Bio-inspired Adaptive Machines
Evolution of Neural Controllers
Laboratory exercises

Sara Mitri and Peter Dürr

April 1, 2008

1 Goal

The goal of this laboratory is to evolve a neural controller for an e-puck robot which has to navigate in an environment as fast as possible while avoiding obstacles (for example, walls). Evolution is done in the Webots simulator and tests of the best evolved controller can be done in simulation or on the real e-puck.

By the end of this laboratory you should have learned something about :

- how to design fitness functions.
- how the parameters of the genetic algorithm (GA) influence evolution.
- how to test whether or not the solution found through evolution can be generalized.
- why evolution is so cool (look out for unthought of strategies, which, by the way, you didn't even have to code).
- why transfers to reality are a pain ;)

2 Theory

2.1 Neural Controller

The e-puck robot is controlled by a neural network which transforms the sensory IR inputs received from the sensors of the e-puck into motor commands

for the left and right wheels of the robot as shown in figure 1. Inputs are scaled to fit the range [0,1] and the neuronal transfer function is chosen to be a hyperbolic tangent ($tanh$).

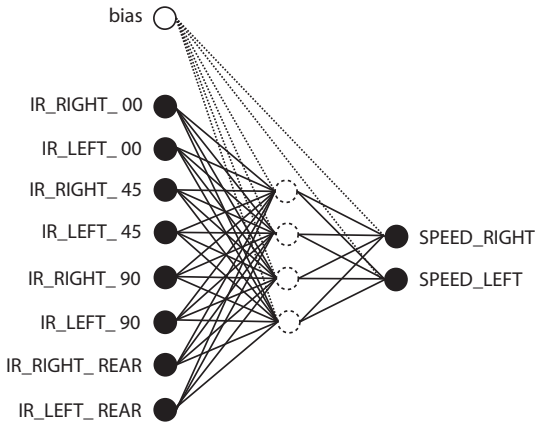

Figure 1: E-puck's neural controller.

2.2 Genetic Algorithm

A genetic algorithm is used to evolve the synaptic weights of the described neural controller. The synaptic weights, which represent the genes of the individuals, are coded using one float value. When put together, the genes form a genome.

A population of individuals is evolved using either roulette-wheel selection or rank-based truncation selection, one-point crossover, weight mutation and elitism. The genomes of the first generation are initialized randomly in the range [-1, 1]. Each individual is evaluated based on the fitness function defined in the *compute_fitness()* function defined in *epuck_drive.c*. After ranking the individuals according to their measured fitness values, the top individuals are copied to the new population (elitism). This allows evolution to make sure that good solutions are not destroyed because of mutation or crossover. The remaining population is generated from the crossover of two randomly paired individuals within the first ranks. One point crossover is applied to each pair with a certain probability and each gene is then mutated with a certain probability.

3 Experiments

Download the evoNN.zip file from the course website, unzip it and double click on *e-puck/worlds/evolution.wbt*, which you should find where you saved the zip file. Then, in the Webots C editor (do not use Microsoft Visual Studio), open *controllers/evolution/evolution.c*, *controllers/evolution/evolution.h*, *controllers/epuck_drive/epuck_drive.c* and *controllers/epuck_drive/epuck_drive.h*.

evolution contains the code for the evolutionary algorithm, whereas *epuck_drive* contains the neural network and the controller that determines what the robot will do. At each generation, *evolution* sends the genomes to *epuck_drive* one by one. *epuck_drive* then translates the genome it receives into a neural network controller, evaluates the controller and sends its fitness back to *evolution*.

The code can be compiled by clicking on the compile button at the top of the screen (the wheel-like icon). You must compile *evolution.c* and *epuck_drive.c* separately.

The neural controller can be configured by modifying the following parameters in *epuck_drive.h*.

NB_INPUTS Number of inputs to the neural network. Possible values are: 2 (front IR senors), 4:(front IR sensors), 6 (front and side IR sensors) and 8 (all IR sensors).

NB_HIDDEN_NEURONS Number of hidden neurons. Normally no hidden neurons are required, however, feel free to play around with this parameter.

NB_OUTPUTS Number of outputs. For this exercise, the number of outputs is always equal to two.

If you change the number of inputs and outputs of the neural controller (in *epuck_drive.h*), you also need to modify the "number of genes" parameter in *evolution.h*.

NB_GENES Number of genes corresponding to the total number of neural network weights controlling the robot (lines in Fig. 1). This can be calculated as NB_INPUTS * NB_OUTPUTS + NB_INPUTS * NB_HIDDEN_NEURONS + NB_HIDDEN_NEURONS + NB_HIDDEN_NEURONS * NB_OUTPUTS + NB_OUTPUTS.

The genetic algorithm can be configured by modifying the following parameters in *evolution.h*.

POP_SIZE Number of individuals per generation.

MUTATION_PROBABILITY Probability of mutating each weight in a genome.

MUTATION_SIGMA Determines the extent to which a value will be mutated (i.e., how far from its current value).

ELITISM_RATIO Ratio of best individuals which are copied to the next generation without modification.

CROSSOVER_PROBABILITY Probability with which crossover occurs.

ROULETTE_WHEEL Choice of selection method. Set to 1 for roulette wheel selection (individuals are selected proportional to their fitness) or 0 for rank-based truncation selection (select best x% of individuals).

3.1 Fitness functions

Design and implement a fitness function which would allow the robot to navigate in the arena as fast as possible and without touching any walls. To do so, modify the function *compute_fitness()* in *epuck_drive.c*. All the elements that you can use to define the fitness are presented in the comments (Note: don't remove the comments, just use the elements in the list to design your fitness function). Once you have implemented and compiled your function, reset the e-puck world (two-round-arrow icon at the top of the viewer) and press the play button. To go faster you can use the FAST button (>>) (Note: If you close the window showing the e-puck's sensor values, the simulation will go much faster). To view the fitness graph of your evolution, run *e-puck/data/plot_fitness.m* in Matlab. Always save your fitness graphs since they will get overridden by future evolutions.
Questions:

- Did you get the desired behaviour with your fitness function?
- What strategies did you observe during the evolution and why were they good/bad in terms of fitness?
- What is the most implicit fitness function you can find?
- Do you understand the fitness graph? Does the evolution converge (i.e., stop getting better)?

3.2 GA Parameters

Re-run the experiment, each time changing a single one of these parameters (or others):

- POP_SIZE 50
- MUTATION_PROBABILITY 0.8
- ROULETTE_WHEEL 1

Question:

- For each evolution, have a look at the fitness graph, do you see a difference with respect to your initial evolution?

3.3 Generalization

We will now test the best individual obtained through evolution. To do so, set the EVOLVING variable in *evolution.h* to '0', compile, reset the simulator and press play.

Questions:

- When you move your robot (SHIFT+mouse) to another area of the arena (close to a wall for example), does it still work?
- When you add obstacles in the environment, does your controller still work? To add obstacles:
  - Select the last node in the scene tree window and click on the insert after button.
  - Choose a Solid node.
  - Select the newly created node, click on its children field and add a Shape (using the colourful object icon at the top of the window), then an Appearance to the Shape and a Material to the Appearance.
  - Now create a Box node in the geometry field of the Shape and set its size to [0.1 0.1 0.1].
  - Create a Box boundingObject in the Solid node with the same size as before.
  - Move the object around with (SHIFT+mouse).

  - Create more obstacles with copy/paste.

- If your controller didn't generalize to these two tests, what could you do to fix the problem?

3.4 Transfer to the Real Robot

Call an assistant when you have found a good controller which you wish to test on a real e-puck.