# Modern Robots: Evolutionary Robotics

Programming Assignment 5 of 10*

## Description

In this assignment you will add your neural network to the robot and you will use the simple hill climber to make the robot move as far as possible.

## Tasks

1. Copy all your code for assignment 4 to the folder you will use for assignment 5. Make sure to keep a working backup of assignment 4.

2. Extract `hw_5_code`, and move `Fit_RobotDistance.hpp` to your assignment 5 folder.

3. To use our neural network as a controller, it is convenient if our neural network defines its inputs and outputs. Open `Ind_NeuralNetwork.hpp` and add the following attributes:

```
size_t _nbOfInputs;
size_t _nbOfOutputs;
```

Now add getters and setters for these attributes. Lastly, add the following function:

```
double getOutputValue(size_t neuronIndex)
```

This function should return the activation value of the output neuron at the `neuronIndex`. For this assignment we will assume that the first `_nbOfInputs` are input neurons, and the `_nbOfOutputs` neurons immediately following those neurons are the output neurons. As such, `getOutputValue` should return the value of the neuron at index: `neuronIndex + _nbOfInputs`.

4. Because we are evolving the controller for a robot, additional neurons and connections can be helpful. Add the following two attributes to NeuralNetwork, and implement the appropriate getters and setters.

```
double _addNeuronMutRate;
double _addConnectionMutRate;
```

*Original material was graciously provided by Josh Bongard. Jeff Clune slightly modified it. Joost Huizinga heavily modified it.

Now extend the mutate function such that there is a `_addNeuronMutRate` chance that a neuron is added, and a `_addConnectionMutRate` chance that a connection is added. To add a connection, simply select a random source neuron and a random target neuron and add a connection between these neurons, with a weight chosen uniform randomly from the range defined by the minimum and maximum values.

New neurons will be 'spliced' onto existing connections. To add a neuron, randomly select a connection and set it to zero (effectively removing that connection). Now add a new neuron to the network, and connect it such that the source neuron of the selected connection becomes its source, and the target neuron becomes its target (this means that, whenever you add a neuron, you also add two connections). The weight of the connection from the source neuron to the new neuron should be 1. The weight of the connection from the new neuron to the target neuron should be equal to the weight of the selected connection.

5. Open `Ind_TableRobot.hpp` and implement the `step` function. The `step` function should execute the following steps:

- Set the inputs of the neural network to be 1 where the touch sensors report a collision, and set them to 0 where the touch sensors do not detect a collision. Call `m_sensors[i]->getValue()` to get the state of a particular touch sensor, where a value of `True` indicates a collision, and `False` indicates no collision.
- Step the network once.
- Set the desired angles of the actuators of the robot according to the output values of the neural network. Scale the output of the neural network to be desired angles in $[-45, 45]$ with `networkOutput * 45`. When you have calculated the desired angle of an actuator, you can set the motor command with `m_motorCommands[i] = desiredAngle`, where `i` is the index of the actuator.

6. Verify that the neural network works correctly. Create a neural network in `main.cpp` with `JOINT_COUNT + SENSOR_COUNT` neurons. Use the getter and setters functions created above to set `_nbOfInputs` to be equal to `SENSOR_COUNT`, and `_nbOfOutputs` to be equal to `JOINT_COUNT`. Create connections from each input neuron to each output neuron. Let the minimum weights and maximum weights be -3 and 3, respectively. Randomize the network. Now set the network as the controller of a robot and run the simulation with:

```
typedef Simulator<TableRobot> sim_t;
sim_t demoApp;
demoApp.initPhysics();
demoApp.getDynamicsWorld()->setDebugDrawer(&gDebugDrawer);
demoApp.buildRobot()->setNeuralNetwork(initialNetwork);
glutmain(argc, argv,640,480,"Simulator",&demoApp);
```

You should now see the robot make a few movements. Make a screen shot after the robot has moved some distance from its original location and insert the resulting figure into your document. It should look like figure 1.

7. Now you need to create a fitness function appropriate for the robot. Open `Fit_RobotDistance.hpp` and implement the evaluate function such that the fitness of the robot is equal to the distance it has moved from the starting location after 1000 time-steps (call `_simulator->step()` to run the simulator for one time step without visuals). Your evaluate function should execute the following steps:

- Reset all activation functions of the network with: `individual.reset()`. Make sure that you implemented this method correctly by verifying that all activation values are indeed zero after calling this method.

- Delete any old robots with: `_simulator->deleteRobots()`.
- Create a new robot with: `_simulator->buildRobot()`.
- Set the individual to be evaluated as the neural network of this new robot.
- Run the simulator for 1000 steps.
- Set the distance the individual has moved to be its fitness. Distance can be easily calculated with: `robot->getPosition().distance(btVector3(0,0,0))`.

8. Comment out your test code in `main.cpp`, and instead create a hill climber that takes a neural network as an individual and the `Fit_RobotDistance.hpp` as a fitness function, and run the hill climber for 500 generations (the mutation rates for connections and biases should be 0.05, the add neuron and connection mutation rates should be 0.2). Do not forget that, in order to create an instance of the robot fitness function, you need to supply it with a pointer to the simulator. To do so, use the following code:

```
typedef IndivFitness<NeuralNetwork<> > ind_t;
typedef Simulator<TableRobot> sim_t;
typedef RobotDistanceFitness<sim_t, ind_t> fit_t;

sim_t demoApp;
demoApp.initPhysics();
fit_t robotDistanceFitnessFunction(&demoApp);
```

Write the fitness over time to a file and plot it with the `plotLine.py` function from homework 1. Add the plot to your document, it should look like figure 2. Make sure your robot achieves a fitness of 18 or higher.

You should also visualize the final network. If everything is implemented properly, you should be able to write the network to a file as follows:

```
ind_t best = hillClimber.getParent();
std::ofstream networkFile("hw5_final_network.scv");
networkFile << best;
networkFile.close();
```

Visualize this network with the `plotNetwork.py` Python script provided with this assignment. The figure should look like figure 3.

Lastly, run the final parent with visuals. To do so, add the following code as the last lines of your `main.cpp` (these should be your last lines of code because `glutmain` never returns):

```
demoApp.deleteRobots();
demoApp.getDynamicsWorld()->setDebugDrawer(&gDebugDrawer);
demoApp.buildRobot()->setNeuralNetwork(hillClimber.getParent());
glutmain(argc, argv,640,480,"Simulator",&demoApp);
```

Make a screen shot when the robot has moved sufficiently far from the starting position. The resulting figure should look like figure 4.

## Deliverables

A pdf document containing the figures resembling figures 1, 2, 3, and 4, and your code.

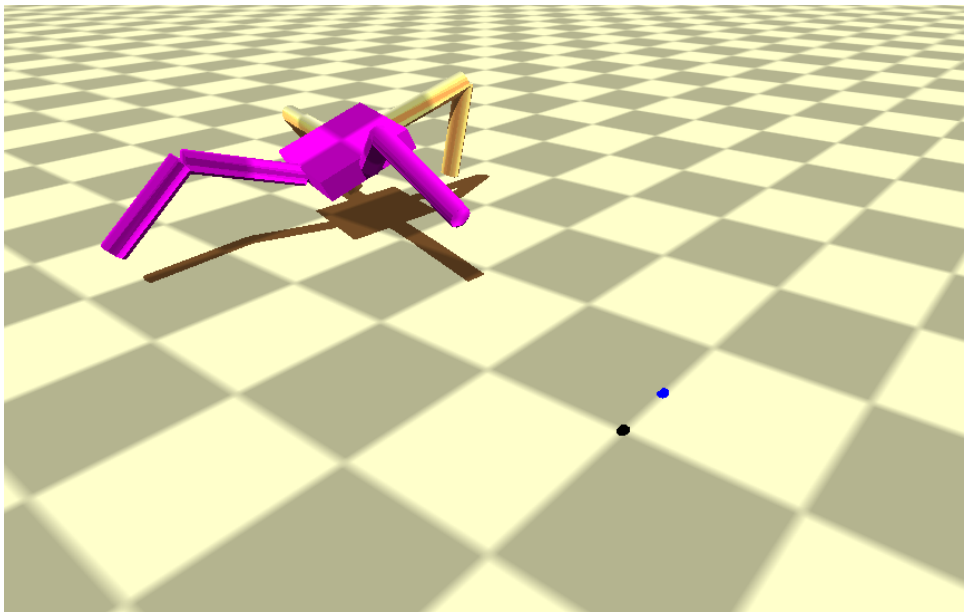

Figure 1: Commands from a random neural network can cause the robot to move some distance from the starting location.

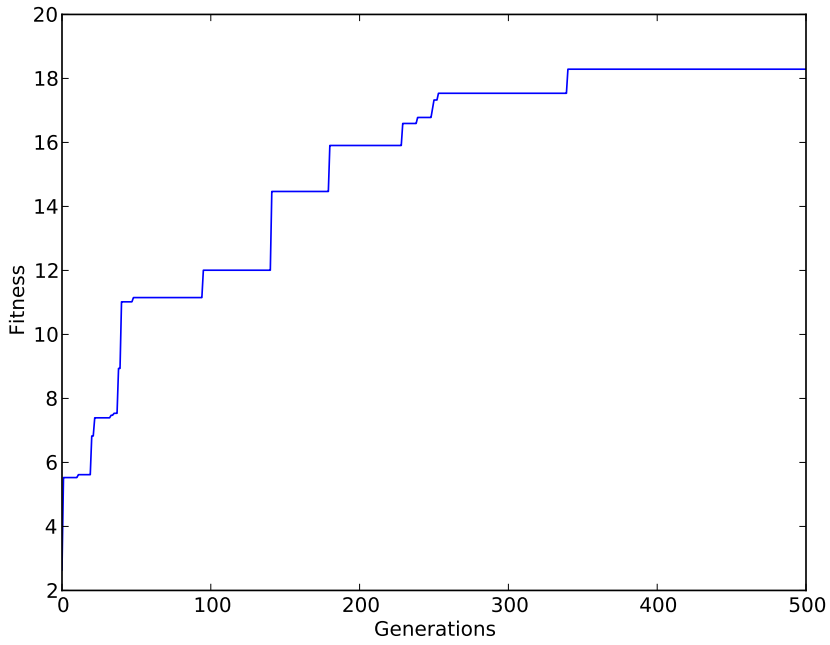


Figure 2: When evolved with a simple hill climber your robot should be able to move at least 18 units away from the starting location (re-run your program with a different seed if this does not happen for you).

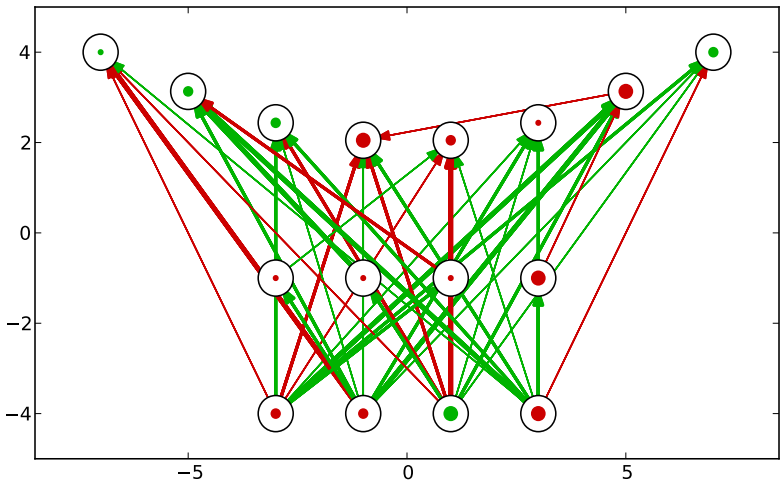


Figure 3: The final network. Note that several nodes and connections have been added.

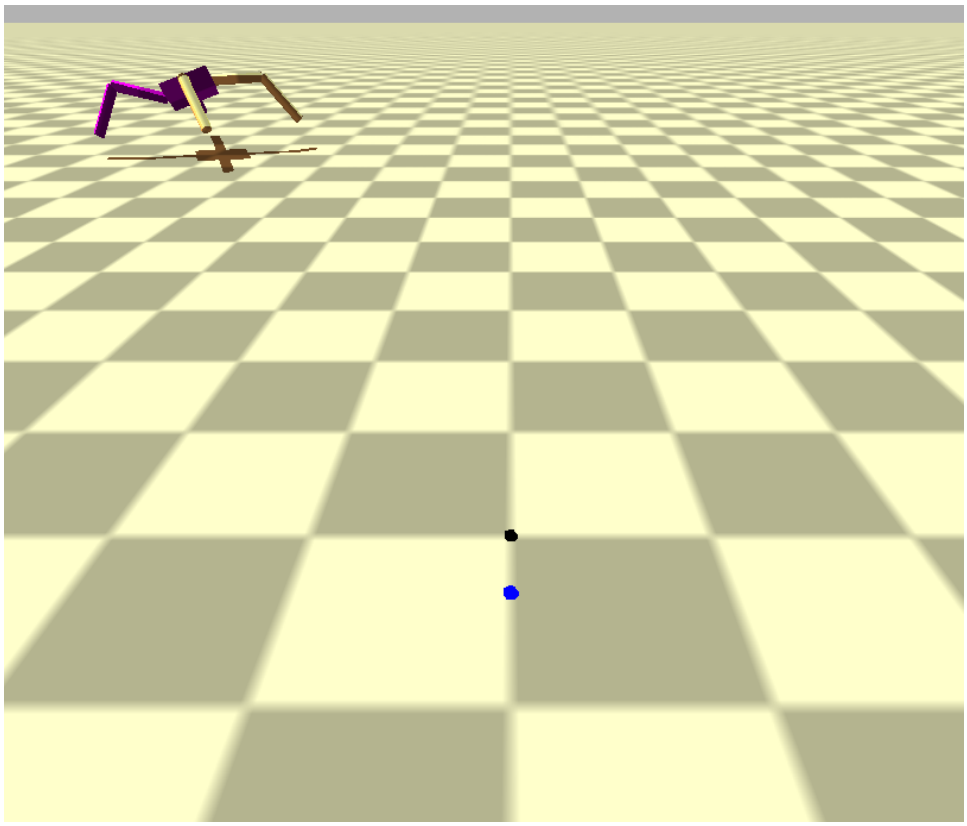

Figure 4: When viewed in the simulator, the evolved neural network should be able to move the robot much farther away from the starting location.