Bio-Inspired Artificial Intelligence

Prof. Giovanni Iacca
giovanni.iacca@unitn.it

Module 9–Lab Exercises

Introduction

Goal. The goal of this lab is to investigate, by means of agent-based simulations, an example of Evolutionary Robotics. In particular, we will focus on a maze navigation robot task, and see how Evolutionary Algorithms can be used to solve this problem.

Getting started. Download the file `09.Exercises.zip` from Moodle and unzip it. This lab continues the use of the *inspyred* framework for the Python programming language seen in the previous labs. If you did not participate in the previous labs, you may want to look those over first and then start this lab's exercises. Additionally, in this lab we will use a custom 2-D robot Java simulator for performing robotic experiments in a simulated environment.

Note once again that, unless otherwise specified, in this module's exercises we will use real-valued genotypes and that the aim of the algorithms will be to *minimize* the fitness function $f(\mathbf{x})$, i.e. lower values correspond to a better fitness!

Exercise 1

In this exercise we will perform an Evolutionary Robotics experiment to evolve the controller of a robot navigating a certain environment. This task is usually referred to as "maze navigation". In principle, this kind of experiments can be done in hardware, i.e. with physical robots and a physical arena (this kind of approach is usually called *embodied* evolution). While embodied evolution is a quite powerful approach, it also presents several challenges due to limited battery lifetime, hardware issues/faults, costs, time-consuming experiments, etc. Here, for simplicity we will perform similar experiments *in silico*, i.e. by means of agent-based simulations. Note however that this is actually the typical approach used in Evolutionary Robotics, where usually simulations are performed first, and then the simulated experiments are replicated *in materio* (i.e. with physical hardware), to assess the so-called "reality-gap" and validate the solution.

Here, we will perform simplified 2-D kinematic simulations[1] of a maze navigation task performed by an E-puck robot, a small ($\sim$ 7 cm diameter) 2-wheeled mobile robot developed at EPFL for research and education purposes[2].

[1] In other words, robots are represented as simple circles that can have inelastic collisions with obstacles. As such, the simulation does not model dynamic forces, such as inertia and friction.

[2] See `http://www.e-puck.org`

Two screenshots of the maze navigation simulator that we will use in this exercise are shown in Figure 1. The GUI allows the user to show/hide various information at runtime, i.e. the trajectory of each robot (with optional fading), the robot ID and distance to target, the range of the IR sensors (as well as the line-of-sight of each IR sensor) and the compass (indicating the front side of the robot). **Note that it is possible to tune the simulation speed by using the slider on the right side of the toolbar (you should do that if you don't want to wait for too long)**. Also, note that although the simulator shows several robots at the same time each robot performs the navigation task as if it was alone, so that in practice there is no collision/overlap with other robots. However robots are simulated together for simulation efficiency purposes. You can test the simulator, with default values and randomly initialized robot controllers, by running the script `runApplet.sh` that is available in the folder `Java2dRobotSim` (to do so, the first time you should make the script executable, by running `chmod a+x runApplet.sh`, and then launch the simulator by typing `./runApplet.sh`).

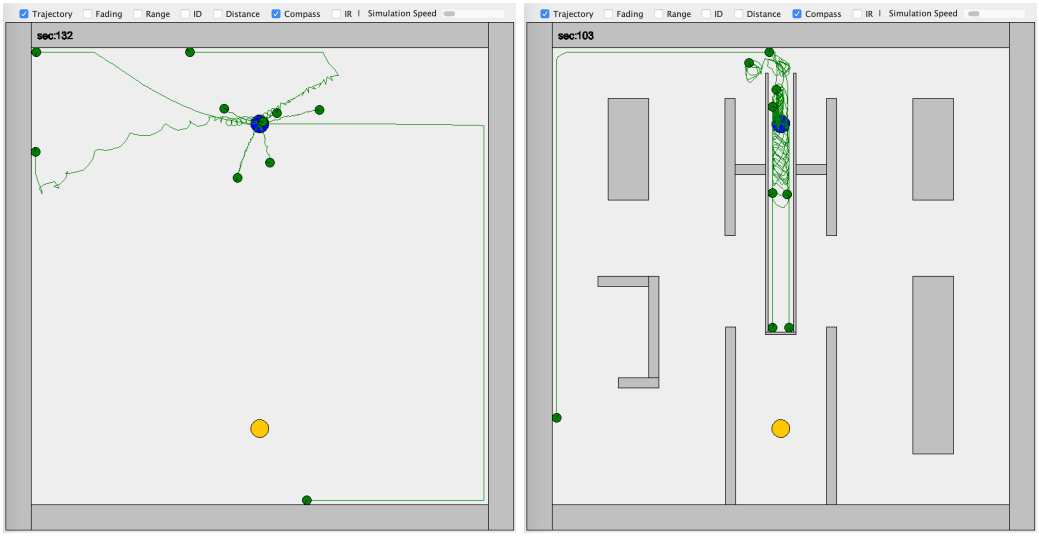

Figure 1: Screenshots of the maze navigation simulator: environment without obstacles (left); environment with obstacles and cul-de-sacs (right).

All the details of the simulator and the environment can be configured by means of a configuration file, see the `.txt` files in the folder `Java2dRobotSim/config`. Two configuration files are provided:

- `config2d_no_obstacles.txt`
- `config2d.txt`

The two files are pretty much the same, except for the section that describes the environment, contained within the tags `# -- obstacles -- #`. Obstacles (and walls) are described as rectangles, whose coordinates (in mm) are defined as `(leftX topY widthX widthY)`, while the size of the arena is defined by the parameters (`environmentWidth environmentHeigth`), also expressed in mm. The difference between the two scenarios is that the first is related to an environment without obstacles, see Figure 1 (left), while the second one refers to an environment that is purposely designed to have several obstacles, as well as "cul-de-sacs", i.e. areas where robots may easily get stuck, see Figure 1 (right). As you will see, the presence of these areas makes the task much more challenging from an evolutionary point of view.

In both experimental scenarios, the robot starts from an initial position (the blue circle), and its goal is to navigate the environment to reach the goal (the yellow circle), see Figure 1. The starting and target coordinates are described by the parameters (`startingX startingY`) and (`targetX targetY`), respectively.

As shown in Figure 2, by default each robot is controlled by a Feed Forward Neural Network (FFNN) that transforms the sensory inputs received from the different sensors of the E-puck into motor commands for the left and right wheels of the robot. As such, the FFNN is configured to have 10 input nodes (8 infrared sensors, distance & bearing to the target position), 4 hidden nodes, and 2 output nodes (speed of the left and right wheel). Different network architectures -including recurrent ones- can be used by changing the parameter `networkType` to one of the possible values {FFNN, ELMAN, JORDAN}[3]. All nodes use a logistic activation function (sigmoid), that can be changed by setting the parameter `activationFunction` to one of the possible values: {SIGMOID, TANH, CLIPPED_LINEAR, CLIPPED_LINEAR_01}[4].

The IR sensors have a range of 21 cm and -for simplicity- return a binary signal 1/0 if an obstacle is detected or not on the line-of-sight of each sensor. The distance sensor returns a value in $[0, d_{max}]$ (in mm) where d_{max} is pre-computed as the maximum distance in the given environment (length of the diagonal of the arena). The bearing sensor returns a value in $[-\pi, \pi]$. All inputs are rescaled in the range $[-1, 1]$ before feeding the network. The outputs of the network are rescaled from the codomain of the activation function ($[0, 1]$ in case of logistic function) to the range $[-0.129, 0.129]$ m/s, which are then translated into translational and rotational speed (by means of a simple unicycle kinematic model). Gaussian noise can be added to the network's outputs to simulate noise on the motors of the robot's wheels (parameter `noiseLevel`, by default set to 0). Each simulation lasts, by default, 5 minutes, with an update frequency of 4 Hz (i.e., 1200 steps with sampling time $dt = 0.25$ s).

Note that, in case of a FFNN with one hidden layer and bias (as that shown in Figure 2), the total number of synaptic weights of the network is given by:

$$n_{weights} = (n_{inputs} + 1) \cdot n_{hidden} + (n_{hidden} + 1) \cdot n_{outputs}$$

In case of a recurrent network, this number must be increased by $n_{hidden} \cdot n_{hidden}$ in the case of a Elman network, and $n_{outputs} \cdot n_{outputs}$ in the case of a Jordan network (in both cases with one hidden layer).

If no hidden layer is present (either the network is recurrent or not), then the total number of weights is:

$$n_{weights} = (n_{inputs} + 1) \cdot n_{outputs}$$

where n_{inputs}, n_{hidden}, and $n_{outputs}$ indicate, respectively, the number of input, hidden and output nodes. Thus, with $n_{inputs} = 10$, $n_{hidden} = 4$, and $n_{outputs} = 2$ the FFNN has 54 weights. By default, all the network's weights range in $[-3, 3]$.

To start the experiments, from a command prompt run[5]:

```
$>python exercise_maze.py
```

[3] `https://en.wikipedia.org/wiki/Recurrent_neural_network#Elman_networks_and_Jordan_networks`

[4] `https://github.com/encog/encog-java-core/tree/master/src/main/java/org/encog/engine/network/activation`

[5] For all the exercises in this lab you may follow the name of the `.py` file with an integer value, which will serve as the seed for the pseudo-random number generator. This will allow you to reproduce your results. Also, please note that in this document `$>` represents your command prompt, do not re-type these symbols.

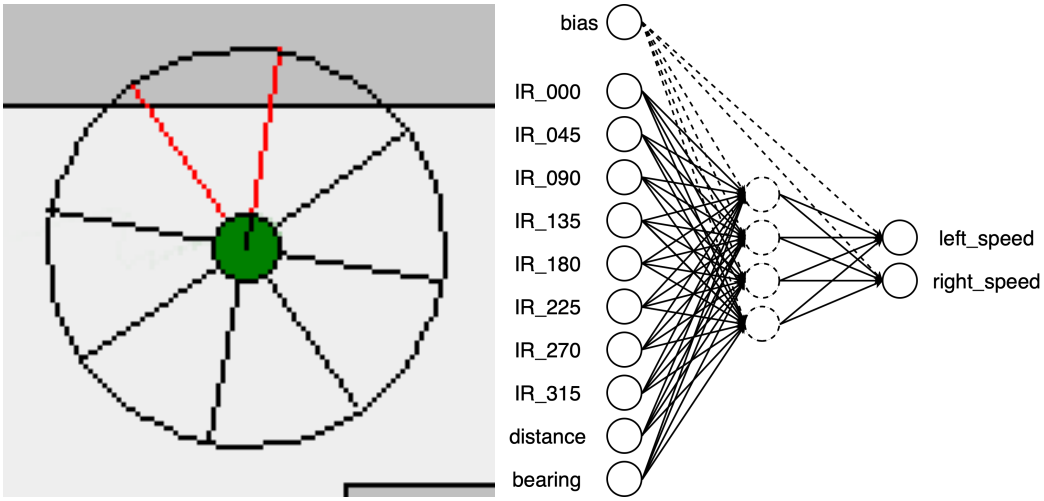

Figure 2: A graphic representation of the robot (left): the compass shows the front side of the robot, while each line within the IR range represents the line-of-sight of each IR sensor (by default, 8 IR sensors are placed around the robot, equally spaced every 45°). Each line-of-sight turns red when the IR detects an obstacle (wall). The corresponding Neural Network controller (right): in this example the Neural Network uses as inputs the robot's distance and bearing (orientation) towards the target position and 8 IR sensors, with 4 hidden nodes. A bias node is also used as input to the hidden and output nodes. The output nodes are the robot's left and right wheel speed, which are then translated into translational and rotational speed.

This script will perform a maze navigation Evolutionary Robotics experiment in the first scenario (environment without obstacles). This experiment uses an Evolutionary Algorithm to evolve the synaptic weights of the described Neural Network. The synaptic weights, which represent the genes of the individuals, are coded using real-coded values (floating point). A population of such individuals is evolved, using tournament selection, Gaussian mutation, n-point crossover, and elitism (see `exercise_maze.py` for more details about the algorithmic configuration and parametrization). The genotypes of the first generation are initialized randomly in the range [-3,3]. Elitism is used to make sure that good solutions are not lost because of mutation or crossover, while the rest of the population is generated by means of the genetic operators. At each generation, the genotypes (i.e., the encoded controllers) generated by the EA are sent via a text file to the simulator (one batch per generation), which then translates the genotypes it receives into a set of Neural Network controllers, evaluates the controllers and sends the fitness back via text file to the EA. At the end of the evolutionary process, the script will generate a results folder named as the seed of the experiment, containing the genotypes of each generation (files `candidates_*.txt`), as well as the best evolved Neural Network (file `best_candidate.txt`). If the `display` flag is set to true, the best individual run simulation will be shown in the GUI[6].

You can "replay" each generation, or just the best candidate, by running this command:

```
$>python run_candidates.py config2d_no_obstacles.txt SEED/candidates_49.txt
```

or

[6]The GUI is shown via the `appletviewer` JDK tool. This tool has been deprecated however starting from JDK 11. In order to use it, you may need to install a JDK version < 11 and change the last lines of the `Java2dRobotSim/runApplet.sh` script accordingly.

```
$>python run_candidates.py config2d_no_obstacles.txt SEED/best_candidate.txt
```

where `SEED` is the seed used in the simulation (e.g., 1542609572).

- Design and implement a fitness function that would allow the robot to *reach the target as fast as possible*. To do so, you can modify the method `evaluator` of the class `RobotEvaluator` coded in the script `exercise_maze.py`. Five metrics are computed by the simulator and can be used to devise different fitness functions, namely:
  - `distanceToTarget`: the distance to target at the end of the simulation (default);
  - `pathLength`: the total distance traveled by the robot;
  - `noOfTimestepsWithCollisions`: the no. of timesteps when the robot had a collision with an obstacle or a wall (ranging in [0,`nrTimeStepsGen`]);
  - `timestepToReachTarget`: the no. of timesteps needed for the robot to reach the target (ranging in [0,`nrTimeStepsGen`]);
  - `timestepsOnTarget`: the no. of timesteps spent by the robot on the target (ranging in [0,`nrTimeStepsGen`]);

  By using one or more of these five quantities properly into one single fitness value[7] (see the variable `fitness_i`), different robot behaviors can be evolved. Please note that, by default, the `RobotEvaluator` is formulated as a *minimization* problem (see the flag `self.maximize = False`). However, if you deem it more appropriate, you can change the flag to turn it into a *maximization* problem. **NOTE**: While you design the fitness function, be careful about divisions by zero and make sure that when you divide one integer variable by another you cast your variables to `float`, to avoid unexpected behaviors due to integer divisions.
- Is the Evolutionary Algorithm able to evolve a Neural Network controller that can reach the target? What kind of motion strategy does it use?
- What is the minimum-complexity Neural Network controller that you can think of?
  Hint: think about the necessity of using all the available sensor inputs in this case, and if any of them can be discarded (see the `config2d_no_obstacles.txt` configuration file to disable inputs). Also, consider reducing the no. if hidden nodes and test different network configurations to identify the simplest controller.
- By looking at the weights of the best evolved Neural Network in the simplest case you just found, can you try to make sense of the controller functioning? (Note that weights appear ordered by layer and, for each layer, by node, the bias weight being the last one.)

Exercise 2

Let us consider now the second scenario (environment with obstacles). To do so, change in the script `exercise_maze.py` the parameter `config_file` to be `config2d.txt`.

First of all, it is interesting to see if the controller evolved in the previous exercise is able to generalize its functioning to this case.

[7] Note that, in principle, also a multi-objective optimization approach could be used in this case. However, in this lab we will focus only on a single-objective approach.

- Take the best Neural Network evolved in the previous exercise and run it in the new scenario, by changing the configuration file as argument to the script `run_candidates.py`:

```
$>python run_candidates.py config2d.txt SEED/best_candidate.txt
```

- What happens in this case? Is the best Neural Network evolved in the previous exercise able to generalize to this new environment? Why?

Consider now running a new evolutionary process from scratch, to evolve a controller specific for this new, somehow harder environment.

- Is the same fitness function you designed in the previous exercise able to guide the evolutionary search also in this case? If not, try to change it appropriately (**Hint**: you may want to embed in the fitness value multiple metrics, and use some weights if needed). Does the best individual evolved in this scenario generalize to the first scenario?
- Try to change the starting and target positions by changing the parameters (`startingX startingY`) and (`targetX targetY`) (make sure that the starting and target positions do not overlap with obstacles!), and see if the best Neural Network you just obtained is able to generalize w.r.t. the starting/target positions.
- Try to make the problem even harder, in the attempt to find a controller that is able to drive the robot to the target *without touching any walls*. What kind of fitness function could you use in this case? You may also want to consider changing the configuration/parameters of the EA (e.g. larger population, higher number of generations, different mutation/crossover parameters) and/or the architecture/parameters of the Neural Network (e.g. use a recurrent network and/or add more hidden nodes).

Instructions and questions

Concisely note down your observations from the previous exercises (follow the bullet points) and think about the following questions.

- What do you think it could change between a simulated and a real-world experiment in the case of a maze navigation task?
- Can you think of some possible applications where a maze navigation robot task could be used? Why would it make sense to use Swarm/Evolutionary Robotics in those cases?