Doctoral Thesis

# Evolution of Spiking Neural Networks for Temporal Pattern Recognition and Animat Control.

*Author:*
Ahmed Abdelmotaleb

*A thesis submitted to the University of Hertfordshire in partial fulfilment of the requirements for the degree of Doctor of Philosophy*

February 2015

*"In science one tries to tell people, in such a way as to be understood by everyone, something that no one ever knew before. But in the case of poetry, it's the exact opposite!"*

Paul Dirac

# *Abstract*

**Evolution of Spiking Neural Networks for Temporal Pattern Recognition and Animat Control.**

I extended an artificial life platform called GReaNs (the name stands for Gene Regulatory evolving artificial Networks) to explore the evolutionary abilities of biologically inspired Spiking Neural Network (SNN) model. The encoding of SNNs in GReaNs was inspired by the encoding of gene regulatory networks.
As proof-of-principle, I used GReaNs to evolve SNNs to obtain a network with an output neuron which generates a predefined spike train in response to a specific input.
Temporal pattern recognition was one of the main tasks during my studies. It is widely believed that nervous systems of biological organisms use temporal patterns of inputs to encode information. The learning technique used for temporal pattern recognition is not clear yet. I studied the ability to evolve spiking networks with different numbers of interneurons in the absence and the presence of noise to recognize predefined temporal patterns of inputs. Results showed, that in the presence of noise, it was possible to evolve successful networks. However, the networks with only one interneuron were not robust to noise.
The foraging behaviour of many small animals depends mainly on their olfactory system. I explored whether it was possible to evolve SNNs able to control an agent to find food particles on 2-dimensional maps. Using firing rate encoding to encode the sensory information in the olfactory input neurons, I managed to obtain SNNs able to control an agent that could detect the position of the food particles and move toward it.
Furthermore, I did unsuccessful attempts to use GReaNs to evolve an SNN able to control an agent able to collect sound sources from one type out of several sound types. Each sound type is represented as a pattern of different frequencies.
In order to use the computational power of neuromorphic hardware, I integrated GReaNs with the SpiNNaker hardware system. Only the simulation part was carried out using SpiNNaker, but the rest steps of the genetic algorithm were done with GReaNs.

# *Acknowledgements*

I am using this opportunity to express my gratitude to everyone who supported me throughout my PhD study, specially my supervisors: Borys Wróbel, Maria Schilstra, Neil Davey, and Volker Steuber. I would like also to thank my parents for their support and my sister Noha for her help. Special thanks to my fiancée Ceyna for encouraging and supporting me. I am also grateful to my friend Adam Peszke for helping me in reviewing the English of my thesis, and to Michał Joachimczak for his help in debugging GReaNs.

The work presented in chapter 8 in this thesis was done in collaboration with Dr. Sergio Davis from University of Manchester. This work was done during CapoCaccia workshops in 2012, 2013, and 2014. I received a scholarship from Convergent Science Network of Biomimetic and Biohybrid Systems (CSN) to attend Capocaccia 2012 workshop. During Capocaccia 2013 and 2014 workshops, I was supported by both University of Hertfordshire and Foundation for Polish Science.

I would like to thank Artur Jarmolowski and Arleta Kucz from Adam Mickiewicz University in Poznan for their support and help for maintaining the logistical issues during my stay in Poland.

I used the PhD thesis of Nicolas Oros as a reference for me to follow the standard of PhD thesis of University of Hertfordshire as he was a previous PhD student there.

I used a latex template to shape up this thesis. I downloaded this template from http://www.latextemplates.com. The authors of this template are Steven Gunn and Sunil Patel. The license of this templatee can be found here.

I used Sharelatex online latex editor for my thesis which allowed me to share the thesis with my supervisors easily.

My work was supported by Adam Mickiewicz University in Poznan, Poland, by the Polish National Science Centre (projects BIOEMERGE, UMO-2011/03/B/ST6/00399, and EvoSN, UMO-2013/08/M/ST6/00922, both awarded to Dr. Borys Wróbel), and by the International PhD Programme ”From Genome to Phenotype: A Multidisciplinary Approach to Functional Genomics” (awarded by the Foundation for Polish Science, co-financed by EU Regional Development Fund—Innovative Economy Operational Programme 2007-2013) at the Adam Mickiewicz University.

# Contents

# List of Figures

# List of Tables

*To my fiancée Ceyna - the moon that brightens the sky of my life...*

# Chapter 1

# Introduction

## 1.1 Motivation and Goals

The work I have carried out during my PhD is part of broader research program to compare the computational properties of Spiking Neural Networks (SNNs) and networks that are not spiking. At the beginning of my studies, I built two SNN models: Leaky Integrate and Fire model (LIF) [1, 2] and Adaptive exponential leaky integrate and fire model (AdEx) [3] based on the previous Gene Regulatory Networks (GRNs) that have been implemented before in the artificial life platform GReaNs (the name stands for Gene Regulatory evolving artificial Networks) [4]. Mapping the GRNs to SNNs in GReaNs allowed me to use theem genetic algorithm which was already implemented before in GReaNs and used for GRNs.

Researchers showed that the brain uses temporal patterns of spikes to encode sensory information [3, 5–11]. It has been shown that temporal coding is used in vision [5], hearing [12], and olfaction [13] (more details in 2.5.1). The exact learning mechanism the brain uses for training the neurons to recognize temporal pattern is not clear yet. Hopfield [14] presented a mechanism for encoding and decoding temporal patterns of spikes. This mechanism is based on the delays of the synapses of the input neurons and a coincidence detection mechanism. Based on Hopfield's work, some studies have been done in training the delays of the input neurons synapses for temporal pattern recognition [15–18].

In contrast, I developed a learning algorithm that is based on evolving only the weights and the topology of the networks using fixed delays for the synapses. After successfully obtaining SNNs able to recognize some patterns of inputs, I investigated the behaviour of these networks and checked their robustness when noisy patterns were used for testing.

The foraging system of many animals depends on the olfactory sensory neurons that allow collecting information about the odours in the environment [19, 20]. Moths for example have a strong olfactory system on which their foraging system depend. These tiny animals with very small brain have the ability to locate chemical sources even if they are miles away from them [19, 20]. The exact way how the sensory information is encoded in animal's brain and how these sensory information is transferred for actions is not clear yet.

During my PhD studies, I investigated the different ways of encoding the sensory information in spiking neural networks and the different ways of updating the forces at the actuators based on these sensory information.

The ability of using SNNs for animat foraging was also investigated in previous work [21–25]. GReaNs platform was used before for evolving GRNs able to control animats for food foraging [26, 27]. Using the same genetic algorithm, I evolved SNNs in GReaNs to control animats for collecting food particles in 2D environment and compare their behaviour with the behaviour of the animats when GRNs were evolved.

Being able to explore the tasks of temporal pattern recognition and animat foraging in GReaNs opened the gate for me to investigate more interesting tasks by merging both tasks together. I replaced the food particles in the food foraging task with sound sources. Each sound source is represented with number of input neurons each with different frequency.

SpiNNaker [28] (Spiking Neural Network Architecture) is a massively parallel computing system which was designed to support large scale spiking neural networks simulations. I was interested in using the computational power of SpiNNaker with GReaNs. I integrated SpiNNaker with GReaNs so that the simulation step of the evolutionary algorithm is done with SpiNNaker and the rest of the steps are carried out using GReaNs.

## 1.2 Contribution to Knowledge

My main contribution to knowledge has been to extend the range of behaviour of spiking neural network models. In particular I show how the inherently temporal behaviour of a spiking neural network can be used in both the decoding of temporally coded information and in the production of temporal behaviour. This has been shown in the more detailed contributions described below:

1. I have created a biologically inspired Spiking Neural Network (SNN) model to allow the evolution of the topology of the SNNs and the weights of their synapses. I

build this model based on Gene Regulatory Network model which was implemented before in GReaNs platform. I used two popular SNN models in my work: Leaky integrate and fire model (LIF) and Adaptive exponential leaky integrate and fire model (AdEx) SNN models. Using genetic algorithm, I evolved the SNN to accomplish number of tasks including generating a predefined spike train in response to specific input, temporal pattern recognition, animat foraging, and temporal pattern recognition with animats.

2. The first main evolutionary task I have carried out during my PhD studies was evolving SNNs able to perform temporal pattern recognition. I have introduced an evolutionary algorithm based on evolving only the topology of the network and the weights of the synapses between the neurons in the network. Learning algorithms have been used before in order to obtain SNNs able to perform temporal pattern recognition [15–18]. These algorithms were based on adjusting only the delays between the neurons in the SNNs. I have used networks with various numbers of interneurons (1, 2, 5, and 10) and I compared their robustness to Gaussian noise with various standard deviation (10, 20, and 30 ms) added to the spike times of the inputs. Furthermore, I studied the different behaviours of the SNNs of two interneurons which are robust to noise and found that the positive feedback loops are important for robustness of the SNNs to noise.

3. I evolved SNNs which were able to control animats in order to perform food foraging. I have introduced various coding strategies for the sensory information represented the concentration of the food. I have also investigated three different ways of determining the thrusts at the actuators of the animats. After using these coding strategies and the ways of determining the thrusts, I compared the evolved animats based on some factors. These factors include the ability of the evolved animat to cope with high food density when it was simulated on a map which contained large numbers of targets.

4. I have investigated the ability to evolve spiking neural networks in order to control an animat able to detect and distinguish between temporal patterns emitted by simulated sound sources. The animat was supposed to effectively discriminate between different sequences of simulated acoustic signals, measure the distance to the sound source, and move towards a desired source. Althought it was not possible to evolve this animat successfully, I presented some suggestions that could lead in the future work to have an animat that could successfully do this task.

5. I have integrated the GReaNs evolutionary software with SpiNNaker system [28] in order to use its computational power to simulate large scale SNNs. The idea was to run the simulation part in the genetic algorithm with SpiNNaker while the

other steps of the genetic algorithm were carried out with GReaNs. I have used two evolutionary tasks for the integration each with different SpiNNaker board. Although the integration did not afford any improvement, this work can be seen as a contribution to theory by presenting a communication protocol between an evolutionary algorithm and a neuromorphic hardware. I also suggested some ideas in order so speed up the integration between GReaNs and SpiNNaker.

## 1.3 Structure of the Thesis

The structure of the thesis is the following:

### 1.3.1 Chapter 2

This chapter presents the literature review I have done during my PhD studies. At the beginning of the chapter, I introduced briefly the different neural networks from the first, second, and third generations. Then I explained two models of artificial neurons commonly used in SNN research and which I used in my work (Leaky Integrate and Fire (LIF) [1, 2] and Adaptive exponential leaky integrate and fire model (AdEx) [3, 29]). Later in this chapter, I presented various neural coding and how sensory and motor information can be represented in the brain using spikes.

The main two tasks I carried out during my PhD studies were evolving SNNs for temporal pattern recognition and animat foraging. In the last part of the chapter, I reviewed the work that have been done previously in these two tasks.

### 1.3.2 Chapter 3

The third chapter describes the GReaNs platform. This platform was used before to evolve gene regulatory networks (GRN) to perform some tasks including controlling multicellular development in three dimensions [30–32], processing signals [33] and controlling animats [26, 27].

At the beginning of the chapter I describe how the GRN is constructed from the genome and the steps of the genetic algorithm which is used to evolve the GRNs. The third chapter includes also a description for the first task I carried on during my PhD study of mapping the GRN in GReaNs to spiking neural network (SNN). In this task I implemented two different SNN models, Leaky Integrate and Fire model (LIF) [2] and Adaptive exponential leaky integrate and fire model (AdEx) [3, 29].

Finally I validated the behaviour of a simple SNN in GReaNs by comparing its behaviour with the behaviour of the same SNN simulated using PyNN package [34] with Brian simulator [35] as a back-end simulator.

### 1.3.3 Chapter 4

This chapter presents the first evolutionary task I have tried in GReaNs after mapping the GRN to SNN successfully. This task includes evolving SNNs to obtain a network able to generate a predefined spike train in response to specific input. This task can be divided to two main parts. First, obtain LIF (or AdEx) SNN able to generate the same spike train (shifted by 5ms, 10ms, or 20ms) generated by a single AdEx (or LIF) neuron when both the SNN and the single neuron are connected to the same input.

The second part consists of obtaining LIF (or AdEx) SNN able to double (generate two spikes 5 ms and 25 ms after each spike) the spike train generated by a single AdEx (or LIF) neuron when both the SNN and the single neuron are connected to the same input. Finally, I tested the generalization of the final networks obtained from the previous two sub-tasks by comparing the behaviour of the single neuron and the evolved SNN when both of them are connected to a different input other than the one which was used during the evolution.

### 1.3.4 Chapter 5

The temporal pattern recognition task which is one of the two main tasks I did during my studies is presented in this chapter. In this chapter I studied the ability to evolve spiking networks, with varying numbers of LIF neurons (1, 2, 5, 10, and unlimited) to recognize predefined temporal patterns of different number of inputs (3 and 4). Then I tested the behaviour of the final networks when Gaussian noise with different standard deviations (10, 20, and 30 ms) was applied to the time of the spike in each input.

Furthermore, I used the noisy inputs during the evolution and checked if I could get final networks without error. It was interesting then to study the difference between the structure of the final networks (with only 2 interneurons) which were evolved in absence of noise and the final networks (with 2 interneurons also) which were evolved with presence of noise.

### 1.3.5 Chapter 6

After evolving SNNs for generating predefined spike trains and for temporal pattern recognition, I explored a more practical task that is relevant for evolutionary robotics. This task includes evolving SNNs for real time control of foraging behaviours. This task is introduced in this chapter. First, I introduce the the model I used of the animat and the description of the simulation environment; this includes the structure of the animat and the food resources on the map. The next section describes the genetic algorithm used in this task.

One of the most important issues covered in this chapter is the various strategies of encoding the sensory information in the SNN. Finally, I introduce the results and discuss about the methods I applied in order to improve the behaviour of the animat.

### 1.3.6 Chapter 7

This chapter presents the description of the integration of the temporal pattern recognition task and the real time control of foraging behaviours task. The description includes the structure of the animat, and the simulation environment. Finally, I introduce the results of the experiments and discuss these results.

### 1.3.7 Chapter 8

The integration work I have done with SpiNNaker [28] (**Spi**king **N**eural **N**etwork **A**rchitecture) is presented in this chapter. SpiNNaker is a massively parallel computing system. Due to its high computational power, it can afford real-time simulation for a large scale SNNs with thousands of neurons. In this chapter, I introduce two versions of integration I have done with two different SpiNNaker boards.

In the first version, the task of evolving LIF SNNs to match the spike train of a single AdEx neuron shifted by 5 ms described in chapter 4 was presented using the small SpiNNaker board with only four chips. In the second version, I used the bigger SpiNNaker board with 48 chips to integrate with GReaNs. The task of evolving SNNs with 10 interneurons for temporal pattern recognition in the presence of noise was explored in this version. Finally, I compare in terms of computational time the results of the two tasks with the results of the same tasks when SpiNNaker boards were not used.

# Chapter 2

# Literature Review

## 2.1 Central Nervous System

The nervous system is responsible for controlling all the body parts and for communications between them. The nervous system consists of two main parts, the central nervous system which includes the brain and the spinal cord and the peripheral nervous system which includes sensory and motor neurons responsible for connecting the central nervous system with all body parts and the environment.

The spinal cord is a long and thin bundle of nervous tissue that resides in the vertebral cavity. It is responsible for a large part of the communication between the brain and the rest of the body. On the other hand, the brain, which resides in the head, is considered the most complex organ in the human body. It is responsible mainly for processing and analysing the information it receives from the sensory neurons located in the peripheral nervous system and taking decision based on it. Most of the actual information processing takes place in the cerebral cortex which plays a very important rule in consciousness, memory, attention, perception, and language.

The structure and the function of the brain have been studied by many experimentalists and theoretical neurobiologists over the past hundred years. These studies showed that the size of the brain varies a lot between animals. African elephant brain contains 257 billion neurons which makes it the animal with the largest counted number of neurons [36]. Most of the neurons in the elephant brain are located in the cerebellum [36]. Sperm whale brain is about two times larger than elephant brain, but the exact number of neurons in the whale brain is not known yet. On the other hand, the predatory rotifer *Asplanchna brightwellii* brain contains only 200 neurons [37].

Human brain has around 80 billions [38] of neurons (Fig. 2.1). These neurons are considered the elementary processing units in the nervous system [39]. Neurons connect to each other in an efficient way to process the incoming signals in order to make decisions and control movements. Neurons communicate with each other with short electrical pulses.

## 2.2 Neuron

Fig. 2.1 shows the biological neuron: for a simplified view, each neuron receives most of the incoming signals through their dendrites, these signals change the voltage (also called membrane potential) of this neuron. When the voltage reaches the value of the threshold, the neuron generates a spike or an action potential which is sent to the connected neurons through the axon. The effect of the presynaptic neurons on the postsynaptic neurons depends mainly on the type of the presynaptic neuron (excitatory or inhibitory) and the strength of the synapses between them.

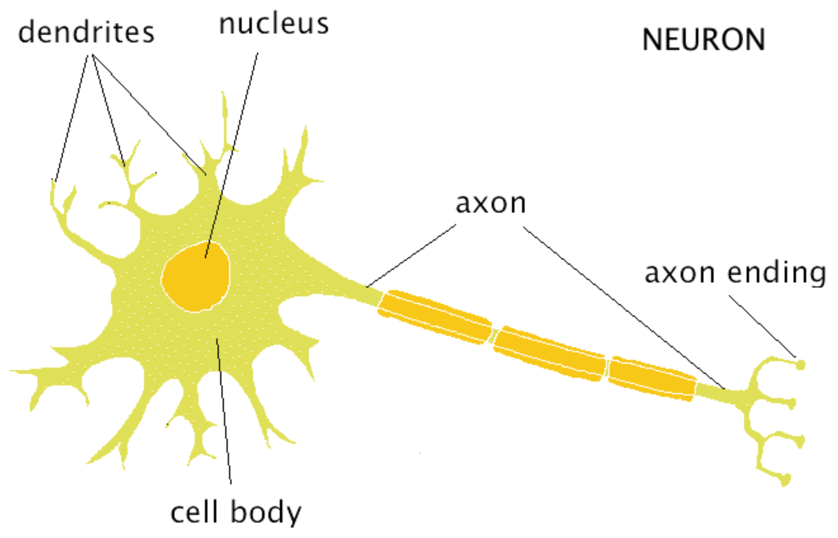


FIGURE 2.1: A simplified structure of a biological neuron. At the middle of the cell body of the neuron resides the nucleus which contains the genetic material. Each neuron receives input spikes from other neurons through its dendrites and cell body. At the output stage, the neuron uses its axon to send spikes to all connected neurons. From http://webspace.ship.edu/cgboer/theneuron.html.

The action potential (Fig. 2.2) generated by a neuron is a fast depolarization resulted from opening of the two ion channels (sodium Na+ and potassium K+) of the neuron. When the membrane potential reaches the maximum value, the membrane is repolarized until the potential reaches the minimum value during the refractory period in which the neuron cannot fires more spikes.

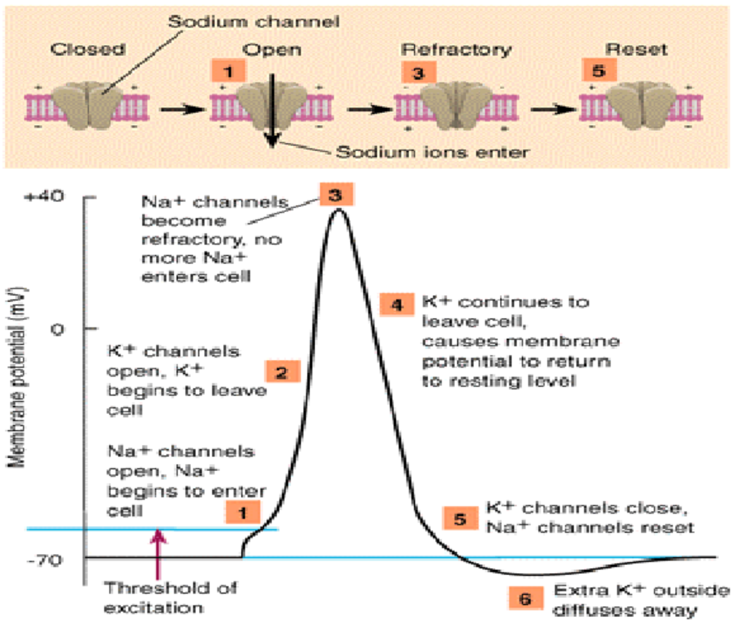

Sodium channel
Closed
Open
Refractory
Reset
1
3
5
Sodium ions enter
+40
0
-70
Membrane potential (mV)
3
Na+ channels become refractory, no more Na+ enters cell
4
K+ continues to leave cell, causes membrane potential to return to resting level
2
K+ channels open, K+ begins to leave cell
1
Na+ channels open, Na+ begins to enter cell
5
K+ channels close, Na+ channels reset
Threshold of excitation
6
Extra K+ outside diffuses away

FIGURE 2.2: A schematic diagram for a spike generated by a neuron. When the membrane potential of a neuron reaches the threshold (1), its ion channels open leading for membrane depolarization (2). When the membrane potential reaches the maximum value (3), the membrane is repolarized and the potential decreases till it reaches the rest potential (5). The potential continue decreasing until it reaches the minimum value during the repfractory period (6). From http://www.bazaarmodel.net/Onderwerpen/neuron/ps2lec1.htm.

The spike is transferred from the presynaptic neuron to the postsynaptic neuron through the synapse between the two neurons. The synapse (Fig. 2.3) is a microscopic gap that lies between the axon of the presynaptic neuron and the dendrites of the postsynaptic neuron. A series of chemical events occur during the transferring of the signal through the synapse. These events include releasing of neurotransmitters (chemical substances) from the presynaptic neuron and receiving them by receptor sites in the postsynaptic neuron.

## 2.3 Spiking Neural Networks

Modeling neural systems has passed a number of generations of research, starting from McCulloch-Pitts threshold neurons [40] which are considered the first generation of artificial neural networks. This model simply considers the neuron as a digital element which sends a binary signal if the sum of the incoming signals, scaled by their weights,

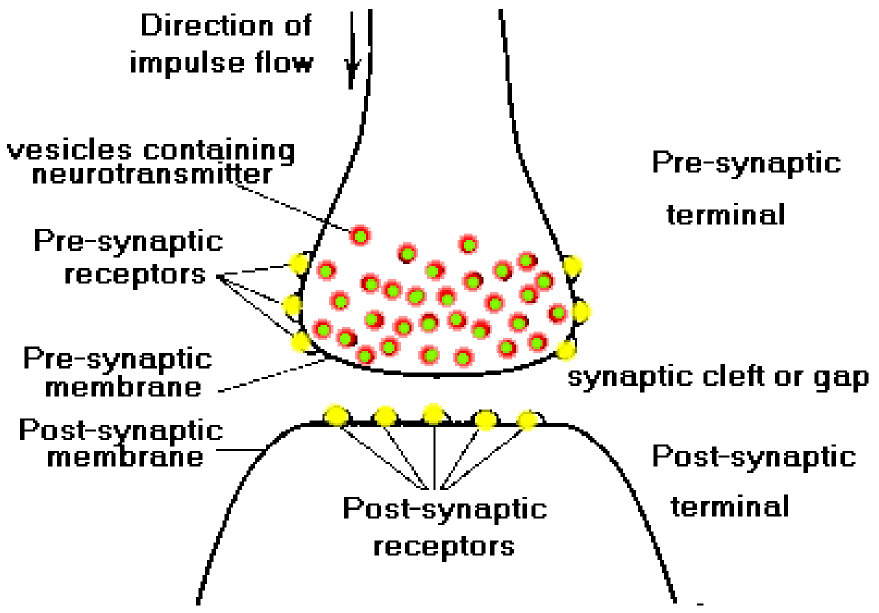


FIGURE 2.3: An example of a synapse between two neurons. The transfer of the spike from the axon of the presynaptic neuron (upper part) to the dendrite of the postsynaptic neuron (lower part) includes transmission of neurotransmitters to the posysynaptic receptors. From http://www.bazaarmodel.net/Onderwerpen/neuron/ps2lec1.htm.

crosses the value of the threshold (Fig. 2.4). This model have been applied on many artificial neural networks, for example, multi-layer perceptrons and Hopfield networks [41].

McCulloch-Pitts's model has been modified in the second generation by replacing the threshold by a continuous activation function (usually sigmoid or hyperbolic tangent [42]) which allows for analog inputs and outputs. The output of the neuron in this network lies in the range [0, 1] when the sigmoid function is used while it lies in the range [-1, 1] when the hyperbolic function is used. Second generation neural networks are considered more powerful as they can be used for analog input and output.

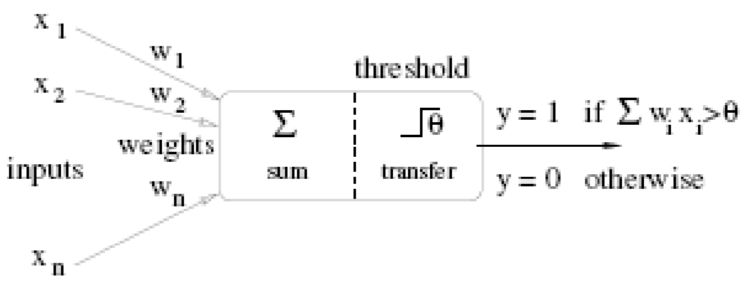


FIGURE 2.4: The first generation model of an artificial neuron. In this model the neuron works as a digital element which sends a binary signal if the sum of the incoming signals $(x_1, ..., x_n)$, scaled by their weights $(w_1, ..., w_n)$, crosses the value of the threshold $(\theta)$. Taken from [43].

The use of spikes (Fig. 2.5) appeared in the third generation of neural models, which is considered even more biologically realistic. Using spikes allows the model neurons in these spiking neural networks (SNNs) [39, 44–46] to communicate with each other using single pulses like real neurons. This facilitates the representation of time in the model.

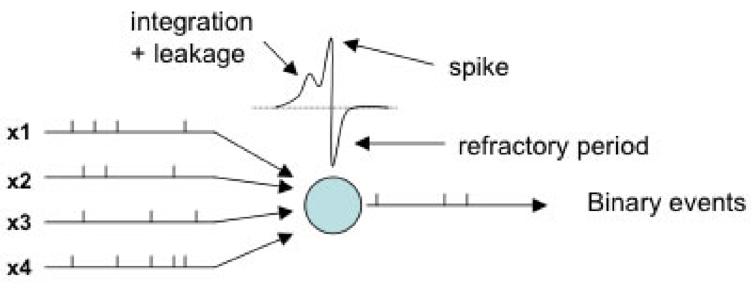


FIGURE 2.5: An example of a spike generated by a neuron. The neuron receives inputs from the connected neurons ($x_1, ..., x_4$) which allows the membrane potential to integrate until it reaches the threshold voltage when the neuron fires a spike. After firing a spike the membrane potential is set to the reset voltage. The neuron needs to wait for a period of time until it can fire another spike (refractory period). Taken from http://lis2.epfl.ch/CompletedResearchProjects/EvolutionOfAdaptiveSpikingCircuits/.

### 2.3.1 Leaky Integrate-and-Fire neural model

The Leaky Integrate-and-Fire (LIF) model [1, 2] is the simplest and most widely used spiking neuron model. In this model, a neuron is represented by a basic electrical circuit (Fig. 2.6). As we can see in the circuit on the right-hand side, an input current $I(t)$ charges a capacitor and flows across a resistor, which are arranged in parallel.

$$I(t) = I(C) + I(R) \tag{2.1}$$

where $I(C)$ is the current which charges the capacitor, and $I(R)$ is the current through the resistor. From Ohm's law:

$$I(R) = \frac{V}{R} \tag{2.2}$$

where $V$ is the voltage over the resistor.
From the definition of the capacity:

$$I(C) = C\frac{dV}{dt} \tag{2.3}$$

Now we can rewrite the equation as follows:

$$I(t) = \frac{V(t)}{R} + C\frac{dV}{dt} \tag{2.4}$$

We can introduce a new constant called membrane time constant:

$$\tau_m = RC \tag{2.5}$$

then we can write the previous equation as following:

$$\tau_m \frac{dV}{dt} = -V(t) + RI(t) \tag{2.6}$$

which is the general equation of the membrane potential in the LIF model. When the membrane voltage of a neuron reaches the value of threshold $\theta$, the neuron generates a spike, and the value of the membrane potential is reset to $V_r$. Often, a short refractory period where the neuron is unresponsive is included by clamping the membrane potential to $V_r$ for the duration of a few milliseconds.

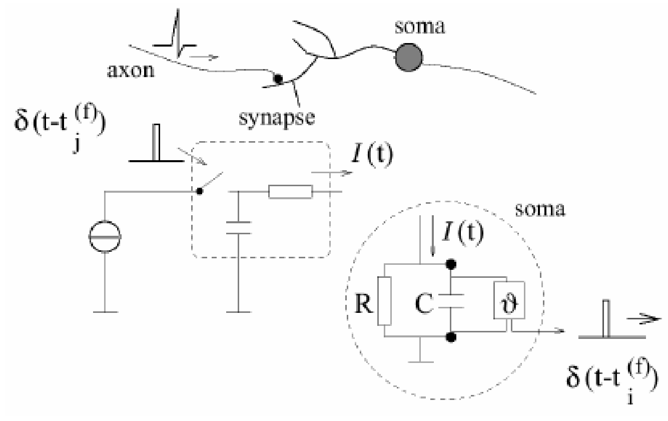


Figure 2.6: Schematic diagram of the LIF model. On the left side, a presynaptic spike arrives at the synapse. A low-pass filter is used to convert the pulse $\delta$ to input current I(t). On the right side, the basic circuit of the neuron which shows the current $I(t)$ charges the RC circuit. If the voltage crosses the value of $\theta$ at time $t_i^{(f)}$, a spike $\delta(t-t_i^{(f)})$ is fired. Taken from [43].

### 2.3.2 Non-Linear Integrate-and-Fire models

The LIF model is too simple to produce many behaviours observed in real neurons. For example, behaviours that are not captured by standard LIF models include bursting and spike rate adaptation, that is, the gradual reduction of spike rate over time. Adding an adaptation variable and a non-linearity can make Integrate-and-Fire models more biologically realistic. The adaptation variable allows for the production of spiking and bursting behaviour of known types of cortical neurons (regular spiking, adapting, delayed spike initiation, bursting, initial bursting, and fast spiking) [47]. The general equations of such models are as following:

$$\frac{dV}{dt} = F(V) - w + I \tag{2.7}$$

$$\frac{dw}{dt} = a(bV - w) \tag{2.8}$$

where $w$ is the adaptation variable, a and b are constants, and the function $F(V)$ varies from model to model.

- Adaptive Exponential (AdEx) LIF Model [3, 29] is one of the non-linear integrate-and-fire models that uses an exponential function (for example $exp(\frac{V-V_t}{\delta})$ where $V$ is the membrane potential, $V_t$ is the threshold potential, and $\delta$ is the slope factor). Both the membrane voltage $V$ and the adaptation variable w are reset when the neuron fires a spike (Fig. 2.7).

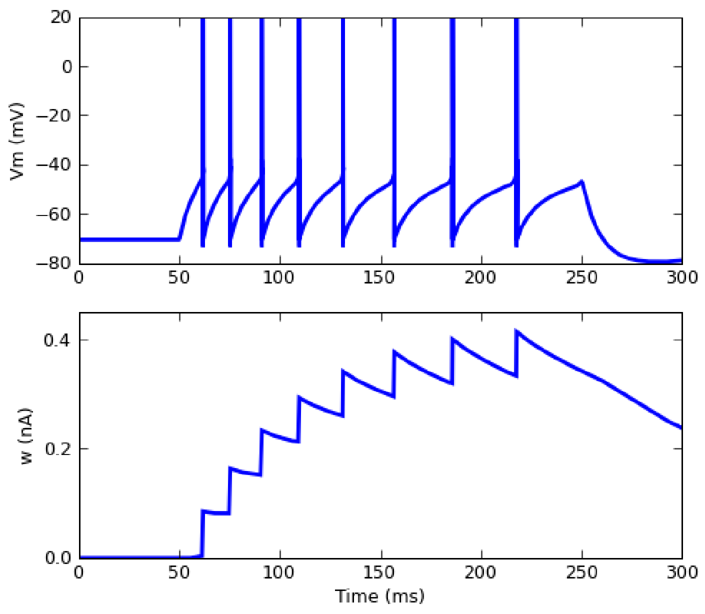


Figure 2.7: The temporal evolution of the membrane potential (top) and the adaptation variable (bottom) in an AdEx model. When the AdEx neuron fires a spike, the value of the adaptation variable is increased by a constant value which decreases the activation of the neuron. Take from [47].

- Izhikevich model [48] is also considered non-linear integrate-and-fie model. This model uses a quadratic function ($xV^2 + yV + z$, where $V$ is the membrane potential and $x$, $y$, and $z$ are constants) for $F(V)$ in Equ. 2.7.

### 2.3.3 Hodgkin and Huxley model

In 1952, Alan Lloyd Hodgkin and Andrew Huxley presented a neural model when they performed some experiments on the giant axon of the squid [49–53]. Their model is based on three different ionic currents, sodium (Na), potassium (K), and a leak current (L). These ionic currents charge the capacitor as following:

$$C\frac{dV}{dt} = -\Sigma_j I_t(j) + I(t) \tag{2.9}$$

where $C$ is the capacitance, $V$ is the membrane potential, $\Sigma_j I_t(j)$ is the sum of the current from all the ionic channels, and $I(t)$ is the injected current.

The current from all the ionic channels ($\Sigma_j I_t(j)$) can be calculated using the following equation:

$$\Sigma_j I_t(j) = g_{Na} m^3 h(V - E_{Na}) + g_K n^4 (V - E_K) + g_L(V - E_L) \tag{2.10}$$

where $g_{Na}$, $g_K$, $g_L$ are constanct conductances, $E_{Na}$, $E_K$, $E_L$ are reversal potentials, and $m$, $n$, and $h$ are called gating variables which are used for activation and deactivation and caclulated using following equations:

$$\frac{dm}{dt} = \alpha_m V(1 - m) - \beta_m V m \tag{2.11}$$

$$\frac{dn}{dt} = \alpha_n V(1 - n) - \beta_n V n \tag{2.12}$$

$$\frac{dh}{dt} = \alpha_h V(1 - h) - \beta_h V h \tag{2.13}$$

where $\alpha$ and $\beta$ are voltage dependent rate constants.

The Hodgkin and Huxley model is not suitable for large number of neurons simulation and for real time simulations as it is very expensive model to implement. To evaluate 0.1

ms of model time using Hodgkin and Huxley model, it takes 120 floating point operations [54].

On the other hand, adaptive leaky integrate and fire models (AdEx and Izhikevich models) allow to adequately simulate the behaviour of cortical neurons [54].

I used only LIF and AdEx models during my studies as both of them are supported in SpiNNaker [28] system with which I was planning to integrate my work.

### 2.3.4 Simulation of Spiking Neural Networks

Simulating SNNs is a field that attracted many researchers and engineers. Many software tools have been implemented for simulating SNNs on personal computers. These tools include the simulators Brian [35], Nest [55], and Neuron [56].

Dedicated hardware systems have been also used for SNN simulation. Hardware simulators can provide real time simulations of SNNs and consume less energy. Based on the approach used for the implementation of the neural models, the hardware simulators can be divided into analog (for example, Neurogrid [57]) and digital (for example, SpiNNaker [28]) hardware simulators.

Analog hardware simulators consume less energy and take less area. It has been shown [58] that analog simulators consume 20 times less energy than digital simulators while they take 5 times area less than digital simulators. On the other hand, the digital hardware simulators are less noisy which make them not sensitive to process variability.

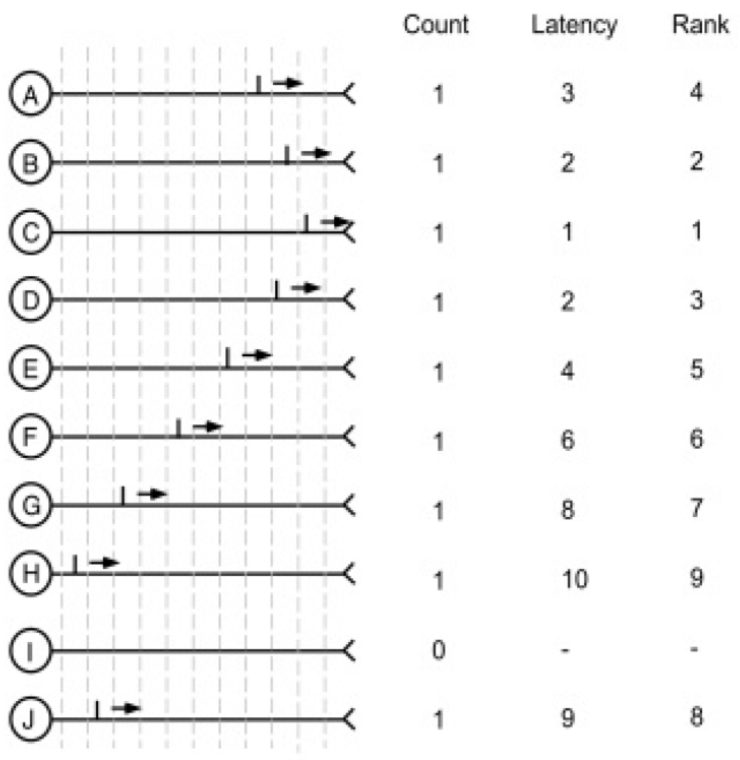
Using all of these software and hardware tools will require writing different scripts for each tool to define the structure and the settings of the SNNs. To make the simulation of SNNs on different simulation tools, PyNN [34] has been developed. PyNN is a simulator-independent platform for building neuronal network models. Using PyNN, the network structure can be described in the Python programming language. PyNN also allows choosing which simulator back-end to be used during the simulation. Both software simulators and hardware simulators can be used as a back-end simulator for PyNN.

## 2.4 Neural coding

The previous section provided a brief overview about some types of neuronal models, and gave examples of simple spiking neuron models that can be used to study the spike-based representation of information in neural networks. The current section explores how sensory and motor information can be represented in the brain using spikes. This work was presented originally in [59].

### 2.4.1 Rate code

A firing rate code is the simplest and most commonly used form of information transmission between neurons. This model is based on considering the sensory neurons as analog-to-frequency converters as the intensity of the stimulus is mapped onto the firing rate, with high stimulus intensity mapped onto high firing rate and low intensity mapped onto low firing rate [59].



| | Count | Latency | Rank |
|---|---|---|---|
| A | 1 | 3 | 4 |
| B | 1 | 2 | 2 |
| C | 1 | 1 | 1 |
| D | 1 | 2 | 3 |
| E | 1 | 4 | 5 |
| F | 1 | 6 | 6 |
| G | 1 | 8 | 7 |
| H | 1 | 10 | 9 |
| I | 0 | - | - |
| J | 1 | 9 | 8 |

FIGURE 2.8: The difference between count, latency, and rank coding schemes for 10 neurons over a time window of 10 ms. Each neuron can generate only one spike. In this simple example, there are (10 + 1) possible states with a count code, while by using a latency code we can get $10^{10}$ states. Finally, there are 10! possible states using a rank code. Taken from [59]

### 2.4.2 Population rate code

This code is a special case of a rate code that is based on counting the number of spikes generated by a number of neurons during a specific time window. The advantage of a population rate code compared to a rate code that relies on a single neuron is that the time window that is required to count spikes is smaller. Fig. 2.8 illustrates a population rate code (here called count code) that operates based on a single spike per neuron. In this particular example there are nine spikes generated during a window of 10 ms, so the population frequency of this population of $N = 10$ neurons is $90s^{-1}$. In order to compare this code with other codes, the amount of data (in bits) that can be represented by this code will be calculated by taking the logarithmic value with base 2 of the possible number of states that can be represented with each code. There are $(N + 1)$ possible states for the count code of this population during this window (from 0 to $N$ spikes). In which means that the maximum amount of data that can be transferred using this code is $log_2(N + 1)$ bits.

### 2.4.3 Binary code

During the observation window, each neuron could either fire one spike or keep silent, so each neuron could be seen as a line in a ten-line digital cable. As we can see in Fig. 2.8, the current state of the population could be described by the binary sequence 1111111101, and the total amount of information that could be transmitted would be $log_2(2^{10})$ bits or in general for $N$ neurons $log_2(2^N)$ bits.

### 2.4.4 Latency code

The latency or timing code is one of the most efficient codes as it is based on the precise timing of the spikes of each neuron. The middle column in Fig. 2.8 shows the latency code of each neuron. Since the observing window is 10 ms, the latency code can take any of the values from 1 to 10 or null. The amount of information that can be transmitted using this code depends on the precision of the determination of the time of each spike. Using a precision of 1 ms, the maximum amount of information that could be transmitted in the observation window $(t)$ is $log_2(t)^N$ bits, where $N$ is the number of neurons. However, although efficient, latency codes are very sensitive to temporal noise in the spike trains.

For a population of neurons, the relative latency code of each of them can be interpret the spiking patterns generated by this population.

### 2.4.5 Rank order code

Instead of looking at the exact timing of the spikes, this code is based on the order in which the neurons fire spikes, thereby addressing the problem of the sensitivity of a latency code to noise. For the population in Fig. 2.8 the order C-B-D-A-E-F-G-J-H-I is transmitted. There are $N!$ different orders that can be generated by $N$ neurons which makes the total amount of information that could be transferred using $N$ neurons $log_2(N!)$ bits.

## 2.5 Temporal Pattern Recognition with Spiking Neural Networks

### 2.5.1 Introduction

In the $20^{th}$ century, it was widely believed that neurons in the brain use firing rate (described in 2.2.1) to encode their sensory information (for example, [60, 61]). One of the early and leading studies in neural coding showed that there is a strong relation between the firing rate of the stretch receptor neurons in the muscles and the force applied to the muscle [62, 63]. Based on these studies, the firing rate coding was used to describe the properties of different sensory neurons in response to various actions. It was used to describe the modality and topographical attributes of cat's cortex neurons in response to different actions including movement of hairs and pressure upon the skin [60]. Moreover, the firing rate was used to describe the properties of anesthetized cat's cortical neurons in response to stimulating its retina separately or simultaneously with light spots of various sizes and shapes [61].

By the end of the $20^{th}$ century, many studies have been made criticizing the ability of firing rate to encode all sensory information. Temporal pattern code was presented as an alternative [3, 5–11]. For example, one of these studies [5] criticized the ability of the firing rate coding to be used in the movement-sensitive neuron in the visual system of the blowfly. The reason behind that is that the course correction of the blowfly takes only 30 ms while the firing rate of the movement-sensitive neuron is in the range between 100 to 200 $s^{-1}$. This constraint limits the number of spikes generated by the movement-sensitive neurons during the course correction of the blowfly to an extent that makes using firing rate code not appropriate.

Other studies [9] proposed that cortical neurons use more than one form of neural coding. For single neurons both firing rate code and temporal structure of the spike trains are used. For large population of neurons, both the population coding and the temporal coding can be used. As I mentioned in the previous section, the population code is considered a special case of the firing rate code which involves counting the spikes generated by a population of neurons. The coordinated-coding uses the relationship between the signals from the neurons in this population to represent the messages in the cortical neurons. This relationship could be the order of the spikes generated by each neuron in the population.

Thorpe and colleagues argued that human brain can recognize 3D objects in less than 400 ms which makes it impossible for the straight forward firing rate code to be used for processing information for vision without using the exact time of spikes [64]. Furthermore, it has been shown that temporal coding is used for processing information for hearing [12] and olfaction [13].

### 2.5.2 Encoding and decoding mechanism for Temporal Pattern Recognition

As we explored in the previous section that it is widely accepted that the brain uses temporal pattern of inputs to encode sensory information. Hopfield suggested an encoding and decoding mechanism for temporal pattern recognition [14]. His mechanism was based on the synaptic delays of the inputs with the temporal patterns, then detecting the coincidences between these inputs (Fig. 2.9). He suggested that radial basis function can be used by the decoding neurons for recognizing specific temporal patterns.

### 2.5.3 Training SNNs for Temporal Pattern Recognition

Delays play a crucial rule in the mechanism suggested by Hopfield. If his mechanism is true, then there must be a learning mechanism for these delays in the brain. Natschläger and Ruf have suggested a structure of a Spiking Neural Networks used for temporal pattern recognition [15]. They also proposed a learning algorithm for the synaptic delays that allows obtaining a network able to differentiate between different temporal patterns.

The network Natschläger and Ruf have presented (Fig. 2.10) contains two layers. The input neuron layer contains all the inputs of the network ($u_1$ to $u_m$). Each input ($u_i$) fires only one spike at time $x_i$ during a total interval $T$. The second layer is the output layers which contains the output neurons ($v_1$ to $v_n$). Since the main job of the output neurons is to calculate a radial basis function (RBF), output neurons are called RBF neurons.

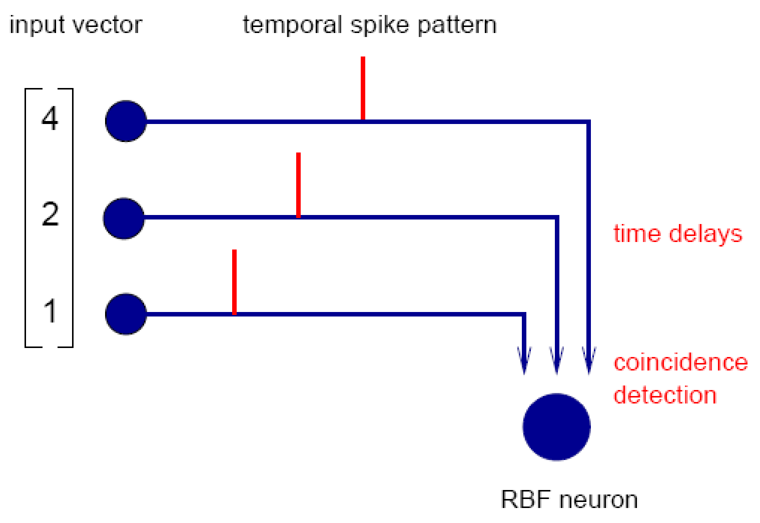


FIGURE 2.9: Hopfield decoding mechanism. The mechanism that was suggested by Hopfield includes two aspects. First, each of the synapse between the inputs and the decoding neuron has delay, then coincidences of the input patterns is detected using radial basis function (RBF) in the decoding neuron which is called RBF neuron. Taken from [65]

RBF neuron is a neuron that spikes only if it observes the same input pattern that the neuron was encoded with.

Each input neurons is connected with RBF neuron by a synapses with weight $w_{ij}$ and delay $d_{ij}$. If the spikes from all input neurons arrive at any RBF neuron at the same time, this will let the RBF neurons fire a spike which will inhibit the other RBF neurons before this RBF neuron inhibits itself. Natschläger and Ruf have used leaky-integrate-and-fire model to model the RBF neurons.

Natschläger and Ruf used an unsupervised learning algorithm for the RBF neurons to be able to cluster input patterns. The idea of their algorithm is to allow each input neuron $u_i$ to be connected with each RBF neuron $v_j$ with multiple synapses each with different weight $w_{ij}^{(k)}$ and delay $d_{ij}^{(k)}$. The values of the weights and delays are initially chosen randomly from predefined ranges. Each RBF neurons $v_j$ should receive at least one spike from all inputs before it spikes. The idea of the algorithm is that each RBF neuron rewards the synapses which drive it to spike and punish the other synapses. The rewarding and punishment mechanism is performed by allowing each RBF neuron to propagate spikes back through its synapses when it fires a spike. Based on the difference between the presynaptic and postsynaptc spikes times each synapse is rewarded (punished) by increasing (decreasing) the synaptic weight respectively. In case that the

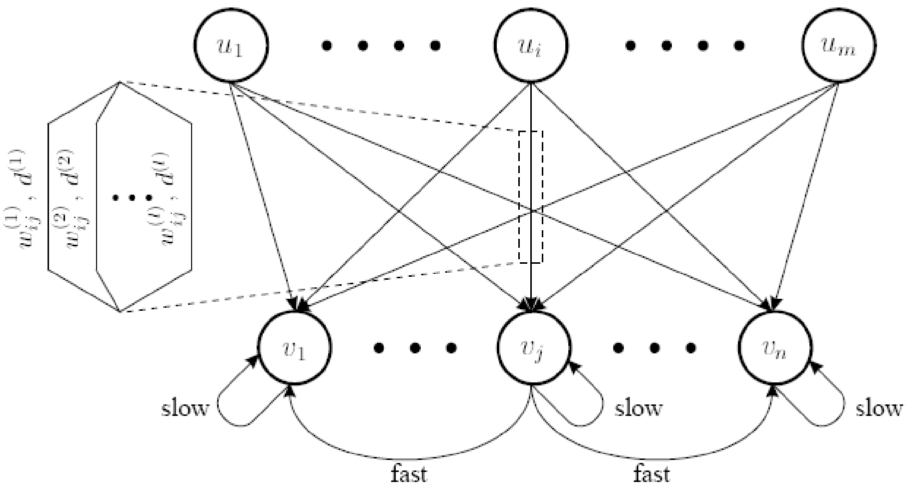


FIGURE 2.10: The structure of Natschläger and Ruf network. Each input neuron $u_i$ is connected with each output neuron $v_j$ by a synapse with weight $w_{ij}$ and delay $d_{ij}$. When the output neuron fires a spike it inhibits the other output neurons and then it inhibits itself.Taken from [15]

difference between the presynaptic and postsynaptc spikes times is small, the synapse is rewarded and vise verse.

Steuber and colleagues also showed that decoding of temporal parallel fibre input patterns can be implemented in a multi-compartmental model of a cerebellar Purkinje cell [16–18, 66]. They used a non-hebbian learning algorithm for training the synaptic delays between the neurons in the network. They used a biochemical mechanism for adapting the synaptic delays. Adapting the synaptic delay was modelled by adapting the latencies of calcium responses after activation of metabotropic glutamate receptors.

## 2.6 Animat Foraging with Spiking Neural Networks

### 2.6.1 Introduction

Many animals depend on their olfactory system for foraging [19]. The foraging system relies on the olfactory sensory neurons to collect information about odours in the environment. The information collected by the sensory neurons is encoded as spikes and sent to the brain through the axons of the sensory neurons [67]. Many coding strategies have been proposed in the olfactory system to encode the information collected from the environment. These coding strategies include the firing rate, the number of the active sensory neurons and the synchronization of firing between the sensory neurons [19]. Recently, Oros and collaborators have investigated the ability of evolving SNNs to control animats for foraging [22]. In the following sections I will cover in more detail the work by Oros and colleagues on controlling agents.

### 2.6.2 The model

A network of simple LIF models was used to control the agent. In this model the membrane potential of every neuron was updated every time step (0.1 $ms$ was used) based on the following equation:

$$\frac{dV}{dt} = -\frac{V}{\tau_m} + \sum_{j=1}^{n} I_j W_j \tag{2.14}$$

where $V$ is the membrane potential of the neuron, $\tau_m$ is the membrane time constant, $n$ is the number of synapses, $I_j$ is the current received from synapse number $j$, and $W_j$ is the weight of this synapse (in units of 1/F). During the experiments, the value of the membrane time constant ($\tau_m$) was set to 50 ms.
The resting potential was set to 0 mV, the threshold $\theta$ was set to 20 mV, and a refractory period of 3 ms was used. After a neuron fires a spike, the synaptic current of all postsynaptic target neurons is given by the following equation:

$$I_j(t) = \left(\frac{t - (t_{spike} + delay)}{\tau_s}\right) exp\left(\frac{1 - (t - (t_{spike} + delay))}{\tau_s}\right) \tag{2.15}$$

where $t_{spike}$ is the time when the presynaptic neuron fired the spike, $delay$ is the conduction delay between the neuron which fired the spike and the neuron which received it, and $\tau_s$ is the synaptic time constant. The synaptic time constant ($\tau_s$) was set to 2 ms. The $delay$ is calculated with the following function:

$$delay = coeff_{delay} \times distance \tag{2.16}$$

where $coeff_{delay}$ is the delay coefficient and $distance$ is the distance between the neuron which fired the spike and the neuron which received it. $coeff_{delay} = 5 \times 10^{-5}$ was used.

### 2.6.3 The agent

Oros and colleagues used an agent similar to a Braitenberg vehicle [68]. In this model (Fig. 2.11) the agent has two wheels, one on the right side and one on the left side;

each of them is controlled by two motor neurons. Each wheel can move forward or backward. One motor neuron supports forward movement and another motor neuron supports backward movement.

On the front of the agent, there are two antennae. Each of these antennae is connected to a sensory neuron. The distance between the two antennae was long enough in order to allow a large difference in the chemical concentration. In the absence of any chemical concentration, the agent will be still be able to move forward thanks to adding a baseline input current (0.5 A/F) to the forward motor neurons. Every 10ms, the velocity of the agent is updated by calculating the difference between the firing rate of the forward motor neurons and the backward motor neurons as following:

$$V_w = K_v \left( \frac{S_{forward} - S_{backward}}{t_{period}} \right) \tag{2.17}$$

where $V_w$ is the velocity of each wheel, $K_v$ is a constant, $t_{period}$ is the period after which the velocity was updated, $S_{forward}$ is the number of spikes fired by the forward motor neuron during $t_{period}$ and $S_{backward}$ is the number of spikes fired by the backward motor neuron during the same period $t_{period}$. $K_v = 0.3$ and $t_{period} = 10$ms were used during the experiments.

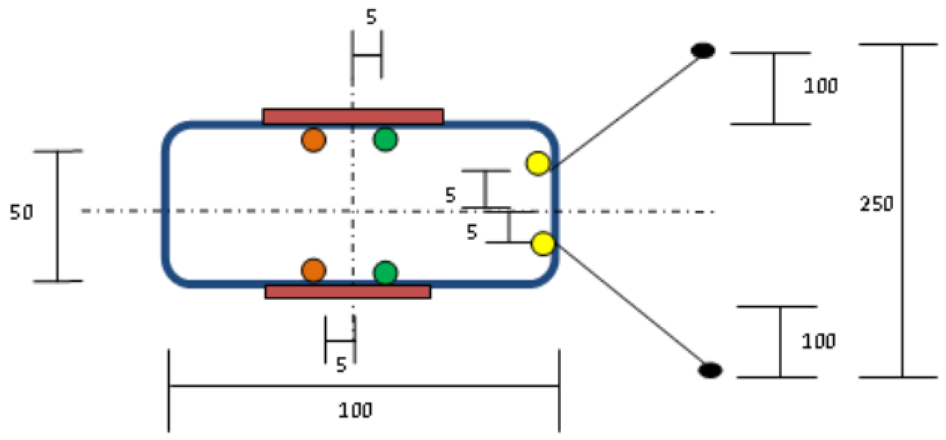


FIGURE 2.11: The agent model used by Oros and colleagues. Two long antennae (black) were connected to sensors (yellow) in order to detect the chemicals and two wheels controlled by 4 neurons 2 of them were responsible for forward movement (green) and the other 2 were responsible for the backward movement (orange). The numbers on the figure represent the dimensions of the agent. Take from [25].

### 2.6.4 The environment

Oros and colleagues used a 2-dimensional map to simulate the world. Only two chemicals were placed on the map. Each chemical was represented as a circle of concentration where the maximum concentration is at the centre of the circle and the concentration gradually decreases with the distance from the centre (Fig. 2.12).

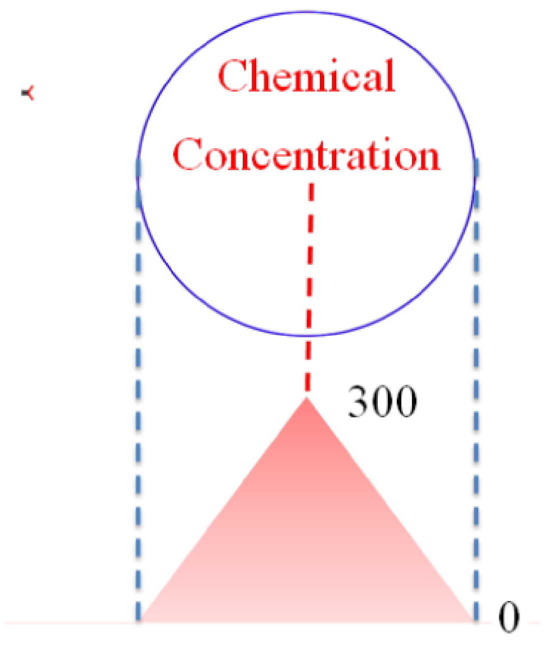


FIGURE 2.12: The representation of chemicals in the model presented by Oros and colleagues. The chemical is represented as a circle where the maximum concentration is at the centre of the circle, and it linearly decreases when we move far from the centre. Taken from [25].

The concentration of each chemical can be calculated at any place on the map using the following equation:

$$c = max((Max - (K \times d)), 0) \tag{2.18}$$

where $c$ is the concentration at any place on the map, $Max$ is the maximum concentration (at the center of the chemical), $K$ is a constant and $d$ is the distance between the center of the chemical and the place where the concentration is calculated. $Max$ = 300 and $K$ = 0.3 were used.

### 2.6.5 Encoding strategies for the sensory information

Oros and colleagues showed that the agent could use both temporal coincidence and firing rate encoding strategies depending on the level of concentration of the chemical at the antennae. When the network used the firing rate encoding only, the agent was not able to detect the difference of the chemical concentration between its two sensors for low concentrations. The network used temporal coincidence encoding for low concentration. With high chemical concentrations, the firing rate encoding was working well.

For the firing rate encoding, choosing the suitable equation to map the concentration of chemicals to sensory neuron current was the main concern for Oros and colleagues. As we can see in Fig. 2.11, the two antennae are connected with 2 sensory neurons. Based on the concentration of the chemical read at the antennae, the sensory neuron current is calculated. The membrane potential is updated every time step based on the sensory neuron current and when the membrane potential reaches the threshold $\theta$, the sensory neuron fires a spike. This leads to an indirect relation between the concentration of the chemical at the antennae and the firing rate of the sensory neuron connected to this antennae (Fig. 2.13).

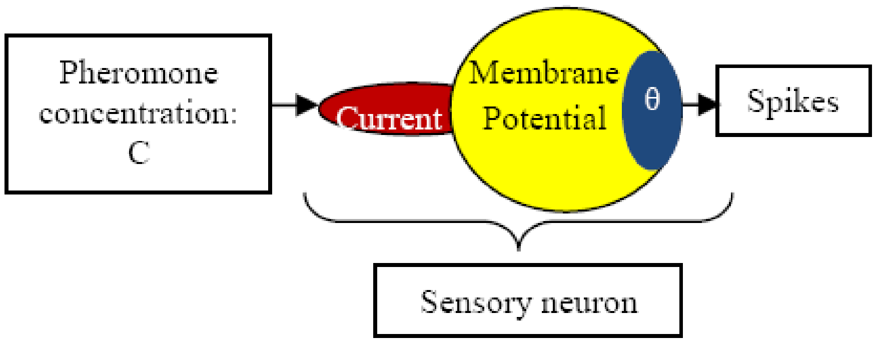


FIGURE 2.13: The relation between the concentration of chemicals at the antennae and the firing rate of the sensory neurons in the model presented by Oros and colleagues. Take from [23].

In order to obtain a model able to detect the small differences between the concentration of the chemicals between the right and left antennae it was very important to find a suitable function to map the chemical concentration to sensory neuron current so that there would be linear relation between the concentration and the firing rate of the sensory neuron. Oros and colleagues tried many equations to map the concentration at the antennae to sensory neuron current [23]. First they started by setting a linear relation between the concentration and the current but the firing rate of the sensory neurons was saturating. They also tried to use Hill function to map the concentration to the sensory neuron current. Hill function was first used by Archibald Hill in 1910 for describing the binding of oxygen to Hemoglobin.

Using the Hill function they got a better relation between the concentration and the firing rate of the sensory neurons but it was not linear yet. Finally they used a sigmoid function with offset. The equation they used was as following:

$$I = K_1 \times \frac{1}{1 + exp\left(\frac{h-C}{K_2}\right)} + b \tag{2.19}$$

where $I$ is the current, $K_1$, $K_2$, $h$, and $b$ are constants, and C is the concentration. $K_1 = 3.9 * 10^4$, $K_2 = 59$, $h$ =691, and $b$ = 0.08 were used. The relation between the concentration and the firing rate (Fig. 2.14) was not exactly linear, but it was still accepted.

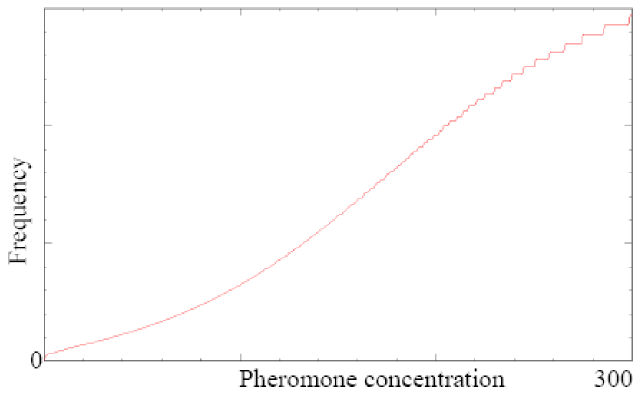


FIGURE 2.14: The relation between the concentration of the chemicals and the firing rate of the sensory neurons using sigmoid function in the model presented by Oros and colleagues. Taken from [23].

### 2.6.6 Adding noise to the neural network

Oros and collaborators noticed that the agent moved straight through and then away from the chemical source when the agent trajectory was directly along the direction of the gradient of the chemical concentration. In this case the values of the concentration of the chemical were equal at the two antennae and the agent could not recognize the position of the chemical source. They added noise to the neural network in order to overcome this problem [21]. Diffusive Ornstein-Uhlenbeck current noise [69] was added to the equation of the total current (Eqn. 2.15). The noise current was calculated as following:

$$\frac{dI(t)}{dt} = -\frac{1}{\tau_I}(I(t) - I_0) + \sqrt{\frac{2\sigma^2}{\tau_I}}\xi(t) \tag{2.20}$$

where $I$ is the total current, $\tau_I$ is the current noise time constant (2ms in their case), $I_0$ is the mean synaptic current (0 in their case), $\sigma$ is the noise diffusion coefficient and $\xi(t)$ is a white Gaussian noise (with mean = 0 and standard deviation = 1). Different values of $\sigma$ were used in the experiments in the range of [0, 0.001].

Adding this coloured noise mimicked the subthreshold voltage fluctuations in real neurons due to the intense network activity [70]. After implementing this noise, the agent was able to stay in the range of the chemical concentration (Fig. 2.15).

In the work I present in Chapter 6 and Chapter 7, I built on the work that had been done by Oros and colleagues. I used the Hill function they used in order to map the food concentration at the sensors to input current.

My work can be considered an extension to the work I have just presented in this chapter. In the work done by Oros and colleagues, they used only one food source from the same type. I allowed the environment to have more food sources (up to 20 food sources), and when the animats ate one of these food sources, this food source disappeared and the concentration map was updated. This task is considered more difficult as the animat should be able to deal with different levels of concentrations.

One of the main extensions I have also done was upgrading the food sources to sound sources. Each sound source produces a different temporal pattern of sounds. The animat should distinguish between them and be able to move forward to only one sound source with a specific pattern.

The differences and the similarities between my work and the work that had been done by Oros and colleagues are presented in more details in chapter 6.

## 2.7 Genetic Algorithm

The genetic algorithm (GA) is one of the heuristic search algorithms which was invented by John Holland in the 1960s [71]. It was inspired by the evolution theory of Darwin (survival of the fittest) and is well described in many books [72–76]. It is an example of the evolutionary algorithms which uses natural evolution techniques for searching optimal solution. These techniques include inheritance, mutation, selection, and crossover.

In nature, crossover occurs in the reproduction process of the chromosomes in which genes from parent chromosomes recombine together to produce new chromosomes. Another operation which also happens during reproduction of chromosomes is mutation. Mutations happen by applying some changes in the produced chromosomes. Fitness is another term that is used as a measurement of how good a solution is. It is kind of

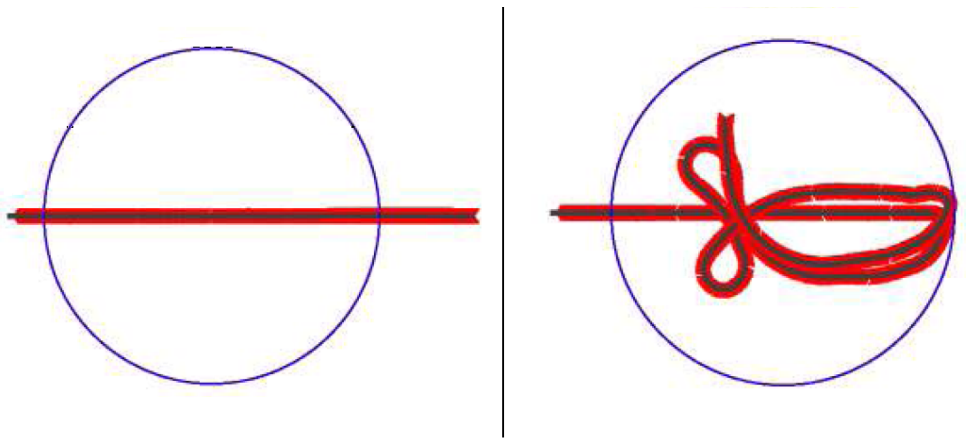


FIGURE 2.15: The behaviour of the animat before and after adding the colored noise in the model presented by Oros and colleagues. After running the agent for 300s, the agent (red path) was not able to stay in the range of the chemical concentration (blue circle) when no noise was added (left panel). When a colored noise was added (right panel) the agent (red path) was able to stay inside the range of the chemical concentration (blue circle) when the same simulation time was used (300s). Taken from [21].

evaluation of the produced chromosomes on which the selection of the chromosomes, which will pass for the next generation, is based on.

In order to apply GA on optimization problem, the first step should be the encoding in which the problem is represented as chromosomes. For example, if we want to apply GA on eight queens problem, one possible representation of the board is a string of 64 bits (encodes a string of genes) for the 64 squares in the boards. The value of each bit in this string shows if this bit is occupied by a queen or not. Another important step is defining the fitness function with which the individuals are evaluated. The fitness function in the eight queens problem would be the number of queens that do not threat any other queen.

More factors should be put in consideration when desiging a GA such as the cross over probability, the probability of mutations, number of individuals in each generation, .. etc.

## 2.8 Gene Regulatory Network

Gene Regulatory Network (GRN) [77–79] is a network between DNA segments in a cell that describes the interaction between genes in this cell. This network is used to formulate the differential equations that represent the kinetics of gene products synthesis and degradation in the cell.

The vertices in the GRN represent genes products while the edges represent the regulation between the genes products. Each edge should have a direction from the regulator gene to the target gene. The target gene is the gene that its expression can be activated or suppressed by the regulator gene. On the other hand, the regulator gene is the gene which controls the target gene.

# Chapter 3

# Mapping GRNs to SNNs in GReaNs platform

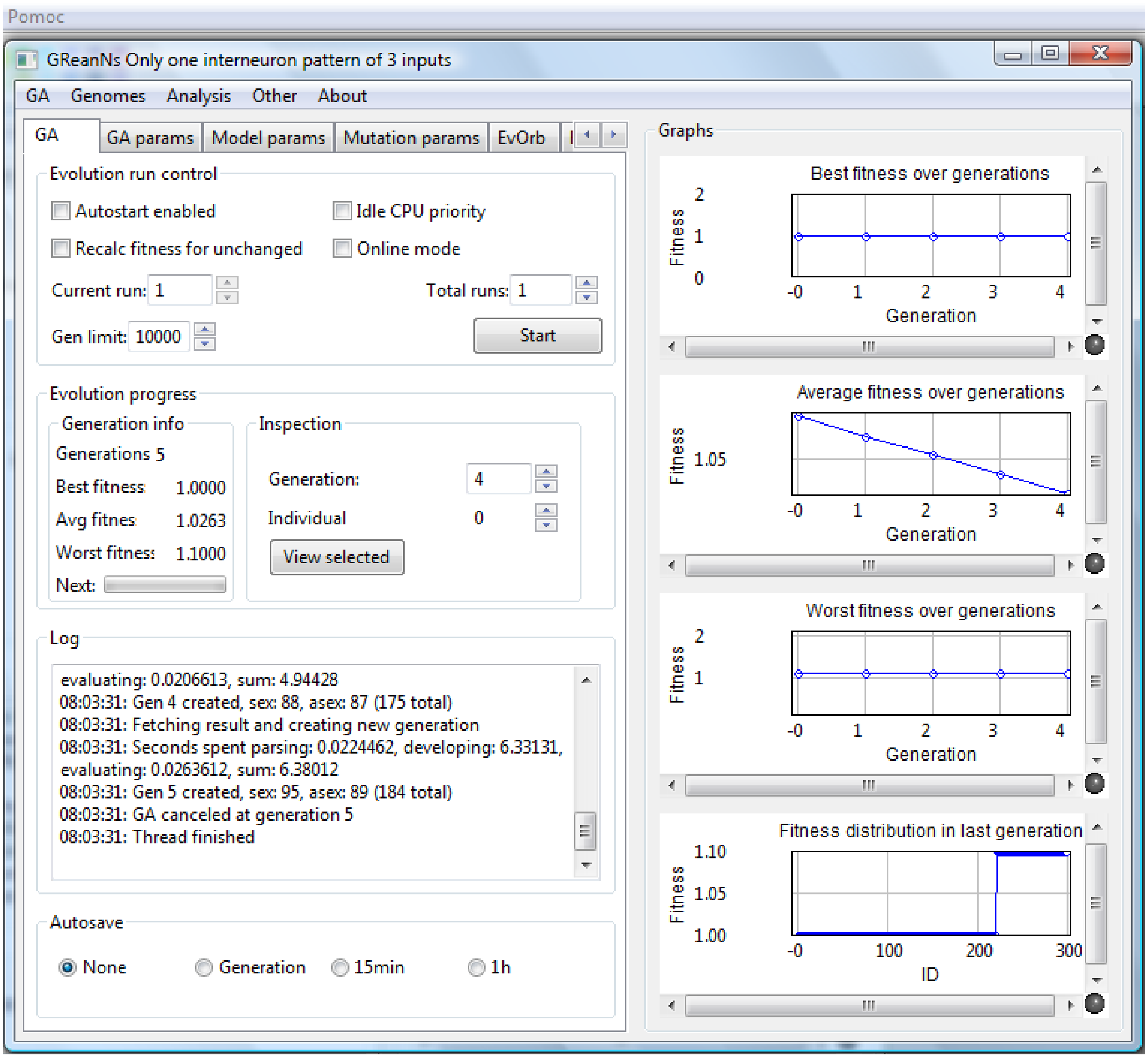
## 3.1 Introduction

The GReaNs is an artificial life software platform which was developed by Michal Joachimczak and Borys Wrobel and used to evolve Gene Regulatory Networks (GRNs).

GReaNs has been used previously for evolving gene regulatory networks for tasks including controlling single cells, as unicellular entities or parts of multicellular bodies in two dimensions [80, 81] (it has been used to transform the structures into soft-bodied animats swimming in a fluid-like environment) and in three dimensions [30–32], processing signals [33], and controlling animats [26, 27]. All the networks which were evolved by GReaNs in the previous work consisted of non-spiking nodes. As I mentioned in 1.1, my work is part of broader research program to compare the computational properties of spiking neural networks and networks that are not spiking, so the first task I conducted in my project was to convert the GRN to a SNN in GReaNs.

## 3.2 From genome to network

The genome in GReaNs was modelled so that it gets the evolutionary features of the biological genomes. It is formed of a string of genetic elements. The genetic element can be either P (promoter), G (gene) or S (a special element which could be an input or output element). The networks in GReaNs (Fig. 3.2) are formed of regulatory units (nodes) with regulatory relations (edges). Each regulatory unit is encoded in the linear genome as a chain of genetic elements of type P succeeded by a chain of genetic elements


Pomoc
GReanNs Only one interneuron pattern of 3 inputs
GA
Genomes
Analysis
Other
About
GA
GA params
Model params
Mutation params
EvOrb
Start
View selected
Graphs
Best fitness over generations
Average fitness over generations
Worst fitness over generations
Fitness distribution in last generation
Fitness
Generation
ID

Evolution run control

Autostart enabled ☐ Idle CPU priority ☐
Recalc fitness for unchanged ☐ Online mode ☐
Current run: 1 Total runs: 1
Gen limit: 10000

Evolution progress

Generation info
Generations 5
Best fitness 1.0000
Avg fitnes 1.0263
Worst fitnes: 1.1000
Next:

Inspection
Generation: 4
Individual 0

Log

```
evaluating: 0.0206613, sum: 4.94428
08:03:31: Gen 4 created, sex: 88, asex: 87 (175 total)
08:03:31: Fetching result and creating new generation
08:03:31: Seconds spent parsing: 0.0224462, developing: 6.33131,
evaluating: 0.0263612, sum: 6.38012
08:03:31: Gen 5 created, sex: 95, asex: 89 (184 total)
08:03:31: GA canceled at generation 5
08:03:31: Thread finished
```

Autosave
[✓ None] Generation 15min 1h

Figure 3.1: A screenshot of the initial window of GReaNs software. On the right side of the figure there are graphs for the history of the best, average, and worst fitness for each generation and graph for the distribution of the fitness in the last generation. On the left side, there are some options to control the evolution run, information about the best, average, and worst fitness in the last generation, and the log of the current running task.

of type G. So each regulatory unit should include at least one promoter followed by at least one gene.

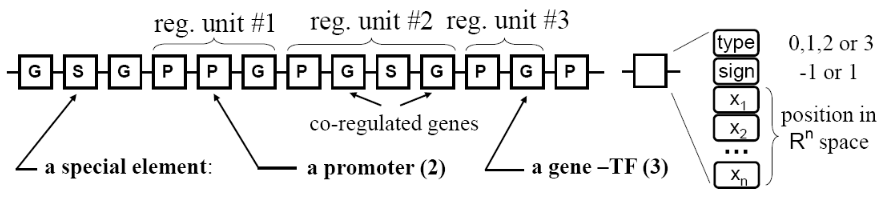


FIGURE 3.2: The structure of the genome in GReaNs. On the left, the structure of the genetic element. On the right, the encoding of regulatory as a chain of promoters followed by a chain of genes. Take from [82].

Each genetic element (Fig. 3.2) is identified by a number of variables: the type (P for promoter, G for gene, or S for external element which could be input or output), the sign (positive or negative which define the kind of interaction between the products and the promoters if it is inhibitory or excitatory) and the coordinates which determine a point in space (product-promoter affinity is a function of the Euclidean distance between them). Based on the number of products (genes) in each regulatory unit and the affinity between them as well as the promoters in other regulatory units, the products concentration is calculated in each step during the simulation.

Based on the distribution of the genetic elements on the two dimensions space, the connections between nodes in the GRN are constructed (Fig. 3.3). If the Euclidean distance between any promoter and any gene in the genome is less than the cutoff distance (5 was used in the work described in this thesis), then based on the product of the signs of the gene and the promoter an excitatory (if the product equals 1) or an inhibitory (if the product equals -1) connection is constructed. The weight of the connection is calculated by the following equation:

$$w = \beta \frac{2(5-d)}{10d+\beta} \tag{3.1}$$

where $w$ is the weight of the connection, $d$ is the distance between the promoter and the gene, and $\beta$ is a factor in order to control the negative relation between the distance and the weight (Fig. 3.4). The variable $d$ can take the values in the range of [0, 5] as the cutoff distance is 5 (there is no connection between any promoter and any gene in the genome if the Euclidean distance between them is more than 5). $\beta = 1$ was used during all the tasks.

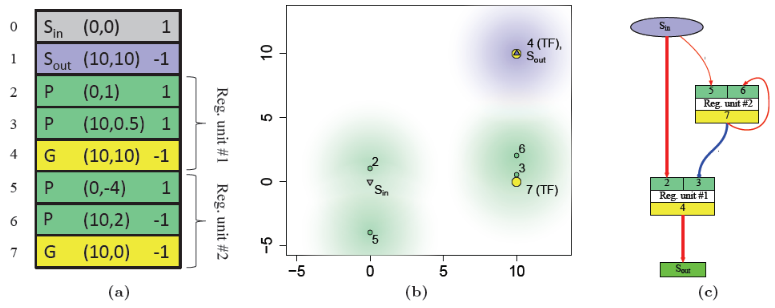


FIGURE 3.3: How edges between the nodes in the GRN are created in GReaNs. The order of the genetic elements on the genome used for identifying the regulatory units, while the positions of the genetic elements in the two dimensions space form the interaction between the regulatory units. (a) a structure of a simple genome. (b) the distribution of the genetic elements of genome in two dimensions space (c) the excitatory (red) and inhibitory (blue) connections between the promoters and the genes if the distance between each of them is less than the cutoff distance. Taken from [26].

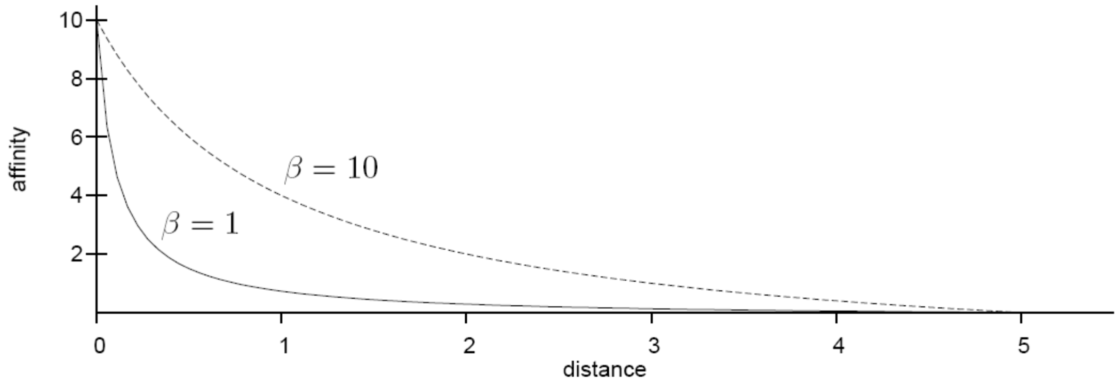


FIGURE 3.4: The relation between the affinity of the connection between any promoter and any gene in the genome and the $\beta$ factor in GReaNs. The figure shows different behaviour of the graph when different values of $\beta$ are used (1 and 10). Taken from [82].

## 3.3 Evolution of networks in GReaNs

A genetic algorithm is used in GReaNs to obtain the required GRN. The genetic operators are applied on the level of the genetic elements by mutating their type, their sign or their position. They are also applied on the level of the genome by changing the order of the genetic elements or changing the number of the genetic elements by adding, deleting, or duplicating them.

At the initialization step, a random population of individuals each with random number of neurons is constructed. In the evaluation step, the simulation of the individuals is carried out and the fitness function is calculated based on the task of the network. The best individuals are then selected and the crossover and mutation operations are applied.

There were two different stopping criteria for the evolutionary algorithm in GReaNs. The first criteria was to stop the algorithm if no improvement was attained in the last specific number of generations, 500 generations for example, while the second criteria is to stop the algorithm after specific number of generations no matter an improvement was attained or not.

There are a number of parameters which are set in GReaNs based on the evolutionary task. Some of these parameters define the structure of the network like the number of regulatory units in the genome (defining the number of the regulatory units that should be decoded, and the rest of the elements in the genome are ignored), the probability of the appearance of each element type when creating new genomes, number of the inputs in the network, number of the outputs, the cutoff affinity between the regulatory elements etc.

Other parameters are responsible for defining the genetic algorithm like the population size in each generation, maximum number of generations, simulation time, number of individuals generated using crossover, elitism factor (number of individuals passing to the next generations without being altered), the probability of mutating every attribute of the genetic elements, probability of duplications and deletions etc.

## 3.4 Mapping the GRN to the SNN

Due to the similarities between the computational characteristics of a node in a biological gene regulatory network and a node in a neural network, I extended GReaNs to simulate and evolve Spiking Neural Networks [83–85].

The extension of GReaNS includes mapping each regulatory unit to a single neuron, the connections between the regulatory units to synapses, and the value of the concentration of the products in the unit to the neuron membrane potential (Table 3.1), in addition to allow each node to fire a spike when the value of the membrane potential exceeded a predefined threshold (in case of LIF model as the AdEx model does not have a threshold). I implemented two models in GreaNs as part of the extension, LIF [2] model and AdEx [3] model.

Table 3.1: Mapping the Gene Regulatory Network to Spiking Neural Network in GReaNs.

| Gene Regulatory Network | Spiking Neural Network |
|---|---|
| Regulatory unit | Neuron |
| Connections between regulatory units | Synapses between neurons |
| Products concentration | Membrane potential |

### 3.4.1 LIF Model in GReaNs

Leaky Integrate and Fire (LIF) is considered one of the simplest and most commonly used spiking neural models. In the simulation part in GReaNs, I used the following differential equation in order to update the membrane potential during each simulation step:

$$\frac{dV}{dt} = \frac{g_L(V_R - V) + g_E(E_{rev,E} - V) + g_I(E_{rev,I} - V)}{C} \tag{3.2}$$

where $V$ is the membrane potential, $V_R$ is the resting potential, $g_L$ is the leak conductance, $g_E$ is the conductance of the excitatory synapses, $g_I$ is the conductance of the inhibitory synapses, $E_{rev,E}$ is the reversal potential of the excitatory input, $E_{rev,I}$ is the reversal potential of inhibitory input, and $C$ is the capacitance of the membrane.

In this model, when the value of the postsynaptic potential in any neuron reaches the threshold voltage ($V_{th}$), the neuron fires a spike and the value of the membrane potential decays to the reset voltage ($V_{reset}$). Based on the Euclidean distance between the genes in the presynaptic node and the promoters of the postsynaptic nodes, and their signs, the postsynaptic conductance is updated. The postsynaptic conductance is decaying exponentially:

$$\frac{g_E}{dt} = \frac{-g_E}{\tau_E} \; and \; \frac{g_I}{dt} = \frac{-g_I}{\tau_I} \tag{3.3}$$

where $\tau_E$ and $\tau_I$ are the decay time constants of the excitatory and inhibitory synaptic conductance respectively. I used the values in Table 3.2 for these parameters. I used the examples for LIF model implemented for Brian simulator [35] as a reference for choosing the model parameters values with small modifications in order to control the firing rate of the neurons.

Table 3.2: The values of the LIF model parameters I used for simulating the SNN in GReaNs.

| Parameter | Value |
|---|---|
| $V_R$ | -65.0 mV |
| $g_L$ | 0.05 $\mu$S |
| $E_{rev,E}$ | 0 mV |
| $E_{rev,I}$ | -70.0 mV |
| $C$ | 1 nF |
| $V_{th}$ | -50.0 mV |
| $V_{reset}$ | -65.0 mV |
| $\tau_E$ | 5.0 ms |
| $\tau_I$ | 5.0 ms |

### 3.4.2 AdEx Model in GReaNs

AdEx model is a two-dimensional extension of the LIF model which is considered a more biologically realistic model. I used the following equations to implement the AdEx model:

$$\frac{dV}{dt} = \frac{g_L(E_L - V + \delta e^{\frac{V-V_T}{\delta}}) + g_E(E_{rev,E} - V) + g_I(E_{rev,I} - V) - W}{C} \tag{3.4}$$

where $V$ is the membrane potential, $E_L$ is the leak reversal potential, $\delta$ is the slope factor, $V_T$ is the threshold potential, $g_L$ is the leak conductance, $g_E$ (or $g_I$) is the conductance of the excitatory (inhibitory) synapses, and $E_{rev,E}$ ($E_{rev,I}$) is the reversal potential of the excitatory (inhibitory) input.

$$\frac{dW}{dt} = \frac{a(V - E_L) - W}{\tau_W} \tag{3.5}$$

where $W$ is the adaptation variable, $C$ is the capacitance of the membrane, $a$ is the adaptation coupling parameter, and $\tau_W$ is the adaptation time constant.
The reset behaviour in this model is slightly different from the LIF model. In addition to changing the value of the membrane potential to $V_{reset}$ when the post-synaptic potential crosses $V_{spike}$ value, the adaptation variable is also changing by adding to it the value of the $b$. I used the values in Table 3.3 for these parameters. Again I used Brain examples which correspond to tonic spiking behaviour as reference for choosing the parameters values with small modifications to control the firing rate of the neurons.

TABLE 3.3: The values of the AdEx model parameters I used for simulating the SNN in GReaNs.

| Parameter | Value |
|---|---|
| $E_L$ | -65.0 mV |
| $\delta$ | 2.0 mV |
| $V_T$ | 50.0 mV |
| $g_L$ | 0.05 $\mu$S |
| $E_{rev,E}$ | 0 mV |
| $E_{rev,I}$ | -70.0 mV |
| $C$ | 1 nF |
| $a$ | 4.0 nS |
| $\tau_W$ | 40.0 ms |
| $V_{reset}$ | -70 mV |
| $V_{spike}$ | -40 mV |
| $b$ | 0.0805 nA |

The same equations used for LIF model were also used here for modeling the postsynaptic conductance.

In both the LIF and the AdEx models I implemented, each neuron could be connected to a postsynaptic neuron by an exitatory synapse and at the same time connected to a postsynaptic neuron by an inhibitory synapse. This allows the neuron to act as an excitatory neuron and as an inhibitory neuron at the same time, which is not biologically common; in biological neurons, in contrast, a given neuron is in most cases either excitatory, or inhibitory [86].

I allowed the neurons to act as excitatory and inhibitory neurons as in the previous tasks that were done in GReaNs using GRNs, each regulatory unit can be connected to other regulatory units with positive and negative connections at the same time. GReaNs could be modified to reflect biology in future work by modifying the way in which genetic elements encode the character of the connection.

### 3.4.3 Validating the SNN implementation in GReaNs

I constructed a simple network of three LIF neurons (Fig. 3.5). Neuron 2 fires four spikes at 1 ms, 10 ms, 50 ms, and 55 ms, while neuron 3 fires once at 40 ms. Both neuron

2 and neuron 3 connected with neuron 1 by equal strength excitatory and inhibitory synapses, respectively.

To validate if the implementation of SNN is correct in GReaNs, I compared the time course of the membrane potential in a neuron 1 simulated in GreaNs with the time course simulated in Brian simulator. I set the simulation time to 100 ms. One feature in GReaNs enables inspecting each neuron and exporting the values of its membrane potential over time. I also exported the network as a PyNN script[34], and simulated it using Brian [35]. The network showed the same behaviour using both GReaNs and Brian (Fig. 3.6). I used the values in Table 3.2 for the LIF model's parameters.

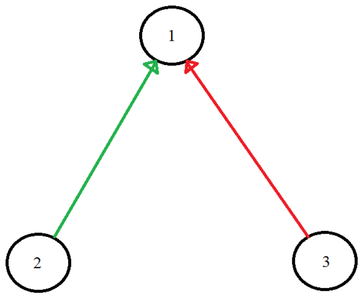


FIGURE 3.5: A simple network of three neurons (circles), neuron 2 and neuron 3 are connected with neuron 1 by excitatory and inhibitory synapses, respectively.

## 3.5 The Evolution of the SNN

I used the same genetic algorithm which was used for evolving the GRN in GReaNs to evolve the SNN. The encoding of the structure of the network in linear genomes described in Fig 3.2 allows for the evolution of the topology of the network and the weights of the synapses between the neurons in the network. The delays of the synapses were constant (1 ms was used) for all the network during the evolution.

## 3.6 Conclusion

We can conclude that mapping the GRN in GReaNs is a suitable approach for encoding SNN that can be used for evolutionary tasks. Implementing this biologically inspired SNN model in GReaNs will allow repeating the previous evolutionary tasks (which used GRN) using SNN and comparing the results obtained before and after adding SNNs to GReaNs.

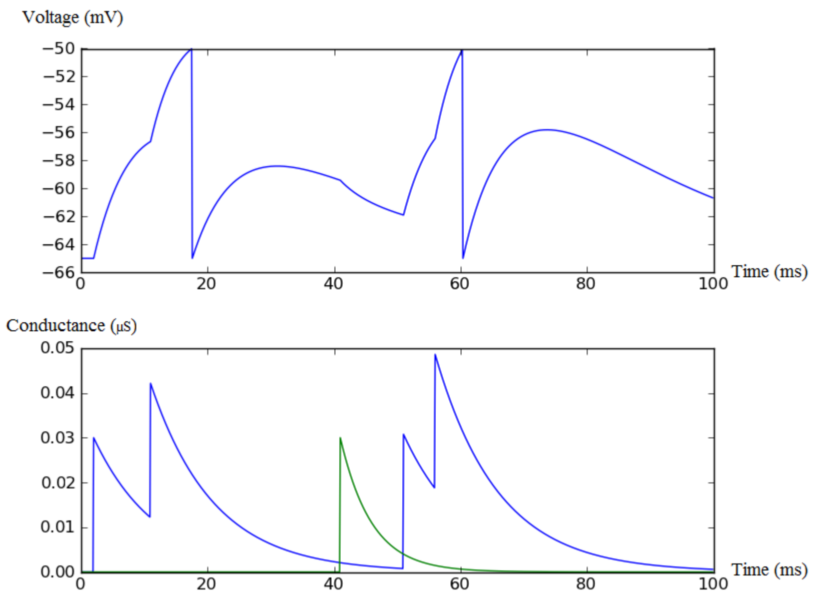


FIGURE 3.6: The behaviour of the network showed in Fig. 3.5 when it was simulated for 100ms. This network was simulated by both GReaNs and by PyNN with Brian as a back-end simulator. The network had the same behaviour with both GReaNs and PyNN. Upper graph: The membrane potential of neuron 1 in the network showed in Fig. 3.5. Lower graph: The conductance of both the excitatory synapses (blue) and the inhibitory synapses (green) of neuron 1.

Chapter 4

# Using GreaNs to Evolve a Spiking Neural Network which Generates Desired Spike Patterns

## 4.1 Introduction

After mapping the GRN to SNN in GReaNs successfully, the next step was to check the ability of evolving SNNs in GReaNs. The first evolutionary task I have tried in GReaNs was evolving spiking neural networks to obtain a network with an output neuron which generates a predefined spike train in response to specific input. This task was an easy task to start with that was investigated before in GReaNs using GRNs. Both the Leaky and Integrate-and-Fire model and the non-linear Adaptive Exponential LIF model (AdEx) were used in this task.

First, I evolved LIF networks to generate the same spike train which was generated by a single AdEx neuron when both the LIF networks and the AdEx neuron were connected to the same input neuron. Exactly the opposite task for AdEx networks: given a spike train which was generated by a single LIF neuron, I evolved AdEx networks to generate the same spike train when both the LIF neuron and the AdeX networks were connected to the same input.

One of the constrains which was imposed in GReaNs is that the direct connection between the input neurons and the output neurons is not permitted. Giving this constrains, it is impossible to achieve this task without a delay in the spike train. I used three different delay levels: 5 ms, 10 ms and 20 ms. Longer delays can also be used, but they will create more difficult tasks, requiring (in principle) more neurons. Furthermore, I extended this task to evolve a network not only able to generate a specific spike train, but to double it (to generate two spikes for each spike in the spike train, one at the same time, and one after 20 ms). From now on, I will call this task as double-shifting task.

For generalization, I tested the difference between the behaviour of the evolved network when it is connected to a different input which was not used during the evolution and the single neuron when it is connected to the same input. Finally, I investigated the statistics on the values of best fitness and the generalization fitness of 10 independent evolved networks.

## 4.2 Genetic algorithm

A genetic algorithm was used to evolve a population of 300 individuals. Each of the genomes in the initial population had 5 regulatory units, encoding a SNN with 5 neurons. The number of promoters and the number of genes in each unit was determined in the same fashion, randomly. A random variable was sampled from a Gaussian distribution (same Gaussian distribution used in the previous tasks in GReaNs, [33] for example), and the number was set to 1 if the variable was smaller than 1. After parsing each genome and building the neural network, each network was simulated. The fitness value of each network is calculated then using the following fitness function:

$$f_{err} = \frac{\alpha\ min(|S_{desired} - S_{GReaNs}|, S_{desired}) + \beta(S_{desired} - M_{GReaNs})}{S_{desired}} \tag{4.1}$$

where $min()$ is the minimum function, $S_{desired}$ is the desired number of spikes, $S_{GReaNs}$ is the number of spikes generated by a network under evaluation, $M_{GReaNs}$ is a match between the actual and desired response, a number that ranged from 0 to the number of desired spikes, and determined as described below, while $\alpha$ and $\beta$ are constant fractions. I used $\alpha + \beta = 1$ in order to limit the value of $f_{err}$ in the range of [0, 1] where the best value is 0 and the worst is 1.

The first term penalized a network if it generated an incorrect number of spikes, but if this number exceeded twice the desired number of spikes, there was no further penalty. The second term in the equation penalized the network if the spikes in its response did not match the required response pattern. Since the original task was to generate matching spikes, I gave more weight to the second term (I used $\alpha = 0.3$ and $\beta = 0.7$). The match was determined by first considering a short (19 ms) temporal window around each spike in the desired response and finding the closest spike in the actual response.

In order to penalize spikes that did not match the required response exactly, the match was determined using a Gaussian function in the following fashion:

$$M_{GReaNs} = \sum_{i=1}^{S_{desired}} e^{\frac{-d_i^2}{15}} \tag{4.2}$$

where $d_i$ is the distance in time (a value between 0 and 9) between the spike in the desired and the closest spike in the actual response. In other words, each spike with exact timing contributed 1 to $M_{Greans}$, a spike 1 ms late or early contributed 0.936, etc., with a spike 9 ms late or early contributing only 0.0045.

After calculating the value of $f_{err}$ for all the individuals, the 300 individuals of the next generation were formed. The best five individuals were copied without any modification (elitism), and tournament selection was used to choose the rest 295 individuals. The selection was performed by choosing two random genomes from the previous generation and selecting the best of them. Mutations were applied then on the 295 individuals and crossover was applied on only 100 individuals.

For the cross-over, single point cross-over was introduced for each genome. The position of the cross-over on the genome was sampled uniformly. The individuals in the parent population were also sampled uniformly and each genome from the parent population was selected with probability 0.5.

Mutations were applied at both the level of genetic elements and at the level of the genome. At the level of genetic elements by mutating the type of the elements (each element could be promoter, gene or special element for input or output) with a probability 0.005, by mutating the sign of the elements (can be - or +) with probability 0.005, or by mutating the coordinates of the elements by adding to each coordinate random value from normal distribution centered at 0. Mutations were applied at the level of genome, with a probability 0.1 for each position on the genome by adding, deleting, or duplicating a chain of elements. For adding or deleting a chain of elements, the length of the chain was sampled from a logarithmic distribution with mean 10. For duplication, the length of the chain was also sampled form a logarithmic distribution and the position where the duplicate chain would be added was chosen from a uniform distribution over the genome.

Apart from the fitness function which I formulated, the rest of the genetic algorithm with the stated parameters values in the last two paragraphs was the same genetic algorithm used before in the previous tasks in GReaNs ([33] for example).

## 4.3 Results

### 4.3.1 Reproducing spike trains

I used the Brian simulator [35] to simulate single AdEx neuron. Then I used GReaNs to evolve LIF networks to generate the same spike train generated by the single AdEx neuron simulated with Brian. Both the AdEx neuron and the LIF networks were connected to the same Poisson spike source with 100 Hz. As I mentioned in the introduction, the spike train was shifted by 5 ms, 10 ms, or 20 ms. The second task was exactly the opposite of the first task by using LIF model in Brian and AdEx model in GReaNs. The results in this section are for the networks with the best $f_{err}$ out of 10 independent runs for each configuration each was evolved for 500 generations.

In order to check if GReaNs successfully trained the SNNs or it just overfitted, I tested the generalization properties of the final SNNs. Generalization was done by checking the behaviour of the final SNNs, when they were connected to any input not used during the evolution. Here are the final results:

- For the shift of 5 ms (Fig. 4.1), and after 500 generations, the best network of LIF neurons had $f_{err} = 0.005$ (0.09 for generalization), while the best network of AdEx neurons had $f_{err} = 0.0$ (0.18 for generalization).

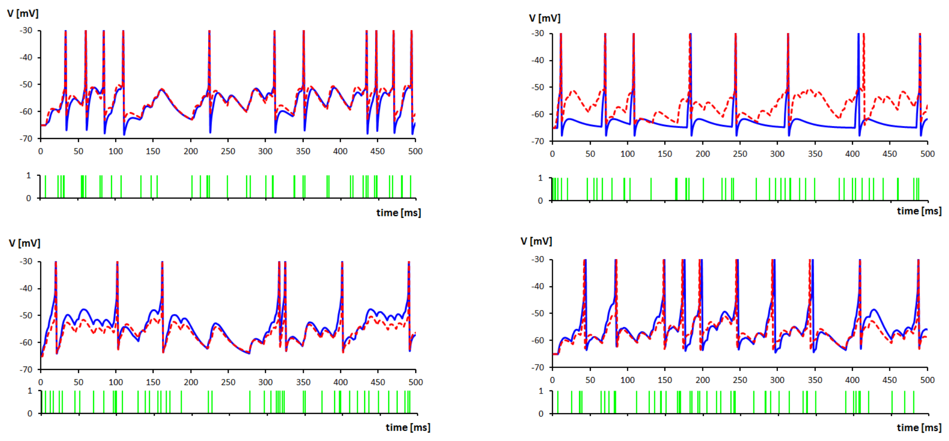


FIGURE 4.1: The behaviour of the champion networks evolved to match a response of a single neuron, but shifted with 5 ms. Upper (lower) panels show the membrane potentials of the output neuron in a LIF (AdEx) network (blue line) matching each spike in the response of one AdEx (LIF) neuron (red), to the input (green) seen during evolution (left panels) and not (right panels.)

- For the shift of 10 ms (Fig. 4.2), and after 500 generations, the best network of LIF neurons had $f_{err} = 0.01$ (0.051 for generalization), while the best network of AdEx neurons had $f_{err} = 0.0$ (0.057 for generalization).

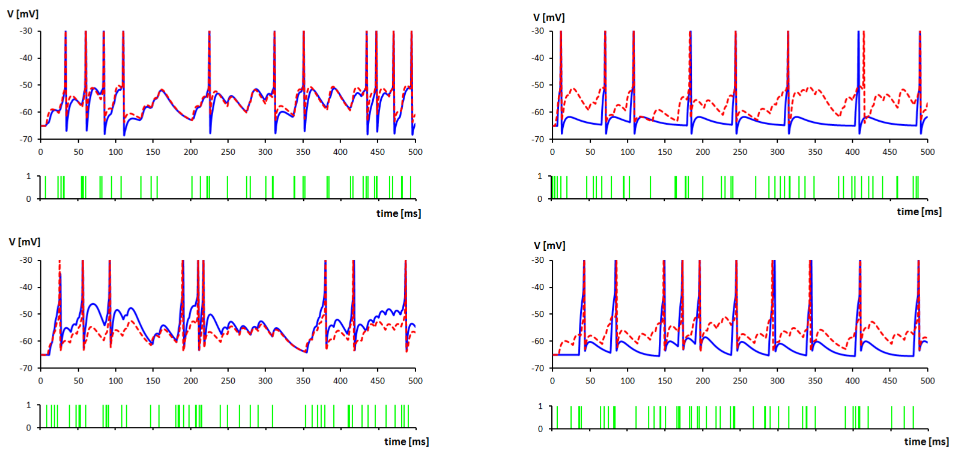


FIGURE 4.2: The behaviour of the champion networks evolved to match a response of a single neuron, but shifted with 10 ms. Upper (lower) panels show the membrane potentials of the output neuron in a LIF (AdEx) network (blue line) matching each spike in the response of one AdEx (LIF) neuron (red), to the input (green) seen during evolution (left panels) and not (right panels.)

- For the shift of 20 ms (Fig. 4.3), and after 500 generations, the best network of LIF neurons had $f_{err} = 0.15$ (0.65 for generalization), while the best network of AdEx neurons had $f_{err} = 0.006$ (0.142 for generalization).

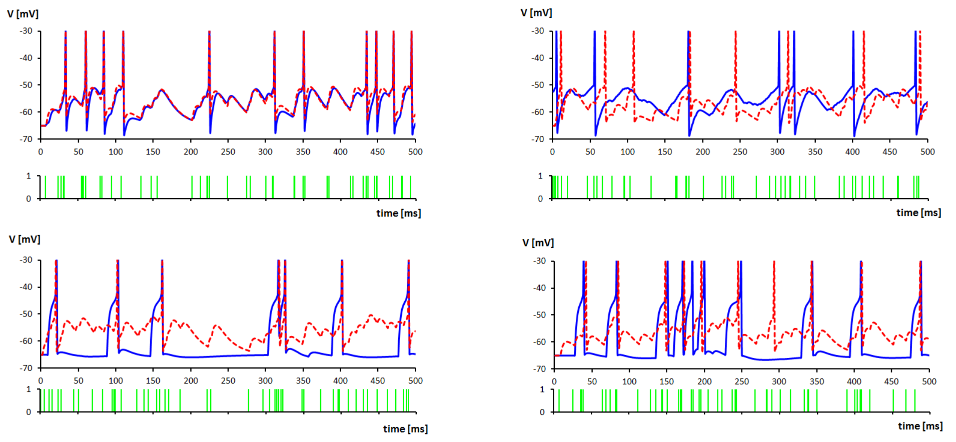


FIGURE 4.3: The behaviour of the champion networks evolved to match a response of a single neuron, but shifted with 20 ms. Upper (lower) panels show the membrane potentials of the output neuron in a LIF (AdEx) network (blue line) matching each spike in the response of one AdEx (LIF) neuron (red), to the input (green) seen during evolution (left panels) and not (right panels.)

### 4.3.2 Double-shifting task

The second evolutionary task I tried in GReaNs was evolving LIF SNNs to double the spike train which was generated by a single AdEx neuron, when both the network and the single neuron are connected to the same input (I used a Poisson spike source with spike frequency 75 Hz) (Fig. 4.4). That task of duplicating the spike train means the following: for every spike in the original response, generates a response with one spike 5ms later, and another 20 ms later. Then I tested the generalization of the final network by comparing the output of the network with the output of the single neuron when another random input was used. The best network of LIF neurons had $f_{err} = 0.11$ (0.33 for generalization), while the best network of AdEx neurons had $f_{err} = 0.17$ (0.53 for generalization)

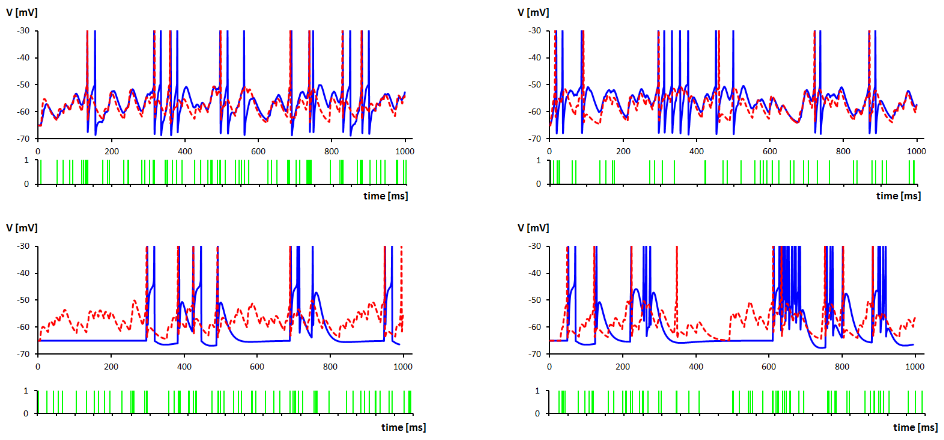


FIGURE 4.4: The behaviour of the champion networks evolved to match a response of a single neuron, but doubled and shifted with 5 ms. Upper (lower) panels show the membrane potentials of the output neuron in a LIF (AdEx) network (blue line) matching each spike in the response of one AdEx (LIF) neuron (red), to the input (green) seen during evolution (left panels) and not (right panels.)

I investigated in more detail the results of evolving (LIF and AdEx) networks to match the output of a single (AdEx and LIF) neuron shifted by 20 ms and the results of the task of duplicating spike train (Table. 4.1). I calculated the average and standard deviation of the $f_{err}$ during the evolution and for the generalization of the 10 independent runs with each task. I also calculated the average and the standard deviation of the size of the networks for each task.

After checking these results (Table. 4.1), we can see that the networks have lower fitness when they are tested with different inputs not used during the evolution. It it also obvious that the task of matching a spike train shifted by 20 ms is much easier for AdEx networks. On the other hand, the task of duplicating spike train is equally difficult for both LIF and AdEx networks.

We can also see that there is a clear overfitting especially when the tasks are getting harder (Table. 4.2). For example, for the task of LIF network matching an output of one AdEx neuron shifted by 20 ms, the network with the best $f_{err}$ during evolution (0.146) had the value of $f_{err} = 0.652$ for generalization (upper figures in Fig. 4.3), while the

TABLE 4.1: The average ± standard deviation and range (square brackets) of the values of $f_{err}$ and size of the networks evolved with GReaNs for the best networks in each of 10 independent evolutionary runs for each task, both for the input spike train used during evolution and for another Poisson spike train (generalization).

| Type of the network and task for evolution | $f_{err}$, input used in evolution | $f_{err}$, another input | network size |
|---|---|---|---|
| LIF network matching an output of one AdEx neuron shifted by 20 ms | $0.239 \pm 0.046$ $[0.146, 0.309]$ | $0.729 \pm 0.137$ $[0.517, 0.963]$ | $8.3 \pm 3.2$ $[5, 15]$ |
| AdEx network matching an output of one LIF neuron shifted by 20 ms | $0.087 \pm 0.065$ $[0.006, 0.158]$ | $0.390 \pm 0.185$ $[0.142, 0.625]$ | $9.6 \pm 4.4$ $[5, 16]$ |
| LIF network matching a doubled-shifted output of one AdEx neuron | $0.235 \pm 0.060$ $[0.112, 0.300]$ | $0.38 \pm 0.058$ $[0.290, 0.434]$ | $11.2 \pm 5.8$ $[4, 26]$ |
| AdEx network matching a doubled-shifted output of one LIF neuron | $0.258 \pm 0.044$ $[0.171, 0.301]$ | $0.478 \pm 0.044$ $[0.373, 0.531]$ | $8.1 \pm 3.9$ $[4, 17]$ |

TABLE 4.2: Comparison between the networks with the best $f_{err}$ values during the evolution and the networks with the best $f_{err}$ values for generalization over the 10 independent evolutionary runs. The numbers out side the parentheses are the values of $f_{err}$ during the evolution while the numbers in the parentheses are the values of $f_{err}$ for generalization

| Type of the network and task for evolution | Best $f_{err}$ during evolution | Best $f_{err}$ for generalization |
|---|---|---|
| LIF network matching an output of one AdEx neuron shifted by 20 ms | 0.146 (0.652) | 0.285 (0.516) |
| AdEx network matching an output of one LIF neuron shifted by 20 ms | 0.006 (0.142) | 0.006 (0.142) |
| LIF network matching a doubled-shifted output of one AdEx neuron | 0.112 (0.327) | 0.239 (0.290) |
| AdEx network matching a doubled-shifted output of one LIF neuron | 0.171 (0.531) | 0.301 (0.373) |

network with the best value of $f_{err}$ during generalization (0.516) had only $f_{err} = 0.285$ during evolution (Fig. 4.5).

The network of AdEx neurons did not have the same behaviour for the same task, as the network with the best $f_{err}$ during evolution (0.006) had also the best $f_{err}$ during the generalization (0.142) (lower figures in Fig. 4.3). The overfitting appeared again in the double-shifting task of one AdEx neuron, as the network with the best $f_{err}$ during the evolution (0.112) had $f_{err} = 0.327$ during the generalization (upper figures in Fig. 4.4), while the network with the best $f_{err}$ during generalization (0.290) had $f_{err} = 0.239$ during evolution (Fig. 4.6).

The overfitting was considerable in the double-shifting task of one LIF neuron. In this task, the network with the best $f_{err}$ (0.171) during the evolution had the worst $f_{err}$ (0.531) during generalization (lower figures in Fig. 4.4), while the network with the best $f_{err}$ (0.373) during the generalization had the worst $f_{err}$ (0.301) during the evolution (Fig. 4.7).

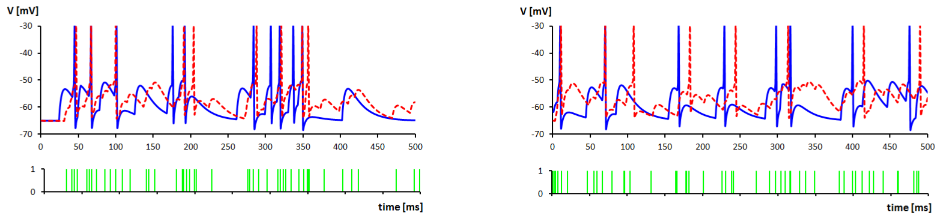


FIGURE 4.5: The behaviour of the best LIF network (in terms of generalization out of 10 independent evolutionary runs) evolved to match the spikes of single AdEx neuron shifted by 20 ms.

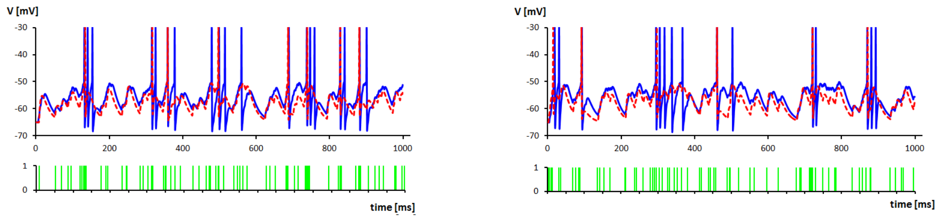


FIGURE 4.6: The behaviour of the best LIF network (in terms of generalization out of 10 independent evolutionary runs) evolved to match the spikes of single AdEx neuron shifted by 5 ms.

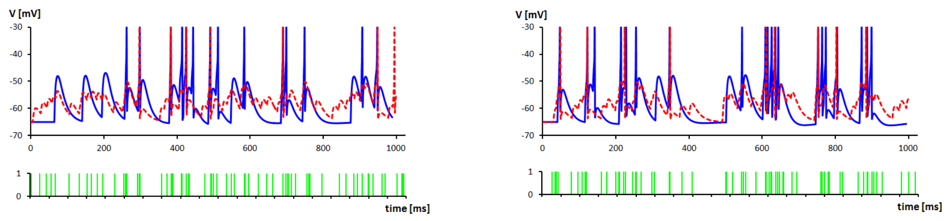


FIGURE 4.7: The behaviour of the best AdEx network (in terms of generalization out of 10 independent evolutionary runs) evolved to double-shift the spikes of single LIF neuron.

## 4.4 Conclusion

The results show that it is possible to use GReaNs to evolve a network connected so that when the input is presented to the input neuron in the network, the output neuron generates the same output (shifted or double-shifted) generated by a single neuron when the same input is presented to this neuron. This task is simple but not trivial and can be uses as a proof-of-principle for evolving SNN in GReaNs. This task opened the gate for more difficult tasks to be carried out using GReaNs.

Chapter 5

# Temporal Pattern Recognition in GReaNs

## 5.1 Introduction

Temporal patterns of spikes is one of the methods the brain uses to encode sensory information [3, 5–11]. The brain uses this encoding for processing information for vision [64], hearing [12], and olfaction [13].

In this chapter I studied the ability of GReaNs to evolve spiking networks, with varying numbers of LIF neurons (1, 2, 5, 10, and unlimited) to recognize predefined temporal patterns of various number of inputs (3 and 4). Then I tested the behaviour of the final networks when Gaussian noise with different standard deviations (10, 20, and 30 ms) was applied to the time of the spike in each input [85].

Furthermore, I used the noisy inputs during the evolution and checked if I could get final networks without error. It was interesting then to study the difference between the structure of the final networks (with only 2 interneurons) which were evolved in absence of noise and the final networks (with 2 interneurons also) which were evolved in presence of noise.

Temporal pattern recognition has been performed before by adjusting synaptic delays between the neurons [16–18], but in the work I have done I allowed only the evolution of the synaptic weights and the structure of the network with fixed synaptic delays.

## 5.2 Genetic algorithm

As I explained in the the previous chapter, a genetic algorithm was used to evolve the genome in GReaNs. Genetic operators were applied at both the genome level and at the genetic elements level. At the genome level, mutations were introduced with a probability of 0.1 for each element, by adding a string of random elements, by deleting an existing string of elements or by duplicating a string of existing elements in the genome in any position in it. The position of the duplication was chosen from a uniform distribution, and the length of the string was sampled from a logarithmic distribution with mean 10. Moreover, one-point cross-over was applied on the genome level with a probability of 0.5 per each genome selected randomly from the parent population and sampled uniformly. The genetic operators were also applied at the level of genetic elements by changing the sign of the elements (- or +) with a probability 0.005, by changing the type of the elements (promoter, gene, or special element) with a probability 0.005, or by changing the coordinates by adding to each coordinate a random value from a normal distribution centred at 0.

The first generation was initialized with 300 individuals. Each individual encoded 5 interneurons. Each interneuron was encoded by strings of promoters and genes. The length of the string encoding each interneuron was sampled from a Gaussian distribution with mean = 1, standard deviation = 1 and minimum value = 1. The size of the population in each generation was fixed during the whole experiment. After the evaluation of all 300 individuals, the new generation was formed by copying the best 5 individuals (elitism), selecting 295 individuals (the selection was performed by choosing 295 times 2 random genomes from the previous generation and selecting the best of them) and applying mutations to all of them and crossover to only 100 (50 randomly chosen pairs) of them.

## 5.3 Temporal pattern recognition with a sequence of four inputs

The first task I have tried was evolving a SNN of LIF neurons to generate the maximum number of spikes when it is presented with a temporal pattern of 4 inputs (1-2-3-4), each with only one spike at the times: 50 ms, 150 ms, 250 ms, and 350 ms, respectively, and to stay silent when it was presented with other patterns of the 4 inputs with only 1 spike on each of them (23 permutations). The period of simulation I used for this experiment was 1000 ms. I used the following function during the evolution to calculate the error:

$$f_{err} = 1 - \frac{\alpha\ min(S_{desired}, S_1)}{S_{desired}} + \frac{\beta \Sigma_{i=2}^{24} min(S_{desired}, S_i)}{S_{desired}} \tag{5.1}$$

where $min()$ is the minimum function, $S_i$ is the number of spikes generated by a network when it is presented with the pattern $i$ during the period between 350 and 1000 ms, $S_{desired}$ is the desired number of spikes, while $\alpha$ and $\beta$ are constants.

I used $S_{desired} = 250$, as the output neuron of the network should start firing spikes only after the last input (350 ms) and since the refractory period of each neuron is 2 ms, the maximum number of spikes that can be generated is $650/2 = 325$, so 250 spikes were considered sufficient. I used $\alpha = 1$, and $\beta = 0.04$ during this task. The second term, $\frac{\beta \sum_{i=2}^{24} min(S_{desired}, S_i)}{S_{desired}}$, was added to penalize the network activity in response to the other 23 patterns, with the same input spike timings but a different order (for example, 2-3-1-4).

After less than 1000 generations, I managed to get an individual (Fig. 5.1) with $f_{err} = 0$. By checking the behaviour of each neuron individually (Fig. 5.2), we can conclude the following:

- Neuron 1 started firing after receiving input 2, so it acts as an input 1 – input 2 detector.
- Because of the excitatory feedback loop of neuron 1, neuron 1 keeps firing at a fixed rate which puts neuron 2 in a plateau state.
- When neuron 1 receives the third input, it starts firing at a higher rate, which makes neuron 2 go to a higher plateau state.
- When neuron 2 receives the fourth input, it starts spiking, so it acts as a neuron 1 - input 3 - input 4 detector.
- Because of the strong excitatory feedback loop of neuron 2 and the strong excitatory connection between neuron 2 and 1, both neuron 1 and neuron 2 keep firing at the maximum rate.
- The excitatory connections between both neuron 1 and neuron and the output allow the output to fire at maximum rate.
- Neuron 3 does not play any role in the process and the behaviour of the network will not change if it were to be removed.

Since neuron 1 worked as input - 1 input 2 detector, the network was silent for all the patterns which did not contain the sub-pattern 1-2. So why the output neuron was not active with other patterns which had the sub-pattern 1-2 (for example, 3-1-2-4)? Neuron 2 needed to receive spikes from neuron 1 followed by one spike from each of input 3 and input 4 in order to start spiking. The sub-pattern 1-2 was followed by spikes from input 3 and input 4 in only 2 patterns; 1-2-3-4 and 1-2-4-3. The pattern 1-2-4-3 did not allow neuron 2 to be active due to the strong inhibitory connection between input 4 and neuron 1.

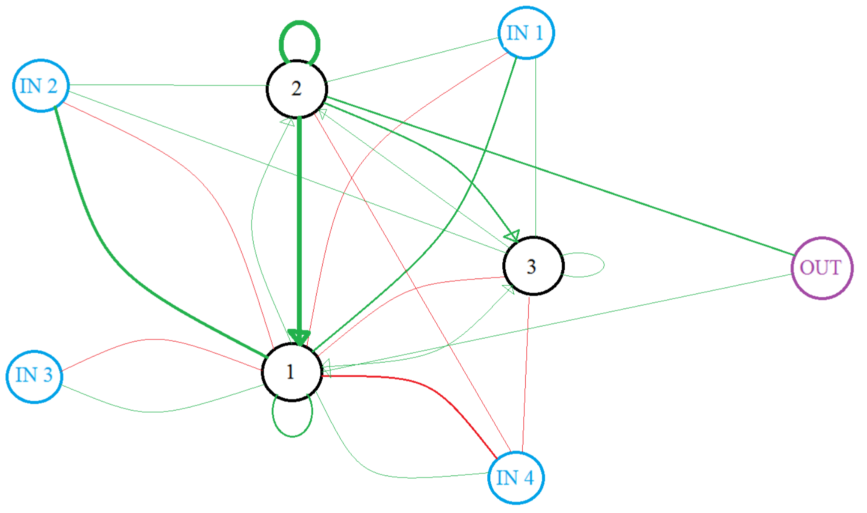


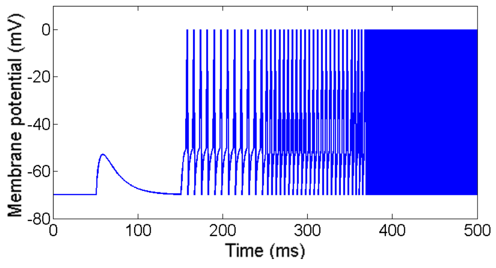
Figure 5.1: The evolved SNN to distinguish one pattern of four inputs. The network has four inputs (cyan) each with only one spike at the times: 50 ms, 150 ms, 250 ms, and 350 ms, respectively, 3 interneurons (black), and one output (purple). The neurons are connected with both excitatory connections (green) and inhibitory connections (red). The thickness of lines signifies the strength of the connections.

## 5.4 Temporal pattern recognition with a sequence of three inputs

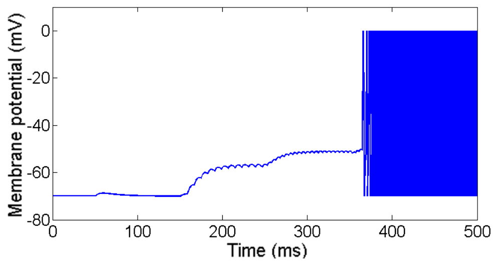
In order to understand the behaviour of the final networks more easily, I decreased the number of inputs to 3 (1 2 3), each with one spike at the times 50 ms, 150 ms, and 250 ms, respectively. I also introduced constraints on the number of the neurons in the network.

### 5.4.1 Pattern 1-2-3 recognition with a varying hard-coded limit on the size of the network

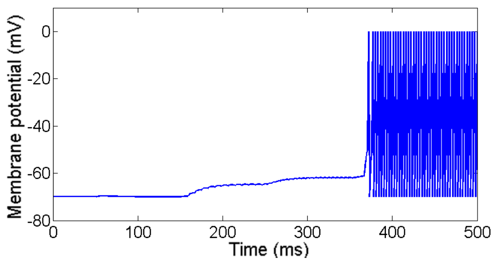
Using three inputs, each generating one spike at the times 50 ms, 150 ms, and 250 ms, respectively, I evolved the SNNs in order to fire at the maximum rate when the network



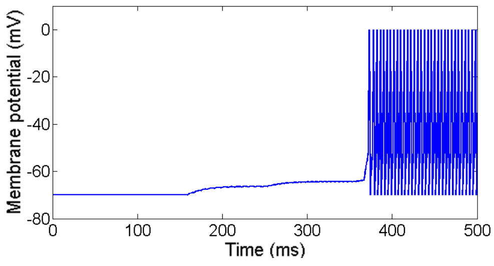
(A) Neuron 1



(B) Neuron 2



(C) Neuron 3



(D) Output

FIGURE 5.2: The behaviour of the neurons of the evolved SNN in (Fig. 5.1) to distinguish the pattern 1-2-3-4. The actual simulation time is 1000 ms, but for clarity only up to 500 ms are shown, the neurons continue spiking afterwards. Note: the horizontal bands are an artifact of plotting.

was presented with the pattern 1-2-3, and to stay silent when it is presented with other patterns (5 permutations). The following function was used to calculate the error:

$$f_{err} = 1 - \frac{\alpha \; min(S_{desired}, S_1)}{S_{desired}} + \frac{\beta \Sigma_{i=2}^{6} min(S_{desired}, S_i)}{S_{desired}} \tag{5.2}$$

where $min()$ is again the minimum function, $S_i$ is the number of spikes generated by a network when it is presented with pattern number $i$ during the period between 250 and 1000 ms, same as previous task, $S_{desired} = 250$ is the desired number of spikes, while $\alpha$ and $\beta$ are constant fractions. I used $\alpha = 1$ and $\beta = 0.2$ during this task so that the weight of the behaviour of the SNN with the first input pattern equals the summation of the weights of the behaviours of the SNN with the other five input patterns. The values of $f_{err}$ vary from 0 for the best network to 2 for the worst network.

Here again the second term, $\frac{\beta \sum\limits_{i=2}^{6} min(S_{desired}, S_i)}{S_{desired}}$, was added to penalize the network activity in response to the other 5 patterns, with the same spike timings but a different order (for example, 2-3-1).

I studied the behaviour of the final network with different number of neurons. First I started with the simplest network with only one neuron (Fig. 5.3). There is only one possible topology of this network since direct connections between inputs and output are not permitted. So the interneuron will act as pattern 1-2-3 detector; there is a strong excitatory connection between the interneuron and the output neuron so that the output will act as an interneuron 1 detector and an excitatory feedback loop for the interneuron allows it to continue spiking.

The reason why the network was silent with other patterns was not clear from checking the structure of the network in Fig. 5.3. The figure shows that the three input neurons were connected with the interneuron with three equal weights excitatory synapses and three equal weights inhibitory synapses. After checking the exact values of the weights of the synapses, I found that the pattern 1-2-3 represents the descending order of the weights of the inhibitory synapses and the ascending order of the excitatory synapses between the inputs and the interneuron. The differences between the values of the weights were very small to appear in the figure. This order minimize the effect of the inhibitory neurons in a way that allow the network to be active only with the pattern 1-2-3 and be silent with all the other patterns.

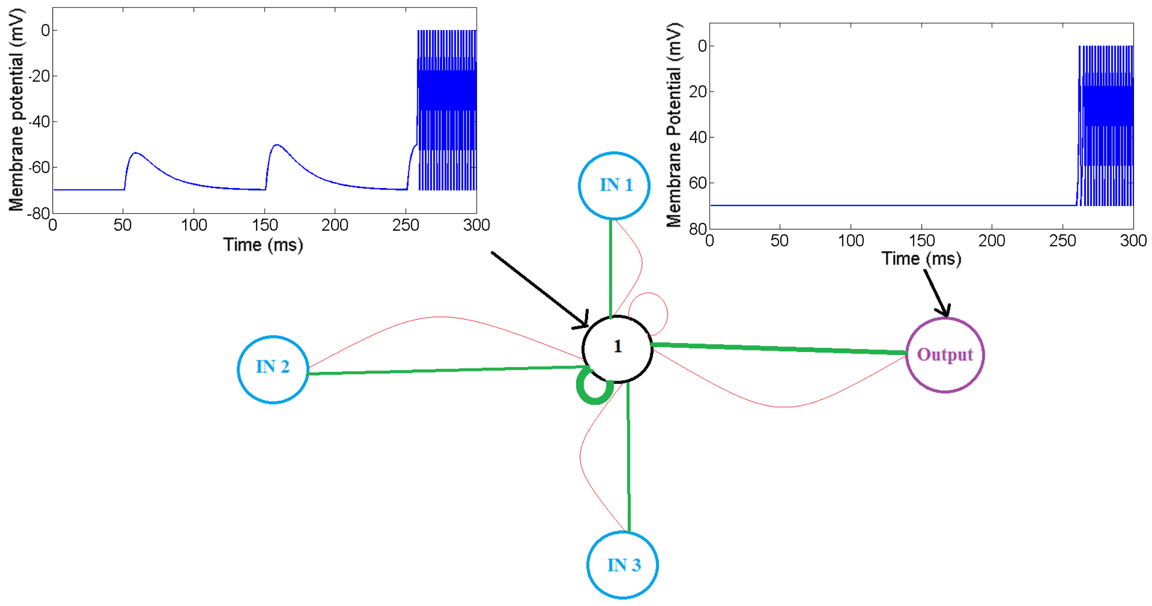


Figure 5.3: The evolved SNN with one interneuron to distinguish one pattern of three inputs. The network has 3 inputs (cyan) each with only one spike at the times: 50 ms, 150 ms, and 250 ms, respectively, 1 interneuron (black), and one output (purple). This network was evolved to distinguish the input pattern 1-2-3 from the 5 other permuted patterns. The neurons are connected with both excitatory connections (green) and inhibitory connections (red). The thickness of lines signifies the strength of the connections. The actual simulation time is 1000 ms, but for clarity only up to 300 ms are shown, the neurons continue spiking afterwards.

I increased the complexity by allowing the network to include two interneurons (Fig. 5.4). In the resulting final network interneuron 2 plays the role of the pattern 1-2 detector, interneuron 2 allows interneuron 1 to move to the plateau state and to stay in stand by, waiting for any small stimulus in order to spike, so interneuron 1 acts as an interneuron 2-input 3 detector, and finally the output detects the pattern interneuron 2- interneuron 1.

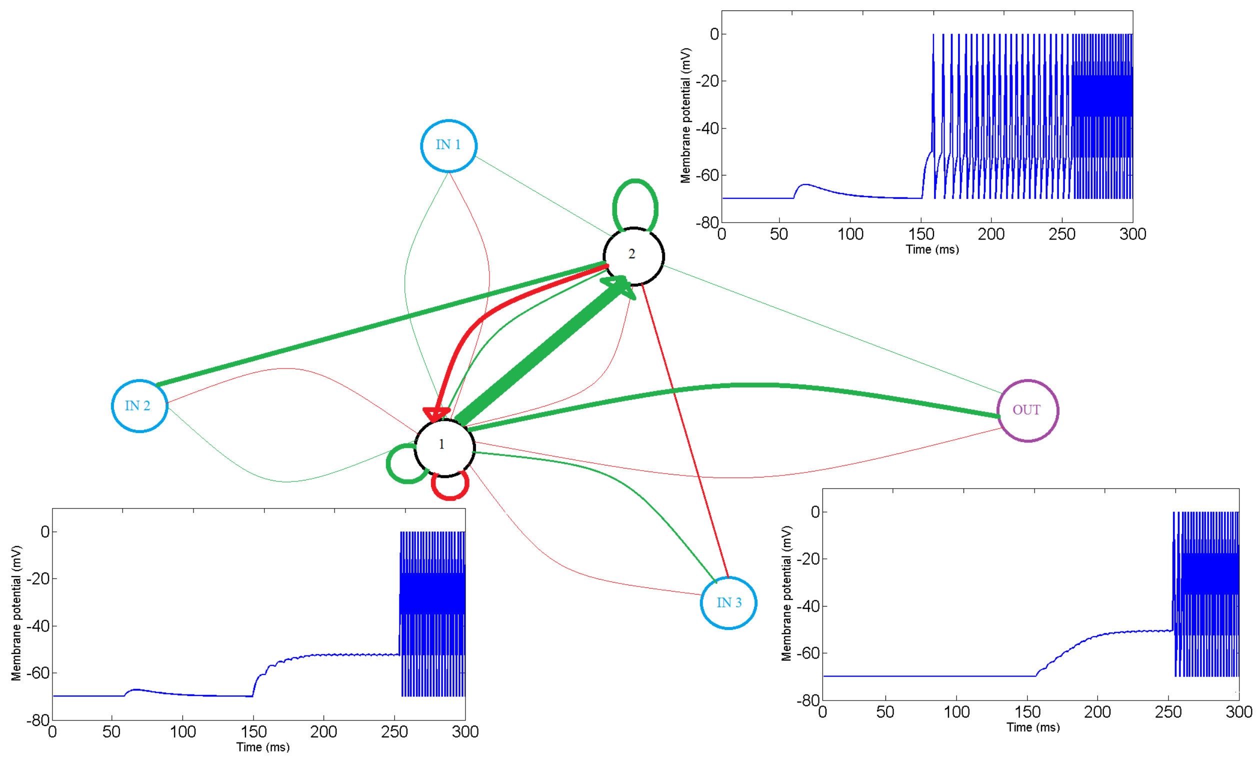


Figure 5.4: The evolved SNN with two interneurons to distinguish one pattern of three inputs. The network has 3 inputs (cyan) each with only one spike at the times: 50 ms, 150 ms, and 250 ms, respectively, 2 interneurons (black), and one output (purple). This network was evolved to distinguish the input pattern 1-2-3 from the 5 other permuted patterns. The neurons are connected with both excitatory connections (green) and inhibitory connections (red). The thickness of lines signifies the strength of the connections. The actual simulation time is 1000 ms, but for clarity only up to 300 ms are shown, the neurons continue spiking afterwards.

Networks with 5 and 10 interneurons were also successfully evolved and managed to distinguish the pattern 1-2-3, but it was very complicated to understand their behaviours.

### 5.4.2 The robustness to temporal noise

I checked the robustness of the evolved networks (with 1, 2, 5, and 10 interneurons) to temporal noise added to spike times (Gaussian noise with standard deviations 10, 20, and 30 ms). I compared the robustness of the network when the temporal noise was not present during the evolution, and when it was present at different levels (10, 20, and 30 ms standard deviation) during the evolution.

Table 5.1: Evolution of robustness to noise in a temporal pattern recognition task in LIF networks with 1 interneuron. The networks evolved with various level of noise in the input (Gaussian noise added to spike times, centered at zero and with indicated standard deviation, sd) were tested with the same (or different) level of noise that (than) was used during evolution. The first value shows the performance of the best network in 10 runs, the values in brackets are averages ± sd.

| Testing | Temporal noise during evolution | | | |
|---|---|---|---|---|
| | no noise | sd = 10 ms | sd = 20 ms | sd = 30 ms |
| no noise | $0(0.09 \pm 0.06)$ | $0.05(0.056 \pm 0.012)$ | $0.05(0.07 \pm 0.04)$ | $0.049(0.08 \pm 0.076)$ |
| sd=10 ms | $0.017(0.48 \pm 0.21)$ | $0.198(0.25 \pm 0.06)$ | $0.2(0.23 \pm 0.1)$ | $0.196(0.235 \pm 0.1)$ |
| sd=20 ms | $0.042(0.55 \pm 0.2)$ | $0.29(0.39 \pm 0.05)$ | $0.22(0.27 \pm 0.09)$ | $0.216(0.271 \pm 0.093)$ |
| sd=30 ms | $0.11(0.56 \pm 0.18)$ | $0.4(0.49 \pm 0.06)$ | $0.25(0.37 \pm 0.09)$ | $0.229(0.361 \pm 0.081)$ |

Table 5.2: Evolution of robustness to noise in a temporal pattern recognition task in LIF networks with 2 interneurons. The networks evolved with various level of noise in the input (Gaussian noise added to spike times, centered at zero and with indicated standard deviation, sd) were tested with the same (or different) level of noise that (than) was used during evolution. The first value shows the performance of the best network in 10 runs, the values in brackets are averages ± sd.

| Testing | Temporal noise during evolution | | | |
|---|---|---|---|---|
| | no noise | sd=10 ms | sd=20 ms | sd=30 ms |
| no noise | $0\ (0.026 \pm 0.052)$ | $0\ (0.04 \pm 0.07)$ | $0.05\ (0.18 \pm 0.2)$ | $0\ (0.08 \pm 0.12)$ |
| sd=10 ms | $0\ (0.36 \pm 0.22)$ | $0\ (0.12 \pm 0.1)$ | $0\ (0.18 \pm 0.066)$ | $0\ (0.145 \pm 0.09)$ |
| sd=20 ms | $0.04\ (0.41 \pm 0.21)$ | $0\ (0.21 \pm 0.16)$ | $0.01\ (0.2 \pm 0.1)$ | $0(0.17 \pm 0.11)$ |
| sd=30 ms | $0.08\ (0.47 \pm 0.19)$ | $0.05\ (0.31 \pm 0.17)$ | $0.06\ (0.28 \pm 0.11)$ | $0.07\ (0.27 \pm 0.11)$ |

I used the evolutionary algorithm to obtain 10 independent networks for each setting (160 networks in total, 4 different network sizes $\times$ 4 different noise levels $\times$ 10 independent runs). When the noise was present during the evolution, each network was evaluated 100 times with random noise, and $f_{err}$ was calculated as the average of the 100 evaluations. I tested each evolved network 100 times for each noise level (no need to test the network 100 times when no noise is present) and calculated the average $f_{err}$, then I calculated the average value and the standard deviation of the average $f_{err}$ for the 10 individual networks with the same settings. The results are shown in Tables 5.1 - 5.4.

### 5.4.3 Network analysis

We can see in the results that it is possible to evolve a network with 2 interneurons which is robust to noise (Table. 5.2). Moreover, it was straightforward to analyse the topology

Table 5.3: Evolution of robustness to noise in a temporal pattern recognition task in LIF networks with 5 interneurons. The networks evolved with various level of noise in the input (Gaussian noise added to spike times, centered at zero and with indicated standard deviation, sd) were tested with the same (or different) level of noise that (than) was used during evolution. The first value shows the performance of the best network in 10 runs, the values in brackets are averages ± sd.

| Testing | Temporal noise during evolution | | | |
|---|---|---|---|---|
| | no noise | sd=10 ms | sd=20 ms | sd=30 ms |
| no noise | $0\ (0 \pm 0)$ | $0\ (0.06 \pm 0.13)$ | $0.05\ (0.09 \pm 0.11)$ | $0.05\ (0.06 \pm 0.02)$ |
| sd=10 ms | $0\ (0.4 \pm 0.25)$ | $0\ (0.07 \pm 0.09)$ | $0.006\ (0.18 \pm 0.06)$ | $0.03\ (0.18 \pm 0.05)$ |
| sd=20 ms | $0\ (0.42 \pm 0.25)$ | $0\ (0.13 \pm 0.12)$ | $0.03\ (0.23 \pm 0.07)$ | $0.07\ (0.2 \pm 0.05)$ |
| sd=30 ms | $0.04\ (0.5 \pm 0.23)$ | $0.054\ (0.24 \pm 0.15)$ | $0.07\ (0.33 \pm 0.01)$ | $0.08\ (0.29 \pm 0.08)$ |

Table 5.4: Evolution of robustness to noise in a temporal pattern recognition task in LIF networks with 10 interneurons. The networks evolved with various level of noise in the input (Gaussian noise added to spike times, centered at zero and with indicated standard deviation, sd) were tested with the same (or different) level of noise that (than) was used during evolution. The first value shows the performance of the best network in 10 runs, the values in brackets are averages ± sd.

| Testing | Temporal noise during evolution | | | |
|---|---|---|---|---|
| | no noise | sd=10 ms | sd=20 ms | sd=30 ms |
| no noise | $0\ (0 \pm 0)$ | $0\ (0.02 \pm 0.06)$ | $0\ (0.06 \pm 0.08)$ | $0\ (0.16 \pm 0.08)$ |
| sd=10 ms | $0.03\ (0.4 \pm 0.29)$ | $0\ (0.02 \pm 0.06)$ | $0\ (0.05 \pm 0.08)$ | $0\ (0.165 \pm 0.08)$ |
| sd=20 ms | $0.14\ (0.46 \pm 0.25)$ | $0\ (0.04 \pm 0.08)$ | $0\ (0.08 \pm 0.1)$ | $0.01\ (0.18 \pm 0.08)$ |
| sd=30 ms | $0.4\ (0.54 \pm 0.2)$ | $0.07\ (0.14 \pm 0.01)$ | $0.02\ (0.17 \pm 0.1)$ | $0.05\ (0.24 \pm 0.09)$ |

of the network with 2 interneurons. I performed 10 independent evolutionary runs of networks with 2 interneurons in the presence of temporal noise with 10 ms standard deviation. After checking the topology of the final networks we can see that these 10 champions can be divided into 3 main categories as following:

#### 5.4.3.1 Category 1

In 3 out of 10 champions (Fig. 5.5), one interneuron (1) acts as a detector of a spike in input (IN2); once this interneuron starts firing at high frequency (because of a recurrent excitatory connection), the second interneuron (2) goes to a plateau state—the membrane potential is higher than the resting potential, but still sub-threshold—. When the third input (IN3) generates a spike, it increases the firing rate of interneuron (1) putting interneuron (2) in a higher plateau. The membrane potential of interneuron (2) is slowly increasing until it reaches the threshold at around 350 ms. Because of this plateau state, the activation of the interneuron (2) does not depend on the time between spikes in the second input (IN2) and the third input (IN3) (which allows for the robustness to noise).

Interneuron (2) acts as a detector of the sub-pattern interneuron (1) - input (IN3). This is why the spike from input (IN2) should arrive before the spike of input (IN3), so the network will be silent for the patterns 1-3-2, 3-2-1, and 3-1-2. Input (IN1) inhibits both interneurons, so they remain silent when presented with input patterns 2-3-1 or 2-1-3. With 1-2-3, interneuron (2) starts spiking at 350 ms (100 ms after the spike on input, at 250 ms), allowing for robustness to noise on input (IN1).

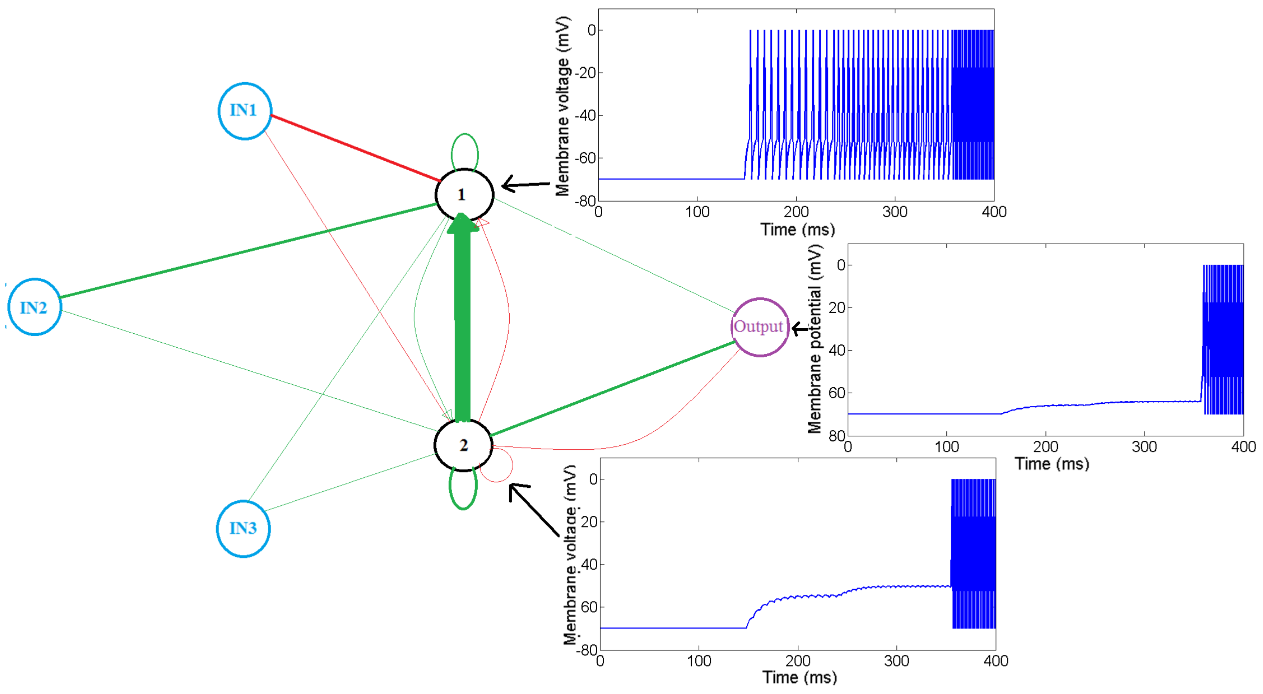


FIGURE 5.5: The final SNN of 3 inputs (cyan) each with only one spike at the times: 50 ms, 150 ms, and 250 ms, respectively, 2 interneurons (black), and one output (purple). The neurons are connected with both excitatory connections (green) and inhibitory connections (red). The thickness of lines signifies the strength of the connections. The actual simulation time is 1000 ms, but for clarity only up to 400 ms are shown, the neurons continue spiking afterwards. Note: The effect of inputs spike is shifted forward or backward due to the noise.

#### 5.4.3.2 Category 2

In 5 out of 10 champions (Fig. 5.6), one interneuron (1) starts spiking after the spike in input (IN1) and thanks to the positive feedback loop it keeps firing, putting the other interneuron (2) into a plateau state. When the spike from input (IN2) arrives, interneuron (1) starts firing at a higher rate, putting interneuron (2) into a higher plateau state. Only a small stimulus—a spike from input (IN3)—is now required so that interneuron (2) starts spiking, and thus also the output. Here also the plateau state allows for robustness to noise—the times between spikes in input (IN1) and (IN2), or (IN2) and (IN3) have no effect on the pattern recognition.

The spike from input (IN1) should arrive first in order to activate interneuron (1) which then puts interneuron (2) into a plateau, thus the patterns 2-1-3, 2-3-1, 3-1-2, and 3-2-1 do not allow the output neuron to be active. It is not clear from Fig. 5.6 why the pattern 1-3-2 does not activate the output neuron. I checked the exact values of the weights of the synapses and found that input (IN2) has a stronger inhibitory synapse with interneuron (1) than the inhibitory synapse between input (IN3) and interneuron (1). While input (IN3) has a stronger excitatory synapses with interneuron (1) than the excitatory synapses between input (IN2) and interneuron (1). This structure allows the sub-pattern 2-3 to have more effect on interneuron (1) than the pattern 3-2, thus the pattern 1-3-2 does not allow the output to be active.

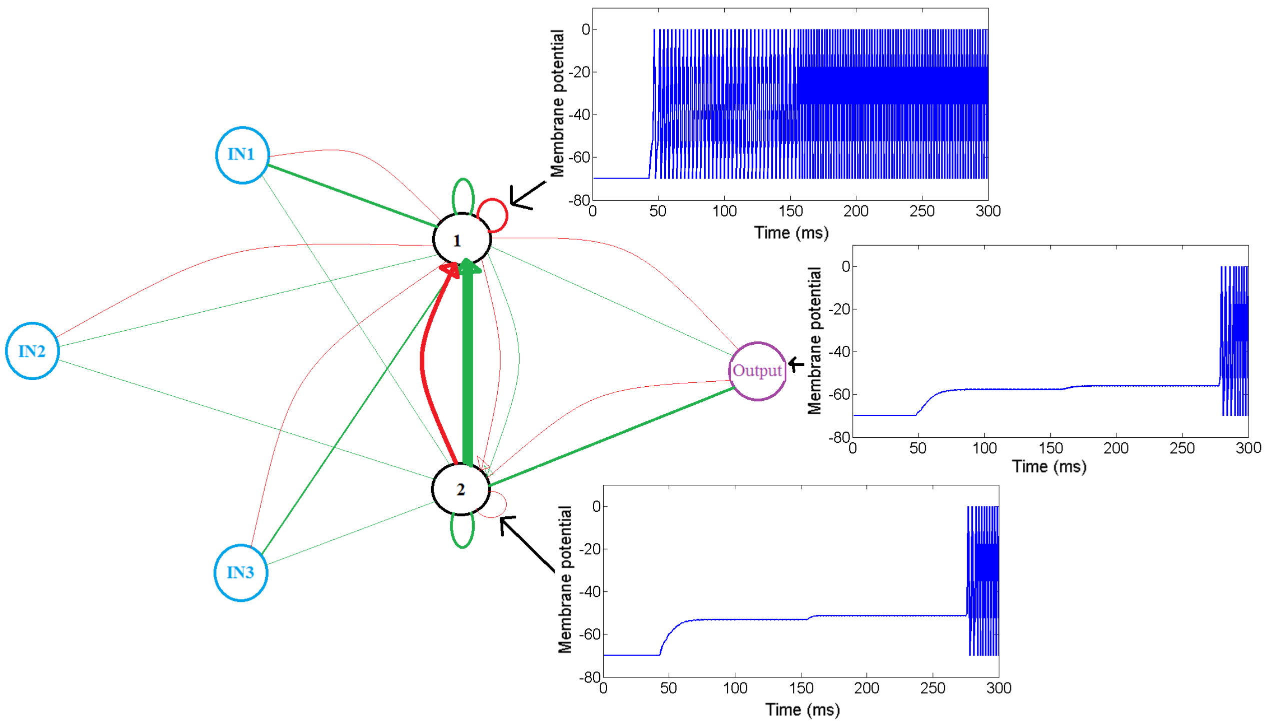


FIGURE 5.6: The final SNN of 3 inputs (cyan) each with only one spike at the times: 50 ms, 150 ms, and 250 ms, respectively, 2 interneurons (black), and one output (purple). The neurons are connected with both excitatory connections (green) and inhibitory connections (red). The thickness of lines signifies the strength of the connections. The actual simulation time is 1000 ms, but for clarity only up to 300 ms are shown, the neurons continue spiking afterwards. Note: The effect of inputs spike is shifted forward or backward due to the noise.

#### 5.4.3.3 Category 3

2 champions out of 10 evolved with 10 ms noise (Fig. 5.7) were in fact only robust to less noise than for which they were evolved, on the timing of the spike in input (IN3)—even though they had zero error when tested with 10 ms noise. This is because one interneuron (1) acts as a detector of the time interval between spikes in input (IN1) and (IN2). If this interval is higher than the network was evolved for, the pattern is not recognized. Otherwise, once interneuron (1) puts the second interneuron (2) into the plateau state, any timing of the spike in input (IN3) is sufficient for correct pattern recognition.

The spike from input (IN2) is not enough to make interneuorn (1) active alone. Input (IN1) should precede input (IN2) in order to make interneuron (1) active, so interneuron (1) acts as a 1-2 sub-pattern detector. For this reason, the patterns 2-1-3 and 2-3-1 do not allow the output to be active. The excitatory synapse between input (IN1) and interneuron (1) is very strong comparing with the excitatory synapse between input (IN3) and interneuron (1). That is why the inputs sub-pattern 3-2 does not activate interneuron (1), thus the patterns 1-3-2 and 3-2-1 can not make the network active. Despite the excitatory synapses between input (IN3) and interneuron (1) is not very strong, but it is enough to increase the firing rate of interneuron (1) to make interneuron (2) active. As a result, the pattern 3-1-2 does not allow interneuron (2) to be active.

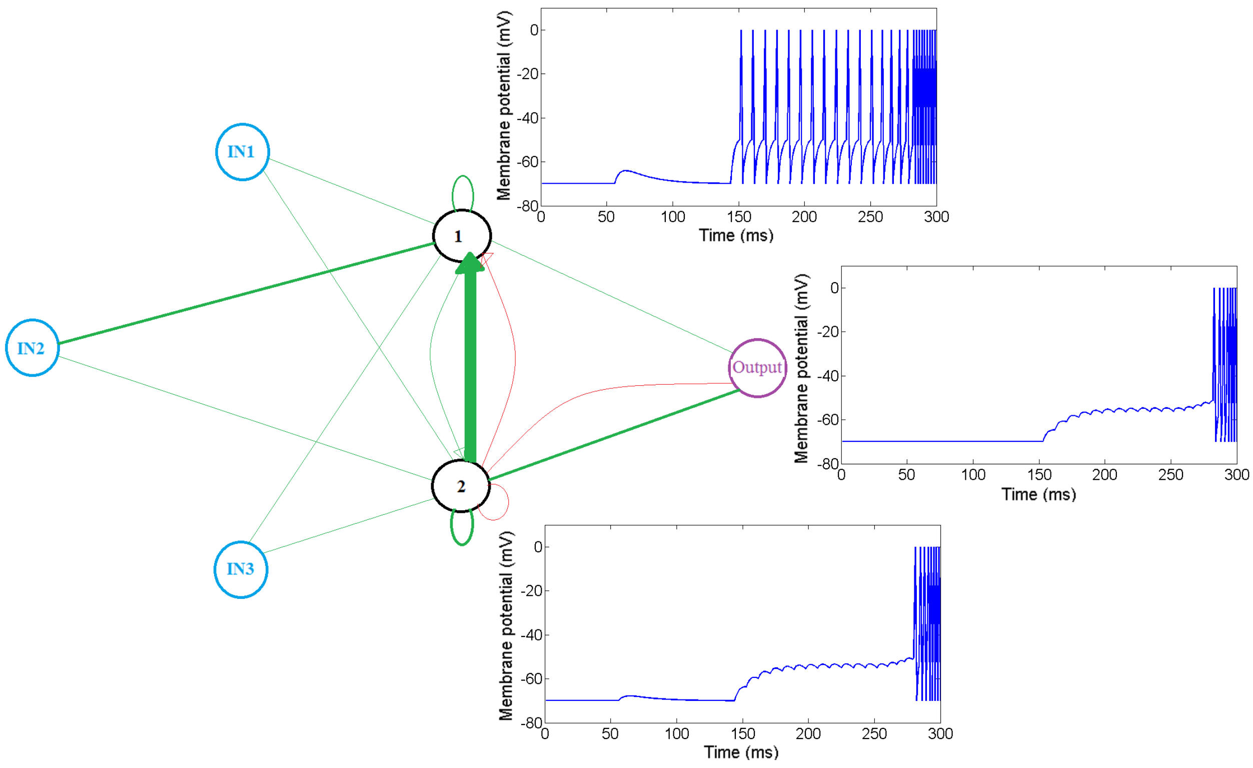


FIGURE 5.7: The final SNN of 3 inputs (cyan) each with only one spike at the times: 50 ms, 150 ms, and 250 ms, respectively, 2 interneurons (black), and one output (purple). The neurons are connected with both excitatory connections (green) and inhibitory connections (red). The thickness of lines signifies the strength of the connections. The actual simulation time is 1000 ms, but for clarity only up to 300 ms are shown, the neurons continue spiking afterwards. Note: The effect of inputs spike is shifted forward or backward due to the noise.

## 5.5 Conclusion

After checking the results of evolving and testing the SNNs with 1, 2, 5, and 10 interneurons with different noise levels (Table 5.1 - 5.4), we can clearly see that there is significant improvement in the values of the $f_{err}$ when we use 2 interneurons instead of 1 interneuron. On the other hand, we can see that there is no big difference in the behaviour of the networks with 2, 5, and 10 interneurons. These results would not be surprising at all after the analysis of the SNNs with 2 interneurons which are robust to the noise (Fig. 5.5 - 5.7).

The analysis shows that the excitatory feedback loops play a crucial role in transferring the network from one plateau state to another, until it becomes active. This plateau behaviour of the network requires at least 2 interneurons in the network (1 interneuron with a positive feedback loop put the other interneuron in the plateau state). We can now also understand why the networks with just one interneuron cannot reach 0 error when noise is present during the evolution.

It was expected that it would be more difficult to evolve large networks with five and ten interneurons than the network with only one and two as large networks have wider search space. The reason behind that could be that the task was too easy for SNNs.

# Chapter 6

# Real Time Control of Foraging Behaviours

## 6.1 Introduction

Olfactory system is used by many animals for food foraging[19]. Olfactory sensory neurons collect information about odours in the environment. This information is encoded as spikes and sent to the brain through the axons of the sensory neurons [67]. The foraging system uses this information to locate and follow food sources (targets) in the environment.

Many coding strategies were proposed as a proper way to encode the information collected from the environment to be used by olfactory system. These coding strategies include the firing rate, the number of the active sensory neurons, and the synchronization of firing between the sensory neurons [19].

Oros and collaborators have investigated the ability of evolving SNNs to control animats for foraging [22]. I built up on their work and extended it by presenting more encoding strategies and allowing more targets in the environment.

It is arbitrary that sensory information represents smell (concentrations of chemical substances) in this chapter. It could as well be any signal that attenuates with distance (light, sound). To simplify the discussion in this chapter, I will be using the metaphor of food sources and an animat that can sense the smell of food.

The GReaNs platform was used in earlier work to explore the ability of evolving GRNs to control real time behaviour [26, 27]. Here I explain how I extended this previous work to allow the use of SNNs instead of GRNs.

## 6.2 The Model

### 6.2.1 Spiking Neural Networks Model

I used the LIF SNN model described in chapter 3 for the animat foraging task as it is considered more simple than the AdEx SNN model.

### 6.2.2 Animat Simulation

A very simple model was used to represent the animat in GReaNs [26, 27]. Each animat is modelled as a circular object with two sensors on each side of the front of the animat and two actuators on each side of the back of the animat. Each sensor is connected to an input neuron, while each actuator is connected to an output neuron. The animat is simulated in a 2D environment, where food resource objects are placed randomly. In the previous work [26, 27], the strength of the food smell generated from each food resource was represented as a circle, where the maximum strength is at the centre of the circle and it gradually decreases moving away from the centre (Fig. 6.2). The strength of the food smell generated by each food source is determined by the following equation:

$$s = \frac{1}{1 + \alpha d} \tag{6.1}$$

where $s$ is the strength of smell at the sensor generated by a single food source, $d$ is distance between the sensor and the food resource, and $\alpha$ is a scaling factor. In the work that had been done before for using GRN to control the agent [26, 27], $\alpha = 0.05$ was used. I needed to increase the scaling factor in order to increase the range of the food particles. Increasing the range of the food particles helped in allowing the animat to smell the food when it was far from them. I used $\alpha = 0.2$ during all the simulations.

Based on the distance between the animat and the surrounding food resources, the strength of the smell of the food is calculated at both sensors by adding the strengths generated by each food resource individually.

When an output neuron is active, the corresponding actuator generates force (thrust) in the direction indicated in Fig. 6.1. The length of this vector is proportional to either:

- The difference between the membrane potential of the output neuron and the threshold. When the membrane potential crossed the threshold, the force of the corresponding actuator is update with the value:

$$Thrust = V - V_{th} \tag{6.2}$$

where $Thrust$ is the value of the thrust generated at the corresponding actuator, $V$ is the membrane potential of the output neuron, and $V_{th}$ is the threshold potential of the output neuron.

This was the first way I implemented to update the actuators force. I decided to start with this method as it was very close to the way the actuators force was updated when GRNs were used, before moving to more realistic methods from the point of view of neuroscience.

- Constant value. When the membrane potential of the output neuron fires a spike, the force of the actuator is increased by a constant value as following:

$$Thrust = \beta \tag{6.3}$$

where $V$ is the membrane potential and $\beta$ is a constant.

- The number of spikes generated by the output neuron during a sliding window. The force at the actuator is updated with the number of spikes fired by the corresponding output neuron within a sliding window with a fixed width, divided by a constant factor. Since the refractory period is 1 ms (we can not have more than one spike each 2 ms as the maximum firing rate is 500 Hz), the force at the actuators is updated each 1 ms.

$$Thrust = \frac{N}{\gamma} \tag{6.4}$$

where $N$ is the number the spikes generated by the corresponding output neuron during the sliding window, and $\gamma$ is a constant factor.

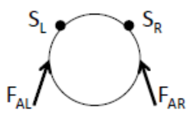


FIGURE 6.1: The model of the animat evolved in order to collect targets on 2D maps. The animat has 2 sensors ($S_L$ and $S_R$) on the front and 2 actuators ($A_L$ and $A_R$) on the back. The two actuators generate forces in the directions $F_{AL}$ and $F_{AR}$ [26].

The movement of the animat is controlled by simple Newtonian physics that was implemented before [26, 27]. Not only do the spikes of the output neurons affect the force of the actuators, but the animats are also affected by the fluid drag which decreases the velocity of the animats. The effect of the fluid drag on the force of the actuator is described by the following equation:

$$F_{new} = F_{old} - c_d d|\overrightarrow{v}| \qquad (6.5)$$

where $F_{new}$ is the new force, $F_{old}$ is the old force, $c_d$ is the drag coefficient, $d$ is the diameter of the animat, and $\overrightarrow{v}$ is the velocity of the animat. $c_d = 1$ was used in all the tasks of evolving animats in this thesis.

Due to this fluid resistance, the speed of the animat gradually decreases in the case when both of the actuators are not active. If one actuator is on and the other is off, the animat will move in a circle, as it is not allowed to rotate in its initial position. This feature was implemented in the previous work by adding a rotation drag force to the system which worked against the rotation of the animat [26, 27].

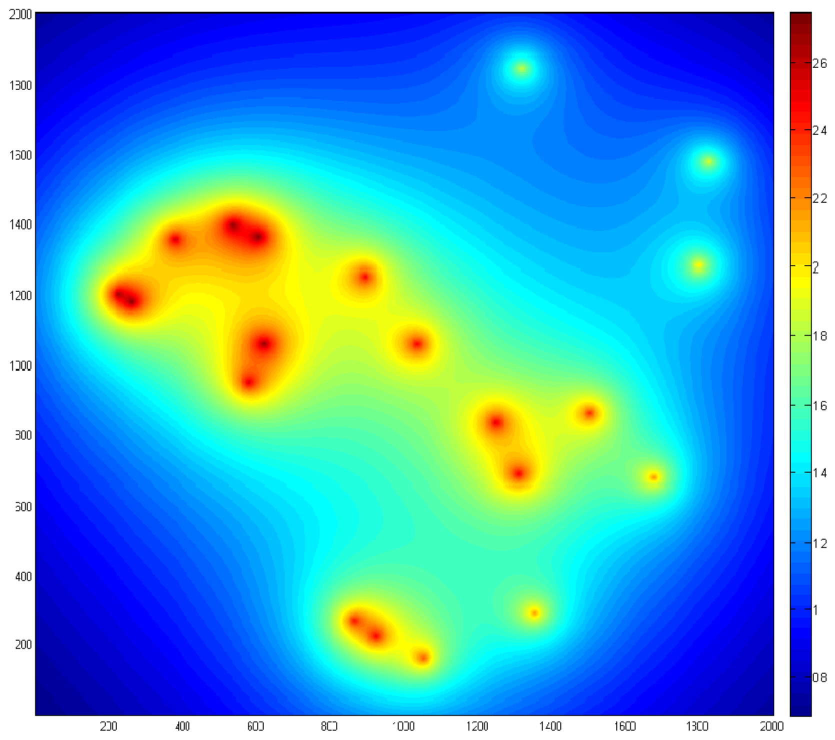


Figure 6.2: The gradation of the strength of food smell generated by 20 targets in GReaNs.

### 6.2.3 Genetic algorithm

As in the previous tasks, the first generation in each evolutionary run was initiated with 300 random SNNs, each with five neurons, the size of the population in each generation was kept constant. In the work that had been done before by Joachimczak and colleagues to evolve GRNs for animat foraging [26, 27], crossover and elitism were disabled. Disabling the elitism means that mutations were applied on all the genomes in the previous generation in order to generate the new generation.

Disabling the elitism put a pressure on the evolutionary algorithm in order to evolve GRNs which could be robust to mutations. Joachimczak and colleagues also reduced the probability of duplication and deletions of regulatory elements, used in the previous tasks, by an order of magnitude in order not to loose the successful genomes very easily. I used the same probability of genetic operators as used previously as this probability already showed that it was successful for animat foraging task.

I also used the same function which was used in the previous work in GReaNs [26, 27] to calculate the error value:

$$f_{err} = 1 - \frac{collected}{total} \tag{6.6}$$

where *collected* is the number of the food resources and *total* is the total number of food resources in the simulation environment. The value of $f_{err}$ varies from 0 for the best animat to 1 for the worst animat.

In the foraging tasks done before with GReaNs when artificial GRNs controlled the behaviour of the neurons [26] it was noticed that this error function led to inefficient behaviour of the animat moving in circles towards the food resources. An example of such behaviour can be seen in Fig. 6.3. The animat moves in circles instead of moving in a smooth path. The animat had this behaviour because the right actuator was always dominating the left actuator. In order to solve this problem, Joachimczak and colleagues multiplied the total error value of the animats that change the direction of their movement at least once during the simulation by 0.9.

In the tasks presented in 6.3.1 and 6.3.2.1, the animats did not show this suboptimal behaviour, so I did not add the change direction reward. I did not add this reward in the tasks presented in 6.3.2.2 and 6.3.2.3 as well, but adding this reward would be useful as half of the evolved animats suffered from this circling movement. In the last tasks in this chapter in 6.3.3, it was mandatory to use this reward as most of the evolved animats suffered from this suboptimal behaviour. I increased the effect of the reward by multiplying the total value of $f_{err}$ by 0.8 instead of 0.9. I will talk again about this change direction reward when I describe each task separately.

During the evolution, the animats were evaluated using more than one random map. The value of $f_{err}$ of each animat was calculated as the average of the $f_{err}$ on all maps.

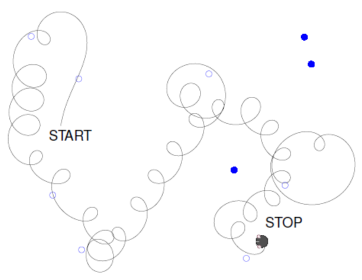


FIGURE 6.3: A suboptimal behaviour of the agent that has been discovered before in GReaNs [26] when the GRN was used to control the agent.

Using a large number of maps during the evolution would allow getting a more exact estimate of fitness for each animat, but it also penalized the animats that moved fast but not in the direction of targets (but that could hit some sources on some of the few random maps by chance) at the beginning of the evolution. Although this movement of the animat was not the goal of the evolution, this movement at the beginning of the evolution could lead to targeted movement (moving toward the target) during the subsequent generations.

On the other hand, using small number of maps could result in obtaining a lucky animat that collects the targets with directional movement. So it was necessary to find the suitable number of maps for evaluating the animats.

## 6.3 Encoding sensory information in the SNNs in GReaNs

I implemented different methods to encode the strength of the food smell calculated at the two sensors.

### 6.3.1 The strength of food smell at sensors to synaptic conductance injection

The first encoding method I implemented to represent the sensory information was the synaptic conductance injection encoding. In this encoding, synapses of all the neurons connected to the input neurons are updated based on the strength of food smell at the sensors.

As a result of the small distance between the two sensors of the animat comparing to the simulation environment, the difference of the strengths of food smell at the right sensor ($S_R$) and the left sensor ($S_L$) was too small that made it difficult to evolve an animat able to find the location of the targets. In the work that had been done before in GReaNs [26, 27], the following sigmoid functions were used:

$$S_1 = \frac{1}{1 + e^{-\epsilon(S_R - S_L)}} \tag{6.7}$$

$$S_2 = \frac{2}{1 + e^{-\epsilon(S_R + S_L)}} - 1 \tag{6.8}$$

where $\epsilon$ was used to control the steepness of the function. $\epsilon = 10$ was used.

Based on the type of the synapses (excitatory or inhibitory) the excitory or the inhibitory synaptic conductance, of the neurons connected to the input neurons, was updated by the one of the following equations:

$$\dot{g_E} = \lambda S \ \ or \ \ \dot{g_I} = \lambda S \tag{6.9}$$

where $S$ could be $S_1$ or $S_2$ and $\lambda$ is a constant. $\lambda = 0.5$ was used.

I used equation 6.2 to update the force of the actuators for the animat in this task.

I did not use the 0.8 decrease of the total error value was rewarded to the animats that change the direction of their movement at least once during the simulation in this task.

In this task, I tried to find the best simulation time for the evaluation of the animat, the suitable number of random maps used for evaluation, and the suitable number of generations for the evolution.

In the first settings, I used 2000 steps (2 s as each step corresponds to 1 ms), 24 random maps for evaluation, and I ran it for 1000 generations. Using these settings, I got an animat with $f_{err} = 0.4979$ (Fig. 6.4). This animat had $f_{err} = 0.527$ when it was tested with other 24 random maps. This animat was able to collect only 11 targets out of 20 targets when it was simulated for 2 s (the same simulation time used during the evolution).

As I explained at the beginning of this chapter, the number of maps used for evaluating was a trade-off. For this reason, I used four random maps for evaluating the animat during the first 500 generations in order to allow directional movement at the beginning of the evolution. Using only four maps allowed also to have high variability in the behaviour of the animats. This high variability allowed promoting big advances early in the evolution. However, late in evolution, the advances are small, so the evaluation

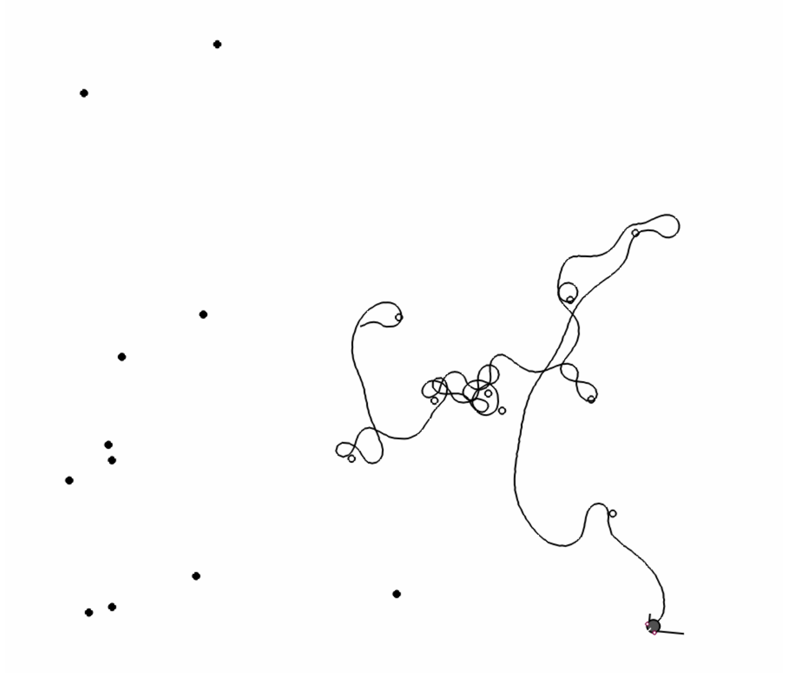


Figure 6.4: The behaviour of the animat that had been evolved using simulation time 2 s and synaptic conductance injection coding when it was simulated for 2 s. The gray circle represents the animat, black circles represent the uncollected targets, white circles represent the collected targets and the continuous line represents the path of the animat during the simulation.

needed to be more precise. For this reason, I increased the number of random maps used for evaluation to 24 in the last 100 generations (600 generations in total).

It was obvious that the animat shown in Fig. 6.4 needed more time in order to collect all targets. I increased the simulation time used during the evolution to 4000 steps (4 s) and started the evolution from the beginning again, but this did not help much.

I increased simulation time used during the evolution to 10 s. As I mentioned above, I decreased the number of random maps used during the evaluation to four maps during the first 500 generations. I ran the evolutionary algorithm for 100 generations more using 24 maps. At the end of the evolution, I got an animat with $f_{err} = 0$ and it had the same value for generalization. Fig. 6.5 shows that the animat was able to collect all targets, however, the simulation time was quite long that gave the opportunity to the animat to move slowly and take a long path.

I repeated the previous task again, but after reducing the simulation time used during the evolution to 6000 steps (6 s). After 600 generations, the best animat I got had $f_{err} = 0.0375$. When this animat was simulated for 6 s, it did not manage to collect all the targets, however after prolonging the simulation time to 10 s (Fig. 6.6), this

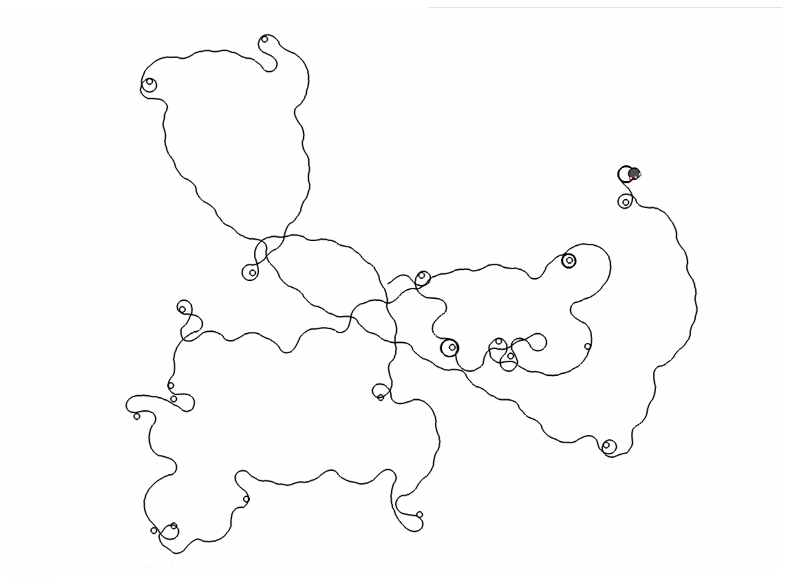


FIGURE 6.5: The behaviour of the animat evolved using simulation time 10 s and synaptic conductance injection coding when it was simulated for 10 s. The gray circle represents the animat, black circles represent the uncollected targets, white circles represent the collected targets and the continuous line represents the path of the animat during the simulation.

animat managed to collect all the targets. Reducing the simulation time used during the evolution from 10 s to 6 s allowed having an animat with higher speed.

In order to make sure that I can always get an animat able to collect all the targets and that obtaining the animat shown in Fig. 6.6 was not just some fluke in a particular run, I investigated the behaviour of 10 champions from independent runs. In each run, I used simulation time 6 s, four random maps for evaluation until the 500$^{th}$, and 24 random maps for the last 100 generations.

The best animat I got out of the 10 champions had $f_{err} = 0$. This animat was able to collect all the targets on the map (20 targets) when only 6 s were used for simulation (Fig. 6.7). Note that Fig. 6.6 and 6.7 show the behaviour of champions from different independent runs on the same random map for comparison.

I investigated the evolutionary history of the best animat shown in Fig. 6.8. As we can see in Fig. 6.8 (lower panel), the curve goes down steeply during the first 100 generations. In the period between the 100$^{th}$ generation and the 400$^{th}$ generation, the curve goes down less steeply but it oscillates very quickly until it reaches 0 before the end of the 400$^{th}$ generations. This oscillation behaviour resulted from the fact that the possible $f_err$ values are discrete as there are only 80 targets in total on four maps.

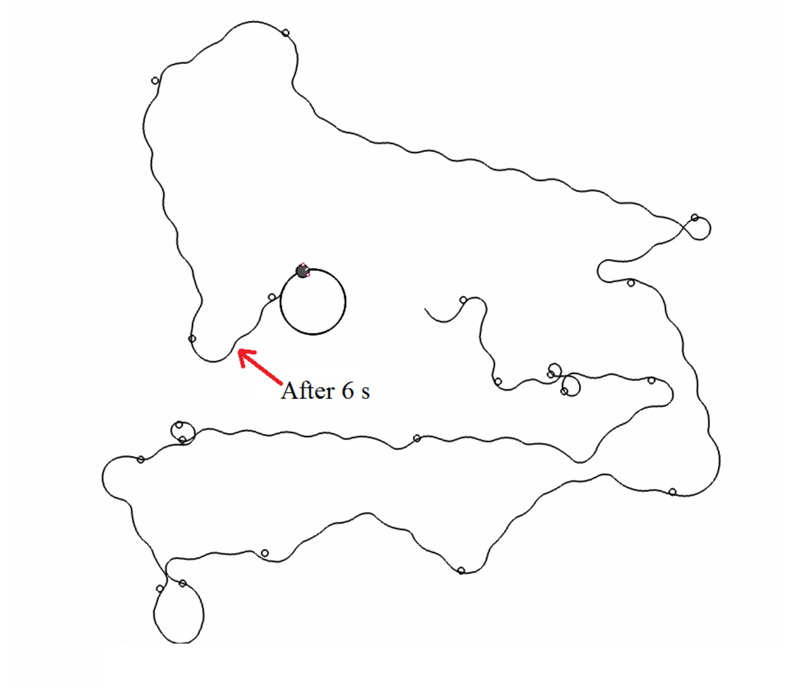


FIGURE 6.6: The behaviour of the evolved animat using synaptic conductance injection coding using simulation time 6 s during the evolution when simulated for 10 s. The gray circle represents the animat, black circles represent the uncollected targets, white circles represent the collected targets and the continuous line represents the path of the animat during the simulation. The red arrow shows where the animat stopped when it was simulated for only 6 s.

At generation 500, the number of maps used for evaluation the animat was increased from four to 24. We can see that this increased the error at the beginning of the last 100 generations, but it goes down again to reach 0 at the end of the evolution.

If we zoom in to compare the trajectory of the $f_{err}$ between the 100$^{th}$ generation and the 400$^{th}$ generation Fig. 6.8 (upper left panel) with the trajectory of the $f_{err}$ between the 500$^{th}$ generation and the 600$^{th}$ generation (upper right panel), we will see that the range of oscillation is less in the last 100 generations. This decrease in the range of oscillation resulted from increasing the number of random maps used for evaluation from four to 24 maps.

The average $f_{err}$ of the best evolved animats from the 10 champions was 0.14 with standard deviation 0.12. Despite no reward was added in order to get rid of circling movement, none of these 10 animats suffered from this suboptimal behaviour.

In order to get an estimate of the fitness values of the best animats out of the 10 champions, I re-evaluated these animats on another 100 random maps. I used 100 random maps as it gave a less variable estimate of fitness than the numbers of maps used during the evolution (4 and 24). I used the same number of targets during the

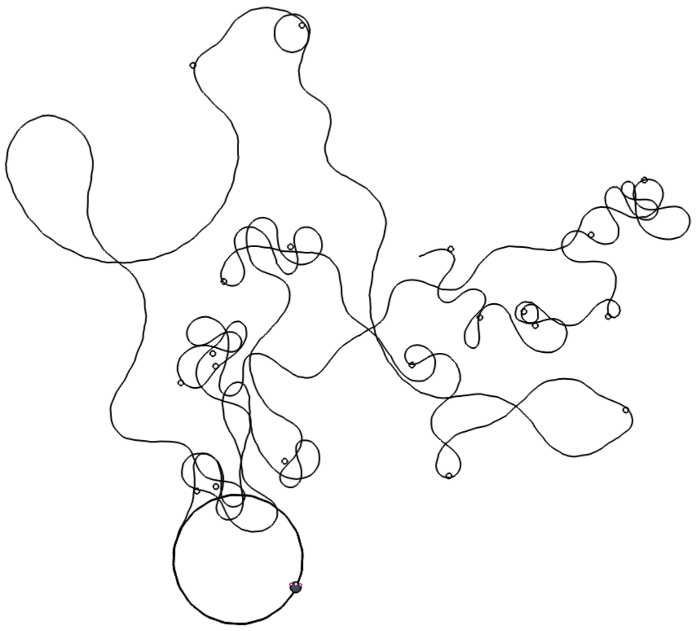


FIGURE 6.7: The behaviour of the best evolved animat using synaptic conductance injection coding using simulation time 6 s during the evolution. The gray circle represents the animat, black circles represent the uncollected targets, white circles represent the collected targets and the continuous line represents the path of the animat during the simulation.

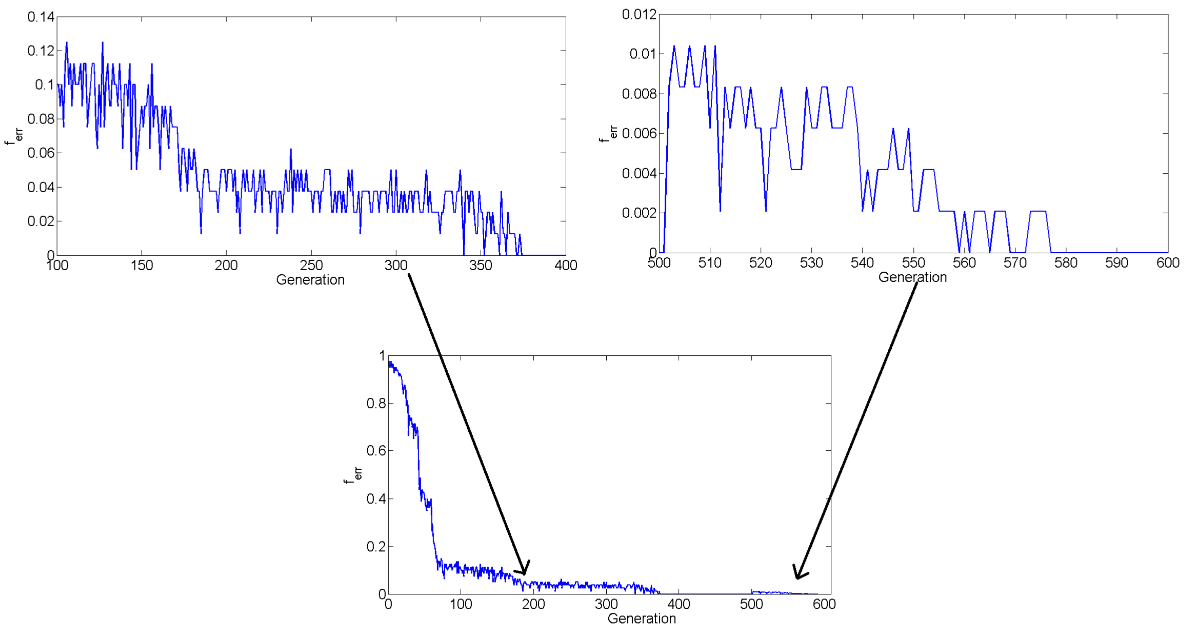


FIGURE 6.8: The evolution history of the best animat evolved using conductance injection coding.

evaluation (20 targets) and I increased the simulation time to 12 s in order not to penalize the slow animats. The average (over the 10 champions) of the average (over the 100 evaluation) $f_{err}$ was 0.07 and the standard deviation of the average $f_{err}$ was 0.06. When the animats were re-evaluated using 100 random maps again, the differences in the estimate $f_{err}$ were about about 0.01.

In the 10 champions evolved here using conductance injections, 20 targets were used. Increasing the number of targets on the map will increase the density of the food concentration. In order to test if the animats evolved using 20 target will be able to cope with higher food density or if they will break down (malfunctioning of the network, stemming possibly from the recurrence, would result in a perfectly circular trajectory before all the targets are reached), I tested the best evolved animat out of the 10 champions after re-evaluating them with 100 random maps.

The initial plan for checking the fragility of the evolved networks was to keep doubling (40, 80, ...etc) the number of targets and testing the best animat until it breaks. After the largest number of targets for which the animat would not malfunction was determined by doubling, I would again perform the search above this number using a logarithmic scale, but now multiplying by 1.1, not 2. When it breaks I took the largest successful number as an indicator for the limit of this animat.

I did not limit the simulation time as movement speed of the animat was not important in this test.

After applying this plan, the best animat showed an interesting reaction. I kept doubling the number of targets until it reached 20480 without breaking (to move in circles or to stop moving at all). Due to computational power limitation of my computer, I was not able to test the animat with more targets. Using the conductance injection encoding resulted in obtaining a SNN very robust with increasing the food density in the environment.

### 6.3.2 Unary coding

The second encoding method I implemented to represent the sensory information was the unary coding. The unary coding is an encoding that is used to represent natural numbers. Each number $n$ is represented using this coding as $n$ ones followed by a zero (4 is represented as 11110). I used this coding to map the value of the food smell at the sensors to the number of active neurons in the input layer. The network had 100 neurons in the input layer which were initially not active. At each simulation step, based on the food smell at the sensors, the number of active neurons in the input layer was determined and varied from 0 to 100 neurons where 0 active neurons represented the minimum smell and 100 active neurons represented the maximum smell.

Although this encoding looks different from the previous one, if we will simplify them to equations, we will find that they are the same. The only difference is that the unary encoding has digital values (1, 2, ... 100) for the synaptic conductance injection, while the encoding explored in the section 6.3.1 has continuous values.

Here again I found 10 champions after 500 generation with four random maps and then 100 generations with 24 random maps.

I have applied the three methods of determining the thrust at the actuators using the unary coding. The results of these tasks can be described as following:

#### 6.3.2.1 Using the $(V - V_{th})$ thrust

The first method of updating the force at the actuators was the $(V - V_{th})$ thrust described in equation 6.2.

The best animat (Fig. 6.9) I got out of the 10 champions had $f_{err} = 0$. This animat was able to collect all the targets during the 6 s used as the simulation period during the evolution.

The evolutionary history of this animat (Fig. 6.10) shared the same properties of the evolutionary history of the best animat evolved using conductance injection. The value of $f_{err}$ goes down very steeply during the first 100 generations as well. The slope of the trajectory started to be less steep after the $100^{th}$ generation but the trajectory was oscillating very quickly with a wide range. After increasing the number of random maps, the trajectory started to oscillate less until it stabilized at around the $520^{th}$ generation.

The average of the $f_{err}$ of the 10 champions was 0.14 with standard deviation 0.18. Again here, none of the evolved animats suffered from the suboptimal behaviour despite not using the reward mechanism.

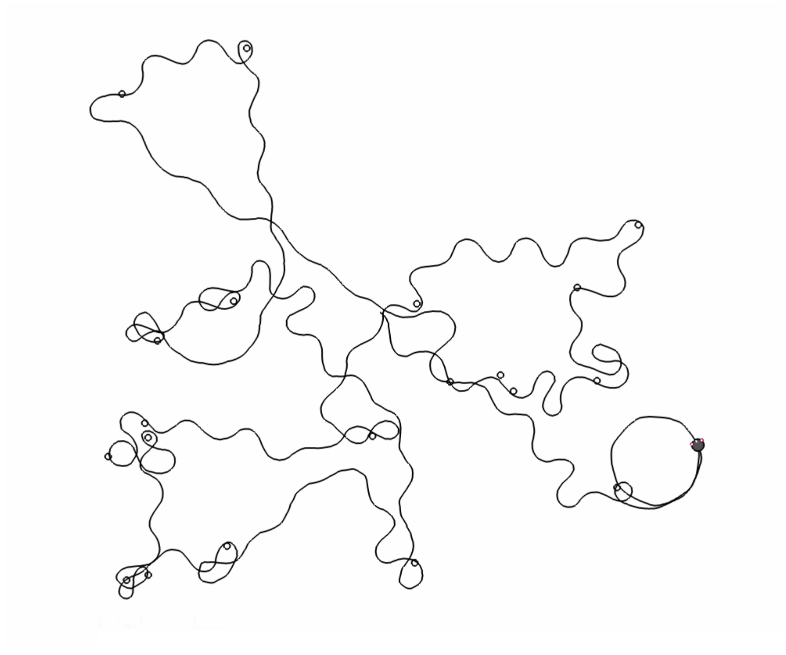


Figure 6.9: The behaviour of the best evolved animat using unary coding using simulation time 6 s. The gray circle represents the animat, black circles represent the uncollected targets, white circles represent the collected targets and the continuous line represents the path of the animat during the simulation.

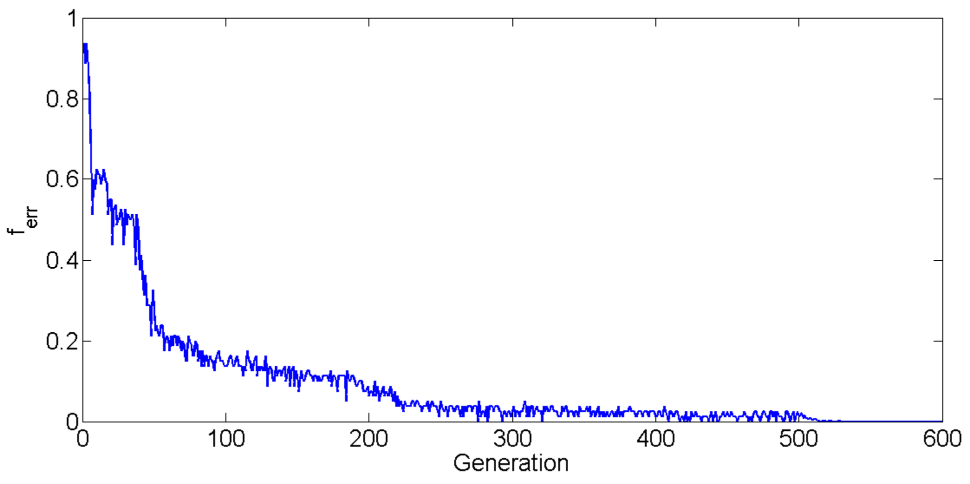


Figure 6.10: The evolution history of the best animat evolved using unary coding.

In order to estimate the ability of the best evolved animats from the 10 champions to collect 20 targets on any random map, I re-evaluated the best animat from each champion on 100 random maps using a simulation time 12 s with 20 targets. The average (over the 10 champions) of the average (over the 100 evaluation) $f_{err}$ was 0.06 and the standard deviation of the average $f_{err}$ was 0.09. When a given champion was re-evaluated using 100 random maps again, the difference in the $f_{err}$ tended to be about 0.001.

In order to test the fragility of the best animat evaluated on the 100 random maps when more food density was presented during the simulation, I followed the same testing plan as in section 6.3.1.

First I started with 40 targets which the animat was able to collect and I increased until I reached to 2560 which the animat was still able to collect. When I doubled the targets to 5120 the animat was able to collect only 658 targets before it started moving in a circle.

I reduced the targets to 2816 ($2560 \times 1.1$), but the animat was also not able to collect all of them and manged to collect only 2470 targets. Since the largest number of targets this animat managed to cope with was 2560, by this measure the best evolved animat here is more fragile than the one investigated in section 6.3.1.

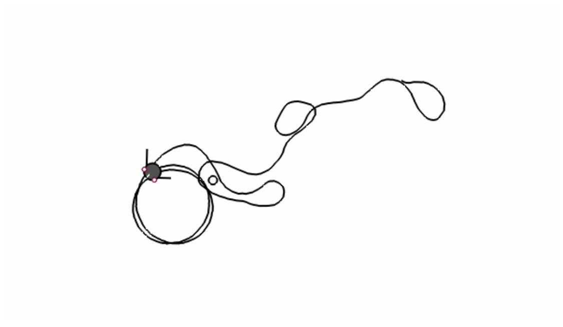
I added the ability to save the history of the thrusts at the actuators during the simulation. This feature will allow me to see how the $(V - V_{th})$ thrust affected the movement of the animat and how does this method differ from other methods of updating the force of the actuators.

I simulated the best animat out of the 10 champions two times. The first simulation was for 1 s when only one target was presented. In order to see if the density of the food concentration will affect the range of the thrusts, I simulated the behaviour of the animat again for 1 s but using 20 targets.

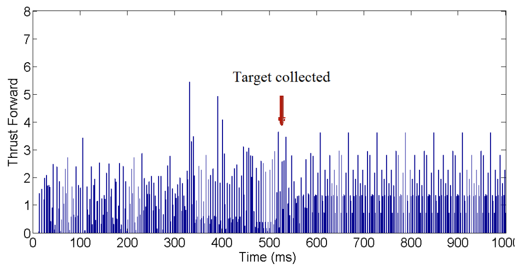
During each simulation I saved the thrust at the left and right actuators at each step. I used the value $\frac{L+R}{\sqrt{2}}$ as an indicator for the thrust forward and the value $\frac{L-R}{\sqrt{2}}$ as the thrust right (it will take a positive value if the animat turns right, and negative if it turns left, i.e. when the right actuator is more active than the left actuator).

Fig. 6.11 shows the simulation of the animat with one target. As we can see, the thrusts can take a wide range of value with high variance. During the last 400 steps and with absence of any concentration, the right thrust was negative; in other words the animat turned left (was on a circular trajectory, going anticlockwise).

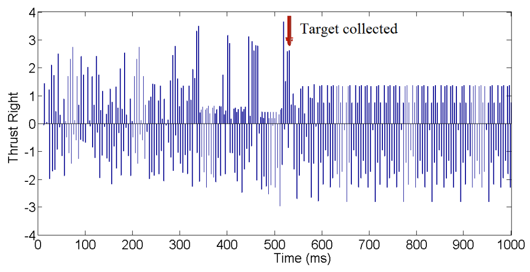
When the animat was simulated with 20 targets (Fig. 6.12), there was a slight increase at the strength of the forward and right thrusts.



(A) The trajectory of the best animat evolved using unary coding and $(V - V_{th})$ thrust when it was simulated for 1 s with only 1 target. The gray circle represents the animat, white circle represents the collected target and the continuous line represents the path of the animat during the simulation

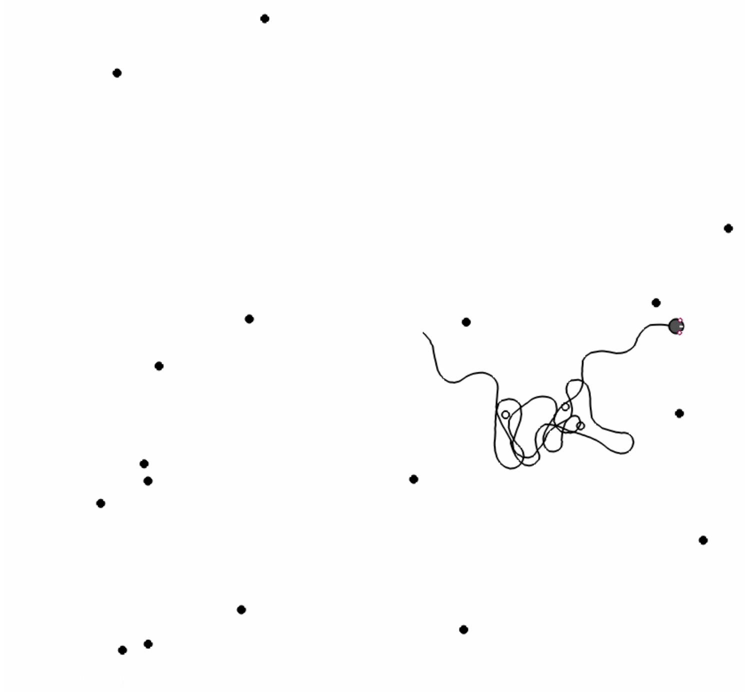


(B) The thrust forward generated by actuators during the simulation.

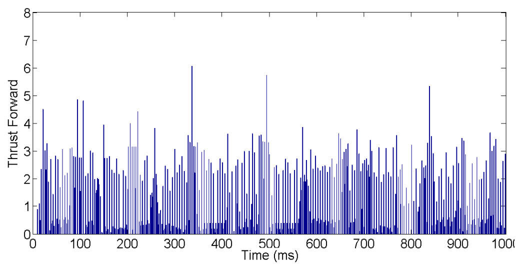


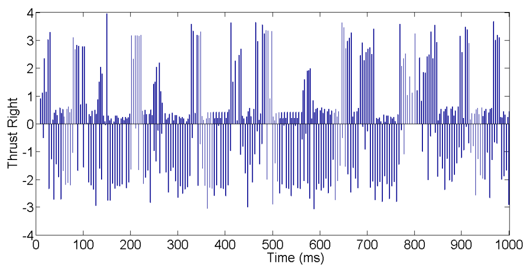
(C) The thrust right generated by actuators during the simulation.

FIGURE 6.11: The right and the forward thrusts generated by the actuators during the simulation of the best animat evolved using unary coding and $(V - V_{th})$ thrust with 1 target on the map.



(A) The trajectory of the best animat evolved using unary coding and $(V - V_{th})$ thrust when it was simulated for 1 s with 20 targets. The gray circle represents the animat, black circles represent the uncollected targets, black circles represent the uncollected targets, white circles represent the collected targets and the continuous line represents the path of the animat during the simulation





(B) The thrust forward generated by actuators during the simulation.

(C) The thrust right generated by actuators during the simulation.

FIGURE 6.12: The right and the forward thrusts generated by the actuators during the simulation of the best animat evolved using unary coding and $(V - V_{th})$ thrust with 20 targets on the map.

#### 6.3.2.2 Using the constant thrust

The second method I used to update the force of the actuators was using a constant thrust. When an output neuron fired a spike in a given simulation step, the corresponding actuator was active and a constant force (thrust) was applied to the animat in the direction shown in Fig. 6.1. I used equation 6.3 with $\beta = 2.5$. This value was close to the average thrusts at the left and right actuators when $(V - V_{th})$ thrust was used.

The best animat out of the 10 champions had $f_{err}$ 0.075. When this animat was simulated for 6 s (the same simulation time used during the evolution) it was not able to collect all the targets.

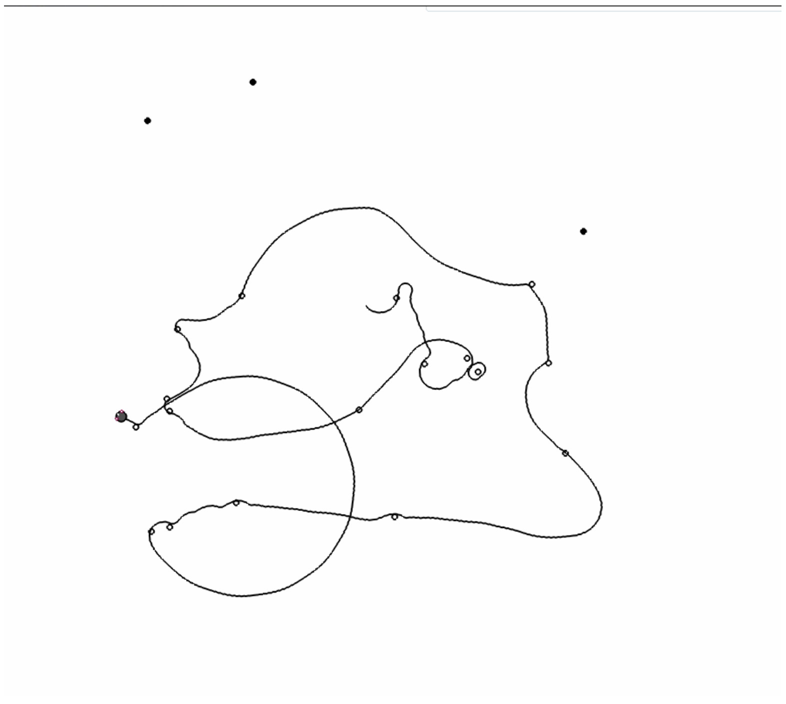


FIGURE 6.13: The behaviour of the evolved animat using unary coding and constant thrust using simulation time 6 s. The gray circle represents the animat, black circles represent the uncollected targets, white circles represent the collected targets and the continuous line represents the path of the animat during the simulation.

Nonetheless, when this animat was simulated for 12 s (Fig. 6.14), it was able to collect all the targets.

The average $f_{err}$ of the best evolved animats of the 10 champions was 0.32 with standard deviation 0.16. The change direction reward was not added here. Adding the change direction reward would improve the results as 5 out of the 10 best evolved animats suffered from suboptimal behaviour with moving in circles.

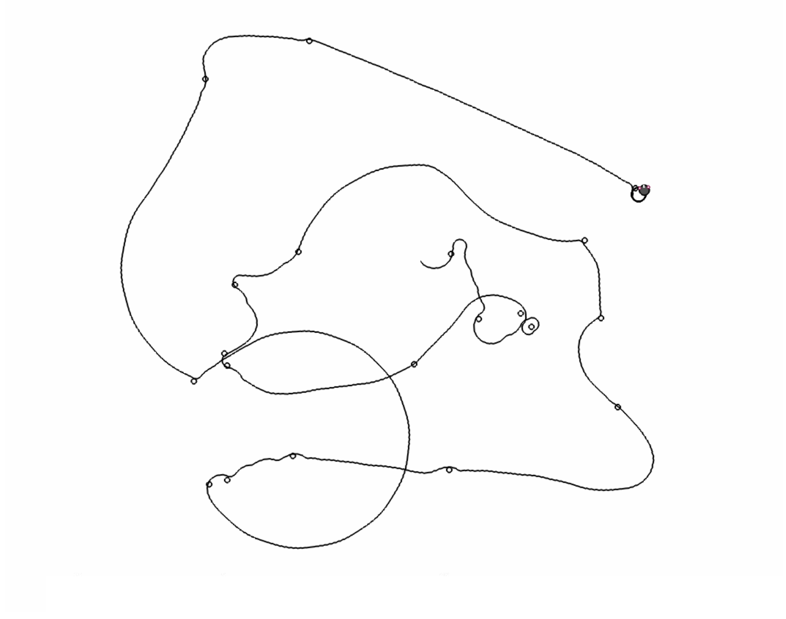


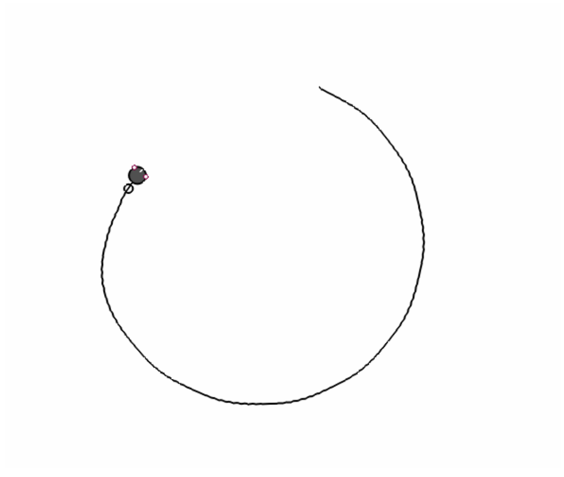
FIGURE 6.14: The behaviour of the evolved animat using unary coding and constant thrust using simulation time 12 s. The gray circle represents the animat, black circles represent the uncollected targets, white circles represent the collected targets and the continuous line represents the path of the animat during the simulation.

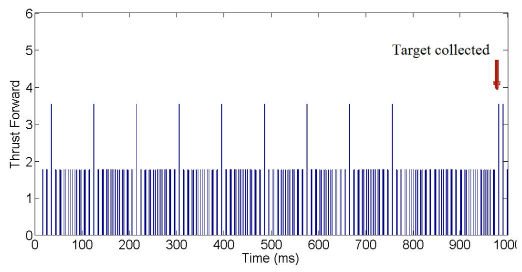
When the 10 evolved animats were re-evaluated on 100 random maps using 20 targets for 12 s, the average (over the 10 animats) of the average (over the 100 evaluation) of $f_{err}$ was 0.18 with standard deviation 0.096. The difference in the $f_{err}$ tended to be about 0.1, when the animats were re-evaluated using 100 random maps again.

The best animat was tested with larger number of targets to check its ability to cope with high food density. The animat managed to collect all targets when 40 targets was presented during the simulation. When 80 targets were presented, the animat was able to collect 34 targets during the first 6 s, then the animat started moving in a circle, which prevented it from collecting more food.

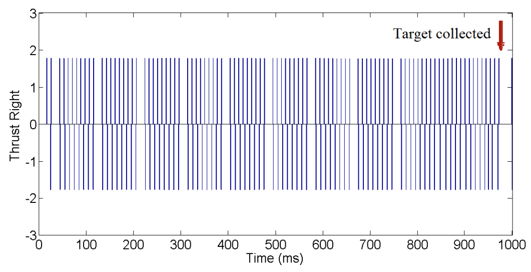
The animat was successfully able to collect all the targets when it was simulated with 44 targets (40 $\times$ 1.1), however, when it was simulated with 48 targets (44 $\times$ 1.1) it managed to collect only 17 and then it moved in circles again without being able to collect any more. So for this animat, the maximum number of targets it managed to cope with was 44. In other words, by this measure, using constant thrust resulted in obtaining an animat more fragile comparing to the best animats discussed in section 6.3.2.1 and section 6.3.1.

Again here I saved the thrusts at the right and left actuators during two simulation. The thrust at any of the two actuator can take only one value of two possible values (0


(A) The trajectory of the best animat evolved using unary coding and constant thrust when it was simulated for 1 s with only 1 target. The gray circle represents the animat, white circle represents the collected target and the continuous line represents the path of the animat during the simulation



(B) The thrust forward generated by actuators during the simulation.

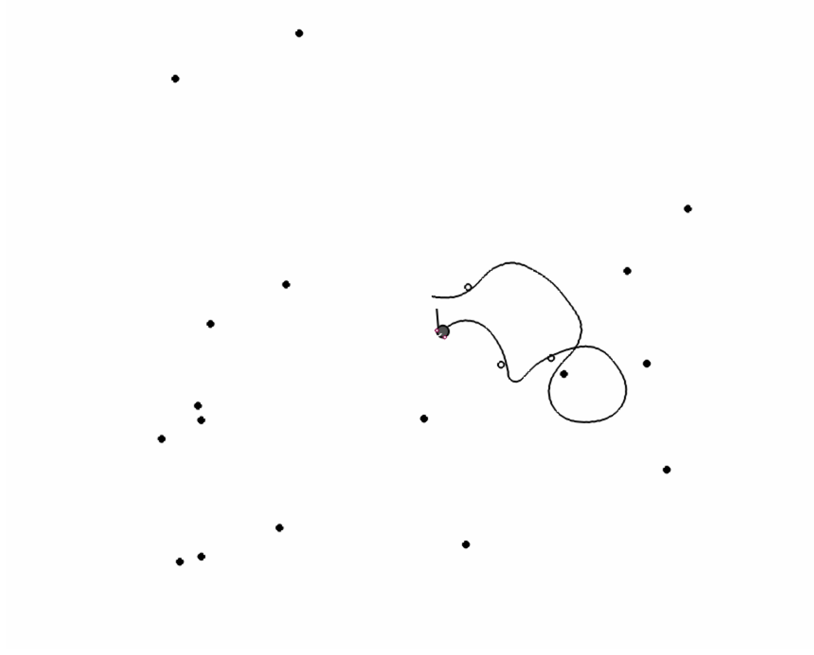


(C) The thrust right generated by actuators during the simulation.

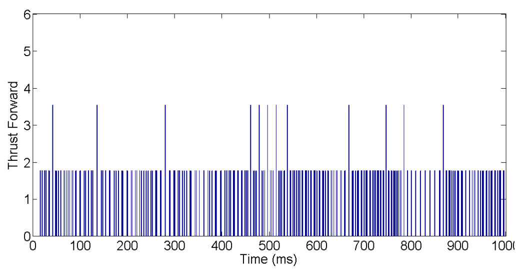
FIGURE 6.15: The right and the forward thrusts generated by the actuators during the simulation of the best animat evolved using unary coding and constant thrust with 1 target on the map.

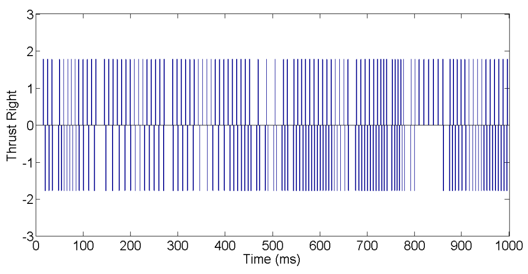
or 2.5). As a result, the thrust forward $\frac{L+R}{\sqrt{2}}$ and the thrust right $\frac{L-R}{\sqrt{2}}$ can take only three values (0, $\frac{\beta}{\sqrt{2}}$, and $\frac{2\beta}{\sqrt{2}}$; $\frac{\beta}{\sqrt{2}}$, 0, and -$\frac{\beta}{\sqrt{2}}$, respectively).

The values of the forward and the right thrusts, when only one target was used (Fig. 6.15), were slightly less than the values when 20 targets were used (Fig. 6.16).



(A) The trajectory of the best animat evolved using unary coding and constant thrust when it was simulated for 1 s with 20 targets. The gray circle represents the animat, black circles represent the uncollected targets, black circles represent the uncollected targets, white circles represent the collected targets and the continuous line represents the path of the animat during the simulation





(B) The thrust forward generated by actuators during the simulation.

(C) The thrust right generated by actuators during the simulation.

FIGURE 6.16: The right and the forward thrusts generated by the actuators during the simulation of the best animat evolved using unary coding and constant thrust with 20 targets on the map.

#### 6.3.2.3 Using the sliding window thrust

The third method I used to update the force of the actuators was using a sliding window thrust presented in equation 6.4 using $\gamma = 15$. At each step, the number of spikes generated by the output neuron during the last 60 ms was counted, divided by the $\gamma$ factor, and used as a thrust for the force at the corresponding actuator.

The number of spikes generated in a period of 60 ms lies in the range [0,30]. Using $\gamma = 15$ allowed the thrust at each step to lie in the range [0,2]. The thrust affects the actuators here every step not like in the case of using the $(V - V_{th})$ and the constant thrusts which only affect the actuators if the corresponding output neurons fire spikes.

The best animat out of 10 champions using this setup had $f_{err} = 0.34$. When this animat was simulated using simulation time 6 s (same time used during the evolution) (Fig. 6.17), it managed to collect only 11 targets out of 20 targets.

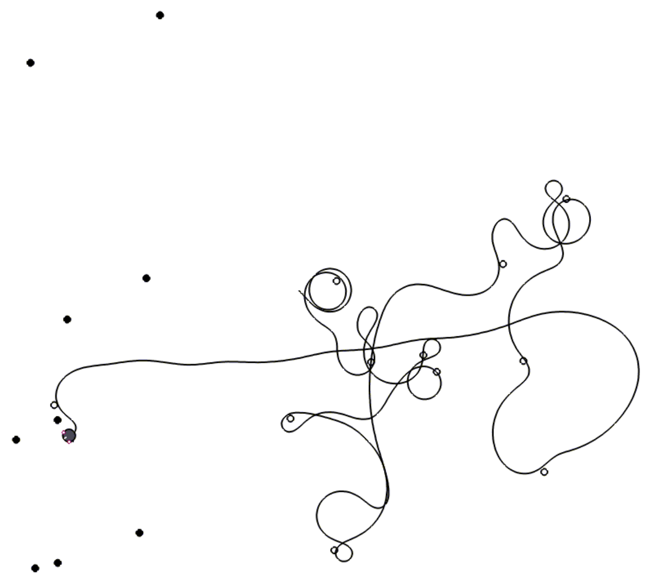


FIGURE 6.17: The behaviour of the best evolved animat using unary coding and sliding window thrust using simulation time 6 s. The gray circle represents the animat, black circles represent the uncollected targets, white circles represent the collected targets and the continuous line represents the path of the animat during the simulation.

However when this animat was simulated for 12 s, it managed to collect all the food (Fig. 6.18).

The average $f_{err}$ of the best animat out of the 10 champions was 0.47 with standard deviation 0.087. Eight animats out of the 10 animats suffered from the circling movement so using the reward here would be helpful as well.

When the 10 champions were re-evaluated on 100 random maps each contained 20 targets for a simulation time 12 s, the average of average $f_{err}$ was 0.27, while the standard deviation of the average was 0.12. The difference in the $f_{err}$ tended to be about 0.02, when the animats were re-evaluated using 100 random maps again.

I tested the ability of the best evolved animat to cope with high food density. As explained before, I started with doubling the number of targets. When 160 targets were present during the simulation, the animat managed to collect all of them. When the

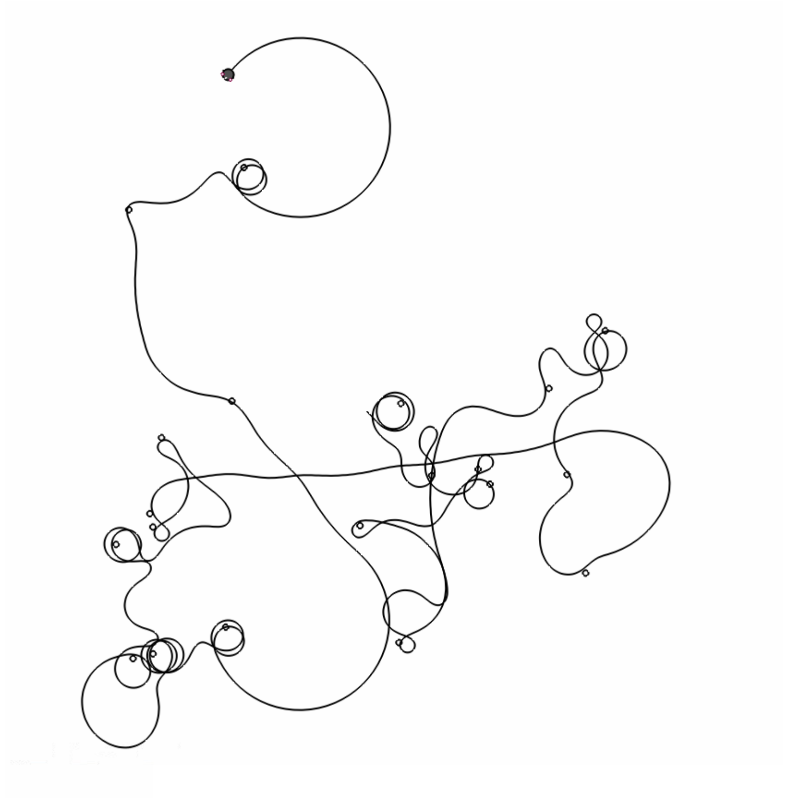
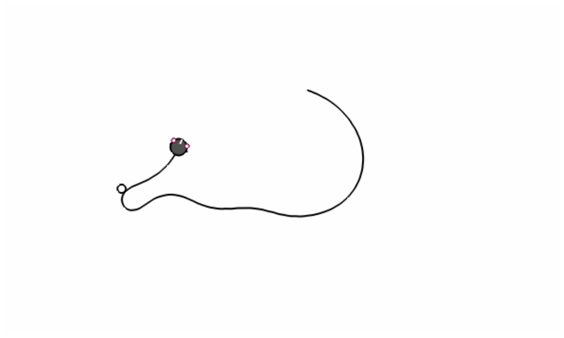


FIGURE 6.18: The behaviour of the best evolved animat using unary coding and sliding window thrust using simulation time 12 s. The gray circle represents the animat, black circles represent the uncollected targets, white circles represent the collected targets and the continuous line represents the path of the animat during the simulation.

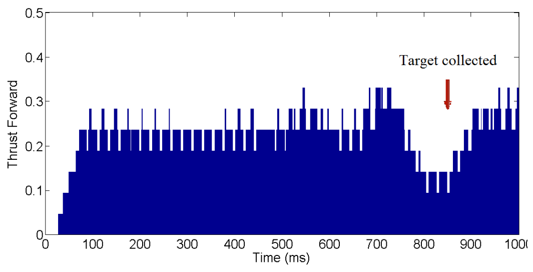
targets were doubled to 320 using the same simulation time, the animat broke down and stopped moving after around 1 s. It was able to collect only 9 targets.

When I reduced the targets to 176 ($160 \times 1.1$), the animat stopped moving after around 2 s, after collecting only 7 targets. The maximum number of targets this animat was able to cope with was 160 targets, therefore, using sliding window results in obtaining an animat more fragile than the animat evolved with conductance injection coding (described in 6.3.1) and the animat evolved with unary coding and $(V - V_{th})$ thrust (described in 6.3.2.1) and less fragile than the animat evolved with unary coding and constant thrust (described in 6.3.2.2).

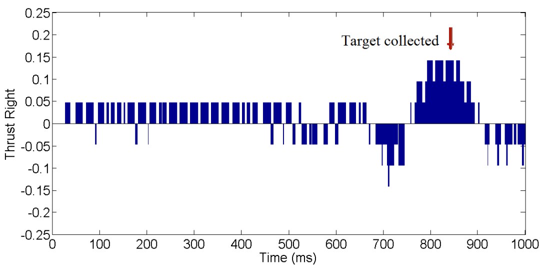
The values of the thrust at the right and left actuators have more variance using the sliding window rather than using the constant thrust. This variance allowed the animat to have a smooth motion. The range of the thrust forward and the thrust right depends on the number of targets presented during the evaluation. For example, when the best animat was tested for 1 s with 1 target (Fig. 6.19), the range of the thrusts was less than the range of the thrusts when 20 targets were used (Fig. 6.20).



(A) The trajectory of the best animat evolved using unary coding and sliding window thrust when it was simulated for 1 s with only 1 target. The gray circle represents the animat, white circle represents the collected target and the continuous line represents the path of the animat during the simulation



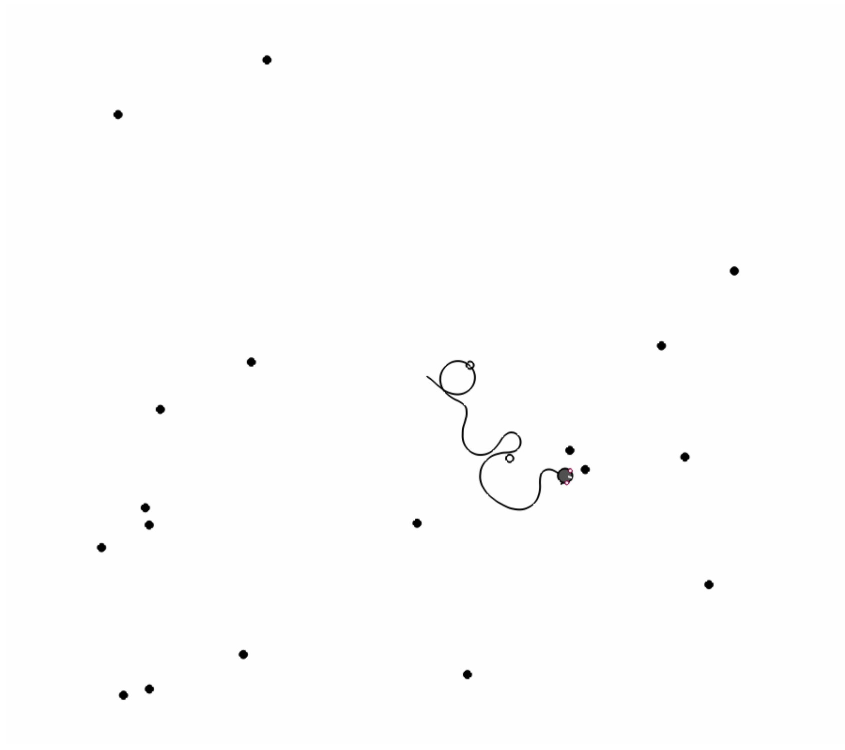
(B) The thrust forward generated by actuators during the simulation.

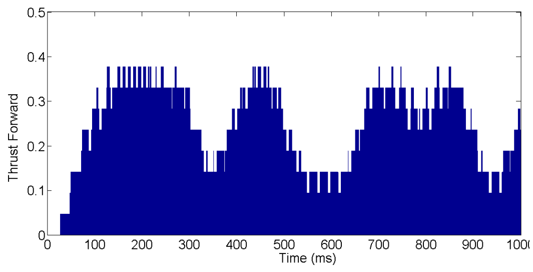


(C) The thrust right generated by actuators during the simulation.

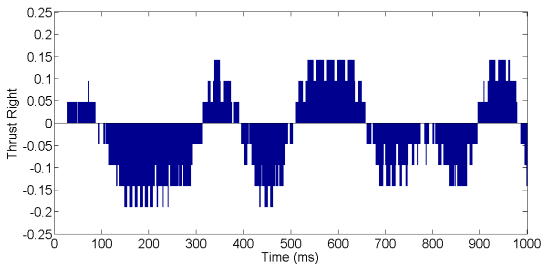
FIGURE 6.19: The right and the forward thrusts generated by the actuators during the simulation of the best animat evolved using unary coding and sliding window thrust with 1 target on the map.

Note that although thrusters work more often for sliding window thrust than in the setup explored previously in sections 6.3.2.1 and 6.3.2.2 (where they could work only when output neurons spiked, and they could not spike more often than every 2 ms), they give smaller thrust each time (Fig. 6.19b vs. 6.15b and 6.11b), and the latter effect overrides the former resulting in slower speed, evidenced by shorter trajectories covered in one second (6.19a vs. 6.15a and 6.11a).


(A) The trajectory of the best animat evolved using unary coding and sliding window thrust when it was simulated for 1 s with 20 targets. The gray circle represents the animat, black circles represent the uncollected targets, black circles represent the uncollected targets, white circles represent the collected targets and the continuous line represents the path of the animat during the simulation



(B) The thrust forward generated by actuators during the simulation.



(C) The thrust right generated by actuators during the simulation.

FIGURE 6.20: The right and the forward thrusts generated by the actuators during the simulation of the best animat evolved using unary coding and sliding window thrust with 20 targets on the map.

### 6.3.3 Encoding the sensory information at sensors as current injection

The third and the most biologically realistic encoding I implemented is encoding the strength of the food smell at the two sensors as current injection to the input neurons. Because the network should be able to calculate the difference between the strength of the food smell at the right and the left sensors, the direct strengths of smell are used ($S_R$ and $S_L$). In the work done before in evolving GRNs for animat foraging in GReaNs, the attempts of using the direct strengths of smell ($S_R$ and $S_L$) were not successful (Borys Wróbel, personal communication)

I used the sliding window method described by equation 6.4 to calculate the thrust during the simulation of the animat. As shown in section 6.3.2.3, when the sliding window was used to calculate the thrusts at the actuator, eight out of 10 animats suffered from circling movement. For this reason, the change direction reward was used for the evaluation of the animats during the evolution.

The most important task in this coding is to find a suitable model to map the strength of food smell to input current. I aimed for a linear relationship between the strength of the food smell at the sensors and the firing rate of the input neurons so that the animat can detect the small differences of concentrations between the right and the left sensors [22]. The following Hill function was used to map the strength of food smell at the two sensors to current injection to the two input neurons:

$$I = K_1 \frac{C^m}{{K_2}^m + C^m} \tag{6.10}$$

where $K_1$ is the scaling factor, $K_2$ is the average strength of smell at the sensors , and $m$ is the Hill coefficient. $K_1 = 30$, $K_2 = 1.5$ (average strength of smell), and $m = 2.5$ were used.

I introduced some modifications in GReaNs so that the current injection encoding would work properly:

- In all the previous tasks using GReaNS, the simulation step duration was 1 ms. Using this step duration did not result in suitable variance of firing rate (Fig. 6.21 - 6.23) using the Hill function (Equation 6.10). The reason behind this is that using a refractory of 1 ms and a simulation step duration of 1 ms, the input neurons can only generate a spike every 2 ms (500 Hz is the maximum frequency), 1 spike every 3 ms (333 Hz), 1 spike every 4 ms (250 Hz) .... etc.

  In order to solve this problem I decreased the step duration to 25 $\mu s$ using the same refractory period (1 ms or 40 steps). If it takes 20 steps for the input neurons with constant food concentration to reach the threshold, then the input neurons will fire spikes after steps: 20, 20+40+20=80, 80+40+20=140, ... etc which is spiking every 60 steps. This will result a spike every 1.5 ms (at about 666 Hz).

Input neurons will fire with maximum firing rate when it takes only 1 step to reach threshold. This will lead to a spike every 41 steps (at about 975 Hz).

Decreasing the step period allowed a wider variance of input neurons firing rate (Fig. 6.24 - 6.26) in response to the current injection.

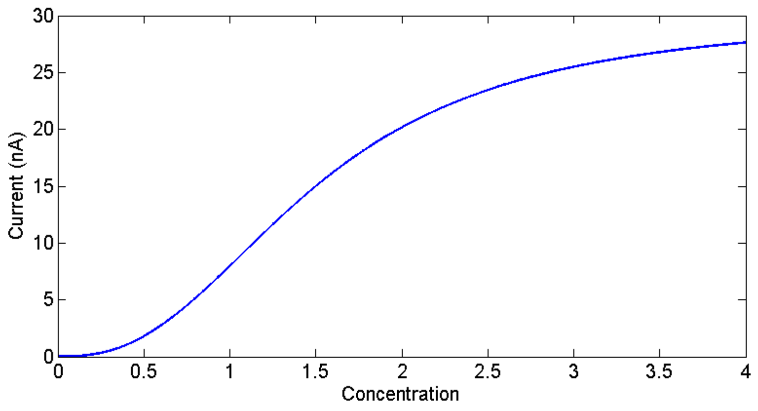


FIGURE 6.21: The relation between the strength of smell at the sensors and the injected current at the input neurons using the Hill function (Equation 6.10) using a step period 1 ms.

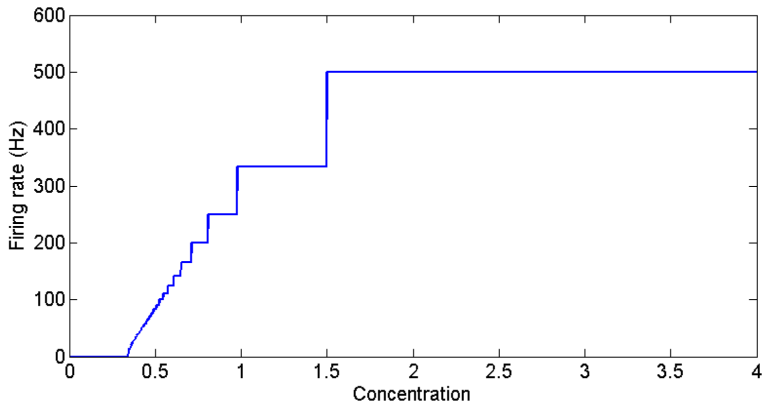


FIGURE 6.22: The relation between the strength of smell at the sensors and the firing rate of the input neurons using the Hill function (Equation 6.10) using a step period 1 ms.

- I increased the radius of the animat (Fig. 6.27) and made the sensors further apart. These modifications will allow a longer physical distance between the right and the left sensors, therefore a bigger difference between the strength of smells at the right sensor and the left sensor (Fig. 6.28).

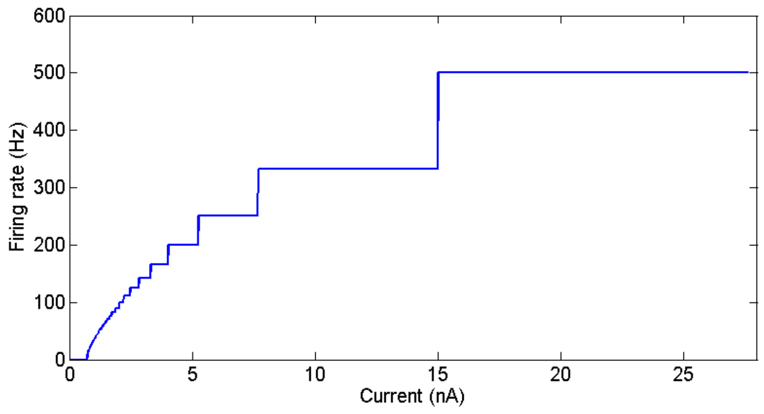


FIGURE 6.23: The relation between the injected current at the inputs and the firing rate of the input neurons using the Hill function (Equation 6.10) using a step period 1 ms.

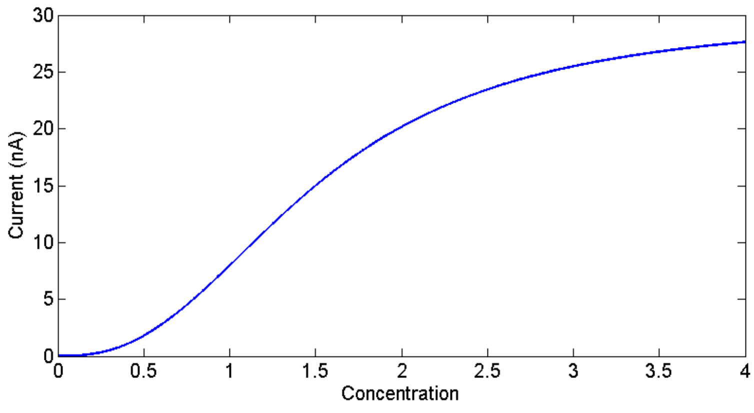


FIGURE 6.24: The relation between the strength of smell at the sensors and the injected current at the input neurons using the Hill function (Equation 6.10) after decreasing the step period to 25 $\mu m$.

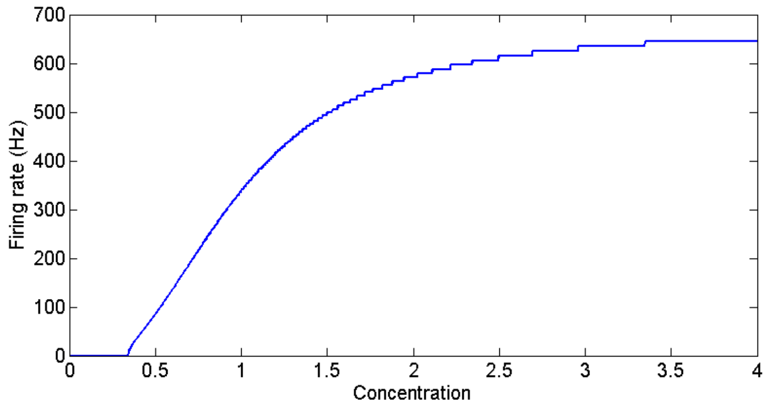


FIGURE 6.25: The relation between the strength of smell at the sensors and the firing rate of the input neurons using the Hill function (Equation 6.10) after decreasing the step period to 25 $\mu m$.

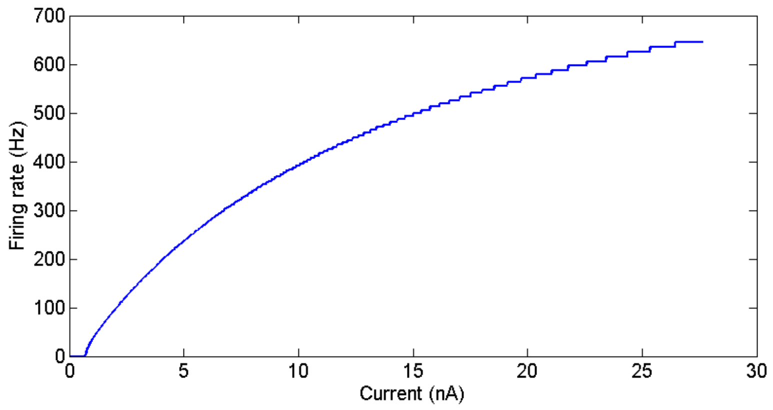


FIGURE 6.26: The relation between the injected current at the inputs and the firing rate of the input neurons using the Hill function (Equation 6.10) after decreasing the step period to 25 $\mu m$.

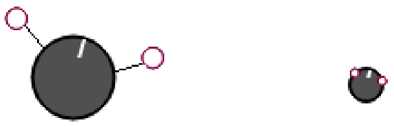


FIGURE 6.27: The difference between the animat before modifications (on the right) and after modifications (on the left).

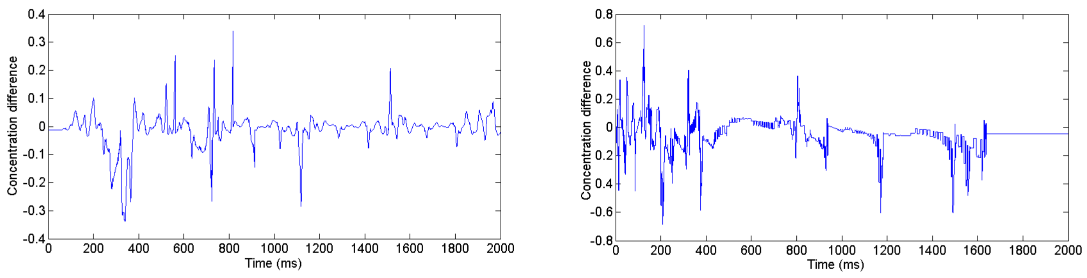


FIGURE 6.28: The difference between the strength of smell at the right sensor and the left sensor before increasing the radius of the animat and the length of the sensors (left) and after increasing the radius of the animat and the length of the sensors (right). This data was recorded when the animat moved randomly on a map with 20 targets. Note that the vertical scale differs by a factor 2.

I applied these modifications and I reduced the number of food resources to only 6 particles to lower the density of targets on the map. This also allowed for shorter simulation times at the beginning of the evolutionary run (when 6 targets were used, collecting 1 on average on random maps had a larger effect on fitness than collecting 1 out of 20).

The evolution mechanism I used here was different than the one used in the previous sections in this chapter. First, I used 1.5 s as a simulation time with 6 targets and 4 random maps for 1000 generations. I ran the evolution for 100 generations more using eight random maps for evaluation. Each animat was evaluated on four maps using again 1.5 s as a simulation time with 6 targets, and on the other four maps using 3 s as a simulation time with 20 targets.

Reducing the number of targets at the beginning of the evolution helped in avoiding random animat movement. This random movement was likely to happen after increasing the size of the animat. On the other hand, with only 6 targets the animat can by unlucky end up far away from the targets on a particular map.

After running 10 independent evolutionary runs for 1000 generations on 4 random maps and with 6 targets, the best animat I obtained had $f_{err}$ = 0.27. This animat managed to collect 4 targets when it was simulated for 1.5 s (Fig. 6.29).

After the last 100 generations in which the number of random maps was increased to 8 and the animat was evolved with 6 targets on 4 of them and with 20 targets on the other 4, the best animat had $f_{err}$ = 0.12. Fig. 6.30 shows the evolutionary history of this animat. This animat was not able to show good behaviour when it was simulated with 20 targets for 3 s (Fig. 6.31). It collected 7 targets during the the simulation but it ignored some targets in its way.

The average $f_{err}$ of the 10 champions was 0.32 with standard deviation 0.1.

When these animats were re-evaluated on 100 random maps with 20 targets using simulation time 24 s, the average of the average of $f_{err}$ was 0.52 with standard deviation 0.18. The change direction reward was disabled during the re-evaluation of the animats, so that the values obtained allow for easy comparison with the $f_{err}$ values reported for the animats I have described previously in this chapter.

The best animat after re-evaluation had average of $f_{err}$ = 0.28, this animat had $f_{err}$ = 0.17 after the evolution. When this animat was simulated for 3 s, it managed to collect only 4 targets out of 20. This animat managed to detect the positions of the targets and was able to move towards them. The reason this animat was not able to collect more targets was that this animat moved slowly, however the best animat after

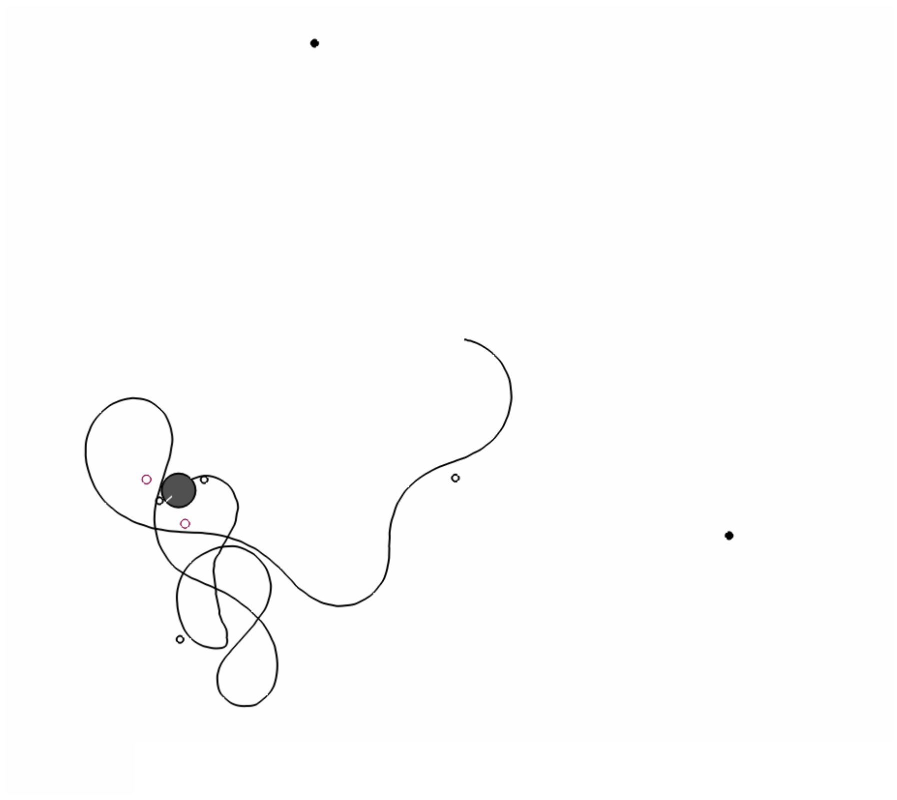


Figure 6.29: The behaviour of the best evolved animat using current injection coding after 1000 generations on a map for 6 targets. The gray circle represents the animat, black circles represent the uncollected targets, white circles represent the collected targets and the continuous line represents the path of the animat during the simulation.

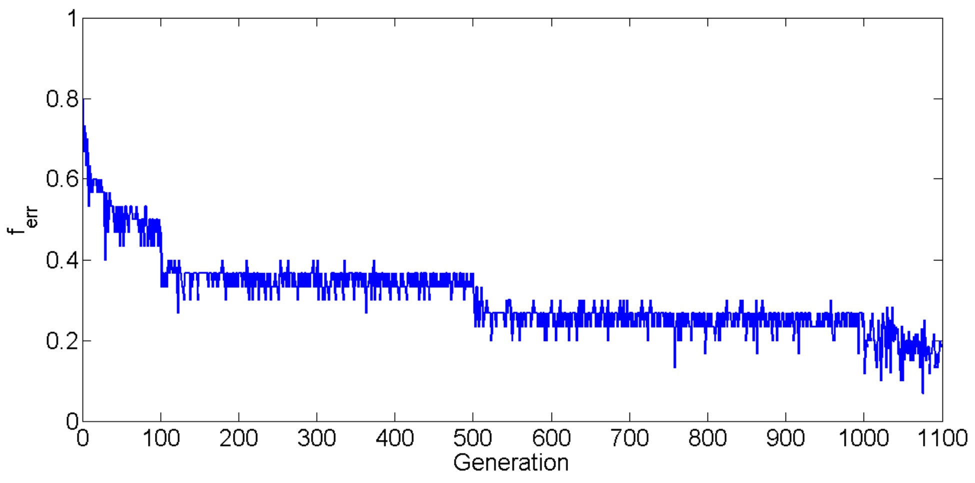


Figure 6.30: The evolution history of the best animat evolved using current injection coding.

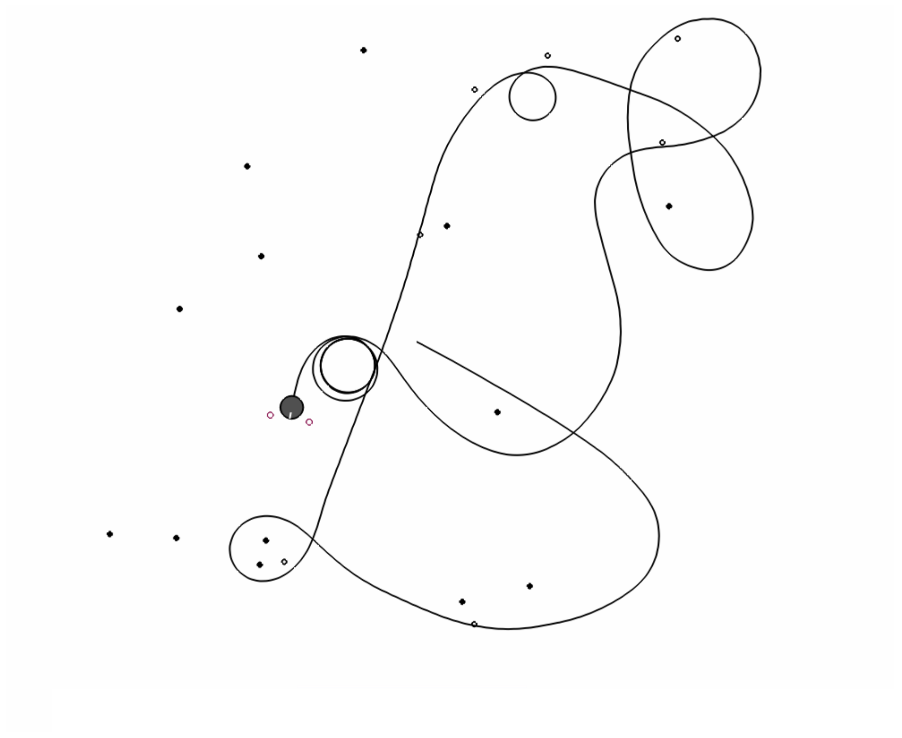


Figure 6.31: The behaviour of the best evolved animat using current injection coding after 1100 generations. The gray circle represents the animat, black circles represent the uncollected targets, white circles represent the collected targets and the continuous line represents the path of the animat during the simulation.

evolution showed in Fig. 6.31 was moving faster and got benefits from its large size to collect targets.

In order to make sure that this animat would be able to continue collecting targets, I increased the simulation time during evaluation to 30 s. Fig. 6.33 showed that the animat was able to collect 17 targets.

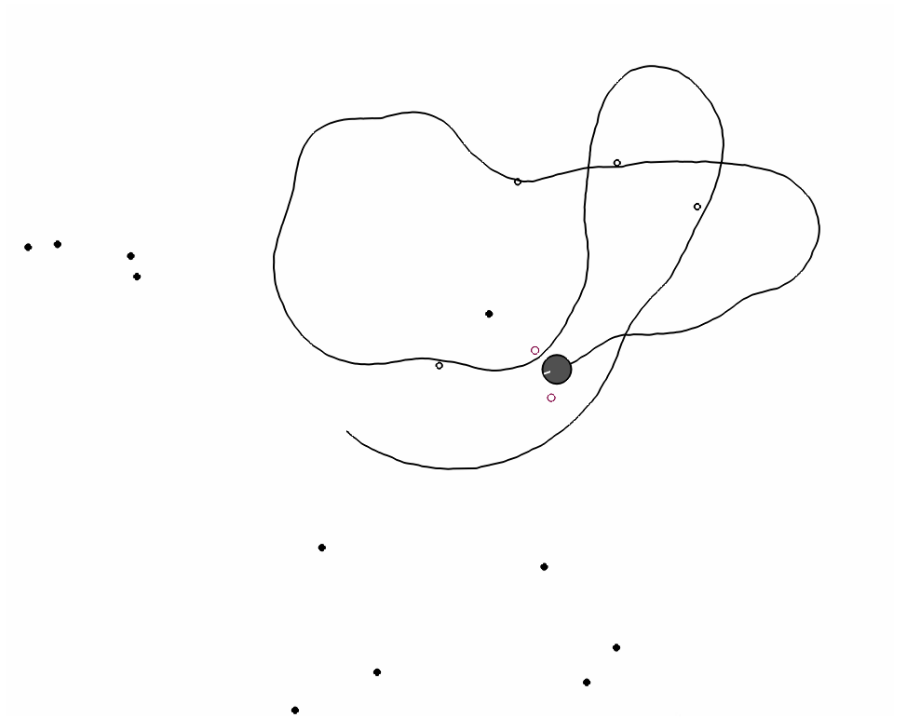


FIGURE 6.32: The behaviour of the best animat after re-evaluation using current injection coding simulated for 3 s. The gray circle represents the animat, black circles represent the uncollected targets, white circles represent the collected targets and the continuous line represents the path of the animat during the simulation.

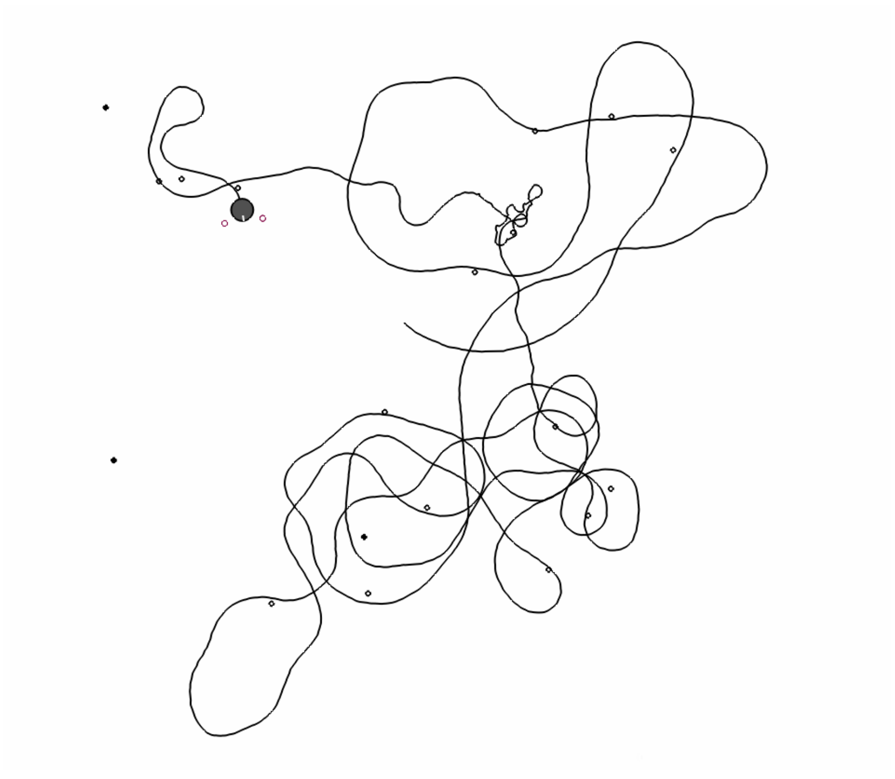


FIGURE 6.33: The behaviour of the best animat after re-evaluation using current injection coding simulated for 30 s. The gray circle represents the animat, black circles represent the uncollected targets, white circles represent the collected targets and the continuous line represents the path of the animat during the simulation.

## 6.4 Conclusion

I have successfully used GReaNs to evolve SNNs able to control animats so that they can detect targets and collect them. I have presented three different coding strategies for the sensory neurons and three different ways of calculating the thrusts at the actuators.

The first setup I had used was using synaptic conductance injection to encode the sensory information and the $(V - V_{th})$ thrust to calculate the thrust at the actuators. The 10 champions evolved using this setup had the least average value for $f_{err}$ and none of them suffered from circling movement problem. When these champions were re-evaluated on 100 maps using longer simulation time, these champions managed to collect all the targets in most of the cases. The best animat among these champions was the least fragile animat when it was simulated on a map with very high food density when compared with animats used other encodings for sensory information explored in this chapter.

When I replaced the synaptic conductance injection coding with the unary coding using the same method of calculating the thrust, the evolved animats had almost the same properties. These animats did not suffer form the circling movement as well. They had the same average value for $f_{err}$ (0.14) but the champions evolved using unary coding had higher standard deviation (0.12 in case of conductance injection coding and 0.18 in case of unary coding). They almost had the same average for $f_{err}$ as the champions using synaptic conductance injection when both of them were re-evaluated on 100 maps (0.07 in case of conductance injection coding and 0.06 in case of unary coding). However, the networks evolved using unary coding had a higher standard deviation again (0.06 in case of conductance injection coding and 0.09 in case of unary coding). These results suggest slightly lower evolvability of the setup with unary coding in comparison to conductance injection. The best network of these champions showed high robustness when it was tested on a map with high food density, however it was less robust than the best animat evolved using the conductance injection encoding.

When the unary coding was used again but using a constant thrust instead of $(V - V_{th})$, there was a drop in the efficiency in the evolved animats. The 10 evolved champions had a higher average for $f_{err}$ after evolution and after re-evaluation comparing to the previous champions. The main reason behind this drop was that using the constant thrust method decreased the variance of the values of the thrusts at the actuators. The decrease in the variance made it harder for the animats to change direction. As a result, 50% of the evolved animats suffered from the circling movement problem. Beside that, the best network among these 10 champions was the most fragile network comparing to animats used other encodings for sensory information explored in this chapter.

When the sliding window was used to calculate the thrust beside using unary coding for sensory information, there was another drop in the efficiency. The animats evolved using this setup had smoother movement comparing to other animats. On the other hand, these animats had problems in changing direction and 80% of them suffered from the circling movement. It was necessary to add the reward for changing direction in this task. The best animat evolved using sliding window thrust and unary coding was less fragile than the best animat evolved using constant thrust and unary coding when both of them were simulated on a map with high food density. However, it was more fragile than the best animat evolved using $(V - V_{th})$ thrust.

The hardest in terms of evolvability but the most biologically realistic task that was done in this chapter was using the current injection coding for sensory information and the sliding window to calculate the thrust at the actuators. The most difficult part of this task was using the direct concentration of the food instead of their summation and difference. First I decreased the simulation step to 25 $\mu s$ and used a Hill function in order to have a linear-like relationship between the concentration of the food and the firing rate of the input neurons. In order to make the difference between the concentration at the right sensor and the left sensor, I increased the size of the animat and made the sensors further apart.

Increasing the size of the animat helped in increasing the difference between the concetration at the right and left sensor, however, it increased the chance of having an animat which could move randomly with high speed using its size to collect food. This is exactly what happened with the best evolved animat. After re-evaluating the evolved champions, the best animat after re-evaluation showed a very good behaviour in detecting the position of targets and was able to collect them.

At the beginning of the chapter, I started from the task of evolving animat using synaptic conductance injection, the artificial variables which represent the summation and the difference of the concentration, and $(V - V_{th})$ thrust. This task was very easy as it was already proved to be working. On the other hand, this task was not biologically realistic. Starting from the easiest task which was the least biologically realistic helped me to develop the reality of the model across the chapter until I achieved the most biologically realistic and the hardest setup at the end of the chapter.

The work I have done in this chapter can be considered as the first step towards evolving spiking neural networks that control animats so that they can detect and move towards specific temporal sequences of simulated sounds.

# Chapter 7

# Temporal Pattern Recognition in Animats

## 7.1 Introduction

After evolving SNNs able to detect temporal patterns of inputs and evolving SNNs able to control animats for directional movement in a concentration gradient, it was interesting then to check the complexity of evolving SNNs which able to do both tasks together.

The idea of this task is to evolve a SNN able to control an agent able to distinguish one sound source type out of several sound source types and collect all sound sources from this type.

Each sound source is represented as a pattern of three different frequencies. The animats are simulated on 2D map with six different sound source types each with unique order of the three frequencies. In this task, animats have three sensors, each of this sensor detects one frequency of the sound sources. Based on the order of the frequencies read by each animat, the animat should be attracted by only one sound source type and ignore the others.

The goal of the evolution is to obtain an animat able to collect the sound sources with a pattern of three different frequencies (1-2-3) and to ignore (not to collect) the sound sources with the other permutations of the three frequencies.

### 7.1.1 The description of the animat and its simulation environment

The same animat explored in section 6.3.1 was used in this task (conductance injection for sensory information, with $(V - V_{th})$ when the output neurons spike to determine thrust). This animat used the synaptic conductance injection coding to encode the sensory information representing the distance between the animat and the sound sources. As in the previous task, the animat had two sensors: one sensor was located on the right front of the animat and the other on the left front of it. Additional three inputs were added to the network to support the network with the temporal pattern information of each sound source (Fig. 7.1).

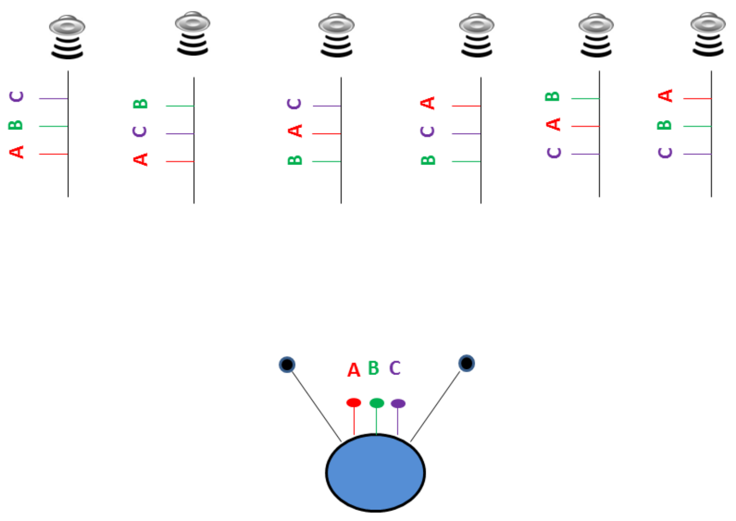


FIGURE 7.1: A simplified description of the task of collecting sound sources. Each animat has five sensors, the outer two sensors for sound source position detection. Each sensor from the other three sensors is responsible for detecting one frequency from the three sound frequencies. Each sound source has a unique pattern of sound tones.

The animat was simulated on 2D map with 6 sound resource types. Each type of them streams different temporal pattern information (pattern of different frequencies). Each animat was simulated six times to be evaluated during every generation. For each simulation, the animat was initially reset and only one type of sound resources were active and the others were silent. The temporal pattern consisted of 3 inputs one of which produced one spike 5 ms into each 100 ms, one of which 10 ms into each 100 ms, and finally one 15 ms into each 100 ms. So during each simulation, only one pattern was presented and was repeated every 100 ms.

### 7.1.2 Genetic algorithm

The same genetic algorithm used in the real time control of foraging behaviours task was used here. The following function was used to calculate the fitness value of each animat:

$$f_{err} = 1 - \frac{collected_{target}}{total_{target}} + 0.2 \sum_{i=1}^{5} \frac{collected_i}{total_i} \tag{7.1}$$

$collected_{target}$ is the number of the collected sound sources from the first type with the pattern 1-2-3, $total_{target}$ is the total number of the sound sources from the first type with the pattern 1-2-3, $collected_i$ is the number of the collected sound sources from the $i^{th}$ type with the other (1-3-2, 2-1-3, etc), $total_i$ is the total number of the sound sources from the $i^{th}$ type with the other patterns (1-3-2, 2-1-3 ..etc).

The goal for the genetic algorithm is to minimize $f_{err}$. $f_{err}$ can be any value in the interval [0,2] where 0 is the best value and 2 is the worst value. The second term in the equation $\frac{collected_{target}}{total_{target}}$ is used to reward the animat that collects the sound sources from the target type.

The third term in the equation $0.2 \sum_{i=1}^{5} \frac{collected_i}{total_i}$ is used in order to penalize the animat which collects the sound sources from the others types. The summation is multiplied by 0.2 in order to treat the contribution to fitness of the 5 trials with wrong source type the same as the contribution of the trial with the correct source type.

The simulation time used for each animat with one particular sound type is 2 s (12 s in total) and during every simulation 10 sound sources from only one type were presented.

As I mentioned before, each animat is simulated six times each with different temporal pattern, then the $f_{err}$ is calculated based on the number of sound sources collected in each simulation. This operation is repeated four times with random maps (24 simulations in total) and the $f_{err}$ is calculated as the average over the four random maps. The simulation time of the animat is 1 s with each temporal pattern on each map (24 s for every animat).

### 7.1.3 Results for this task

After running the evolutionary algorithm for 1000 generations, the evolutionary algorithm was not able to do proper improvement 7.2. The animat tended to collect all the sound sources regardless of their type.

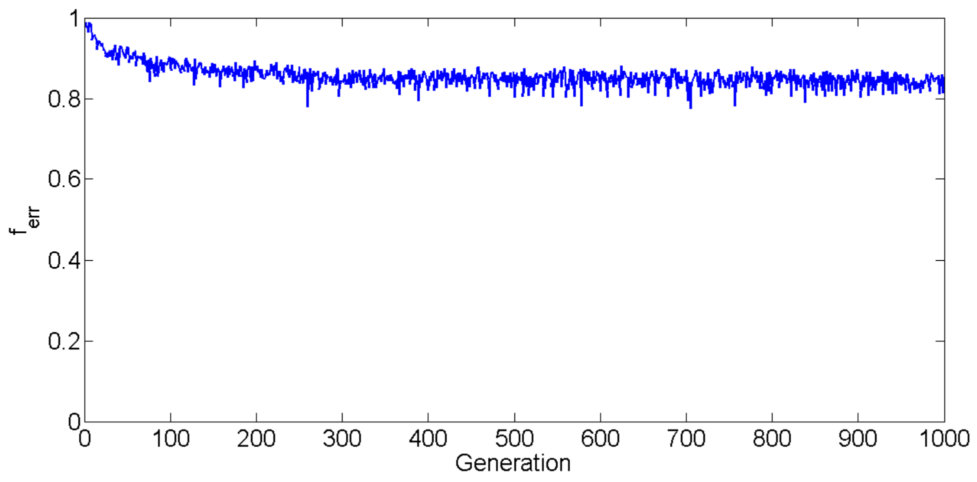


FIGURE 7.2: The evolution history of the evolved animat to collect sound sources with pattern 1-2-3 and ignore sound sources with other patterns.

### 7.1.4 Using two sound source types

In order to analyse and understand the failure of the animat to distinguish between the sound sources, I made the task less complex by reducing the number of sound source types to only two types. The animat should be able to collect only the sound sources from the first type and ignore the sound sources from the other type. I changed the fitness function to be as following:

$$f_{err} = 1 - \frac{collected_{target}}{total_{target}} + 0.5\frac{collected_{neg}}{total_{neg}} \tag{7.2}$$

$collected_{target}$ is the number of the collected sound sources from the first type with the pattern 1-2, $total_{target}$ is the total number of the sound sources from the first type with the pattern 1-2, $collected_{neg}$ is the number of the collected sound sources from the type the animat should neglect with the pattern 2-1, $total_{neg}$ is the total number of the sound sources from the type the animat should neglect with the pattern 2-1.

I multiplied the term $\frac{collected_{neg}}{total_{neg}}$ by 0.5 in order to give the behaviour of the animat with the first sound source type the double weight of the behaviour of the animat with the second sound source type. I used 10 sound sources from each type same as I did in the previous section.

After 1000 generations I got an animat with $f_{err} = 0.78$. Fig. 7.3 shows the evolutionary history of the best animat in the population. The history shows that the evolutionary algorithm managed to make improvement during the last the last 100 generations, however, when I ran it for 9000 more generations (10000 generations in total), the least $f_{err}$ was 0.78 as well.

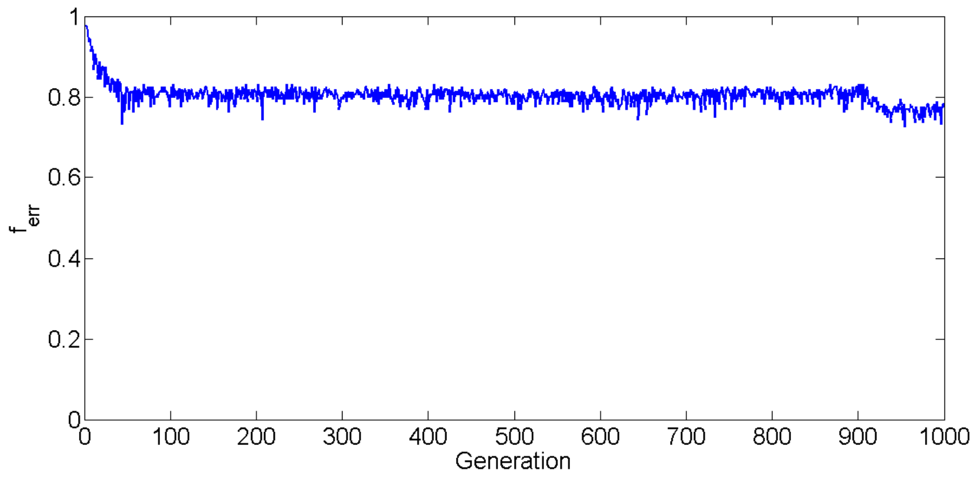


FIGURE 7.3: The evolution history of the best animat in the population to collect sound sources with pattern 1-2 and ignore sound sources with pattern 2-1.

After analysing the behaviour of the animat with the sound sources from the first type (Fig. 7.4) and the second type (Fig. 7.5), we can see that the animat collected the sound sources from both types. The evolutionary algorithm was not able to establish the pattern recognition part of the fitness function.

In the pattern recognition tasks presented in chapter 5, the reward for the pattern recognition was clear in the fitness function that allowed the SNNs to evolve. At the beginning of the evolution, the SNNs were either active or silent with all the patterns. However, the number of spikes generated by the output neuron with each pattern was part of the fitness function. This allowed the evolutionary algorithm to increase the number of spikes for the first pattern and decrease the number of spikes for the other patterns until I got the final SNNs presented in chapter 5.

One suggestion that could help the evolutionary algorithm to establish the pattern recognition part is to add to the fitness function a term that reward the SNNs which have an interneuron which performs temporal pattern recognition. The interneuron will act as the output neuron of the SNNs presented in chapter 5. Having an interneuron that could distinguish between pattern will allow the animat to show different behaviour based on the pattern of the sound source type.

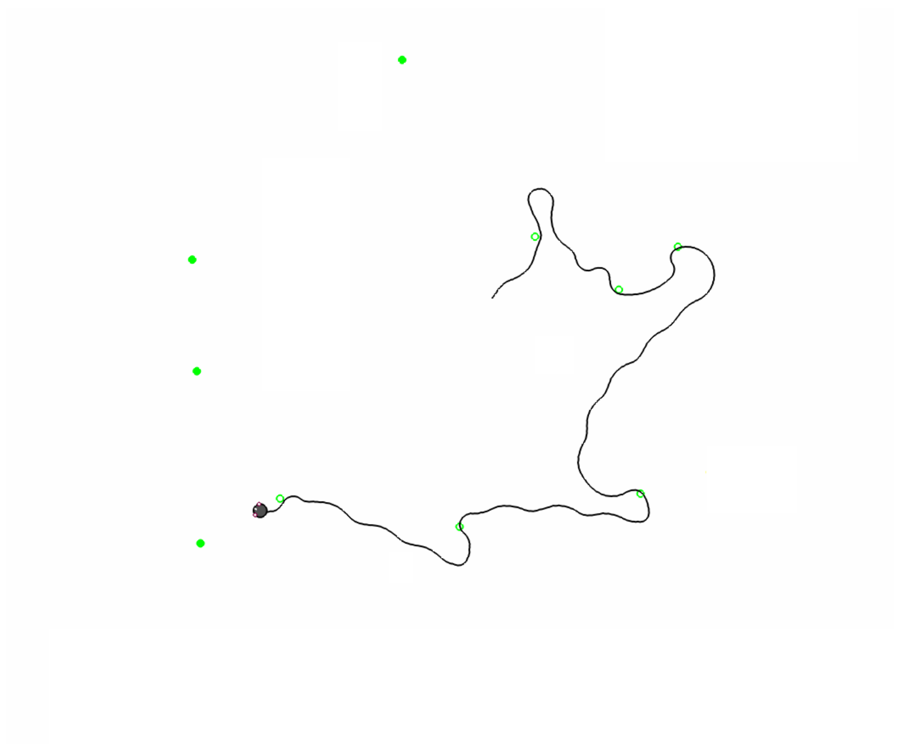


FIGURE 7.4: The behaviour of the evolved animat when the sound sources with the first pattern were presented. The gray circle represents the animat, green circles represent the uncollected sound sources with the first pattern, white circles represent the collected sound sources with the first pattern and the continuous line represents the path of the animat during the simulation.

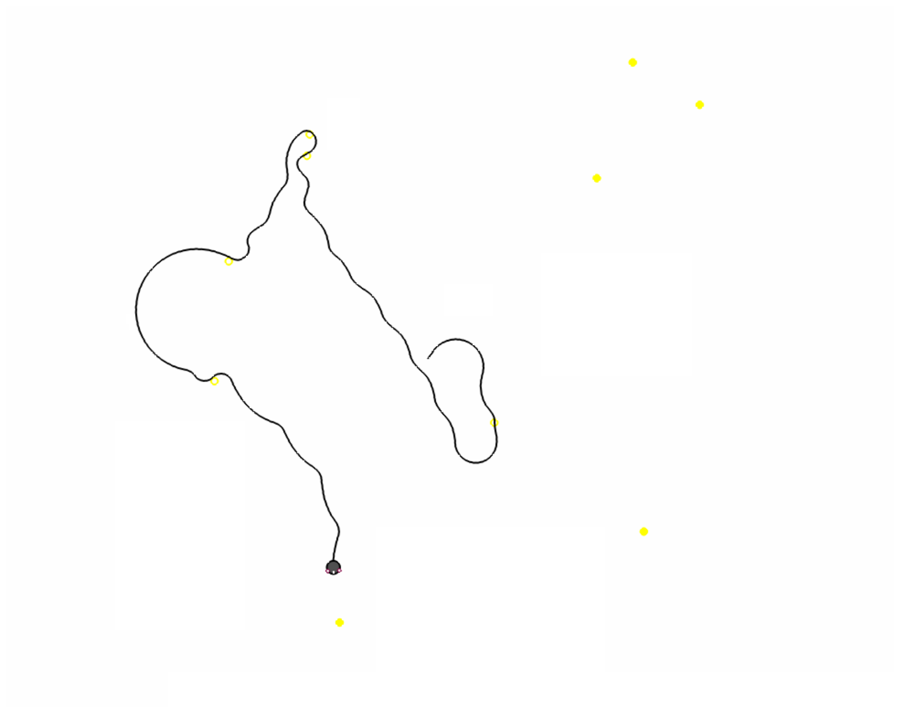


FIGURE 7.5: The behaviour of the evolved animat when the sound sources with the second pattern were presented. The gray circle represents the animat, yellow circles represent the uncollected sound sources with the second pattern, white circles represent the collected sound sources with the second pattern and the continuous line represents the path of the animat during the simulation.

## 7.2 Conclusion

After merging the temporal pattern recognition task and the animat foraging task, the new task was complex that did not allow the evolutionary algorithm to be able to improve the behaviour of the evolved animats. Using only the number of the sound sources collected from each type as the main and the only parameter in the fitness function ,without adding a more direct term for the pattern recognition behaviour, made it impossible for the SNNs to evolve. However, I believe that this complex task can still be achieved especially if we change the fitness function to add a direct reward for ability of SNNs to distinguish between input temporal patterns.

Chapter 8

# Integrating GReaNs with SpiNNaker

## 8.1 Introduction

SpiNNaker [28] (**Spi**king **N**eural **N**etwork **A**rchitecture) is a massively parallel computing system which is designed to support large scale spiking neural networks simulations. SpiNNaker was built at University of Manchester in order to afford real-time simulation for a strongly connected network of thousands of neurons. Each SpiNNaker machine consists of a number of SpiNNaker multicore chips.

I integrated the evolutionary software, GReaNs, with SpiNNaker in order to use the computational power of this multiprocessor system for real-time simulation of the SNNs during the evaluation step in the genetic algorithm. I used two types of the SpiNNaker board: the board with four chips, and the board with 16 chips. Each chip on SpiNNaker board is a Globally Asynchronous Locally Synchronous (GALS) system. Each chip includes 18 ARM968 processor nodes and each processor can simulate up to 1000 neurons.

SpiNNaker supports both LIF and AdEx models which are the models I implemented in GReaNs. One of the advantages of SpiNNaker is that it can be used as a simulation backend for PyNN [34] scripts, which made the integration easier. Most of the work done in the integration was carried out during the CapoCaccia Cognitive Neuromorphic Engineering Workshop in collaboration with Dr. Sergio Davis from University of Manchester.

## 8.2 The integration model

The first necessary step in the integration was to find the most efficient protocol for GReaNs and SpiNNaker to allow communication between them. The genetic algorithm experiments described in this chapter follow the setup formulated in chapter 4 and 5, with the same settings. To remind the reader, the first step of the genetic algorithm was to initialize the whole population of 300 SNNs, then the simulation of the SNNs started. Based on the result of the simulation, the fitness values were calculated. Finally, the new generation was formed after applying tournament selection, crossover and mutation. The new generation was simulated again and so on.

In the simulations described in this chapter, the simulation step was carried on using SpiNNaker, and the rest of the genetic algorithm was completed on GReaNs. For this reason, the spikes are collected, and received by the GReaNs software running on the CPU, which will thus be responsible for calculating the fitness value for each individual, applying mutations, applying crossover, and build the new generation.

Figure 8.1 shows the communication protocol between GReaNs and SpiNNaker. First, a random population of individuals (here, 300) is initialized, each individual with number of regulatory units (here, 5). GReaNs exports the all population as a PyNN script which is sent to SpiNNaker. A software is used to configure SpiNNaker with the PyNN script and the simulation starts. Another software is used to output the spike times of all individuals to a file. This file is saved on the hard disk. GReaNs reads the output file and the fitness value for each individual is calculated based on the current task of the evolution. The new generation is formed by selecting the best individuals in the selection step to be used for the crossover and the mutation part. GReaNs exports the new generation as a PyNN script again to be sent to SpiNNaker and so on.

In all the experiments I ran on SpiNNaker, each generation had 300 individuals (SNNs) each with only 1 output. It was important to know the place where the output neuron of each individual resides in order to record the spikes generated by each neuron. I divided the 300 output neurons into 5 groups each with 60 neurons; each group was sent to a separate processor of the first chip on the board. The main neurons of each individual were grouped together and all the input neurons of each individual were also grouped together. The distribution of the 300 groups of the main neurons and the 300 groups of the input neurons over the processors were handled by the SpiNNaker built-in software which is responsible for configuring the board (Fig. 8.2). An example of the PyNN script generated by GReaNs is shown in Appendix A.

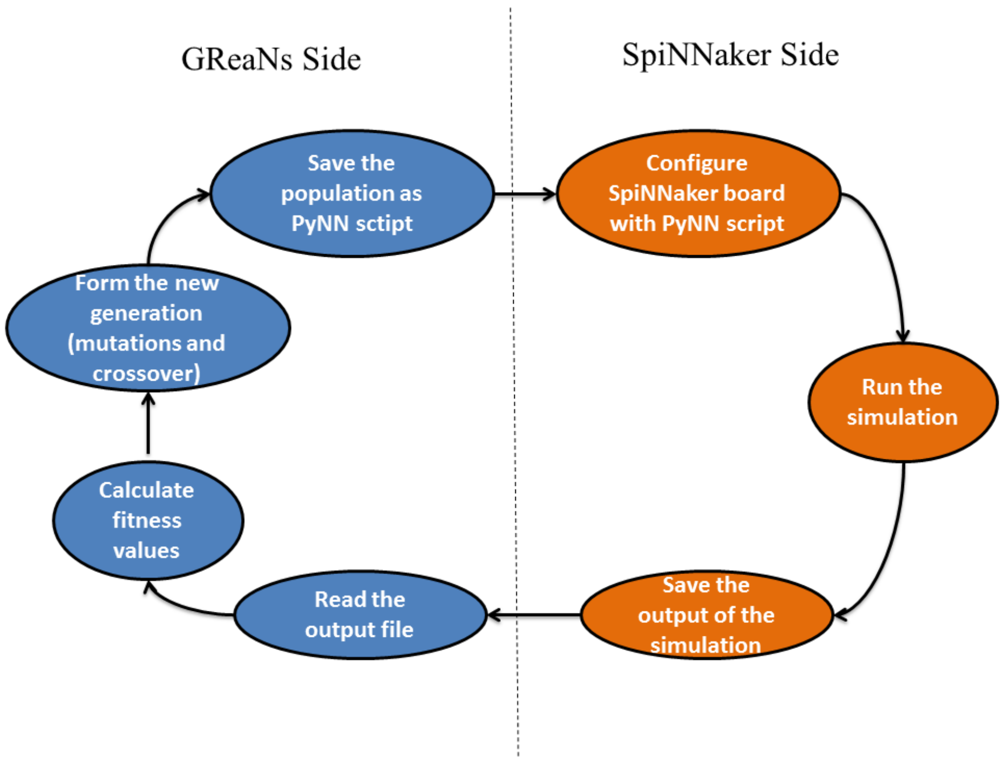

GReaNs Side
SpiNNaker Side
Save the population as PyNN sctipt
Configure SpiNNaker board with PyNN script
Run the simulation
Save the output of the simulation
Read the output file
Calculate fitness values
Form the new generation (mutations and crossover)

FIGURE 8.1: The communication protocol between GReaNs and SpiNNaker. The tasks in the blue oval were executed on the CPU, while the tasks in the orange oval were executed on the SpiNNaker board.

## 8.3 Results

I implemented 2 versions of the integration between GReaNs and SpiNNaker, each version with a different evolutionary task and on a different SpiNNaker machines.

### 8.3.1 Initial communication protocol with the small SpiNNaker board

The first version of the integration was done with the small SpiNNaker board with 4 chips (Fig. 8.3). In this version I run the task of evolving a LIF SNNs to match the spike train of a single AdEx neuron shifted by 5 ms described in chapter 4. As I mentioned before, GReaNs generates a PyNN script for the current generation and sends it to SpiNNaker which runs the simulations and sends the result back to GReaNS in the form: $individualnumber \rightarrow spiketime$. The period of the simulation of this task is 1 second. Using a laptop with Core 2 Due processor each with frequency 2.66 GHz, each generation takes around 2 seconds for this task. Using SpiNNaker as a backend simulator, each generation takes around 1.5 minutes. Most of the computation time was taken in configuring the board at the beginning of the simulation, so no improvement was achieved at all.

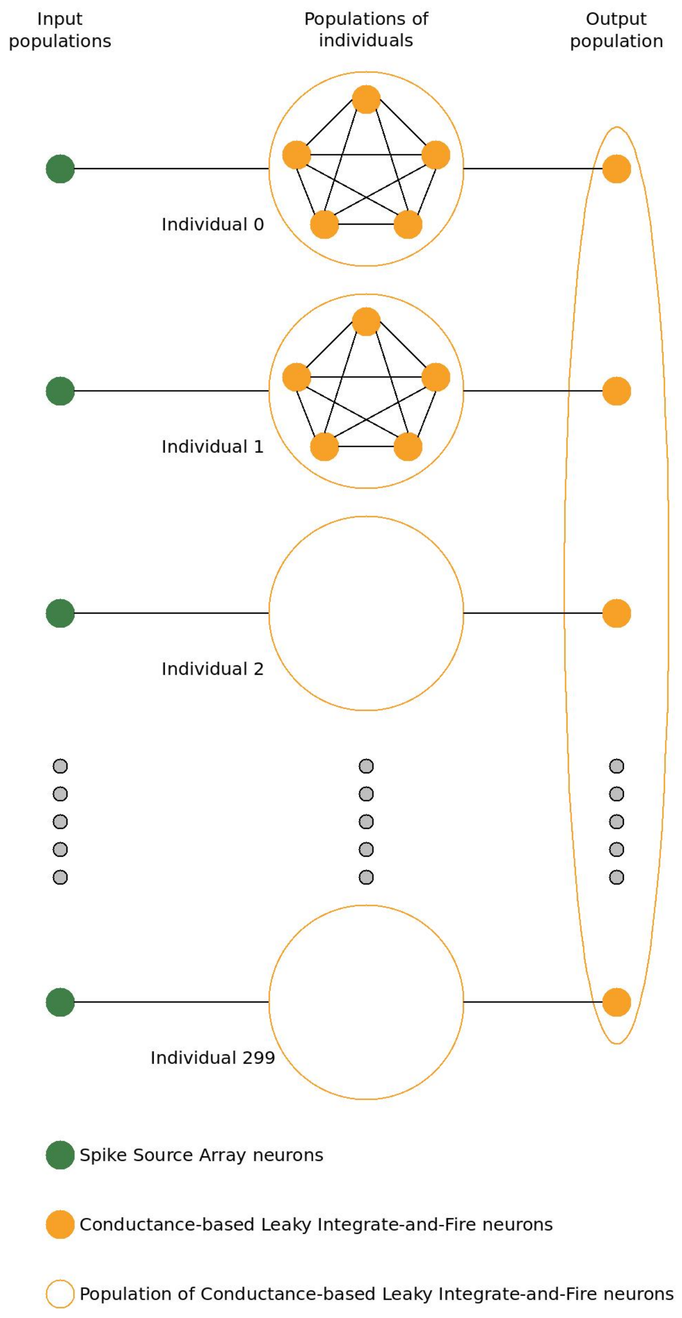

Input populations
Populations of individuals
Output population
Individual 0
Individual 1
Individual 2
Individual 299
Spike Source Array neurons
Conductance-based Leaky Integrate-and-Fire neurons
Population of Conductance-based Leaky Integrate-and-Fire neurons

FIGURE 8.2: The distribution of the neurons of the 300 SNNs over SpiNNaker board. Using both the small and big SpiNNaker boards, the neurons of each network was divided to 3 categories: input neurons, interneurons, and output neurons. The distribution of the input neurons and the interneurons over the board was handled by the built-in software in SpiNNaker. The output neurons were distributed on only the first chip. The order of the processor on the first chip where each output neuron resided was given in the configuration file. The placement of the output neurons on the SpiNNaker board allowed recording the membrane potential of each output neuron, accordingly the spikes of each neurons were written in the output file. An example of the PyNN script generated by GReaNs is shown in Appendix 6.

To sum up, integration of GReaNs with SpiNNaker using this protocol for the task of evolving a LIF SNNs to match the spike train of a single AdEx neuron shifted by 5 ms results in much slower performance.

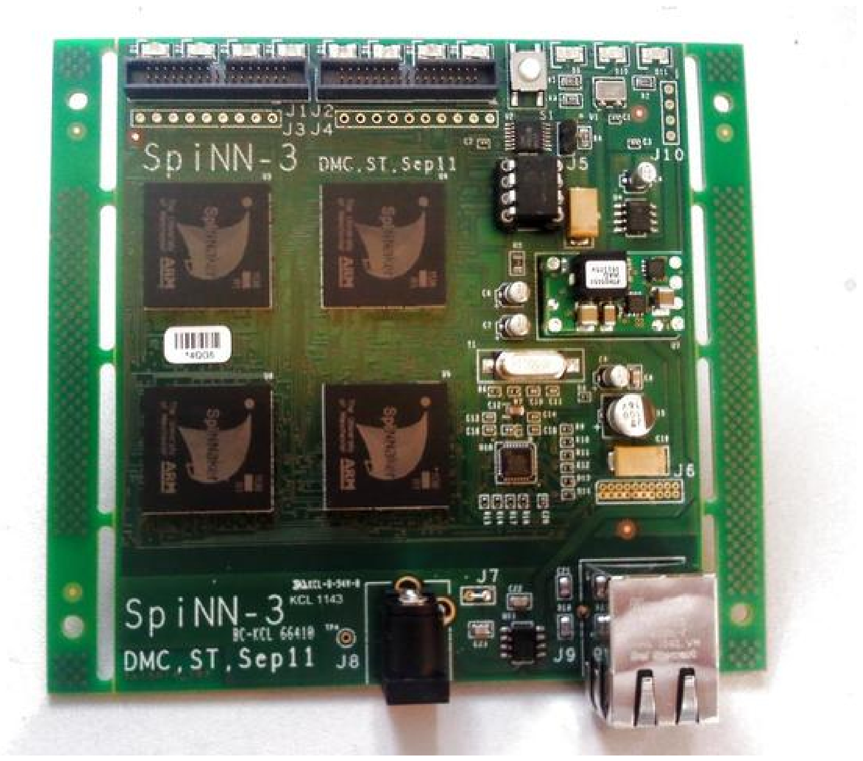


FIGURE 8.3: The small SpiNNaker board with only 4 chips which was used in the first version of the integration.

### 8.3.2 Communication protocol with the big SpiNNaker board

In this version a different task was run and a bigger board with 48 chips (Fig. 8.4) was used to support more computational power for the simulation. I ran the Temporal Pattern Recognition task (described in chapter 5) with the presence of noise during the evolution. Each SNN contained 10 neurons and a Gaussian noise with 10 ms standard deviation was used. The simulation time of each network with each input pattern and a random noise was 1 second. Each network should be simulated with 6 different input patterns and this should be repeated 100 times each with random noise (600 simulations or 600 seconds).

In the experiment described in section 8.3.1, the configuration time was the bottleneck of the integration. So it was important to find a suitable way of configuring the SpiNNaker board in order to avoid sending the same SNN 600 times to be simulated on SpiNNaker. For this reason, each network was sent only one time and was simulated with all the input patterns. I concatenated the different input patterns together in a long input stream with 1 second gap between each pattern to allow the activity in the network to die out. The patterns were randomly ordered. The out of the network should only fires with highest frequency when it receives the input pattern 1-2-3 and be silent for other patterns. The behaviour of the SNNs during the 1 second gaps was not neither rewarded nor penalized. This task is harder than the task presented in chapter 5. The activity of each network should be inhibited by the end of the 1 second in which it receives the pattern 1-2-3 (or in the 1 gap second after it).

I only used 100 random patterns for each network so the total simulation time of each network was 199 seconds (100 for the patterns and 99 for the gaps). Using my laptop again for simulation, each generation took around 1 minute. Using SpiNNaker for simulation, each generation took around 7 minutes (2:19 for the real time simulation and the rest for the configuration and generating the output file). As we can see, the simulation time of this task using my laptop (1 minute) is even less than the real time simulation (2:19 minutes), so it is impossible to get any improvement using SpiNNaker with this task.

## 8.4 Conclusion

After comparing the simulation time on my laptop and on SpiNNaker we can say that by using this protocol of communication between GReaNs and SpiNNaker we will not be able to get any improvement with SpiNNaker unless we use more individuals per generation or if larger networks are needed. Another possible way to get improvements in the simulation time is to run more than one task on SpiNNaker independently. One more suggestion that could dramatically improve the performance is to perform all the genetic algorithm steps on a SpiNNaker board so that we do not waste time in the communication between GReaNS and SpiNNaker. But in this case we will have to endure the hardship of implementing the genetic algorithm with a low level programming language.

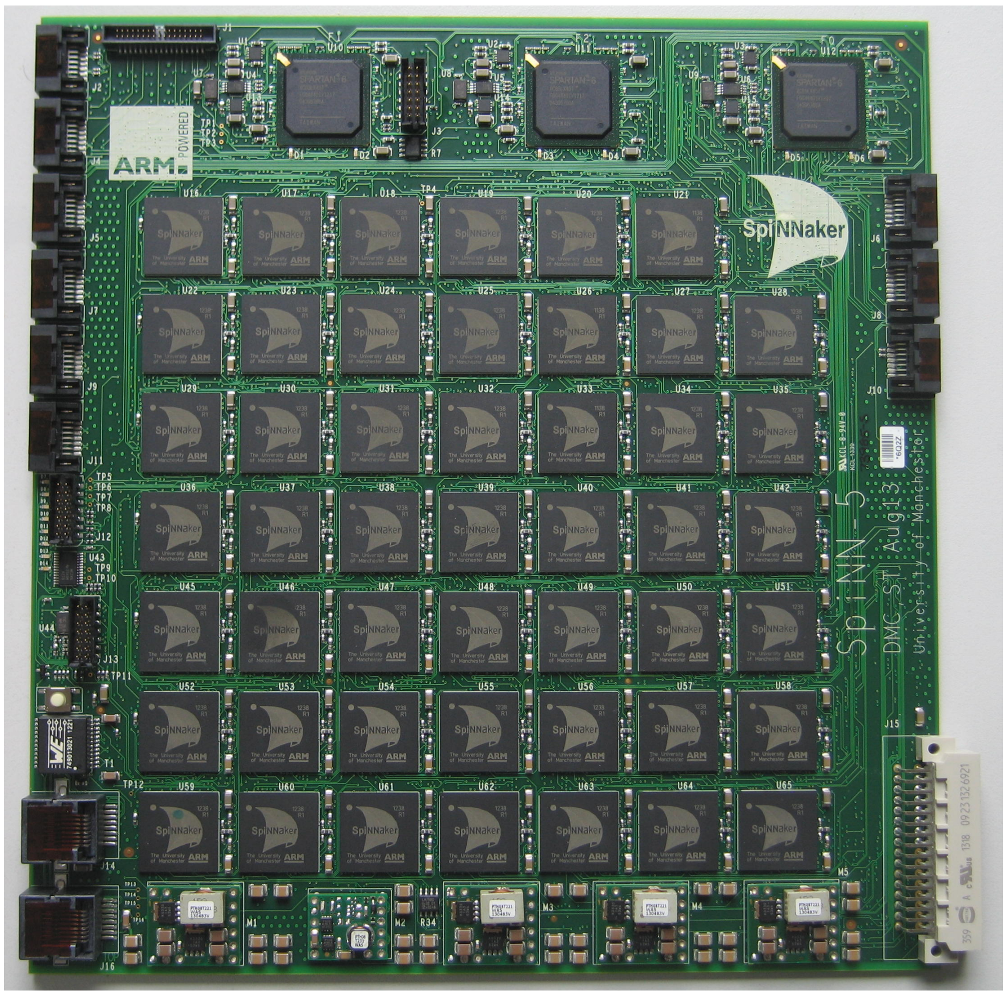


FIGURE 8.4: The big SpiNNaker board with 48 chips which was used in the second version of the integration.

# Chapter 9

# General Conclusion and Future Work

This chapter presents the general conclusion of the research work which I have done during my PhD studies. There is already a conclusion at the end of each chapter. Here I will go beyond the short conclusion sections of chapters 3-8 and merge all these conclusions together to give a big picture of the work I have done in all these chapters.

The second section in this chapter presents the future work that could be done from my point of view. This work could be an expansion for the model I have used for evolving Spiking Neural Networks (SNNs), more tasks in temporal pattern recognition, expanding the animat control tasks, and protocol for integration with more neuromorphic hardware.

## 9.1 General Conclusion

We can conclude from the work I had done during my PhD studies the following:

1. Due to some similarities in the structure of Gene Regulatory Networks (GRNs) and SNNs, it is possible to extend the GRNs to behave as a SNN. Furthermore, the evolutionary algorithm which was used for evolving the GRNs could be used for evolving the SNNs for simple tasks like matching spike trains (chapter 4) or even more difficult tasks like temporal pattern recognition (chapter 5) and animat control (chapters 6 and 7).

2. As shown in chapter 5, evolving the topology and the synaptic weights of SNNs using fixed synaptic delays is sufficient to obtain SNNs able to perform temporal

pattern recognition. These SNNs also showed robustness when Gaussian noise was added to the spike times of the inputs.

Results showed that the evolved SNNs were more robust to noise when the noise was present during the evolution. Positive feedback loops were playing an important role in moving the interneurons of the SNNs from one plateau state to another, until it became active. This plateau behaviour could only be observed when the SNNs have at least 2 interneurons, for this reason the SNNs with only 1 interneuron showed less robustness to noise.

3. Using the same evolutionary algorithm which depends on evolving the topology and the synaptic weight of the SNNs, SNNs can be evolved to control an animat for directional movement in a concentration gradient. The concentration gradient could be simply a smell of food, or furthermore could be a sound which is represented by a pattern of of inputs with various frequencies.

4. I managed to integrate the SNN evolutionary software that I extended to integrate with the SpiNNaker neuromorphic hardware. Despite the computational power of SpiNNaker which affords real time simulation for large SNNs, I did not manage to get any improvement in the time of the evolution. The main reason was that the tasks I used for the integration were too simple and did not require such computational power. The simulation time of the SNNs on CPU took less than the real time simulation of the SNNs. One more reason was that the configuration of SpiNNaker took long time and it was required to configure SpiNNaker for every generation during the evolutionary algorithm even when only the weights of the SNNs were updated.

   In order to get improvement in the simulation time with SpiNNaker, other tasks which require more computation power should be used. This tasks may require simulating SNNs with more neurons, or have more individuals in each generation. One more suggestion is to move the all genetic algorithm in SpiNNaker. This will help in avoiding configuring the SpiNNaker board for each generation.

## 9.2 Future Work

The future work that could be done based on my work is divided to three categories as following:

### 9.2.1 SNNs model and the evolutionary algorithm

More work can be done in extending the SNNs models I used. I have implemented only two SNNs models (Leaky Integrate-and-fire and Adaptive Exponential LIF), but more models could be implemented. One of these models could be Hodgkin-Huxley model [49].

Neuromorphic hardware systems, especially the analogue hardware systems, suffer from noise as a result of the mismatch of the electrical connections in the circuits on their boards. This noise can lead to unexpected behaviour of the SNN when it is simulated using these hardware systems. In order to avoid having different behaviour between simulating SNNs on a CPU and simulating them on neuromorphic hardware systems, this noise could be added to the SNNs when they are simulated on a CPU. These noises could include noise for the neuron parameters and the synaptic weight.

This work includes presenting more types of noise like noise for the neuron parameters and the synaptic weight. This noise will be also useful for integrating with neuromorphic hardware especially analogue hardware devices as they are more noisy. Adding this noise will simulate the mismatch of the electrical connections in the circuits on the board.

Furthermore, I had only implemented one type of AdEx neurons which corresponds to tonic spiking, but more types with other parameters could be added. This will allow the AdEx model to produce various realistic behaviours like phasic spiking, tonic bursting, and phasic bursting. These behaviours will enrich the genetic algorithm and will lead to more realistic behaviours for the SNNs.

The evolutionary algorithm could also be extended by allowing the evolution of the delays of the synapses between the neurons. Moreover, mutations could also be done on the changing the types of the LIF and AdEx neurons which I just presented in the previous paragraph.

### 9.2.2 Temporal pattern recognition

The temporal pattern recognition task is also considered an opened door for more work. One task that could be done is exploring the ability of SNNs to distinguish between different temporal patterns but all are presented at the same input stream during the same simulation. I have tried this task when I integrated with SpiNNaker system, but I was more interested in comparing the simulations times rather investigating the ability of evolving SNNs for this task.

The temporal pattern recognition task can be extended for general pattern recognition tasks. For example, SNNs can be evolved to control an animat for maze navigation. Imagine that we have an animat located in a maze and it is required to control this animat for course correction based on the pattern of signs (could be colours) it observes.

A finite state machine (FSM) can be used to recognize input patterns. FSM can be used to describe languages or to represent regular expressions. When the FSM receives an input, this input changes the state of the FSM until it reaches the final or the accepting state. The way FSM works is quite similar to the behaviour of the SNNs described in chapter 5 which were used for temporal pattern recognition. Thomas Natschläger and Wolfgang Maass presented an approach for using SNNs as finite state machines [87]. They used a learning algorithm based on training the synaptic weights of the SNNs to obtain finite state machines.

The temporal pattern recognition task can be extended in order to obtain finite state machine that could for example represent regular expressions. Each letter can be encoded as a temporal pattern. The training set of the SNNs should include both valid and invalid strings. The SNN should be trained so that the output neuron should be active with only the valid strings.

### 9.2.3 Evolving SNNs for animat control

I have presented in chapter 7 a model of evolving SNNs which control animats in order to distinguish between sound source types and to collect one of them. Each sound type was represented as a pattern of inputs with different frequencies. The evolutionary algorithm was not able to evolve an animat to achieve this task. The main reason behind this failure was that the evolutionary algorithm was not able to establish the temporal pattern recognition sub-problem and the animat tended to collect all the sound sources regardless their type.

One suggestion that could help the evolutionary algorithm to establish the pattern recognition part is to add to the fitness function a term that reward the SNNs which have an interneuron which performs temporal pattern recognition. The interneuron will act as the output neuron of the SNNs presented in chapter 5. Having an interneuron that could distinguish between pattern will allow the animat to show different behaviour based on the pattern of the sound source type.

This work can be extended for more interesting tasks. These tasks could include evolving SNNs to control simulated birds that have the characteristics of males and females. The females are flying and producing patterns of sounds (singing), and the males trying to follow and catch them. One further task is to co-evolve several simulated species of animats with a complex prey-predator relationship.

# Bibliography

[1] Louis Lapicque. Recherches quantitatives sur l'excitation electrique des nerfs traitée comme une polarization. *Journal de Physiologie et de Pathologie Générale*, 9:620–635, 1907.

[2] Peter Dayan and Laurence F. Abbott. *Theoretical neuroscience.* MIT Press, 2005. ISBN 0262541858.

[3] Wulfram Gerstner and Romain Brette. Adaptive exponential integrate-and-fire model as an effective description of neuronal activity. In *J Neurophysiol*, volume 94, pages 3637–3642, 2005.

[4] Borys Wróbel and Michał Joachimczak. Using the genetic regulatory evolving artificial networks (greans) platform for signal processing, animat control, and artificial multicellular development. In Taras Kowaliw, Nicolas Bredeche, and Doursat René, editors, *Growing Adaptive Machines.* Springer Berlin Heidelberg, Berlin, 2014.

[5] William Bialek, Fred Rieke, Robert Raimond de Ruyter van Steveninck, and David Warland. Reading a neural code. *Science*, 252:1854–1857, 1991.

[6] Wulfram Gerstner, Richard Kempter, J Leo van Hemmen, and Henry Wagner. A neuronal learning rule for sub-millisecond temporal coding. *Nature*, 383:76–78, 1996.

[7] Gilles Laurent. Dynamical representation of odors by oscillating and evolving neural assemblies. *Trends in Neurosciences*, 19:489–496, 1996.

[8] Fred Rieke, David Warland, Robert Raimond de Ruyter van Steveninck, and William Bialek. *Spikes: Exploring the Neural Code.* MIT Press, 1997.

[9] R Christopher deCharms and Anthony Zador. Neural representation and the cortical code. *Neuroscience*, 23:613–647, 2000.

[10] Ehud Ahissar and Amos Arieli. Figuring space by time. *Neuron*, 32:185–201, 2001.

[11] John Huxter, Neil Burgess, and John O'Keefe. Independent rate and temporal coding in hippocampal pyramidal cells. *Nature*, 425:828–832, 2003.

[12] Philip Joris and Tom C T Yin. A matter of time: Internal delays in binaural processing. *Trends in Neurosciences*, 30:70–78, 2007.

[13] Jeffry Isaacson. Odor representations in mammalian cortical circuits. *Current Opinion in Neurobiology*, 20:328–331, 2010.

[14] John Hopfield. Pattern recognition computation using action potential timing for stimulus representation. *Nature*, 376:33–36, 1995.

[15] Thomas Natschläger and Berthold Ruf. Spatial and temporal pattern analysis via spiking neurons. *Network: Computation in Neural Systems*, 9:319–332, 1998.

[16] Volker Steuber and David Willshaw. Adaptive leaky integrator models of cerebellar purkinje cells can learn the clustering of temporal patterns. *Neurocomputing*, 26: 271–276, 1999.

[17] Volker Steuber and Erik De Schutter. Rank order decoding of temporal parallel fibre input patterns in a complex purkinje cell model. *Neurocomputing*, 44:183–188, 2002.

[18] Volker Steuber and David Willshaw. A biophysical model of synaptic delay learning and temporal pattern recognition in a cerebellar purkinje cell. *Journal of Computational Neuroscience*, 17:149–164, 2004.

[19] Tristram D. Wyatt. *Pheromones and Animal Behaviour: Communication by Smell and Taste*. Cambridge University Press, 2003.

[20] Ryohei Kanzaki. How does a microbrain generate adaptive behavior? In *International Congress Series*, volume 1301, pages 7–14. Elsevier, 2007.

[21] Nicolas Oros, Volker Steuber, Neil Davey, Lola Cañamero, and Rod Adams. Optimal noise in spiking neural networks for the detection of chemicals by simulated agents. In *Proceedings on the Eleventh International Conference on Artificial Life*, 2008.

[22] Nicolas Oros, Volker Steuber, Neil Davey, Lola Cañamero, and Rod Adams. Adaptive olfactory encoding in agents controlled by spiking neural networks. In *Lecture Notes in Computer Science: From Animals to Animats 10*, pages 148–158, 2008.

[23] Nicolas Oros, RG Adams, Neil Davey, Lola Cañamero, and Volker Steuber. Encoding sensory information in spiking neural network for the control of autonomous agents. 2008.

[24] Nicolas Oros, Volker Steuber, Neil Davey, Lola Cañamero, and Rod Adams. Evolution of bilateral symmetry in agents controlled by spiking neural networks. In *Artificial Life, 2009. ALife'09. IEEE Symposium on*, pages 116–123. IEEE, 2009.

[25] Nicolas Oros. *An Artificial Life Perspective on Olfactory Systems: Evolving Neural Coding, Developmental Symmetry and Odour Recognition in Agents.* PhD thesis, University of Hertfordshire, 2010.

[26] Michał Joachimczak and Borys Wróbel. Evolving gene regulatory networks for real time control of foraging behaviours. In *Artificial Life XII: Proceedings of the Twelfth International Conference on the Simulation and Synthesis of Living Systems*, pages 348–355. MIT Press, 2010.

[27] Borys Wróbel, Michał Joachimczak, Alberto Montebelli, and Robert Lowe. The search for beauty: Evolution of minimal cognition in an animat controlled by a gene regulatory network and powered by a metabolic system. volume 7426, pages 198–208. Springer Berlin Heidelberg, 2012.

[28] Alexander D. Rast, Xin Jin, Francesco Galluppi, Luis A Plana, Cameron Patterson, and Steve Furber. Scalable event-driven native parallel processing: the spinnaker neuromimetic system. In *Proceedings of the 7th ACM international conference on Computing frontiers*, pages 21–30, 2010.

[29] Jonathan Touboul. Bifurcation analysis of a general class of nonlinear integrate-and-fire neurons. *SIAM Journal on Applied Mathematics*, 68:1045–1079, 2008.

[30] Michał Joachimczak and Borys Wróbel. Evo-devo *in silico*: a model of a gene network regulating multicellular development in 3D space with artificial physics. In *Artificial Life XI: Proceedings of the Eleventh International Conference on the Simulation and Synthesis of Living Systems*, pages 297–304. MIT Press, 2008.

[31] Michał Joachimczak and Borys Wróbel. Evolution of the morphology and patterning of artificial embryos: Scaling the tricolour problem to the third dimension. In *Advances in Artificial Life. Darwin Meets von Neumann: Proceedings of the Tenth European Conference on Artificial Life (ECAL 2009)*, volume 5777 of *Lecture Notes in Computer Science*, pages 35–43. Springer, 2011.

[32] Michał Joachimczak and Borys Wróbel. Open ended evolution of 3d multicellular development controlled by gene regulatory networks. In *Artificial Life XIII: Proceedings of the Thirteenth International Conference on the Simulation and Synthesis of Living Systems*, pages 67–74. MIT Press, 2012.

[33] Michał Joachimczak and Borys Wróbel. Processing signals with evolving artificial gene regulatory networks. In *Artificial Life XII: Proceedings of the Twelfth International Conference on the Simulation and Synthesis of Living Systems*, pages 203–210. MIT Press, 2010.

[34] Andrew P. Davison, Daniel Brüderle, Jochen Eppler, Jens Kremkow, Eilif Muller, Dejan Pecevski, Laurent Perrinet, and Pierre Yger. Pynn: A common interface for neuronal network simulators. *Frontiers in neuroinformatics*, 2, 2008. ISSN 1662-5196.

[35] Dan Goodman and Romain Brette. Brian: a simulator for spiking neural networks in python. *Frontiers in Neuroinformatics*, 2, 2008. ISSN 1662-5196.

[36] Suzana Herculano-Houzel, Kamilla Avelino, Kleber Neves, Jairo Porfirio, Debora Messeder, Larissa Mattos Feijó, José Maldonado, and Paul Manger. The elephant brain in numbers. *Frontiers in Neuroanatomy*, 8(46), 2014.

[37] Alois Herzig, Ramesh D Gulati, Christian D Jersabek, and Linda May. *Rotifera X: Rotifer Research: Trends, New Tools and Recent Advances*, volume 181. Springer Science & Business Media, 2006.

[38] Frederico AC Azevedo, Ludmila RB Carvalho, Lea T Grinberg, José Marcelo Farfel, Renata EL Ferretti, Renata EP Leite, Roberto Lent, Suzana Herculano-Houzel, et al. Equal numbers of neuronal and nonneuronal cells make the human brain an isometrically scaled-up primate brain. *Journal of Comparative Neurology*, 513: 532–541, 2009.

[39] Wulfram Gerstner and Werner M Kistler. *Spiking neuron models: Single neurons, populations, plasticity.* Cambridge University Press, 2002.

[40] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics*, 5:115–133, 1943.

[41] John J Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the national academy of sciences*, 79:2554–2558, 1982.

[42] G Joya, MA Atencia, and F Sandoval. Hopfield neural networks for optimization: study of the different dynamics. *Neurocomputing*, 43(1):219–237, 2002.

[43] Wulfram Gerstner and Werner M. Kistler. *Spiking Neuron Models.* Cambridge University Press, 2002.

[44] Wolfgang Maass. Networks of spiking neurons: The third generation of neural network models. *Neural Networks*, 10:1659–1671, 1997.

[45] Wolfgang Maass and Christopher M Bishop. *Pulsed neural networks.* MIT press, 1999.

[46] Samanwoy Ghosh-Dastidar and Hojjat Adeli. Spiking neural networks. *International journal of neural systems*, 19:295–308, 2009.

[47] W. Gerstner and R. Brette. Adaptive exponential integrate-and-fire model. *Scholarpedia*, 4:8427, 2009.

[48] E.M. Izhikevich. *Dynamical Systems in Neuroscience: The Geometry of Excitability and Bursting*. MIT press, 2007.

[49] Alan L Hodgkin and Andrew F Huxley. A quantitative description of membrane current and its application to conduction and excitation in nerve. *The Journal of physiology*, 117:500, 1952.

[50] AL Hodgkin and AF Huxley. The components of membrane conductance in the giant axon of loligo. *The Journal of physiology*, 116(4):473–496, 1952.

[51] Allan L Hodgkin and Andrew F Huxley. Currents carried by sodium and potassium ions through the membrane of the giant axon of loligo. *The Journal of physiology*, 116(4):449–472, 1952.

[52] Allan L Hodgkin and Andrew F Huxley. The dual effect of membrane potential on sodium conductance in the giant axon of loligo. *The Journal of physiology*, 116(4): 497–506, 1952.

[53] Ao L Hodgkin, AF Huxley, and B Katz. Measurement of current-voltage relations in the membrane of the giant axon of loligo. *The Journal of physiology*, 116(4): 424–448, 1952.

[54] Eugene M Izhikevich. Which model to use for cortical spiking neurons? *IEEE Transactions on Neural Networks*, 15(5):1063–1070, 2004.

[55] Marc-Oliver Gewaltig and Markus Diesmann. Nest (neural simulation tool). *Scholarpedia*, 2:1430, 2007.

[56] T. Carnevale. Neuron simulation environment. 2:1378, 2007. revision 91576.

[57] Rae Silver, Kwabena Boahen, Sten Grillner, Nancy Kopell, and Kathie L Olsen. Neurotech for neuroscience: unifying concepts, organizing principles, and emerging tools. *The Journal of Neuroscience*, 27:11807–11819, 2007.

[58] Antoine Joubert, Bilel Belhadj, Olivier Temam, and Rodolphe Heliot. Hardware spiking neurons design: Analog or digital? In *The 2012 International Joint Conference on Neural Networks (IJCNN)*, pages 1–5. IEEE, 2012.

[59] Simon J. Thorpe, Arnaud Delorme, and Rufin VanRullen. Spike-based strategies for rapid processing. *Neural Networks*, 14:715–725, 2001.

[60] Vernon B Mountcastle. Modality and topographic properties of single neurons of cat's somatic sensory cortex. *Journal of Neurophysiology*, 20:408–434, 1957.

[61] David H Hubel and Torsten N Wiesel. Receptive fields of single neurones in the cat's striate cortex. *The Journal of Physiology*, 148:574, 1959.

[62] Edgar D Adrian. *The basis of sensation.* WW Norton & Company, 1928.

[63] Edgar D Adrian and Yngve Zotterman. The impulses produced by sensory nerve-endings part ii. the response of a single end-organ. *The Journal of physiology*, 61: 151–171, 1926.

[64] Simon Thorpe, Denis Fize, and Catherine Marlot. Speed of processing in the human visual system. *Nature*, 381:520–522, 1996.

[65] Volker Steuber. *Computational Models of Intracellular Signalling in Cerebellar Purkinje Cells.* PhD thesis, The University of Edinburgh, 1998.

[66] Volker Steuber, David Willshaw, and Arjen Van Ooyen. Generation of time delays: Simplified models of intracellular signalling in cerebellar purkinje cells. *Network: Computation in Neural Systems*, 17:173–191, 2006.

[67] Eric Kandel, James H. Schwartz, and Thomas Jessell. *Principles of Neural Science 4th Edition.* McGraw-Hill, 2000.

[68] Valentino Braitenberg. *Vehicles: Experiments in Synthetic Psychology.* MIT Press, Cambridge, MA, 1984.

[69] George E Uhlenbeck and Leonard Salomon Ornstein. On the theory of the brownian motion. *Physical review*, 36:823–841, 1930.

[70] Michael Rudolph and Alain Destexhe. Characterization of subthreshold voltage fluctuations in neuronal membranes. *Neural Computation*, 15:2577–2618, 2003.

[71] John H Holland. Adaptation in natural and artificial systems: An introductory analysis with applications to biology, control, and artificial intelligence. 1975.

[72] Lawrence Davis. Genetic algorithms and simulated annealing. 1987.

[73] Lawrence Davis. Handbook of genetic algorithms. 1991.

[74] John J Grefenstette. Optimization of control parameters for genetic algorithms. *Systems, Man and Cybernetics, IEEE Transactions on*, 16(1):122–128, 1986.

[75] David E Goldberg and John H Holland. Genetic algorithms and machine learning. *Machine learning*, 3(2):95–99, 1988.

[76] Zbigniew Michalewicz. *Genetic algorithms+ data structures= evolution programs.* Springer Science & Business Media, 2013.

[77] Guy Karlebach and Ron Shamir. Modelling and analysis of gene regulatory networks. *Nature Reviews Molecular Cell Biology*, 9(10):770–780, 2008.

[78] Michael Levine and Eric H Davidson. Gene regulatory networks for development. *Proceedings of the National Academy of Sciences of the United States of America*, 102(14):4936–4942, 2005.

[79] Jeff Hasty, David McMillen, Farren Isaacs, and James J Collins. Computational studies of gene regulatory networks: in numero molecular biology. *Nature Reviews Genetics*, 2(4):268–279, 2001.

[80] Michał Joachimczak, Taras Kowaliw, Rene Doursat, and Borys Wróbel. Brainless bodies: Controlling the development and behavior of multicellular animats by gene regulation and diffusive signals. In *Artificial Life XIII: Proceedings of the Thirteenth International Conference on the Simulation and Synthesis of Living Systems*, pages 349–356. MIT Press, 2012.

[81] Michał Joachimczak and Borys Wróbel. Co-evolution of morphology and control of soft-bodied multicellular animats. In *Proceedings of the Fourteenth International Conference on Genetic and Evolutionary Computation*, GECCO '12, pages 561–568. ACM, 2012.

[82] Michał Joachimczak. *Evolution of gene regulatory networks and artificial embryogenesis in a simulated 3D environment.* PhD thesis, Gdansk University of Technology, 2012.

[83] Borys Wróbel, Ahmed Abdelmotaleb, and Michał Joachimczak. Evolving spiking neural networks in the greans (gene regulatory evolving artificial networks) platform. pages 19–22. EvoNet2012: Evolving Networks, from Systems/Synthetic Biology to Computational Neuroscience Workshop at Artificial Life XIII, 2008.

[84] Borys Wróbel, Ahmed Abdelmotaleb, and Michał Joachimczak. Evolving networks processing signals with a mixed paradigm, inspired by gene regulatory networks and spiking neurons. In *The 7th International Conference on Bio-Inspired Models of Network, Information, and Computing Systems (BIONETICS)*, 2012.

[85] Ahmed Abdelmotaleb, Neil Davey, Maria Schilstra, Volker Steuber, and Borys Wróbel. Evolving spiking neural networks for temporal pattern recognition in the presence of noise. pages 965–972. Artificial Life 2014, MIT Press, 2014.

[86] John Carew Eccles, P Fatt, and K Koketsu. Cholinergic and inhibitory synapses in a pathway from motor-axon collaterals to motoneurones. *The Journal of Physiology*, 126:524–562, 1954.

[87] Thomas Natschläger and Wolfgang Maass. Spiking neurons and the induction of finite state machines. *Theoretical Computer Science*, 287:251–265, 2002.

# An example of a PyNN script generated by GReaNS

```
#----------------------------------------------------------------------------
#-------------------------The Header-----------------------------------------
#----------------------------------------------------------------------------
 import pacman103.front.pynn as p
p.setup(timestep=1.0, min_delay = 1.0, max_delay = 1.0)

#----------------------------------------------------------------------------
#----------------Output populations part-------------------------------------
#----------------------------------------------------------------------------


output_population_params0 = {'v_rest':[-70, -70, -70, -70, -70, -70, -70, -70, -70, -70, -70,
-70, -70, -70, -70, -70, -70, -70, -70, -70, -70, -70, -70, -70, -70, -70, -70, -70, -70, -70, -70,
-70, -70, -70, -70, -70, -70, -70, -70, -70, -70, -70, -70, -70, -70, -70, -70, -70, -70, -70, -70,
-70, -70, -70, -70, -70, -70, -70, -70, -70, ],
'tau_m':[20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20,
20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20,
20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, ],
'i_offset':[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ],
'v_thresh':[-50, -50, -50, -50, -50, -50, -50, -50, -50, -50, -50, -50, -50, -50, -50, -50, -50,
-50, -50, -50, -50, -50, -50, -50, -50, -50, -50, -50, -50, -50, -50, -50, -50, -50, -50, -50, -50,
-50, -50, -50, -50, -50, -50, -50, -50, -50, -50, -50, -50, -50, -50, -50, -50, -50, -50, -50, -50,
-50, -50, -50, ],
'tau_syn_E':[5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5,
5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, ],
'tau_syn_I':[5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5,
5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, ],
'v_reset':[-70, -70, -70, -70, -70, -70, -70, -70, -70, -70, -70, -70, -70, -70, -70, -70, -70, -70,
-70, -70, -70, -70, -70, -70, -70, -70, -70, -70, -70, -70, -70, -70, -70, -70, -70, -70, -70, -70,
-70, -70, -70, -70, -70, -70, -70, -70, -70, -70, -70, -70, -70, -70, -70, -70, -70, -70, -70, -70,
-70, -70, ],
'cm':[20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20,
20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20,
20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, ]}
output_population0 = p.Population(60, p.IF_curr_exp, output_population_params0, la-
bel='output_pop0')
output_population0.set_mapping_constraint({'x':0, 'y':0, 'p':2})
#——————————————————————————————————————
#——The rest of the output populations are written here——————
#——————————————————————————————————————


#——————————————————————————————————————
#————————-Main populations part——————————————————
#——————————————————————————————————————
 main_population_params0 = {'v_rest':[-70,-70,-70,-70,-70,],
                            'tau_m':[20,20,20,20,20,],
                            'i_offset':[0,0,0,0,0,],
                            'v_thresh':[-50,-50,-50,-50,-50,],
                            'tau_syn_E':[5,5,5,5,5,],
                            'tau_syn_I':[5,5,5,5,5,],
                            'v_reset':[-70,-70,-70,-70,-70,],
                            'cm':[20,20,20,20,20,]}
main_population0 = p.Population(5, p.IF_curr_exp, main_population_params0, label='main_pop0')
main_population_params1 = {'v_rest':[-70,-70,-70,-70,-70,],
                            'tau_m':[20,20,20,20,20,],
                            'i_offset':[0,0,0,0,0,],
                            'v_thresh':[-50,-50,-50,-50,-50,],
                            'tau_syn_E':[5,5,5,5,5,],
                            'tau_syn_I':[5,5,5,5,5,],
                            'v_reset':[-70,-70,-70,-70,-70,],
                            'cm':[20,20,20,20,20,]}
main_population1 = p.Population(5, p.IF_curr_exp, main_population_params1, label='main_pop1')
#——————————————————————————————————————
#——The rest of the main populations are written here——————
```

```
#------------------------------------------------------------------------------


#------------------------------------------------------------------------------
#----------------Input populations part ----------------------------------------
#------------------------------------------------------------------------------
 spike_times0 = [[50,],[150,],[250,],]
spike_source0 = p.Population(3, p.SpikeSourceArray,{'spike_times': spike_times0})
spike_times1 = [[50,],[150,],[250,],]
spike_source1 = p.Population(3, p.SpikeSourceArray,{'spike_times': spike_times1})
#------------------------------------------------------------------------------
#----The rest of the input populations are written here-------------
#------------------------------------------------------------------------------


#--------------------------------------------------------------------
#--------Setting the weights of all the synapses---------
#--------------------------------------------------------------------


#--------------------------------------------------------------------
#-----Synapses for the individual number 0 --------------
#--------------------------------------------------------------------
 #Weights of the excitatory synapses between the neurons in the same main network
mainPopExc0 = []
#Weights of the inhibitory synapses between the neurons in the same main network
mainPopInh0 = []
#Weights of the excitatory synapses between the neurons in the main network and the
spike source population
mainPopSpikeSourceExc0 = []
#Weights of the inhibitory synapses between the neurons in the main network and the
spike source population
mainPopSpikeSourceInh0 = []
#Weights of the excitatory synapses between the neurons in the main network and the
output population
mainPopOutputExc0 = []
#Weights of the inhibitory synapses between the neurons in the main network and the
output population
mainPopOutputInh0 = []
mainPopExc0.append((0,3, 0.001395, 1.0))
```

```
mainPopInh0.append((0,4, 0.111922, 1.0))
mainPopInh0.append((1,0, 0.001757, 1.0))
mainPopInh0.append((3,2, 0.018668, 1.0))
mainPopExc0.append((3,3, 0.060916, 1.0))
mainPopInh0.append((3,4, 0.084304, 1.0))
mainPopExc0.append((4,0, 0.006365, 1.0))
mainPopInh0.append((4,4, 0.057073, 1.0))
mainPopSpikeSourceExc0.append((1,1, 0.044086, 1.0))
mainPopSpikeSourceInh0.append((1,2, 0.074775, 1.0))
mainPopSpikeSourceExc0.append((1,3, 0.462100, 1.0))
mainPopSpikeSourceInh0.append((1,4, 0.021583, 1.0))
mainPopSpikeSourceInh0.append((2,1, 0.006121, 1.0))
mainPopSpikeSourceExc0.append((2,2, 0.031790, 1.0))
mainPopSpikeSourceInh0.append((2,3, 0.088703, 1.0))
mainPopSpikeSourceExc0.append((2,4, 0.069823, 1.0))
mainPopToMainPopExcProj0 = p.Projection ( main_population0, main_population0,
p.FromListConnector (mainPopExc0), target = 'excitatory')
mainPopToMainPopInhProj0 = p.Projection ( main_population0 , main_population0,
p.FromListConnector (mainPopInh0), target='inhibitory')
spikeSourcePopToMainPopExcProj0 = p.Projection (spike_source0, main_population0,
p.FromListConnector (mainPopSpikeSourceExc0), target='excitatory')
spikeSourcePopToMainPopInhProj0 = p.Projection ( spike_source0, main_population0,
p.FromListConnector ( mainPopSpikeSourceInh0 ) , target = 'inhibitory' )
MainPopToOutputPopExcProj0 = p.Projection ( main_population0 , output_population0,
p.FromListConnector (mainPopOutputExc0) , target = 'excitatory' )
MainPopToOutputPopInhProj0 = p.Projection ( main_population0 , output_population0
, p.FromListConnector (mainPopOutputInh0) , target = 'inhibitory' )
#——————————————————————————————————
#——The weights of the rest of synapses are set here—————
#——————————————————————————————————


#——————————————————————————————————
#———————————The Footer————————————————
#——————————————————————————————————
 output_population0.record()
output_population1.record()
output_population2.record()
output_population3.record()
```

```
output_population4.record()
p.run(11000)
spikes_0 = populations[0].getSpikes(compatible_output=True)
spikes_1 = populations[1].getSpikes(compatible_output=True)
spikes_2 = populations[2].getSpikes(compatible_output=True)
spikes_3 = populations[3].getSpikes(compatible_output=True)
spikes_4 = populations[4].getSpikes(compatible_output=True)
f = open('spike_times.txt', 'w')
for i in spikes_0:
        f.write(str(int(i[1])) + ';' + str(int(i[0]) + 0) + '\n')
for i in spikes_1:
        f.write(str(int(i[1])) + ';' + str(int(i[0]) + 60) + '\n')
for i in spikes_2:
        f.write(str(int(i[1])) + ';' + str(int(i[0]) + 120) + '\n')
for i in spikes_3:
        f.write(str(int(i[1])) + ';' + str(int(i[0]) + 180) + '\n')
for i in spikes_4:
        f.write(str(int(i[1])) + ';' + str(int(i[0]) + 240) + '\n')
```

# Published Papers

- Ahmed Abdelmotaleb, Neil Davey, Maria Schilstra, Volker Steuber, and Borys Wrobel. Evolving spiking neural networks for temporal pattern recognition in the presence of noise. pages 965-972. Artificial Life 2014, MIT Press, 2014.
- Borys Wrobel, Ahmed Abdelmotaleb, and Michal Joachimczak. Evolving network-sprocessing signals with a mixed paradigm, inspired by gene regulatory networks and spiking neurons. In The 7th International Conference on Bio-Inspired Models of Network, Information, and Computing Systems (BIONETICS), 2012.
- Borys Wrobel, Ahmed Abdelmotaleb, and Michal Joachimczak. Evolving spiking neural networks in the greans (gene regulatory evolving artificial networks) platform. pages 19-22. EvoNet2012: Evolving Networks, from Systems/Synthetic Biology to Computational Neuroscience Workshop at Artificial Life XIII, 2008.