Imperial College London
Department of Computing

Evolving spiking neural network controllers for virtual animats that explore and forage in unknown environments

Alexandre de Brébisson (ahd11)

Submitted in partial fulfilment of the requirements for the
MSc Degree in Advanced Computing of Imperial College London

September 2012

Abstract

This project is about designing and simulating some autonomous mobile animats controlled by spiking neural networks. Genetic algorithms will be used to evolve the controllers.

Issues are threefold.

Firstly it intends to find some *efficient* spiking neural controllers for evolutionary robotics using the Quadratic Integrate and Fire model (QIF) and Izhikevich's model. Some comparisons with the more classic Continuous Time Recurrent Neural Networks (CTRNN) are conducted using some introduced indexes to characterise their *efficiency.* Results show that spiking neuro-controllers (and especially Izhikevich's based ones) are faster to converge, less sensitive to the genetic initial population and better in escaping local minima.

Secondly the project studies the capability of such controlled animats to perform some biologically inspired tasks. Some successful animats have been designed to explore, forage, graze and diversify their diet in unknown environments. Interesting behaviours emerge such as doing some periodic pirouettes to efficiently scan the world.

Thirdly the dynamics of corresponding neural controllers is studied. Some controllers appear to swap between stable and unstable attractive modes with different corresponding behaviours.
Experiments show that background noise current can play an important and useful role by generating randomness in behaviours.
Artificial lesions, such as removing connections between neurons, are carried out and show that spiking controllers have less crucial connections than CTRNN ones.

Acknowledgements

I would like to thank Professor Murray Shanahan for the enthusiasm, the advice and the explanations he has provided me all along the project.

Thanks also to my classmates Francesco and Richard for their interest in my project and the rewarding discussions we have had.

Thanks to the proof-reader who will recognize himself.

I also express my gratitude to my flatmates who have continually been playing with the simulator and who convinced me to assign a green color to food items and a red one to poison items.

Contents

Introduction

Evolutionary robotics is the scientific field where robots are designed with evolutionary methods, in particular genetic algorithms. It originated in the beginning of the 90s and rapidly spread out over many labs around the world. Genetic algorithms rapidly appear to be an efficient way to design neuro-controllers for various tasks. Nevertheless some limiting problems arose such as the time consumption of the algorithms and their convergence when the complexity of the robots and of the tasks scales up.

Different types of neuro-controllers have been studied in evolutionary robotics. Feed forward static architectures have made way to recurrent dynamic architectures with new types of neurons. Nowadays Continuous Time Recurrent Neural Networks (CTRNN) are commonly used because they are capable of displaying complex dynamics. Some biologically more realistic spiking neurons have been tested and in specific situations appear to be more efficient than artificial neurons. Nevertheless this field hasn't been much investigated yet compared to CTRNN controllers.

This project therefore focuses on the use of spiking neural networks to control biologically-inspired robots in animals inspired exploration and foraging tasks. Dynamics of such evolved controllers will be studied. The report is divided in sixth chapters:

1. First chapter is about the background of the thesis. It will go through the main fields related to this project.
2. In the second chapter the robot design, its controller, its environment, the evolution algorithm, some experiments procedures and some comparison indexes are described.
3. The third chapter is dedicated to the robot simulator software that has been designed in order to test the controllers. It is perfectly possible to skip it by only reading the last section that summarizes the functionalities of the software and the challenge that have been faced.
4. Fourth chapter is about a foraging experiment.
5. An experiment requiring exploration and grazing from the robots is conducted in the fifth chapter.
6. An experiment where the robot has to diversify its diet is conducted in the sixth chapter.

The appendices relate some unsuccessful experiments and the possible reasons of their failures.

1 | Background

In this chapter, some of the key subjects related to the project are discussed and some terms are defined. In particular, neural networks models and genetic algorithms are introduced.

1.1 Animats

An animat is a biologically-inspired robot imitating some animals behaviours such as exploration of an unknown environment or foraging. In 1984, Braitenberg published his famous book *Vehicles: Experiments in Synthetic Psychology* [1] where he describes some simple animats with direct connections between sensors and motors.
An animat is composed of the three following elements [2]:

- sensors to probe its environment,
- a controller that receives the sensors signals as inputs and computes outputs,
- actuators that receive the controller's outputs as inputs.

Their study is interesting to design flexible, robust and autonomous robots that can operate in unknown situations, and also to understand some of the autonomy and decisions making processes of animals [3].

As explained in 2.2, the robots used in this project are animats.

1.2 Static versus dynamic neural networks

Neural networks can be classified in two categories.

Static neural networks (also named feedforward neural networks) compute the outputs directly from the inputs through feedforward connections. There is not loop in the network and the outputs therefore only depend on the current inputs (there are no delays as well).

On the other hand dynamic neural networks have recursion which can take different forms: loops in the network, recurrent neurons or state variables in neuron models. Therefore, their outputs not only depend on the inputs but on the previous states of the network as well. Such networks can therefore have more complex dynamics such as displaying time-dependant patterns or exhibiting memory capabilities. Nevertheless they are more difficult to train. These neural networks are more biologically realistic as animal brain networks are also dynamic.

Note that a dynamic neural network is said to be *fully recurrent* if every neuron is connected to all other neurons. Some neurons are the output ones whereas the others are called *hidden neurons*.

Two categories of dynamic neural networks will be used in this project: *Continuous Time Recurrent Neural Networks (CTRNN)* and some *recurrent spiking neural networks* that are more biologically plausible. Both are continuous-time in opposition to discrete-time.

It is particularly difficult to find by hand the appropriate weights of such networks in order to perform a given task. Evolutionary robotics is a well-known way to compute them.

1.3 Continuous-time recurrent neural networks

The Continuous Time Recurrent Neural Network was introduced by Beer in 1990 [4] and grew up quickly in the robot controllers field. Five years later, it was shown that it could approximate the trajectory of any dynamical system [5]. Beer showed that such networks are also capable of chaotic behaviours.

A CTRNN neuron (also called leaky integrator) is described by its activation potential y_i that varies according to the following differential equations.

$$\tau_i \frac{dy_i}{dt} = -y_i + \sum_{j=1}^{n} w_{ij}\sigma(y_j - \Theta_j) + \sum_{k=1}^{m} w_{ik} I_k(t)$$

Where:

- τ_i : Time constant of node i
- y_i : Potential of node i
- w_{ij} : Weight of connection from node j to node i
- $\sigma(x)$: Sigmoid of x
- y_j : Potential of node j
- Θ_j : Bias of node j
- w_{ik} : Weight of connection from external input k to node i
- $I_k(t)$: External input k (if any)

y_i can be seen as the membrane potential of a neuron and $\sigma(x)$ as its mean firing rate making the model biologically plausible.
In practise, they are capable to display rich dynamics and to learn complex patterns. Blynel and Floreano [6] successfully designed CTRNN controlled robots capable to remember the correct path in a simple maze.

1.4 Spiking neural networks

Spiking models are a class of more realistic neurons than artificial or CTRNN neurons: they imitate the electrical signals and potentials of biological neurons. Each spiking neuron is characterised by a membrane potential. A neuron is said to *fire* when its membrane potential reaches a specific threshold. When it fires, it sends a spike towards all other connected neurons. Its membrane potential is then reset and the neuron cannot fire for a short period of time called the *refractory period.*
The output of a spiking neuron is therefore binary (spike or not) but it can be converted in a continuous signal over time. In this project, a frequency approach is used: the activity of a neuron over a short period of time is converted into a mean firing rate.

According to [7], spiking neural networks need fewer nodes to solve problems than conventional artificial neural networks. They are said to be computationally more powerful.

1.4.1 Quadratic and Fire neuron model

Quadratic and Fire (QIF) neuron is a spiking neural model which potential v is defined by the following equation:

$$\tau \frac{dv}{dt} = a(v_r - v)(v_c - v) + RI$$

Where:

- v : Neuron potential
- τ : Time constant
- v_r : Resting potential of the neuron
- v_c : Critical potential
- I : Input current of the neuron
- R : Resistance constant
- a : constant factor

The firing process takes place if the potential v reaches a threshold and if the refractory period has run out:

If $\begin{cases} v > V \\ t - t_{spike} > a \end{cases}$ then the neuron spikes the variables are reset: $\begin{cases} v \leftarrow v_r \\ t_{spike} \leftarrow t \end{cases}$.

With $R = 1$, $\tau = 5$, $v_r = -65$, $v_c = -50$, $a = 0.2$, $V = -30$ and $I = 20$ the potential of the QIF neuron is approximated on figure 1.1 using Euler method with a step size $dt = 0.2$. These parameters are suitable to approximate the activity of a regular excitatory biological neuron.

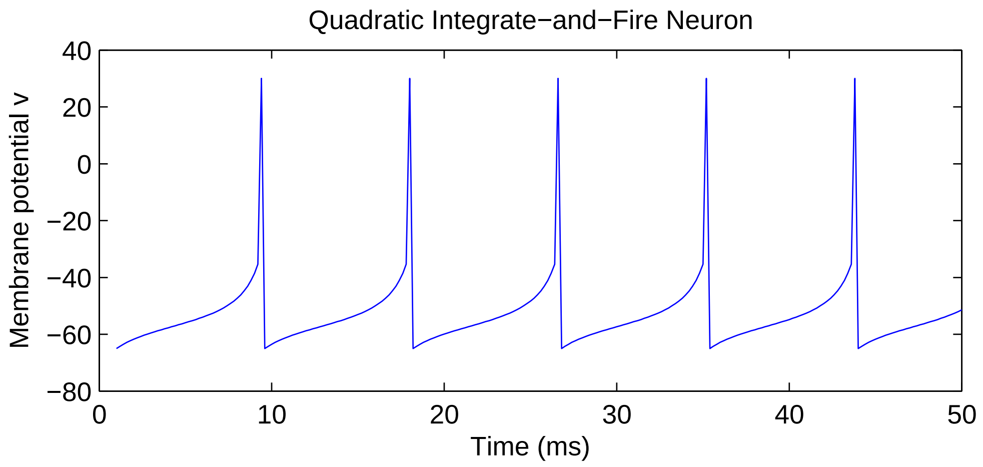


Figure 1.1: Potential of a Quadratic and Fire neuron receiving a constant current

In a network, the input current I_i of the neuron i is equal to:

$$I_i = \sum_{j=1}^{n} w_{ij}\delta_j + \sum_{k=1}^{m} w_{ik} I_k(t)$$

Where:

- w_{ij} : Weight of connection from node j to node i
- δ_j : Binary output of neuron j (0 or 1)
- w_{ik} : Weight of connection from external input k to node i
- $I_k(t)$: Binary external input k (if any)

1.4.2 Izhikevich's model

This model has been introduced by Eugene Izhikevich in 2003. It is more biologically accurate than quadratic and fire neurons and can display a wide repertoire of signalling behaviours.

The membrane potential v and a *recovery potential* u are governed by the following equations:

$$\begin{cases} \frac{dv}{dt} = 0.04v^2 + 5v + 140 - u + I \\ \frac{du}{dt} = a(bv - u) \end{cases}$$

If $v > 30$ then the neuron spikes and potentials are updated according to:

$$\begin{cases} v \leftarrow c \\ u \leftarrow u + d \end{cases}$$

a, b, c, d are four parameters of the neuron.

The higher the recovery potential u, the more difficult the neuron fires. It is thus responsible for the refractory period. Without this variable, the model would be equivalent to the QIF one.

Figure 1.2 shows the evolution of the neuron potentials when submitted to a constant current. The values of parameters $a = 0.02$, $b = 0.2$, $c = -65$, $d = 8$ and $I = 10$ are chosen to imitate the dynamics of a regular excitatory biological neuron.

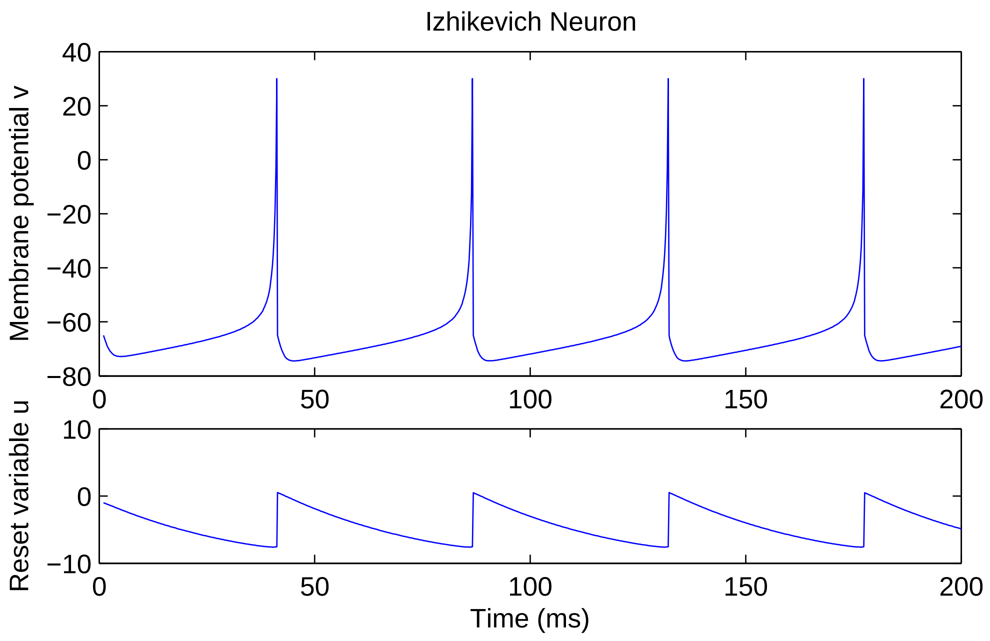


Figure 1.2: Potentials of an Izhikevich neuron receiving a constant current

In a network, the input current I_i of the neuron i is equal to:

$$I_i = \sum_{j=1}^{n} w_{ij}\delta_j + \sum_{k=1}^{m} w_{ik} I_k(t)$$

Where:

- w_{ij} : Weight of connection from node j to node i
- δ_j : Binary output of neuron j (0 or 1)
- w_{ik} : Weight of connection from external input k to node i
- $I_k(t)$: Binary external input k (if any)

1.5 Evolutionary robotics

Evolutionary robotics were originally developed at the beginning of the 1990s and rapidly became a promising tool to build robot controllers without designing them by hand. But soon some limiting problems arose such as the time consumption and the convergence of the algorithms when the complexity of the robots and the tasks scaled up. These reasons led to a slowdown in evolutionary robotics research at the end of the 90s. Currently some researchers are focusing on trying to design some more efficient evolution algorithms in order to faster explore the space of candidate controllers.

The main idea of evolutionary robotics is to evolve a population of robots (i.e. the parameters of their controllers) in order to improve some of them at each generation such that the best robots emerge. Genetic algorithms used to do so are discussed in section 1.6.

Generally, evolution is simulated and resulting optimal parameters are then transferred to real robots. Some studies [8] explain that this transfer can be challenging as simulation must take noise, which exists in the physical world, into account. It is all the more a time consuming process that physical robots are slow. Above all, it is a risky and expensive process as initial random robots can crash or damage their components.

Evolutionary robotics fits particularly well the evolution of neuro-controller and the field of study is called Neuroevolution. Dave Cliff, Philip Husbands and Inman Harvey are some of the precursors in the evolution of neural networks [9]. They showed in 1992 that neural-network control architectures can be evolved to produce efficient visually guided robots. They also showed that such evolved controllers are capable of adaptive behaviours.

One of the most limiting factor of evolutionary robotics it the amount of time that evolution can take to reach good solutions even when it is simulated. New algorithms are regularly released. NEAT is one of the most efficient one for evolving neuro-controllers. It has been developed by Kenneth Stanley in 2002 [10] and evolves both network weights and architecture.

1.6 Genetic algorithms

Genetic algorithms were originally developed in the 1960s by John Holland. It is an optimization algorithm inspired by Darwin species evolution with the mechanism of natural selection. The optimisation problem is to maximize a fitness function. To achieve that, a population of candidate solutions (individuals), defined by some parameters (their genomes), iteratively evolve towards better solutions. The initial population is generally randomly generated. The $(n+1)^{th}$ generation is computed from the n^{th} one according to the following rules:

- Selection of the fittest individuals to reproduce
- Fittest individuals give birth to offspring through crossover and mutation

Figure 1.3 summarizes the main steps of the genetic algorithm used in this project.

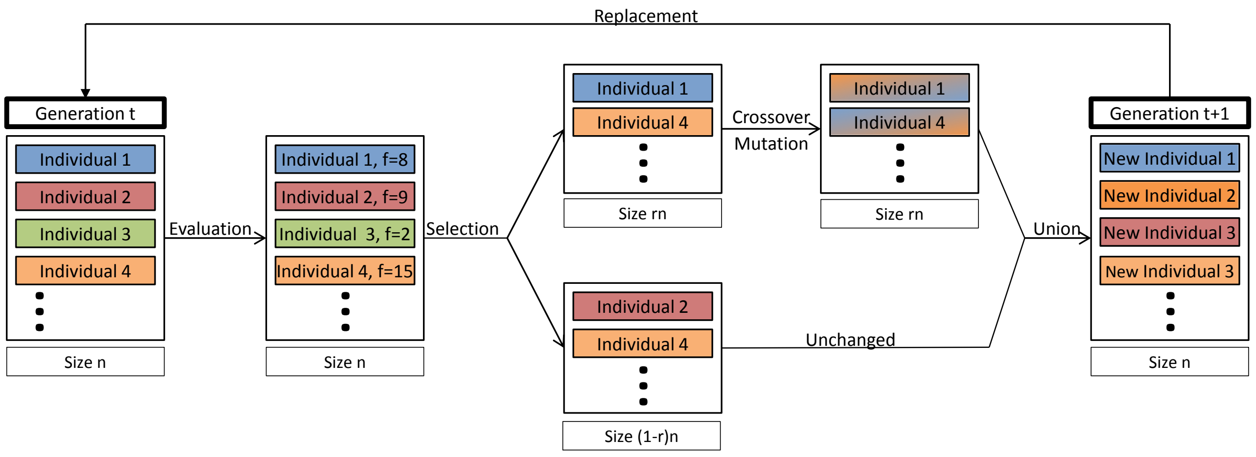


Figure 1.3: Steps of the genetic algorithm.

Crossover consists in merging two individuals whereas mutation is changing a gene of the genome to a random value with a small probability.
Mutations notably permit to escape local minimums and to explore radically new potential solution

spaces. Genetic algorithms can often rapidly localize good solutions.
It is the combination of both crossover and mutation operators that permit successful evolutions.

As genetic algorithms are biologically inspired, it seems a good technique to design some realistic neural networks.

1.7 Evolution of spiking neuro-controllers

Some researchers have successfully evolved spiking neural networks to control robots. Dario Floreano and Claudio Mattiussi [11] built a physical vision-based robot controlled by *Spike Response Model* neurons and capable of navigating in irregularly textured environments without hitting obstacles. Evolution was conducted online on real robots.

Spike Response Model neurons have also been used to control a Khepera robot [12] with a vision-based navigation task where the robot has to follow walls. Evolution was conducted online on real robots.

Some scientists [13] have compared spiking neuro-controllers to conventional artificial neural ones and found that the former are superior in terms of evolutionary success and network simplicity.

This project will try to confirm some of these results by comparing QIF and IZHI networks to CTRNN ones.

1.8 Command neurons

Eytan Ruppin [14] has revealed the presence of command neurons when the neural controlled robot has to switch between several behaviours. He used a kind of CTRNN network to control foraging robots in an environment where some food items were scattered in a corner of the map. He showed that the best robot's evolved strategy is first to look for this corner (*exploration*) and then stay in the food zone in order to eat all the resources (*grazing*). The study demonstrated that this behaviour transformation is due to the activations or deactivations of command neurons that determine the robot's behaviour strategy.

In this project, we will try to see if such neurons can exist in spiking networks.

2 Methodology and description of the experiments

This chapter provides a description of the main elements used in experiments. It introduces some analysis methods as well as indexes to compare the controllers and their evolutions. The last section defines the qualities of a good controller in the context of this project.

2.1 2D world

The simulated robots move in a 2D square world where some food or poison resources can randomly be scattered at the beginning of each simulation. The randomness of this distribution can be modified as explained later in 2.4.

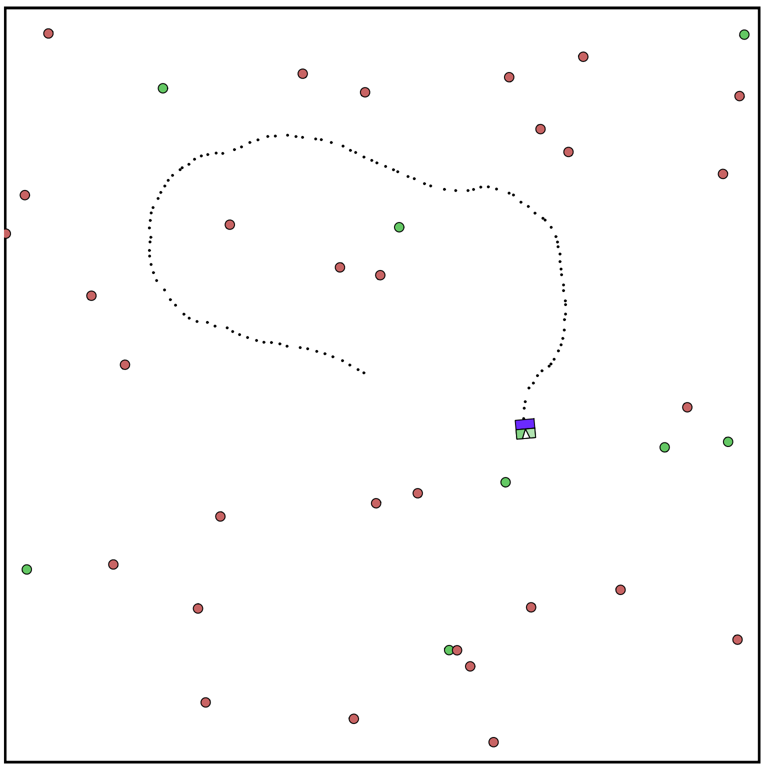


Figure 2.1: Example of a possible 2D world where food items are green and poison items are red. Here the distribution is spatially uniform.

Robots can go through the borders of the 2D world.

Each item has two coordinates x and y. Food items are represented by green balls whereas poison items are represented by red balls.

2.2 Robot design

This project is about simulating robots, not building them. Nevertheless models try to be physically realistic and the transfer of their characteristics to a real robot might be possible.

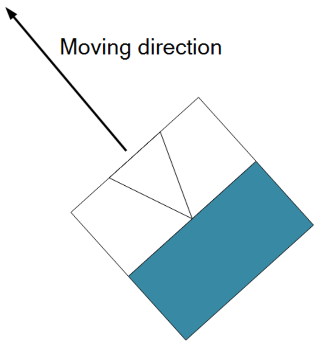


Figure 2.2: Representation of a robot. The bottom color purpose is to identify the robot.

2.2.1 Motors and differential Drive

The robot will move thanks to two motors independently controlling the velocity of their corresponding wheels. By varying the control signals received by the motors, the robot can be steered in any direction. Some iterative relations providing the next coordinates of the robot from the previous ones and the velocities of the wheels need to be computed. To do so, the robot's instantaneous motion is assumed to be circular (see figure 2.3). Let's consider the two positions of the robot at time t and $t + \Delta t$ where Δt is very small.

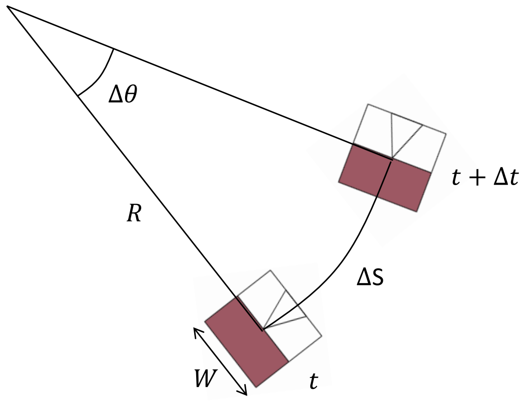


Figure 2.3: The robot's instantaneous motion is assumed to be circular with radius R and distance W between its wheels. ΔS represents the length of the arc of the circle between t and $t + \Delta t$, $\Delta\theta$ is the variation of angle θ between t and $t + \Delta t$.

Let v_L and v_R respectively be the left and right wheel velocities in the world frame at time t and let's assume they are constant between t and $t + \Delta t$. Considering each wheel separately, we then have:

$$\begin{cases} (R - \frac{W}{2})\Delta\theta = v_L \Delta t \\ (R + \frac{W}{2})\Delta\theta = v_R \Delta t \end{cases} \implies \begin{cases} R = \frac{W(v_R+v_L)}{2(v_R-v_L)} \\ \Delta\theta = \frac{(v_R-v_L)}{W}\Delta t \end{cases} \tag{2.1}$$

Now let's consider the motion of the centre of the robot. Let $x(t)$ and $y(t)$ be its cartesian coordinates in the world frame at time t and Δx, Δy their variations between t and $t + \Delta t$. $\frac{(v_L+v_R)}{2}$ is the velocity of the centre of the robot. Figure 2.4 shows the robot's motion between t and $t + \Delta t$.

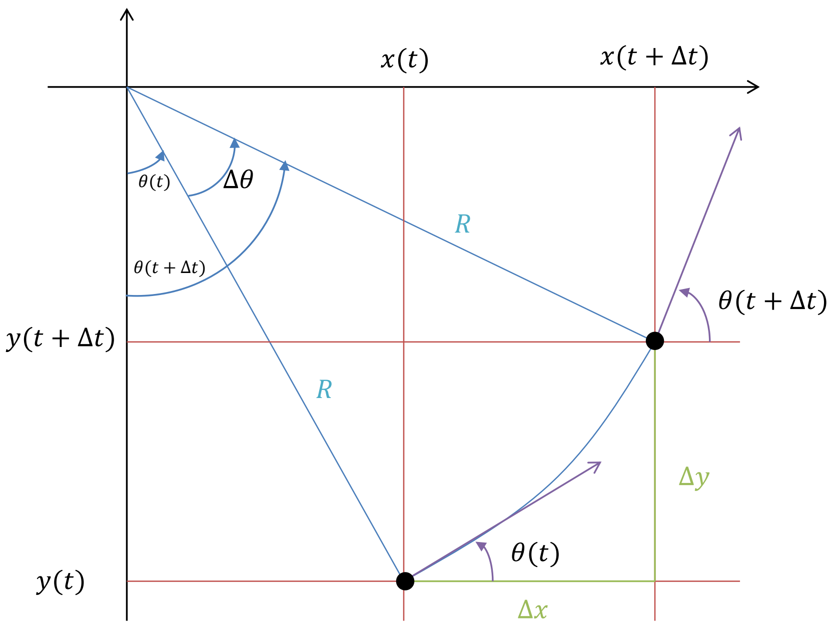


Figure 2.4: We still assume that the robot describes a circle. In addition we assume that $\Delta\theta$ is small

From figure 2.4, we have:

$$\begin{cases} \Delta x = R(sin(\theta(t+\Delta t)) - sin(\theta(t))) \\ \Delta y = -R(cos(\theta(t+\Delta t)) - cos(\theta(t))) \end{cases} \tag{2.2}$$

As we assume that $\Delta\theta$ is small, we can only keep the first order of sinus and cosinus Taylor series:

$$\begin{cases} sin(\theta(t+\Delta t)) = sin(\theta(t)+\Delta\theta) \simeq sin(\theta(t)) + cos(\theta(t))\Delta\theta \\ cos(\theta(t+\Delta t)) = cos(\theta(t)+\Delta\theta) \simeq cos(\theta(t)) - sin(\theta(t))\Delta\theta \end{cases} \tag{2.3}$$

The following iterative equations follow from the reduction of equations 2.1, 2.1 and 2.3:

$$\begin{cases} \theta(t+\Delta t) = \theta(t) + maxRot \times (v_R - v_L) \\ x(t+\Delta t) = x(t) + cos(\theta(t))\frac{(v_L+v_R)}{2} \\ y(t+\Delta t) = y(t) + sin(\theta(t))\frac{(v_L+v_R)}{2} \end{cases}$$

Where $maxRot$ is a constant determining the maximum angle the robot can rotate between two successive positions. Motor intensities are linearly converted into the wheels velocities.

2.2.2 Sensors

Sensors will let the robot probe its nearby environment.

The main combination of sensors used in this project is composed of:

- three distance sensors to detect **food items** (left/middle/right sides). Each sensor only detects the closest item in its scope. The closer the item, the higher the signal generated by the sensor.
- three distance similar sensors to detect **poison items**.

In the last experiment 6.1, another sensor will be introduced, returning 1 if the last eaten item of the robot is a poison item and zero if it is a food item. It is a proprioceptive sensor.

Each sensor returns a scalar measured value between 0 and 1. Distance sensors has a limit scope which is approximatively seven times the width of the robot.

Figure 2.5: Example of a robot having three distance sensors to probe food items: sensor 1 is the left food item sensor, 2 the middle one and 3 the right one. The higher the measured value of the sensor is, the darker the green fill color is. Inactive sensors are white. Poison sensors are not represented.

2.3 Neural controller

A fully recurrent neural network is used to control the robot.
The sensors described previously in 2.2.2 transmit their measures to the inputs of the controller whereas the robot's motors (see 2.2.1) are controlled by the outputs of the controller. Therefore the controller has six inputs and two outputs. Figure 2.6 shows the principal architecture of the controller used in this project. A few other architectures with multiple layers have also been tested.

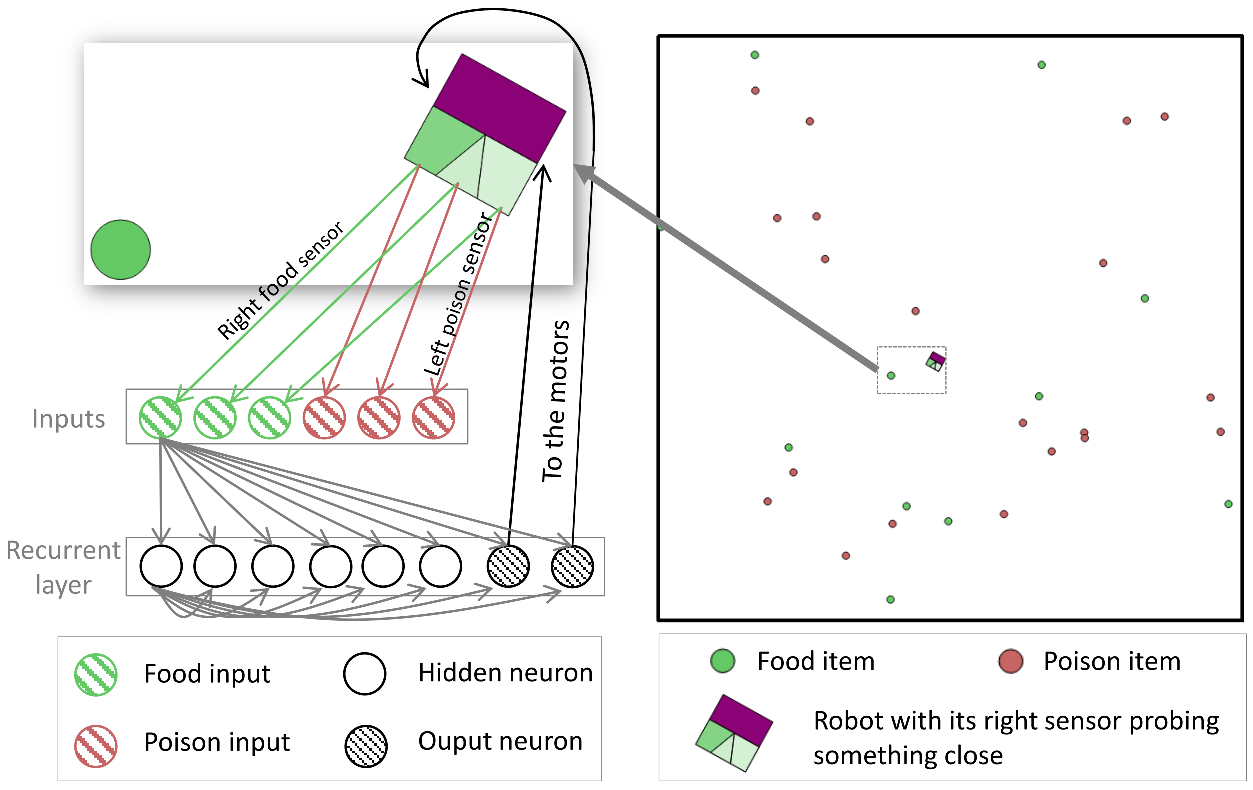


Figure 2.6: Fully recurrent neural network controller and its corresponding inputs and outputs

A fully recurrent network is represented by its weights matrix $(w_{ij})_{i<n,j<m}$. w_{ij} is the weight from neuron/input j to neuron i. Typically $m > n$ because inputs are connected to the rest of the network but not the contrary. In that case, the controller has only one layer of neurons.

2.3.1 Neuron models

Three models of neurons will be compared in this report:

- Continuous Time Recurrent model (CTRNN),
- Quadratic Integrate and Fire model (QIF),
- Izhikevich's model (IZHI).

The spiking neurons' potentials are reset if their values go below -80.
CTRNN potentials cannot exceed 20 or -20.

2.3.2 Inputs processing

CTRNN networks

Sensors inputs (measured values between 0 and 1) are sent to the neurons after being weighted.

Spiking networks

At each iteration, each sensor returns a scalar value between 0 and 1 that is converted or not in a spike. To do so, a random number k between 0 and 1 is generated. If $k < sensorValue$ then a spike is emitted to all the other neurons (after being weighted).

Some noisy background current is sent to a specific hidden neuron (not one of the motor neurons). We will refer to this neuron as the *noisy neuron*. Actually this background current's main interest is at the very beginning of the simulation when the neural network needs to be "switched on" (see 2.3.4). It triggers some activity in the network which, in the case of good evolved robots, becomes self-sustained.
In some cases, this background current, when maintained all along the simulation, can enable the robot to have interesting exploration strategies as it will be shown later in the experiments.

2.3.3 Outputs processing

CTRNN networks

The potentials of the motor neurons are linearly converted to the motor speeds.

Spiking networks

For each motor neuron, a mean firing rate is computed over 100 network cycles and is then linearly converted in motors speeds.

2.3.4 "Switch on the network"

A the beginning of each simulation, spiking networks need some inputs to trigger some activity in the network. Unfortunately at the initial position of the robot, sensors might not detect any items. Therefore some noise inputs need to be generated in order to "switch on" the network.

In this project, noise input is created by sending some random background current to a specific hidden neuron of the network which will be referred to as *the noisy neuron*.

A few other alternatives had been investigated such as:

- Setting the initial potentials with random values, instead of neurons resting values.

- Sending some random background current to all the neurons.

The advantage of having only one *noisy neuron* is that evolution can customize the level of noise other neurons receive from this *noisy neuron.* If we had some random background current for all neurons, this would compel all neurons to receive some random noise and it would potentially affect the dynamics of the network and therefore the behaviour of the robots. Imagine a robot describing a memorized pattern when suddenly a motor neuron receives a background current that completely disturbs its dynamics and thus the memorized pattern.

2.4 Resources distribution

The resources are the food and poison balls. Their distribution is something crucial that determines the complexity of the robot's behaviour.

Different spatial distributions has been tested in this project:

- uniform distribution of both food and poison items. We expect the robot to be only reactive to its inputs, by moving towards food items and avoiding poison items.
- uniform distribution of the poison items and grouping food items in uniformly distributed locations. We expect the evolved robot to switch between an exploration phase where the robot tries to find a group of food and a grazing phase where the robot stays in the food zone until all food items are eaten.
- deterministic distribution of food and poison items. We expect the robot to learn the pattern of locations of the items.

2.5 Genetic algorithm

An initial population of 100 robots is randomly generated. At each iteration of the algorithm, a simulation is processed for each robot (one robot at a time in the 2D-world) and a fitness function is evaluated at the end of each of these individual simulations. This fitness function measures the performance of the robot for the task assigned. Best robots according to their fitness value are randomly selected to reproduce and give birth to the next generation.

For clarity, these are the main terms used in the rest of the report:

- a simulation: This is actually the phase when a robot's fitness is evaluated. It represents a ride in the 2D-world and lasts for a fixed amount of time defined by a constant variable (usually 500 robot's steps).
- an evolution: It represents the whole genetic evolution of a population of robots. Several evolutions usually run at the same time in this project in order to have more accurate results.
- a generation: The term can refer to an iteration of the genetic algorithm or its population at a specific iteration. Generation$_i$ is the evolution population after i iterations of the genetic algorithm.

The genome of each individual is composed of the weights of its neural network. For some specific simulations, some neurons parameters such as the memory factors, refractory periods or delays can also be included. A default genome is composed of:

- weights of the networks with values ranging from -1 to 1 (with a scaling factor, usually 5).
- delays of signal transitions between neurons with values ranging from 1 to 10.
- for QIF neurons, refractory periods with values ranging from 5 to 10.
- for CTRNN neurons, time constants with values ranging from 1 to 50.

- for CTRNN neurons, bias with values ranging from -10 to 10.

A new generation is composed of:

- half individuals from the previous one (genome unchanged).
- half individuals resulting from crossovers and mutations.

Selection is done with a roulette-wheel technique. First individuals are sorted by their fitness values. Then a random number k between 0 and the total sum of all the individuals fitness values is uniformly generated. It indicates which individual to choose as explained on figure 2.7.

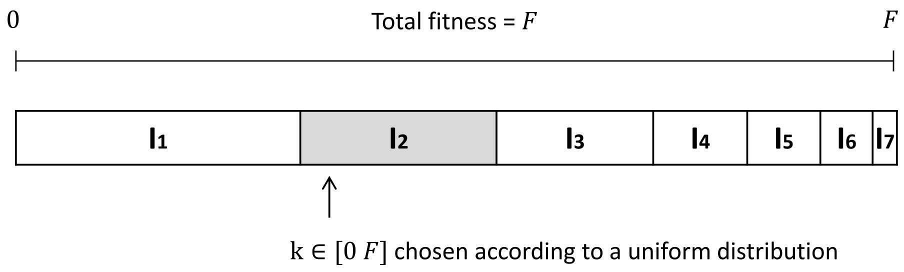


Figure 2.7: Roulette-wheel technique.

The resulting probability of selection of individual i is therefore $p_i = \frac{f_i}{\Sigma_{j=1}^{N} f_j}$ where f_i is the fitness of individual i in the population. The roulette-wheel technique works only if fitness values are positive. In this project, the values might be negative, therefore the lowest fitness of the population is subtracted to all other individual fitnesses for the roulette-wheel technique.

Mutations occur on a gene with a probability of 5%. This choice is the result of simulations based on the scenario described in section 4.1. Figure 2.8 shows the importance of a good mutation rate value.

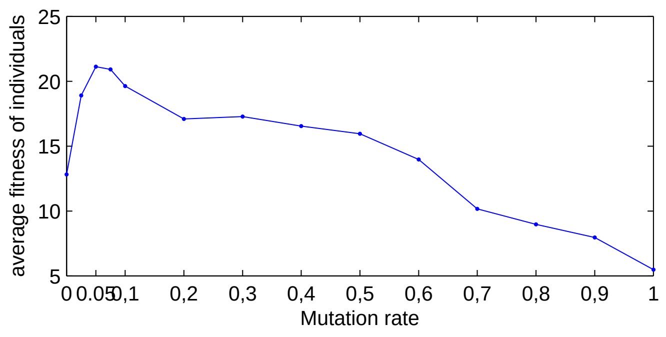


Figure 2.8: Mean fitness in function of the mutation rate. For each rate, ten evolutions have been averaged to have more accurate results.

If the mutation rate is too low, only the crossover operation will increase the population fitness. If the mutation rate is too high, the new individuals are totally random and results are very poor.

Crossover operation is done with a barycentric method: the child genome is the barycentre of its two parents. More precisely, a random number in the interval $k \in [-0.5\ 1.5]$ is chosen and the child genome is given by the following formula: $genome_{child} = k \times genome_{parent1} + (1-k) \times genome_{parent2}$. One-point and two-point crossovers have been implemented and tested but the results were slightly worse compared to the barycentric method.

Figure 2.9 is an example of the way an evolution is summarized in the rest of this report. The blue curve is the mean fitness over the best half of the population at each generation.

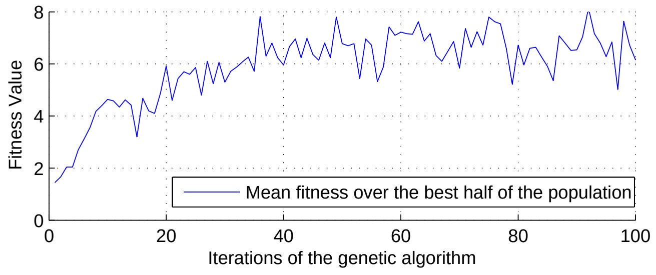


Figure 2.9: Mean fitness of an evolution

The reason for taking only the best half of the population instead the whole population is because the worst half is characterised by a high fitness variance level. Actually most of the mutated and crossed individuals do not perform well the task and are in the second half whereas individuals which do not change across the generations are usually among the best.

Evolution's speed and convergence can heavily depend on the initial population. There are at least three ways to avoid this problem:

- Have a very large initial population. Evolutions take more time.
- Run several independent evolutions. These evolutions can run on different computers.
- Use more sophisticated algorithms to escape local minima, such as using simulated annealing or using adaptive mutation and crossover rates.

The second solution is adopted in this project as the Condor system 3.5 is used to parallelize the evolutions. Therefore, experiments of this project process 50 evolutions at each time and return four curves as shown on figure 2.10.

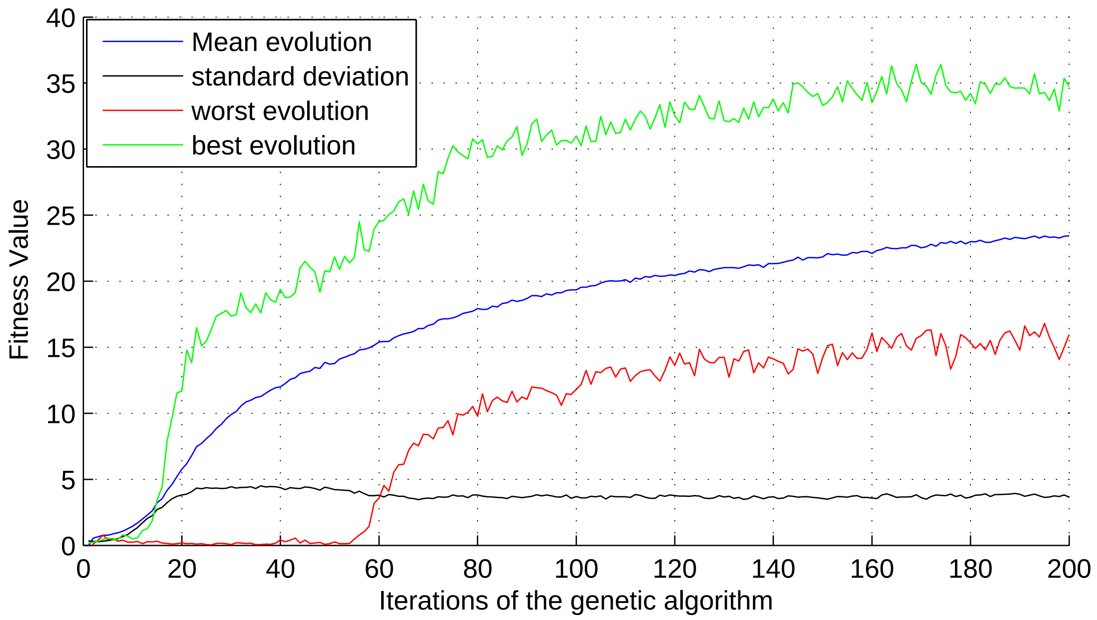


Figure 2.10: Type of plots used in this project to present evolution's experiments results. On this plot, 50 evolutions with the same initial settings but different initial populations are summarized

The blue curve called *Mean evolution* is actually the mean mean fitness of the 50 evolutions. For each evolution, the mean fitness is computed over the population so we have 50 mean fitness curves. By taking the mean of these curves, the blue curve is obtained.
Similarly the green curve should rather be called the *mean fitness of the best evolution* over the 50 evolutions.

By convenience, the blue curve is sometimes called the *mean fitness* of the evolution in the rest of the report.

2.5.1 Selection of the best controller/robot

After running a set of evolutions such as in 2.10, it is useful to select the best evolved robot. To do so, the best evolution is first selected (the green curve) and then the best individual of its population needs to be found. To achieve that, five simulations are run for every individual corresponding robot and a mean fitness is averaged over these five simulations for each individual. The best individual is the one with the highest mean fitness value. Nevertheless, due to the high variability of initial environments of the simulations, the "selected robot" might not always be the "real best robot".

2.5.2 End of an evolution

This paragraph is about how to decide when to stop an evolution.
It is quite hard to find good terminal conditions to stop the genetic algorithm as we do not know the maximal possible fitness (which can potentially be infinite in some of these project's experiments). Therefore evolutions are usually stopped when "they seem to have finished". In other words, evolutions of this project are run during a long time to reach an apparent stability of the maximal fitness value obtained.

Note that some evolutions may be stuck in local minima which sometimes require a lot of simultaneous mutations to escape, which can be a very unlikely event.

2.5.3 No safe comparison based on fitness values between different models

The fitness values between two robots controlled by controllers using different neuron models can not be compared safely because the fitness highly depends on the velocities of the resulting robots. These velocities specifically depend on the neuron models and the way networks' outputs are converted into the wheels' velocities differ from spiking networks and CTRNN networks. For example a spiking network with a low refractory period (low maximal mean firing rate) is likely to be slower than a high refractory period network (high maximal mean firing rate).

2.5.4 Evolution speed index

The evolution speed index used in this project is intended to measure the average convergence speed of an evolution. It is defined as the ratio between the mean fitness after 50 generations and the mean fitness after 200 generations. For a set of evolutions, as on figure 2.10, the speed index is calculated on the *Mean volution* curve. Figure 2.10 speed index is then equal to $13.5/23.5 = 0.57$. The faster the evolution, the higher the index. This index permits to compare the speed of different evolution experiments that do not necessarily have the same scale fitness values.

2.5.5 Deviation index

Evolutions can be really dependant on the initial population of individuals and this might be a problem. Therefore a *deviation index* is introduced to measure the variance among evolutions. Figure 2.11 shows five evolutions mean fitnesses, the only difference being the initial population.

The *deviation index* is defined as the ratio between the standard deviation and the mean evolution fitness at a specific generation. For evolutions described on figure 2.10, at generation 200, the deviation index is $3.65/23.4 = 0.15$. The smaller the index is, the less variable evolutions are for the corresponding experiment.

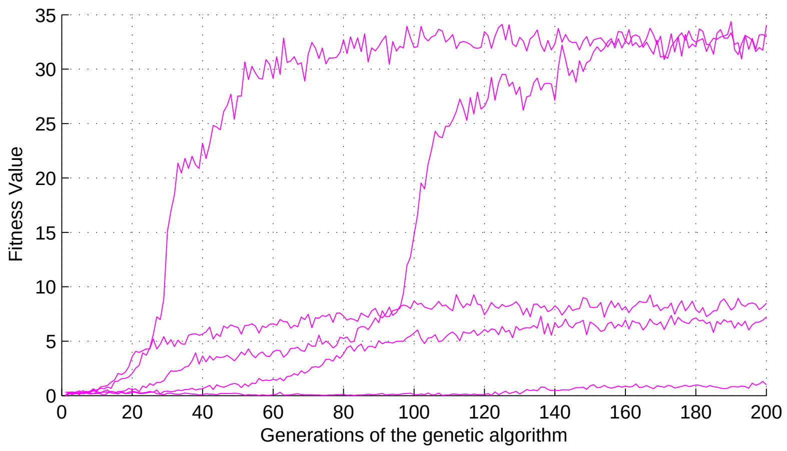


Figure 2.11: Five evolutions of same controller model with different initial population can have really different mean fitnesses.

The deviation index doesn't depend on the actual value of the fitness and therefore can be used to compare different evolution experiments. In the rest of the project, it is used to compare the evolutions of different controllers in diverse environments and with different tasks.

2.6 Lesion experiments

In order to figure out the importance of each connection in a network, we can set one by one each corresponding weight to zero and observe how the controlled robot behaves differently by measuring its new fitness during a simulation. In the rest of the report, this procedure is named a *lesion experiment* and a *lesion* is the single action consisting in setting one weight to zero. These experiments will be conducted on the best evolved controllers.

Lesion experiment results are displayed in a 2D array having the size of the weights' matrix of the network. Each $cell_{ij}$ value represents the averaged fitness value of the controlled robot with the corresponding $weight_{ij}$ set to zero. The averaging operation is necessary to reduce the variance as robot's environments are generated pseudo-randomly. In the rest of the project, each fitness value is averaged over 200 simulations. Figure 2.12 gives an example matrix of an *lesion experiment* results.

	1	2	3	1*	2*	3*	4*	5*	6*
1	−3	31	31	29	30	29	32	31	30
2	1	23	30	31	−1	4	31	27	15
3	31	30	31	30	31	31	31	31	30

Figure 2.12: Example of a lesion matrix of a neural controller having three hidden neurons and 6 inputs

Let's call F the mean fitness value of the controlled robot when there are no lesion at all. In these *Lesion experiments*, we are not interested in the new fitness value but rather in its comparison with F. Black-background cells are those which value is below one third of F. These black cells indicate that the lesion has a dramatic impact on the efficiency of the corresponding controlled robot.
White-background cells are those which value is above 9/10 of F. These white cells mean that the lesions have no effect on the efficiency of the corresponding controlled robot.
Grey cells such as $cell_{2,6^*}$ indicate that the lesions have a significant but limited impact on the efficiency of the corresponding robot controller.

The results permit to visualize how the controller relies on a large number of neurons and

connections or if only of small part of the network is used.
This experiment is obviously not exhaustive. We can indeed think about a controller with two lesions that do not have any effect when they occur separately but impact the controller's behaviour when combined.

2.7 Characteristics of a good controller

Aims of the project's experiments are not only to produce good controllers but also to see how they can reliably be produced.

A good evolving controller has the following qualities:

- it performs well the task it has been evolved for: its fitness needs to be high.
- a **high** speed index (refer to 2.5.5). It means that its convergence will be fast (on average) and won't need many generations to return good controllers.
- a **low** deviation index (refer to 2.5.4). It means that its evolution is unlikely to fall in local minima and that evolutions with the same initial settings will converge towards similar controllers. This is particularly important for engineering/industrial applications.

A quality that is not essential except in some particular situations is to have a lesion matrix 2.6 with as few black cells as possible. It means that the network is robust enough to face some lesion damages up to a certain extent. This can happen in electrical spiking circuits for example.

3 | Simulator design

The robot simulator is a software intended to evolve some populations of controllers and to observe their corresponding evolved robots.

This chapter describes the implementation details of the software that was developed to evolve and simulate the controlled animats. This software design chapter can be skipped. However a summary section is available at its end.

3.1 Requirements

Evolutionary robotics and neural networks can be really time consuming. The software design had therefore to be as efficient as possible. In order to run optimized experiments, the graphical interface had to be completely separated from the processing one. In the following parts they will be referred as the **graphical** part and the **processing** part.
For processing simulations the Condor system appeared to be a useful tool. Condor is a high-throughput batch-processing system allowing to use idle computers of the department of computing. This system would permit to run in parallel more than two hundreds evolutions.

As different neural networks with different neural models were going to be tested on different experiments and environments, the object-orientated program had to be as generic as possible.

C++ was chosen as one of the fastest programming language with the Qt graphical libraries. The genetic algorithms and neural networks have been manually implemented.

3.2 Processing part architecture

The final program is highly object oriented and its processing design can be visualized with a UML diagram displayed on figure 3.1.

Three parts can be distinguished:

- the neural network part in green,
- the simulation part in blue,
- the evolution part in red.

3.2.1 Neural network part

This part is composed of two main classes: `NeuralNet` and `Layer`.
`Layer` manages one layer of neurons.
`NeuralNet` manages a neural network architecture.

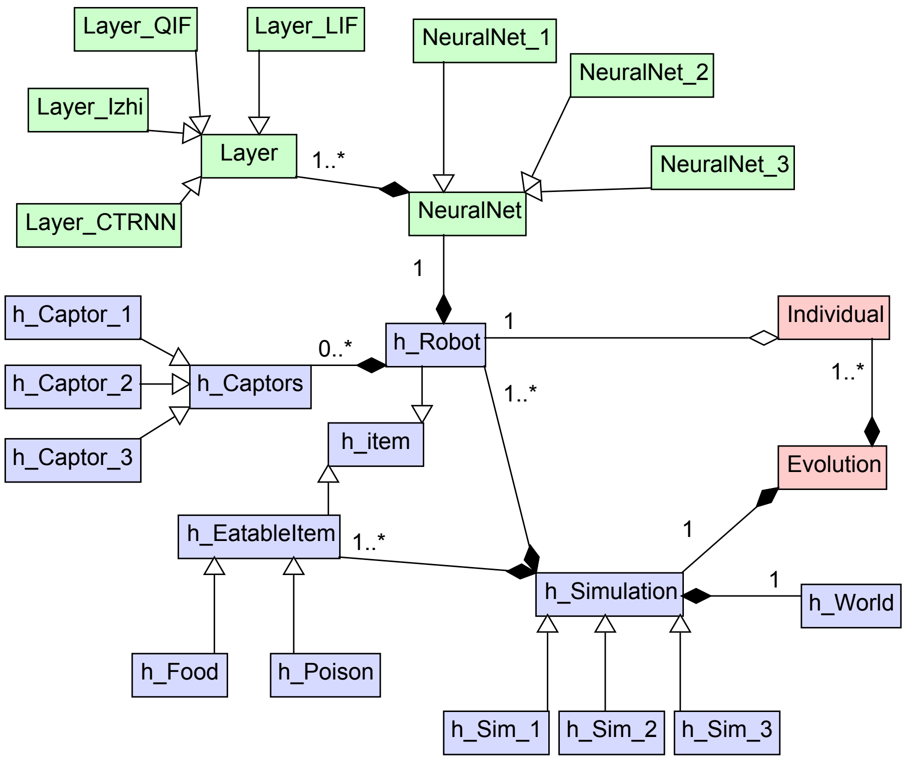


Figure 3.1: UML diagram of the processing part.

`Layer`

This class manages the neurons, their parameters and their potentials.
Each subclass defines a new type of neuron. There are for example the following subclasses `Layer_QIF`, `Layer_IZHI`, `Layer_CTR` which are the three neuron models used in the experiments of this project. But other neuron models have been implemented such as Integrate and Fire neurons and other non-realistic versions of spiking neurons.

`Layer` and its subclasses store the parameters and the state of a layer of neurons:

- the number of neurons in the layer,
- the parameters/constants of the neurons (each neuron can have different parameters),
- the state variables of the neurons: potentials, recovery variables, internal, external currents.

`NeuralNet`

`NeuralNet` is in charge of the architecture of the neural network.
By inheriting this class, networks with arbitrary layer-based architectures can be designed by only re-implementing two virtual functions: one responsible of initializing the network structures and the other one to compute the currents.

`NeuralNet` stores a vector of `Layer` objects:

```
QVector< Layer* > layers;
```

Each layer of the neural network can be connected to itself or to any other one. The relation between two layers is represented by a matrix of weights. The class has therefore a vector of matrices of weights as member field:

```
QVector< QVector< QVector<qreal> >* > delays;
```

Each weight matrix has its corresponding delays matrix:

```
QVector< QVector< QVector<int> >* > delays;
```

`NeuralNet` is responsible for the computation of the currents received by each neuron. This is done in the virtual function *computeCurrents* which is re-implemented in each child class. It is in this function that the weights are used and actually that the network architecture is completely defined.

3.2.2 Simulation part

This part deals with all the elements of a simulation.

`h_World`

This class represents the 2D world. Its primary role is simply to store the world's size.

`h_Item`

`h_Item` is the base class for all inputs that can be added to the world. It stores the coordinates of the item.

`h_Robot` is the base class for all the different kinds of robots used in the experiments. Each specific robot is a child class of this base class. Two children classes differ in their sensors.

`h_EatableItem` is the base class for food and poison items

`h_Simulation`

This class manages all the items composing a simulation: world, robots, food/poison items. It contains some lists of these elements.

It is also in charge of the distribution of food/poison items. A child class of`h_Simulation` has to re-implement the following virtual pure member functions:

```
virtual void resetRobot(h_Robot* robot) = 0;
virtual void resetFood(h_EatableItem *food) = 0;
virtual void resetPoison(h_EatableItem *poison) = 0;
```

Re-implementing these functions allow to choose where items are initialized and how they are reset when consumed.

3.2.3 Evolution part

The evolution is composed of two classes: `Evolution` and `Individual`. These classes manage the evolution of the robots.

`Evolution` is a generic class that permits to evolve a population of `Individual` objects. It implements all the genetic algorithms steps. Different selection and crossover methods have been implemented. It doesn't apply specifically to robot's evolution and can potentially for many other problems.
Each `Individual` object stores the genome of the represented individual. Each `Individual` is linked to a `h_Robot` object as the genome is composed of parameters of the robot's controller. The genotype of `Individual` is highly customizable in the sense that it can store various genes from different types and from different ranges of values.

3.3 Graphical part architecture

A Model-view design pattern has been implemented in order to easily separate the view from the model part. Figure 3.2 shows the UML diagram of the graphical part.

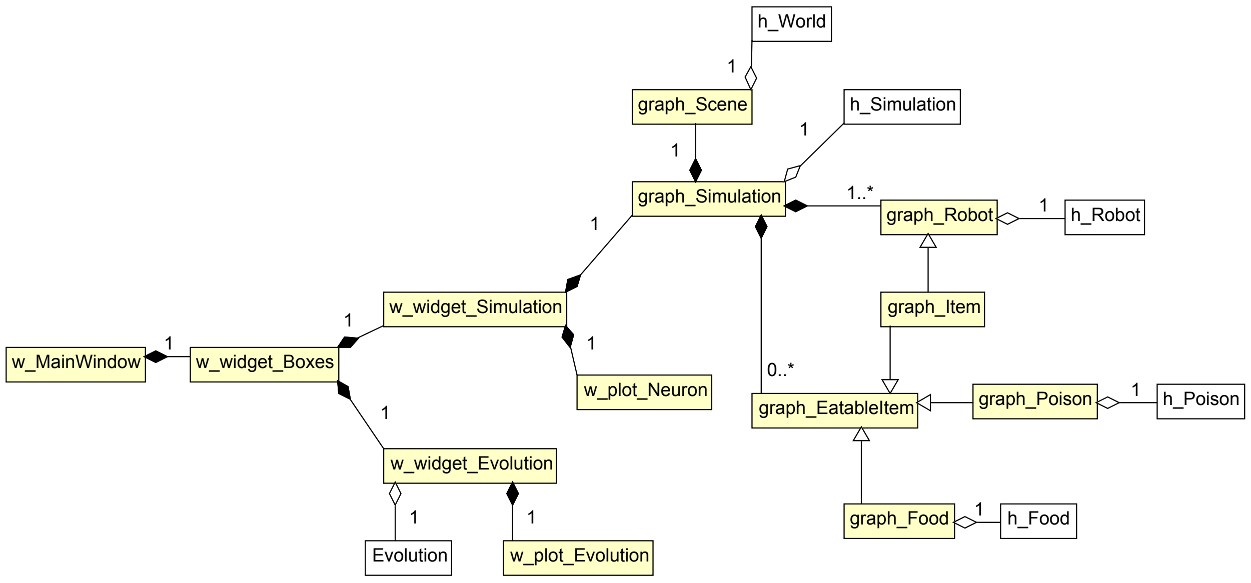


Figure 3.2: UML diagram of the graphical part.

The yellow-background classes are the graphical classes whereas the white-background classes are the corresponding processing ones.

3.3.1 Main window

`w_MainWindow` is the main class of the graphical part. It manages all the useful components for graphical experiments.
It is composed of the class `w_widget_Boxes` that is a *tab widget* with two tabs. The first one permits to manage simulations (class `w_widget_Simulation`) and the second one permits to manage the evolutions (class `w_widget_Evolution`).

3.3.2 Simulation tab

The simulation tab is divided in two sections: the world view and the neurons' plots (see 3.3).

the world view

A view of the 2D environment map is displayed and has several useful functionalities to make it convenient to use:

- the world is scrollable/zoomable when it doesn't fit the view. Zoom is done by using the wheel button of the mouse and pressing the control key. It is also possible to drag the map with the mouse cursor instead of scrolling.
- It is possible to change the speed of the simulation by modifying the time laps between two steps of the robots. By default, it is 100msec + the neural computations. This is done thanks to the top left corner slider and spinbox. A Pause/Resume button is also available.
- Below the view the length of the black trail that the robot leaves when it moves can be adjusted.

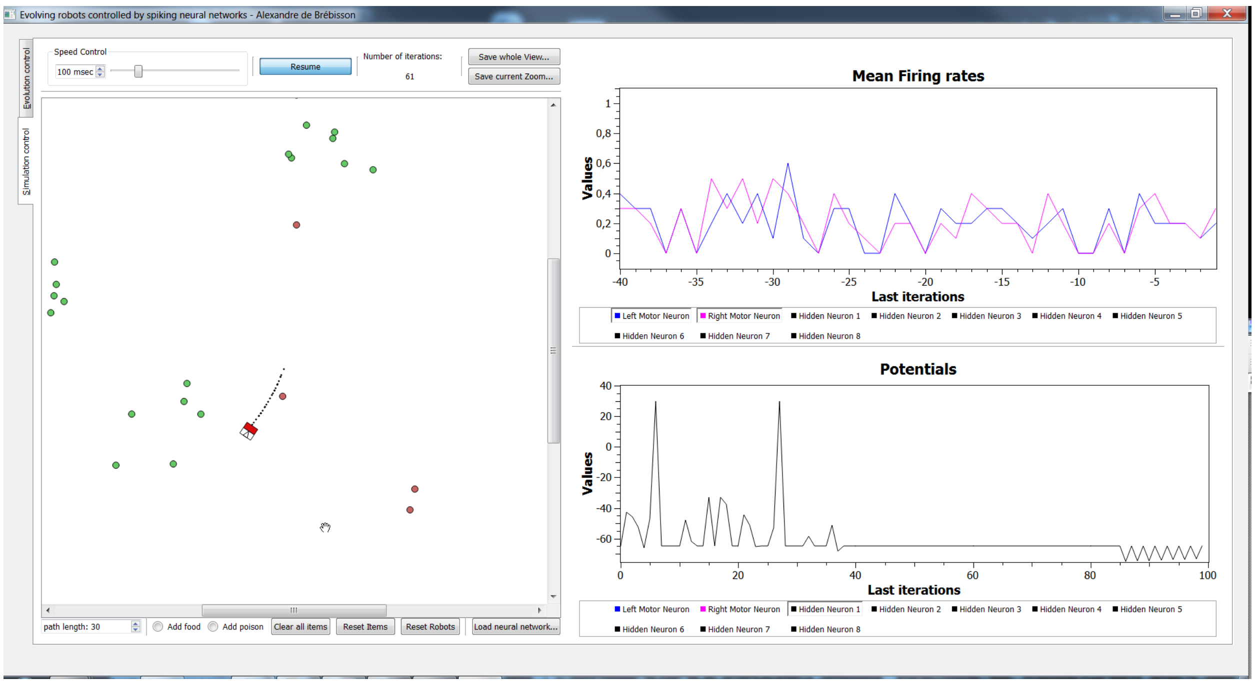


Figure 3.3: Simulation tab with all its functionalities. The controller is a spiking neural network.

- Still below the view, a few controls enable the user to manage the food/poison resources by adding/removing them with the mouse at the pointed locations on the world. It is also possible to clear all items. This last functionality is useful to visualize how the robots react without any external inputs. The distribution can also be reset.
- With the button *Load neural network...*, it is also possible to load some specific neural controllers into the simulation. This is particularly useful in order to store and test good evolved controllers.
- At the top right corner, two buttons enable to take some high quality snapshots of the simulation view.

The view uses anti-aliasing to soften the shapes and only modified zones between two simulation iterations are updated (static resources won't be unnecessarily updated).

the neurons' plots

The plots on the right of the window 3.3 depend on the neuron model used in the robots' controllers. If they are spiking neurons, then potentials and mean firing rates are plotted. If they are CTRNN neurons, only potentials are plot. It is possible to deactivate/active some specific curves by clicking on their legends. On figure 3.3, only left and right motor neurons' mean firing rates are displayed and only "Hidden Neuron 1" potential is displayed.

3.3.3 Evolution tab

The evolution tab is where the evolution of the robots is managed. A button and a spinbox permit to add as many generations as the user want. A progress bar indicates the progress of the evolution.

The window is composed of a main graph that represents the mean individual fitness over the generations in 3.4. The graph updates in real-time and its axis scale automatically.

On the bottom right corner, three buttons are available:

- "Save Graph..." generates a vectorial picture of the graph

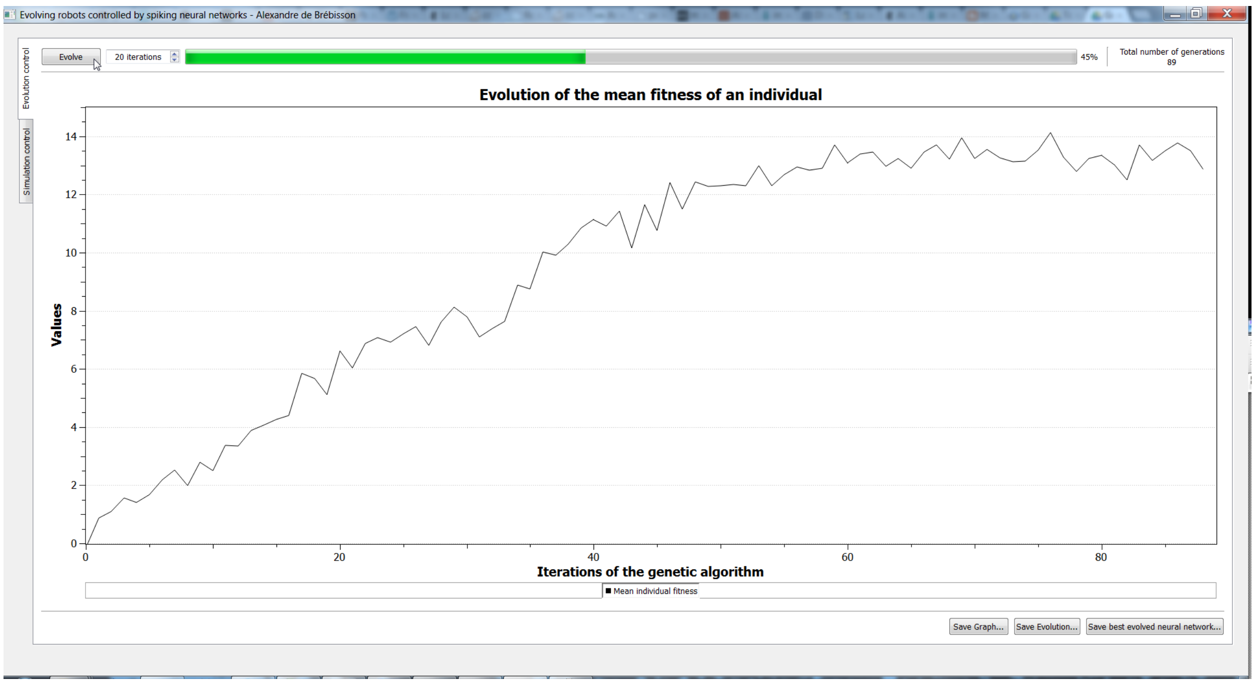

Evolving robots controlled by spiking neural networks - Alexandre de Brébisson
Evolution control
Simulation control
Evolve
20 iterations
45%
Total number of generations
89
Evolution of the mean fitness of an individual
Values
Iterations of the genetic algorithm
■ Mean individual fitness
Save Graph...
Save Evolution...
Save best evolved neural network...

Figure 3.4: Evolution tab with all its functionalities

- "Save Evolution..." permits to save an array representing the graph
- "Save best evolved neural network..." permits to save the best evolved neural network returned by the evolution in a special file format .net

3.4 Genericness of the code

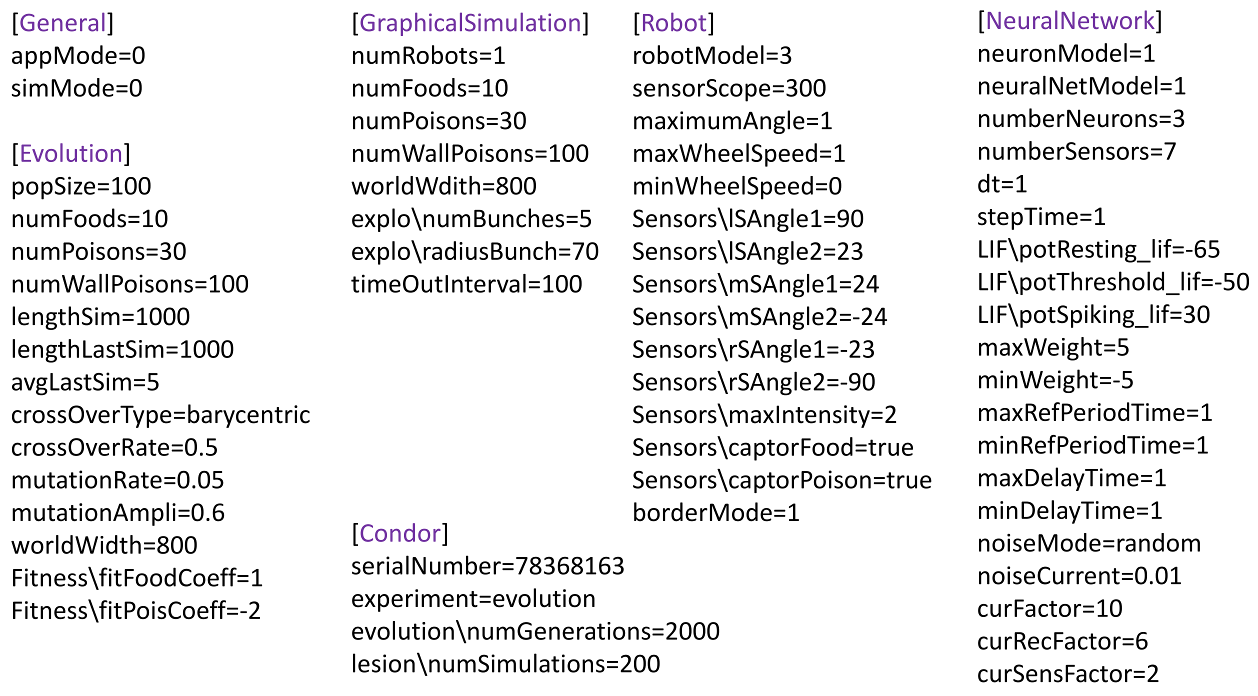
The design of the software is flexible enough to easily create different experiments. It is indeed particularly convenient to add:

- new types of robots,
- new items distribution,
- new types of neurons,
- new types of neural networks with new architectures,
- new types of sensors,
- new genes for the genetic algorithm.

In order to manage the important quantity of parameters in the simulator, a `Settings` class storing all the parameters/constants of the software is used. Every time the simulator is launched, the `Settings` object updates itself from a ".ini" file and every time the simulator is closed, the `Settings` object saves the parameters/constants in the ".ini" file. This technique allows the user to easily change the parameters of the simulation without having to re-compile the code again. The ".ini" file describes the scenario of an experiment. Figure 3.5 show all the parameters stored in the ".ini" file.

`Settings` also implements a *Factory* design pattern to create all the possible versions of the listed items in 3.4:

```
h_Simulation* createSim(...) const;
NeuralNet* createNeuralNet(...) const;
```



```
[General]
appMode=0
simMode=0

[Evolution]
popSize=100
numFoods=10
numPoisons=30
numWallPoisons=100
lengthSim=1000
lengthLastSim=1000
avgLastSim=5
crossOverType=barycentric
crossOverRate=0.5
mutationRate=0.05
mutationAmpli=0.6
worldWidth=800
Fitness\fitFoodCoeff=1
Fitness\fitPoisCoeff=-2

[GraphicalSimulation]
numRobots=1
numFoods=10
numPoisons=30
numWallPoisons=100
worldWdith=800
explo\numBunches=5
explo\radiusBunch=70
timeOutInterval=100

[Condor]
serialNumber=78368163
experiment=evolution
evolution\numGenerations=2000
lesion\numSimulations=200

[Robot]
robotModel=3
sensorScope=300
maximumAngle=1
maxWheelSpeed=1
minWheelSpeed=0
Sensors\lSAngle1=90
Sensors\lSAngle2=23
Sensors\mSAngle1=24
Sensors\mSAngle2=-24
Sensors\rSAngle1=-23
Sensors\rSAngle2=-90
Sensors\maxIntensity=2
Sensors\captorFood=true
Sensors\captorPoison=true
borderMode=1

[NeuralNetwork]
neuronModel=1
neuralNetModel=1
numberNeurons=3
numberSensors=7
dt=1
stepTime=1
LIF\potResting_lif=-65
LIF\potThreshold_lif=-50
LIF\potSpiking_lif=30
maxWeight=5
minWeight=-5
maxRefPeriodTime=1
minRefPeriodTime=1
maxDelayTime=1
minDelayTime=1
noiseMode=random
noiseCurrent=0.01
curFactor=10
curRecFactor=6
curSensFactor=2
```

Figure 3.5: All the parameters of the program that can be changed without compiling. They represent an experiment scenario.

```
Layer_Spiking* createLayer(...) const;
h_Robot* createRobot(...);
graph_Robot* createGraphRobot(...);
```

3.5 The Condor system

Condor is a high-throughput batch-processing system allowing to use idle computers of the department of computing.

Two reasons explain why this system has been particularly useful in this project:

- The evolution process can take a long time as neural networks are computationally expensive.
- Evolutions can heavily depend on the initial population which is randomly generated. Thus a huge number of evolutions must be averaged.

The Condor system permits to run in parallel more than fifty evolutions. Final results are then averaged.

3.6 Summary

The resulting program has almost 10,000 lines (10% of comments) and more than 70 classes.

It is highly flexible in order easily change simulation scenarios. It is indeed easy to change:

- robots and their sensors,
- neural networks and their architectures,
- neuron models and their parameters,
- resources distributions in the 2D fields,
- genome variables and evolution parameters,
- tasks robots has to solve,
- evolution algorithms such as the crossover technique.

To change the configuration, no compilation is required, instead parameters can be modified in a .ini file that has 70 fields. In particular, there is a boolean field specifying whether the simulator has to be launched in a graphical or processing mode. All experiments of this project are configured only by modifying the ".ini" file.

In processing mode, up to 200 idle computers of the computing department can be used to run evolutions and experiments thanks to the Condor system.

Special attention has been dedicated to design a user friendly graphical interface with many useful functionalities.

Results of evolution/simulations/experiments can be saved both in graphical and processing modes. Their format is suitable for Matlab analysis. It is also possible to save and load some neural networks in the simulator.

Special attention has been devoted to design efficient and optimized algorithms. A profiler program has been used to detect most expensive parts of the program.

4 | Foraging

4.1 Description of the experiment

In this experiment, the robot has to forage in an unknown world where some food an poison items have been uniformly scattered. It is a basic version of animal foraging.

4.1.1 Sensors

In this experiment, the robot has **six** distance-sensors:

- three sensors to detect food items (left/middle/right sides). Each sensor only detects the closest item in its scope. The closer the item, the higher the measured value.
- three similar sensors to detect poison items.

4.1.2 Resources distributions

There are 10 food items and 30 poison items uniformly distributed on a square map. Once a resource is eaten, it is reset at another location.

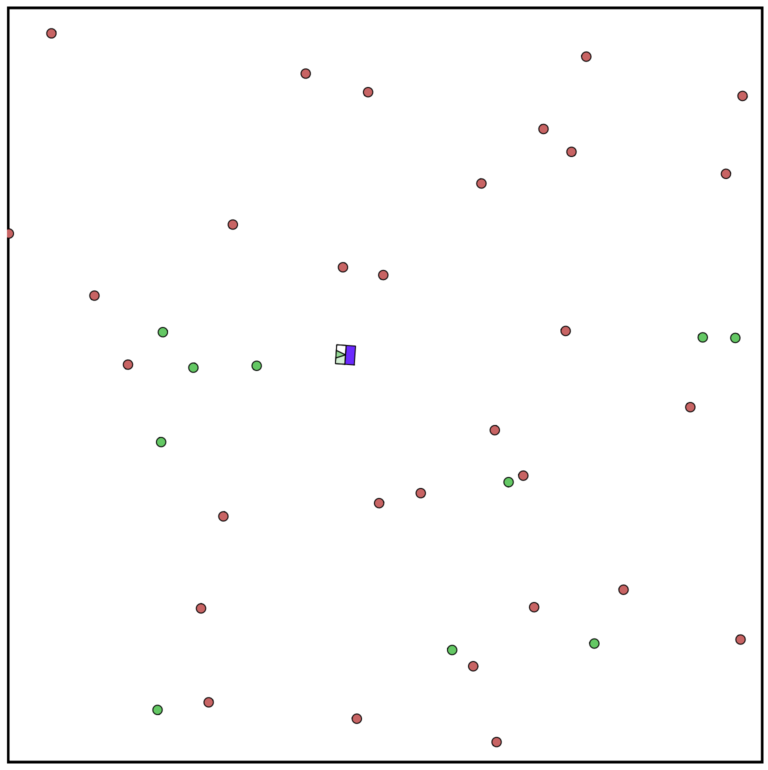


Figure 4.1: Example of distribution of food and poison items

4.1.3 Fitness function

After each simulation, the fitness of an individual is computed according to the following formula:

$$fitness = numberEatenFoodItems - 2 \times numberEatenPoisonItems$$

It means that initial random individuals are likely to have negative fitness values as the number of poisons items is greater than the number of food items and because of the coefficient 2 in the fitness equation.

Each simulation runs 1000 iterations (1000 robot's steps).

4.2 Evolution comparison

Figure 4.2 shows the evolution of the mean fitness of an individual through the generations of the genetic algorithm for the different neuron models.

The fitness values between two models can not be compared for the reasons explained in 2.5.3.

In the following, the best evolved robots of each best evolution of each experiment (each best one of each green curve) are named with their corresponding network properties. For example `for_QIF_3` is the best one controlled by a 3-QIF-neuron network.

4.2.1 Fitness and number of neurons

For each neuron model, the number of neurons can be compared based on the fitness values.
IZHI networks seem to be slightly more efficient with more neurons whereas CTRNN neurons seem slightly less. Nevertheless these differences are small and can simply be caused by stochastic effects due to a limited number of samples.
On the whole there is no big performance difference when the size of the controller is increased.

4.2.2 Evolution speed

Table 4.3 permits comparison of the evolutions based on the evolution speed index described in 2.5.4.

	3 neurons	10 neurons
CTRNN	4/12=0.33	3/8=0.37
QIF	13/24=0.54	14/24=0.58
IZHI	14/20=0.7	17/22=0.77

Figure 4.3: Speed indexes of the evolutions. Each value is the ratio between the mean fitness of the network after generation 50 and the mean fitness after generation 200.

IZHI networks are the fastest to evolve, followed by QIF and finally CTRNN ones. 10-neuron controller are slightly faster to evolve.

Contrary to QIF and IZHI models, CTRNN networks can still improve significantly if longer evolutions are run (figure 4.4).

The faster evolution of the spiking network in terms of generations can perhaps be explained by the fact that one iteration of the genetic algorithm on spiking networks needs much more computation than on a CTRNN network. Indeed, for each robot movement, spiking networks need to run 100 times in order to compute the mean firing rates of the neurons whereas CTRNN networks only use one run to update potentials (see 2.3.3). Nevertheless evolutions of both spiking and CTRNN

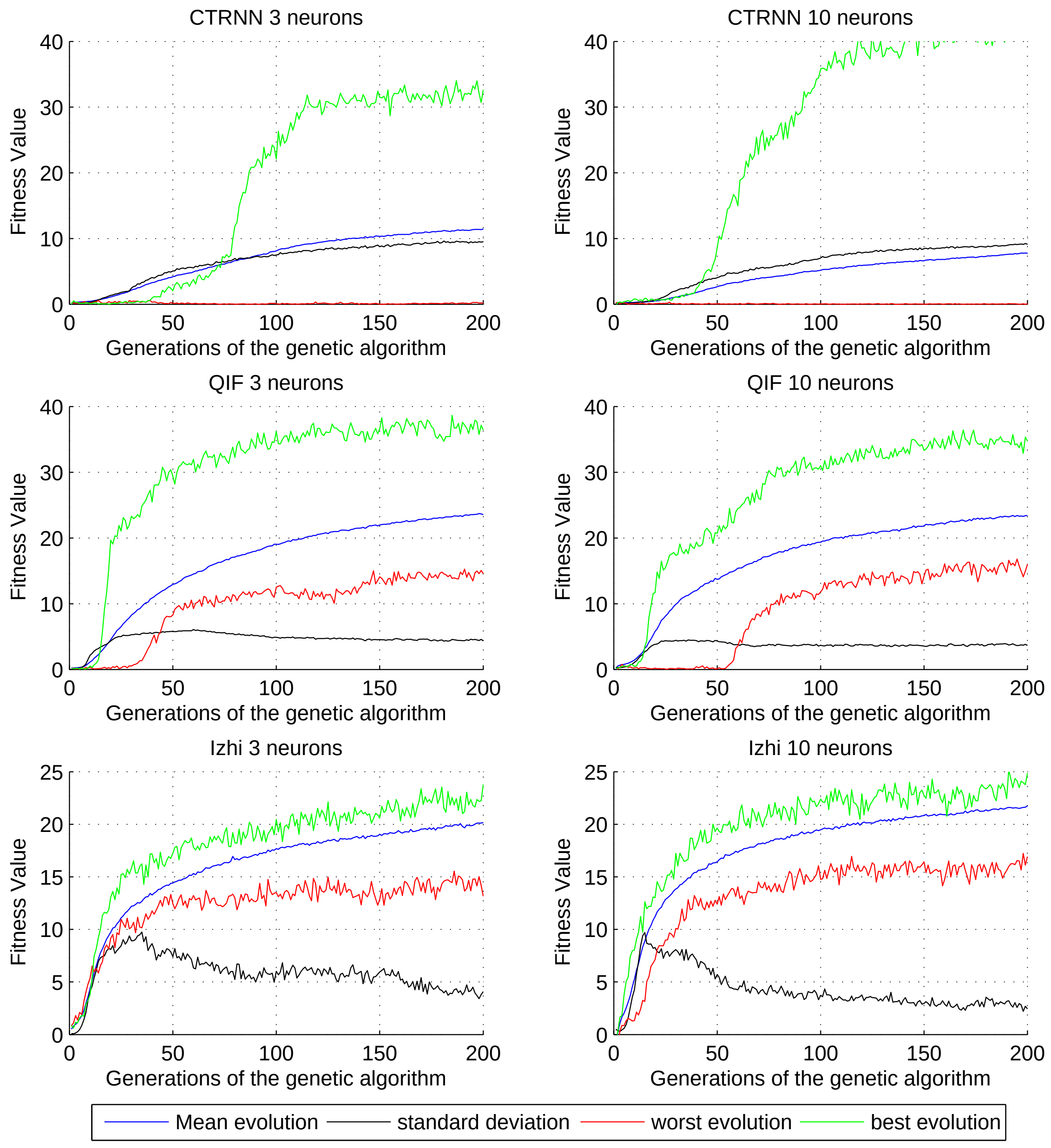


Figure 4.2: Comparison of the evolutions of different controllers over 200 generations. For each figure, 50 evolutions have been run.

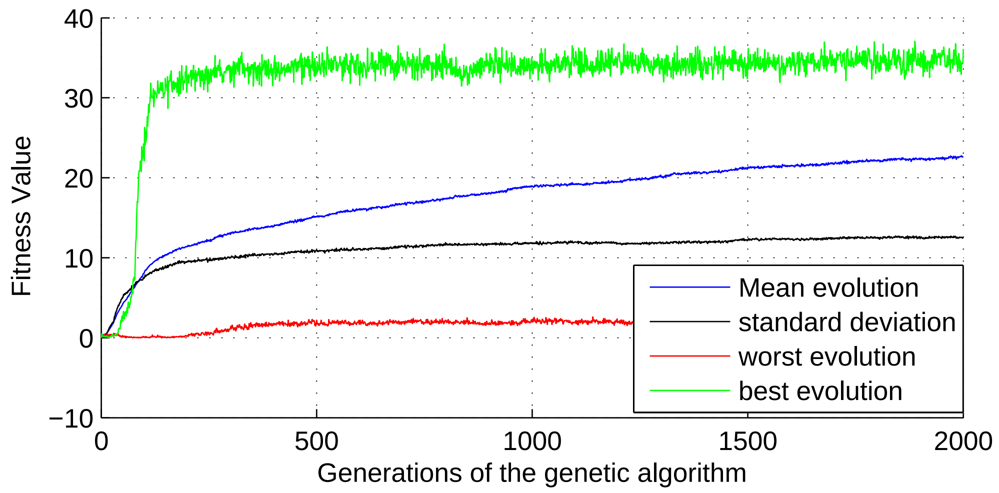


Figure 4.4: The mean fitness of CTRNN controllers can still increase

networks need approximately the same amount of processing time to "finish" (see 2.5.2). More accurate experiments should be done to measure this properly.

4.2.3 Deviation

Standard deviation cannot be directly compared between models as it depends on the fitness value. Nevertheless the ratio between the mean fitness and the standard deviation is comparable as it is explained in 2.5.5. Table 4.5 compares the deviation indexes for the different controllers.

	3 neurons	10 neurons
CTRNN	9.5/11.5=0.83	9/8=1.12
QIF	4.5/24.5=0.18	4/24.5=0.16
IZHI	4/20=0.2	2.5/22=0.11

Figure 4.5: Deviation indexes of the evolutions.

The deviation index is very low for spiking networks ($\simeq 0.1 - 0.2$) whereas it is very high for CTRNN networks ($\simeq 1$). For CTRNN networks, we can especially observe that some evolutions converge towards zero-fitness networks which correspond to robots that do not move. On the other hand they can potentially produce very good controllers with fitness ratio of 4 compared to the mean fitness (green curve).

4.3 Emergence of interesting behaviours

Best evolved robots perfectly head towards food items and avoid poison items. Nevertheless they usually do not use all their sensors to do so.
To move towards food items, best robots need no more than two sensors to spot and approach the food items.
To avoid poison items, robots generally only use their middle poison captor and when the robot is close enough to the poison item, it avoids it by turning in another direction.

Best evolved robots are quite efficient in some tricky situations:

- when the robot is surrounded by poison items, CTRNN best robots can reverse.
- when a food item lies behind some poison items, robots still manage to catch the food. In this case. the controller must integrate several inputs (see 4.6).

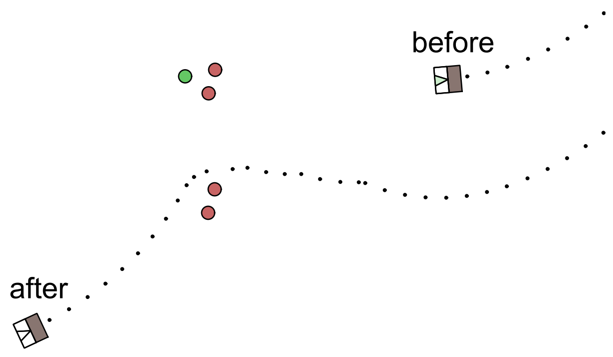


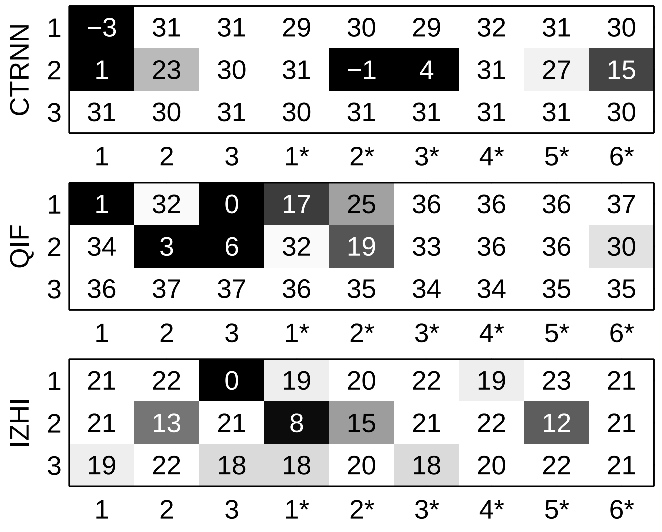
Figure 4.6: Integration of several sensor inputs (some sensors detect food and some others poison)

4.4 Dynamics of the evolved networks

Evolved controllers are capable to maintain some neural activity when they are not submitted to any sensor inputs or background current. This activity is particularly important for motor neurons in order to move as fast as possible between resource items. We can also observe that for QIF networks, the evolved refractory periods of the motor neurons are very low (close to the lowest possible value of 5 msec).

To visualize the importance of each connection in the networks, we can conduct some *lesion experiments* as explained in 2.6.

Following figures 4.7 and 4.8 show the results of such *lesions experiments*. The fitness has been averaged over 200 simulations in order to reduce variance.



CTRNN	1	2	3	1*	2*	3*	4*	5*	6*
1	−3	31	31	29	30	29	32	31	30
2	1	23	30	31	−1	4	31	27	15
3	31	30	31	30	31	31	31	31	30

QIF	1	2	3	1*	2*	3*	4*	5*	6*
1	1	32	0	17	25	36	36	36	37
2	34	3	6	32	19	33	36	36	30
3	36	37	37	36	35	34	34	35	35

IZHI	1	2	3	1*	2*	3*	4*	5*	6*
1	21	22	0	19	20	22	19	23	21
2	21	13	21	8	15	21	22	12	21
3	19	22	18	18	20	18	20	22	21

Figure 4.7: These lesions were done on the best 3-neuron controllers resulting from each evolution shown on 4.2. Neuron 1 corresponds to the left motor, neuron 2 to the right. Neuron 3 is an hidden neuron. Inputs 1* is the left food, 2* the right food and 3* the middle food sensor. Inputs 4* is the left poison, 5* the right poison and 6* the middle poison sensor.

4.4.1 Difference between 3 and 10 neurons networks

On 10-neuron controllers, a large proportion of connections seem useless whereas only half of the lesions have no effect on 3-neuron controllers. 10-neuron networks seem therefore exaggerated for this task.

For both 3 and 10-neuron controllers, most crucial connections are those pointing to the two motor neurons 1 and 2. The two first lines of each network are indeed the darkest ones. The motors recurrent connections are usually very important to maintain some activity when no inputs are submitted to the sensors. If we delete one of them, the robots usually do not move anymore (or just a little bit thanks to the slight background current received by the *noisy neuron*) because the activity of the network can not self-sustain.
Now if we observe the robots behaviours when we remove some connections from the inputs to the motors, we see that the robots usually still move but can not react anymore to the corresponding input and the fitness can be strongly altered.
If we look carefully to the connections from the inputs, we observe that the evolved robots perform well even if some of their inputs are removed. This confirms the observations from 4.3 showing that good evolved robots do not need to react to all their sensors and that only two food sensors and the middle poison sensor are enough to get very efficient robots.

CTRNN network

	1	2	3	4	5	6	7	8	9	10	1*	2*	3*	4*	5*	6*
1	34	19	2	46	18	46	46	45	37	1	24	43	40	46	45	38
2	40	11	17	46	0	46	45	46	5	26	40	23	38	46	41	39
3	45	46	46	45	32	46	46	47	46	46	46	45	46	46	46	45
4	46	46	47	46	46	46	45	46	46	45	47	46	46	46	47	46
5	46	46	46	46	46	46	46	46	46	46	46	46	46	46	46	45
6	47	47	47	46	46	46	45	46	46	46	46	47	46	46	47	45
7	47	46	46	45	45	46	46	45	46	45	45	46	46	46	46	45
8	46	46	46	46	46	46	46	46	46	46	46	46	46	46	46	46
9	45	46	46	45	45	46	47	46	46	46	45	46	46	46	47	46
10	45	46	3	46	46	46	47	45	44	44	46	46	46	44	46	46

QIF network

	1	2	3	4	5	6	7	8	9	10	1*	2*	3*	4*	5*	6*
1	19	34	34	33	32	34	21	34	30	34	14	31	31	30	38	28
2	37	22	36	34	26	35	27	34	19	37	30	32	33	36	32	32
3	34	34	34	34	34	34	33	34	34	33	33	34	36	32	34	33
4	34	34	34	34	34	34	34	34	34	34	32	34	35	35	35	35
5	34	35	9	33	34	34	33	34	33	34	35	34	37	35	33	34
6	34	33	34	34	34	34	34	34	34	34	35	33	33	32	35	34
7	34	23	35	34	33	34	35	35	32	35	32	33	35	34	31	37
8	35	32	34	34	34	34	32	34	33	34	36	33	36	35	33	34
9	36	34	36	34	23	33	35	30	24	27	35	33	27	33	23	37
10	34	34	34	34	34	34	34	34	34	34	35	32	34	34	33	35

IZHI network

	1	2	3	4	5	6	7	8	9	10	1*	2*	3*	4*	5*	6*
1	21	21	22	22	27	14	21	25	23	22	2	25	26	24	25	22
2	24	−2	24	23	22	23	23	26	23	11	13	22	19	22	6	14
3	24	24	23	23	24	25	23	24	23	24	23	24	23	24	23	24
4	23	23	23	24	24	23	23	23	24	24	23	24	23	22	23	25
5	22	23	24	22	25	23	23	24	23	25	24	24	25	24	24	24
6	25	25	24	23	23	24	25	22	22	23	23	24	24	23	23	21
7	23	24	22	24	24	21	23	23	24	23	23	23	23	22	25	25
8	24	25	23	23	23	22	23	24	23	22	22	23	23	22	24	24
9	23	24	24	23	24	24	22	23	23	23	23	23	24	24	24	24
10	25	22	22	24	23	23	22	19	23	23	21	23	24	23	24	26

Figure 4.8: These lesions were done on the best 10-neuron controllers resulting from each evolution shown on 4.2. Neuron 1 corresponds to the left motor, neuron 2 to the right. Neurons 3 to 10 are hidden neurons. Inputs 1* is the left food, 2* the right food and 3* the middle food sensor. Inputs 4* is the left poison, 5* the right poison and 6* the middle poison sensor.

4.4.2 Difference between neuron models

CTRNN networks seem to have more crucial connections than spiking networks which are slightly more balanced in the sense that they have more grey cells and less black ones.

`for_QIF_10` seems particularly robust as it only has two black cells. Its 9th hidden neuron seems particularly important as the 9th line contains five grey cells.

4.5 Conclusion

This experiment has permitted to design some very efficient controllers for the given foraging task. We have observed that 10 neurons with 3 food sensors and 3 poison sensors is unnecessary and that 3-neuron controllers with only two food sensors and one poison sensor can efficiently perform the task.

Evolved controllers are capable to maintain some activity with no background fire (except at the really beginning to "switch on" the network). This is an interesting result especially with only 3 neurons and some refractory periods greater than 5 msec.

Among the 3-neuron models, Izhikevich-based neuro-controllers are the fastest to evolve and have the lowest deviation when several independent evolutions are run. These are two valuable qualities, especially for engineering/industrial applications.
CTRNN neuro-controllers take much longer to evolve and their evolutions deviation index is very high. It means that their evolution depends heavily on the initial population. Despite this major drawback, some lucky evolutions can generate some very efficient evolved controllers.

Evolved robots are mostly reactive sensors-motors systems with no internal states. This is confirmed by the lesions experiments showing that most neurons and connections of 10-neuron controllers are useless and that inputs$\rightarrow$outputs connections and motors$\leftrightarrow$motors ones are the most important.

The next experiment introduces a new task that should produce more complex dynamics and behaviours.

5 | Exploration and Grazing

5.1 Description of the experiment

In this experiment, the robot has to explore and forage in an unknown world where food items are gathered in small scattered groups. It is a slightly more plausible version of animal foraging.

This new task is more complex because it will require the robot to alternate between two behaviours:

- an exploration one where the robot has to find a group of food items
- a grazing one where the robot has to eat all food items of the discovered group

To complicate the task and to force the robot to display more complex behaviours, both right and left sensors have been disabled meaning that the robot can only see forward. From this, we can expect the neural network to show interesting time-dependant dynamics.

Note that on this section's figures, world items' sizes are twice larger than the real simulator ones in order to be visible.

5.1.1 Sensors

In this experiment, the robot has **two** distance-sensors:

- one sensor to detect food items in front of the robot. It only detects the closest item in its scope. The closer the item, the higher the measured value.
- one similar sensor to detect poison items in front of the robot.

5.1.2 Resources distributions

There are 30 poison items uniformly distributed on a square map.
There are 30 food items gathered in five zones/groups. Each zone is uniformly distributed on the map. Inside a zone, food items are uniformly distributed.
Once a resource is eaten, it disappears, it is not reset at another location. The world's width is 3 times bigger compared to the previous foraging experiment.
Figure 5.1 shows an example of the distribution.

5.1.3 Fitness function

After each simulation, the fitness of an individual is computed according to the following formula:

$$fitness = numberEatenFoodItems - 2 \times numberEatenPoisonItems$$

Each simulation runs 1000 iterations (1000 robot's steps).

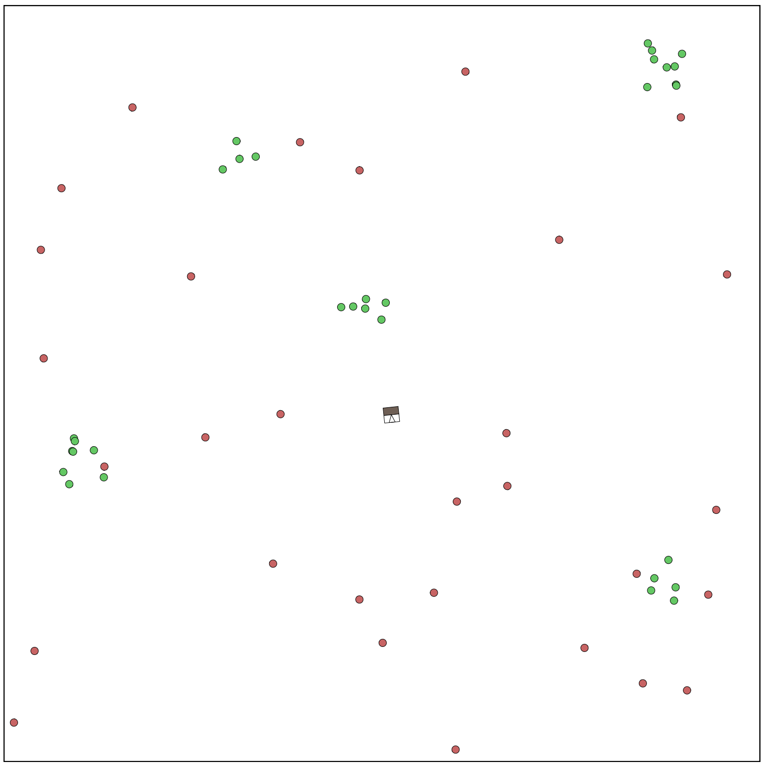


Figure 5.1: One possible distribution of food and poison items. Items are twice their normal size for printing purpose.

5.2 Evolution comparison

Figure 5.2 shows the evolution of the mean fitness of an individual through the generations of the genetic algorithm for the different models of neurons.

Once again the fitness values between two models can not be compared for the reasons explained in 2.5.3.

In the following, the best evolved robots of each best evolution of each experiment (each best one of each green curve) are named with their corresponding network properties. For example `explo_QIF_3` is the best one controlled by a 3-QIF-neuron network.

5.2.1 Fitness and number of neurons

10 neurons seem to give on average better results for the CTRNN networks. After 200 generations, the fitness of the mean evolution of the 10-neuron CTRNN network is 13 whereas it is 10 for the 3-neuron CTRNN network. After 2000 generations the same gap persists, with a mean fitness of 15 for the 3-neuron network and a mean fitness of 18 for the 10-neuron network. Nevertheless this might simply be caused by stochastic effects due to a lack of samples as the deviation index is high for the CTRNN networks.

On the whole there is no big performance difference when the size of the controller is increased.

5.2.2 Evolution speed

Table 5.3 permits to compare the evolutions based on their speed index described in 2.5.4.

IZHI networks are once again the fastest to evolve, followed by QIF and finally CTRNN ones. This is true for both 3 and 10 neurons.

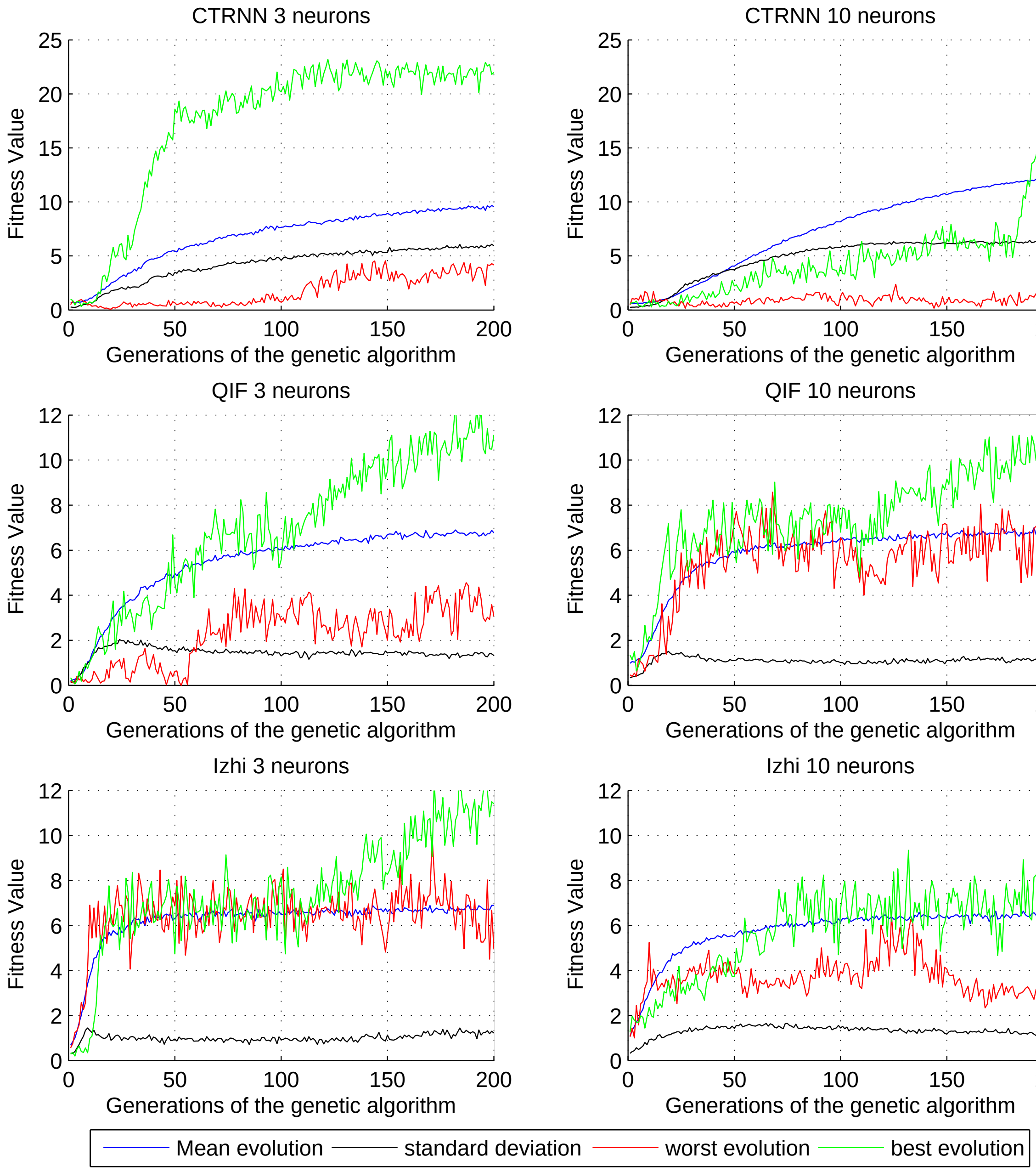


Figure 5.2: Comparison of the evolutions of different controllers over 200 generations. For each figure, 50 evolutions have been run.

	3 neurons	10 neurons
CTRNN	5.5/9.5=0.58	14/12.5=0.32
QIF	5/7=0.71	6/7=0.86
Izhi	6.5/6.5=1	5.7/6.4=0.89

Figure 5.3: Evolution speed indexes for the different controllers. Each value is the ratio between the mean fitness of the network at generation 50 and the mean fitness at generation 200.

While QIF and IZHI networks cannot evolve much further after 200 generations, CTRNN networks can still improve significantly. After 2000 generations, the mean fitness of the 3-neuron CTRNN controller is 15 and 18 for the 10-neuron one.

5.2.3 Deviation

Standard deviation cannot be directly compared between models as it depends on the fitness value. Nevertheless the ratio between the mean fitness and the standard deviation is comparable as it is explained in 2.5.5. Table 5.4 shows the deviation indexes for the different controllers.

	3 neurons	10 neurons
CTRNN	6/10=0.6	6/12.5=0.48
QIF	1.5/7=0.21	1.5/7=0.21
Izhi	1/7=0.14	1/6.5=0.15

Figure 5.4: Deviation indexes for the different controllers.

The deviation indexes are low for spiking networks ($\simeq 0.1 - 0.2$) whereas they are high for CTRNN networks ($\simeq 0.5$). For these latter ones, we can especially observe that some evolutions converge towards zero-fitness controllers which correspond to robots that do not move. On the other hand they can potentially produce very good controllers with fitness ratio of 2.5 compared to the mean evolution fitness.

5.3 Emergence of interesting behaviours

5.3.1 Exploration strategies

Spiking networks

Most evolutions of spiking networks do not give efficient robots for the exploration phase.
All IZHI neuron controlled robots are tracing a circle with a small radius which is very inefficient as they find food zones only if these ones are very close to the initial position of the robot. `explo_Izhi_3` and `explo_Izhi_10` won't thus be studied.
Most QIF neurons controlled robots describe the same inefficient pattern of doing small circles but approximately a quarter of the evolutions gives more efficient robots describing some larger circles and capable of efficiently exploring the world. It is the case of `explo_QIF_10`.

`explo_QIF_10`

`explo_QIF_10` describes a large circle when there are no poison nor food items on the map. It is quite efficient to explore the world (see 5.5).

`explo_QIF_10` performs well (see 5.6) by alternating between an exploration and a grazing modes. Nevertheless, after eating all the items of a food zone, the robot keep circling quite a lot of time before leaving the *grazing mode*. On figure 5.6, the former food groups are clearly recognizable as the robot has sometimes looped for a while before leaving the empty food zones.

While exploring (no inputs), the controller's potentials are periodic (see 5.7). Such neurons have therefore periodic mean firing rates (almost constant).

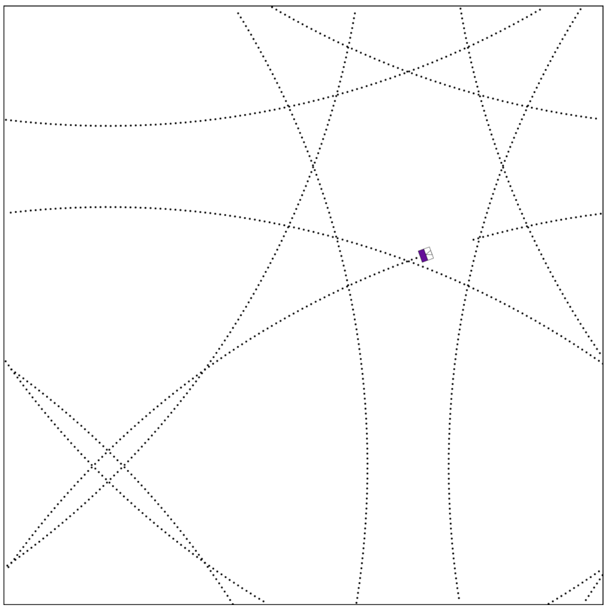


Figure 5.5: `explo_QIF_10` describing a circle trajectory when it doesn't receive any inputs

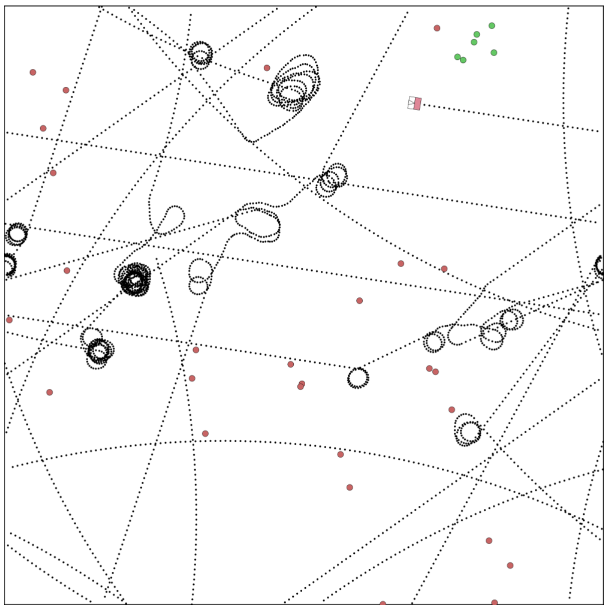


Figure 5.6: `explo_QIF_10`'s trajectory example while exploring and grazing

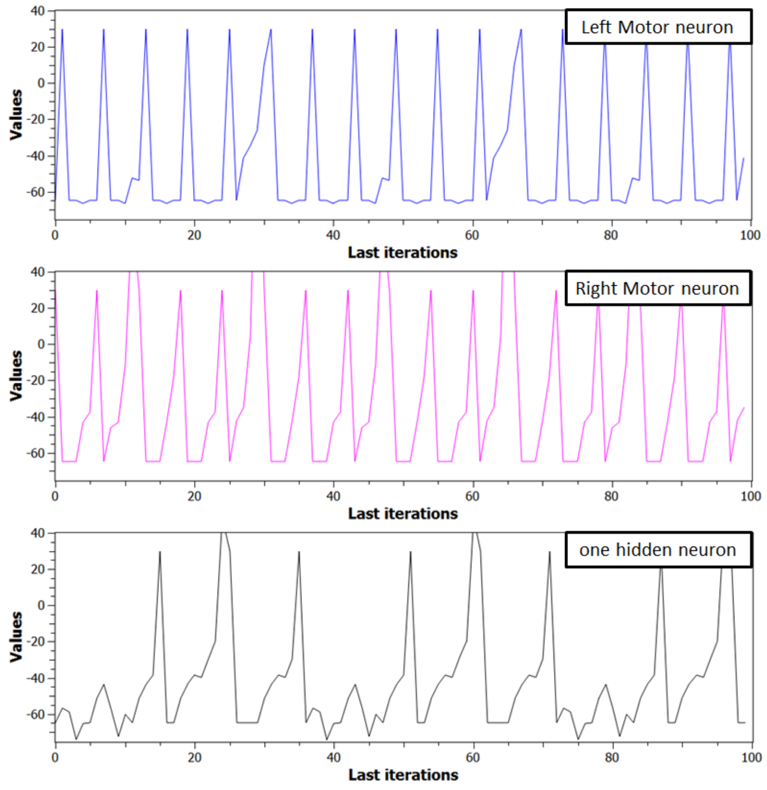


Figure 5.7: `explo_QIF_10`'s motor potentials and a hidden neuron's potential that presents a more complex periodicity.

`explo_CTRNN_3`

`explo_CTRNN_3` displays an interesting exploration strategy when it doesn't receive any inputs (no poison/food items). It is the composition of two movements:

- a large scale movement (figure 5.8) that isn't a cycle but rather a random "moving circle" (a circle with a center that randomly moves over time). It allows the robot to explore the world very efficiently.
- a small scale movement (figure 5.10) consisting in going forward for a moment and then doing a pirouette. As the robot only has a forward sensor, this pirouette is a mean to scan a 360 degree view around the robot.
  It is interesting to see that the distance/time between two pirouettes is stochastic.

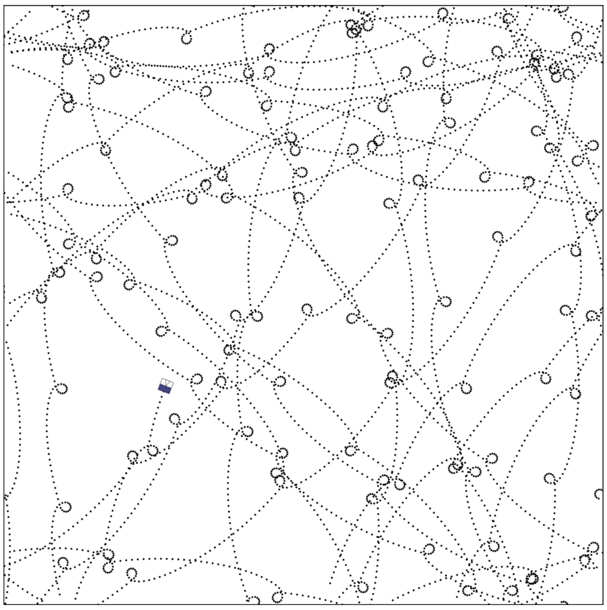


Figure 5.8: `explo_CTRNN_3` describing a general "non-cyclic" trajectory when it doesn't receive any inputs

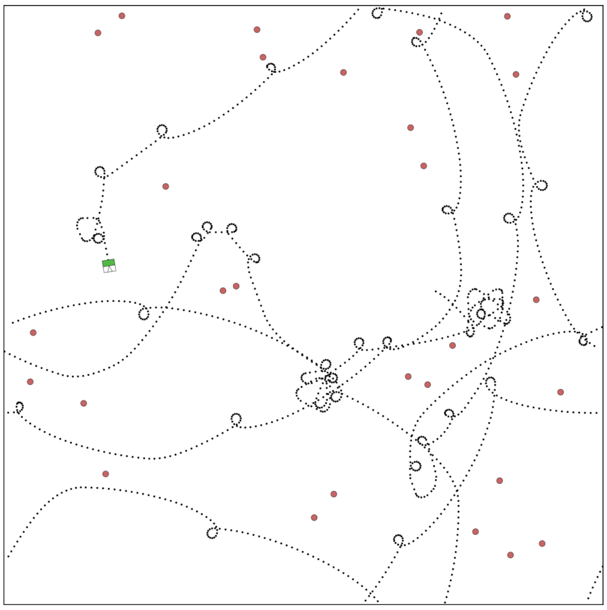


Figure 5.9: `explo_CTRNN_3`'s path example after having eaten all the food items

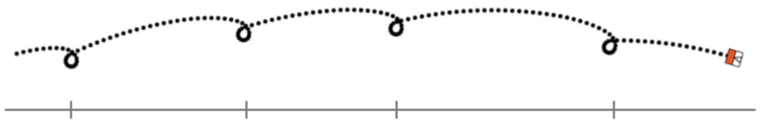


Figure 5.10: `explo_CTRNN_3` shows randomness in the distance between two pirouettes

Doing such pirouettes is an interesting behaviour often exhibited by animals. For example some ant species from the *Cataglyphis* family living in the Sahara desert memorize the location of their nest thanks to sun's angle measurements. When an ant ventures out, it regularly does a pirouette to measure the sun's angle in order to memorize the way back to the nest.

The randomness in the robot's exploration is due to the random noise that the network receives from the *noisy neuron*.

Figure 5.9 shows the path of the robot when poison items and groups of food items are scattered. Poison items are avoided and contributes to diversify the path. When a group of food items is spot, the robot moves and enter in a grazing mode described in 5.3.2.

5.3.2 Approach strategies

`explo_QIF_10`

Figure 5.11 shows `explo_QIF_10` spotting and eating the food items of a food zone.

The controller has therefore:

- one stable attracting dynamics: the exploration one
- one unstable attracting dynamics: the circling one after eating all food items of a food zone

On figure 5.6, it was already possible to see that the time spent in the circling attractor is stochastic. This can potentially be explained by two reasons: either the dynamics are chaotic or it is the random background current received by the *noisy neuron* that is responsible for this random time spent in the circling attractor.

To investigate, a small modified simulation has been run: the background current received by the *noisy neuron* is only activated at the beginning in order to "switch on" the controller and then is shut down. The results are categoric: when the robot spots a food zone, eat all the items and enter the circling state, it never stops circling. The circling attractor is now stable.

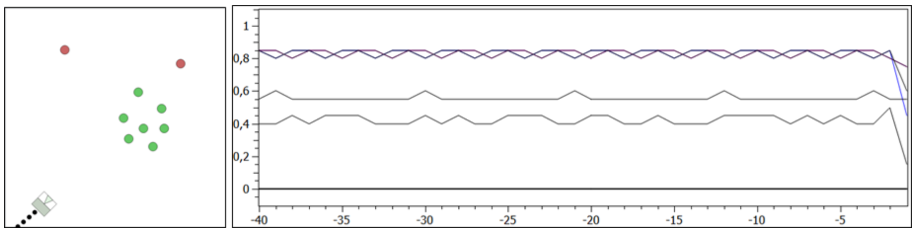
In conclusion the background current received by the *noisy neuron* is extremely important. It is indeed responsible for the instability of the circling attractor (which is a essential).

`explo_CTRNN_3`

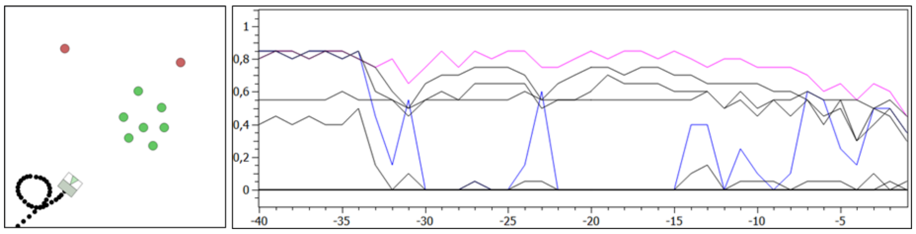
Figure 5.12 show `explo_CTRNN_3` spotting and eating the food items of a food zone.

This approach tactic appears to be very efficient. Contrary to `explo_QIF_10`, `explo_CTRNN_3` has only one attractor that is disturbed by sensor inputs. Actually, the robot's behaviour can be described by the following algorithm:

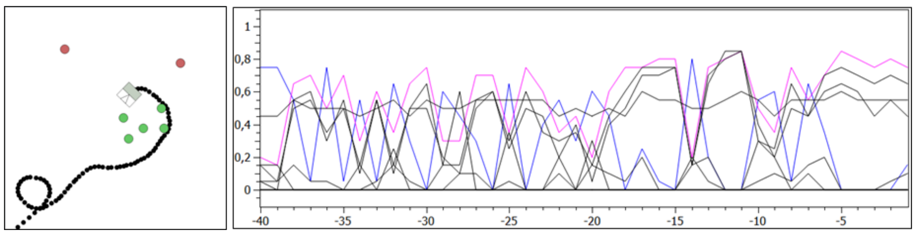
```
1. Move forward
2. Start a full pirouette. If a food item is spot ahead, stop the pirouette. Go back to 1.
```



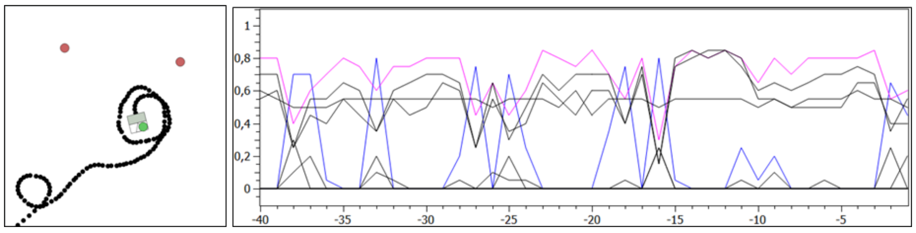
(a) First the robot does a pirouette just after spotting the first food items. This pirouette doesn't seem directly useful.



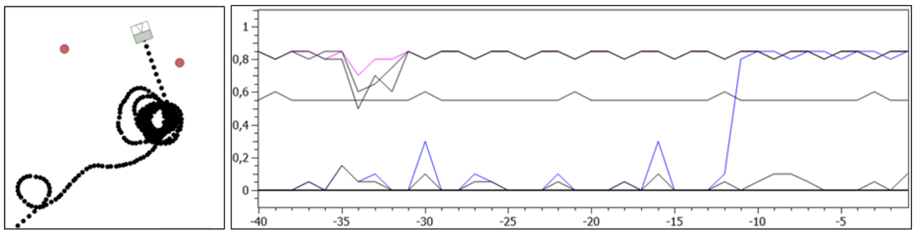
(b) After finishing the pirouette, it spots again the closest food item and this time the robot moves toward the zone



(c) As the robot arrives closer to the zone, it starts a surrounding of the food zone going anticlockwise.

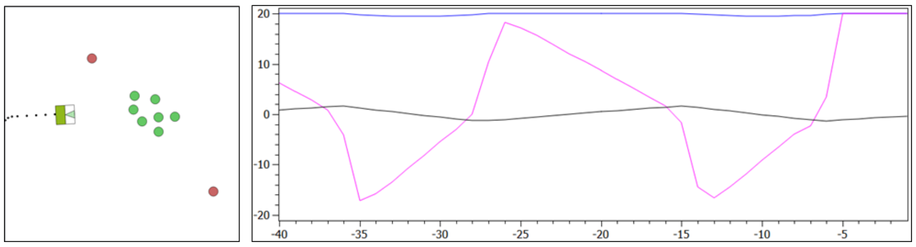


(d) Little by little the robot eats the peripheral items and form a spiral converging towards a circle

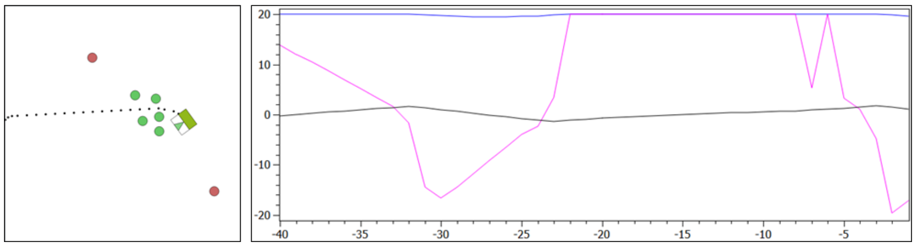


(e) After eating all the food items, the robot keeps circling for a while and then suddenly leaves the former food zone.

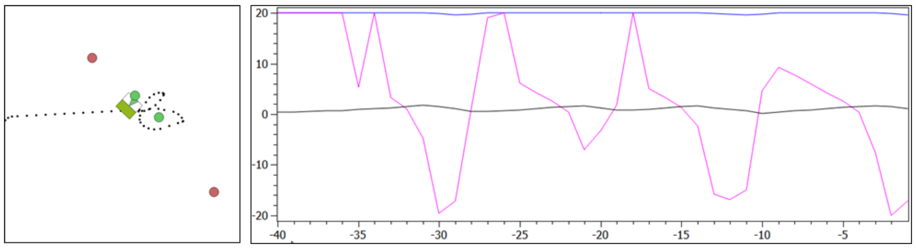
Figure 5.11: Approach of `explo_QIF_10`. The blue curve is the mean firing rate of the left motor neuron. The magenta curve is the mean firing rate of the right motor neuron.



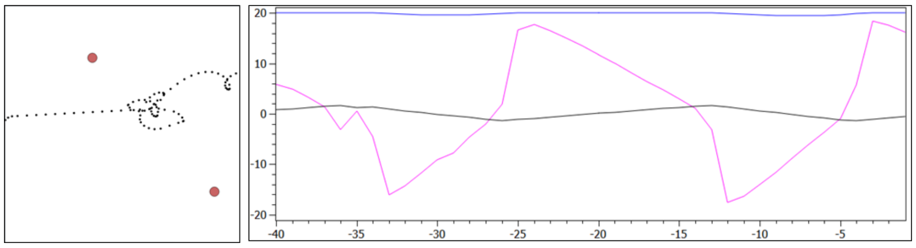
(a) Before spotting the first food items, the right motor neuron potential is oscillating giving the behaviour of figure 5.10. The left motor neuron potential is almost constant and close to 20, the maximal value.



(b) Once the food item spotted, the robot moves straight forward in its direction and eats it. The robot then turns back as there are no more forward food item.



(c) The robot then turns back several times in order to catch the other food items. Note that the right motor neuron potential is still constant. Robot's actions are only due to the left motor neuron's potential variations



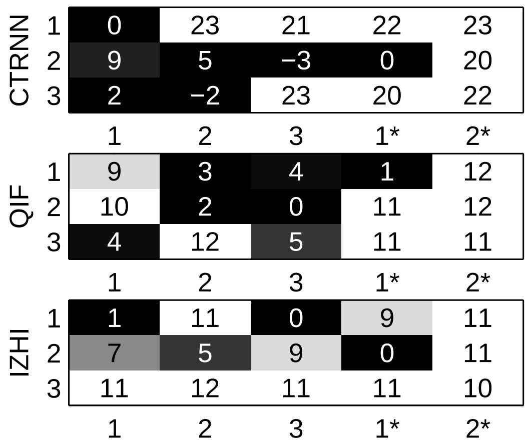
(d) After eating all the food items, the robot comes back to its exploration mode: doing some small pirouettes and moving forward.

Figure 5.12: Approach of `explo_CTRNN_3`. The blue curve is the potential of the left motor neuron. The magenta curve is the potential of the right motor neuron.

5.4 Lesion experiments

To visualize the importance of each connection in the networks and to understand some of the behaviours, some *lesion experiments* as explained in 2.6 have been conducted.

Following figures 5.13 and 5.14 show the results of such *lesions experiments*. Each fitness value has been averaged over 200 simulations in order to reduce the variance.



CTRNN	1	2	3	1*	2*
1	0	23	21	22	23
2	9	5	−3	0	20
3	2	−2	23	20	22

QIF	1	2	3	1*	2*
1	9	3	4	1	12
2	10	2	0	11	12
3	4	12	5	11	11

IZHI	1	2	3	1*	2*
1	1	11	0	9	11
2	7	5	9	0	11
3	11	12	11	11	10

Figure 5.13: These lesions were done on the best 3-neuron controllers resulting from each evolution shown on figure 5.2. Neuron 1 corresponds to the left motor, neuron 2 to the right. Neuron 3 is an hidden neuron. Inputs 1* is the middle food sensor, 2* is the middle poison sensor.

Compared to the previous experiment, there are overall more dark cells. It confirms that the network dynamics is more complex.

5.4.1 CTRNN networks

`explo_CTRNN_3` has only one black cell on its first line. In is in compliance with the previous observations from 5.3.1. The black cell corresponds to the connection $1 \leftarrow 1$, i.e. the recurrent connection of the right motor neuron. We saw indeed that the right motor neuron's potential is constant whatever the sensors inputs are.

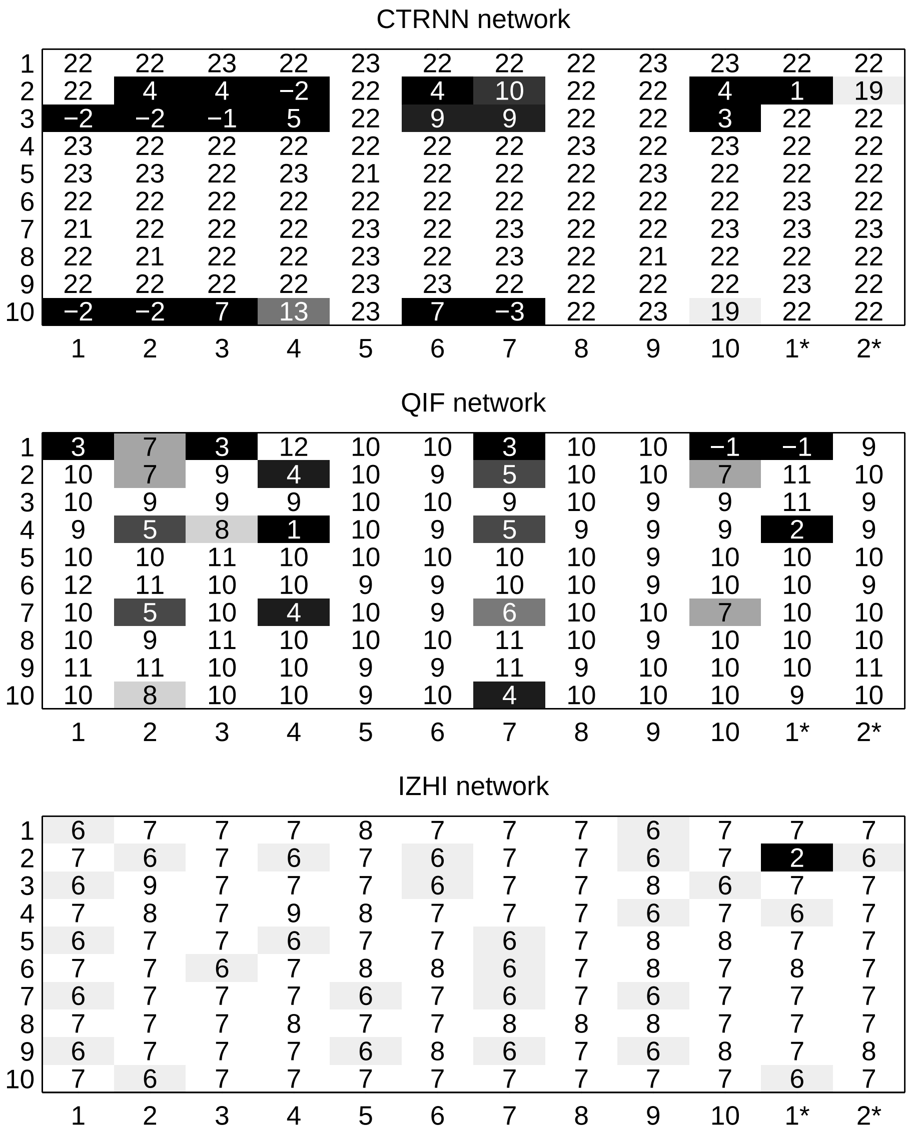
`explo_CTRNN_10` has 3 important lines and 8 important columns whereas `for_CTRNN_10` had only 2 lines and 5 columns. 10^{th} hidden neuron of `explo_CTRNN_10` is really important.

5.4.2 QIF networks

Among the 10-neuron controllers, `explo_QIF_10` has the most balanced lesion matrix. A few important lines and columns show up: lines 1,2,4,7,10 and logically columns 2,4,7,10. The network seems quite complex as a lot of connections are really crucial: there are 14 black/dark grey cells. This complexity perfectly fits the complexity of the robot's behaviours.

5.4.3 IZHI networks

`explo_IZHI_3` and `explo_IZHI_10` do not perform well the task. In particular, only one lesion really affects the performance of `explo_IZHI_10`: it corresponds to the weight from the food sensor input to the right motor neuron.



CTRNN network

	1	2	3	4	5	6	7	8	9	10	1*	2*
1	22	22	23	22	23	22	22	22	23	23	22	22
2	22	4	4	−2	22	4	10	22	22	4	1	19
3	−2	−2	−1	5	22	9	9	22	22	3	22	22
4	23	22	22	22	22	22	22	23	22	23	22	22
5	23	23	22	23	21	22	22	22	23	22	22	22
6	22	22	22	22	22	22	22	22	22	22	23	22
7	21	22	22	22	23	22	23	22	22	23	23	23
8	22	21	22	22	23	22	23	22	21	22	22	22
9	22	22	22	22	23	23	22	22	22	22	23	22
10	−2	−2	7	13	23	7	−3	22	23	19	22	22

QIF network

	1	2	3	4	5	6	7	8	9	10	1*	2*
1	3	7	3	12	10	10	3	10	10	−1	−1	9
2	10	7	9	4	10	9	5	10	10	7	11	10
3	10	9	9	9	10	10	9	10	9	9	11	9
4	9	5	8	1	10	9	5	9	9	9	2	9
5	10	10	11	10	10	10	10	10	9	10	10	10
6	12	11	10	10	9	9	10	10	9	10	10	9
7	10	5	10	4	10	9	6	10	10	7	10	10
8	10	9	11	10	10	10	11	10	9	10	10	10
9	11	11	10	10	9	9	11	9	10	10	10	11
10	10	8	10	10	9	10	4	10	10	10	9	10

IZHI network

	1	2	3	4	5	6	7	8	9	10	1*	2*
1	6	7	7	7	8	7	7	7	6	7	7	7
2	7	6	7	6	7	6	7	7	6	7	2	6
3	6	9	7	7	7	6	7	7	8	6	7	7
4	7	8	7	9	8	7	7	7	6	7	6	7
5	6	7	7	6	7	7	6	7	8	8	7	7
6	7	7	6	7	8	8	6	7	8	7	8	7
7	6	7	7	7	6	7	6	7	6	7	7	7
8	7	7	7	8	7	7	8	8	8	7	7	7
9	6	7	7	7	6	8	6	7	6	8	7	8
10	7	6	7	7	7	7	7	7	7	7	6	7

Figure 5.14: These lesions were done on the best 10-neuron controllers resulting from each evolution shown on 5.2. Neuron 1 corresponds to the left motor, neuron 2 to the right. Neurons 3 to 10 are hidden neurons. Inputs 1* is the middle food sensor, 2* is the middle poison sensor.

5.5 Conclusion

A more complex task has led to more complex controllers and behaviours.

Once again Izhikevich's neuron controllers are the fastest to evolve and have the lowest deviation when several independent evolutions are run. However, they have failed to provide good controllers for the given task.
CTRNN controllers take longer to evolve and their variance among several independent evolutions is very high but some lucky evolutions have returned very efficient controllers.

Successful evolved robots are capable of complex behaviours: they keep switching alternatively between exploration and grazing modes. Network dynamics of `explo_QIF_10` shows the presence of two attractors, one stable and the other unstable. The instability of the grazing attractor is crucial and experiments have proved it was due to the background current noise that can therefore be very useful.
CTRNN controllers are doing scanning pirouettes similar to some animals' ones.

Lesion experiments confirm the complexity of the network dynamics revealing two times more black cells than in the previous experiment.

6 | Diversifying diet

6.1 Description of the experiment

In this experiment, the robot has to forage in an unknown world where two types of food items have been scattered. The robot has to **alternatively** eat the two types. This kind of situation can be observed in nature. For example, some leafcutter ants from the *Atta* family regularly change their plants diet. Scientists believe this behaviour prevents the plants from dying (by completely loosing their leaves) and helps them to regenerate in order to become new future resources for the ants. This means that the ants are aware of the former plant species they were eating. In this experiment, this knowledge have been replaced by a state sensor.

6.1.1 Sensors

In this experiment, the robot has **seven** sensors:

- three sensors to detect A food items (left/middle/right sides). Each sensor only detects the closest item in its scope. The closer the item, the higher the measured value.
- three similar sensors to detect B food items.
- one state sensor that returns 1 if the last eaten item of the robot belongs to A class and zero if it belongs to B class.

The robot is expected to integrate several inputs in order to take the right decision.

6.1.2 Resources distributions

There are 10 A items and 30 B items uniformly distributed on a square map. Once a resource is eaten, it is reset at another location.

6.1.3 Fitness function

Every time the robot successively eats two different food items, its fitness is increased by one. Otherwise if two successive eaten items belong to the same food class, its fitness is decreased by one.

Each simulation runs 500 iterations (500 robot's steps).

6.2 Evolution comparison

Figure 6.1 shows the genetic evolutions of the different controllers.

Once again fitness values between two models can not be compared for the reasons explained in 2.5.3.

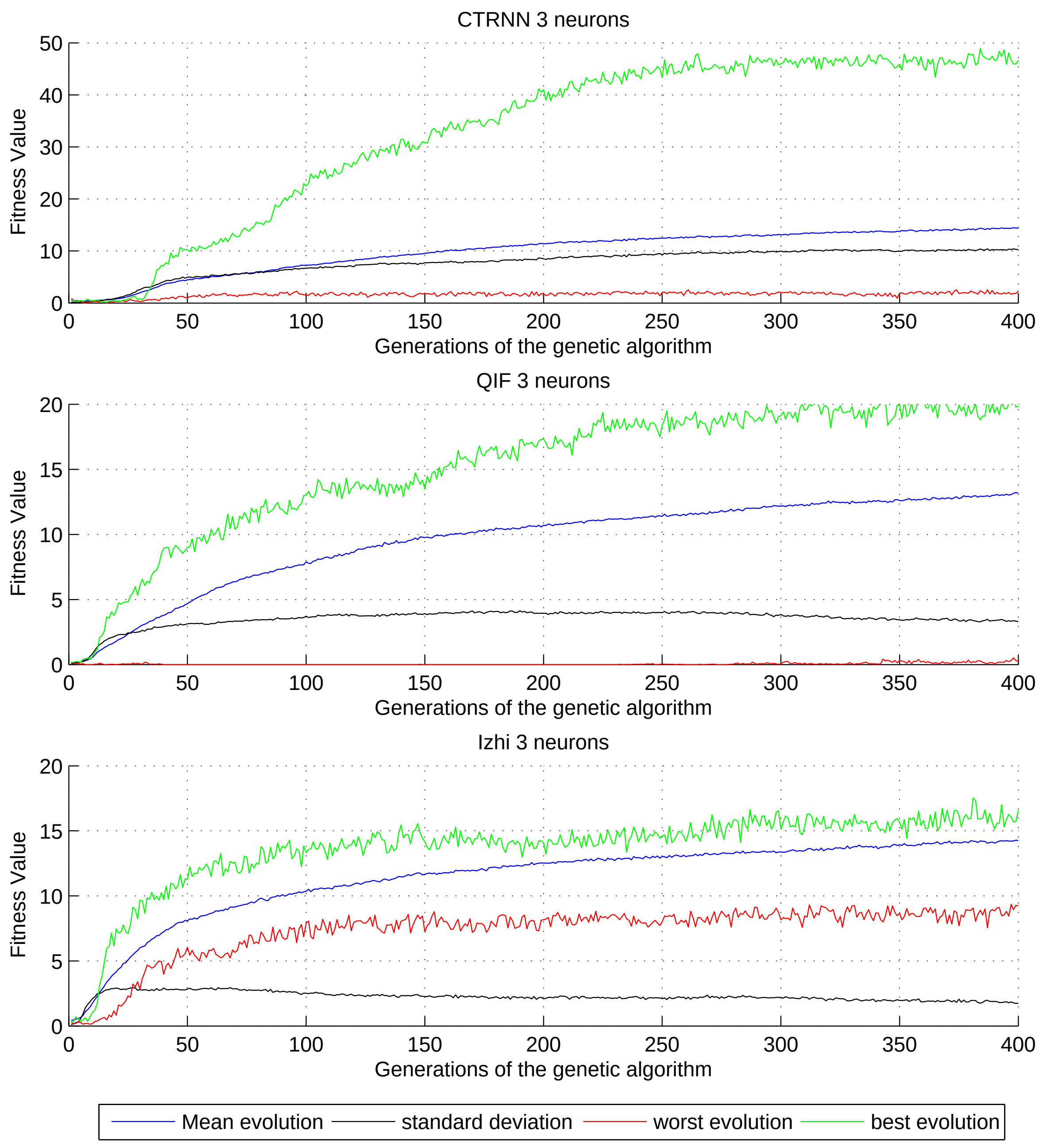


Figure 6.1: Comparison of the evolutions of different controllers over 400 generations. For each figure, 50 evolutions have been run.

In the following, the best evolved robots of each best evolution of each experiment (each best one of each green curve) are named with their corresponding network properties. For example `switch_QIF_3` is the best one controlled by a 3-QIF-neuron network.

6.2.1 Evolution speed

Table 6.2 permits to compare the evolutions based on the evolution speed index described in 2.5.4.

These observations confirm those from previous experiments: IZHI controllers are the fastest to evolve, followed by QIF ones and finally CTRNN ones.

Neuron model	Speed index
CTRNN	4.4/11.4=0.38
QIF	4.7/10.7=0.44
IZHI	8.1/12.5=0.65

Figure 6.2: Speed indexes of the three different controllers

6.2.2 Deviation

Table 6.3 permits to compare the evolutions based on the evolution deviation index described in 2.5.5.

Neuron model	Deviation index
CTRNN	10.3/14.5=0.71
QIF	3.3/13.2=0.25
IZHI	1.7/14.3=0.12

Figure 6.3: Deviation indexes after the 400th generation of the three different controllers

These observations confirm those from previous experiments: IZHI controllers have the lowest variance among several independent evolutions, followed by QIF controllers and finally CTRNN ones.

6.3 Dynamics of the evolved networks

6.3.1 Two working modes

For all best controllers returned from evolutions, their network dynamics are swapping between two modes:

- in the first mode, the robot looks for A items and avoid B items
- in the second mode, the robot looks for B items and avoid A items

The two modes are therefore radically different, the robot inverting its entire behaviour.

Figures 6.4, 6.5 and 6.6 show the modes transition of the different controllers.

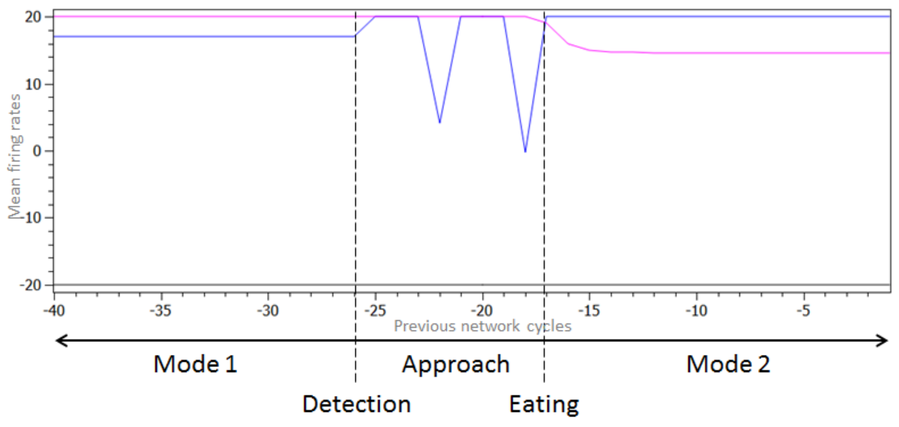


Figure 6.4: The potentials show the two modes of `switch_CTRNN_3`. The magenta curve corresponds to the right motor neuron, the blue one to the left motor neuron and the black one to the hidden neuron.

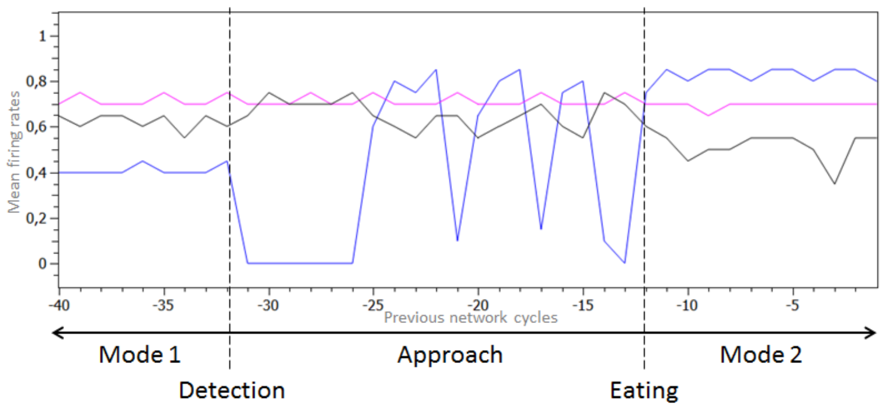


Figure 6.5: The mean firing rates show the two modes of `switch_QIF_3`. The magenta curve corresponds to the right motor neuron, the blue one to the left motor neuron and the black one to the hidden neuron.

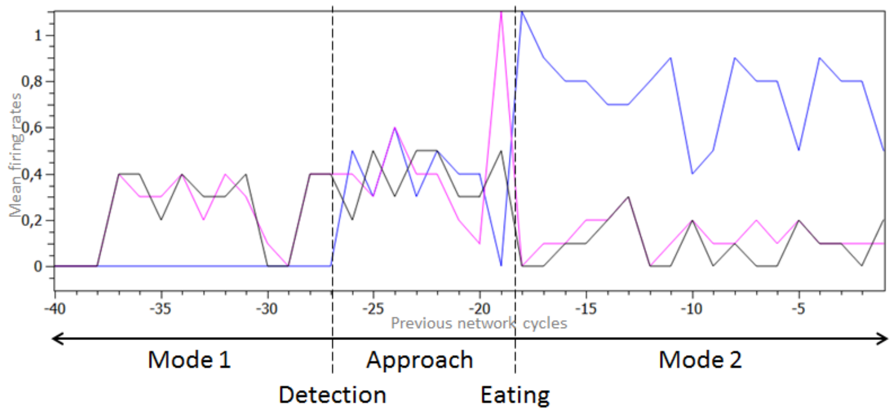
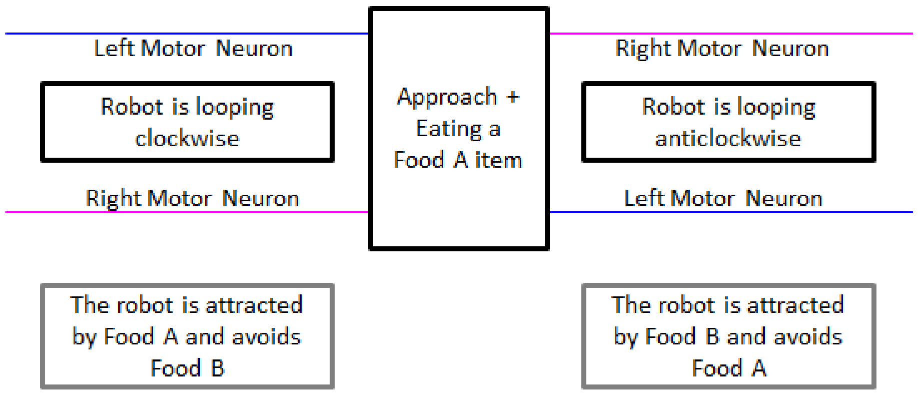


Figure 6.6: The mean firing rates show the two modes of `switch_IZHI_3`. The magenta curve corresponds to the right motor neuron, the blue one to the left motor neuron and the black one to the hidden neuron.

The two modes are characterised by different levels of potentials/mean firing rates of the motor neurons. Figure 6.7 summarizes the general transition between the two modes.



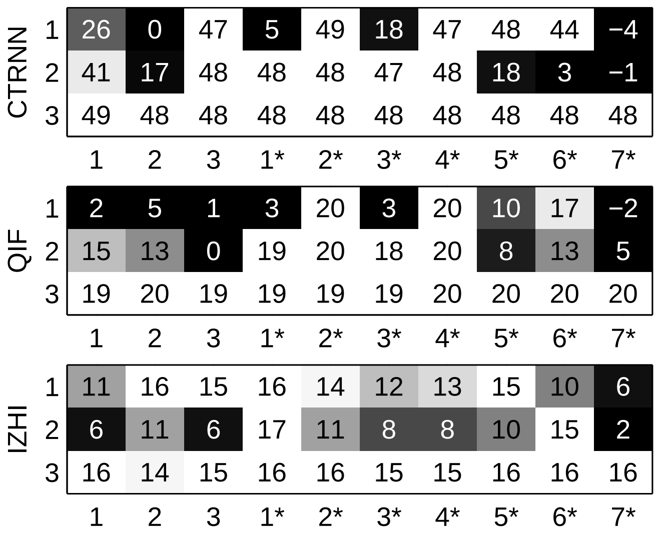
Left Motor Neuron
Robot is looping clockwise
Right Motor Neuron
Approach + Eating a Food A item
Right Motor Neuron
Robot is looping anticlockwise
Left Motor Neuron
The robot is attracted by Food A and avoids Food B
The robot is attracted by Food B and avoids Food A

Figure 6.7: Blue and magenta curves represent the potentials for the CTRNN controller and the mean firing rates for the spiking controllers.

6.3.2 Lesion experiments

To visualize the importance of each connection in the networks and to understand some of the behaviours, some *lesion experiments* as explained in 2.6 are conducted.

Following figure 6.8 shows the results of such *lesions experiments*. The fitness has been averaged over 200 simulations in order to reduce the variance.



CTRNN	1	2	3	1*	2*	3*	4*	5*	6*	7*
1	26	0	47	5	49	18	47	48	44	−4
2	41	17	48	48	48	47	48	18	3	−1
3	49	48	48	48	48	48	48	48	48	48

QIF	1	2	3	1*	2*	3*	4*	5*	6*	7*
1	2	5	1	3	20	3	20	10	17	−2
2	15	13	0	19	20	18	20	8	13	5
3	19	20	19	19	19	19	20	20	20	20

IZHI	1	2	3	1*	2*	3*	4*	5*	6*	7*
1	11	16	15	16	14	12	13	15	10	6
2	6	11	6	17	11	8	8	10	15	2
3	16	14	15	16	16	15	15	16	16	16

Figure 6.8: These lesions were conducted on the best 3-neuron controllers resulting from each evolution shown on 5.2. Neuron 1 corresponds to the left motor, neuron 2 to the right. Neuron 3 is an hidden neuron. Inputs 1* is the left food, 2* the right food and 3* the middle food sensor. Inputs 4* is the left poison, 5* the right poison and 6* the middle poison sensor. Input 7* is the state sensor.

The white third lines for the three controllers point out that connections **to** neuron 3, the *noisy neuron*, seem useless. Nevertheless the *noisy neuron* is crucial for the spiking controllers: for both QIF and IZHI controllers, the third column has some black cells. Its role is to provide some activity at the very beginning of the simulation. As explained before, a CTRNN controller don't need any noisy background current and therefore the third column of its lesion matrix is white.

Two lesions particularly affect the efficiency of the controllers: $1 \leftarrow 7^*$ and $2 \leftarrow 7^*$. They correspond to the state sensor connections to the motors. Table 6.9 logically shows that the connections weights are very high. The key point to notice is the fact that one weight is positive and the other one is negative for each controller. This explains the transition of the motor potentials or mean firing rates from figure 6.7.

Controller	weight of $1 \leftarrow 7^*$	weight of $2 \leftarrow 7^*$
`switch_CTRNN_3`	-0.98	0.85
`switch_QIF_3`	-0.94	0.90
`switch_IZHI_3`	0.98	-0.98

Figure 6.9: Weights of the connections from the state sensor to the motor neurons for the three best controllers.

The inversion of the motor mean firing rates/potentials is the key explanation of the behaviour inversion of the robots. In the two modes, sensors' influence on the motor neurons is unchanged. Let's for example consider the left A item sensor. In both modes, its action on motor MFR/potentials will be the same: when activated, the robot's left A item sensor increases the potential/MFR of the right motor neuron. However, the resulting motor MFR/potentials depend on the mode:

Mode 1:

- in this mode, the left motor neuron potential/MFR is greater than the right motor neuron one.
- when activated, the robot's left A item sensor increases the potential/MFR of the right motor neuron. The right motor neuron potential/MFR becomes greater than the left motor neuron one, forcing the robot to turn left towards the A item.

Mode 2:

- in this mode, the left motor neuron potential/MFR is lower than the right motor neuron one.
- when activated, the robot's left A item sensor increases the potential/MFR of the right motor neuron. The right motor neuron potential/MFR becomes even greater than the left motor neuron one, forcing the robot to turn even more on the right so much that it avoids the A item.

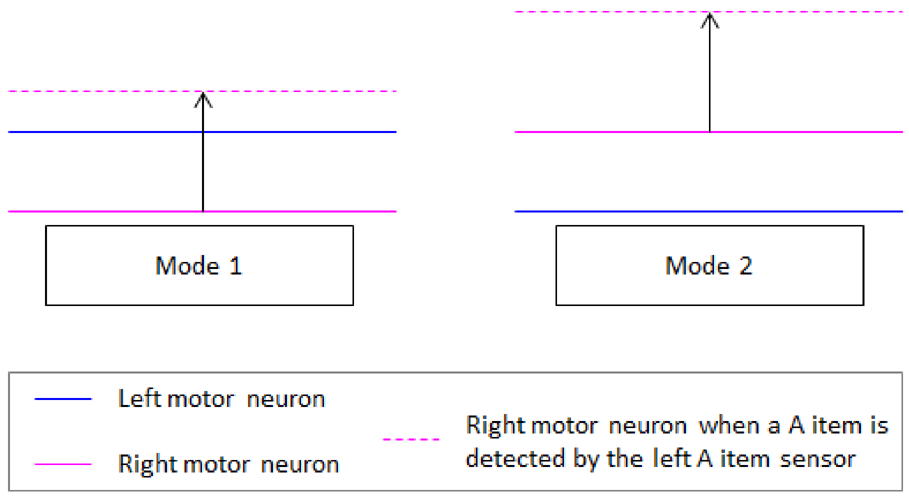

Mode 1
Mode 2
Left motor neuron
Right motor neuron
Right motor neuron when a A item is detected by the left A item sensor

Figure 6.10: The impact on the robot's behaviour of a left A item sensor input is radically different in the two modes.

The previous explanation was given for the left A item sensor but it is the same mechanism with the five others.

6.4 Conclusion

This experiment has permitted to design some very efficient controllers for the given task. We have seen that a complex behaviour can have very simple explanations and can be generated by a very simple controller of only 3 neurons. By only inverting the mean firing rates/potentials of the motor neurons, the controllers' behaviour regarding A and B items is inverted as well.

Once again Izhikevich's neuron controllers are the fastest to evolve and have the lowest deviation when several independent evolutions are run.
CTRNN controllers take longer to evolve and their variance among several independent evolutions is very high but some lucky evolutions have returned very efficient controllers.

Conclusion

Results

Results of this project are threefold:

- Through various animal-inspired experiments, some qualities and limits of spiking controlled evolutionary robotics have been assessed. Some successful very simple controllers have indeed been evolved to perform very well some sensors-reactive tasks (*Foraging*, *Diversifying diet*). For more complex experiments (*Exploration and Grazing*), spiking networks have exhibited some limits and with the exception of a few successful ones, most evolutions led to inefficient controllers.
  Nevertheless it returns that spiking neuro-controllers and especially Izhikevich's based ones are faster to evolve and less sensitive to initial genetic population than CTRNN controllers. These qualities may be interesting for engineering or industrial processes that require fast, reliable and safe ways to evolve robots' controllers.

- Some evolved controllers display interesting dynamics pointed out by the *lesions experiments*. The successful spiking QIF-controllers for the *Exploration and Grazing* experiment switch alternatively between two internal attractive modes that respectively control the robot's exploration strategy and the grazing one. Noisy background current reveals to be crucial to escape the grazing mode. It also permits to give room to some more random exploration strategies that are more efficient than simple lines or circles.

- Finally some interesting simple behaviour tactics have emerged from the animal-inspired experiments of this project. In the *Exploration and Grazing* experiment, some evolved robots display a converging spiral around a group of items in order to eat all of them. Other controllers display some periodic pirouettes to efficiently scan the world. In the *Diversifying diet* experiment, a very simple inverting mechanism has emerged: the only difference between the two modes is the inversion of the motor speeds.

Lessons learned

This project was an excellent opportunity to get introduced to different topics:

- robotics and especially evolutionary robotics,
- genetic algorithms,
- animats and biologically-inspired tasks,
- design of spiking neural controllers and all related issues,
- software design in C++ and Qt framework.

Future work

Further steps could be:

- Include learning activity in addition to evolution, possibly using a Spike Time-Dependent Plasticity (STDP) rule.
- Use more sophisticated genetic algorithms such as adaptive algorithms or NEAT.
- Multi-agent experiments.

Bibliography

[1] Valentino Braitenberg. Vehicles: Experiments in synthetic psychology, 1984. 5

[2] Stewart W. Wilson, J. Meyer. Animat. *Scholarpedia*, 4(5):1533, 2009. 5

[3] R. D. Beer. Biologically inspired robotics. *Scholarpedia*, 4(3):1531, 2009. 5

[4] R.D. Beer. Intelligence as adaptive behaviour: An experiment in computational neuroethology. 1990. 6

[5] R.D. Beer. On the dynamics of small continuous-time recurrent neural networks. 1995. 6

[6] D. Floreano J. Blynel. Exploring the t-maze: evolving learning-like robot behaviors using ctrnns. 2003. 6

[7] W.Maass. Networks of spiking neurons: the third generation of neural network models. *Australian Conference on Neural Networks*, 1996. 6

[8] Eytan Ruppin. Evolutionary autonomous agents: A neuroscience perspective. *Macmillan Magazines Ltd*, February 2002. 9

[9] D. Cliff I. Harvey, P. Husbands. Issues in evolutionary robotics. *The University of Sussex*, 1992. 9

[10] Kenneth Stanley. Efficient reinforcement learning through evolving neural network topologies. 2002. 9

[11] Dario Floreano and Claudio Mattiussi. Evolution of spiking neural controllers for autonomous vision-based robots. *Swiss Federal Institute of Technology (EPFL)*, 2001. 10

[12] Martin Colley Victor Callaghan Hani Hagras, Anthony Pounds-Cornish and Graham Clarke. Evolving spiking neural network controllers for autonomous robots. *University of Essex*, 2002. 10

[13] Eytan Ruppin Keren Saggie-Wexler, Alon Keinan. Neural processing of counting in evolved spiking and mcculloch-pitts agents. *School of Computer Science, Tel-Aviv University,*, 2004. 10

[14] Eytan Ruppin Ranit Aharonov-Barki, Tuvik Beker. Emergence of memory-driven command neurons in evolved artifial agents. *The Hebrew University of Jerusalem, Tel-Aviv University*, 2001. 10

[15] P.Husbands D.Cliff, I.Harvey. Evolving visually guided robots. 1992.

[16] Fabien Moutarde. A robot behavior-learning experiment using particle swarm optimization for training a neural-based animat. *Mines ParisTech*, 2008.

[17] Lefteris Doitsidis Andrew L. Nelson, Gregory J. Barlowb. Robotics and autonomous systems. *Androtics, Carnegie Mellon University, Technical University of Crete*, 2008.

[18] Murray Shanahan. Imperial college neurodynamics course notes. 2012.

A | Memorizing and diversifying diet

The experiment is the same as in 6.1 where the robot has to switch between one kind of item to the other one and so one. The only difference is the way the last sensor works. Now the sensor returns 0 most of the time except at the precise moment when the robot eats an item of any kind where the sensor returns 1. Therefore, the robot has now to figure out itself which next class item it has to eat. The very short time laps when the input equals 1 has to trigger the new mode of the robot and so on.

Figure 6.1 shows the genetic evolutions of the different 3-neuron controllers for the given task.

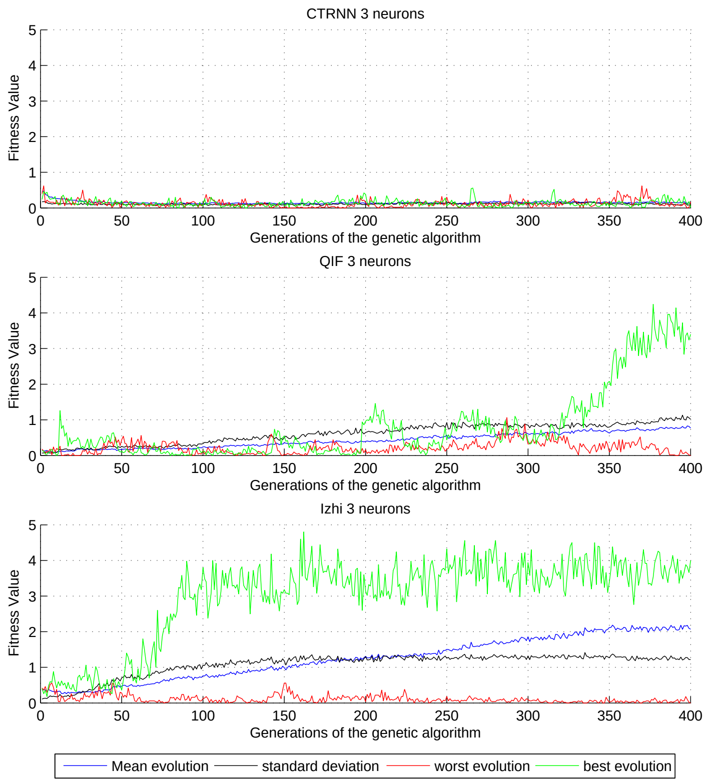


Figure A.1: Comparison of the evolutions of different controllers over 400 generations. For each figure, 50 evolutions have been run.

Evolutions fail to produce efficient CTRNN controllers. Spiking controllers are slightly better especially the IZHI-neuron ones.

Controllers with more neurons have been evolved but the results were similar.

B | Memorizing deterministic food patterns

In this experiment, the robot has no sensor. Its motion only relies on its own controller's states and dynamics to move. There is no poison item and the food distribution is deterministic. There is only one food item at a time on the world. When the robot eats it, it disappears and a new food item pops out at the next position of the deterministic pattern. Having food items appearing one after another forces the robot to follow the deterministic pattern in order to have the best fitness. The more food items the robot eats, the more it is rewarded.

Circles and lines have successfully been memorized.

Several complicated food patterns have been tested, such as describing a spiral (see figure B.1).

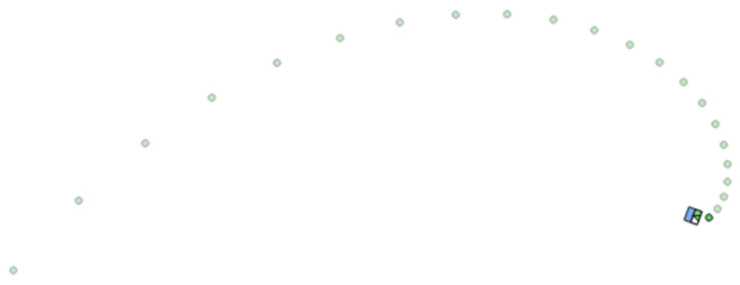


Figure B.1: A complicated deterministic distribution. Note that actually only one item is present on the map at time t. As soon it is eaten, the next one pops out.

Although a large number of parameters have been tested (up to 20 neurons), evolutions with the three studied neuron models have all failed to provide any good robot capable of describing the spiral. Increasing the controllers' size up to 20 neurons does not change anything. At best, controllers describe a circle tangent to the beginning of the spiral.

Describing a spiral only based on some neural network internal states and dynamics is much more challenging than a circle. A circle is periodic and is simply produced by two constant non-identical wheel velocities. A spiral might need some kind of counting process with some memorized firing patterns.

C | Complex behaviours and large neural networks

The reason why controllers from appendix A cannot memorize at least an approximation of the spiral is probably due to the lack of neurons. 20 neurons was the maximum tested (beyond, evolution takes too much time).

A few experiments indeed support this interpretation. For example, totally random initial controllers with a high number of neurons can yet display some very complex patterns. Figure C.1 shows the trail of a 150-IZHI-neuron controlled robot with no background current. The network is just "switched on" (see 2.3.4) at the beginning with the same current sent to all the neurons.

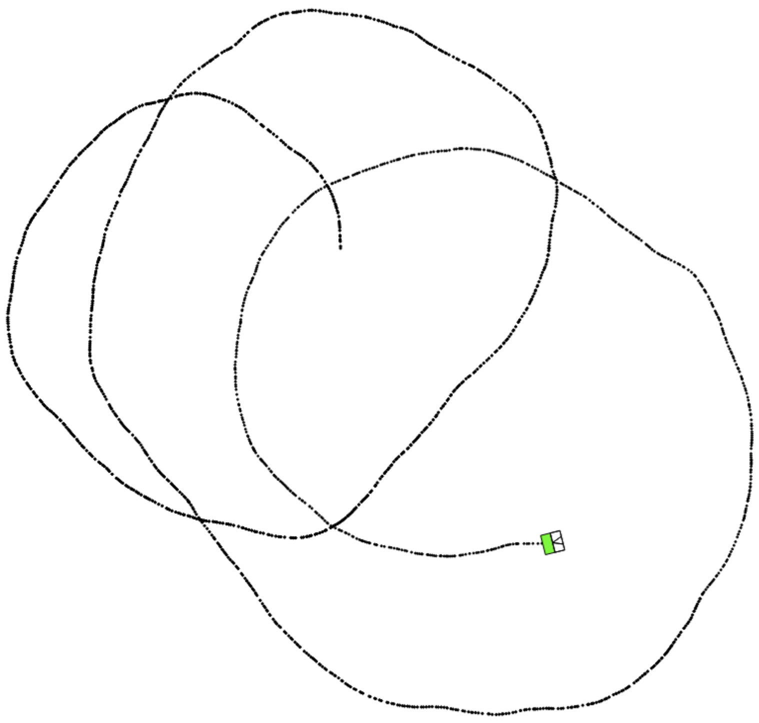


Figure C.1: Large neuro-controllers are potentially capable of memorizing complex patterns

The controller's activity is probably not periodic. These observations tend to suggest that some large enough controllers might **potentially** be able to memorize some complex patterns.