac{\theta}{\tau_\theta} \tag{7.1}$$

$$V_{thresh} = V_{thresh}^{rest} + \theta \tag{7.2}$$

Whenever an output spike is generated:

$$\theta = \theta + \theta_+ \tag{7.3}$$

Figure 7.5 graphically shows what explained up to now. In this case the parameter θ_+ has been increased to 0.5, an higher value with respect to the one used in the real model, in order to better visualize it.

7.4 Learning

The learning method used to train the network is the *spike timing dependent plasticity*, in its version with the *synaptic traces*, explained in sections 6.3 and B.

Let's again analyze the method in a graphical way. Figure 7.6 shows the pre-synaptic trace evolution. In this case whenever an input spike is received the trace is set to A_{LTP}. In absence of new spikes it is then exponentially decreased.

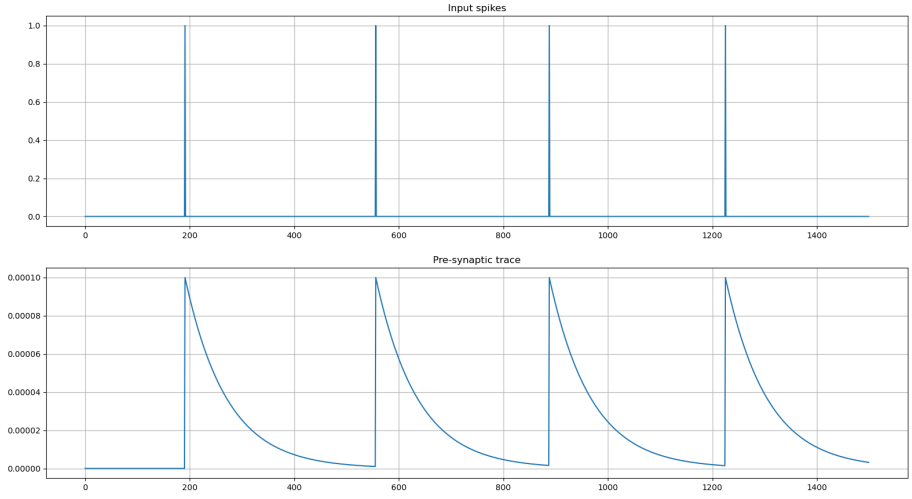


Figure 7.6: Pre-synaptic trace

The same behaviour is true for the post-synaptic trace in presence of output spikes. In this case the value of A_{LTD} is forced to be negative in order to make the visualization clearer. In this case the post-synaptic trace will be added to the corresponding weight. In practice A_{LTD} is positive and the post-synaptic trace is subtracted from the weight, leading to a mathematically equivalent result.

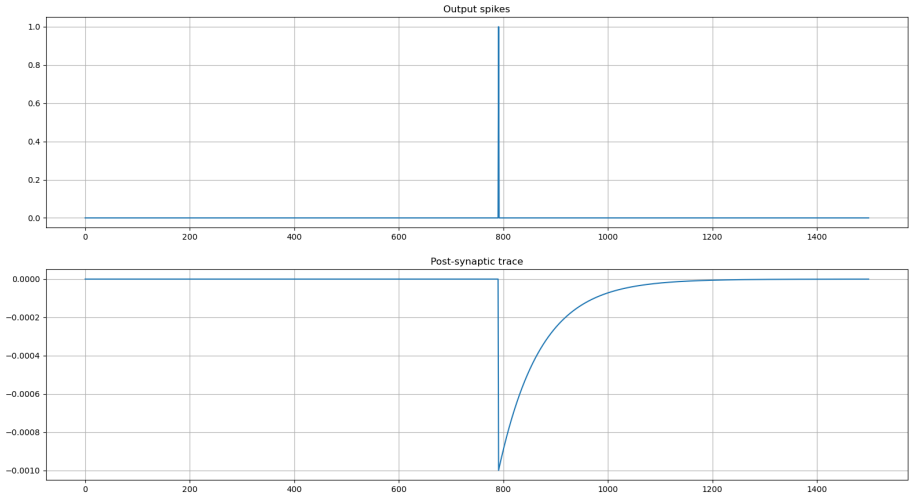


Figure 7.7: Post-synaptic trace

Also in this case the parameters have been slightly modified to make the representation clearer. In particular the exponential time constant has been set to 80ms instead of the 20ms used in practice.

7.5 Normalization of the weights

The unsupervised learning method described in the previous section reveals too slow to be used in practice. To avoid the problem a normalization step is included along the training. This is a weak point of the model for two main reasons:

1. There isn't a biological demonstration of such a process. Each synapse is increased or decreased by itself, without having a shared mechanism that keeps the various weights limited through normalization.

2. It is computationally heavy because it requires a division and a subsequent multiplication.

The normalization allows also to keep the weights limited, avoiding their uncontrolled growth along the training.

Chapter 8

Generation of the input spikes

Up to now the model has been described supposing to work with generic spikes trains as input. One last point to analyze, before proceeding with the practical simulation of the model, is how to convert the input data into such format. Generally in fact the data are not natively encoded into binary spikes, unless for example the sensor used to obtain them is designed to directly generate an output in this form. They are usually represented in form of numbers, being them integer, fixed point or floating point.

The goal is to obtain an encoding of the input spikes that reflects the input data and at the same time is coherent with the typical stimuli of a biological system. Poisson processes come in handy in this sense[28].

8.1 Poisson processes

A Poisson process is a particular type of stochastic process which simulates the succession, continuous in time, of events that are independent one from the other[29]. Moreover, such events have a probability distribution called a Poisson distribution.

A Poisson distribution is a discrete probability distribution which describes the probability for a certain number of independent events to happen within a determined time window, knowing that λ events take place on average within a known time interval[30].

In practice the input data, are directly associated to the parameter λ. In other words each input number is treated as an average spiking rate. So the higher is the

number the higher will be the amount of spikes within a fixed time interval. The spikes are then randomly generated within such an interval following a Poisson distribution with λ as a parameter.

8.2 Generation of spikes trains with a Poisson distribution

As said before each input number is interpreted as the input firing rate, that is the average frequency with which the spikes are presented to that specific input. The question now is: how to convert such a frequency into a train of randomly distributed binary spikes?

Both the python simulation and the hardware accelerator work in a discrete-time domain. This means that the time dimension is divided into small intervals of fixed duration and the elaboration is performed at the end of each of these intervals. The maximum amount of spikes that can be received within a single interval, being it the smallest possible temporal resolution, is one. On the contrary the minimum amount of spikes is obviously zero. So in general the probability to have a spike within a single time interval can assume any value between zero and one.

The input frequency can be normalized by the duration of the time step, in order to find the average amount of spikes generated within the interval.

$$Spikes\ per\ interval = (Input\ frequency) \cdot dt \tag{8.1}$$

At this point the input numeric data has been translated into the average number of spikes per time step. If the probability to have a spike within the time interval has a uniform distribution between zero and one the average number of spikes can be directly interpreted as the probability for a spike to arrive within the time interval. This is true supposing:

$$(Input\ frequency) \cdot dt < 1 \tag{8.2}$$

This means that if the frequency is grater than the maximum achievable with the chosen time step the conversion cannot be performed, which is quite an obvious requirement.

So in practice the conversion is performed as follows:

1. Normalize the input data multiplying it by the time step duration.

2. Generate a series of random numbers with uniform distribution between zero and one. The length of the series corresponds to the duration of the spike train in time steps. This can be obtained as:

$$Time\ steps = \frac{Time\ duration\ of\ the\ train}{dt} \tag{8.3}$$

3. Compare the input number with each generated value. If the input is grater than the random number then generate a spike, that is set the output to *boolean True*. If instead it is lower set the output to *boolean False*. The uniform distribution of the random number makes the probability to generate a spike exactly equal to the input number.

Chapter 9 presents the format of the input data. In the analyzed case each input data is associated with a pixel and represents its position onto a grey scale, going from total black to total white. The described method however can be used also with different kinds of data, being them audio samples coming from a digital microphone or temperature values read from a proper sensor. In general it can be applied for any kind of data that can be represented in a numerical form.

Chapter 9

MNIST dataset

The dataset used to train and test the developed model is the *MNIST*[31] (Modified National Institute of Standards and Technology). It is a database of handwritten digits, frequently used to benchmark machine learning applications. It has been chosen because of its simplicity and the large amount of projects that use it and that can be used as a reference for the accuracy and performances obtained with the developed model.

9.1 Dataset content and charateristics

The MNIST, as the name itself highlights, is a modified version of the NIST, a database of handwritten digits and characters. The data consist in black and white images with a resolution of $28 \cdot 28$ pixels. Each image correspond to an handwritten number between 0 and 9 and is associated with a label, that indicates the represented number. The data are organized as follows:

1. 60000 images, half of which taken from the training part of the NIST and half taken by the test part.

2. 10000 images, organized in the same way, that is half from the training set and half from the test set of the NIST.

The choice to split the NIST training and test data depends from the fact that the training set was collected among Census Bureau employees, and so it is composed by much clearer and easier to recognize data. On the contrary the test set was collected among high-school students and so its data are harder to interpret. In order to make the learning algorithm independent from the choice of the training and test sets the data have been mixed. Figure 9.1 shows some examples of the images contained in the MNIST dataset.

Figure 9.1: Example of mnist images, Josef Steppan, CC BY-SA 4.0, via Wikimedia Commons

As said before both the training and the test data consist of images and labels associated to them. These are organized in four different files, two for the training and two for the test:

1. train-images-idx3-ubyte: training set images
2. train-labels-idx1-ubyte: training set labels
3. t10k-images-idx3-ubyte: test set images
4. t10k-labels-idx1-ubyte: test set labels

9.2 IDX file format

The four files containing the entire database of handwritten digits are stored in IDX format, commonly used to store vectors and multidimensional matrices of various numerical types. It is worth to note that the bytes inside the file are stored in high-endian order (MSB first), suitable for non-Intel processors. Working on an Intel platform groups of bytes, for example forming a 32 bit integer, must be reversed in order to be correctly interpreted. The file is composed by an header

part, which describes the characteristics of the data, and by the data themselves, as follows:

magic number
size in dimension 0
size in dimension 1
...
size in dimension N
data

9.2.1 Magic number

The magic number is a 4 byte (32 bit) integer, with the first two bytes fixed at 0 and the second two bytes respectively encoding:

1. The type of the stored data. It uses a specific encoding for each data type:
   - 0x08: unsigned byte
   - 0x09: signed byte
   - 0x0B: short (2 bytes)
   - 0x0C: int (4 bytes)
   - 0x0D: float (4 bytes)
   - 0x0E: double (8 bytes)
2. The number of dimensions of the data structure. The simplest way to store the data is in form of a single dimension array, that is a sequence of data in the specified format. In this case the dimension is 1. A dimension of 2 corresponds to an array of arrays, which is a matrix, and so on.

9.2.2 Data dimensions

These are again four bytes integers, expressing the dimensions of the data structure stored into the file. They count a number of elements which corresponds to what reported in the magic number.

In order to better understand how the information is encoded in the header and how data are organized in the MNIST database, tables 9.1 and 9.2 report the the first section of the training images and labels files. It can be seen that the images are stored in form of a three dimensions data structure: this consists of an array of images, each image stored as a two-dimensional matrix with 28 rows and

28 columns. Each pixel corresponds to an unsigned byte with a value between 0, complete white, and 255, complete black. Figure 9.2 shows the $28 \cdot 28$ pixels image corresponding to the number 5, with the numerical values of the pixels reported.

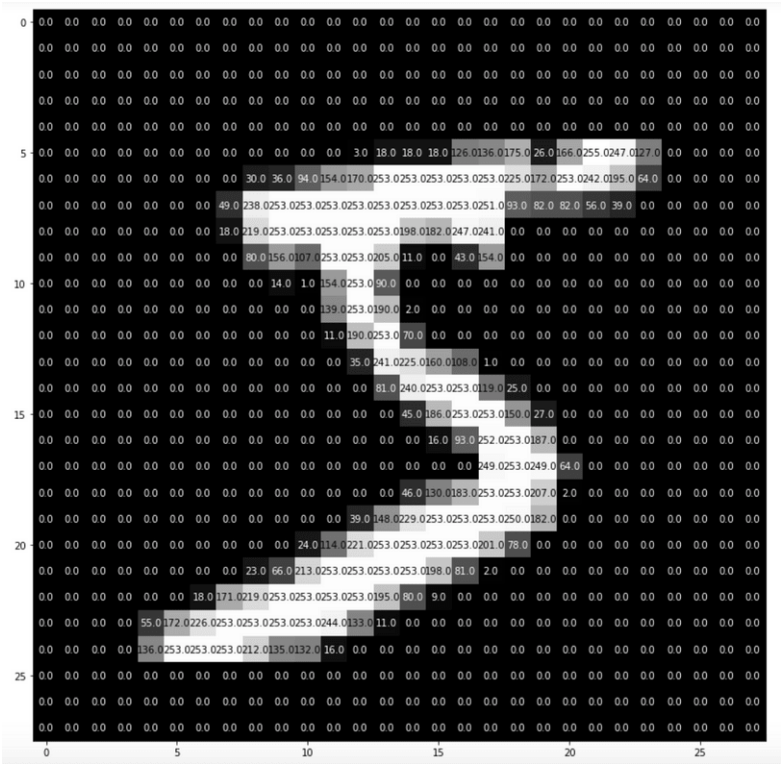


Figure 9.2: Mnist number with numerical values of the pixels[32].

The information about the data type and the number of dimensions can be seen encoded in the magic number as 0x08 (unsigned byte) and 0x03 (three dimensions).

offset	type	value	description
0000	32 bit integer	0x00000803	magic number
0004	32 bit integer	60000	dimension 0
0008	32 bit integer	28	dimension 1
0012	32 bit integer	28	dimension 2
0016	unsigned byte	first byte	data
0017	unsigned byte	second byte	data
...	...	...	...

Table 9.1: MNIST images file initial section

The situation is similar for the labels, but here each label is stored in form of an unsigned byte, sufficient to represent values between 0 and 9. For this reason the dimension is one, information encoded in the less significant byte of the magic number (0x01).

offset	type	value	description
0000	32 bit integer	0x00000801	magic number
0004	32 bit integer	60000	dimension 0
0008	unsigned byte	first label	data
0009	unsigned byte	second label	data
...	...	...	...

Table 9.2: MNIST labels file initial section

9.3 Algorithm to load the MNIST dataset

In order to be able to use the data to train the model they need to be loaded from the file in which they are stored into a proper data structure. For this aim the python *numpy* (numerical python) library is used. This comes with a useful data structure, the *numpy* array, that is a classical array with the desired number of dimensions, particularly suitable for the task at issue.

The flow followed to load the data is the following one:

1. First of all the entire content of the file is loaded into a memory buffer as row bytes.

2. The magic number is then read and decoded in order to extract the number of dimensions of the data structure, and so the number of four bytes integers to read, and the data type.

3. The proper amount of dimensions is read and stored in form of a *numpy* array.

4. At this point, with all the information needed, the data are read in block and loaded into a single-dimensional *numpy* array.

5. Finally the image block is reshaped in form of a two-dimensional *numpy* array. In this way the images are stored as entries of a *numpy* array, in form of single-dimensional *arrays* with $28 \cdot 28 = 784$ elements. This is a shape that is suitable for the training or test of a neural network: the input layer of the net will be composed of 784 elements, one for each pixel, as shown in figure 9.3.

Reshaping the images in their original format with 28 rows and 28 columns would be less effective.

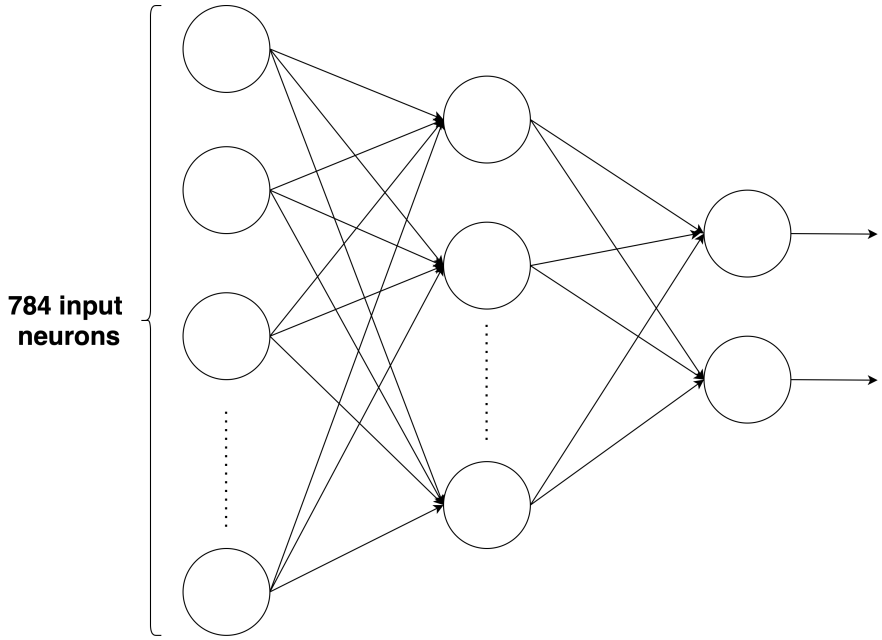


Figure 9.3: Example of a neural network with a single input layer with 784 elements.

Appendix F shows the flowcharts with the detail of the algorithm used.

Chapter 10

Python interface

The spiking neural network can be seen from the outside as a black box which receives spikes as an input and returns spikes as an output, as shown in figure 10.1. As said before however traditional systems do not work with binary spikes but with numbers. For this reason, before opening the black box and analyzing the implementation of the neural network at different levels of abstraction a proper way to interface it must be define.

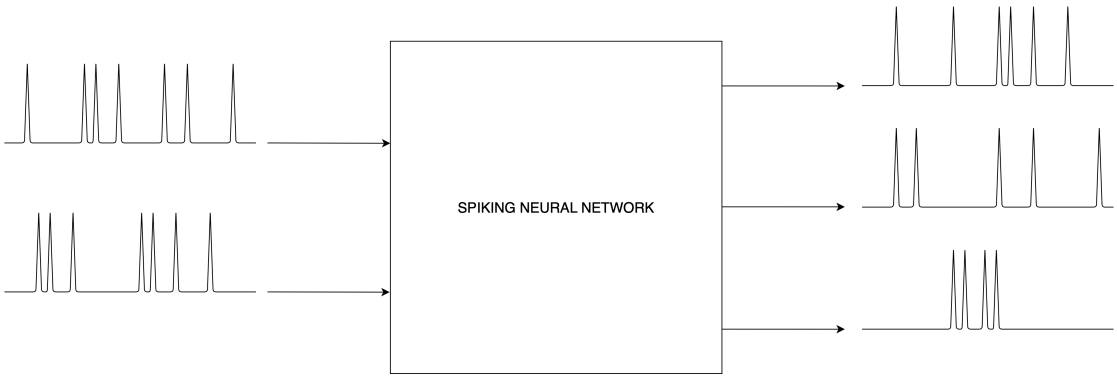


Figure 10.1: Spiking neural network as a black box

To develop such an interface python language has been chosen. Sections 10.1 and 10.2 briefly explain some of the main reasons for this choice, while next sections explains the main implementation details. Finally appendix G and H report the complete algorithms used to train and test the network, treating the neural network as a black box.

10.1 Why python

Python is an interpreted programming language, first in the ranking of the most used programming languages in 2021. It is well documented and, thanks to its interpreted nature, it works at a quite high level of abstraction, allowing the programmer to perform quite complex tasks with few instructions and without the need to worry about the low level implementation details. This characteristic will be better clarified in section 10.2. It is portable over about any platform and this makes it very versatile.

Nowadays there are lots of *Python* libraries that simplify the programmer's work by providing high level functions that unburden him/her from the technical implementation details. To stay on the machine learning topic, for example, there are a lot of frameworks, so as *TensorFlow*, *Keras*, *Torch*, *Theano* and so on, that allows to create, train, optimize and test a neural network without the need to manually implement all the mathematical details on which the network is based. Libraries like these can save a lot of programming time during the development of a model and this is only a specific example: there are libraries to do almost everything and this makes the difference in the choice of *Python* over other programming languages. One of these library is Brian 2[33], specifically designed for spiking neural networks. So in the development phase having an external interface developed in the same language of the model itself makes the interaction between the two easier and reduces the design time.

Last but not least *Python* is completely open source.

The version used to simulate the model is *Python* 3.8.5, that is the most recent one available with *Anaconda 3* in July 2021. *Anaconda 3* is used because of the simplicity in the installation of new libraries and for its portability.

10.2 NumPy and vectorization

NumPy is a python mathematical library, the name of which stays for *Numerical Python*. It has a lot of high level mathematical functions available. For example an exponential function with the Neper's number e as a base can be performed as follows:

```
1  import numpy as np
2
3  np.exp(x)
4
```

Listing 10.1: Python NumPy example

One of the most useful features of *NumPy* is what is called *vectorization*. To understand what it means let's first briefly analyze how Python code is executed.

In section 10.1 it has been said that Python is as interpreted language. This means that there is an external program, called the interpreter, that reads the Python script and executes the instructions contained in it one by one. This allows to perform complex instructions at an higher abstraction level, the interpreter will simply translate it in a sequence of simpler instructions, long as needed, that the computer's processor is able to understand. This translation however implies a computation overhead which makes performance poor with respect to compiled languages. To overcome the problem NumPy provides the *array* data structure. This is a NumPy specific implementation of the classical array data structure. It can have multiple dimensions and so allows to create single-dimensional arrays, two-dimensional matrix and in general multi-dimensional vectors.

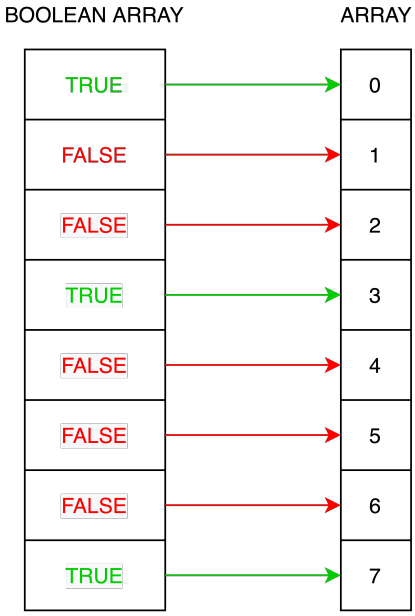


Figure 10.2: Example of NumPy boolean addressing

The *vectorization* consists in the possibility to apply a specific function to an entire array of the above type. So for example in listing 10.1 x can be a multi-dimensional array and is not required to be a pure number. The big advantage is that in this way the interpreting step is performed only once for all the elements of the array, and then the required operation is executed in loop over the whole data structure. The bigger is the input array the smaller is the relative impact of the interpreting overhead.

Vectorization can be also used to obtain a parallel addressing of the arrays, called *boolean addressing*. Instead of explicitly looping over the whole array and manually select the elements that respect some predetermined criteria, NumPy allows to use a boolean array as an index. Schemes like the one reported in figure 10.2 will be used along this chapter to better visualize the parallel access or operations applied to an array.
It is worthwhile to note that figure 10.2 does not mean that the operation is executed in parallel on the entire array in practice. The loop is simply moved to a lower level of abstraction and executed in a faster language, generally C. If then the processor makes a certain degree of parallelization available, NumPy tries to optimize the execution by using it. However this kind of scheme can be useful to understand what is going on at Python level.

10.3 Interfacing the network from the input side: encoding the input data into proper spikes trains

Chapter 8 presents a complete way to convert a real number into a train of spikes. The practical implementation of the method is the same used by *Peter Diehl* and *Matthew Cook*[27].
The rate of each spikes train is set equal to a normalized version of the pixel's value, following equation 10.1

$$rate = \frac{pixel's\ value}{8} \cdot inputIntensity \tag{10.1}$$

Where *inputIntensity* is a variable parameter initialized by default to

$$inputIntensity = 2 \tag{10.2}$$

The obtained values is interpreted as a frequency, so is expressed in Hertz. The pixels can assume values between 0 and 255. So, with the initial value of the *inputIntensity* reported in 10.2 the input firing rate is limited between $0Hz$ and $63.75Hz$.
In order to effectively interpret the output provided by the network a minimum spiking threshold is set. If the total count of spikes generated by the network as an output is lower than the threshold the input data are presented again, with the intensity increased by one. The process is repeated until the minimum output spikes count is reached. By default the minimum firing threshold is set to:

$$countThreshold = 5 \tag{10.3}$$

10.4 Interfacing the network from the output side: interpreting the output spikes

The other aspect of the interface regards the interpretation of the spikes generated by the network as an output. Also in this case the conversion from spikes to real numbers consists in considering the average firing rate of each output neuron. To do this the output interface simply counts the number of spikes generated by each neuron in the output layer. In order to obtain the corresponding firing rate it is sufficient to divide the obtained values by the total duration of the temporal window along which the output is analyzed. This corresponds to the time duration of the input spikes trains. Since this quantity is the same for each neuron normalizing the spike count by such duration does not bring any additional information. For this reason the computation is not performed and the spikes counts are directly used to interpret the output of the network.
So the complete structure, with the input and output interfaces becomes the one reported in figure 10.3

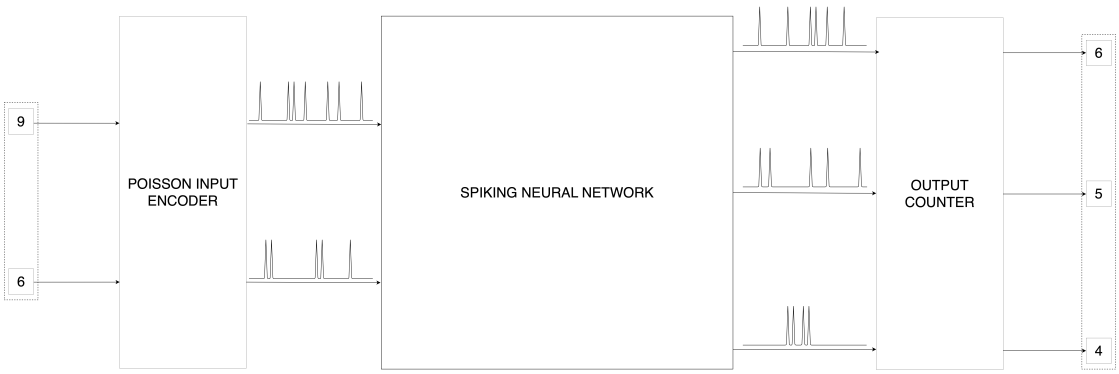


Figure 10.3: Spiking neural network with input and output interfaces

10.5 Training cycle over a single image

Now that information can be exchanged with the spiking neural network a training procedure can be defined. Figure G.2 shows the steps performed to train the network over a single image of the dataset. In order to make the training over a new image independent from the precious one the network is run for a resting period with its input forced to zero. In this way no input spike is provided to the network and the membrane potential tends towards its resting value, without being affected by the previous training cycle.
As said before the training method used within the network is the *spike timing*

dependent plasticity, which is an unsupervised method. This means that the input labels are not directly used during the training to modify the network hyper-parameters. However a method to interpret the output spikes counts and to use them to evaluate the accuracy of the network must be set.

10.6 Output classification

First of all each neuron must be associated with a label. In the ideal case when a specific image is presented at the input of the network only the neurons associated with the label corresponding with such image fire, while the others remains silent. The real situation is obviously less perfect than this, there will be neurons that will fire when an image different from their label is presented, and neurons associated with the right label that will not fire. So it would be better to say that each neuron is associated to the label for which the firing probability is the highest.

At the beginning of the training there is no way to estimate the output classification and so each neuron in the output layer is associated with a label that is different from all the ones available within the dataset. So by default all the outputs are associated with the label -1.

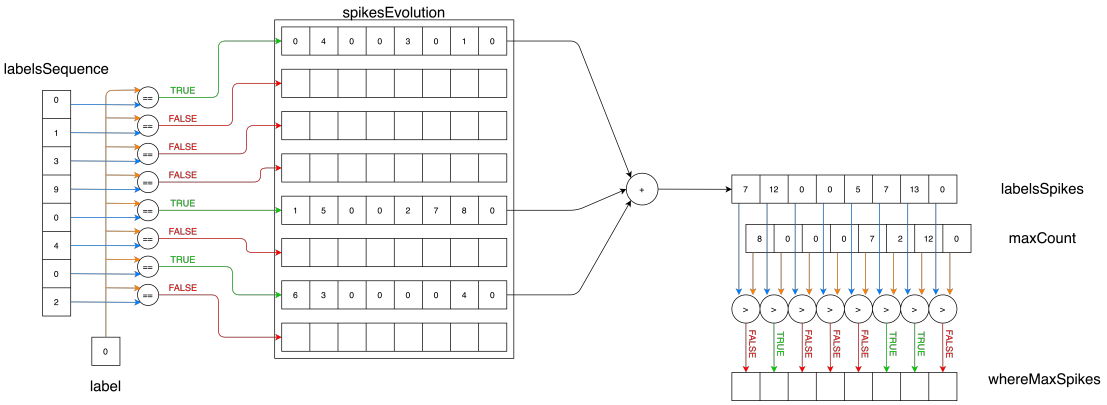


Figure 10.4: Update the assignments of the output layer

Along the training the number of spikes generated by each neuron is monitored and each of them is associated to the label for which the count is the highest. The label update is performed periodically after a fixed amount of input images. In this way *vectorization* can be used to improve the performance by analyzing the history of the network during a certain amount of cycles as a whole.

Figure 10.4 shows the procedure used to associate the neurons to the label zero, while figure G.8 reports all the steps required for the computation. The label zero

is used only as a reference but the update loop is obviously repeated for all the labels.

It can be seen that the maximum count associated to each neuron is reset before performing the classification. In this way the update is not affected by possible errors performed at the previous iteration. It is possible in fact that a neuron generates an anomalous amount of spikes in correspondence of one specific image and is therefore associated with the corresponding label. It could be however that this is not the correct label for which the neuron will generate the maximum amount of spikes on average. Updating the maximum counts along the images without periodically reset it would cause the anomalous spikes count to mask the subsequent iterations, implying a wrong classification.

10.7 Accuracy evaluation

Now that the output classification is determined it can be used to estimate the accuracy of the network.

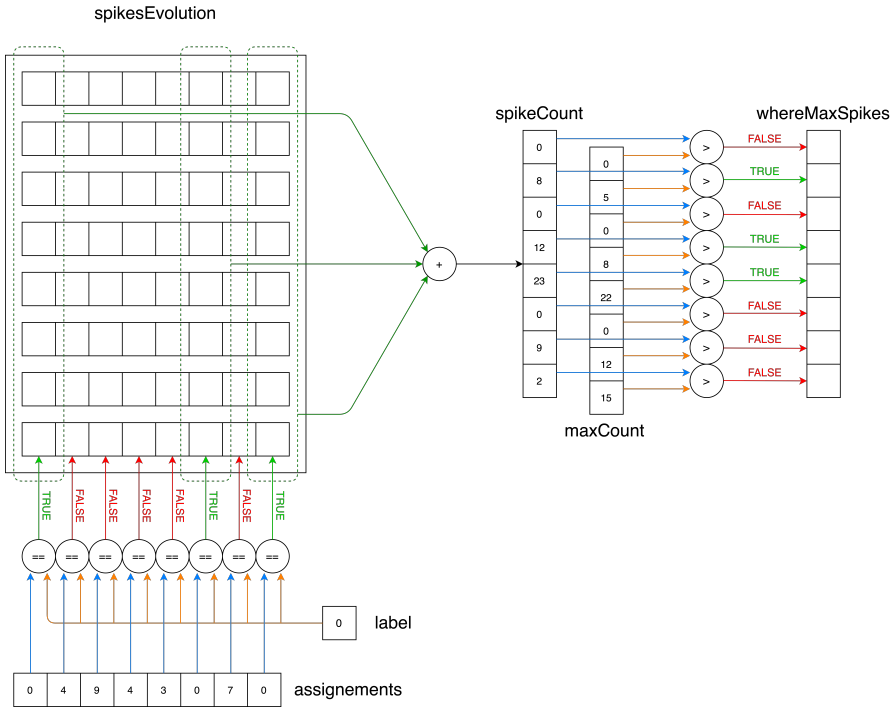


Figure 10.5: Find the instants for which the spikes count is grater than the maximum for label 0

Again the evaluation is performed periodically both to exploit *vectorization* and to

keep trace of the evolution of the accuracy along the training/test. The accuracy is estimated by monitoring the classification performed by the network for a fixed amount of images and by comparing it to the known sequence of correct labels associated to such images. The result is expressed in percentage as:

$$\frac{correctly\ classified\ labels}{total\ number\ of\ labels} \cdot 100\% \tag{10.4}$$

Figure 10.5 shows the *vectorized* implementation of method, while figure G.6 reports all the steps required for the computation.

10.8 Store the hyper-parameters of the network

Finally, once the network has been trained all its hyper-parameters are stored into proper files in order to be used later on in case the network is run in test mode. Figures K.1 and K.2 for example show how, during the initialization of the network hyper-parameters, these can be loaded from file in test mode.

Chapter 11

Simulation of the network using Brian 2

11.1 Brian 2 simulator

Brian 2 [33] is a simulator for spiking neural networks written in Python. It is the second version of the *Brian simulator* [34]. Nowadays there are lots of available simulators for spiking neural networks[35], such as NEURON[36], GENESIS[37] and NEST[38]. All of them are optimized to efficiently simulate SNN and are widely used for large-scale modeling and complex biophysical models but they generally use a dedicated language. This implies a relatively long time overhead to learn how the simulator works and to practically write the required code. Machine learning is generally a quite iterative process, in which the developer tries an idea, models it in using a certain coding language and then simulates it to evaluate the results. This loop is repeated many times in order to tune the network's hyperparameters, to test different architectures and so on. An optimization in the development procedure should involve all the steps of such a process. If for example the simulator is perfectly optimized to run the code at the maximum possible speed but then requires the programmer weeks to learn how to use it and many days to develop the code it is not optimal. The complete review of the most used simulation tools written by Roman Brette and others[35] can be used as a reference. *Brian 2* (and *Brian* before it) has the big advantage of being written in Python. It can be directly integrated into a script and used together with other useful libraries such as NumPy, matplotlib and similar. This makes the work of the programmer much easier. In addition it tries to optimize whenever possible, for example by running functions at a lower level of abstraction, C or C++, if the system supports the compilation of the code in such languages.

11.2 Network data structure

Brian 2 makes a lot of data structure available for the simulation of a spiking neural network. In particular the ones used to simulate the developed model are:

1. *NeuronGroup*: object associated with an arbitrary numerous group of spiking neuron. This can be provided with the equations that describe the behaviour of the neurons.

2. *Synapse*: object which represents the connection between two *NeuronGroup* elements. It allows to specify the way in which the neurons belonging to the two groups are connected.

3. *PoissonGroup*: group of pseudo-neurons which convert the input firing rates in trains of spikes with a Poisson distribution.

4. *SpikeMonitor*: object which allows to monitor the spikes generated by a certain *NeuronGroup*. It allows to record the temporal evolution of the spikes, useful for the plot. However this feature is quite memory hungry and so is disabled through the flag *record = False*. This is used to keep trace of the counts of the spikes generated by the output layer.

5. *Network*: allows to enclose all the previously mentioned elements in a single object, that can be then passed to the functions for the update.

All these elements together allow to obtain the structure shown in figure 10.3, which corresponds to a Brian 2 *network* object, containing all the other objects. Sections from I to I shows the complete algorithm used to create and initialize the data structure following the user requirements.

11.3 Select training or test mode

The *Synapse* object, as the *NeuronGroup*, accepts in input the description of the temporal evolution of its state variables, in form of differential equations. This allows to specify, when the network is created, a learning rule to apply to the desired synapses. So in this phase it is possible to decide if the network will be run in training or test mode: if in fact no equation is specified the weights of the synapses will not be modified along the evolution of the network, that in this way is used in test mode.

11.4 Convert the image into spikes trains

The block that is in charge of the conversion of the input numerical data into trains for spikes is associated with a dedicated object in Brian 2. As a consequence the only detail to worry about in the conversion is to set the firing rates associated with each input to the pixels values. The two operations needed are reported in I.7.

11.5 Make the network evolve over the input spikes trains

Finally sections from I to I shows the steps followed to run the network, being it in training or test mode. It can be observed that in general a dedicated instruction is used whenever an access to the internal variables is required. This is due to the *encapsulation* provided by the *Network* class: the internal state variables are not directly readable or modifiable but they must be accessed through two internal functions, *get_states()* and *set_states()*.

As said before, the chosen data structure modifies also the output evaluation functions. In particular the *SpikeMonitor* object keep trace of the spikes generated along the whole training. The count relative to a specific image can be obtained subtracting the total amount of spikes generated up to the previous image, so this is an additional operation that in the handmade implementation is not required. The alternative would be to reset the monitor at every iteration.

It can be seen that no details on the internal computations regarding the temporal evolution of the network are provided. This is because at this level all the calculations are performed internally by Brian 2. It is possible to select the way in which equations will be solved choosing between *event-based* and *step-based*. Next chapters will explain the difference in detail. The *event-based* solution is selected to improve the performance. Also the mathematical method used to solve the differential equations can be set. In this case, being the model linear the *exact* method is selected. Both of the choices are different from the practical solutions used within the more detailed model first and the hardware accelerator later, but at this level the model is being simulated without care for the internal details, so all the choices are oriented towards the fastest possible simulation.

Chapter 12

Manual simulation of the network using python

Now that the model is well defined and simulated a new step towards a lower level of abstraction can be performed. In particular, instead of having all the internal equations solved by a simulator the detailed required computations are now made explicit. The methods used at this level will be then hardwired within the hardware accelerator, so some design choices must be taken at this point. Let's in particular analyze the difference between an *event-based* method and a *step-based* one.

12.1 Event-based solution

The event-based solution, as the name itself suggests, is based on the arrival of an input event for the computation of the evolution of its state variables. This means that the network is updated only if at least one input spike is received in input. In particular the exponential evolution of the membrane potential, the threshold and the synaptic traces is computed in an exact way only when a spike arrives at the inputs. The result is then increased or decreased due to the arrival of the spike itself.

This is an optimum solution in terms of efficiency, because it avoids useless computations when the only update consist in an exponential decrease. When the spike is received the arrival time is stored. It is then used in correspondence of the next spike to compute the elapsed time and so the exponential decrease. This is also the method used in Brian 2 and in fact the two settings regarding the solution of the state equations are:

1. *Event-based*: this is what has been just described.

2. *Exact*: the integral is exactly computed along the interval between two spikes.

Thinking to the hardware acceleration this is only optimum for the power consumption. In fact no computations are performed if no spikes are received, so the network is run only for the strictly required time. However all these advantages imply some costs:

1. The network requires to manage an event queue, ordering the spikes on the basis of their arrival time. This is quite a complex task to perform in hardware and requires additional resources.

2. Being the interval between two spikes unknown the exponential must be explicitly computed. This requires a dedicated circuit or an approximated solution based on LUTs, that in both cases require again additional resources.

There are a lot of hardware accelerators that use such a solution. One example is *Minitaur*[39].

12.2 Step-based solution

The goal of this thesis project is to design an hardware accelerator that is as small as possible, in order to maximize the number of neurons that can be integrated with equal hardware resources available, or to leave space for an higher degree of parallelization, with a consequent performance improvement. For this reason the chosen updating alogorithm is the step-based. This strongly simplifies both the control part, avoiding the use of an ordered queue, and the elaboration part, removing the necessity to precisely compute the exponential functions.

The step-based solution implies to solve the differential equations that characterize the model in an iterative way, updating all the state variables at every cycle. It allows to use the methods reported in sections A.2 and B.1, strongly simplifying the required computations.

The drawback is that the network is always active, also in absence of input spikes. A possible solution to reduce the power consumption that derives from this consists in updating the state variables until they are grater than a fixed threshold. After this quantity their variation can be considered sufficiently small to approximate them with their steady state value. The neuron in this case can enter in a wait state in which it is no more updated until a new spike is received[40].

There are also hybrid models which are able to select between *step-based* and *event-based* depending on the workload. In this particular case the model used to

develop such an accelerator[41] is very similar to the one presented in this thesis and applied to the same problem of digits recognition on the MNIST, so it could be interesting to take it as a reference.

12.3 Network data structure

Also in this case before analyzing the updating algorithm it is worthwhile to look at the data structures used to describe the network.

12.3.1 Input Poisson layer

First of all the Poisson layer is treated as a distinct element. The data structure associated with it is simply a NumPy two-dimensional Boolean array containing the temporal evolution of the spikes. In particular it has a number of rows equal to the duration of the spikes trains, expressed in number of temporal steps, and a number of columns that corresponds to the total amount of input data, so in this case to the dimension of the image expressed in pixels:

$$28x28 = 784 \tag{12.1}$$

12.3.2 Neural network

All the network components are described using dictionaries.

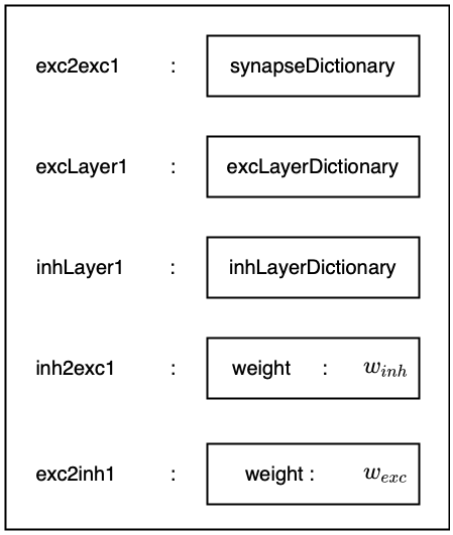
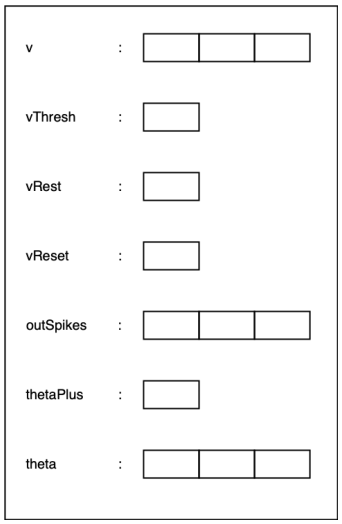


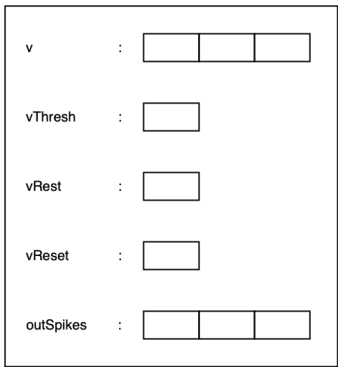
Figure 12.1: Network dictionary

The choice of this particular data structure depends on the possibility to explicitly label each element, which makes the code self-explanatory and more readable.

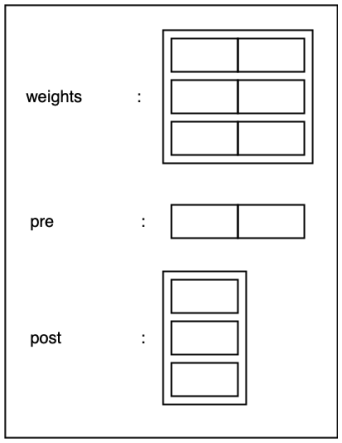
Figure 12.1 shows the complete dictionary used to describe the network, supposing as always to work with the structure shown in figure 7.1. Each element is associated with a dedicate dictionary, which completely describes it. Here for the sake of visualization only the dictionaries of the inter-layer synapses are made explicit. These correspond to the connections between the excitatory layer to the inhibitory one and vice versa. The only parameter that characterize them is the weight of the connection, equal for all the neurons.



(a) Excitatory layer dictionary



(b) Inhibitory layer dictionary



(c) Synapse dictionary

Figure 12.2: Detailed data structures

Figure 12.2 shows the details of the inner dictionaries describing the different components of the network. A possible improvement in the data structures could be to make all the elements coherent in terms of shape. This means in particular to modify *v*, *outSpikes* and *theta* to make their shape equal to the one of *post*. In this way each layer would be associated with a column vector, that would also be easier to visually associated with the network abstract structure. Sections from J to J shows the required steps to initialize the described structure.

12.4 Convert the image into spikes trains

Differently from what seen with the Brian 2 simulation, the conversion must be performed manually. Figures J.5 and J.6 show the details of the computations required. Again *vectorization* is used to reduce to a minimum the interpreting overhead.

First of all a NumPy two-dimensional array of values uniformly distributed between zero and one is created. Its dimension are the same of the data structure that will be used to store the temporal evolution of the spikes associated to each pixel, as described in section 12.3.1. Such array is used following the algorithm explained in section 8.2.

12.5 Make the network evolve over the input spikes trains

The complete algorithm used to perform the inner computations necessary to update the network following the developed model is reported in sections from J to J. All the updates are performed in a step-based way, using the equations explained in sections A.2 and B.1. Also in this case *vectorization* is used and all the neurons belonging to a layer are updated with a single instruction. This implies the use of boolean addressing, shown in figure 10.2 to select only the desired neurons. Since the spikes are intrinsically binary they are stored in form of boolean arrays and so they can be directly used to perform such an addressing.

12.6 Choice of the temporal step

Once defined all the parameters of the network, including the time constants of the exponential functions, the temporal step must be set. This was not required in Brian 2 because Δt was included within the network object, even if it was modifiable from outside. In order to make the manual simulation coherent with what has been obtained with Brian 2 the chosen value for the time step is the same used by default by the simulator, that is:

$$\Delta t = 100\mu s \tag{12.2}$$

This value affects the update of all the states variables within the network since the computations are step-based.

Chapter 13

Simplification of the network for the hardware acceleration

At this point the developed model is completely described and simulated in detail and so it is ready to be translated into a dedicated hardware component. However there are some simplification that can be applied in order to make the integration onto an hardware platform more efficient without remarkably modifying the model itself.

13.1 Remove the inhibitory neurons

The inhibitory neurons are a simplified version of the excitatory one, with a constant threshold that does not depend on the output spiking activity. Each of them presents a single input connection, originated from the correspondent excitatory neuron. The weight of such connection is set to a value that is sufficient to make the membrane potential exceed the threshold when a single spike is received, as explained in section D.5. This means that, whenever an excitatory neuron generates a spike the corresponding inhibitory node responds generating in turn a new spike that is then brought to all the other excitatory neurons in the layer. As a consequence the output of each inhibitory element is the exact copy of its excitatory counterpart. Therefore it is useless to dedicate a physical component, with its own temporal evolution, to the generation of the inhibitory spikes, it is sufficient to directly connect each excitatory output to all the others nodes in the layer.

The update of the membrane potential at this point is computed considering the input spikes and the ones generated by the layer itself in the previous update cycle. In this way, speaking about the python simulation, the inhibitory layer is reduced to a NumPy array which simply stores the spikes generated by the excitatory neurons in order to use them in the next iteration. This almost halves the required resources.

13.2 Homeostasis

The problem of the homeostasis is that it requires a very high time constant. The exponential decay of the threshold is in fact much slower if compared to the membrane potential. See sections 7.3 and D.1 for more details.
Looking at how the differential equations are iteratively solved it can be seen that the result depends from the quantity:

$$\frac{\Delta t}{\tau_\theta} \tag{13.1}$$

Section 12.6 reports the chosen value for the time step and substituting the value of τ_θ the result is:

$$\frac{\Delta t}{\tau_\theta} = \frac{100\mu s}{10^7 ms} = \frac{10^{-1} ms}{10^7 ms} = 10^{-8} \tag{13.2}$$

In order to evaluate what does this means in terms of the target hardware architecture let's convert the result using 2 as a base:

$$10^{-8} = 2^{log_2(10^{-8})} = 2^{-8 \cdot log_2(10)} \simeq 2^{-27} \tag{13.3}$$

Working with a fixed point architecture with a single integer bit this means that the parallelism must be at least 28 bits, but even higher to obtain acceptable results in terms of representation of an exponential decay.

Let's now analyze the impact of the exponential decrease on the threshold. With an exponential function of the type

$$e^{-\frac{t-t_0}{\tau}} \tag{13.4}$$

τ represents the time difference $t - t_0$ for which the magnitude is reduced by a quantity:

$$e^{-\frac{\tau}{\tau}} = e^{-1} \simeq 0.37 \tag{13.5}$$

so in other words for which the magnitude is reduced down to the 63% of its value. In the analyzed case the time required for such a reduction is $\tau_\theta = 10^7 ms$.

The default duration of an elaboration cycle over a single image is 350ms, while the rest period lasts 150ms. So the amount of images required for the homeostasis parameter to reach the 63% of its magnitude is:

$$\frac{10^7 ms}{350ms + 150ms} = 20000 \tag{13.6}$$

Being the exponential decreasing so slow a possible alternative is to remove it. In this case the homeostasis only consists in an increment of the threshold when an output spike is generated.

In order to avoid to analyze each image two or more times due to the growing threshold the θ_+ parameter is reduced a bit with respect to the value reported in section D.1. So in this case:

θ_+	0.05mV

Table 13.1: Modified value of the homeostasis parameter

An interesting consequence of such modification is that the accuracy of the network starts from lower values and then tends to grow more regularly, with reduced oscillations. More studies should be conducted to investigate the phenomena, at the moment it is only reported as a curiosity.
The result is that the parallelism of the architecture can be much smaller and no iterative updates are required for the threshold, which requires an update only if an output spike is generated.

13.3 Bring the network into a rest state

The method used to bring the network to rest before analyzing a new image consists in running it for a fixed period, by default 150ms, with its inputs at zero, so without any input spike. Working with a step-based architecture and with a time step of 0.1ms means that the network is forced to perform

$$\frac{150ms}{0.1ms} = 1500 \tag{13.7}$$

cycles only to make the membrane potential tend towards its rest value. So in order to avoid such a waste of time and accepting a small approximation the membrane potential is simply forced to its rest value at the end of each image.

13.4 Results

Since the model has been slightly modified, even if the reasoning presented up to now shows that this has a negligible effect the network is simulated again and the results are presented in a separate section, in order to be compared with the original ones.

Chapter 14

Design of the hardware accelerator

Now that the model has been exhaustively simulated it is ready to be translated into an hardware accelerator. As repeatedly said in the previous chapters, the goal is to minimize the area occupied by the circuit in order to leave space to a large amount of neurons or to an higher level of parallelism. For this reasons the neuron is realized with the minimum possible resources. This sets a lower boundary to the performance.

The two main methods with which the circuit can be trained are:

1. *Offline training*: the model does not dynamically learn while it is running. There is a dedicated phase in which the training happens, being it on a dedicated hardware accelerator or using a software version of the model. The obtained parameters are then loaded into the accelerator and used to evaluate the input data. If a new training is required the accelerator is stopped and then restarted with the new parameters.

2. *Online training*: in this case the learning happens while the accelerator itself is working. This means that the circuit can dynamically learn from new input data and implies the presence of a training system on board.

The design of the circuit is performed gradually. First of all a structure able to run a pre-trained network is developed. In this case the training is performed offline using the software simulation and the weights are then loaded into the accelerator.

14.1 Parallelization degree and performance improvement

The goal of the hardware accelerator, as the name itself suggests, is to reduced the time required to evaluate the model by performing the required computations with a certain degree of parallelization. The ways in which this can be achieved are mainly two:

1. All the neurons belonging to a specific layer are independent one from the other, so they can be updated in parallel.
2. All the input spikes increase or decrease the membrane potential and all of them must be considered before evaluating if the potential has exceeded the threshold. This means that there is not an order that the weights belonging to the active synapses must follow to be added to the current value of the membrane potential. As a consequence in the extreme case all the weights can be added together in parallel, requiring a single updating cycle.

Being the usable hardware resources limited, a solution in which all the neurons work in parallel, elaborating all their inputs in parallel is probably not feasible. The first solution consists in updating all the neurons in parallel, computing the weights sum in sequence. As a consequence the time required for a complete update of the network, if compared with the software solution in which all the computations are performed in parallel is:

$$T_{acc} = \frac{T_{sw}}{N_{neurons}} \tag{14.1}$$

This is obviously an approximate result which doesn't consider the interpreting overhead present in the python version, the possible degree of parallelization made available by the processor and other possible details, but it works well as a first approximation.

Another interesting feature is that, one determined the time required by a single neuron to be updated, it remains constant with the number of neurons within a layer, differently from a sequential solution.

Let's now analyze the design details that characterize the developed accelera

14.2 Circuit to test the model: offline training

The key points in this first phase are the model of the membrane potential and the way in which it is translated into a dedicated circuit. The synapses are represented

through static weights that modifies the potential when a spike arrives on the relative connection.

The network is composed by three hierarchical levels:

1. Neuron: core of the accelerator. This is where the developed model is implemented.
2. Layer: group of neurons working in parallel.
3. Network: group of layers working as a pipeline.

14.2.1 Neuron

Figure L.2 shows the datapath used to perform the required computations. The features to notice are:

1. The presence of a single adder. This represents the minimum resources solution.
2. The absence of a multiplier, substituted with a shift operation.

The single adder choice implies that a basic operation as the exponential decay of the membrane potential is split into two steps, one to subtract the quantity

$$\frac{\Delta t}{\tau} \cdot V[n] \tag{14.2}$$

from V[n] and the other to shift the result up by adding

$$\frac{\Delta t}{\tau} \cdot V_{rest} \tag{14.3}$$

As a consequence, since the neurons are updated in parallel and so they must be synchronized one with the other, a *pause* step, in which the network does not perform any computation, is required after the generation of an output. This is because the reset of the membrane and the homeostatic increase of the threshold can be computed in parallel and require a single step. The details about the control part are reported in figure L.3.

The only multiplication operation is the one required by equation 14.2. In this case τ is a parameter of the model and cannot be modified, but Δt can be chosen in a quite free manner. With a proper choice of its value the quantity

$$\frac{\Delta t}{\tau} \tag{14.4}$$

becomes a negative power of two and can be computed with a simple bit-shift instead of requiring an explicit multiplication. With $\Delta t = 100\mu s$ the result is:

$$\frac{\Delta t}{\tau} = \frac{10^{-4}s}{10^{-1}s} = 10^{-3} \tag{14.5}$$

The nearest power of two is $2^{-}10 = 0.977 \cdot 10^{-3}$, which corresponds to a 10 bits right shift. This also sets a minimum boundary to the parallelism of the architecture.

Finally each neuron has its own control unit. This is required because each neuron must be checked separately to evaluate if its membrane potential has exceeded the threshold or not, and since all of them work in parallel a single control unit cannot be used. Each neuron is driven through two start signals:

1. *EXP_EXC_START*
2. *REST_INH_START*

These are used in two different phases along the membrane potential update. In particular:

1. To start the network. In this case there are two distinct operations that the neuron can perform:

   - Update its membrane, resetting it if it exceeds the threshold and exponentially decreasing it otherwise.
   - Reset the membrane to its rest value at the end of an image before starting with the next one.

   The two start signals can be used to select the desired operation, following the truth table shown below.

EXP_EXC_START	REST_INH_START	Operation
1	-	Update the membrane potential
0	1	Reset the membrane potential to its rest value
0	0	Return in idle

Table 14.1: Truth table of the two starts in the first phase

2. To update the membrane with the input spikes. In this case the neuron gives the possibility to:

- Update the membrane with the weights of the excitatory synapses and then with the inhibitory ones.
- Update the membrane only with the weights of the excitatory synapses.
- Update the membrane only with the weights of the inhibitory synapses.
- Avoid the update of the membrane and return in idle.

In this way if there are no input or inhibitory spikes no useless update cycles are performed. The truth table is reported also in this case:

EXP_EXC_START	REST_INH_START	Operation
1	1	Both excitatory and inhibitory
1	0	Only excitatory
0	1	Only inhibitory
0	0	Return in idle

Table 14.2: Truth table of the two starts in the second phase

14.2.2 Layer of neurons

Now that the neuron architecture is defined it can be used to create one layer of the network. Sections from L.4 to L.8 show all the components of an example layer of three neurons.

What is important to notice is that the layer is not a simple group of neurons put in parallel, but it has additional elements and its own control unit. The reason is that a system to manage the neurons update is required, since the parallelism is not complete and the inputs are elaborated sequentially.

Figure L.5 shows the circuit used to select the inputs. It computes the logical OR of all the input spikes. If the result is zero, that means that there is no input spikes the proper start code is provided to the neurons to avoid the update on those specific inputs, being them excitatory or inhibitory. If instead at least one spike is present the inputs are provided to the neurons one at a time through the use of the multiplexer. Here the selection is managed through a counter, that is updated until it reaches the total number of inputs.

The same counters used to select the excitatory or inhibitory spikes are used also by the control unit to run the update of the neurons until all the elements have been considered.

The excitatory counter has also a third use, that is the selection of the correct weight related to the currently considered input. This is explained more in detail in subsection 14.2.3 For the inhibitory part this is not necessary because all the connections have the same weight.

14.2.3 Synapses

Figure L.9 shows the architecture of the synapses block. It is a simple register file with one input port and one output port. All the synapses corresponding to a layer are put together and treated as a single component, as shown in figure L.10.

First of all a loading phase is necessary, since the accelerator uses the weights obtained through an offline training. All the synapses belonging to a layer can be updated in parallel, writing a single weight for each synapse. So a number of cycles equal to the number of inputs is required for each layer.

Once the weights are loaded the desired one can be selected by simply providing the address of the corresponding input. In this way the weights can be accessed sequentially using the input counter.

14.2.4 Network

Finally multiple layers can be put together to form a network. An example structure with two layers is shown in figure L.11. Each layer has its own group of synapses from which it reads the weights. The layers are connected sequentially one to the other. All of them are connected in parallel to the signals:

1. RST_N: asynchronous reset that affects all the layers at the same time.
2. REST: synchronous signal to reset the network to its rest state before analyzing the next image.

Looking at image L.7 and at how the input and output signals are managed by the control unit in figure L.8 it can be seen that the input and output protocol are the same for the signals:

1. START and NEXT_LAYER_START
2. PREV_LAYER_END and END

When receiving an active START the layer updates the membrane potentials, generating the spikes where necessary, and then gives the start to the next layer.

In this way the spikes are quickly propagated along the network and most of the computations, corresponding to the elaboration of the input spikes, are performed in parallel.

At the end of the elaboration each layer enters into a wait state and remains in it until the previous layer has finished in turn. This guarantees a synchronization between layers that can have different dimensions and different elaboration flows, that depend on the presence of spikes on their inputs. The first layer has a constant '1' in input since there is not a previous layer able to provide the ending signal. The last layer has both its NEXT_LAYER_START and END signals connected to the output of the network. They can be used to interface the network as follows:

1. Start the network, giving it an active START and providing the required spikes sequence as an input.
2. Each time the NEXT_LAYER_START becomes active, meaning that the network has completed the elaboration over a single cycle of input spikes, increment the counter of the active neurons.
3. When the network has finished the elaboration, so when the END signal becomes active provide the REST signal to reset the network to its rest state.

Chapter 15

Future work

15.1 Test of the developed circuit

Now that the design is ready it can be implemented using VHDL language and tested in order to compare its performance to the one obtained with the python simulation. This part is not present in the thesis work but will be published in a dedicated paper.

15.2 Online training

As said before the architecture developed up to now is able to run a pre-trained model. It would be interesting to consider also the second alternative, the online training, in order to evaluate the performance improvement also in this case. To do this a further study on possible alternatives to weight normalization should be considered, in order to keep the area as limited as possible. One promising possibility seems to be the insertion of random delays in the input and inter-layer excitatory connections.

15.3 FPGA implementation

Once the accelerator is tested and ready it can be synthesized into a netlist which targets an FPGA platform. In this way the designed circuit can be truly tested, going beyond the simple simulation. A possible target for the accelerator could be an evaluation board which includes an FPGA, a microprocessor and a rough version of Linux. In this case a small driver could be used to interface the accelerator. Instead of performing the network inner computations using python or leaving it to some dedicated library functions, as done using Brian 2, the accelerator can

be used in this case. In such a scenario there are many possibilities that can be tested. In particular there are three possible levels in which the various parts of the interface can be implemented:

1. From the input point of view the conversion of the image into trains of spikes can be performed within the python, which then sends the spikes towards the board; by the Linux driver, which then sends it to the hardware accelerator; with a dedicated hardware. In this case the image is simply sent to the FPGA and then converted step by step for the desired number of cycles. This implies to generate random values with an hardware component so it would probably involve the use of LFSR or similar components, that must be hosted within the FPGA.

2. From the output point of view the count of the output spikes could be again performed through dedicated counters within the FPGA, by the processor or by the python program.

3. The evaluation part, including the computation of the accuracy and the classification update, again could be performed by the on board processor or using the python program.

In general there are many possibilities that can be tested and it would be interesting to evaluate the optimal solution, considering also the transmission overhead due to the necessity to send data from the computer to the board, from the board memory to the hardware accelerator and back in the opposite direction. In this way the performance of the the various solutions, the python manual simulation, the Brian 2 version and the hardware accelerator can be truly compared, taking in account all the possible delays introduced along the computation chain.

15.4 Python framework

Finally, once the accelerator is tested, together with all the possibilities mentioned above, the last step could be to realize a python framework able to create the accelerator by directly writing the VHDL code, following the user's requests and to interface it in the best possible way. This would allow the possibility to test different architectures, with different numbers of layers of different size without the necessity of redesigning the accelerator every time.

Appendix A

Current based synapse: mathematical analysis

This appendix reports the detailed mathematical analysis of the current-based synapse and of its impact on the membrane potential. All of the calculations have been developed independently from any verified text or resource, so its correctness is not completely guaranteed. Take it only as a reference.

Figure 5.1 shows the equivalent circuit of the model, used as a reference for the following computations.

A.1 Continuous time

Let's start by computing the total current that flows through the membrane capacitance:

$$I_{tot} = I_{in} + I_{rest} \tag{A.1}$$

$$I_{rest} = \frac{V_{rest} - V(t)}{R_M} \tag{A.2}$$

$$I_{tot} = C_M \cdot \frac{dV(t)}{dt} \tag{A.3}$$

$$I_{in} = \sum_{i=1}^{N} I_i \tag{A.4}$$

$$C_M \cdot \frac{dV(t)}{dt} = \frac{V_{rest} - V(t)}{R_M} + \sum_{i=1}^{N} I_i \tag{A.5}$$

This implies that the equation which describes the temporal evolution of the membrane potential is given by the following first order differential equation:

$$\frac{dV(t)}{dt} + \frac{1}{\tau} \cdot V(t) = \frac{1}{\tau} \cdot V_{rest} + \frac{1}{C_m} \cdot \sum_{i=1}^{N} I_i \tag{A.6}$$

The general solution of a first order differential equation in the form

$$\frac{dx(t)}{dt} + a_0(t) \cdot x(t) = g(t) \tag{A.7}$$

can be obtained as follows:

$$x(t) = e^{-A(t)} \cdot \left[c_1 + \int g(t) \cdot e^{A(t)} dt \right] \tag{A.8}$$

or alternatively:

$$x(t) = e^{-A(t)} \cdot \int_{t_1}^{t} g(t) \cdot e^{A(t)} dt \tag{A.9}$$

where

$$A(t) = \int_{t_0}^{t} a_0(t) dt \tag{A.10}$$

$$a_0(t) = \frac{1}{\tau} \tag{A.11}$$

$$A(t) = \int_{t_0}^{t} \frac{1}{\tau} dt = \frac{t - t_0}{\tau} \tag{A.12}$$

$$g(t) = \frac{1}{\tau} \cdot V_{rest} + \frac{1}{C_m} \cdot \sum_{i=1}^{N} I_i \tag{A.13}$$

As a consequence the general solution for the membrane potential is:

$$V(t) = e^{-A(t)} \cdot \int_{t_1}^{t} \left(\frac{1}{\tau} \cdot V_{rest} + \frac{1}{C_m} \cdot \sum_{i=1}^{N} I_i \right) \cdot e^{A(t)} dt \tag{A.14}$$

Thanks to the linearity of the integral the expression can be rewritten as:

$$V(t) = e^{-A(t)} \cdot \left(\int_{t_1}^{t} \frac{1}{\tau} \cdot V_{rest} \cdot \cdot e^{A(t)} dt + \int_{t_1}^{t} \frac{1}{C_m} \cdot \sum_{i=1}^{N} I_i \cdot e^{A(t)} dt \right) = \tag{A.15}$$

$$= e^{-A(t)} \cdot [V_1(t) + V_2(t)] \tag{A.16}$$

Let's now analyze the two components of $V(t)$ separately.

A.1.1 Computation of the first part of the expression

$$V_1(t) = \int_{t_1}^{t} \frac{1}{\tau} \cdot V_{rest} \cdot e^{A(t)} dt = \tag{A.17}$$

$$= \frac{V_{rest}}{\tau} \cdot e^{-\frac{t_0}{\tau}} \int_{t_1}^{t} e^{\frac{t}{\tau}} dt \tag{A.18}$$

$$V_1(t) = V_{rest} \cdot e^{\frac{t0}{\tau}} \cdot \left(e^{\frac{t}{\tau}} - e^{\frac{t_1}{\tau}} \right) = V_{rest} \cdot \left(e^{\frac{t-t_0}{\tau}} - e^{\frac{t_1-t_0}{\tau}} \right) \tag{A.19}$$

A.1.2 Computation of the second part of the expression

$$V_2(t) = \int_{t_1}^{t} \frac{1}{C_m} \cdot \sum_{i=1}^{N} I_i \cdot e^{A(t)} dt \tag{A.20}$$

Again thanks to the linearity of the integral:

$$V_2(t) = \frac{1}{C_m} \cdot \sum_{i=1}^{N} \int_{t_1}^{t} I_i \cdot e^{A(t)} dt \tag{A.21}$$

At this point the expression of the generic current I_i can be expanded. Here the current spike is treated as an ideal Dirac delta with area equal to the weight w_i. The index i simply represent a generic input synapse. In the example reported in figure 5.1 there are three input synapses, so three different currents I_1, I_2 and I_3, each with its own weight w_1, w_2 and w_3.

$$V_2(t) = \frac{1}{C_m} \cdot \sum_{i=1}^{N} \int_{t_1}^{t} w_i \cdot \delta(t_i - t) \cdot e^{A(t)} dt \tag{A.22}$$

It is known that the Dirac delta can be used to sample a generic function $f(t)$ through the convolution operation, as shown in equation A.23. The convolution

returns the value of the function in the sampling point if the delta belongs to the integration interval and zero otherwise.

$$\int_{t_1}^{t} \delta(t_0 - t) \cdot f(t)dt = \begin{cases} f(t_0) & if\ t_1 < t_0 < t \\ 0 & if\ t_0 < t_1\ and\ t_0 > t \end{cases} \tag{A.23}$$

So applying equation A.23 to equation A.22 leads to:

$$V_2(t) = \frac{1}{C_m} \cdot \sum_{i=1}^{N} s_i \cdot w_i \cdot e^{\frac{t_i - t_0}{\tau}} \tag{A.24}$$

where

$$\begin{cases} s_i = 1 & if\ t_1 < t_0 < t \\ s_i = 0 & if\ t_0 < t_1\ and\ t_0 > t \end{cases} \tag{A.25}$$

A.1.3 Computation of the complete membrane potential expression

Finally, substituting the two computed parts into equation A.16 the complete expression of the temporal evolution of the membrane potential can be computed.

$$V(t) = e^{-\frac{t-t_0}{\tau}} \cdot \left(V_{rest} \cdot e^{\frac{t-t_0}{\tau}} - V_{rest} \cdot e^{\frac{t_1-t_0}{\tau}} + \frac{1}{C_m} \cdot \sum_{i=1}^{N} s_i \cdot w_i \cdot e^{\frac{t_i-t_0}{\tau}} \right) \tag{A.26}$$

$$V(t) = V_{rest} + \left(\frac{1}{C_m} \cdot \sum_{i=1}^{N} s_i \cdot w_i \cdot e^{\frac{t_i-t_0}{\tau}} - V_{rest} \cdot e^{\frac{t_1-t_0}{\tau}} \right) \cdot e^{-\frac{t-t_0}{\tau}} \tag{A.27}$$

For $t_i \longrightarrow t_0$, so supposing that all the spikes arrive in the same instant t_0, and for $t_i = t_0$, so integrating from t_0 over, the expression becomes:

$$V(t) = V_{rest} + \left(\frac{1}{C_m} \cdot \sum_{i=1}^{N} s_i \cdot w_i - V_{rest} \right) \cdot e^{-\frac{t-t_0}{\tau}} \tag{A.28}$$

Finally, in order to simplify the expression C_M can be considered equal to 1. As a consequence all the other values, like R_M or the generic weight w_i, are set to values that allow to keep the model unchanged. Another possible way to remove the C_M contribute is to include it within the weights. In this way both R_M and

C_M are kept to biologically plausible values and the weights are modified to keep their contribute unchanged, as reported in equation A.29

$$w_i^{'} = \frac{w_i}{C_M} \tag{A.29}$$

In both cases the membrane potential expression becomes:

$$V(t) = V_{rest} + \left(\sum_{i=1}^{N} s_i \cdot w_i - V_{rest} \right) \cdot e^{-\frac{t-t_0}{\tau}} \tag{A.30}$$

A.2 Discrete time

Equation A.30 provides an expression that allows to explicitly compute the temporal evolution of the membrane potential. Working with a python simulation or with an hardware accelerator all the computations are performed in a *discrete-time* domain. This means that the *time* does not vary in a continuous way, but can only assume fixed values, equally spaced one from the other. The possible solutions to compute the membrane potential evolution in this case are two:

1. Explicit computation of V(t) using equation A.30. In this case t_0 becomes the last instant in which one spike was received and t will assume discrete values.

2. Iterative computation starting from equation A.6. All the following equations explain in detail this last solution.

Working with a sufficiently short time interval Δt the voltage derivative can be well approximated by its *difference quotient*, as shown in equation A.31.

$$\frac{V(t+\Delta t) - V(t)}{\Delta t} + \frac{1}{\tau} \cdot V(t) = \frac{1}{\tau} \cdot V_{rest} + \frac{1}{C_m} \cdot \sum_{i=1}^{N} I_i \tag{A.31}$$

The equation allows to find the value of $V(t+\Delta t)$, that is the membrane potential variation with respect to its value V(t) after a short time increment Δt.

$$V(t+\Delta t) = V(t) - \frac{\Delta t}{\tau} \cdot V(t) + \frac{\Delta t}{\tau} \cdot V_{rest} + \frac{\Delta t}{C_m} \cdot \sum_{i=1}^{N} I_i \tag{A.32}$$

Again the properties of the Dirac delta can be used to proceed in the computations.

In particular the ideal delta can be defined as an impulse with infinite amplitude and zero base and with unitary area. As a consequence:

$$\lim_{\Delta t \to 0} \Delta t \cdot \delta(t - t_0) = \begin{cases} 1 & if\ t_0 \in \Delta t \\ 0 & if\ t_0 \notin \Delta t \end{cases} \tag{A.33}$$

So, for a sufficiently small interval Δt the quantity $\Delta t \cdot \delta(t - t_0)$ represents the area of the delta, supposing that the delta belongs to the interval itself. Otherwise the result of the expression becomes zero. Multiplying the delta for a constant value w_i means to impose a value w_i to its area, and so:

$$V(t + \Delta t) = V(t) \cdot \left[1 - \frac{\Delta t}{\tau}\right] + \frac{\Delta t}{\tau} \cdot V_{rest} + \frac{1}{C_m} \cdot \sum_{i=1}^{N} s_i \cdot w_i \tag{A.34}$$

where again s_i is used to identify the presence or absence of the delta within the interval. The same suppositions made at the end of section A.1 can be now applied to the capacitance C_M.

$$V(t + \Delta t) = V(t) \cdot \left[1 - \frac{\Delta t}{\tau}\right] + \frac{\Delta t}{\tau} \cdot V_{rest} + \sum_{i=1}^{N} s_i \cdot w_i \tag{A.35}$$

Finally, working with elaboration steps instead of explicit time values, equation A.35 can be rewritten as:

$$V[n + 1] = V[n] \cdot \left[1 - \frac{\Delta t}{\tau}\right] + \frac{\Delta t}{\tau} \cdot V_{rest} + \sum_{i=1}^{N} s_i \cdot w_i \tag{A.36}$$

Appendix B

Synaptic traces. Mathematical analysis

Section 6.3 presented the synaptic traces as a possible practical implementation of the STDP learning rule. This section aims to demonstrate that the two methods are coincident.

Let's start by integrating the temporal evolution of the synaptic traces:

$$\int_{t_0}^{t} \frac{da_{pre}}{a_{pre}} = -\int_{t_0}^{t} \frac{dt}{\tau_{pre}} \tag{B.1}$$

$$log[a_{pre}(t)] - log[a_{pre}(t_0)] = -\frac{t - t_0}{\tau_{pre}} \tag{B.2}$$

$$log\left[\frac{a_{pre}(t)}{a_{pre}(t_0)}\right] = -\frac{t - t_0}{\tau_{pre}} \tag{B.3}$$

$$a_{pre}(t) = a_{pre}(t_0) \cdot e^{-\frac{t-t_0}{\tau_{pre}}} \tag{B.4}$$

Supposing $t_0 = t_{in}$ to be the instant in which an input spike is received, and knowing from the updating rule reported in equation 6.4 that $a_{pre}(t_in) = A_{LTP}$:

$$a_{pre}(t) = A_{LTP} \cdot e^{-\frac{t-t_{in}}{\tau_{pre}}} \tag{B.5}$$

So when an output spike is generated the equation is evaluated in t_{out}. Imposing $t_{out} - t_{in} = \Delta t$:

$$a_{pre}(t) = A_{LTP} \cdot e^{-\frac{\Delta t}{\tau_{pre}}} \tag{B.6}$$

In the same way $a_{post}(t)$ can be computed. The final result leads to:

$$a_{post}(t) = A_{LTD} \cdot e^{\frac{\Delta t}{\tau_{post}}} \tag{B.7}$$

B.1 Iterative computation of the synaptic traces

Working in a discrete-time domain, with a sufficiently short time step, the characteristic equations of a_{pre} and a_{post} can be rewritten as:

$$\frac{a_{pre}(t + \Delta t) - a_{pre}(t)}{\Delta t} = -\frac{a_{pre}(t)}{\tau_{pre}} \tag{B.8}$$

$$\frac{a_{post}(t + \Delta t) - a_{post}(t)}{\Delta t} = -\frac{a_{post}(t)}{\tau_{post}} \tag{B.9}$$

So to find the value of a_{pre} and a_{post} in the iteration $t + \Delta t$ it is sufficient to compute:

$$a_{pre}(t + \Delta t) = a_{pre}(t) - \frac{\Delta t}{\tau_{pre}} \cdot a_{pre}(t) \tag{B.10}$$

$$a_{post}(t + \Delta t) = a_{post}(t) - \frac{\Delta t}{\tau_{post}} \cdot a_{post}(t) \tag{B.11}$$

Appendix C

Unsupervised learning of digit recognition using stdp

The model used to describe the membrane potential is the *conductance based leaky integrate and fire* model. The *"conductance based"* attribute identifies the variable conductance that characterizes the synapses. This is an alternative for the *current based* model, in which the input current spike, multiplied by the weight of the synapse, directly increases the membrane potential. In this case instead the input event modifies the conductance of the synapse which, connected to a constant potential, let a larger current pass through it.

C.1 Electrical equivalent and model of the membrane potential

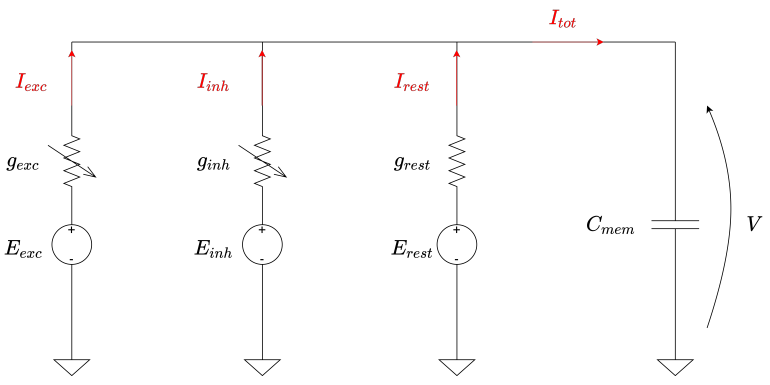


Figure C.1: Equivalent circuit of the conductance based model

Figure C.1 shows the electrical equivalent of the system. In static conditions the conductance of the excitatory and inhibitory connections is zero. In this situation the capacitor is charged or discharged, depending on the voltage between the two terminals of C_{mem}, through g_{rest} and tends to a steady state voltage E_{rest}. Whenever an input spike arrives through the excitatory or inhibitory connections it increases the relative conductance and connects the capacitor to the corresponding voltage source. E_{exc} is positive and tends to increase the capacitor's voltage, while E_{inh} is negative and tends to decrease it. Equations C.1 to C.11 show the mathematical details of the membrane potential model.

$$I_{exc} = g_{exc} \cdot (E_{exc} - V) \tag{C.1}$$

$$I_{inh} = g_{inh} \cdot (E_{inh} - V) \tag{C.2}$$

$$I_{rest} = g_{rest} \cdot (E_{rest} - V) \tag{C.3}$$

$$I_{tot} = I_{exc} + I_{inh} + I_{rest} \tag{C.4}$$

$$I_{tot} = C_{mem} \cdot \frac{dV}{dt} \tag{C.5}$$

$$C_{mem} \cdot \frac{dV}{dt} = g_{exc} \cdot (E_{exc} - V) + g_{inh} \cdot (E_{inh} - V) + g_{rest} \cdot (E_{rest} - V) \tag{C.6}$$

Normalizing by g_{rest}:

$$\frac{C_{mem}}{g_{rest}} \cdot \frac{dV}{dt} = \frac{g_{exc}}{g_{rest}} \cdot (E_{exc} - V) + \frac{g_{inh}}{g_{rest}} \cdot (E_{inh} - V) + (E_{rest} - V) \tag{C.7}$$

At this point to simplify the notation the following quantities are defined:

$$\tau_V = \frac{C_{mem}}{g_{rest}} = C_{mem} \cdot R_{rest} \tag{C.8}$$

$$g_{exc}^{nd} = \frac{g_{exc}}{g_{rest}} \tag{C.9}$$

$$g_{inh}^{nd} = \frac{g_{inh}}{g_{rest}} \tag{C.10}$$

Finally, imposing $g_{rest} = 1nS$ the values of g_{exc}^{nd} and g_{inh}^{nd} are equal to g_{exc} and g_{inh}, but without unit of measurement. To furtherly simplify the notation the superscript *nd* (which stays for *no dimensions*) is removed.

$$\tau_V \cdot \frac{dV}{dt} = g_{exc} \cdot (E_{exc} - V) + g_{inh} \cdot (E_{inh} - V) + (E_{rest} - V) \tag{C.11}$$

C.2 Temporal evolution of the synapses' conductance

As said before the conductance of the excitatory and inhibitory connections is increased when an input spike arrives and then decreases exponentially if no other pulses are received.

$$\frac{dg_{exc}}{dt} = -\frac{g_{exc}}{\tau_{g_e}} \tag{C.12}$$

$$\frac{dg_{inh}}{dt} = -\frac{g_{inh}}{\tau_{g_i}} \tag{C.13}$$

This makes the dependence of the membrane potential from the time non-linear. For this reason when solving the equations within *Brian 2* the attribute *euler* is used instead of *exact*. See the *Brian 2* documentation for more details.

C.3 Dependence of the threshold voltage from the spiking activity (homeostasis)

In order to keep the activity of each neuron constant and to avoid a frequently active node to reduce the contribute of the others a threshold control mechanism is used. This is a negative feedback mechanism in which the threshold voltage is increased every time a spike arrives. In this way the more frequent the output firing rate the higher the threshold becomes, reducing in turns the output firing rate. Again in absence of further output spikes the threshold is exponentially decreased with respect to time towards a steady state value, until a new spike is generated. Equation C.14 reports the temporal evolution of the θ parameter, that is used to dynamically modify the threshold voltage, while equation C.15 shows the dependence of the threshold voltage from θ

$$\frac{d\theta}{dt} = -\frac{\theta}{\tau_\theta} \tag{C.14}$$

$$V_{thresh} = V_{thresh\_0} + \theta \tag{C.15}$$

Appendix D

Model parameters

The model parameters are very similar to the ones used by Peter U. Diehl and Matthew Cook[27]. The chosen values try to be coherent with the ones measured in a biological brain, with few exceptions, given by the different size of a real brain and the developed model.

D.1 Excitatory neurons

The membrane potential model of the excitatory neurons represents the main example in which the parameters have been modified with respect to the biological ones in order to improve the network accuracy. In this case the exponential time constant has been increased from the typical 10ms or 20ms to 100ms. This allows to better estimate the input spiking rate, which is lower if compared to the one measured within a biological brain, but only because the amount of neurons involved in the computation is in turns much lower. See Peter Diehl and Matthew Cook's article for more details[27].

Reset potential	-60.0mV
Rest potential	-65.0mV
Threshold voltage	-52.0mV
θ_+	0.1mV
θ starting value	20mV
τ_V	100ms
τ_θ	$10^7 ms$

Table D.1: Excitatory neurons parameters

D.2 Inhibitory neurons

Reset potential	-45.0mV
Rest potential	-60.0mV
Threshold voltage	-40.0mV
τ_V	10ms

Table D.2: Inhibitory neurons parameters

D.3 Connection between excitatory layers

The parameters presented in this section are valid both for the connection of the input layer to the first excitatory one and for the connection between consecutive excitatory layers. These are the only two synapse types on which the learning is applied. It can be noticed that the STDP is asymmetric, with a higher value for the *long term potentiation*.

A_{LTP}	$10^{-3}mV$
A_{LTD}	$10^{-4}mV$
τ_{LTP}	20ms
τ_{ltd}	20ms

Table D.3: STDP parameters

D.4 Connection from excitatory to inhibitory layer

All the connections between the excitatory and inhibitory layers are one-to-one connections. Each excitatory neuron has an inhibitory one associated to it. The weight of this single connection is the same for all the neurons and is chosen to be sufficiently high to make the membrane potential immediately exceed the threshold. In particular, for the inhibitory neuron:

$$vThresh - vRest = -40.0mV - (-60.0mV) = 20mV \tag{D.1}$$

Weight	21mV

Table D.4: Excitatory connection parameters

D.5 Connection from inhibitory to excitatory layer

The connection between the inhibitory neurons and the excitatory ones is instead of the type one-to-others, in the sense that each inhibitory neuron is connected to all the excitatory ones except from the one that presents an excitatory connection with it. Again all the weights are set to a constant value, equal for all the neurons. In this way the generation of an excitatory spike has the same inhibitory effect on all the other neurons.

Weight	-15mV

Table D.5: Inhibitory connection parameters

D.6 Other parameters

Two other parameters that deserve to be cited are:

Scale factor	0.3
Shift factor	0.01mV
Normalization factor	78.4mV

Table D.6: Weights parameters

1. The scaling factor used in the generation of the weights: the weights are obtained through a random initialization during the creation of the network. In the most general case random numbers are generated in a range between 0 and 1 and then scaled within the desired interval. This interval, once fixed the voltage parameters of the neurons, depends on how many input the each neuron receives. If the values weights are too low no output spike is generated, while if they are too high too many spikes can be produced and this can negatively affect the learning or the performance of the network. The scale factor is a parameter that needs to be finely tuned along the design. The value reported in table refers to the connection of the input layer to the first excitatory one, but could not be valid for the other layers.

2. Shift factor: together with the scale factor this decides the range in which the weights are generate. In particular the scale factor decides the width of the range, while the shift factor modifies its lower and upper bounds. It can for example guarantee that no null weights are present after the initialization.

Again the reported value refers to the connection between the input layer and the first excitatory one.

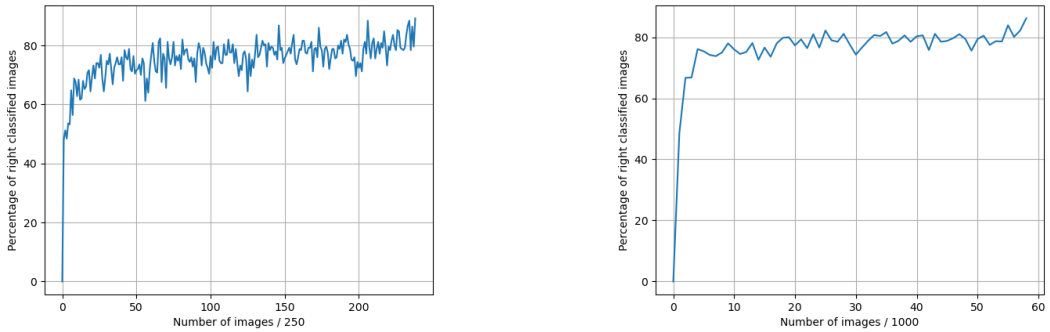
3. Normalization factor: the training process includes a normalization step which guarantees limited values of the weights. The normalization should keep the weights at values that are coherent with the ones obtained during the initialization, without increasing or decreasing them too much. As before the reported value is valid for the input connection.

Appendix E

Results

The architecture studied during the development of the software simulation is characterized by four-hundred neurons. So first of all the results obtained with such structure are reported. The accuracy and its evolution along the training are the same for the Brian 2 simulation and for the handmade model and so they are unified in a single section. The time required for the computation is different in the two cases and so a comparison between the two is presented. The results obtained after the simplification of the model are presented in a separated section because the accuracy evolution results slightly modified in that case. Finally an estimation of the performance that can be obtained with the hardware accelerator is presented.

E.1 Accuracy of the model on the training set



(a) 250 images update **(b)** 1000 images update

Figure E.1: Evolution of the training accuracy

The final accuracy that can be reached with the developed model on the training set is around 86%. Figure E.1a and E.1b show the temporal evolution of the accuracy along the training evaluated respectively every 250 and 1000 images. The parameters of the model are the ones reported in chapter D. It can be observed that the increase in the accuracy is quite slow after about 10000 images, but still present. In addition its growth is not smooth but characterized by many oscillations. This is normal considering that the training is unsupervised and the output classification is gradually adapted.

E.2 Accuracy of the model on the test set

The average accuracy reached on the training set is around 82.0% and this is compatible with the training accuracy so there is not a relevant overfitting of the training data.

E.3 Training and test duration

One other relevant parameter to evaluate is the duration of both the training and the test procedures over the complete datasets. The measurements have been performed on a MacBook Pro 2016, with an Intel Core i5 dual-core processor. The obtained results are:

Version	Training	Test
Brain 2	20h 32min	1h 30min
Manual	24h 10min	1h 45min
Simplified manual	15h 24min	1h 06min

Table E.1: Training and test time

Let's now analyze the obtained data:

1. Difference between the Brian 2 and the manual versions: a relevant difference in the time required for both the training and the test can be observed between the two versions. This depends on how the code is executed in the two cases: as explained in section 11.1 Brian 2 tries to optimize the computations whenever possible, accelerating them using C and C++. If compared with the pure python implementation this avoids the interpreting overhead and reduces the required time.

2. Reduction of the training and test time in the simplified manual version. Here the modifications that allow for a faster execution are many:

- The absence of inhibitory neurons, which reduces the needed computations.
- The direct reset of the membrane to its rest potential instead of waiting for a full rest period.
- The simplified homeostasis update also brings a little advantage, but it is not so significant.
- The reduction of the homeostatic increase of the threshold implies that fewer images require to be analyzed multiple times.

E.4 Estimation of the performance of the hardware accelerator

Since the designed accelerator for the moment allows only an offline training the evaluation is performed considering a test cycle over a single image. Table E.2 reports the results obtained on the different software versions as a comparison.

Version	Duration
Brain 2	0.23s
Manual	0.27s
Simplified manual	0.16s

Table E.2: Test time for a single image

Working with a training interval of 350ms and with a step duration of 0.1ms the total amount of cycles is

$$N_{train} = \frac{350ms}{0.1ms} = 3500 \tag{E.1}$$

In order to estimate the number of clock cycles required by the update of each neuron figure L.3 can be considered. It can be seen that the exponential decay and the membrane reset both require two clock cycles. In the worst case in which each single cycle presents at least one active spike, all the input excitatory and the inhibitory spikes must be considered. In this case, considering a structure with a single layer of 400 neurons, the total amount of clock cycles needed for a compete update is:

$$N_{update} = 2 + 784 + 400 = 1186 \tag{E.2}$$

In order to consider the propagation of the signals from the layer's control unit to the neurons, and being this an estimation the value can be rounded to

$$N_{update} = 1200 \tag{E.3}$$

So, since the neurons work in parallel, the overall amount of cycles required to update the network on a single image is:

$$N_{tot} = N_{train} \cdot N_{update} = 3500 \cdot 1200 = 4.2 \cdot 10^6 \tag{E.4}$$

In order to compare this quantity with the software versions, let's consider the minimum time required to test an image in python, obtained with the ... version. The goal of the accelerator, as the name itself suggest, is to reduce te elaboration time. So with the upper limit of 0.16s the required clock period is:

$$T_{clk}^{max} = \frac{0.16s}{4.2 \cdot 10^6} = 3.8 \cdot 10^{-8}s \tag{E.5}$$

which corresponds to a clock frequency of:

$$f_{clk}^{min} = \frac{1}{T_{clk}^{max}} = 26MHz \tag{E.6}$$

So a platform with a clock frequency of at least 26MHz is required to outdo the software performance.

An analysis of the input and inhibitory spikes distribution shows that around the 60% of the time steps do not present any active spike. In this case the network update is reduced to the elaboration of one of the spikes set between the excitatory and the inhibitory one, or, in the best case in which there are no spikes on both sides, to the simple exponential decay or membrane reset, operations that require two clock cycles.

All this analysis does not consider the overhead introduced by the transport of the data from the computer to the evaluation board, from the processor to the FPGA and the two same path in the opposite direction.

Appendix F

Load MNIST

Load dataset

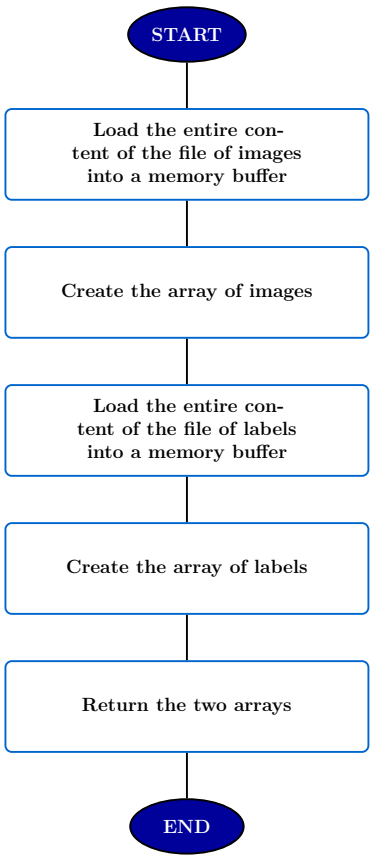


Figure F.1: Load the entire dataset in two arrays of labels and images. Function *loadDataset()*.

Load the entire content of the file

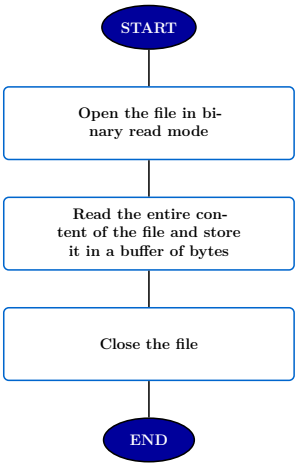


Figure F.2: Load the entire content of the file into a memory buffer. Function *readFile()*.

Create the array of images/labels

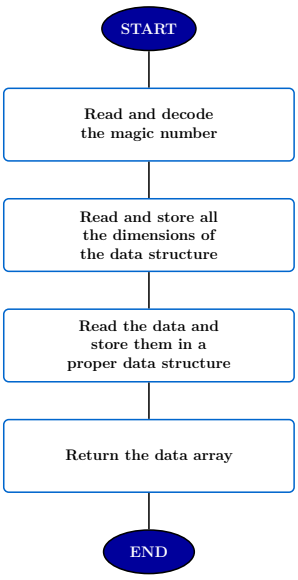


Figure F.3: Convert the memory buffer into a *numpy* array. Function *idxBufferToArray()*.

Read and decode the magic number

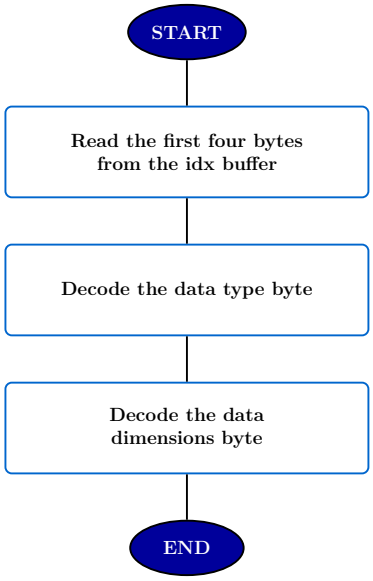


Figure F.4: Read and decode the magic number. Function *magicNumber()*.

Read the data and store them in a proper data structure

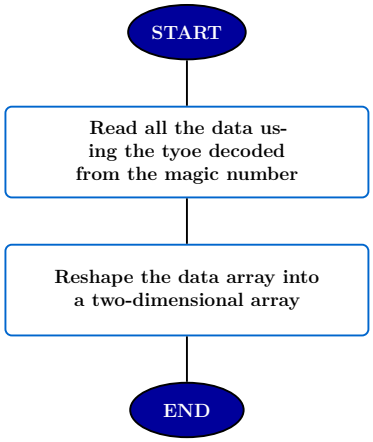


Figure F.5: Read the entire data buffer and store it into a *numpy* array. Function *loadData()*.

Reshape the array of data into a two dimensional array

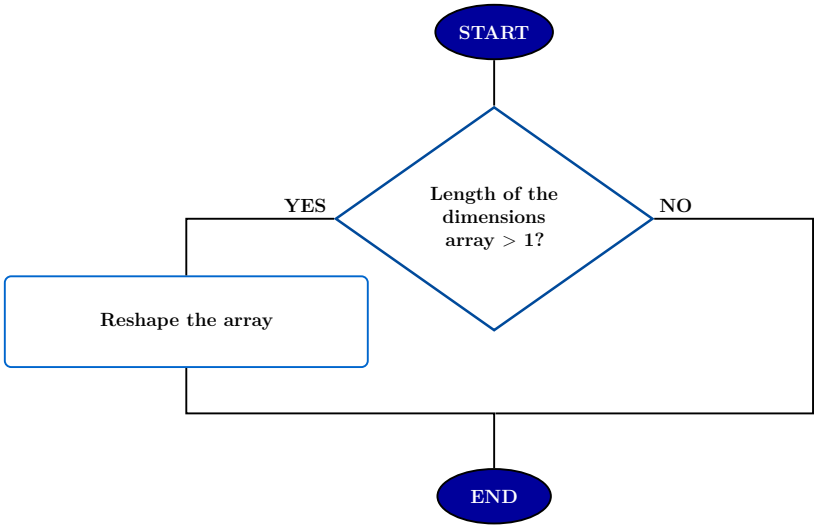


Figure F.6: Reshape the data if necessary. Function *reshapeData()*.

Appendix G

Training algorithm

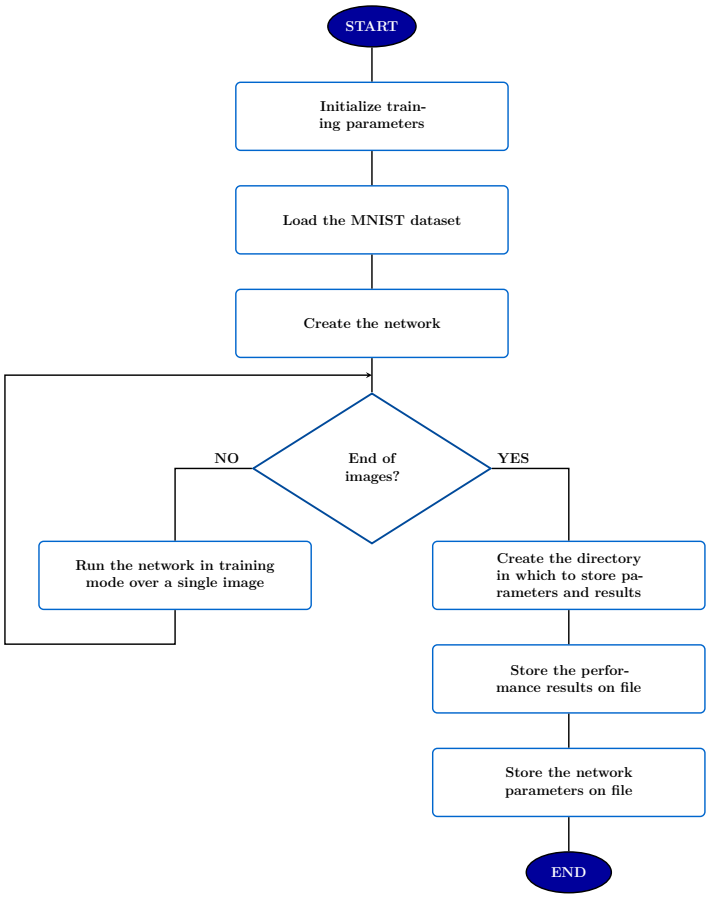


Figure G.1: Complete training algorithm. Main script.

Run the network over a single image

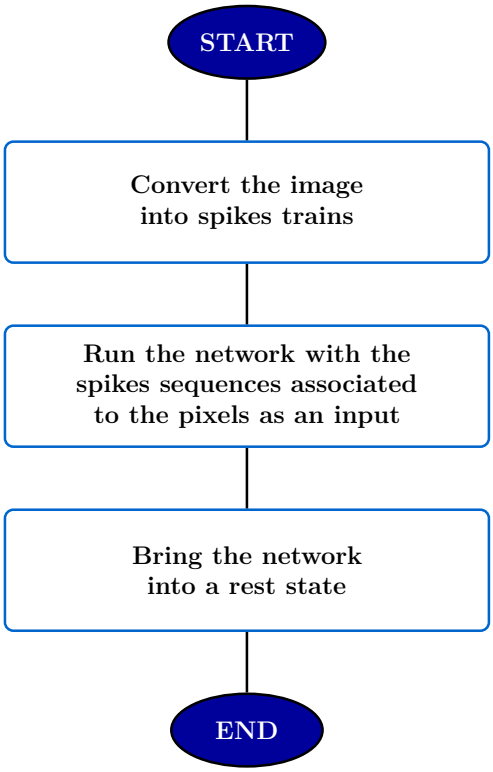


Figure G.2: Function *singleImageRun()*

Run the network with the spikes sequences associated to the pixels as an input

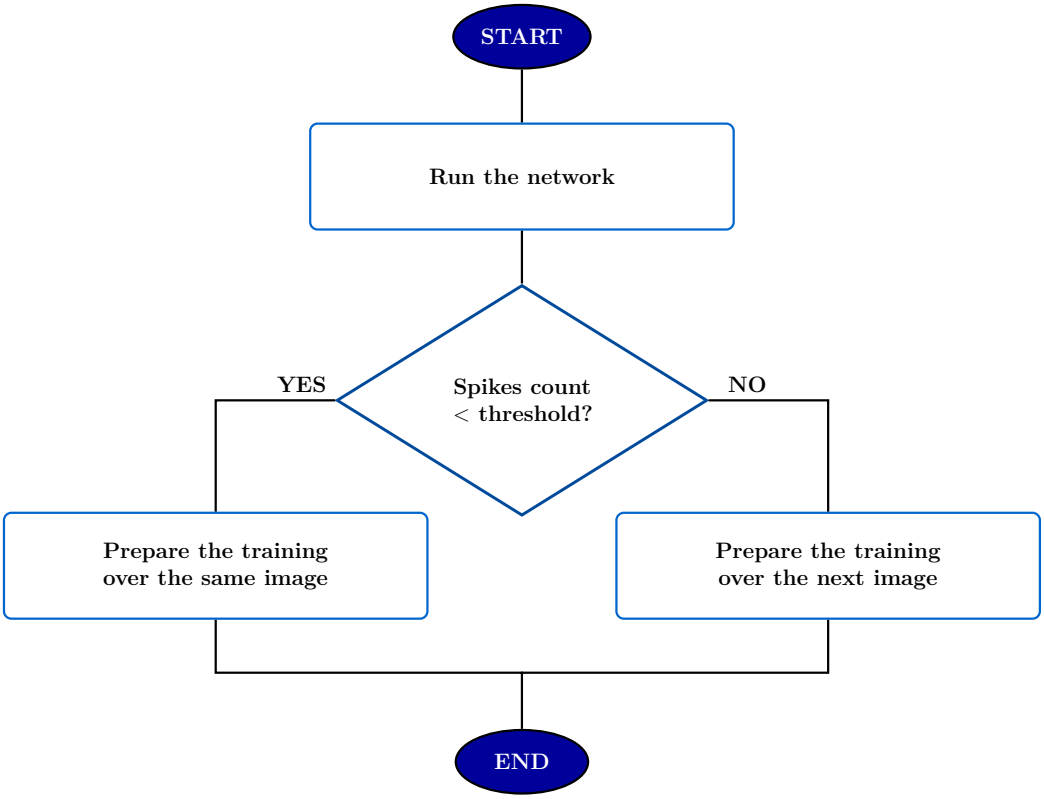


Figure G.3: Function *runNetwork()*

Prepare the training/test over the next image

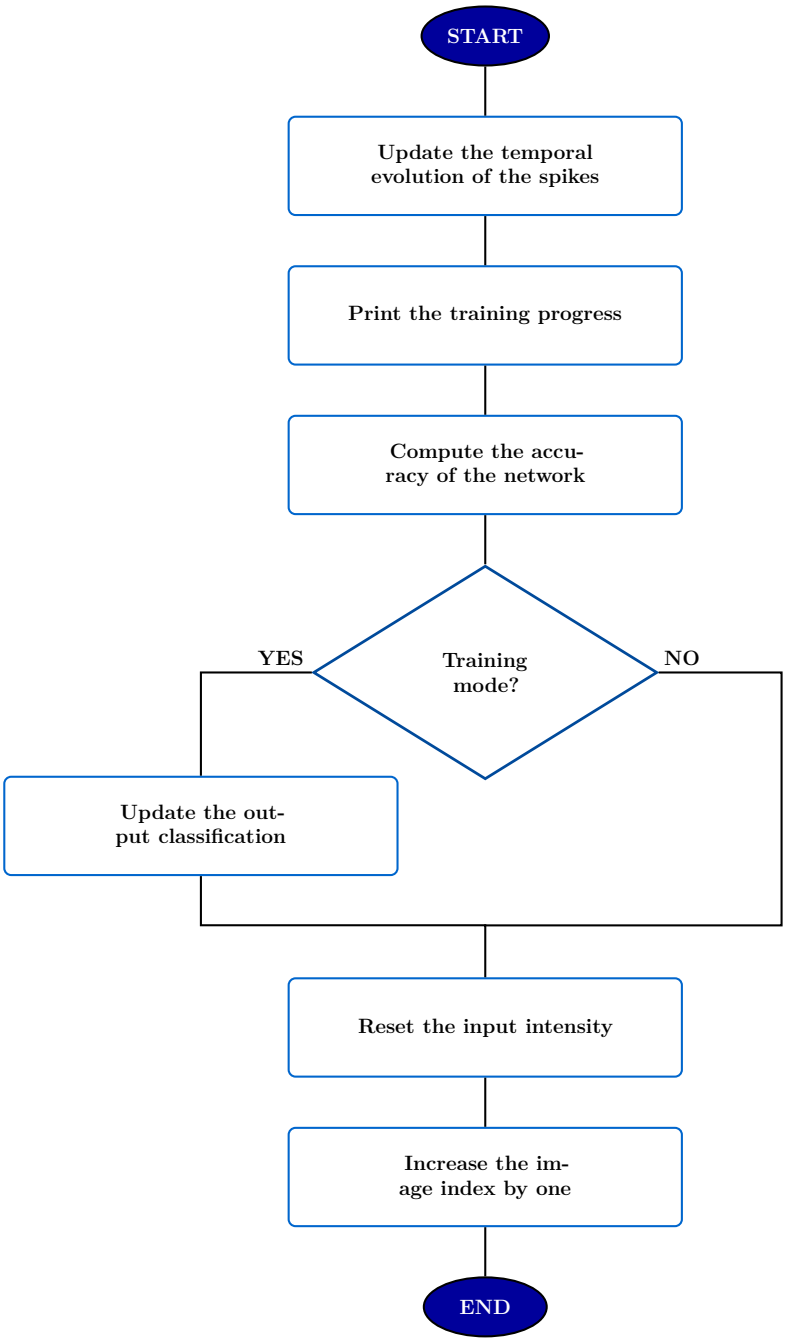


Figure G.4: Function *nextImage()*

Print the training/test progress

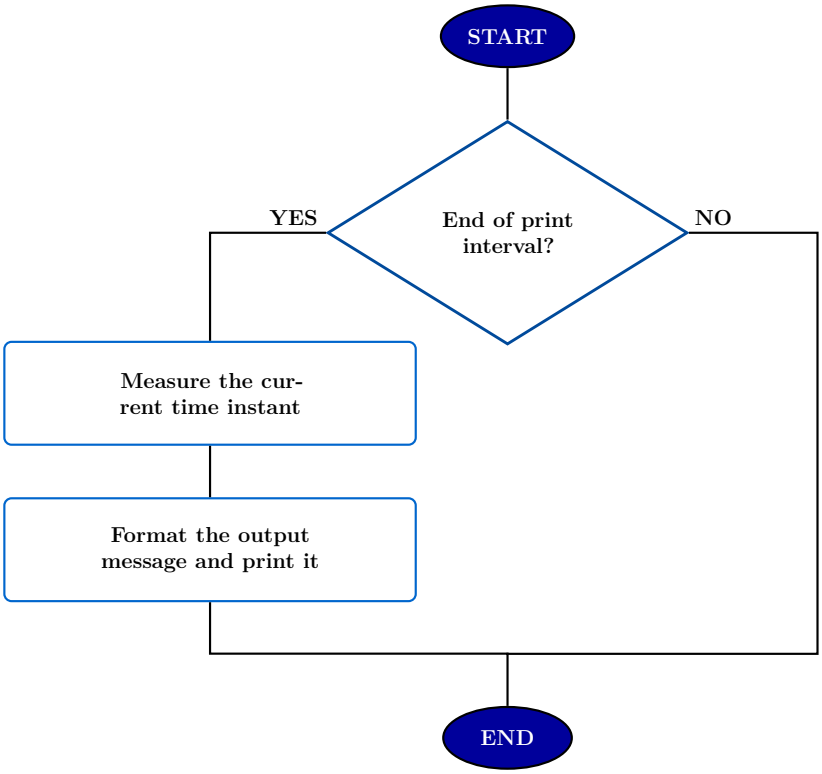


Figure G.5: Function *printProgress()*

Compute the accuracy of the network

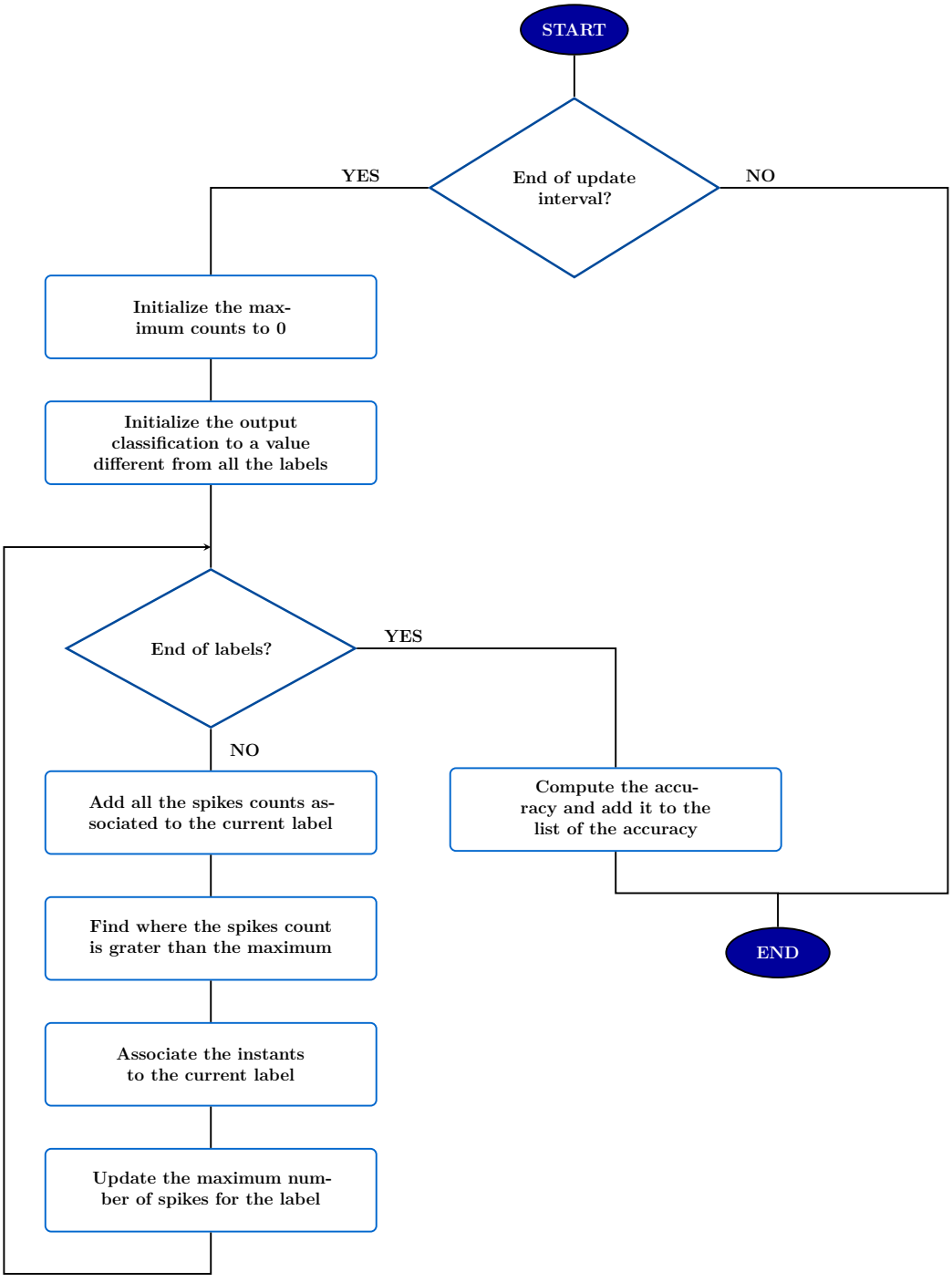


Figure G.6: Function *computePerformance()*

Compute the accuracy and add it to the temporal evolution of the accuracy

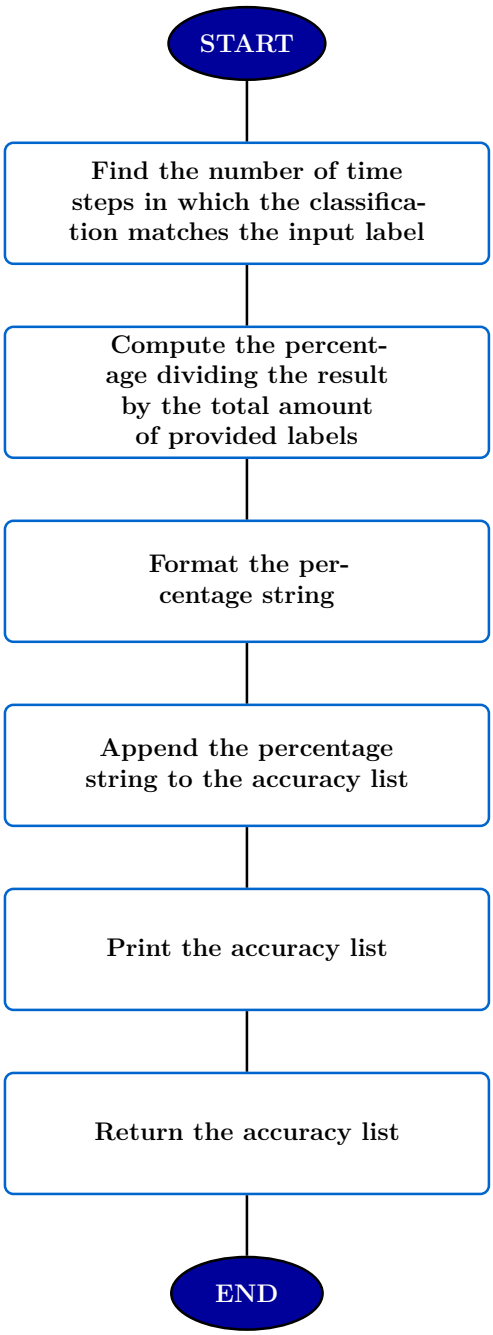


Figure G.7: Function *updateAccuracy()*

Update output classification

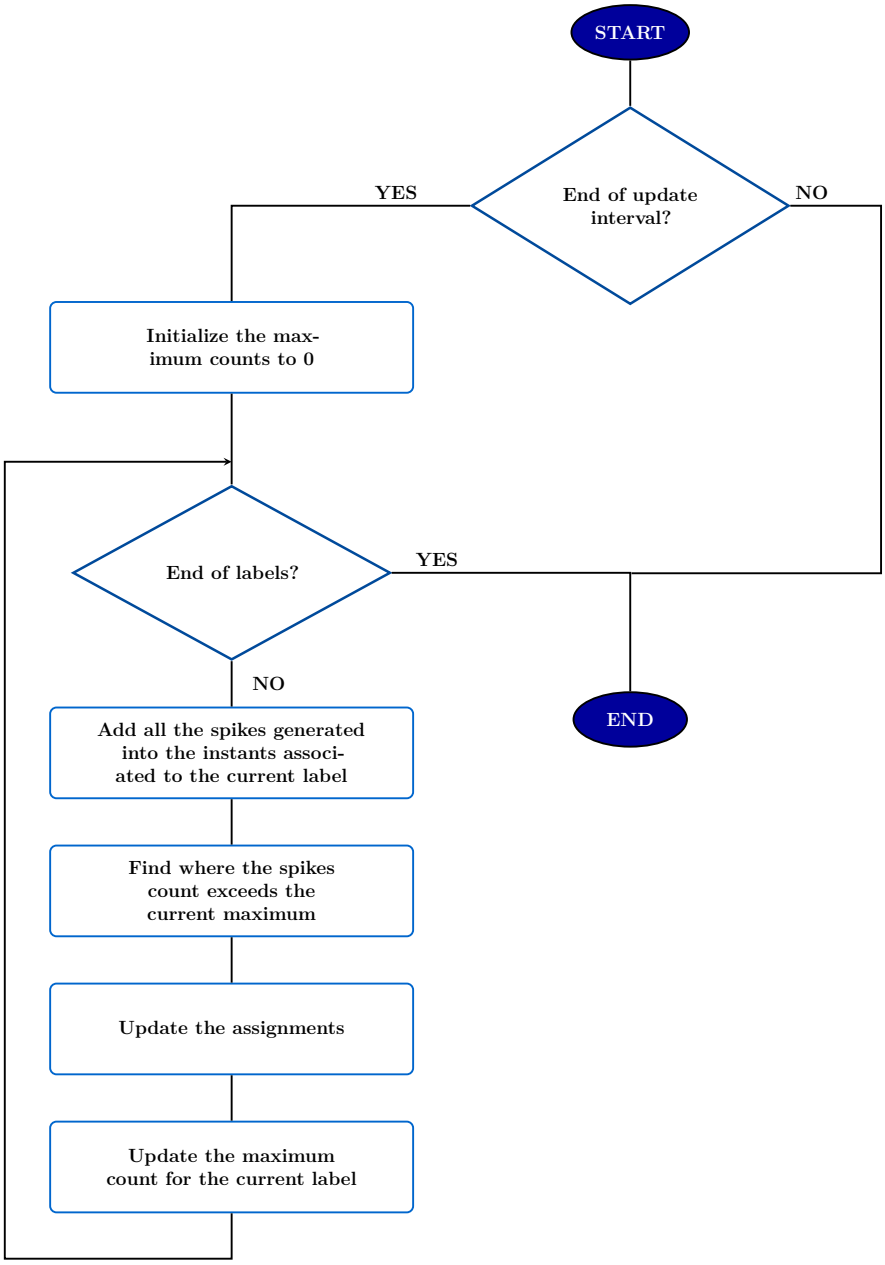


Figure G.8: Function *updateAssignements()*

Prepare the training/test over the same image

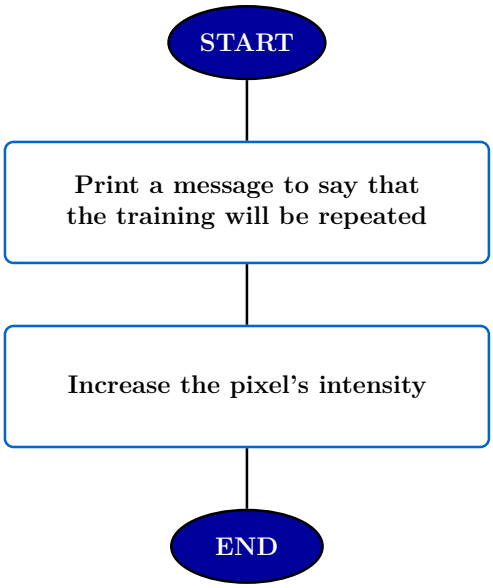


Figure G.9: Function *repeatImage()*

Bring the network into a rest state

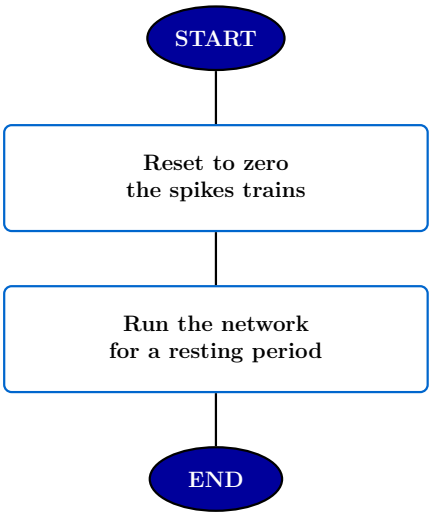


Figure G.10: Function *rest()*

Create the directory

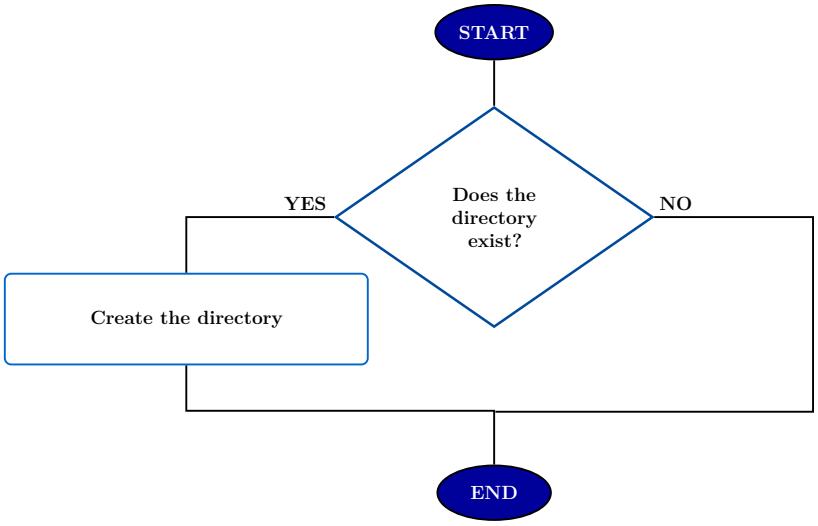


Figure G.11: Function *createDir()*.

Store the performance results on file

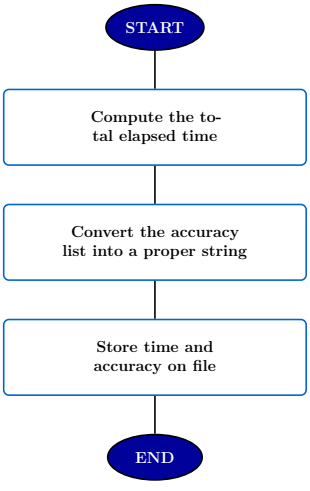


Figure G.12: Function *storePerformance()*

Store the network parameters on file

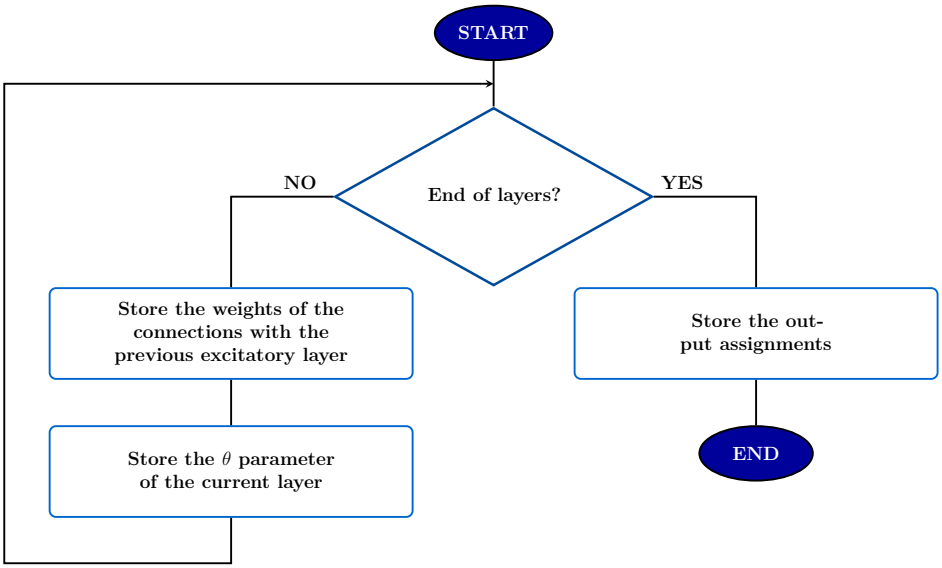


Figure G.13: Function *storeParameters()*

Appendix H

Test algorithm

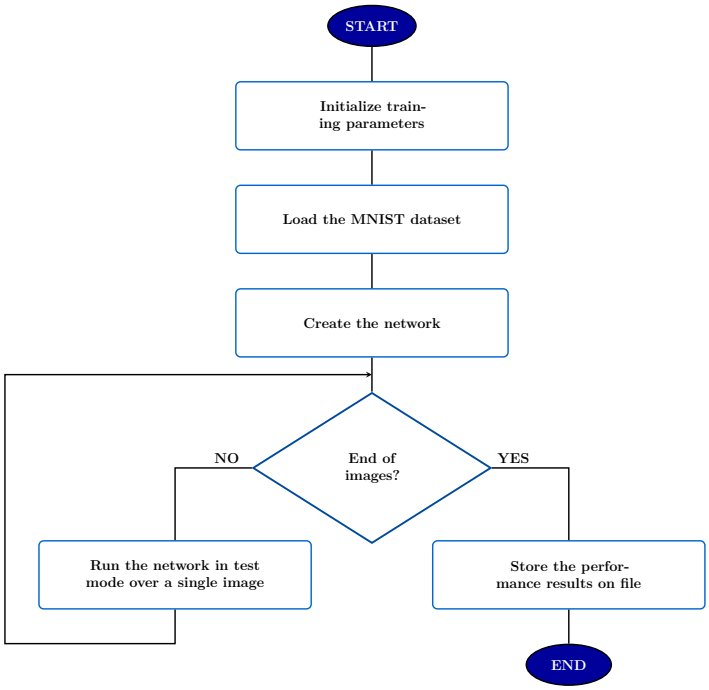


Figure H.1: Complete test algorithm. Main script.

Appendix I

Brian 2 simulation

Create the network

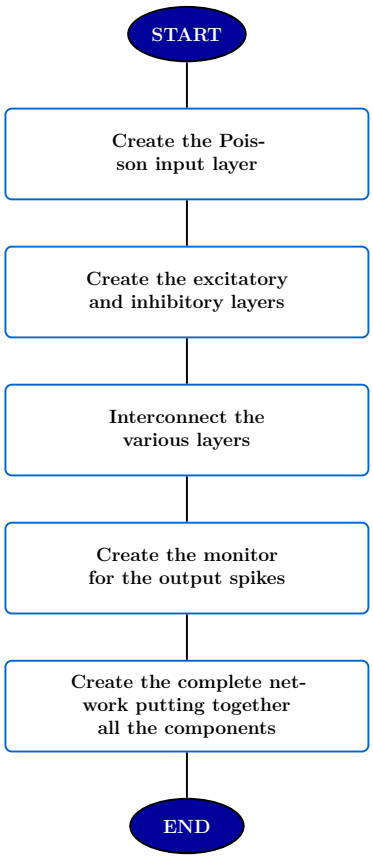


Figure I.1: Function *createNetwork()*

Create the excitatory and inhibitory layers

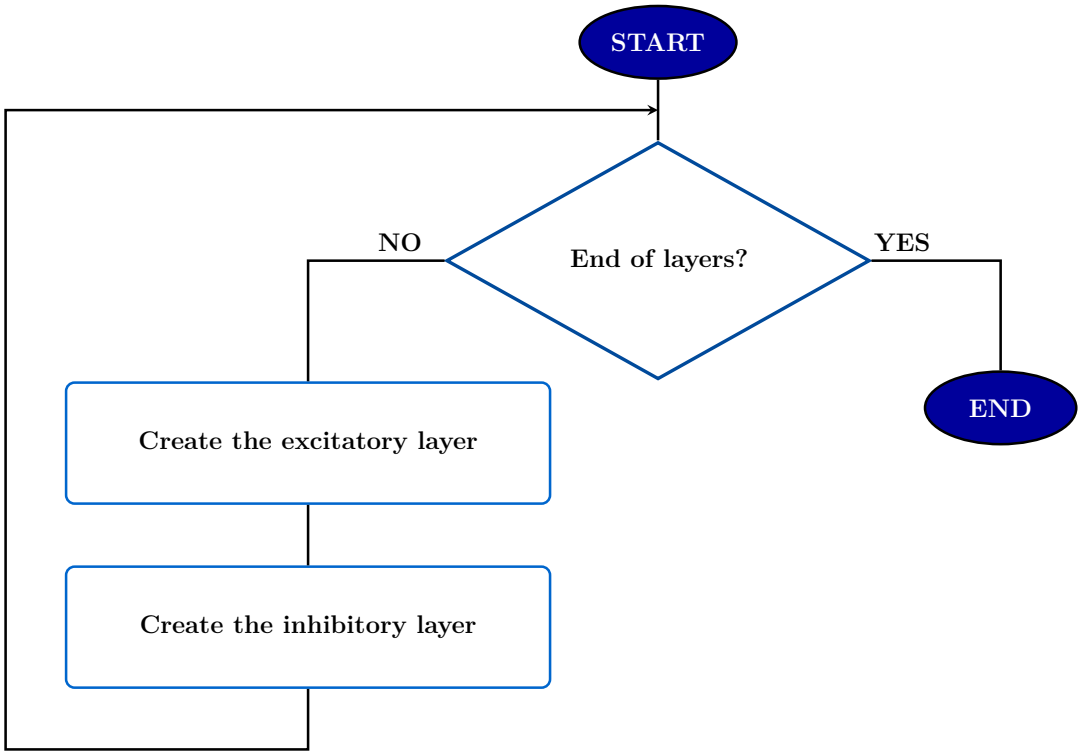


Figure I.2: Function *createLayersStructure()*

Create the layer

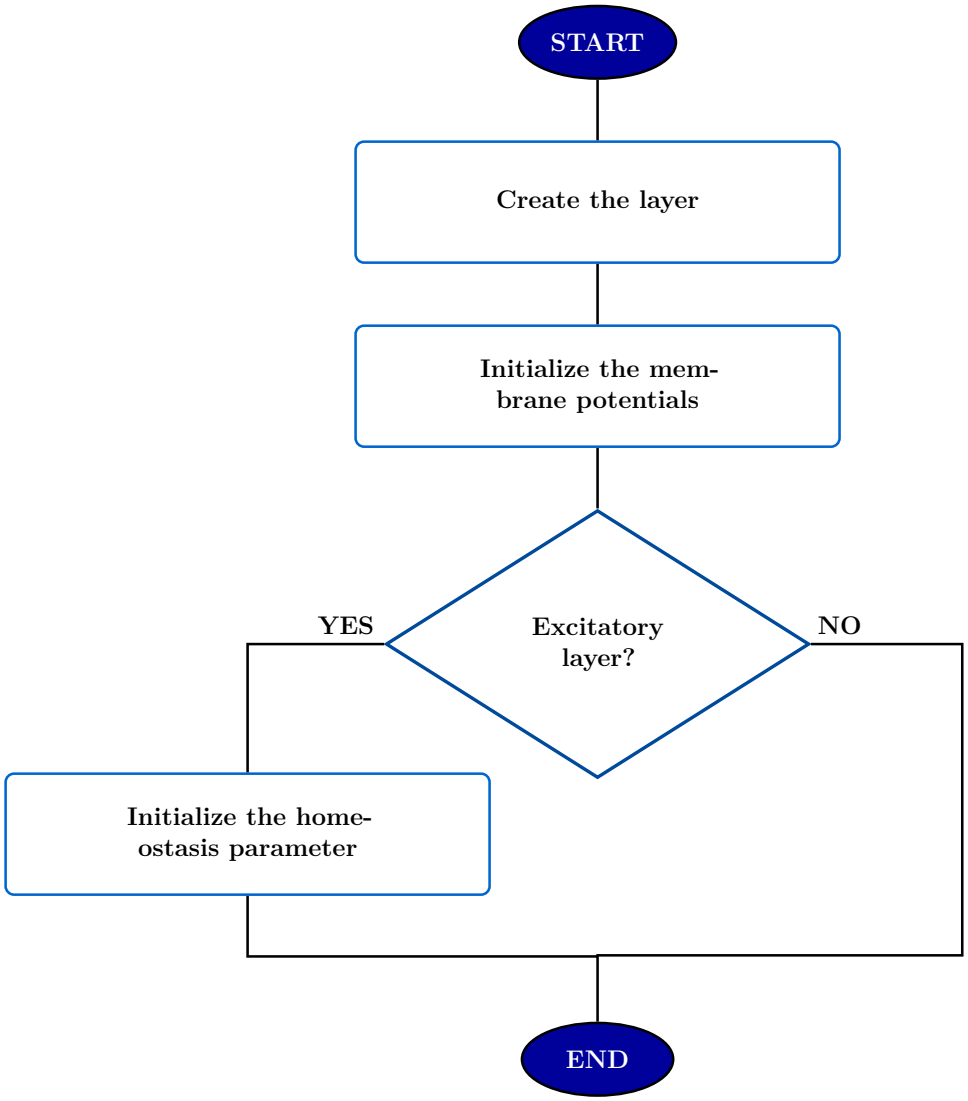


Figure I.3: Function *createLayer()*

Interconnect the various layers

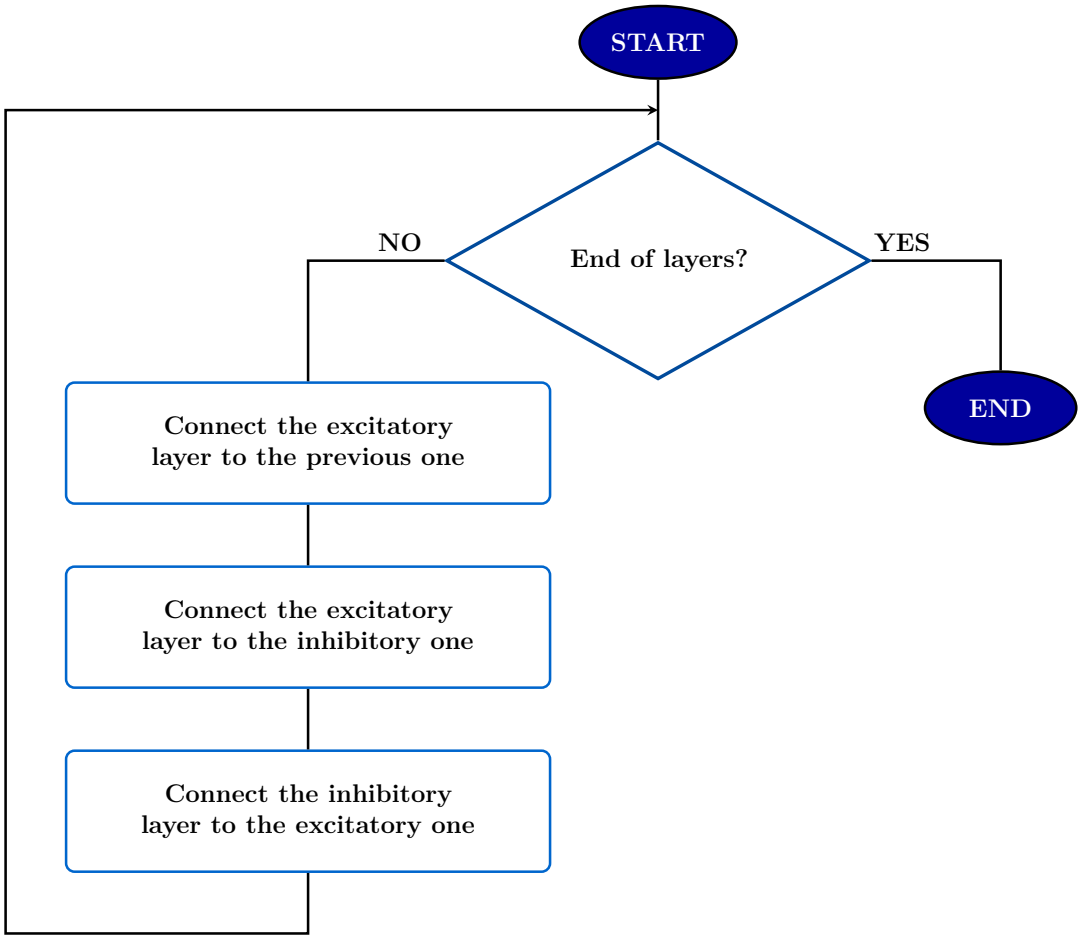


Figure I.4: Function *connectLayersStructure()*

Connect the excitatory layer to the previous one

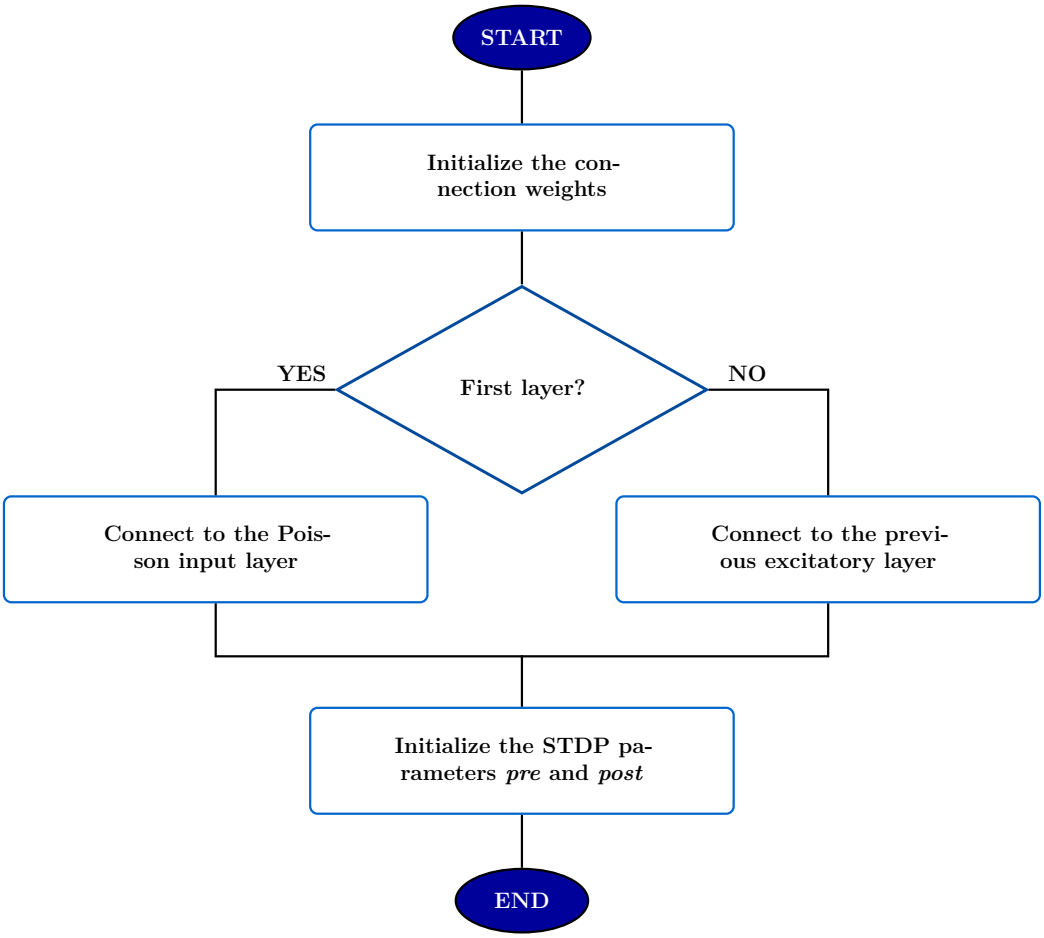


Figure I.5: Function *exc2excConnection()*

Connect layers

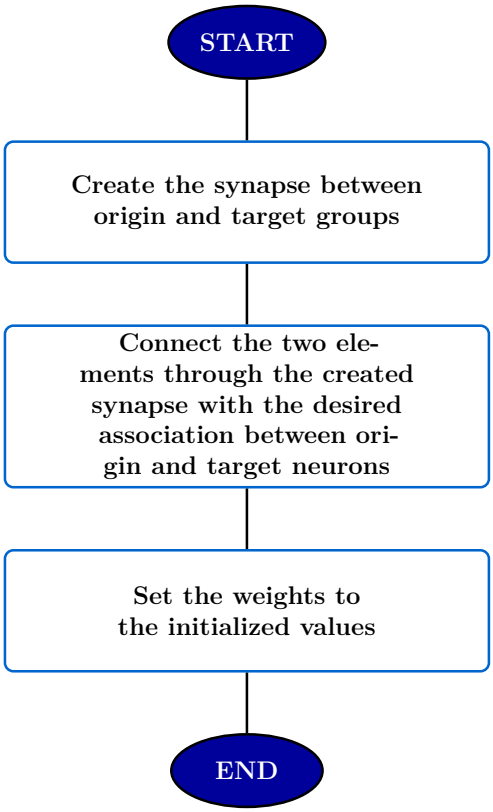


Figure I.6: Function *connectLayers()*

Convert the image into spikes trains

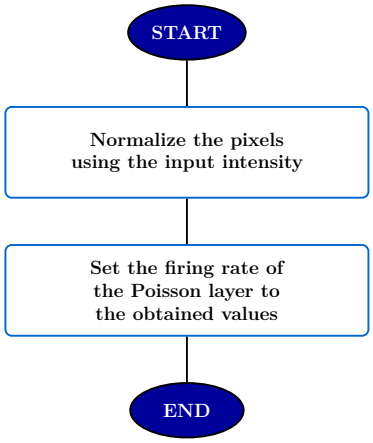


Figure I.7: Function *imgToSpikeTrains()*

Train the network over the pixels spikes trains

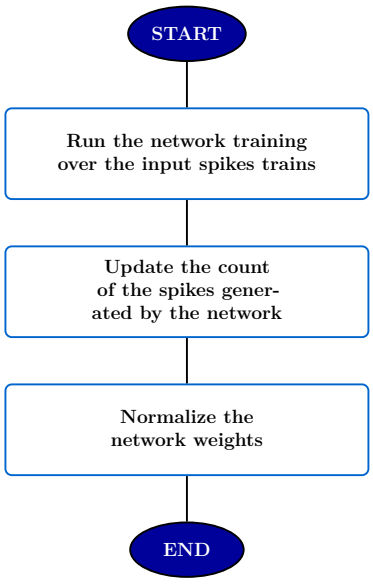


Figure I.8: Function *run()*

Update the count of the spikes generated by the network

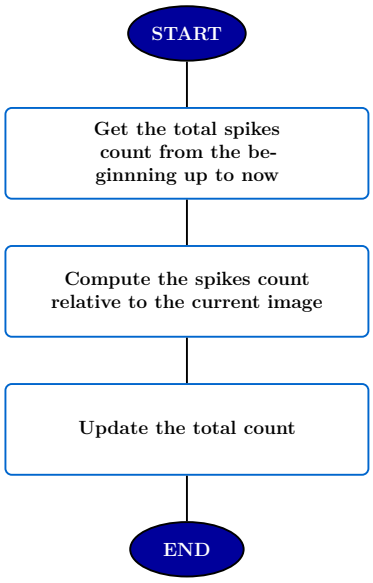


Figure I.9: Function *updatePulsesCount()*

Normalize the network weights

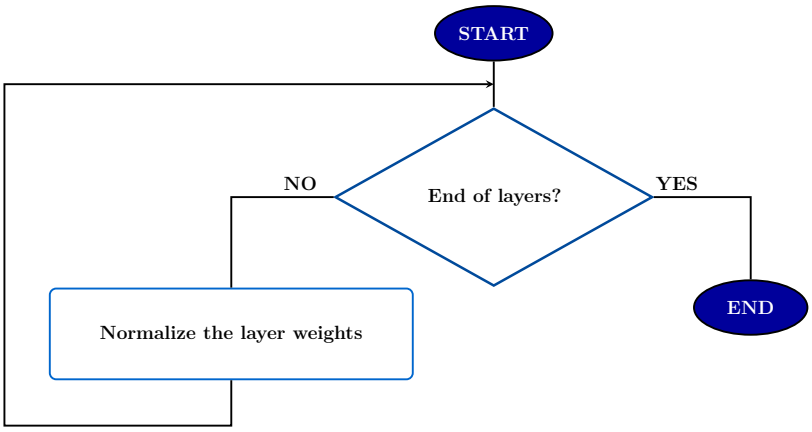


Figure I.10: Function *normalizeNetWeights()*

Normalize the layer weights

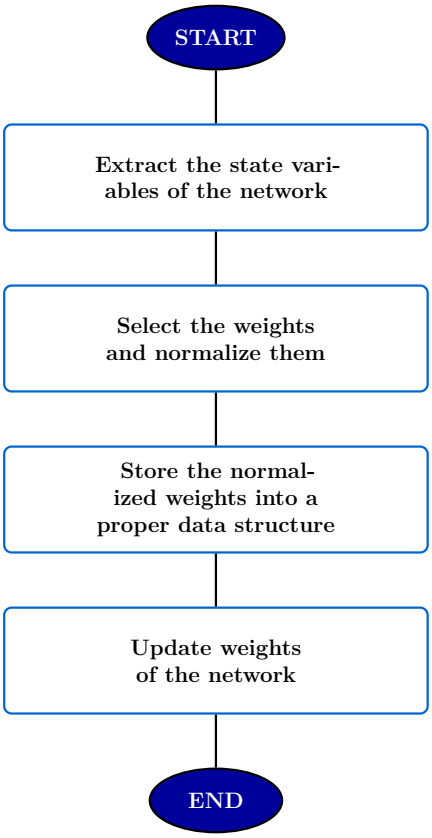


Figure I.11: Function *normalizeLayerWeights()*

Normalize the weights

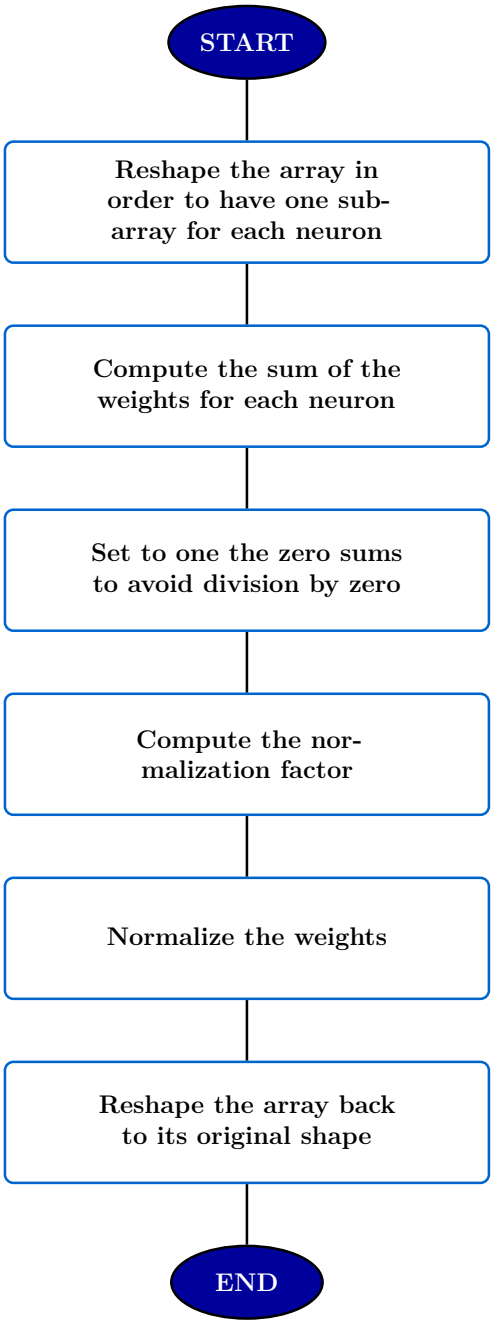


Figure I.12: Function *normalizeWeights()*

Appendix J

Custom implementation

Create the network

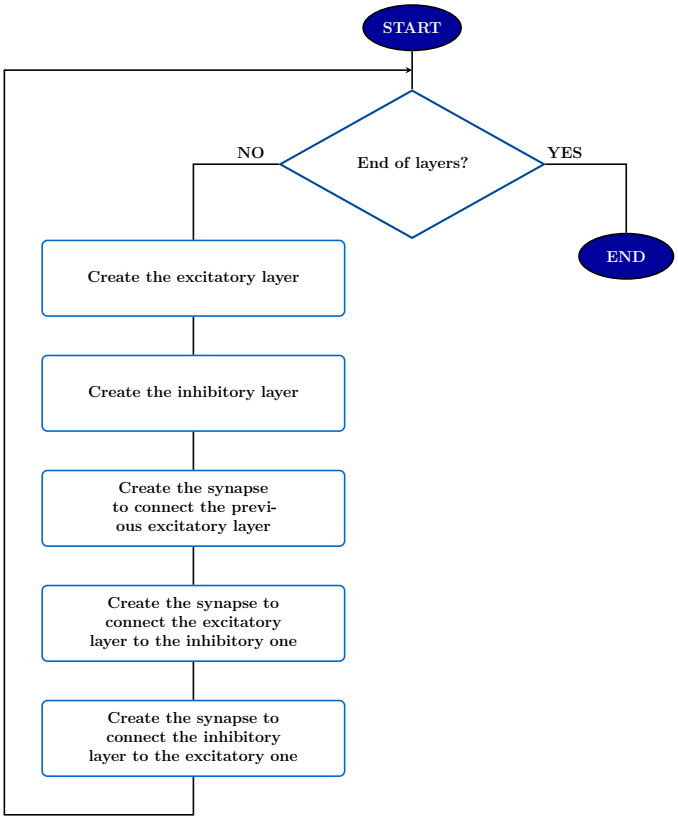


Figure J.1: Function *createNetwork()*

Create the layer

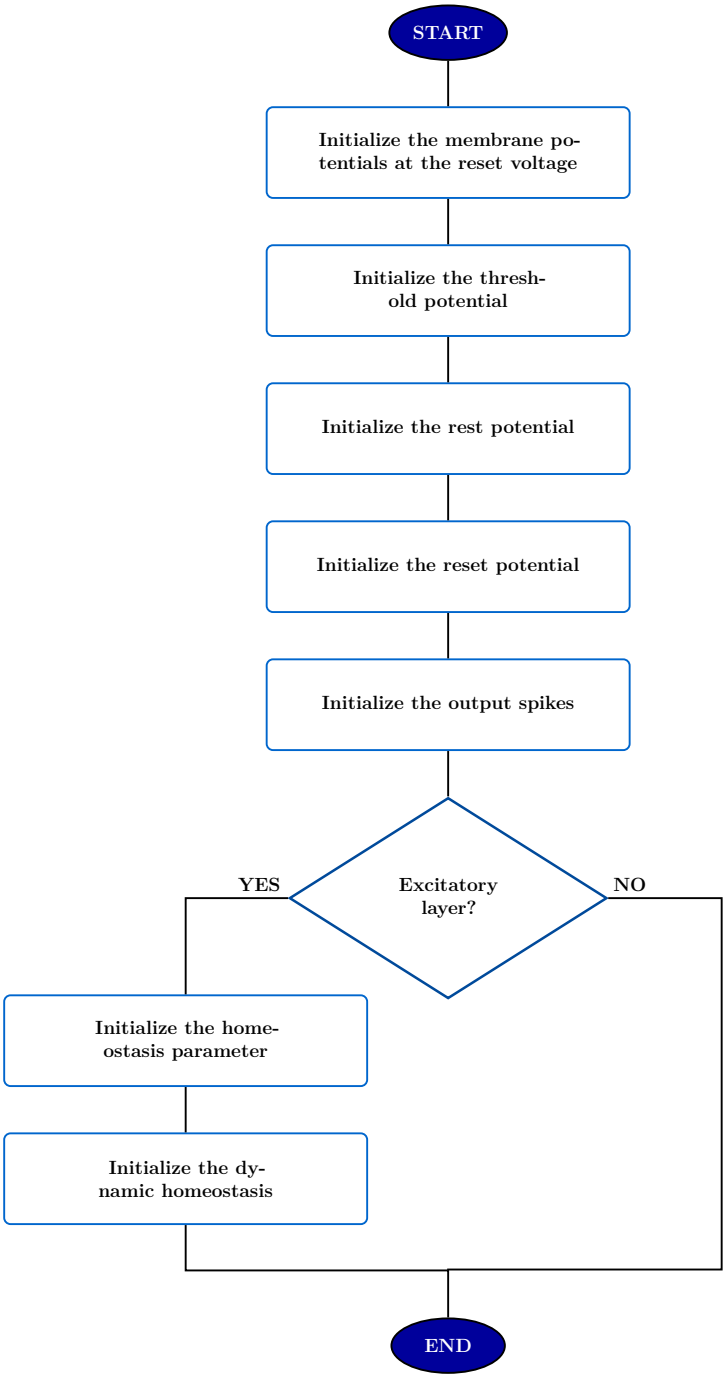


Figure J.2: Function *createLayer()*

Create the synapse to connect the previous excitatory layer

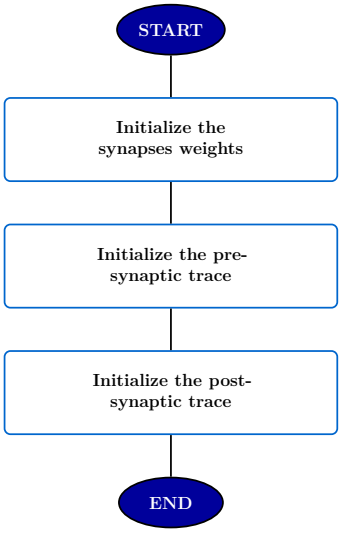


Figure J.3: Function *intraLayersSynapses()*

Create the synapse to connect the excitatory layer to the inhibitory one and vice versa

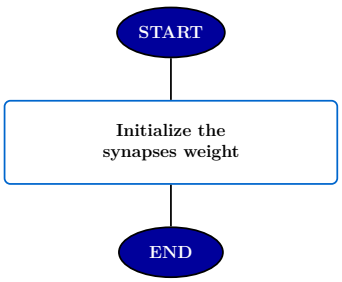


Figure J.4: Function *interLayerSynapses()*

Convert the image into spikes trains

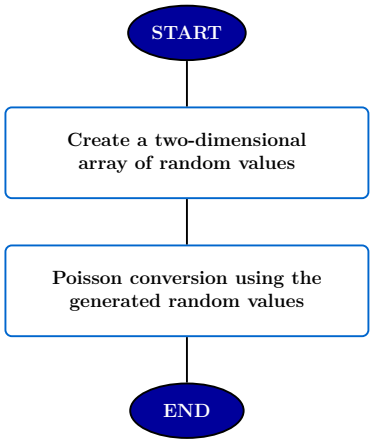


Figure J.5: Function *imgToSpikeTrains()*

Poisson conversion using the generated random values

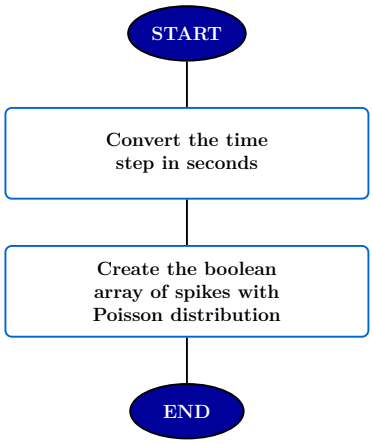


Figure J.6: Function *poisson()*

Run the network over the pixels spikes trains

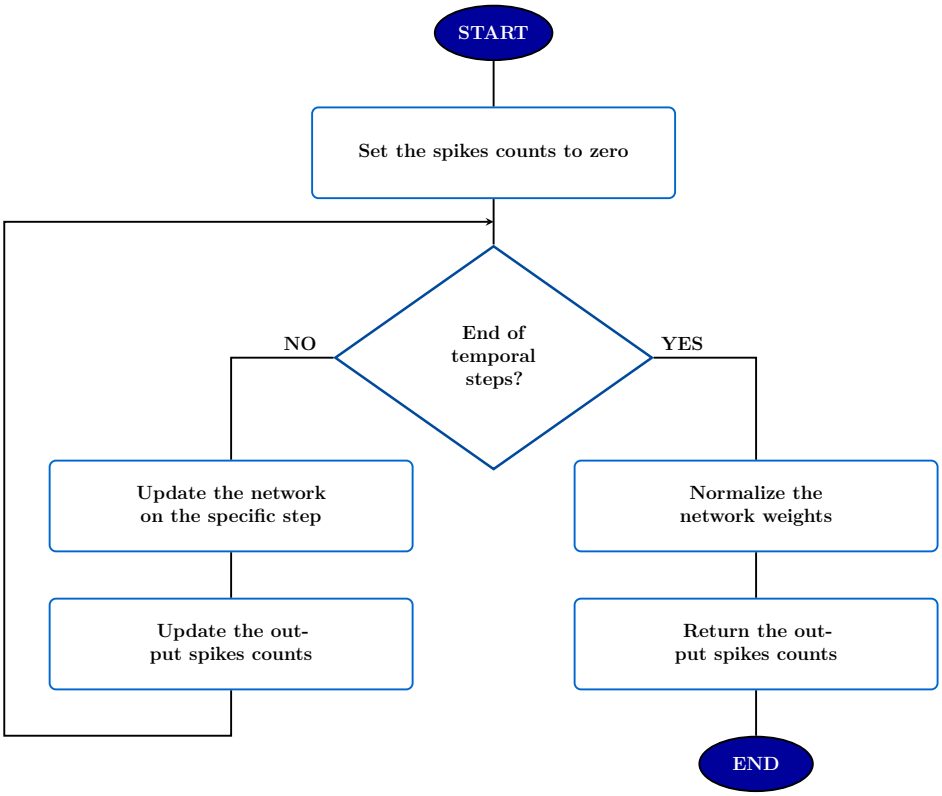


Figure J.7: Function *run()*

Update the network on the specific step

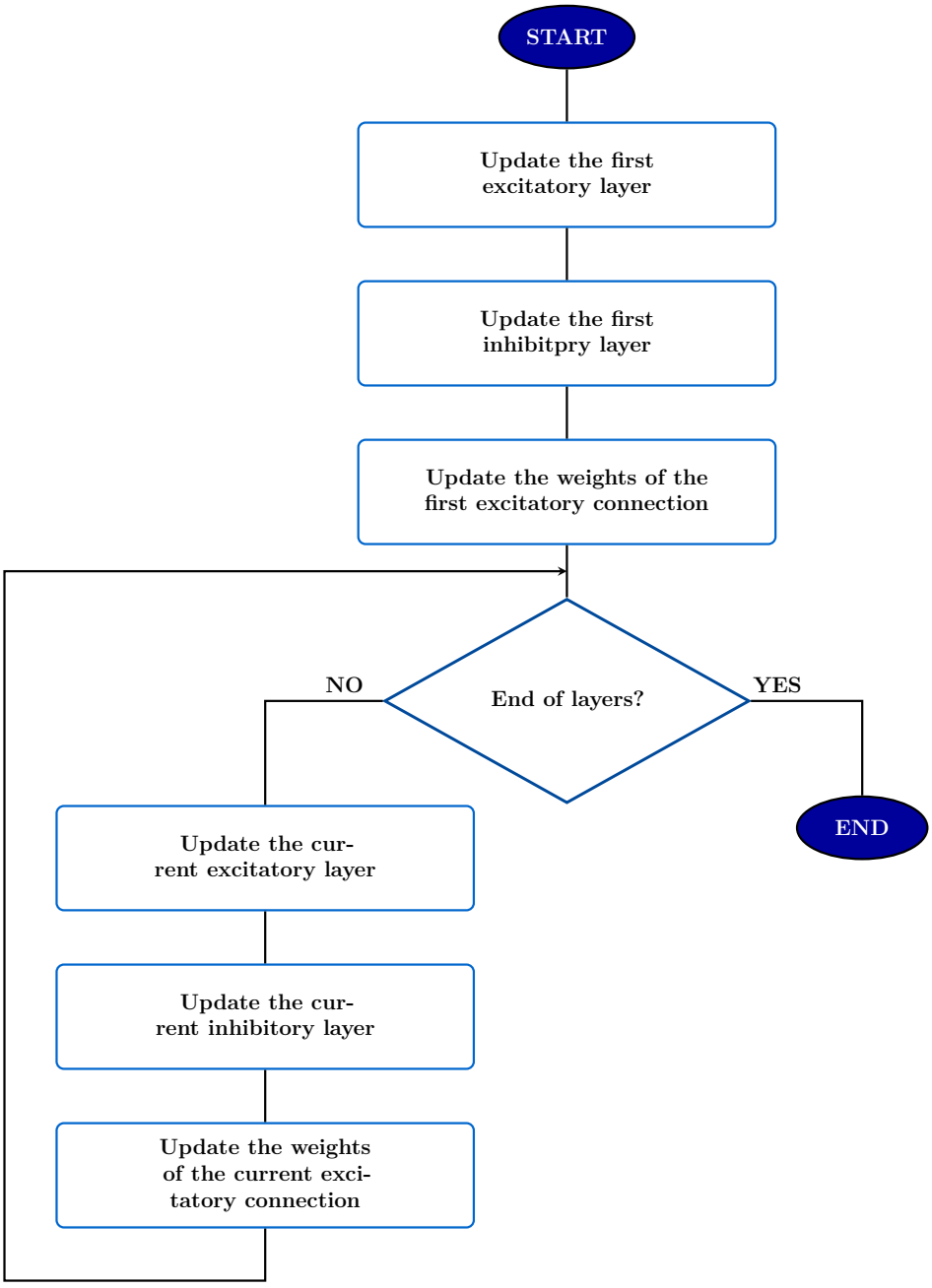


Figure J.8: Function *updateNetwork()*

Update the excitatory layer

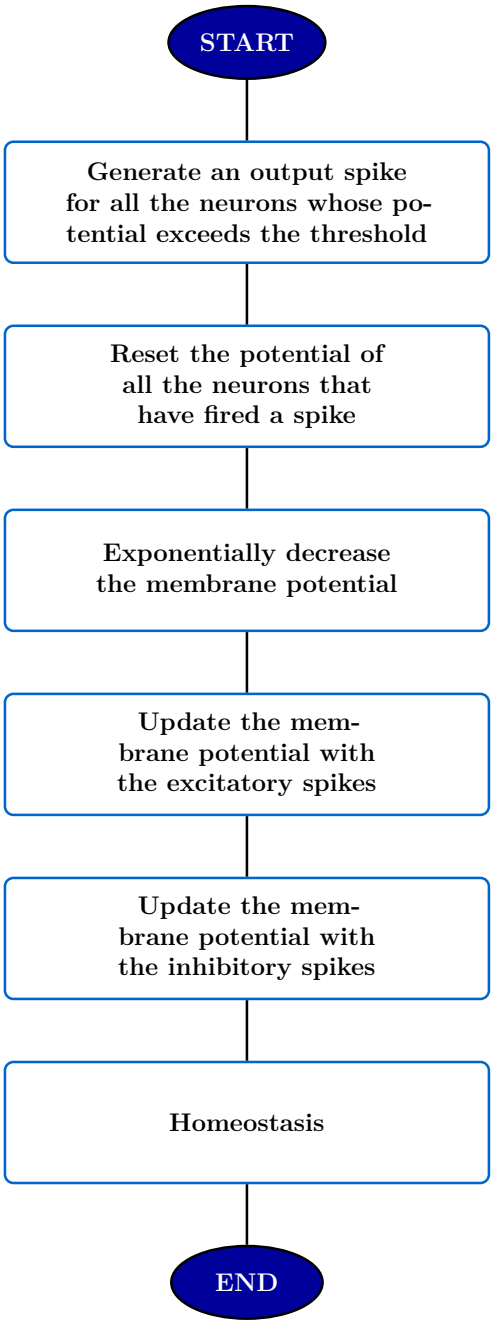


Figure J.9: Function *updateExcLayer()*

Update the inhibitory layer

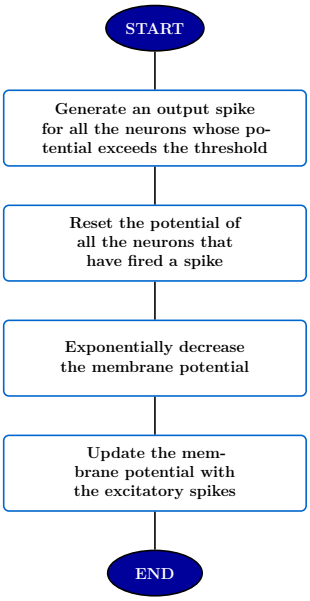


Figure J.10: Function *updateInhLayer()*

Update the membrane potential with the inhibitory spikes

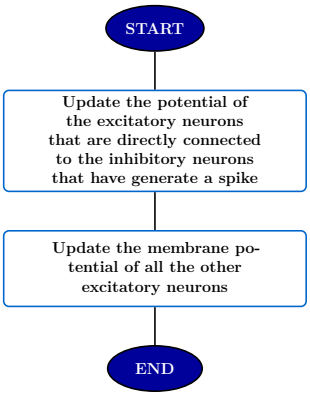


Figure J.11: Function *all2othersUpdate()*

Total number of connected active inhibitory neurons

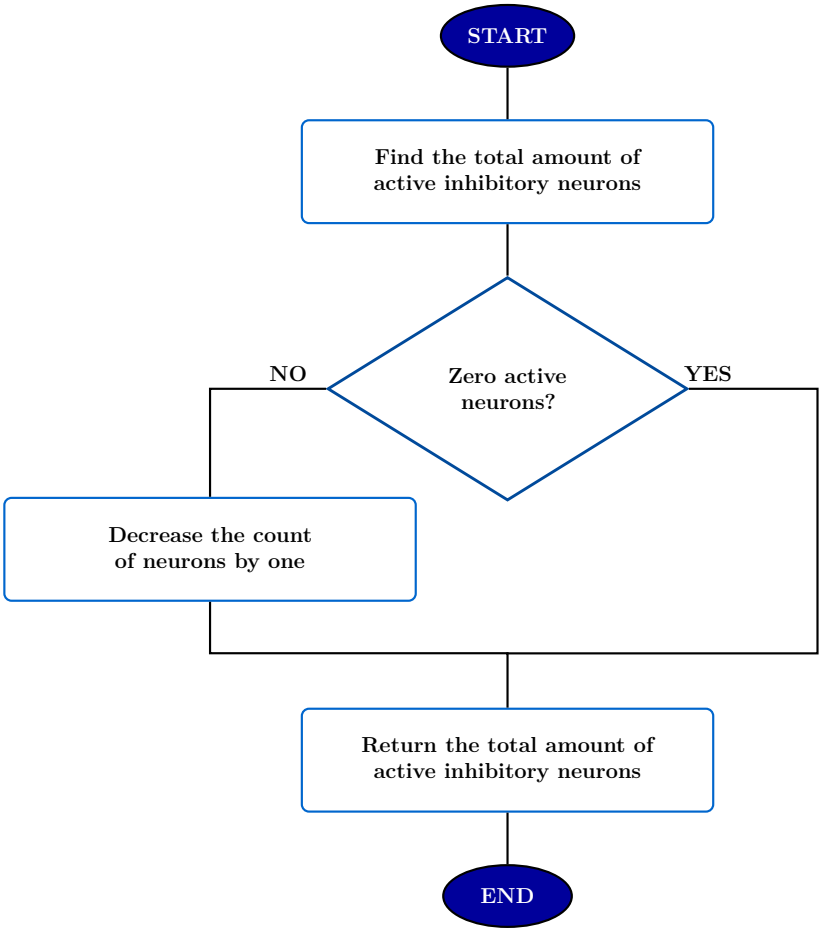


Figure J.12: Function *unconnectedSpikes()*

Homeostasis

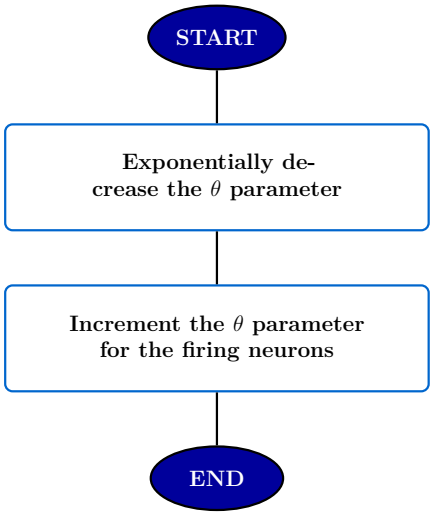


Figure J.13: Function *homeostasis()*

Update the weights of the excitatory connection

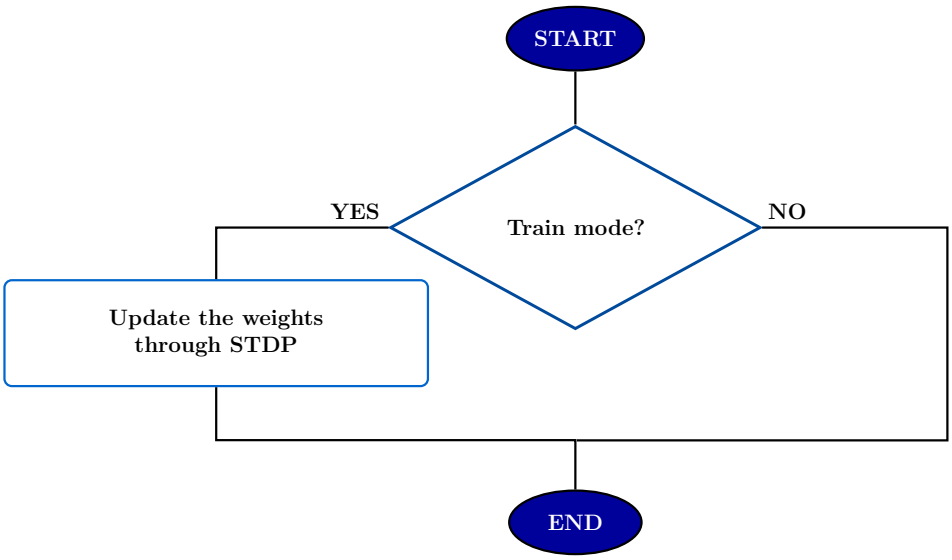


Figure J.14: Subportion of *updateNetwork()*, this has not a dedicated function.

Update the weights through STDP

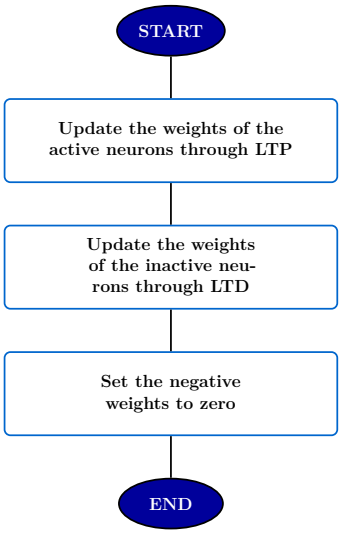


Figure J.15: Function *stdp()*

Update the weights of the active neurons through LTP

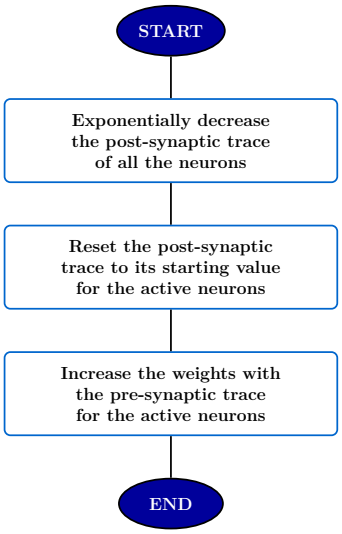


Figure J.16: Function *ltp()*

Update the weights of the active neurons through LTD

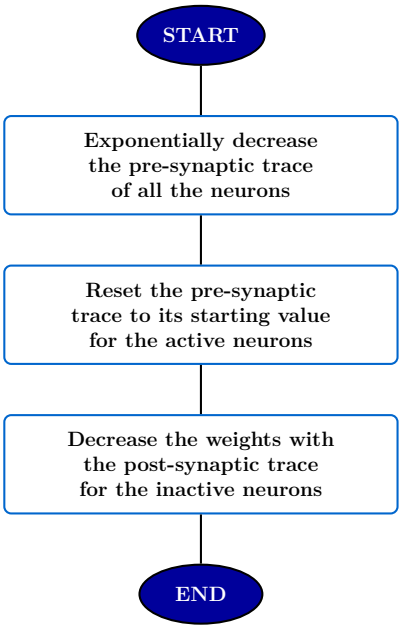


Figure J.17: Function *ltd()*

Normalize the network weights

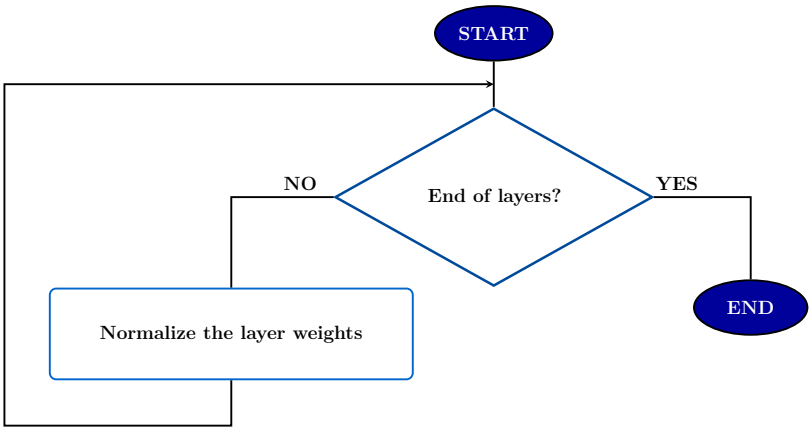


Figure J.18: Function *normalizeNetWeights()*

Normalize the layer weights

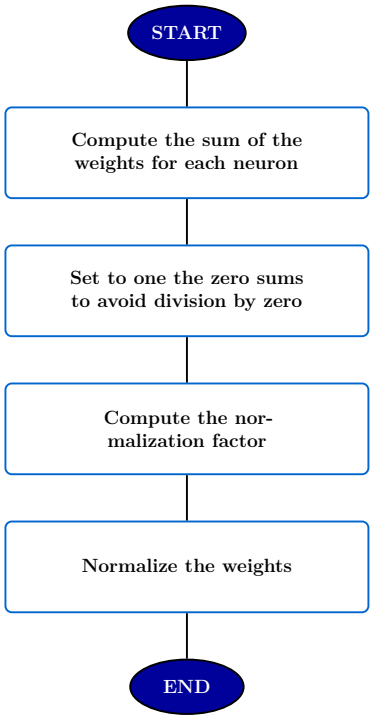


Figure J.19: Function *normalizeLayerWeights()*

Appendix K

Common functions

Initialize the homeostasis parameter

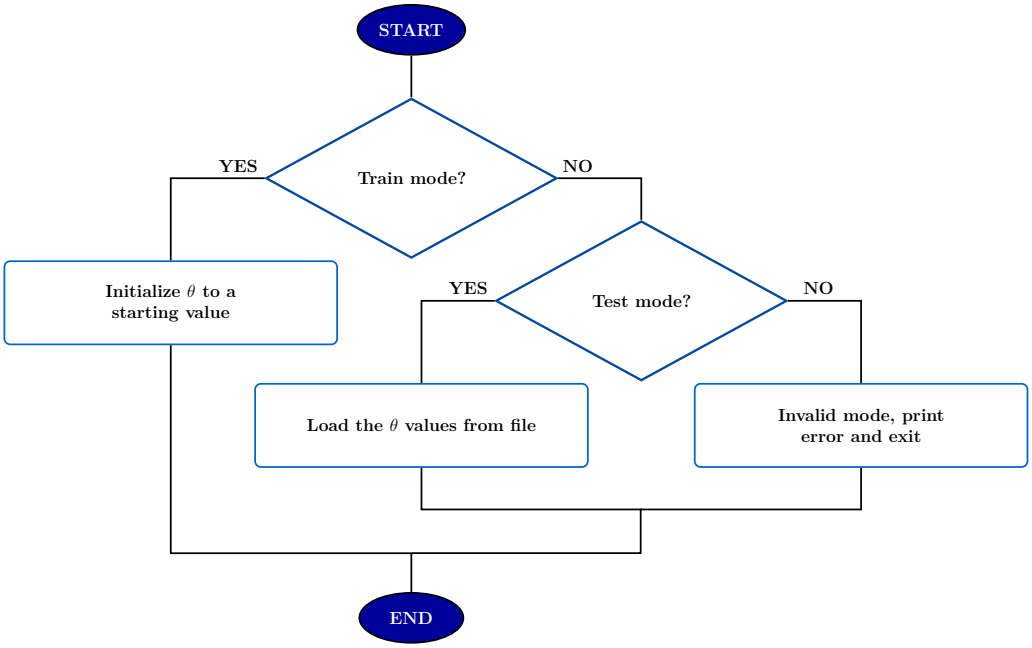


Figure K.1: Function *initializeTheta()*

Initialize the weights

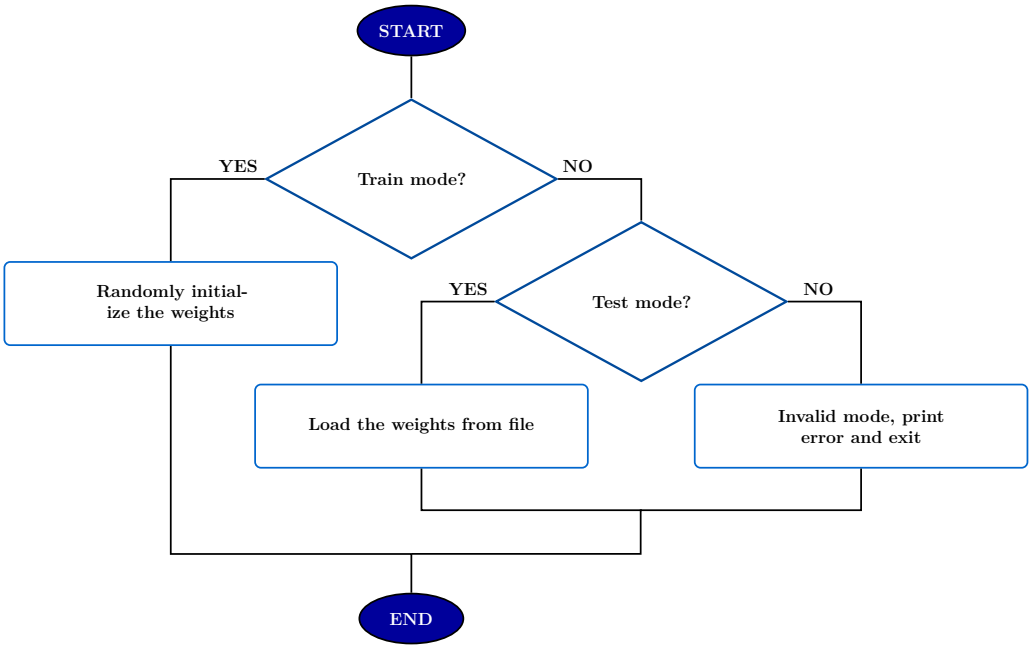


Figure K.2: Function *initializeWeights()*

Appendix L

Architecture

L.1 Neuron complete architecture

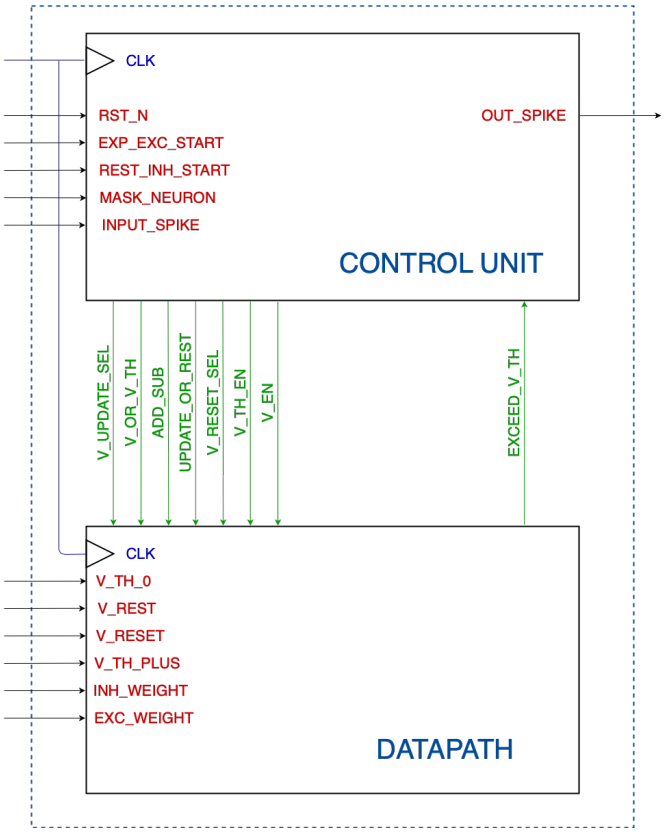


Figure L.1: Complete architecture of the neuron

L.2 Neuron datapath

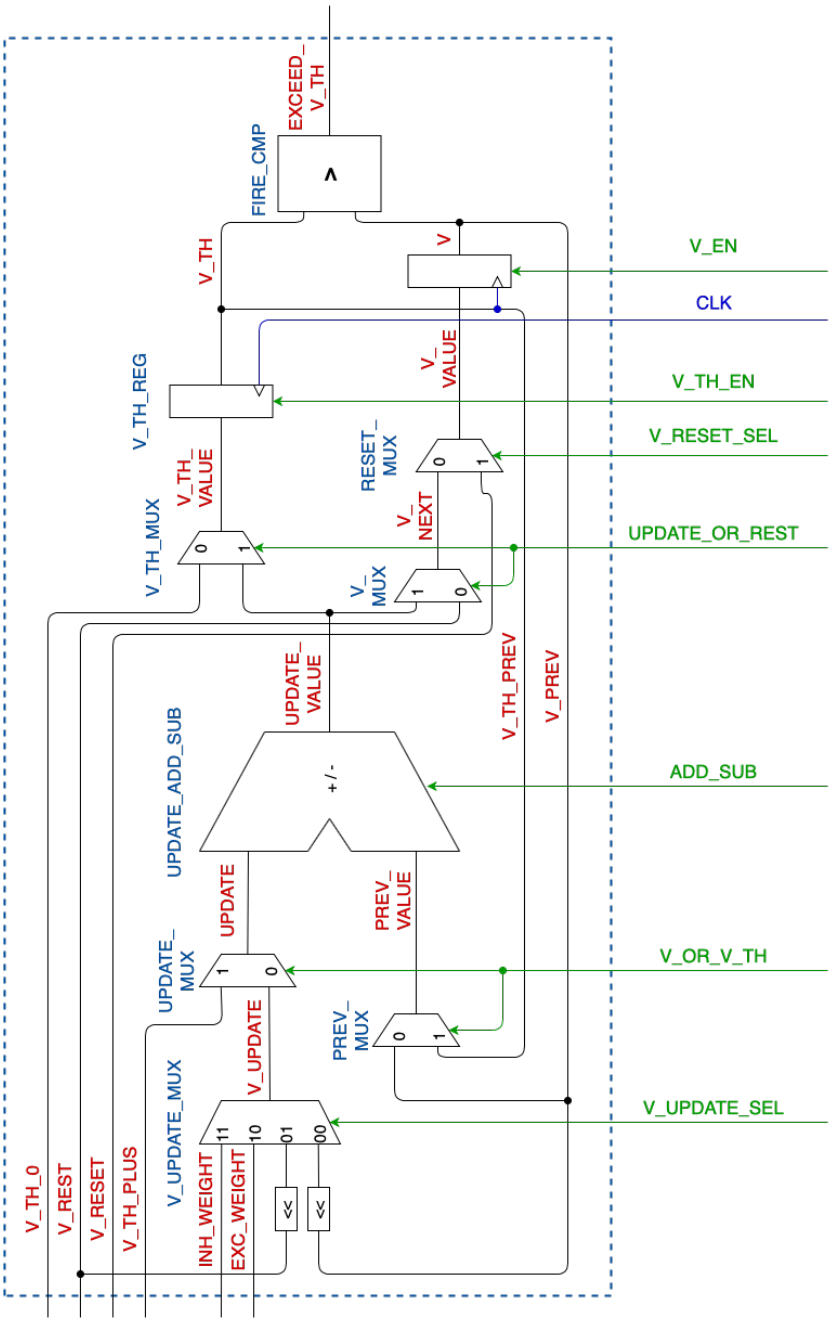


Figure L.2: Datapath of the neuron

L.3 Neuron control unit

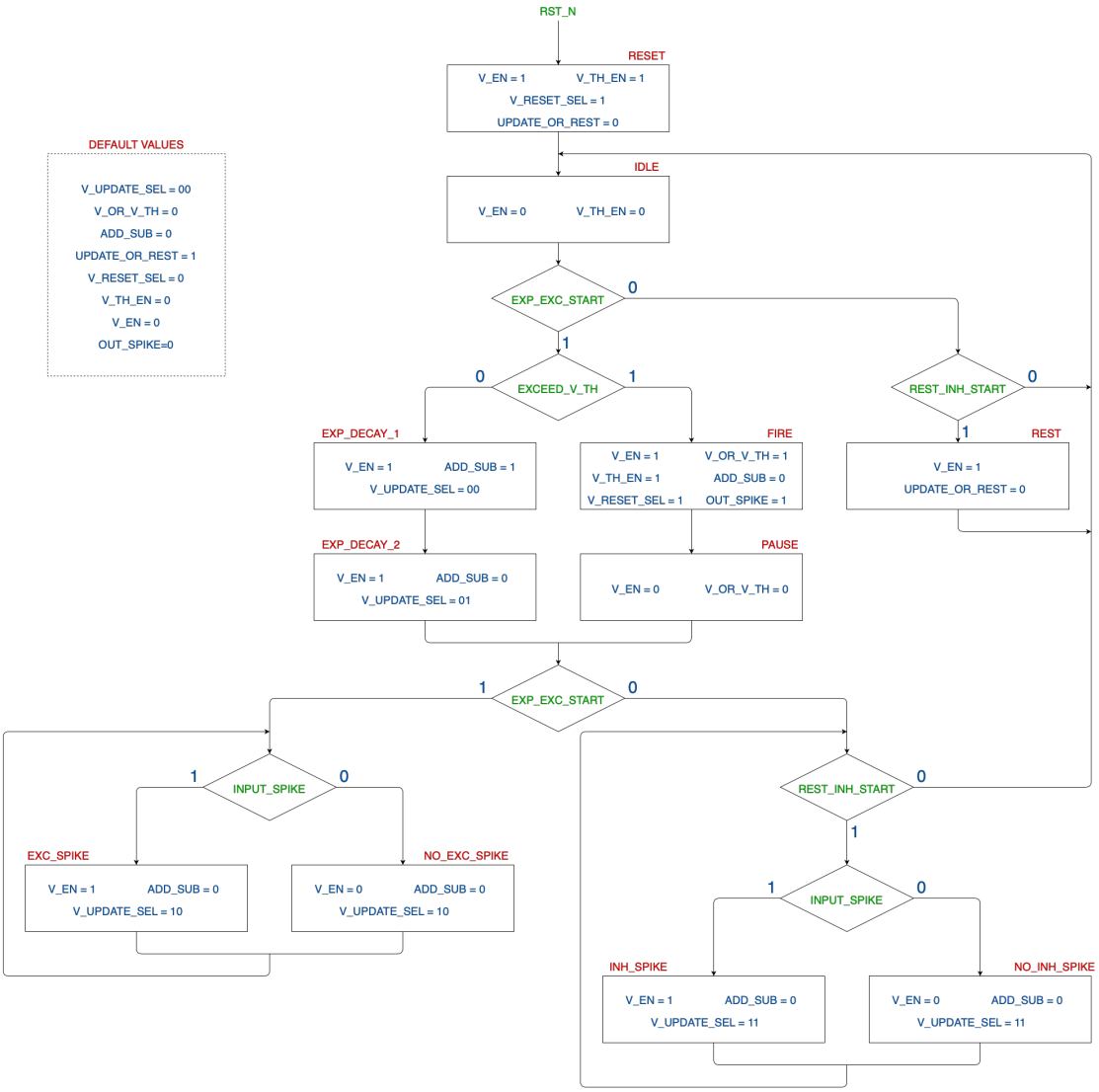


Figure L.3: Neuron ASM chart

L.4 Layer datapath

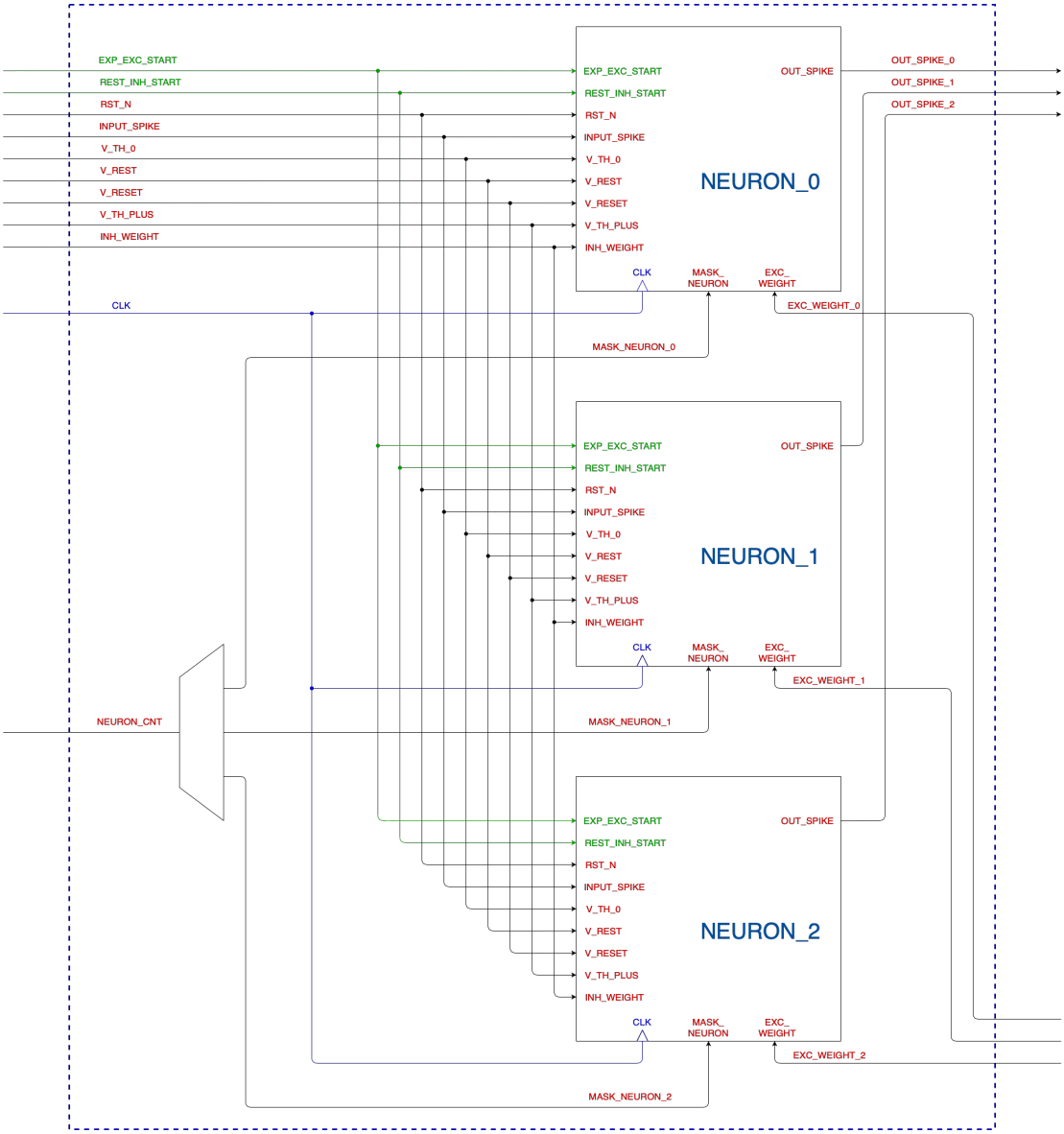


Figure L.4: Datapath of a layer of neurons

L.5 Input selection circuit

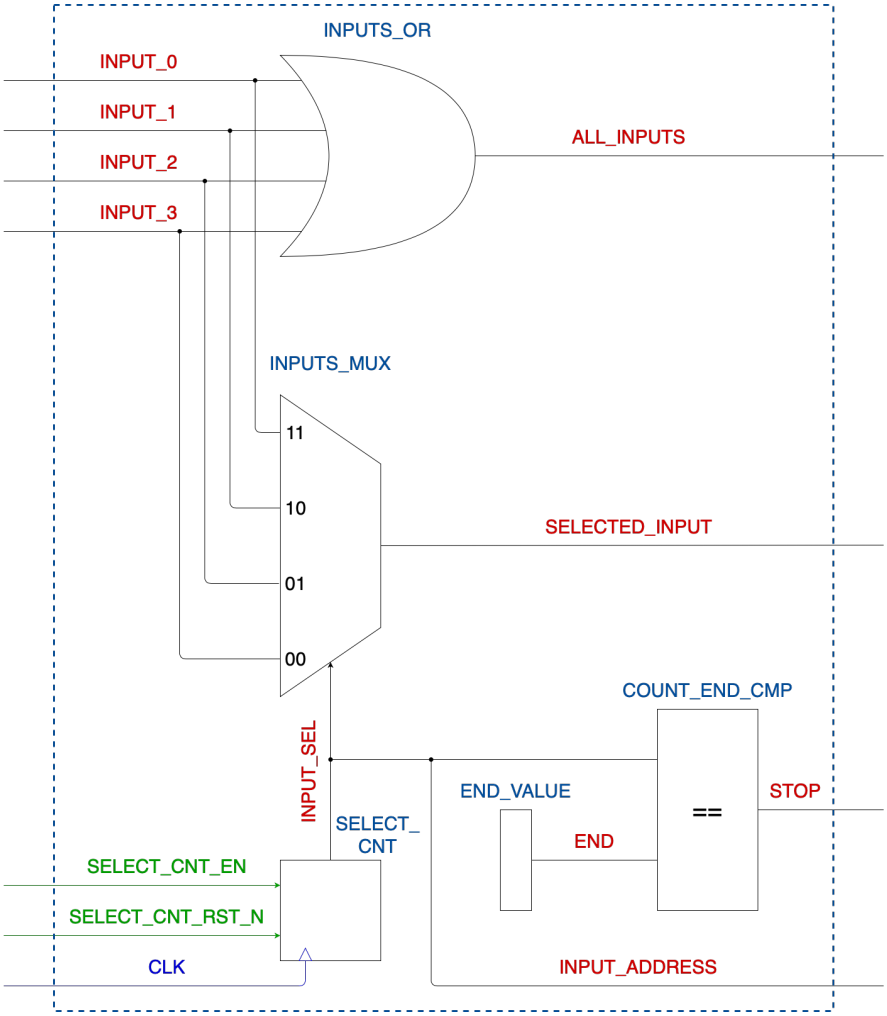


Figure L.5: Input selection circuit

L.6 Complete layer datapath

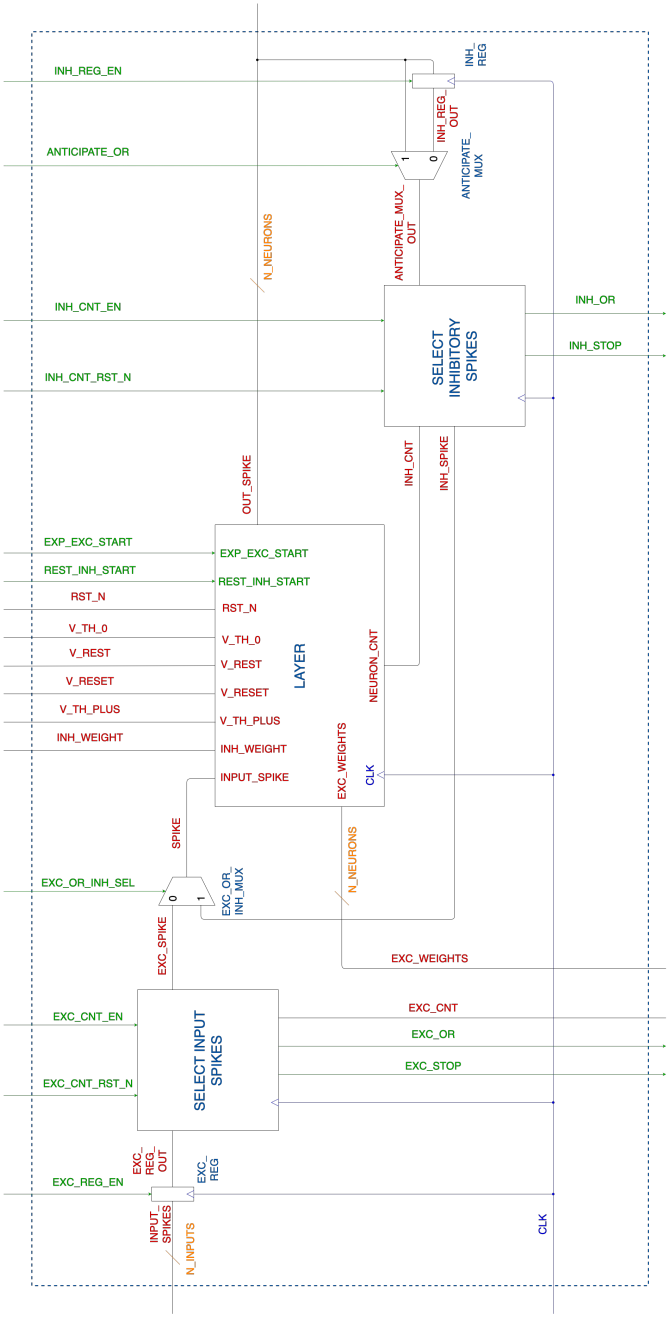


Figure L.6: Complete layer datapath

L.7 Complete architecture of the layer

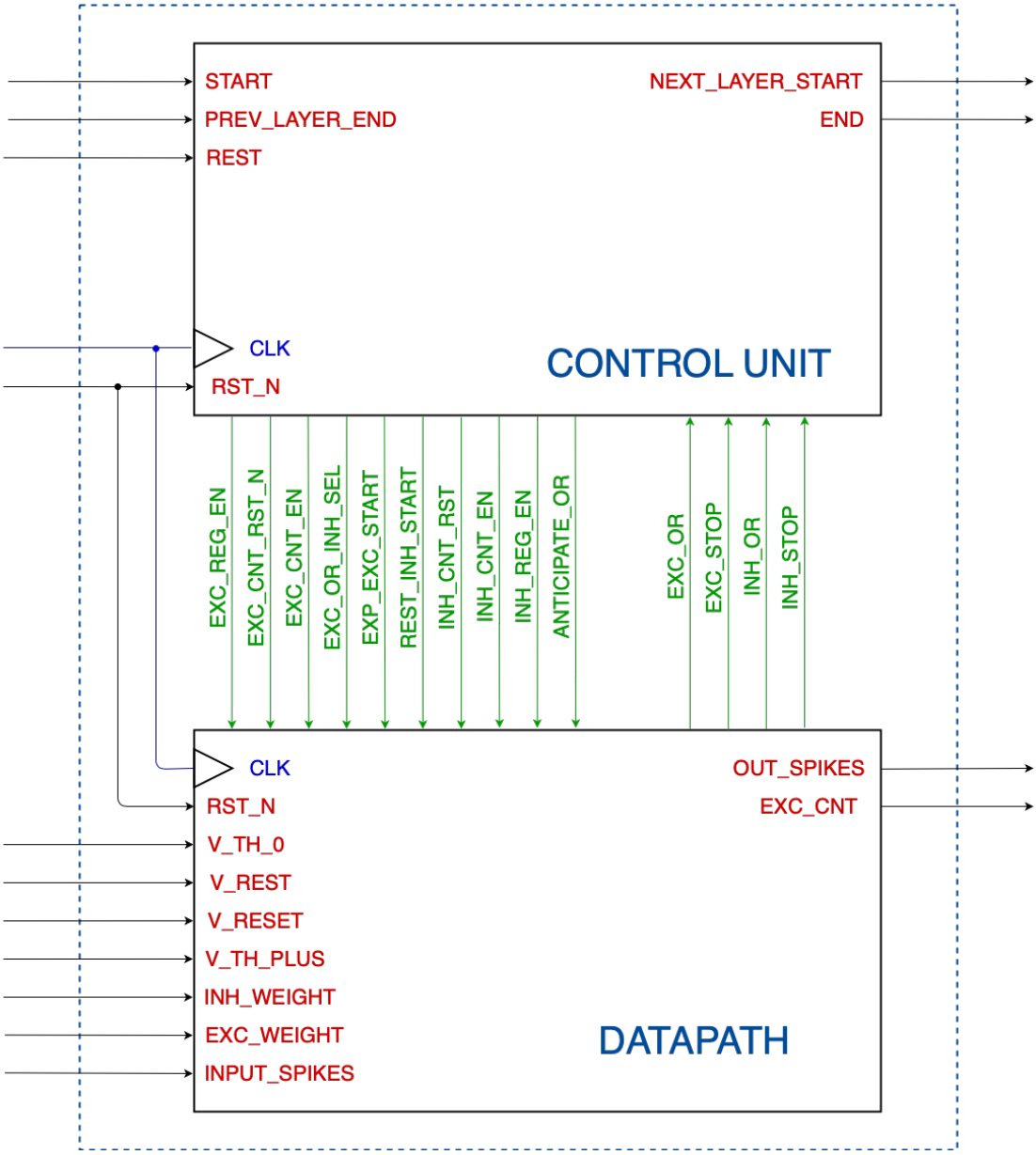


Figure L.7: Complete architecture of the layer

L.8 Layer control unit

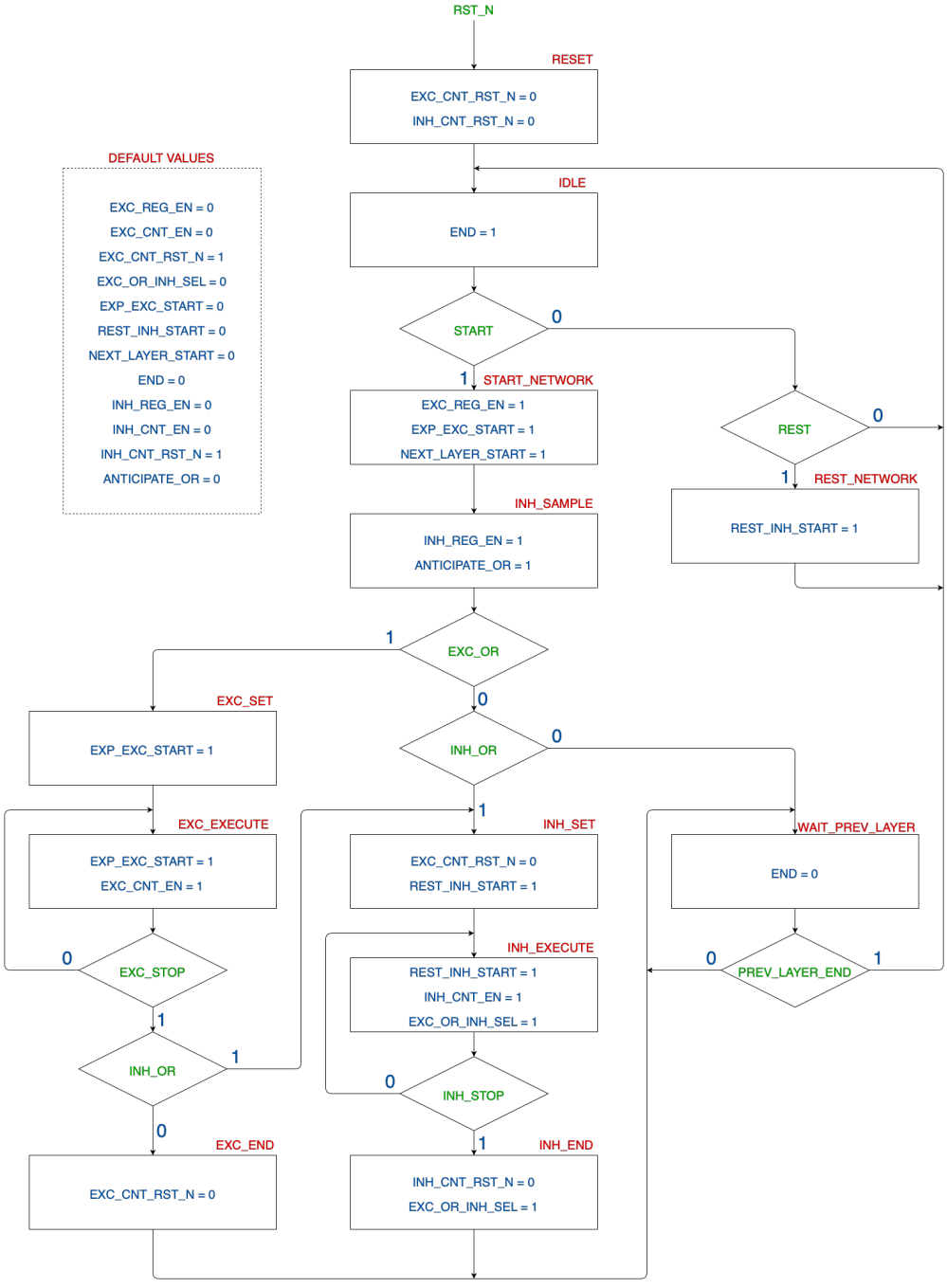


Figure L.8: Layer ASM chart

L.9 Synapse

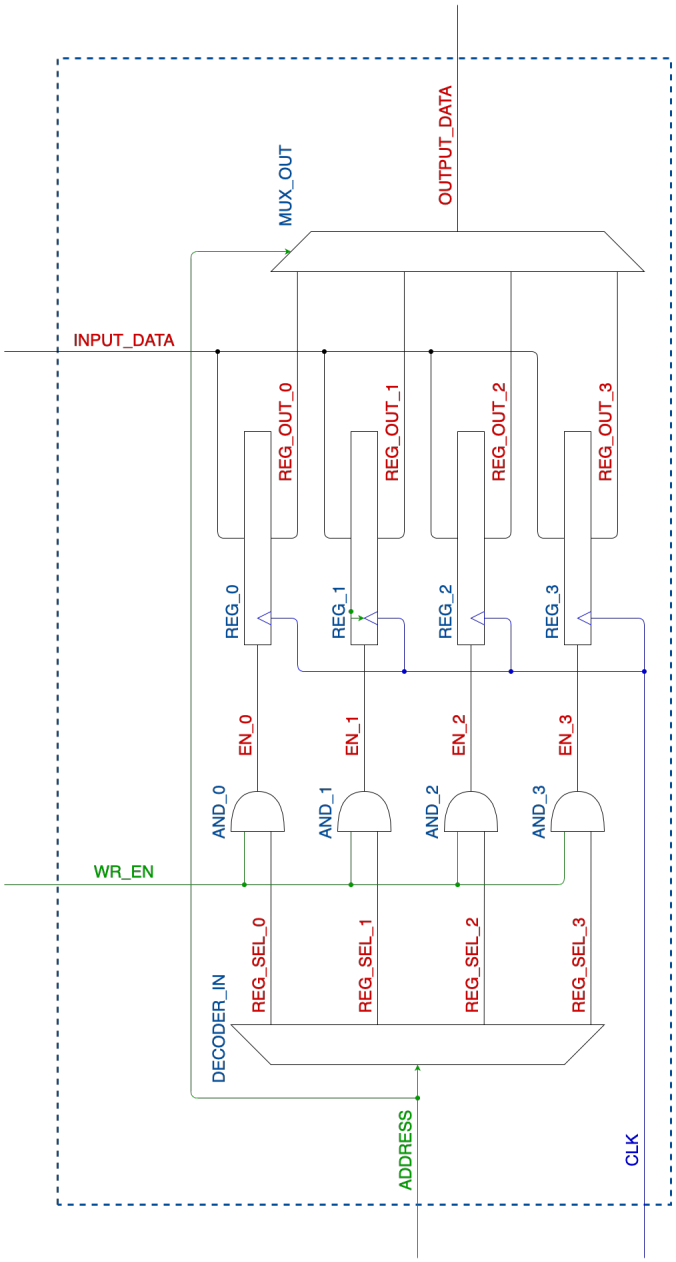


Figure L.9: Synapses

L.10 Synapse layer

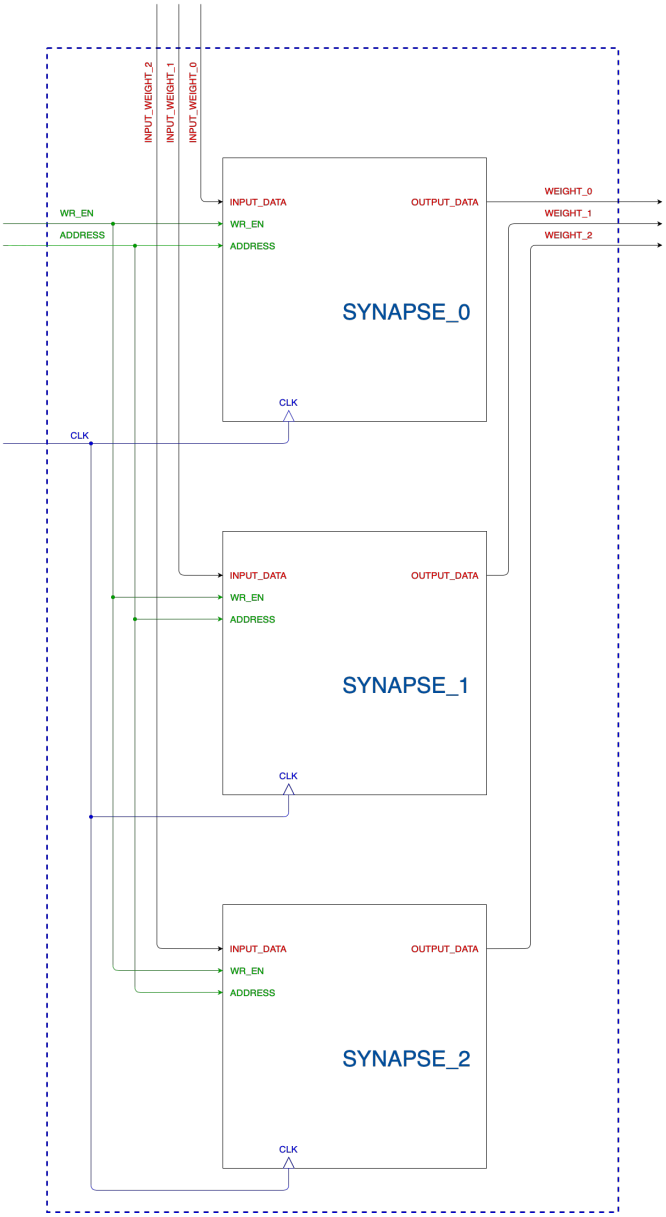


Figure L.10: Synapses layer

L.11 Network

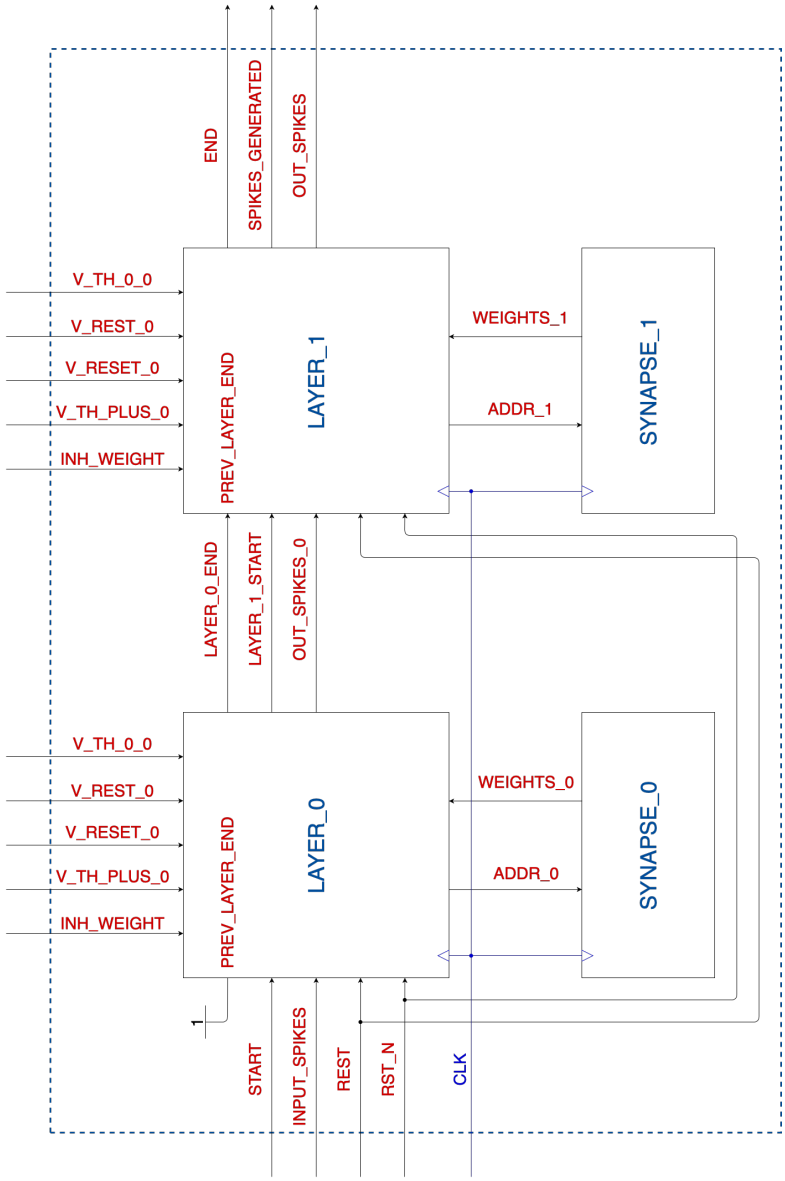


Figure L.11: Network

Bibliography

[1] Wolfgang Maass. «Networks of spiking neurons: The third generation of neural network models». eng. In: *Neural networks* 10.9 (1997), pp. 1659–1671. ISSN: 0893-6080 (cit. on p. 6).

[2] Wikipedia contributors. *Edwin Smith Papyrus — Wikipedia, The Free Encyclopedia*. [Online; accessed 11-July-2021]. 2021. URL: `https://en.wikipedia.org/w/index.php?title=Edwin_Smith_Papyrus&oldid=1027769253` (cit. on p. 9).

[3] Wikipedia contributors. *Luigi Galvani — Wikipedia, The Free Encyclopedia*. [Online; accessed 11-July-2021]. 2021. URL: `https://en.wikipedia.org/w/index.php?title=Luigi_Galvani&oldid=1032853351` (cit. on p. 9).

[4] Wikipedia contributors. *Reticular theory — Wikipedia, The Free Encyclopedia*. [Online; accessed 11-July-2021]. 2020. URL: `https://en.wikipedia.org/w/index.php?title=Reticular_theory&oldid=994544104` (cit. on p. 11).

[5] Focus. *Come funziona il cervello*. June 2017. URL: `https://www.focus.it/scienza/salute/come-funziona-il-cervello` (visited on 06/26/2017) (cit. on p. 12).

[6] Ernst-August Seyfarth. «Julius Bernstein (1839-1917): pioneer neurobiologist and biophysicist». In: *Biological Cybernetics* 94 (Jan. 2006), pp. 2–8 (cit. on p. 14).

[7] Nicolas Brunel and Mark C. W. van Rossum. «Lapicque's 1907 paper: from frogs to integrate-and-fire». In: *Biological Cybernetics* 97 (Jan. 2007), pp. 337–339 (cit. on p. 16).

[8] Louis Lapicque 1907. «Quantitative investigations of electrical nerve excitation treated as polarization». Trans. by Nicolas Brunel and Mark C. W. van Rossum. In: *Biological Cybernetics* 97 (Jan. 2007), pp. 341–349 (cit. on pp. 16, 26, 29).

[9] A. L Hodgkin and A. F Huxley. «A quantitative description of membrane current and its application to conduction and excitation in nerve». eng. In: *The Journal of physiology* 117.4 (1952), pp. 500–544. ISSN: 0022-3751 (cit. on pp. 16, 23).

[10] Wulfram Gerstner Henry Markram and Per Jesper Sjöström. «A history of spike-timing-dependent plasticity». In: *Frontiers of synaptic neuroscience* (Aug. 2011) (cit. on p. 17).

[11] Alexander Bain (1818-1903). *Mind and body*. H.S. King (England), 1873 (cit. on p. 18).

[12] William James (1842-1910). *The princiles of psychology*. Henry Holt and Company, 1890 (cit. on p. 18).

[13] Sir M. Foster. *A textbook of physiology*. Macmillan, 1897. Chap. The central nervous system, by C. S. Sherrington (cit. on p. 18).

[14] C. S. Sherringhton. *On plastic tonus and proprioceptive reflexes*. Jan. 1909 (cit. on p. 18).

[15] Donald O. Hebb. *The organization of behavior*. Elsevier Inc, 1949 (cit. on p. 19).

[16] Jerzy Konorski. *Conditioned reflexes and neuron organization*. Cambridge University Press, 1948 (cit. on pp. 20, 21).

[17] L. H. Haibt N. Rochester J. H. Holland and W. L. Duda. «Test on a cell assembly theory of the action of the brain using a large digital computer». In: (1956) (cit. on p. 20).

[18] F. Rosemblatt. «The perceptron: a probabilistic model for information storage and organization in the brain». In: *Physiological review* 65.6 (1958) (cit. on p. 21).

[19] Leon N. Cooper Elie L. Bienenstock and Paul W. Munro. «Theory for the development of neuron selectivity: orientation specificity and binocular interaction in visual cortex». In: (1981) (cit. on p. 21).

[20] William B. Levy and Oswald Steward. «Synapses as associative memory elements in the hippocampal formation». In: *Brain research* 175.2 (1958), pp. 233–245 (cit. on p. 22).

[21] Greg J. Stuart and Bert Sakmann. «Active propagation of somatic action potentials into neocortical pyramidal cell dendrites». In: *Nature* 367 (Jan. 1994), pp. 69–72 (cit. on p. 22).

[22] P. J. Helm H. Markram and B. sakmann. «Dendritic calcium transients evoked by single back-propagating action potentials in rat neocortical pyramidal neurons». In: *The Journal of Physiology* 485 (1 May 1995), pp. 1–20 (cit. on p. 22).

[23] Michael Frotscher Henry Markram Joachim Lübke and Bert Sakmann. «Regulation of Synaptic Efficacy by Coincidence of Postsynaptic APs and EPSPs». In: *Science* 275 (5297 Jan. 1997), pp. 213–215 (cit. on p. 22).

[24] Archibald Vivian Hill. «Excitation and accommodation in nerve». In: *Proceedings of the Royal Society of London. Series B - Biological Sciences* 119.814 (1936), pp. 305–355. DOI: 10.1098/rspb.1936.0012. eprint: https://royalsocietypublishing.org/doi/pdf/10.1098/rspb.1936.0012. URL: https://royalsocietypublishing.org/doi/abs/10.1098/rspb.1936.0012 (cit. on p. 26).

[25] E.M Izhikevich. «Simple model of spiking neurons». eng. In: *IEEE transactions on neural networks* 14.6 (2003), pp. 1569–1572. ISSN: 1045-9227 (cit. on p. 30).

[26] Eugene M Izhikevich, Tomaso A Poggio, and Terrence J Sejnowski. *Dynamical Systems in Neuroscience: The Geometry of Excitability and Bursting*. eng. Computational neuroscience. Cambridge: MIT Press, 2006. ISBN: 9780262090438 (cit. on p. 30).

[27] Peter U. Diehl and Matthew Cook. «Unsupervised learning of digit recognition using spike-timing dependent plasticity». In: *Frontiers in Computational Neuroscience* 9 (Aug. 2015). DOI: https://doi.org/10.3389/fncom.2015.00099. URL: https://github.com/peter-u-diehl/stdp-mnist (cit. on pp. 36, 62, 99).

[28] David Heeger. «Poisson Model of Spike Generation». In: (Oct. 2000) (cit. on p. 50).

[29] Wikipedia contributors. *Poisson point process — Wikipedia, The Free Encyclopedia*. [Online; accessed 20-September-2021]. 2021. URL: https://en.wikipedia.org/w/index.php?title=Poisson_point_process&oldid%20=1042299082 (cit. on p. 50).

[30] Wikipedia contributors. *Poisson distribution — Wikipedia, The Free Encyclopedia*. [Online; accessed 20-September-2021]. 2021. URL: https://en.wikipedia.org/w/index.php?title=Poisson_distribution&oldid=1044332175 (cit. on p. 50).

[31] Corinna Cortes Yann LeCunn and Christopher J.C Burges. *The MNIST database*. URL: http://yann.lecun.com/exdb/mnist/ (visited on 06/15/2021) (cit. on p. 53).

[32] Cecilia Shao. *Real-time numbers recognition (MNIST) on an iPhone with CoreML from A to Z*. Nov. 2018. URL: https://medium.com/comet-ml/real-time-numbers-recognition-mnist-on-an-iphone-with-coreml-from-a-to-z-283161441f90 (visited on 06/15/2021) (cit. on p. 56).

[33] Romain Brette Marcel Stimberg and Dan Goodman. «Brian 2, an intuitive and efficient neural simulator». In: *eLife* (Aug. 2019). DOI: https://doi.org/10.7554/eLife.47314. URL: https://briansimulator.org (cit. on pp. 60, 67).

[34] Dan Goodman and Romain Brette. «Brian: a simulator for spiking neural networks in Python». In: *Frontiers in Neuroinformatics* 2 (Nov. 2008). DOI: https://doi.org/10.3389/neuro.11.005.2008. URL: https://readthedocs.org/projects/brian/ (cit. on p. 67).

[35] Romain Brette et al. «Simulation of networks of spiking neurons: A review of tools and strategies». eng. In: *Journal of computational neuroscience* 23.3 (2007), pp. 349–398. ISSN: 0929-5313 (cit. on p. 67).

[36] Nicholas T. Carnevale and Michael L. Hines. *The NEURON Book*. Cambridge University Press, 2006. DOI: 10.1017/CBO9780511541612 (cit. on p. 67).

[37] James M. Bower and David Beeman. *The book of GENESIS*. Springer, New York, NY, 1998. DOI: https://doi.org/10.1007/978-1-4612-1634-6 (cit. on p. 67).

[38] M. Gewaltig and M. Diesmann. «NEST (NEural Simulation Tool)». In: *Scholarpedia* 2.4 (2007). revision #130182, p. 1430. DOI: 10.4249/scholarpedia.1430 (cit. on p. 67).

[39] Daniel Neil and Shih-Chii Liu. «Minitaur, an Event-Driven FPGA-Based Spiking Network Accelerator». eng. In: *IEEE transactions on very large scale integration (VLSI) systems* 22.12 (2014), pp. 2621–2628 (cit. on p. 71).

[40] Hajar Asgari, Babak Mazloom-Nezhad Maybodi, Raphaela Kreiser, and Yulia Sandamirskaya. «Digital Multiplier-Less Spiking Neural Network Architecture of Reinforcement Learning in a Context-Dependent Task». eng. In: *IEEE journal on emerging and selected topics in circuits and systems* 10.4 (2020), pp. 498–511 (cit. on p. 71).

[41] Sixu Li, Zhaomin Zhang, Ruixin Mao, Jianbiao Xiao, Liang Chang, and Jun Zhou. «A Fast and Energy-Efficient SNN Processor With Adaptive Clock/Event-Driven Computation Scheme and Online Learning». eng. In: *IEEE transactions on circuits and systems. I, Regular papers* 68.4 (2021), pp. 1543–1552 (cit. on p. 72).