ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
GRADO EN INGENIERÍA INFORMÁTICA

GRAMÁTICA DE GRAFOS PARA COMPORTAMIENTO COMPLEJO

GRAPH GRAMMARS FOR COMPLEX BEHAVIOR

Realizado por
Pablo Andrés Martínez
Tutorizado por
Francisco J. Vico Vela
Departamento
Lenguajes y Ciencias de la Computación

UNIVERSIDAD DE MÁLAGA
MÁLAGA, JUNIO 2016

Fecha defensa:
El Secretario del Tribunal

Resumen

Actividad de fondo (background activity) es el nombre dado al fenómeno biológico por el que el cerebro de un organismo vivo nunca está en completo reposo. La comunidad neurocientífica lo relaciona con funciones cognitivas como la memoria y la exploración de experiencias sensoriales previas.

Las redes neuronales artificiales surgieron originalmente como modelo del sistema nervioso. En la informática, han sido especialmente utilizadas para la aproximación de funciones y reconocimiento de patrones. No obstante, la dinámica y topología de los paradigmas típicamente utilizados no son adecuados para modelar procesos como la actividad de fondo. Cuando se pretende replicar comportamientos de redes neuronales reales, el modelo más apropiado es el denominado "red neuronal de disparos" (*spiking neural network*, SNN), cuyos elementos siguen una dinámica más fiel a las neuronas biológicas.

Nuestro objetivo es desarrollar un algoritmo que genere topologías de redes SNN capaces de mantener actividad de fondo. La topología de una SNN es descrita mediante un grafo, por lo que el primer aporte de este trabajo es un formalismo gramatical para la generación de éstos. Dicho formalismo es empleado por un proceso automático de búsqueda para encontrar SNN que mantengan actividad de fondo. La búsqueda es realizada por un algoritmo evolutivo, que aplica sucesivas transformaciones a una población de SNN, haciéndolas gradualmente más aptas para cumplir el objetivo propuesto.

Dado que las distintas SNN de la población son independientes entre sí, el tiempo de ejecución del algoritmo puede ser notablemente menor al paralelizarse. Los resultados de este trabajo se han obtenido en el nodo de la Universidad de Málaga de la Red Española de Supercomputación, utilizando 40 núcleos para dichas ejecuciones. En la práctica, el tiempo de simulación se reduce un orden de magnitud respecto a un procesador de cuatro núcleos convencional, mejorando sustancialmente la interacción en el proceso de obtención y estudio de los resultados.

Palabras clave: actividad de fondo, gramática, grafo, red neuronal de disparos, algoritmo evolutivo, computación bioinspirada.

Abstract

Background activity is the biological phenomenon that prevents the brain of an alive organism from reaching a state of complete inactivity. The neuroscientist community claims that it is related to cognitive functions such as memory and the exploration of previously sensed experiences.

Artificial neural networks were originally developed as a nervous system model. In Computer Science, they have been applied in function approximation and pattern recognition problems. However, dynamics of the typically used paradigms are not appropriate for the replication of processes such as background activity. When the goal is to reproduce the behavior of real neural networks, the most adequate model is the Spiking Neural Network (SNN), whose elements closely resemble the biological neurons.

Our objective is to develop an algorithm that generates SNN topologies able to maintain background activity. The topology of an SNN is described as a graph, thus, the first contribution of this project is a grammar formalism to generate them. That formalism is applied by an automated search process in order to find SNNs that are able to maintain background activity. This search is done by an evolutionary algorithm, which develops a population of SNNs and applies successive transformations to them, gradually increasing their ability to fulfill the proposed objective.

Considering that the different SNNs of the population are independent of each other, the time required to execute the algorithm can be noticeably reduced when using parallel computation. In order to obtain the results discussed in this document, the program was run over 40 cores of the local supercomputing node, which is part of the Spanish Supercomputing Network. The resulting execution time is decreased in an order of magnitude compared to the one that would be required in a quad-core personal computer. This was crucial for the development of the project, as it considerably improved our ability to manage the process of obtaining and studying the results.

Keywords: background activity, grammars, graphs, spiking neural network, evolutionary algorithm, bioinspired computation.

Contents

Chapter 1

Introduction

1.1 On the interest of background activity

In recent years, the phenomenon of background activity has drawn an increasing amount of attention. It is a continuous and spontaneous activity, present in the cerebral cortex even when the organism is sleeping or anesthetized.

As explained in a doctoral thesis focused in this phenomenon (Seamari, 2016), its existence was verified during the 1930s when the electroencephalography started to develop. However, at first it was interpreted as 'neuronal noise', i.e. activity that does not imply relevant information or that is derived from physiological processes, such as breathing and heart beating. In the last decades, this point of view has been changed, as it was confirmed that this ever-present spontaneous fluctuations were relevant on some computational functions of the brain. Examples of those functions are the exploration of previous sensorial experiences and the ability to create new memories.

Neuroscientist are interested in knowing more about this phenomenon: what causes it, which are the conditions needed for it to happen and how it interacts with other brain processes. This study could yield important advancements on the way we understand how the brain works. That, in turn, could help to detect malfunctions on brain activity and be relevant in solving mental health issues or problems related to brain physiology.

However, analysis on real specimens is limited. The information that is acquired from an electroencephalography can not tell anything about individual neurons. Other methods require to insert an electrode in the brain through surgery. None of them give decisive information on how neurons interact with each other either. Therefore, much

of the research done in the field has to be based on hypothesizing possible interactions and constraints, to then check if the expected results match with the ones obtained experimentally.

Computer Science is ideal to aid in this process. A program can reproduce possible models of nervous systems and simulate them to verify their properties. Thanks to it, scientists can obtain the theoretical result of their hypothesis through an automated process, easing the scientific labour.

1.2 The objective of this project

This project aimed to develop an algorithm capable of generating artificial neural networks that, after receiving an initial stimulation, were able to maintain a long lasting activity. This behavior is meant to be an approximation of biological background activity. Although being simpler, it shares a fundamental characteristic: it is auto-maintained, needing not any kind of input (once started) to persist over time.

The approach taken to fulfil this goal comprised two main milestones. The first one was to formalize a method of generating graphs that was scalable and easy to manage by a search algorithm. Those graphs were then interpreted as the topology of an artificial neural network. The concept of formal grammars was applied here, defining a grammar that generates strings capable of representing any graph. The second goal was to implement an evolutionary algorithm tasked with the search of a network able to maintain background activity.

When both milestones were achieved, we run the algorithm to check if the desired results were met. To do so, we had access to a supercomputation node to execute the most demanding simulations. The obtained results were then analyzed to infer the properties of the networks designed by the algorithm.

This whole approach to meet our objective and analyze what we obtained was oriented in a scientific fashion. This, in itself, was an important goal of the project, being an experience on how research is managed. Applying the knowledge gained through the degree and acquiring new capabilities were part of the academic experience that this project represents.

1.3 State of the art in artificial neural networks

Originally, artificial neural networks were developed as an imitation of the biological nervous system. However, due to the computational complexity associated to simulate thousands of neurons interacting with each other, the model was extremely simplified. This model is widely known and applied in computer engineering, thus we will refer to it as the 'classical' neural network model.

It ignores the complex dynamics of biological neurons, defining them as a computation machine that always yields the same output for a given set of inputs, and does it instantaneously. These groups of neurons interact with each other through the connections defined between them, each one having an associated weight. This weight determines the ability of a given neuron of the network to activate its connected partners.

The computation that each artificial neuron executes is mathematically represented (Eq. 1.1) as adding up all the inputs it receives and applying a function to this value. Those inputs are weighted differently depending on the connection it was received from. Usually, an additional parameter is given to the neuron model, which represents the ease that it shows to be activated. This parameter is known as *threshold*. Figure 1.1 illustrates the usual representation of this kind of neurons and the symbols associated to them.

$$f(\sum_{i=1}^{n}(w_i x_i) - \theta) \tag{1.1}$$

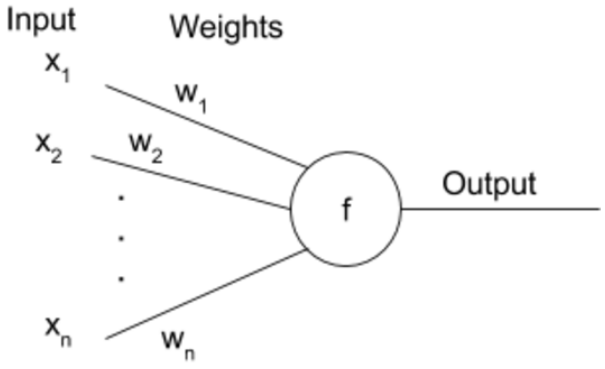

FIGURE 1.1: The basic artificial neuron model

In this model, the parameters that define the network are mainly the *matrix of weights* and the *function* that yields the output. The weights of the connections are usually obtained through a *learning algorithm*, which finds the best values to make the network

match a specific complex function. This model has been very successful at solving some demanding problems in computer engineering, specifically those related to pattern recognition and function approximation.

However, this model has little application in the field of neuroscience. It is useful to execute certain computations, but has important limitations when simulating the behavior of a biological nervous system. This is due to real networks not yielding an output only dependent to the input received at a certain instant.

However, the model just explained is not the only existing one. In (McCulloch & Pitts, 1943) it is proposed a neural computation model prototype that inspired two distinct approaches of artificial neuron research. One of them developed into what is the current state of classic neural network models. The other one was focused on developing models similar to biological nervous systems, expecting to fulfill the goal of simulating biological processes.

That second approach yielded what is known as Spiking Neural Network models (SNN). Its main characteristic is that each neuron has a variable, the *membrane potential*, that changes over time. The neuron is activated if it reaches a certain value of *membrane potential*, which in turn is dependent on the amount of stimuli that it has received throughout an extended period of time. This approach is much more appreciated by the neuroscience community as it is a more accurate option to mimic the behavior of a real nervous system.

The most celebrated SNN model is the one based on the interpretation of a biological neuron proposed in (Hodgkin & Huxley, 1952). It considers the neuron as an electrical circuit. This circuit charges a capacitor and, once it reaches a certain value of stored energy, the circuit changes its polarization and all the charge escapes the capacitor, yielding an almost instantaneous output of energy spike. This model can be mathematically formalized through a system of differential equations. These equations are not trivial to solve and, thus, computationally time consuming. This is the main reason why SNN models have not been subject to the same amount of research that has been invested in other neural network models.

In this project, we focus on a more simplified Spiking Neural Network model. Specifically, we are using an *Integrate & Fire* model of SNN, defining discrete time-steps. The specifications of the model are explained in detail in a dedicated chapter of this document.

In the proposed model of this project, the dynamic of all neurons is exactly the same, and the constants that define it match those that commonly happen in the biological

nervous system. Besides, all connections have the same weight associated to them. Therefore, the only parameter that changes from a network to another is the way its neurons are connected.

1.4 Evolutionary algorithm as a searching method

Evolutionary algorithms are optimization search algorithms that follow the principles of the theory of evolution developed by Charles Darwin. Their approach to solve a given problem is based on iteratively changing a set of possible solutions (called *population*) and discard those individuals (i.e. possible solutions) that are less fitted.

To implement an evolutionary algorithm, three things are needed: a formal expression of the individuals (known as the *genome*), the set of changes that can be applied to them (i.e. the *mutation rules*) and a method to test how promising the individuals are (the *fitness function*). All of them must be carefully chosen. The genome and mutation rules must have some properties that allow the algorithm to reach more fitted individuals after each iteration. The fitness function must be representative enough to be able to determine the best individual in a group of similar ones.

The main characteristic of this approach is that the search is not driven by a powerful heuristic that, at each point, chooses which change to apply. Instead, the changes are made first, randomly chosen from a set of mutation rules. Then, the fitness function tests the individuals by trying to solve the problem with them, and assigns a score to each one depending on how well they did. The individuals with a better score tend to be represented in the population of the next iteration, while the poorest ones disappear. Thus, the whole population gets gradually better at solving the problem, hopefully reaching the solution at some point.

The developer of the algorithm does not need to formally analyze the search space, avoiding the task of finding mathematical patterns to build up a criteria to guide the search. In the evolutionary approach, we have to focus on defining a scale to measure the performance of the individuals at the simulations and to develop the set of mutation rules able to gradually increase that performance. Those objectives are demanding, but when the optimization search has to be done in an environment where no mathematical model can be found, there is no other option. This type of methods that do not require the designer to completely understand how the solution should be are called *Soft Computing*.

Given that the objective of this project is to provide a tool to help in understanding the conditions and implications of background activity, we have no information about how to search neural networks that show it. This turns into a natural decision of choosing to apply evolutionary algorithms for our application.

Another relevant characteristic of evolutionary algorithms is that they are intrinsically parallel. This is due to the fact that they maintain a population of independent individuals and mutate and test them separately. Then, although the searching method is based on trial and error, fair efficiency can be obtained through the usage of parallel computing.

1.5 Structure of the document

Here, the content of each chapter is briefly described:

- The first chapter, will be dedicated to our first contribution to the field: a method to define a graph using a string of symbols. This representation differs from the usual mathematical expression of a graph, showing some relevant properties that will be discussed throughout this chapter.
- The second chapter is focused on defining the Spiking Neural Network model applied in this project. This model is based on a discrete time scale simulation that follows an *Integrate & Fire dynamic*.
- The third chapter is dedicated to the explanation of the evolutionary algorithm. In it, the set of mutation rules and fitness function designed are defined. Those and the genome expression that was described in the first chapter are analyzed and their properties are discussed. Additionally, a last section is dedicated to the brief explanation of other approaches that were considered throughout the project, justifying why they were discarded.
- The fourth chapter shows the results that were obtained by applying the model that has been defined. It provides an in-depth analysis of them, showing that the objective was met.
- Finally, the conclusion chapter offers an overview of the project, the analysis of its relevance and the learning that implied developing it. It also proposes further study to continue researching on the field opened by this project.

Additionally, two appendices are included. One of them focused on the software that was used at the implementation phase of the model. The other one dedicated to the infrastructure where the simulations were run, i.e. the supercomputing node.

Chapter 2

A grammatical formalism for graph generation

Formal grammars are an important tool applied in many aspects of Computer Science. Their main goal is to define a syntax, i.e. how symbols can be combined to create strings that satisfy certain properties. For example, all programming languages have an associated grammar that defines how valid programs must be written. The set of well-formed formulas in any logic are also defined through grammars and, in general, any representation that depends on arranging symbols in a certain format has its own grammar. The whole set of strings that follow a given grammar is known as the *language generated* by it.

Grammars are defined as a tuple (N, T, P, S) where each element corresponds to:

- *Non-terminal symbols* (N): A set of symbols that help in defining the grammar but will never appear in a string of the generated language.
- *Terminal symbols* (T): The set of all symbols that can appear in the strings of the generated language.
- *Derivation rules* (P): The set of rules that determine how to generate the language. This rules are defined as $A \rightarrow B$, where A and B are strings made up from non-terminal and terminal symbols. It means that if A is found inside a string then A can be removed, placing B instead.
- *Axiom* (S): A special non-terminal symbol.

If a sequence of derivation rules can transform the axiom into a string of terminal symbols, then the obtained string is an element of the language generated by the grammar.

Defining a grammar is one of the different options to formalize a language of strings. Others, such as explicitly defining the set or designing language recognition automatas have their own advantages. In the case of grammars, their main benefit is that the strings in the language are defined recursively, through the repetitive application of the same derivation rules.

That characteristic is useful when the strings are going to be given a specific meaning (i.e. a semantic is defined over the syntax). Then, the algorithm that decodes these strings to obtain the object represented by them (e.g. the program object out of the source code) is easily designed. It is only needed to define the changes that each of the derivation rules apply to the represented object.

We will use this formalism to define a language which strings are then interpreted as a representation of a graph. These strings will be used as the genomes for the evolutionary algorithm already mentioned in the introduction chapter. Thus, we must be sure that all strings follow a specific format. Then, an automated process is needed to obtain the graph represented by each one of them, which is designed taking advantage of the grammar formalism.

2.1 Brief overview on graphs

A graph defines how a set of elements, known as vertices, are connected to each other. Those connections are known as edges. Therefore, a graph G is a set of vertices V and edges $E = V \times V$, represented as $G = (V, E)$.

If the set of edges verifies that $\forall(u, v) \in E \Rightarrow (v, u) \in E$, then it is said that the corresponding graph is *undirected*. On the other hand, if the property is not verified, the associated *directed graph* (usually referred to as *digraph*) can define the topology of a system where the connection between its nodes work just in one direction.

An interesting subset of graphs are *tree graphs*. Those are acyclic undirected graphs, meaning that there is only one path to traverse it from a vertex to another. In computer engineering, trees are widely applied given their desirable characteristics to describe hierarchies. This hierarchy is defined by choosing a vertex, which will be the only element in the highest hierarchy level, known as the root vertex. Then, the rest of the

vertices are ordered depending on the length of the path from each one of them to the root.

2.2 Grammar for tree generation

Given *trees* are a simpler version of graphs, there are already many efficient ways to represent and generate them. We will introduce one of them, as it is the formalism we started with and extended in order to be able to generate any graph.

Any tree can be expressed as a sequence of *balanced brackets*. The language of those strings can be generated by the following grammar:

$$\begin{aligned} N &= \{A, T\} \\ T &= \{[,]\} \\ P &= \{A \to [T] \\ &\qquad T \to TT \\ &\qquad T \to [T] \\ &\qquad T \to [\,]\} \\ S &= A \end{aligned} \tag{2.1}$$

The result of any sequence of derivation rules yields a balanced bracket string:

$$A \to [T] \to [TT] \to^2 [[T][T]] \to^2 [[[\,]][TT] \to^2 [[[\,]][[\,][\,]]] \tag{2.2}$$

A tree is built from a string of balanced brackets by creating a vertex for each pair of brackets. The process of obtaining it is illustrated in Figure 2.1 and defined case by case following the derivation rules:

- Given $A \to [T]$, all strings will be enclosed between a pair of brackets. This rule creates the root vertex and shows it is connected to a group of subtrees.
- $T \to TT$ indicates that another subtree is created. The root of both subtrees is connected to the same vertex of a higher hierarchy (its *parent*).

- $T \rightarrow [T]$ creates a root vertex for the subtree. That vertex will be connected to the corresponding subtrees generated by the bracketed non-terminal symbol.
- $T \rightarrow [\,]$ creates another vertex. This vertex will not be connected to any other subtree (it is a *leaf*).

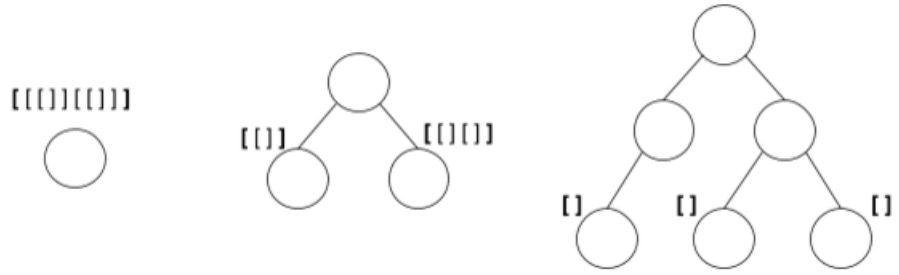

FIGURE 2.1: Building a tree from a balanced bracket string

2.3 Grammar for graph generation

As it was already explained, trees are acyclic graphs, meaning that there is only one path to traverse it from one vertex to another. On the other hand, generic graphs can have an arbitrary amount of paths between any pair of vertices. Therefore, any graph can be defined by set of trees that overall covers each and every of the paths in it. This is known as *graph partitioning* into trees and is the core idea of our proposed formalism.

To define a graph from a set of trees is necessary that they have some vertex in common. Otherwise, the graph would just be a disconnected set of trees. This means that a labelling system is necessary in order to define which vertices are shared by different trees. The following grammar is similar to the one introduced before, but adding the ability to generate those labels and more than one tree in a single sequence:

Definition 2.1: Grammar for graph expressions

It is defined as $G = (N, T, P, S)$:

$$
\begin{aligned}
N &= \{E, L\} \\
T &= \{[,], |\} \\
P &= \{E \to EE \\
&\qquad E \to [LE] \\
&\qquad E \to \varepsilon \\
&\qquad L \to |L \\
&\qquad L \to \varepsilon\} \\
S &= E
\end{aligned}
\tag{2.3}
$$

Where the ε symbol represents the empty string

□

Definition 2.2: GE Language

The language generated by the grammar formalized in Definition 1.1. will be referred to as the *Graph Expression Language*.

□

Given the set of strings GE, the next step is to define the functions needed to obtain the graph represented by it. This functions are defined recursively, covering all cases of the grammar derivation rules in order to ensure all possible strings are taken into account.

Definition 2.3: Labels function n

This function associates a natural number to each sequence of vertical bars, $n : \{|\}^* \to \mathbb{N}$

$$
n(s) = \begin{cases} n(s') + 1 & if \quad s = |s' \\ 0 & if \quad s = \varepsilon \end{cases}
\tag{2.4}
$$

□

Definition 2.4: Root labels function l

The function $l : GE \to 2^{\mathbb{N}}$ returns the set of root vertex of the different trees in the given expression $w \in GE$

$$l(w) = \begin{cases} l(v) \cup l(u) & if \quad w = uv \\ \{n(s)\} & if \quad w = [sv] \\ \varnothing & if \quad w = \varepsilon \end{cases} \tag{2.5}$$

Where $u, v \in GE, s \in \{|\}^*$

□

An example of the application of this function should help in understanding it:

$$l([\,|\,[\,||\,]][\,|||\,[\,||||\,[\,||\,]]]) = l([\,|\,[\,||\,]]) \;\cup\; l([\,|||\,[\,||||\,[\,||\,]]]) = \{1\} \;\cup\; \{3\} = \{1, 3\} \tag{2.6}$$

Definition 2.5: Graph function g

Given the universal set of digraphs U, we define the function $g : GE \to U$ that associates each $w \in GE$ with its corresponding graph.

$$g(w) = \begin{cases} g(v) \cup g(u) & if \quad w = vu \\ g(v) \cup (\{n(s)\} \cup l(v), \{(n(s), m) \mid m \in l(v)\}) & if \quad w = [sv] \\ (\varnothing, \varnothing) & if \quad w = \varepsilon \end{cases} \tag{2.7}$$

Where $u, v \in GE, s \in \{|\}^*$ and the union operation applied is the union defined over graphs: $(V, E) \cup (V', E') = (V \cup V', E \cup E')$

□

Then, again an example is convenient to show how the graph is built. The graph generated is illustrated in Figure 2.2:

$$
\begin{aligned}
&g([\,|\,[\,||\,][\,|||\,]][\,||||\,[\,|||\,[\,||\,]]]) = g([\,|\,[\,||\,][\,|||\,]]) \cup g([\,||||\,[\,|||\,[\,||\,]]]) = \\
&\quad = g([\,||\,][\,|||\,]) \cup (\{1,2,3\},\{(1,2),(1,3)\}) \cup g([\,|||\,[\,||\,]]) \cup (\{3,4\},\{(4,3)\}) = \\
&\quad = g([\,||\,]) \cup g([\,|||\,]) \cup (\{1,2,3,4\},\{(1,2),(1,3),(4,3)\}) \cup (\{2,3\},\{(3,2)\}) = \\
&\quad = (\{2\},\varnothing\}) \cup (\{3\},\varnothing\}) \cup (\{1,2,3,4\},\{(1,2),(1,3),(4,3),(3,2)\}) = \\
&\quad = (\{1,2,3,4\},\{(1,2),(1,3),(4,3),(3,2)\})
\end{aligned}
\tag{2.8}
$$

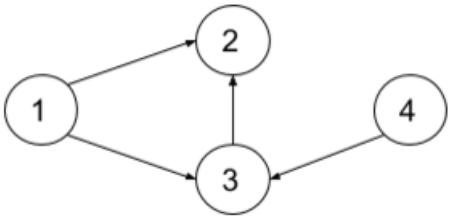

FIGURE 2.2: Graph represented by the string $[\,|\,[\,||\,][\,|||\,]][\,||||\,[\,|||\,[\,||\,]]]$.

It must be noted that building a digraph from a set of trees has an additional advantage. As trees define certain hierarchy between its vertices, it can be (and is) applied as the criteria to decide the direction of the edges. As shown through the previous example (Figure 2.2), the direction of the edge goes from the vertex closer to the root to the one further from it.

The first property that should be studied is whether or not this formalism can represent any digraph as a string. As the following proposition shows, this holds for our graph formalism:

Proposition 2.1: Every digraph has a string in GE that represents it

Given a digraph $G = (V_G, E_G)$, a string $w \in GE$ exists such that $g(w) = G$. To prove this, we define the function $s : U \to GE$:

$$
s(G) = \begin{cases} \varepsilon & if \quad G = (\varnothing,\varnothing) \\ [\,|^n\,] \cdot s((V_G - \{n\},\varnothing)) & if \quad n \in V_G \; and \; E_G = \varnothing \\ [\,|^n\,[\,|^m\,]] \cdot s((V_G, E_G - \{(n,m)\})) & if \quad (n,m) \in E_G \end{cases}
\tag{2.9}
$$

□

The previous proof is quite trivial, as any edge can be independently expressed in a *bracketed section* (i.e. a substring enclosed between an opening bracket and a closing

bracket). However, it should be noticed that this formalism allows different string representations of the same digraph. For example, the digraph illustrated in Figure 2.2, was built from a different string that the one obtained through the application of the function defined in the previous proposition. The same digraph can also be expressed through the string [| [||][||| [||]]][|||| [|||]].

All of these representations are equivalent, being the one defined in Proposition 2.1 very similar to the typical mathematical representation of a graph (i.e. just a set of edges over a set of vertex). However, the other two strings that have been associated with the graph illustrated in Figure 2.2 are more interesting. The difference is that the connectivity of the graph is dependent on the structure of the string.

This characteristic is the fundamental advantage of this formalization. All bracketed sections create a local topology, based on that of a tree-graph. In this way, the distribution of the brackets within the string gives explicit information about the connectivity of the graph. Therefore, there is no need to process the whole string in order to obtain relevant information about the graph represented by it. This property allows us to define a search algorithm that creates new graphs from existing ones through changes that either preserve the overall topology of the graph or modify it in a smart way.

Chapter 3

A Spiking Neural Network model

As it was already explained in the introduction of this document, we applied a different neural network model than the one classically applied in Computer Science. This model, known as Spiking Neural Network (SNN) is focused on simulating, with a higher level of detail, the process of every single neuron being stimulated during certain amount of time, until the point it fires its own output to the network.

The most relevant characteristic is that the stimuli that a neuron receives and sends are always the result of individual neurons firing a single spike. Each neuron stores the information of its own *membrane potential* (i.e. the state of excitation that it has at the moment). The dynamics of this model are intrinsically time dependent, as a neuron will not fire if it receives a single stimulation, but rather increase its membrane potential with every received spike. Although biological neural network develop their activity over time in an asynchronous fashion, we chose to simplify our approach and consider time to be a discrete variable. Therefore, all membrane potentials are updated synchronously every unit of time. This unit is what we call a *time step* of the simulation. In order to explain the dynamics of our model, some key constants affecting the membrane potential must be defined:

- *Threshold potential* (θ): The membrane potential which, once reached, causes the neuron to quickly increase its potential (through a process called *depolarization* in biological neurons), resulting in the emission of an energy pulse into the network (the spike). The value used in our model is $-30mV$, as it is a typical one in biological neurons.

- *Spiking potential* (p): The membrane potential that the neuron reaches when firing a spike. A typical biological value is $10mV$ and is the one applied in our model.
- *Hyperpolarization potential* (h): Once a spike has been fired, the neuron quickly changes its potential again, now acquiring the lowest value of it. This value is set at $-70mV$ in our model, again inspired in the potential of biological neurons.
- *Resting potential* (r): If a neuron is not being stimulated, its membrane potential tends to stabilize at this potential value. A typical biological value is $-50mV$ and is the one applied in our model.
- *Decay rate* (d): If the neuron is not receiving any stimuli, its potential gradually changes until reaching the *resting potential* value. This change happens either when the membrane potential is higher than the resting value (decreasing it) or when it is lower (increasing it). The rate of this change is defined in our model for every time step as the approximation to the resting value by a 10% of the difference between the current membrane potential and the resting potential.
- *Spike Strength* (δ): This value determines how much the membrane potential of a neuron is increased when receiving a single spike. In biological neurons, this value is dependent to physiological variables, and neurons do not linearly increase their potential with every spike. Our model simplifies this aspect of the neuron dynamic and gives a fixed value to the parameter.

The simulation of the activity in the network is done by applying the dynamic of each neuron at every time step. Figure 3.1 shows the curve described by the membrane potential when a neuron fires. In this project, the dynamic follows an *Integrate & Fire* model. This model is applied for every neuron at each time step. To explain it, we will consider a binary matrix W that describes the topology of any given SNN. Its elements are represented by $W(i, j)$ and determine whether or not there is a connection *from* the *i-th* neuron *to* the *j-th* neuron. Also, the membrane potential of each *i-th* neuron is given by a variable, $x(i, t)$, that indicates its value at the *t-th* time step. The dynamic is described in two stages:

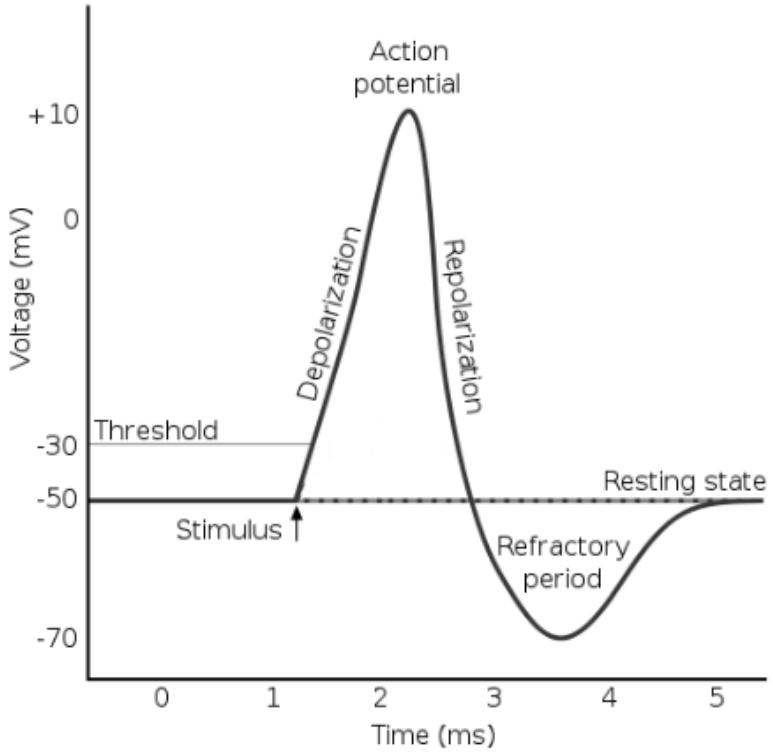

FIGURE 3.1: Membrane potential during a spike
(adapted from www.khanacademy.com).

- The first stage *integrates* the inputs received. To do so, all the neurons that fired at the previous time step are found:

$$s(i,t+1) = \begin{cases} 1 & if \quad x(i,t) = p \\ 0 & if \quad x(i,t) \neq p \end{cases} \tag{3.1}$$

  Then, the new membrane potential value is obtained. This calculation takes into account the stimuli received and the decay rate:

$$I(i,t+1) = x(i,t) + \sum_{j=0}^{N} (w(j,i) \cdot s(i,t) \cdot \delta) + d \cdot (r - x(i,t)) \tag{3.2}$$

- The second stage evaluates which neurons exceeded the threshold potential. The membrane potential of those neurons is then updated to be firing at the current time step. The neurons that fired at the previous time step are hyperpolarized. The membrane potential of the rest of the neurons is changed normally:

$$x(i,t+1) = \begin{cases} p & if \quad I(i,t+1) > \theta \\ h & if \quad x(i,t) = p \\ I(i,t+1) & if \quad otherwise \end{cases} \tag{3.3}$$

Neurons following this model change their membrane potential value at each time step, increasing it when some stimuli reaches them or decreasing when they are not excited. Figure 3.2 illustrates this dynamic. When a membrane potential peak reaches the threshold, the neuron is automatically depolarized and fires a spike, to then reset to the hyperpolarization potential.

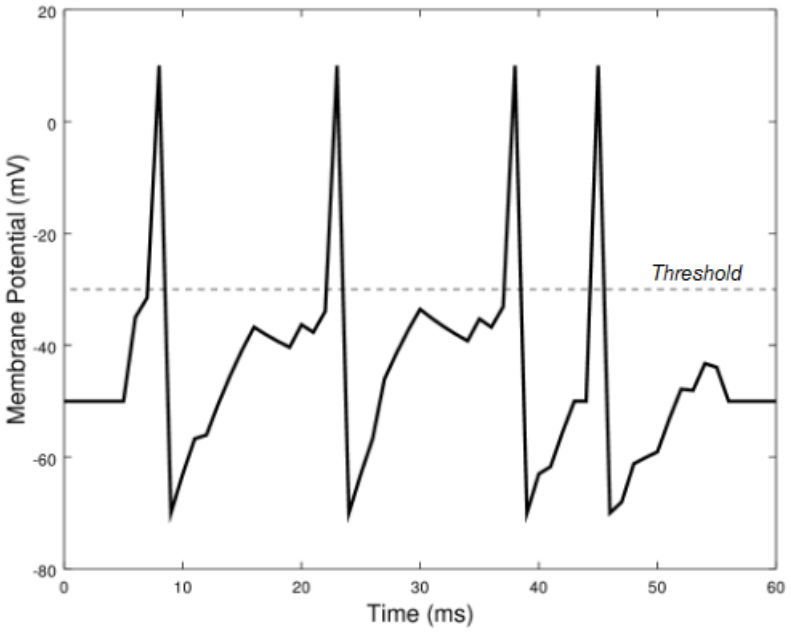

FIGURE 3.2: Dynamics of an artificial neuron following our model.

Considering spikes are acting only through a single time step, the biological interpretation of this unit should be the amount of time that a neuron is firing. This value is estimated to be of $1ms$, according to neurophysiological recordings.

To assign the value for the *spike strength* parameter, we had to choose one that was neither too large to make the neuron fire when few stimuli reach it, nor too small that it made inviable that enough spikes happen to excite the neuron. In biological neurons, five spikes in a short window of time (i.e. around $3ms$) are considered to be enough to activate a resting neuron. Considering that the difference between the resting potential and the threshold potential is $20mV$, a spike strength value of $5mV$ is assigned. This value makes a resting neuron fire if four spikes reach it in a single time step. If the

spikes do not arrive synchronously, then more stimuli will be needed to activate it, as the membrane potential will decay over time.

Chapter 4

Optimization of SNNs with evolutionary algorithms

An evolutionary algorithm is a paradigm of optimization algorithms that is inspired by Darwins evolution theory. It applies the concepts of *random mutation* to change the elements that are being optimized and a *selective pressure* as a criteria for favoring the best mutations.

In order to implement an evolutionary algorithm, a way to represent the possible solutions of the problem has to be designed. This possible solutions are referred to as *individuals*. This representation is a formal description of the characteristics of those individuals and it is usually a string with a well-defined format. This representation is called *genome of the individual* and must satisfy some conditions:

- There must be an automated process to obtain the individual from a given genome. This process is called *decodification*.
- All valid genomes must yield a valid individual when that decodification is applied.
- All individuals must have at least one genome that can generate them through the application of the decodification.

In our case, the valid individuals are any digraph and the valid genomes are all the strings in GE. The decodification process is the function given in Definition 2.5, which satisfies the two first conditions. The third condition is proved to hold in Proposition 2.1.

The evolutionary algorithm starts creating a set of genomes, which are very simple. The set individuals represented by them is called the *population*. The algorithm then

enters a loop, applying in each iteration subtle changes in the genomes of the population, yielding a new set of individuals every iteration (i.e. a *generation*). Not all of the individuals are kept from an iteration to another: some are discarded and some are duplicated. This decision is made by a selection routine that favors those that are more promising to solve the problem.

The changes made to the genomes in each iteration are called *mutations*. Those are done by randomly choosing and applying one of the *mutation rules* specified by the developer of the evolutionary algorithm. Any mutation rule is bound to two conditions:

- It can not be *lethal* (i.e. the resulting genome has to be a valid one).
- It should minimize *disruption*, (i.e. the mutation applied to the genome must not yield a major change on the individual), what ensures that the population is optimized gradually. Big changes on the individuals at every iteration would cause the search to be random.

The definition of the mutation rules is the way the developer of the evolutionary algorithm can control how the search space is going to be traversed. Therefore, they must be carefully selected in order to give the evolutionary algorithm the means to reach the individuals that would solve the problem.

After the mutation is applied, the genomes are decodified and the new individuals are tested. A *fitness function* must be designed. This score must be better the more promising is the individual in achieving the objective. This function is how the search is guided, favoring those genomes which individuals obtain a better fitness score. Similar individuals (i.e. those that only differ in some mutations) will have a similar fitness value, but it must not be the same, in order to inform the algorithm which individual to favor the most.

One of the main characteristics of evolutionary algorithms is that the fitness function is not a formal analysis on how promising the individual is (i.e. a heuristic). It is just a routine that *tests the individual on the field*, applying it to solve the problem (or a simplified variation of it) and check how well it did. As explained in the introductory chapter, this characteristic implies that the user of the algorithm does not need to know any information about how the fittest individual would be.

Once the score for each individual is given, the selection routine takes place. Its goal is to choose from the tested individuals those that will be part of the population for the next iteration. This selection is dependent on the score obtained for each individual,

favoring those that performed better. The process starts by defining a range of values and assigning an interval of them to every individual. The length of this interval is given by the score the individual obtained. None of the intervals share values in common (i.e. they are all disjoint to each other). Then, as many random numbers as individuals in the population are generated within the whole range of values. The individual that had assigned the interval in which the random number occurred, is then included in the population for the next iteration.

In this way, the most fitted individuals will have a larger interval associated to them, increasing their probability to be selected. The same individual can be selected more than once, therefore the best individuals will leave more 'descendants' for the next iteration, which will start by mutating them (so it is not likely that two identical individuals reach the next application of the fitness function).

The individuals that obtained the worst scores will be less likely to leave offspring in the new population. However, thanks to the selection being random, some of them will prevail. This is useful to avoid that the few best individuals end up filling the whole population with their descendants, taking out other branches of promising individuals that were not as well fitted in certain iterations but also had a chance of achieving the objective after some more mutations. On the other hand, the population must be large enough (i.e. over a hundred individuals) in order to be very unlikely that no descendant of the best individuals is included.

To further ensure that the population never decreases its performance, an *elitist option* is applied. This means that, at each iteration, the best individuals are kept for the next iteration and are marked, so they will not mutate. A recommended proportion of elite individuals in the population is about 1%. Given this criteria is completely deterministic, we know for sure that the best individual of any following iteration will at least be equally fitted than the one just found.

4.1 The proposed model

As it has already been discussed, the effectiveness of the evolutionary algorithm depends on three aspects: the genome expression, the set of mutation rules and the fitness function. Those have to be carefully chosen in order to obtain the desired results. The genome applied is the graph representation explained in the chapter '*A grammatical formalism for graph generation*' of this document. In this section, the set of mutation rules and the fitness function will be presented. An analysis of them will be

done, in order to justify each of the characteristics they show, explaining why they are needed.

In the proposed model, the evolutionary algorithm tries to develop a SNN from a starting set of unconnected neurons. It searches for a topology to connect them in order to maintain background activity. The population that is given to the algorithm in its initialization is a set of neural networks that already have a certain number (N) of neurons. This parameter is defined for each execution of the algorithm and determines the approximate size of the network that will be generated. The genome of each of the individuals of the initial population is:

$$[\,][\,|\,][\,||\,]\ldots[\,|^{N-1}\,] \tag{4.1}$$

The next step is to define the set of mutation rules. Those change the genome of the population of the last iteration to yield new genomes. Therefore, they are the way the evolutionary algorithm takes advantage of the genome representation, exploiting its characteristics to make local changes on the topology of the SNN. Given that the main property of the developed genome expression is that the graph is built up from the union of tree-like structures, the mutation rules must then focus on managing this structures.

4.1.1 Set of mutation rules

To explain the mutation rules designed, we need to formalize some concepts that will help us manage the information contained in the genome. We will be using the mathematical concept of vector, defining some functions over it:

Definition 4.1: Relevant operations over vectors

- Given any vector x, we define $||x||$ as the number of elements in it.
- Given any vector x, the *i-th* element of the vector is defined as $x(i)$.
- Given any two vectors x, y we define their *concatenation* as the binary operation:

$$x * y = \langle x(1), x(2), \ldots, x(||x||), y(1), y(2), \ldots, y(||y||)\rangle \tag{4.2}$$

□

Definition 4.2: The *structure* vector, S

Given a certain genome $w \in GE$, we define a vector which elements are in $\mathbb{N} \cup \{0, -1\}$. It comprises the distribution of brackets throughout the genome, pairing together the balanced brackets by indicating the distance between the opening and closing one. We define it through a recursive function that parses the whole genome:

$$S(w) = \begin{cases} S(u) * S(v) & if \quad w = uv \\ \langle 2 + ||S(v)||, 0 \rangle * S(v) * \langle -1 \rangle & if \quad w = [sv] \\ \varnothing & if \quad w = \varepsilon \end{cases} \tag{4.3}$$

Where $u, v \in GE, s \in \{|\}^*$ and the operator $(*)$ is the previously defined concatenation of vectors.

□

Definition 4.3: The *labels* vector, L

Given a certain genome $w \in GE$, we define a vector which elements are in $\mathbb{N} \cup \{0, -1\}$. It comprises the information about the different labels in the genome:

$$L(w) = \begin{cases} L(u) * L(v) & if \quad w = uv \\ \langle -1, n(s) \rangle * L(v) * \langle -1 \rangle & if \quad w = [sv] \\ \varnothing & if \quad w = \varepsilon \end{cases} \tag{4.4}$$

Where $u, v \in GE, s \in \{|\}^*$ and the operator $(*)$ is the previously defined concatenation of vectors. The function n is the one introduced in Definition 2.3.

□

Given a pair of these vectors representing the same genome, recovering it is trivial: The non zero elements of the *structure* vector indicate the appearance of a bracket, while non negative elements of the labels vector indicate how many vertical bars appear in that certain position. We will refer to this function as $r(S(w), L(w)) = w$. Then, we define the set of mutation rules by obtaining these vectors from the original genome and changing them as the mutation requires.

Definition 4.4: Leaf duplication

Given any genome $w \in GE$ we first obtain both vectors $S(w)$ and $L(w)$. We will refer to them as s and l respectively. Then, the mutation rule generates two new vectors s' and l'.

First, a random index $1 \leq i \leq ||s||$ is selected, such that $s(i) = 2$. This index is used to obtain the section of the vectors to be duplicated:

$$s_d = \langle s(i), s(i+1), s(i+2) \rangle$$
$$l_d = \langle l(i), l(i+1), l(i+2) \rangle$$

Then, a second random index $1 \leq j \leq ||s||$ is selected, such that $s(j) > 0$. The section is placed where the index determines.

$$s' = \langle s(1), \ldots, s(j), s(j+1) \rangle * s_d * \langle s(j+2), s(j+3), \ldots, s(||s||) \rangle$$
$$l' = \langle l(1), \ldots, l(j), l(j+1) \rangle * l_d * \langle l(j+2), l(j+3), \ldots, l(||l||) \rangle$$

The genome retrieved is the one resulting from $r(s', l')$.

□

Definition 4.5: Tree duplication

Given any genome $w \in GE$ we first obtain both vectors $S(w)$ and $L(w)$. We will refer to them as s and l respectively. Then, the mutation rule generates two new vectors s' and l'.

First, a random index $1 \leq i \leq ||s||$ is selected, such that $s(i) > 2$. This index is used to obtain the section of the vectors to be duplicated:

$$s_d = \langle s(i+2), s(i+3), \ldots, s(i+s(i)-1) \rangle$$
$$l_d = \langle l(i+2), l(i+3), \ldots, l(i+s(i)-1) \rangle$$

Then, a second random index $1 \leq j \leq ||s||$ is selected, such that $s(j) > 0$. The section is placed where the index determines. However, its labels must be changed first:

$$l'_d(x) = \begin{cases} l_d(x) & if \quad l_d(x) = 0 \\ |l_d(x) + l(j+1) - l(i+1)| & if \quad l_d(x) \neq 0 \end{cases} \tag{4.5}$$

It is important to notice that when the label is updated, the absolute value of it is taken. This is due to negative labels can not be represented by a genome, so when a label reaches zero vertical bars, the rule starts adding them instead of subtracting. Finally, both sections are placed at their respective position.

$$s' = \langle s(1), \ldots, s(j), s(j+1) \rangle * s_d * \langle s(j+2), s(j+3), \ldots, s(||s||) \rangle$$
$$l' = \langle l(1), \ldots, l(j), l(j+1) \rangle * l'_d * \langle l(j+2), l(j+3), \ldots, l(||l||) \rangle$$

The genome retrieved is the one resulting from $r(s', l')$.

□

Definition 4.6: Tree removal

Given any genome $w \in GE$ we first obtain both vectors $S(w)$ and $L(w)$. We will refer to them as s and l respectively. Then, the mutation rule generates two new vectors s' and l'.

A random index $1 \leq i \leq ||s||$ is selected, such that $s(i) > 0$. This index is used to decide what section of the vectors will be removed:

$$s' = \langle s(1), \ldots, s(i-2), s(i-1), s(i+s(i)+1), s(i+s(i)+2), \ldots, s(||s||) \rangle$$
$$l' = \langle l(1), \ldots, l(i-2), l(i-1), l(i+s(i)+1), l(i+s(i)+2), \ldots, l(||l||) \rangle$$

The genome retrieved is the one resulting from $r(s', l')$.

□

These mutation rules do not change the network by adding or deleting individual connections, but rather managing already existing structures:

- *Leaf duplication*: This rule extends already existing structures in the genome, creating an additional connection. It is needed for the algorithm to create the

structures that the other two rules will manage. Therefore, it is of most importance at the first iterations of the algorithm.

- *Tree duplication*: This rule copies an already existing structure and repeats it in another place of the network. The duplicated structure connects a different group of neurons thanks to the change applied to the labels. It is important to notice that this mutation rule can create new neurons that were not included in the initial individual. This mutation rule is the most important of the set, as is the one that makes possible that a beneficial structure (i.e. one that helped some neurons to reciprocally activate each other) is repeated in other places of the network, increasing the global ability of the SNN to maintain background activity.
- *Tree removal*: This rule takes out a whole structure from the genome. It is relevant in the last iterations of the algorithm, as it can prune unnecessary structures.

This management of groups of edges is made possible by the representation of the genome, as it is in it where the explicit structures are shown through the nested brackets hierarchy. These mutation rules would be difficult and inefficient to apply using the typical mathematical representation of graphs, due to the need to look for edges that share vertex in common.

Besides, it should be noticed that the criteria to make the random choice of the substring to be duplicated in *leaf duplication* and *tree duplication* are mutually excluding. This enables us to define both in a single mutation rule:

Definition 4.7: General duplication

Given any genome $w \in GE$ we first obtain both vectors $S(w)$ and $L(w)$. We will refer to them as s and l respectively. Then, a random index $1 \leq i \leq ||s||$ is selected, such that $s(i) \geq 2$. This index is used to obtain the section of the vectors to be duplicated.

If the index selected verifies $s(i) = 2$, then the process described in Definition 4.4 is applied, resulting on a *leaf duplication* mutation. Otherwise, $s(i) > 2$, and the process described in Definition 4.5 is applied, resulting on a *tree duplication* mutation.

□

The evolutionary algorithm randomly chooses which mutation to apply, either *general duplication* or *tree removal*. However, if both rules had the same probability to happen, the tree removal rule would end up destroying any structure created by the general duplication rule. In order to prevent it, we assign a relative frequency of them happening.

After testing different values, it was finally set to be twice probable to duplicate than to remove. This value is enough for the general duplication rule to create structures that will persist in time.

We must also analyze if the desirable properties for the set of mutation rules are met. These are defined at the beginning of this chapter and ask the mutation rules to be neither *lethal* nor *disruptive*.

We first discuss that the defined mutation rules are not be lethal. This is trivially verified as all rules duplicate or remove sections enclosed between brackets, maintaining the brackets balanced. The duplication rules always place the copied section right after a label, where bracketed sections can appear in any valid genome. Therefore, the resulting genome is always an element of GE. Then, we explain why these mutation rules are not highly disruptive.

The duplication rules add more connections to the graph. This will only affect on increasing the overall ability of the neurons to activate each other. On one hand, the leaf duplication rule just creates a single connection, having a low impact on the dynamic of the network. On the other hand, the tree duplication rule copies a structure that is already present in the network. Thus, its complexity and contribution to the fitness of the network is relative to what the SNN already shows, not making any sudden change to the dynamic it already had.

The tree removal rule must also be considered. Its application is actually potentially disruptive, as whole structures could disappear. However, taking into account that it is applied half the amount of times that the general duplication rule, it is intuitive to understand that at the moment this rule removes part of a structure, it would have already been duplicated elsewhere in the genome. Therefore, this rule will rarely take out a structure which disappearance causes the dynamic of the network to dramatically change, as there should be plenty of structures throughout the network. All of these structures should have similar relevance, as they are copies of each other. Then, rather than being an obstacle, this rule helps in the development of the network, as it changes the structures, giving an increased diversity of them.

4.1.2 Fitness function

Having defined and analyzed the mutation rules, now it is necessary to give the fitness function. By it, we give the algorithm the specification of the objective it must try to satisfy. This objective is to develop SNNs that are able to maintain background activity.

Therefore, the fitness function will simulate the behavior of the neural networks and give a score to each one of them. The score will be greater for the SNNs that maintain activity for a longer period of time.

The fitness function is applied to every individual of the population, at each iteration of the algorithm. First, the topology of the SNN of each individual must be obtained through the decodification of its genome. Then, the neural network is simulated, following the model described in the chapter of this document titled '*A Spiking Neural Network model*'. A starting stimulation is needed in order to make any activity happen. This is due to the neurons being stable at their resting potential if no spikes reach them. This starting stimulation is done at the beginning of the simulation by forcing a percentage of the neurons to be at their spiking potential (i.e. firing just when the simulation starts). The percentage that proved to be most adequate is 30%, being a balance between activating enough neurons to excite others while not leaving too many neurons hyperpolarized at the next time step.

The spikes of the activated neurons at the start of the simulation should increase the membrane potential of other neurons. The potential of these neurons may get to the point of making them fire in the next time step of the simulation. If the topology of the SNN allows it, then the starting stimulation could be enough to provoke the activation of an amount of neurons that will in turn activate others. Through this time step simulation, neurons will follow a cycle of firing a spike, resetting their potential, receiving spikes from other neurons and integrating them to rise their membrane potential to the point of firing again.

The chain reaction will last for a period of time steps. This process of neurons being able to activate others is not helped by any kind of external stimulation, excluding the activation of neurons at the start of the simulation. Thus, all the stimuli that the neurons receive comes from another neuron which was active one time step ago. This is interpreted as background activity, in the sense it is maintained by its own, not requiring any external input. Therefore, the score given to the SNN is the amount of time-steps that at least had an active neuron.

As the activity could be maintained indefinitely, a maximum amount of time steps, T is defined. When this value is reached, the simulation stops. This simulations trace is stored in a binary matrix P, which each element $P(t, i)$ shows if the *i-th* neuron was firing at the *t-th* time step. The score (α) given to the SNN is in the range of 0 to 1, and is calculated as the proportion of time steps at which at least a neuron fired:

$$\alpha = \frac{||\{t \mid \exists i, P(t, i) = 1\}||}{T} \tag{4.6}$$

This method of calculating the fitness score puts each SNN of the population to test. It simulates them and measures their ability to maintain background activity, assigning each of them an individual score. That score is higher the better they manage to fulfill the objective. Thus, it happens to be a valid fitness function, as the activity of two similar neural networks (i.e. only distinguished by a few number of mutations) will slightly differ, and the one that lasted longer will be assigned a higher score.

This whole model is appropriate enough to satisfy the objective we had established. The neural networks obtained by applying it are able to maintain background activity, as it will be shown and discussed in the chapter dedicated to the results. However, we decided to consider an additional criteria in the fitness function evaluation. This criteria is explained in the following section.

4.2 A more refined model

The problem that the previous model had is understood when studying the activity of the neural networks that it generated. This study is shown in the chapter of this document dedicated to the analysis of the results. What was found is that their neurons fired at a rate too quick compared to a biological neuron.

As we wanted the resulting SNNs to be as close to their biological counterpart as possible, we decided to consider an additional criteria in the fitness function evaluation. This criteria aims to favor those neural networks that not only maintained background activity, but also presented a slower activation rate of their neurons.

To define this criteria, a new parameter was added, namely G. This parameter determines the desired amount of time steps between any pair of consecutive spikes of any given neuron. Analyzing the simulation trace, P, we can obtain the vector of gaps between spikes for each neuron:

$$g_i(k) = t_{k-1} - t_{k-2}, k \in [1, ..., n] \tag{4.7}$$

$$\text{where} \quad t_{n-1} < t_n, \quad P(t_{n-1}, i) = 1, \quad P(t_n, i) = 1, \quad \forall t' \in (t_{n-1}, t_n) : P(t', i) = 0$$

The arithmetical average of the vector is calculated and represented as $\overline{g_i}$. Using this value, another score is obtained, which also ranges from 0 to 1. The way to calculate it is to compare the average gap of each neuron with the G parameter. The closer the

value is to that parameter, the better score the neuron receives. The comparison is done by applying the following function:

$$\chi(i) = \begin{cases} \overline{g_i}/G & if \quad \overline{g_i} \leq G \\ \\ 2 - \overline{g_i}/G & if \quad \overline{g_i} > G \end{cases} \tag{4.8}$$

Then, the arithmetical average of these values is computed (β), obtaining a representative score for the whole neural network. Finally, the fitness score (γ) is given by the combination of this *average gap score* (β) and the *activity score* (α) calculated by the original fitness function:

$$\gamma = \begin{cases} \alpha & if \quad \alpha < 1 \\ \alpha + \beta & if \quad \alpha \geq 1 \end{cases} \tag{4.9}$$

This implies that the criteria just explained is only applied when the maximum *activity score* is reached (i.e. the network maintains background activity throughout the simulation). Therefore, the evolutionary algorithm first searches for SNNs that are able to present background activity and, once obtained, optimizes those neural networks to make their activity the most similar to a biological system as possible. Calculating the sum of both scores is the simplest way to combine their information and allows that the overall fitness score is easily interpreted: if it is below 1, the network has not yet developed a satisfying background activity; if it is in the range from 1 to 2, the network is being refined using the criteria defined by G.

4.3 Variations of the model

The model just explained was the final product of a process of designing different approaches, testing them and analyzing their properties. These took a considerable portion of the time dedicated to the project, and each one of them helped to get a better insight on the objective that we set as our goal. Thus, the most important ones are worth to be briefly discussed:

- *Developing increasingly larger SNN*: The original approach was to create an initial population of networks that were comprised by a single neuron. Specifically designed mutation rules would then add additional genome sections and increase

the length of the labels, gradually increasing the amount of neurons in the network. This approach proved to be unworkable, as an unreasonable amount of iterations were needed in order to create SNNs with enough neurons to be able to maintain background activity.

- *Weightening the connections*: Following the principle of the classical model of neural networks, we decided to test what would happen if we specified in the genome the weight of the connection between each pair of neurons. The resulting SNNs tended to create networks where the connections were too heterogeneous: the activity was maintained by a small proportion of the network, that was strongly connected. This behavior is opposed to the way biological neurons interact with each other, thus the approach was discarded.

- *Considering a slightly different SNN model*: Based on a letter published in Nature (Debanne, D. Bialowas, A. & Rama, S., 2012, see bibliography), we considered the option of designing a different SNN model. In it, the update of the membrane potentials of neurons would not only be affected by the spikes that reached them, but also depending on the value of the membrane potentials of the neurons connected to them. Although this approach would have helped the gradual development of the fitness score during the algorithm execution, we finally decided that such a complex interaction between neurons was not in concordance with the level of detail present in our SNN model.

- *Applying a learning algorithm after the evolution*: We considered the option of applying a certain learning algorithm used in SNN models, the STDP (Spike-timing dependent plasticity, see bibliography) after the evolution phase had finished. That would have helped giving a final refinement of the generated networks. We finally decided against using it, as we considered more important to focus on obtaining the best result using the tools initially proposed.

Chapter 5

Results

5.1 Maintaining background activity

The algorithm developed is now put to test. To do so, the program that implements it is executed for different configurations of its parameters. The results shown in this chapter were obtained running the program in the supercomputing node, as the configuration needed for the algorithm to develop interesting networks was too demanding for a personal computer. This aspect is further explained in the corresponding appendix dedicated to the supercomputing node. Now, we focus on showing and discussing the results.

We will be focusing in the SNNs obtained by a program configured with the following parameters:

- *Population size*: 200 individuals were evolved in parallel. The parameter was adjusted to this value after an empirical process of testing which one yielded the best performance. The larger the population, more mutations are tried and, therefore, it tends to be easier to find fitter individuals. However, above 200 neural networks, this benefit was hardly noticeable, while increasing the time required to execute the program.
- *Amount of generations*: 2500 iterations of the evolutionary algorithm were necessary to obtain the results shown here.
- *Amount of elite individuals*: the 2 best individuals of each generation were not changed at the next iteration, following the *elitist option* (with the usual 1% of the population).

- *Initial amount of neurons*: 40 was the largest value that could be applied in a reasonable amount of time.
- *Proportion of initially activated neurons*: 30% of the neurons were activated at the start of all simulations. This value is a balance between activating enough neurons to excite others and not leaving too many neurons hyperpolarized for the next time step.
- *Simulation time*: 100 time steps were analyzed for every simulation of each of the SNNs. Considering a perfect score is given to the SNNs that maintain activity throughout the simulation, this value had to be the largest as possible. However, it makes a great impact on the performance of the algorithm, thus a balance had to be empirically found.
- All the membrane potential related values were the same for all executions, corresponding to the ones defined in the chapter '*A Spiking Neural Network model*'.

The first aspect to discuss is how the SNNs improved during the evolution process. Figure 5.1 illustrates this point. The algorithm succeeds at developing a SNN that maintains activity throughout the simulation. This network has a total of 88 neurons.

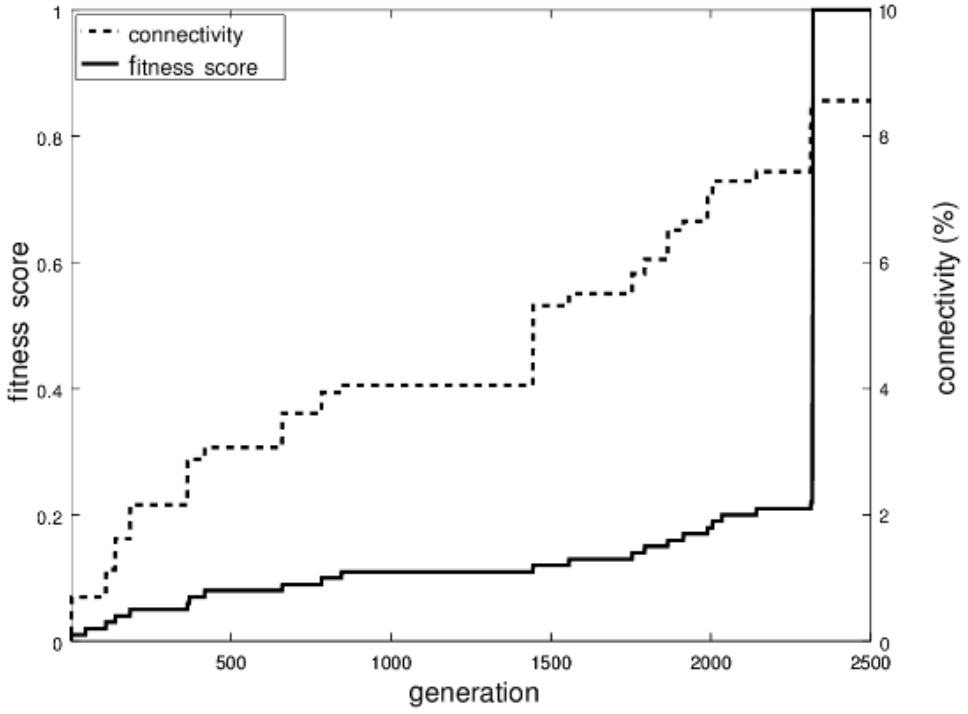

FIGURE 5.1: Fitness and connectivity of the best individual in each generation.

At first glance, the way the fitness score develops is strikingly abrupt at the end of the execution. This phenomenon can be explained rather intuitively. At the beginning, the initially activated neurons are not able to excite even a single neuron. The successive generations start yielding SNNs that have enough connections to maintain activity for the few first time steps. A step up in the fitness score is achieved every time the activity lasts for another instant, which at some cases require a considerable increase of connections. At some point, hyperpolarized neurons have to be activated once again in order to extend the duration of the activity. When this happens to a significant amount of neurons, they are then enough to be able to excite others. And here is the catch: the connections required to do so are already present. Thus, at some point there is no necessity of adding more connections for a chain reaction to happen, creating a long lasting activity in the network.

Given that the *elitist option* is applied, the best individual of each generation is always at least as fitted as the one from the previous one. Thus, the fitness score does not drop at any point. Actually, each flat section corresponds to a single SNN that persisted through a period of generations. During it, no fitter individual was found.

The dashed line indicates the density of connections in the network. It should be noticed that it shows a monotonous increase, accompanying the fitness score. This indicates that the fitness of the individual was strongly favored by the amount of connections it had. This is intuitive, as more connections turn into a greater amount of interaction between neurons.

It should also be noticed that the biggest leaps in the amount of connectivity happen after the larger flat areas. Thus, the periods of generations that do not yield a fitter individual are actually being helpful. Through them, the fittest individual is reproduced and parts of its structure duplicated, increasing the overall fitness of the population. Once the individuals of the population are almost as fit as the best one, there are a larger amount of candidates to become the next elite. Then, the chance that a mutation is able to provide a significant improvement is at its peak. It may also be happening that during that period another individual was developed, being distinctively different to the one that has been the elite until that point, and able to overtake the challenge that the previous elite descendants could not achieve.

We can observe the activity of the final SNN during its simulation (Figure 5.2). It is maintained for 122 time steps and shows an unpredictable pattern during the whole simulation.

The activity of the network abruptly terminates after the highest peak of active neurons is reached. This is due to a quarter of the neural network being hyperpolarized at the

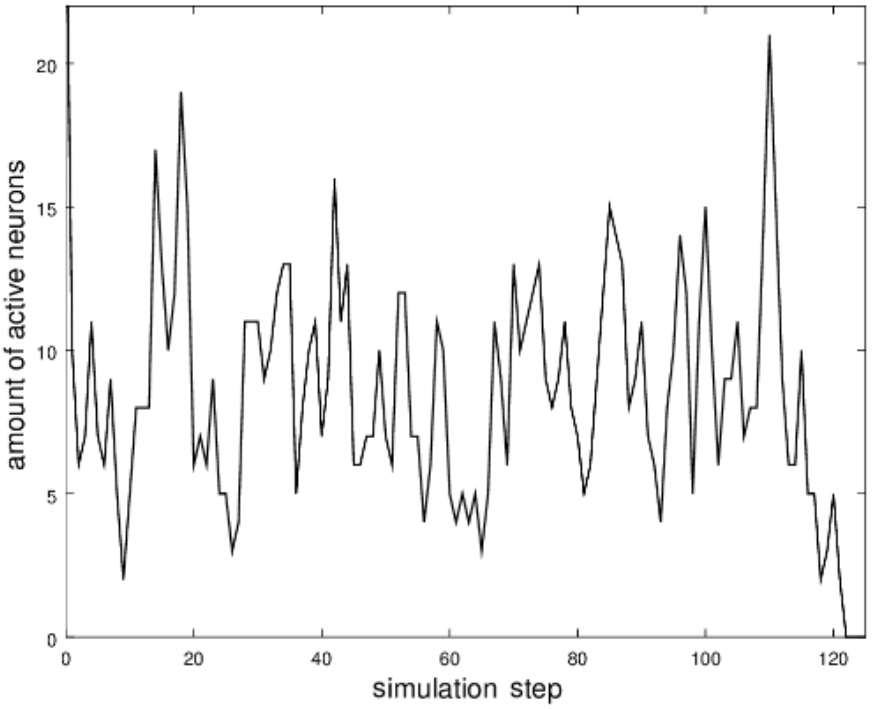

FIGURE 5.2: Amount of active neurons at each instant. No obvious patterning shows up.

next time step. The amount of spikes fired do not excite enough neurons and, thus, the amount of spiking neurons through the following iterations gradually decreases. Although some peaks of neuron activity also happen, they are not enough to reactivate the group of neurons that synchronously fired during the previous time step. This behavior is repeated through the whole simulation, alternating activity peaks with depressions. It was the overly high peak that broke the balance and caused the activity to stop.

If we generate SNNs by randomly creating connections between the neurons, we will see that they can also present background activity. When the amount of connections in a network is above the 15% of the total quantity of possible connections (i.e. all pairs of neurons), the randomly generated networks are able to maintain activity throughout the simulation. However, this activity is cyclic in almost all cases (Figure 5.3), unlike the one developed by our evolved SNNs, which is unpredictable.

Certain values of connectivity yield randomly generated networks that actually present the non cyclic behavior observed in all the evolved SNNs. At these ranges of connectivity, the random SNNs can also happen to be unable to maintain any activity at all.

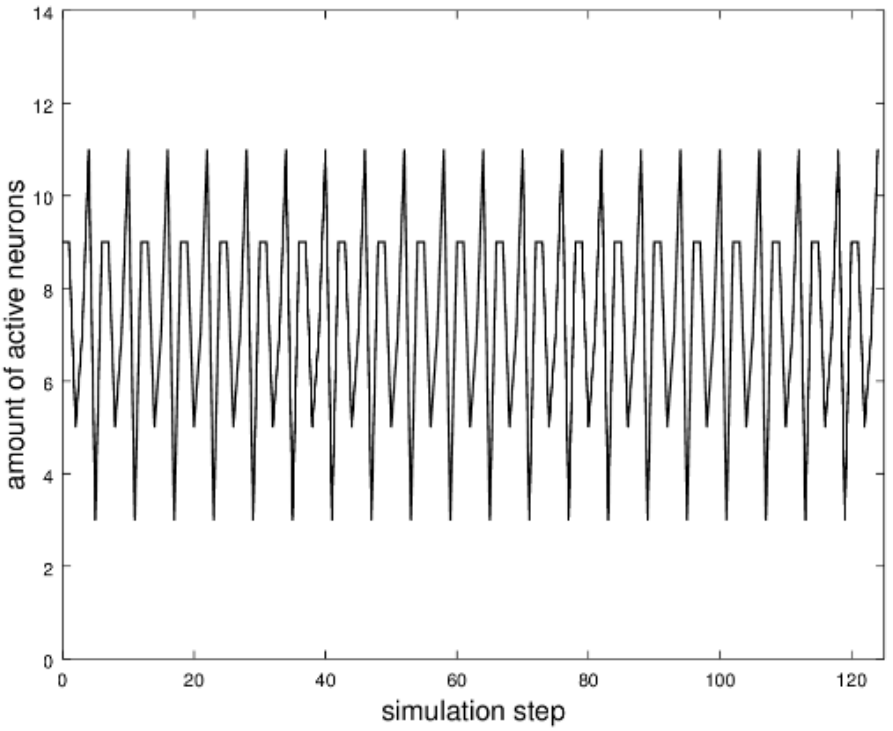

FIGURE 5.3: Activity on a randomly generated network. The underlying activity has been verified to be cyclic.

5.2 A more realistic physiology

Through the application of the algorithm designed during this project, we obtained SNNs that are capable of maintaining background activity. But randomly connected networks can also present this behavior. What is interesting about having an algorithm that searches for them is that we can define more complex fitness functions. By its means, we can obtain SNNs that behave in a way that would be costly to find with random search.

One of the most interesting neural oscillations, which have been strongly related with cognitive abilities, are the *theta waves* (Vertes, 2005). During them, neurons fire at a rate of approximately $5Hz$. Given each time step in our model is considered to last $1ms$, our artificial neurons should fire once each 200 time steps to reproduce this behavior. This is not a feasible goal in our case, as for an activity that slow to persist, it would be necessary to have larger networks (i.e. comprised by a greater amount of neurons). This is due to a signal propagated in the network needs to excite a minimum number of neurons every time step in order to be maintained, otherwise there would not be enough spikes to activate others. This value is in average 10 neurons activations per

time step in our model (see Figure 5.2) and, considering our networks are built by approximately 80 neurons, a 12.5% of the network needs to be activated. If the total amount of neurons was higher, this percentage could be reduced, enabling them to fire less frequently.

Our goal was to check if it was possible to reduce the firing rate of our neurons in a significant amount, and achieve better results than randomly generated networks. We fixed the goal of our evolved SNNs to maintain a firing rate of one order of magnitude higher than biological neurons (i.e. $50Hz$). Then, the distance between spikes should be 20 time steps.

The fitness function was adapted to guide the algorithm to meet this goal. The changes applied have been described in the last section of the previous chapter, fixing the value of the G parameter to 20 time steps. Then, if we put a randomly generated network to the test (Figure 5.4) we can measure how it behaves.

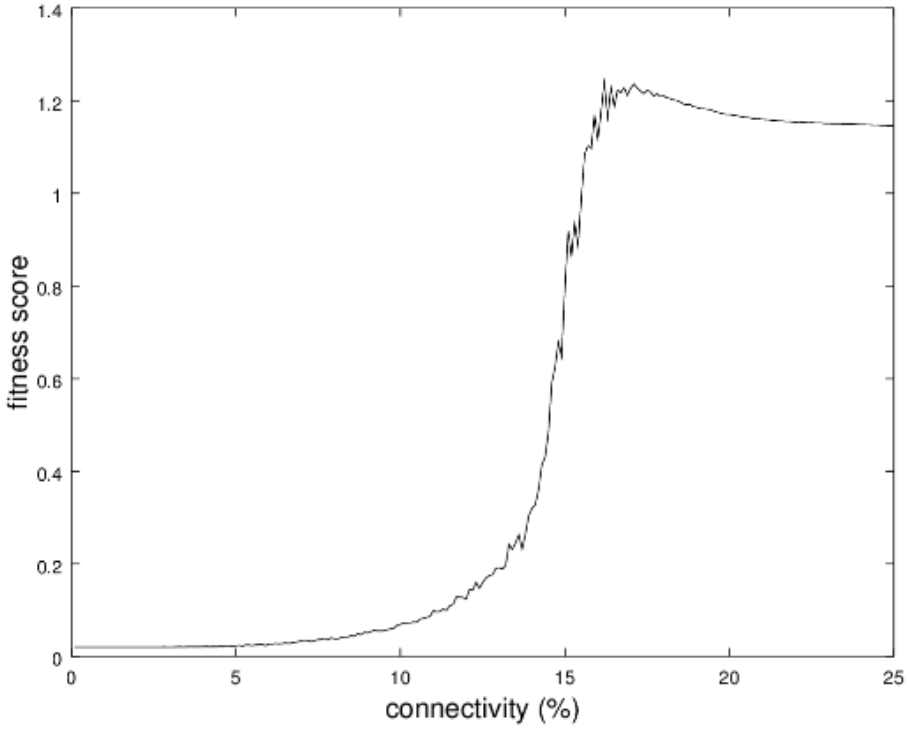

FIGURE 5.4: Fitness score of randomly generated networks depending on their connectivity.

A 15% of connections are needed for background activity to be present in the network. The score given to the best randomly generated network does not reach 1.3, which refers to an average gap of 6 time steps between spikes.

To prove that our algorithm was able to find networks that perform better than randomly generated SNNs, the new implementation of the program was executed. Figure 5.5, shows the increase of the fitness score of the best SNN of each generation. Only the last iterations are shown, from the point the population already presented background activity.

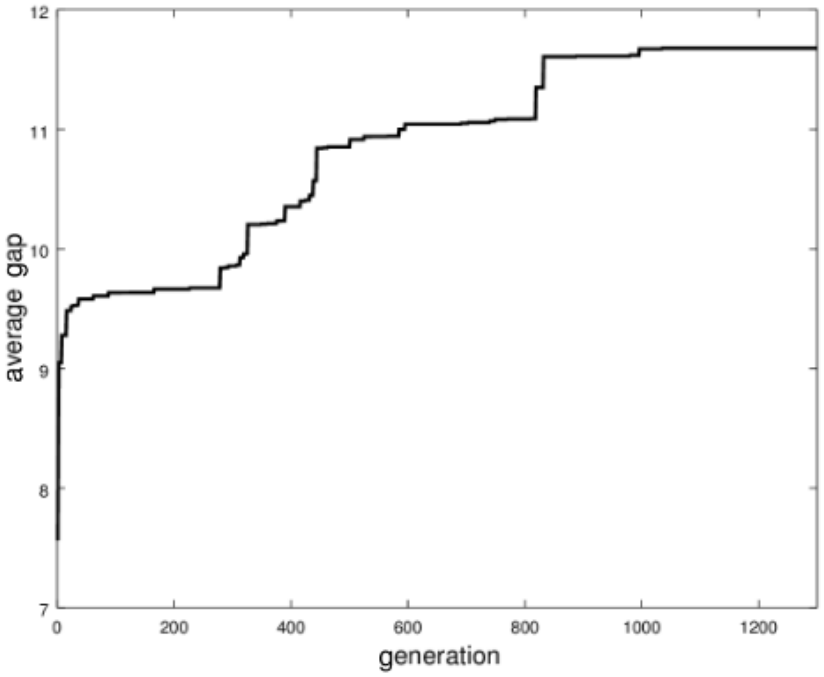

FIGURE 5.5: Average gap between spikes of the best individual in each generation.

Although the goal of 20 time steps between spikes was not reached, the average gap obtained was larger than the one present in randomly generated SNNs (it was almost doubled). This proves that our algorithm is a well guided search that can be oriented to fulfill a goal related to the activity of a SNN.

5.3 Analysis of the resulting network

We now focus on how neurons in the evolved SNNs interact with each other. In this section we study the relation between neurons that fired in a certain instant and those that were active during the previous time steps. First, we consider all pairs of networks and check how many times one fired either a single or two time steps before the other did. This number is represented in Figure 5.6 (left). As this amount increases, the pixel relating both neurons gets darker.

When a vertical or horizontal white line appears in the figure, it is showing that the corresponding neuron did not fire at all during the whole simulation. This is a consequence of the way the labels in the genome (i.e. sequences of vertical bars) are changed by the *tree duplication* mutation rule. The genome of the initial individuals had 40 neurons, and the *tree duplication* rule increased this amount to a value of 88. But it is not necessary that all labels in the range between these values appear in the genome. The inactive neurons do not have any connection with other neurons, so they can not receive stimuli and be excited. (The diagonal white line indicates that a neuron never activates twice in a lapse of three consecutive time steps.)

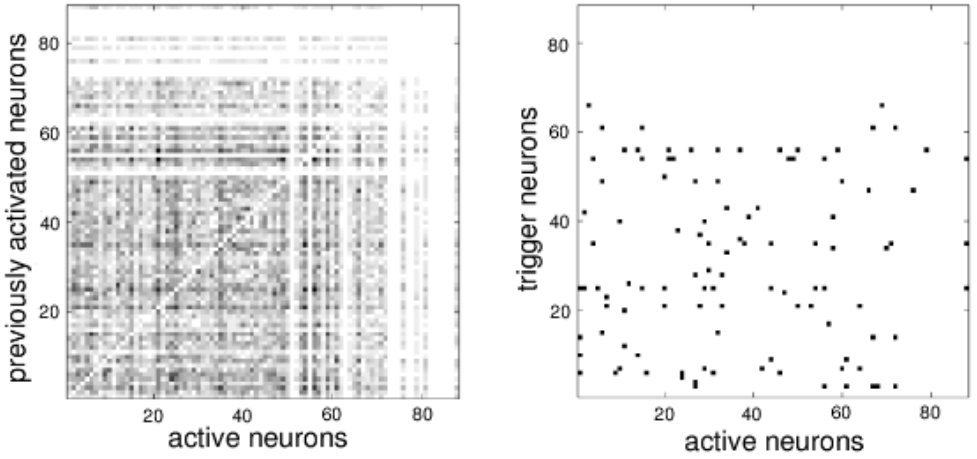

FIGURE 5.6: Frequency of a neuron firing after another has fired (left), where white means no previous activation, and black represents a maximum value of 17. Neurons that are significantly relevant on the activation of others (right) are marked with black dots.

If the white lines are ignored, the image is essentially gray. This means there is no clear pattern of activation between groups of neurons as there are a few amount of neurons that consistently fire after other one did (i.e. there are not that many dark pixels). As we already discussed, if no patterns are repeated, the neural network must be maintaining a non cyclic activity.

However, some pair of neurons seems to actually be related, as some sporadic black pixels appear. Figure 5.6 (right) gives a clearer insight of this correspondence. In this second graphic, a dot appears if the neuron on the horizontal axis noticeably tends to fire before the neuron on the vertical axis does. Thus, it is interpreted that the activation of the first neuron is specially relevant in triggering the activation of the latter. This graph shows that there are some neurons which are more influential than others, as the 25th neuron triggers the consistent activation of 11 neurons, while others do not seem to have any special relevance. We thus detected some kind of hierarchy between

neurons. Considering genomes are defined by a tree-like inspired representation, it seems to be natural that the networks evolved tend to maintain tree structures. This would justify why some neurons are more relevant than others, depending on them being the root of a tree structure, influencing all neurons that form it.

Lastly, we consider the correlation between neurons, determined by their tendency to both activate after certain similar patterns. Figure 5.7 shows, in darker gray, when two neurons share a similar set of trigger neurons. In this case, a great amount of them seem to be clearly related to each other. On the other hand, when the trigger neurons do not match, the correlation is negative and it is represented in lighter gray.

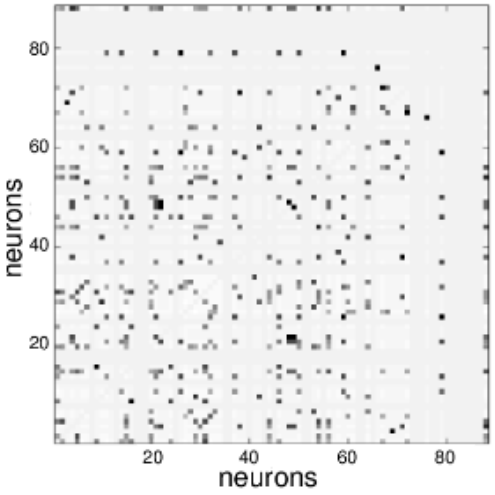

FIGURE 5.7: Correlation of neurons activated by similar groups of neurons (triggers). Darker pixels indicate a higher positive correlation. Light gray indicates negative correlation.

The fact that some pair of neurons share a strong bound to be activated by the same neurons strengthens the previous assumption. The activity of the network seems to reflect the tendency of the topology to be structured in trees. In this case, groups of highly correlated neurons would match with the different descendants of a particular root neuron, which is a relevant trigger of their activation.

Chapter 6

Conclusions

Neuroscientists claim that all the cognitive abilities are given by the way the cells of nervous systems interact with each other. The brain is a versatile machine that is built up by a huge amount of neurons. They work locally, receiving inputs of their neighbour cells and firing output according to their dynamics. However, the signal that a neuron generates is propagated throughout the network, affecting distant cells. The cooperation of all of them elicits a behavior that is not inherent to the neuron itself. When systems are formed by elements that show this capability of fulfilling more complex tasks when cooperating in large groups, it is said that they exhibit *emergent properties*. The SNNs that we have obtained are able to reproduce certain complex behavior: maintaining a signal over time, which is actually an emergent property.

The fact that these networks are obtained through an evolutionary process, that achieves better performance than random search, is quite relevant. It implies that a well informed selection was applied (through the fitness function evaluation), being able to recognize that some topologies were more beneficial than others. Thus, this emergent property is not only given by the elements of the system (which in our case were considerably simplified), but also by the actual way of them cooperating. This aspect is what neuroscientists are now trying to understand: how neurons interact with each other to execute complex tasks. Creating programs like the one developed in this project helps to achieve this goal, as the generated neural networks can be freely examined, in contrast to the limited experimentation that can be done on real specimens.

This project describes an algorithm that generates simplified neural networks capable of maintaining background activity. Thus, it gives the opportunity to conduct further research on what type of topologies favor it. Besides, the fitness function can be redefined to guide the search of a different activity property, as it was shown by adding the

criteria of favoring larger gaps between spikes. Thus, the program should also be valued as an infrastructure to search neural network topologies that show any observable behavior specified in the fitness function.

The mutation rules and fitness function designed were key to the fulfilment of our objective. But the contribution of the genome expression should not be overlooked either. It is by its means that the mutation rules can operate in a way that favors the appearance of complex structures. If the genome had not explicitly informed about the overall structure of the network, the mutation rules would have been left to operate with unrelated groups of connections.

As it has already been discussed in the previous chapter, one interesting characteristic of the SNNs developed by our algorithm is the fact that the activity maintained in them is not cyclic. Designing a network that has a cyclic activity is trivial: connecting groups of neurons in a ring structure is enough. Maintaining a signal without it entering a loop is much more demanding. The mathematical study of this kind of behavior is known as *Chaos Theory*, and has a wide research effort behind it. We have not checked that this behavior is in fact chaotic, but it seems like so given no repetitive pattern can be found. What we can claim for sure is that the behavior we obtained is given by the physiology of the membrane potential. Without it, the neurons in the networks would always respond the same way for a given input. That would imply the appearance of well defined patterns when cooperating in groups, which in turn would lead to a cyclic behavior.

6.1 Further research

The objective of the project was successfully met, but it leaves open an interesting field of study about the topologies that have been obtained. What now follows is to analyze what structures are built by the algorithm, and understand why they are capable of displaying complex emergent behaviors such as a seemingly chaotic activity.

Background activity maintains brain neurons in a constant fluctuation of their membrane potential. Therefore, when an input is given to the network, it propagates through it while combining with its background signal. The resulting output might be affected in a relevant way, thus being dependent to these fluctuations. We could then consider an agent able to interact with its environment (e.g. a robot) controlled by one of our SNNs. Its background activity would determine its internal state, affecting the way it reacts to the stimuli received from the environment. If the topology of the SNN could be

adapted through a learning process for the agent to fulfil tasks, it could yield interesting properties applicable on the Artificial Intelligence field.

6.2 Other contributions

Apart from being a formal piece of research, this project was also an academic experience. Its contribution to my education can be considered in two aspects. On one hand, I have applied the knowledge that I, as a student, have accumulated through my stay in this university. Specifically, the theoretical courses on formal languages (both their syntax and semantics) have been most helpful. In those, we learnt how to formally define the format and meaning given to strings of symbols, focusing in its application to design programming languages. However, this formalization has proven to be equally effective to describe a language of genomes representing graphs, and all the functions related to them. Realising the flexibility of these formalisms was, in itself, an instructive experience. The familiarity I have developed with them will surely be helpful in the future.

On the other hand, while working in this project, I have learnt and developed some capabilities that were not taught during any of the degree courses. Those relating to technical resource management have been discussed in the appendices of this document. Gaining experience on using version control software is fundamental for any computer engineer, and having the opportunity to use a supercomputer was very valuable, as it is an important tool on the scientific discipline. I have also broaden my knowledge about models of computing, by working with evolutionary algorithms and an unconventional type of neural networks. Both are interesting fields of study that are receiving an increasing amount of attention.

This project was done not only for completing my degree, but also as research supported by a scholarship from our University. The scholarship, referred to as 'Beca de Iniciación a la Investigación' is granted to students interested in starting their career as researchers. It was the main motivation for orienting this project to be a scientific study, giving me the opportunity to experience for the first time how research is conducted. What I have learnt by developing this project alongside my tutor, Francisco J. Vico, will undoubtedly be relevant in my future. Understanding how to manage time and effort to be efficient, remaining true to the principles of research and being constant until reaching the objective, are some aspects that can not be learnt in any other way.

6.3 Technical aspects of the project

Considering this project was oriented in a scientific approach, we decided to maintain this focus when writing its document. Thus, we prioritized analyzing the properties of the algorithm developed and the obtained results, rather than focusing on the technical considerations of the work done. In order to maintain a fluent structure throughout the document, we decided not to include technical considerations in its body. This follows the goal of presenting the project as a formal piece of research where the relevant information is the method applied in itself, and not how it was implemented. The most relevant technical aspects of the project are summarized in the two appendices of this document.

Besides, no software engineering abstractions were needed to develop the program, as the system designed does not require complex interactions between its modules. It is based on four clearly separated tasks: the genome decodification, the application of mutation rules, the network simulation and the fitness function evaluation. Each one receives the output given by the module that acted before it, iteratively repeating the cycle until the maximum amount of generations are reached. Each one of these modules was scripted in its own file, and a *Main* script coordinates their iterative execution.

Chapter 7

Conclusiones

La comunidad neurocientífica afirma que toda habilidad cognitiva proviene del modo en que las células de los sistemas nerviosos interactúan entre sí. El cerebro es una máquina versátil formada por una enorme cantidad de neuronas. Éstas trabajan localmente, recibiendo estímulos provenientes de células vecinas y lanzando señales según la dinámica interna que tengan. No obstante, la señal que una neurona produce es propagada por toda la red, afectando a células distantes. La cooperación de todas ellas permite la aparición de comportamientos que no son inherentes a la propia neurona. Cuando un sistema está formado por elementos que muestran la capacidad de llevar a cabo tareas más complejas al cooperar en grandes grupos, se dice que éste exhibe *propiedades emergentes*. Las SNN que hemos obtenido son capaces de reproducir cierto comportamiento complejo: mantener una señal a lo largo del tiempo, lo que es considerado una propiedad emergente.

El hecho de que estas redes se hayan obtenido por un proceso evolutivo, que logra mejores resultados que una búsqueda aleatoria, es considerablemente relevante. Implica que un criterio de selección adecuado fue aplicado (mediante la evaluación de la función objetivo, *fitness function*), siendo capaz de diferenciar que ciertas topologías eran más beneficiosas que otras. Por ello, esta propiedad emergente no viene dada sólo por los elementos del sistema (que en nuestro caso fueron considerablemente simplificados), sino que también es influida por el modo en que éstos cooperan entre sí. Este aspecto es lo que los neurocientíficos se están centrando en entender actualmente: cómo interactúan las neuronas entre sí para realizar tareas complejas. Crear programas como el desarrollado a lo largo de este proyecto ayuda a satisfacer este objetivo, ya que las redes neuronales generadas pueden ser examinadas libremente, a diferencia de la limitada experimentación disponible sobre especímenes reales.

Este trabajo describe un algoritmo que genera redes neuronales simplificadas, pero capaces de mantener actividad de fondo. Por tanto, da la posibilidad de continuar con el estudio de las topologías que han favorecido esto. Aparte, la función objetivo (*fitness function*) puede ser redefinida para guiar la búsqueda de otra propiedad de la actividad, tal y como fue demostrado al añadir el criterio para favorecer mayores distancias entre disparos. Por tanto, el programa debe ser también valorado como una infraestructura para la búsqueda de topologías de redes neuronales que presenten un comportamiento observable especificado en la función objetivo.

El diseño de las reglas de mutación y función objetivo (*fitness function*) ha sido clave para el cumplimiento de nuestro objetivo. Pero la contribución de la expresión genética tampoco debe ser subestimada. Es gracias a ella que las reglas de mutación pueden operar de un modo tal que favorezca la aparición de estructuras complejas. Si el genoma no informase explícitamente de la estructura general de la red, las mutaciones no podrían haber sido aplicadas con el mismo éxito. Sin la guía de los paréntesis balanceados, las reglas de mutación se habrían limitado a trabajar con grupos de conexiones no relacionados entre sí.

Como ya ha sido discutido previamente, una de las características interesantes de las SNN desarrolladas por nuestro algoritmo es el hecho de que la actividad mantenida en ellas no es cíclica. Desarrollar una red que presente actividad cíclica es trivial, basta con conectar grupos de neuronas en una estructura de anillo. Mantener una señal sin entrar en bucle es mucho más difícil. La rama de las matemáticas dedicada al estudio de este tipo de comportamiento es conocida como *Teoría del Caos*, y tiene un amplio esfuerzo investigador dedicado a ella. No hemos comprobado que el comportamiento de las redes generadas sea realmente caótico, pero así lo parece, dado que no ha sido posible encontrar patrones repetitivos. Lo que podemos asegurar es que, en nuestro caso, este comportamiento viene dado por la fisiología del potencial de membrana. Sin él, las neuronas de la red responderían siempre del mismo modo cuando recibiesen el mismo estímulo. Esto supondría la aparición de patrones bien definidos cuando cooperasen en grupos, que a su vez llevaría a un comportamiento cíclico.

7.1 Investigaciones futuras

El objetivo del proyecto ha sido cumplido satisfactoriamente, pero deja abierto el estudio en profundidad de las topologías que han sido obtenidas. Lo que correspondería

continuar investigando es qué estructuras son desarrolladas por el algoritmo, y entender por qué éstas son capaces de mostrar comportamientos emergentes complejos tales como una actividad aparentemente caótica.

La actividad de fondo mantiene las neuronas en una constante fluctuación de su potencial de membrana. Por ello, cuando un estímulo es dado a la red, éste se propaga por ella, combinándose con la señal de fondo. La salida resultante puede haber sido afectada de un modo relevante, siendo por tanto dependiente de estas fluctuaciones. Podríamos considerar un agente capaz de interactuar con su entorno (e.g. un robot) controlado por una de nuestras SNN. Su actividad de fondo determinaría el estado interno, afectando el modo en que éste reacciona a los estímulos que recibe de su entorno. Si la topología de la SNN pudiese ser adaptada mediante un proceso de aprendizaje que llevase al agente a cumplir la tarea encomendada, podría resultar en propiedades interesantes aplicables en el campo de la Inteligencia Artificial.

7.2 Otras contribuciones

Aparte de ser un trabajo de investigación, este proyecto ha supuesto también una experiencia académica. Su contribución a mi educación puede ser considerada en dos aspectos principales. Por un lado, he aplicado el conocimiento que, como estudiante, he acumulado a lo largo de mi estancia en esta universidad. Concretamente, las asignaturas que trataban sobre lenguajes formales (tanto su sintáxis como semántica) han sido especialmente útiles. En ellas, aprendimos a definir formalmente el formato y significado dado a cadenas de símbolos, centrándonos en su aplicación para el diseño de lenguajes de programación. Sin embargo, esta formalización ha probado ser igualmente efectiva para describir el lenguaje de genomas que representa grafos, y todas las funciones relacionadas con éstos. Descubrir la flexibilidad de estos métodos de formalización ha sido, en sí mismo, una experiencia instructiva. La familiaridad que he desarrollado con ellos me será sin duda de ayuda en el futuro.

Por otro lado, mientras trabajaba en este proyecto, he aprendido y desarrollado algunas capacidades que no han sido enseñadas en ninguna de las asignaturas de la carrera. Aquellas relacionadas con cuestiones técnicas han sido explicadas en los apéndices de este documento. Obtener experiencia en el uso de software para control de versiones es fundamental para cualquier ingeniero informático, y haber tenido la oportunidad de usar un supercomputador es muy valioso, al ser una importante herramienta en la disciplina científica. Además, he expandido mis conocimientos en modelos de computación, trabajando con algoritmos evolutivos y un tipo no convencional

de redes neuronales. Ambos son campos de estudio de interés que están recibiendo una creciente atención.

Este proyecto fue desarrollado no sólo para completar el grado, sino también como investigación avalada por una beca de nuestra Universidad. Dicha beca, conocida como "Beca de Iniciación a la Investigación" es concedida a estudiantes interesados en comenzar su formación como investigadores. Ha sido la motivación principal por la que este proyecto ha sido desarrollado como un estudio científico, dándome la oportunidad de experimentar por primera vez cómo se realiza la tarea de investigación. Lo que he aprendido durante el desarrollo de este proyecto junto a mi tutor, Francisco J. Vico, será sin duda relevante para mi futuro. Entender cómo distribuir tiempo y esfuerzo para ser eficiente, mantenerse fiel a los principios de la investigación y ser constante hasta obtener resultados, son algunos aspectos que no pueden ser aprendidos de ningún otro modo.

7.3 Aspectos técnicos del trabajo

Considerando que este proyecto ha sido desarrollado siguiendo un estilo científico, decidimos mantener este enfoque también al escribir este documento. Por ello, hemos dado prioridad a analizar las propiedades del algoritmo desarrollado y los resultados obtenidos, por encima de la descripción de los aspectos técnicos del trabajo. Con la intención de mantener una estructura fluida a lo largo de todo el documento, hemos decidido no incluir cuestiones técnicas en el cuerpo de la memoria. De esta manera, perseguimos dar al documento el formato de una memoria científica, donde la información relevante es el método aplicado en sí, y no cómo éste haya sido implementado. Los aspectos técnicos más relevantes han sido resumidos en los dos apéndices de este documento.

Por otro lado, no han sido necesarias las herramientas de modelado propias de la ingeniería del software, dado que no se ha desarrollado un sistema que requiera de interacciones complejas entre sus módulos. Éste está basado en cuatro tareas claramente separadas: el decodificador de genomas, la aplicación de reglas de mutación, la simulación de las redes y la evaluación de la función objetivo (*fitness function*). Cada uno recibe la salida dada por el módulo que actúa antes que él, repitiendo un ciclo iterativo hasta que el máximo número de generaciones es alcanzado. Cada uno de estos módulos fue codificado en su propio fichero, y una rutina *Main* coordina su ejecución iterativa.

Bibliography

Seamari, Y. (2016). *Spatio-temporal structure of spontaneous slow-wave oscillation and identification of Up and Down cortical states in simultaneous intra- and extracellular recordings in vivo* (Doctoral thesis).

Hodgkin, A. L., & Huxley, A. F. (1952, 08). A quantitative description of membrane current and its application to conduction and excitation in nerve. *The Journal of Physiology, 117*(4), 500-544. doi:10.1113/jphysiol.1952.sp004764

Debanne, D., Bialowas, A., & Rama, S. (2012, 11). What are the mechanisms for analogue and digital signalling in the brain? *Nature Reviews Neuroscience Nat Rev Neurosci, 14*(1), 63-69. doi:10.1038/nrn3361

Vertes, Robert P. "Hippocampal Theta Rhythm: A Tag for Short-term Memory." *Hippocampus* 15.7 (2005): 923-35. Print.

Scholarpedia, *Spike-timing dependent plasticity* (Web page). http://www.scholarpedia.org/article/Spike-timing_dependent_plasticity

GNU Octave, *Basic documentation manual* (Web page). https://www.gnu.org/software/octave/doc/v4.0.0/

GNU Octave-Forge, *Documentation of extra packages* (Web page) http://octave.sourceforge.net/docs.html

Mercurial, *Documentation* (Web page) https://selenic.com/hg/help/

Appendix A

Software used

A.1 Linux

The Ubuntu 14.04 distribution of Linux was used throughout all stages of the project. This Operating System offers a more reliable and transparent infrastructure on which to develop any serious project. The only negative aspect of it is that, not being as user-friendly as other OS, the ability to solve technical problems is often required. For a computer engineer, this actually turns to be a positive learning experience.

A.2 GNU Octave

The algorithm described in this project was programmed on GNU Octave, a programming language oriented to the implementation of numerical calculations. It is almost identical to MatLab, being considered its unofficial open-source replica.

It is a high-level interpreted language, specialized to operate with matrices. The topology of our SNNs are described through a connectivity matrix, and its state is stored in a vector (see '*A Spiking Neural Network model*' chapter). Thus, The dynamic of our networks requires an intense use of matrix operations (multiplication and updating), and simulating them is needed at each iteration to evaluate their fitness score. Thus, choosing a MatLab-like programming language was recommended. Additionally, an advantage of using this software is its highly prepared graphic tools, which were used to draw the figures present in this document.

We decided to use Octave because it is free-software. This did not only allow us to work freely without worrying about its licence, but also ensures that anyone can use our code. Besides, given the similarity between Octave and MatLab, our program could also be run in the latter with some subtle changes.

A.3 Mercurial

In order to find the best graph representation and evolutionary parameters, considering different options and testing them was necessary. This implied the implementation of many different versions. To manage all the written source code in a well-organised way, the use of a version control software was required. The one we chose was Mercurial (based on personal preference), using Bitbucket as our online storage service.

Version control software provides its user with the ability to transparently manage the historical of changes of a directory. It is based on maintaining a hidden copy of the information contained in it that can be accessed through the use of commands. It allows the user to easily shift between the different versions of the work developed in the managed directory. Through its use, it was possible to maintain the parallel development of different implementations, to test them separately and then compare them.

The different versions are not automatically stored: it is the user who decides when a version is relevant or stable enough to be included in the historical, through the use of the *commit* command. To shift between versions, the *update* command is used. If at any point the user restores a prior version and changes it, the *committed* version will generate a new *branch*, which describes a new changeset that has its own historical from that point. Additionally, the changes made to a certain branch can be applied to another one, using the *merge* command. This operation is done automatically, and if any incompatible changes are found, the user is asked to decide which to apply.

The management of the versions can be done locally, as described until this point, maintaining the historical of changes in your own computer. However, it is more appropriate to maintain this information stored in an online repository. The way the online repository and the local copy are updated is through the use of the *push* (upload) and *pull* (download) commands. This enables to manage the directory distributively, allowing its user to access it from any computer and cooperate with other developers in a common project. The repository containing the project and its whole historical can be found at https://bitbucket.org/PabloAndres/proyecto.

Appendix B

On supercomputers

The algorithm developed in this project was the final result of an iterative process of designing, implementing, testing and analyzing each of the different genome expressions, mutation rules and fitness functions we thought could be beneficial. Thus, we needed to execute the program each time a new option was considered, and not only once, but multiple times for a wide range of parameter configurations. This is the typical situation when developing evolutionary algorithms, as it is simulation-driven instead of guided by pure mathematical heuristics that can be thoroughly analyzed without executing the program.

The amount of iterations of the algorithm that are needed for it to reach the objective is around 2500 (see the chapter dedicated to the discussion of the results). Each of them requires to run a simulation of each individual SNN, of a population around 200. As these simulations are not trivial, they require a noticeable amount of time to complete, which takes roughly a second. Therefore, a complete execution of the algorithm will require around 100 hours to finish if executed in a single processor.

Fortunately, evolutionary algorithms are easily implementable in parallel architectures. Each individual of the population is independent from the rest, so it can be simulated separately. Thus, the time required to execute the program is reduced in a factor proportional to the number of processors dedicated to the task. However, the time spent in managing the processes must also be considered. The main thread has to create all child processes, assign them to the cores and recover their result. This whole procedure also takes some time and prevents the speed-up of parallelism to be as effective as dividing the time required by the amount of processors available.

The computer that was used to develop the project has a quad-core microprocessor (Intel Core 2 Quad Processor Q6600), which meant that the program would require a

couple of days to execute. That would have been a huge limitation, considering we needed to execute the program multiple times throughout its development. Consequently, we requested access to the local supercomputing node and used it to run our simulations. Thanks to it, we could run the executions in a decent amount of time (Table B.1).

Steps per simulation	50	100	200
Time required	*389 min*	*566 min*	*958 min*

TABLE B.1: Time required to execute, in 40 cores, a program with three different simulation lengths

But, even with this speed up, if we could only run a single program at a time, the development of the project would be still limited. This is due to our need to test the same program for different parameter configurations to be able to appropriately analyze it. Thus, another fundamental advantage of using a supercomputer was that we could execute different programs simultaneously, making use of a greater amount of cores.

The supercomputing node that was used is part of the Spanish Supercomputing Network and has over two thousand available cores. However, it is not advisable to run programs over more than 80 processors, as it is the maximum amount of cores in a single machine. Using more would require to manage memory shared between computers, which is extremely time consuming. Besides, requesting a whole machine to run a single job is not recommended either, as other users are also requesting the resources and rarely a 80-core machine is completely free. Therefore, to avoid having to wait for the resources to be available, it is preferable to request 40 cores and be able to run two programs in two different machines at the same time.

B.1 Running a program on a supercomputing node

In order to execute a program in the supercomputing node, first the source code has to be uploaded to its storage memory. That is done by the *sftp* protocol. Using it through a Linux terminal is as simple as starting the connection with the command '*sftp useraccount*'. Then, after providing the password associated with the account, the *get* and *put* commands can be executed, respectively downloading and uploading the files given as their argument. Each user has its own private directory where all their source codes and other files are stored.

Then, a similar procedure is done to access the supercomputing node through *ssh*. Once the authentication of the account is done, it is possible to manage your private directory, execute basic shell commands (i.e. *cd*, *mkdir*, *vi*...) and launch jobs to be executed in the supercomputing node.

The computer where the user is logged acts as an interface. In order to execute programs, a script has to be run. The script indicates the software to be used to execute the program (i.e. octave in our case), the location of the program itself and the specification of the resources requested. All files created during the execution are placed in the user's private directory.

The program to be executed must be prepared to be run in parallel. This is quite simple in Octave, as the *parallel* package offers a function that manages the execution. It asks the system to create the number of process specified, which will be distributed among the available cores. Then it assigns the execution of a given source code to each of them. The source code is the same for all processes, being the input for each one (specified in a cell array) what differs the executions. Thus, in our program, the source code specified to be executed in parallel is the mutation, decodification of the genome, and simulation of each individual. The input for each process is the genome of the different individuals in the population.