Modern Robots: Evolutionary Robotics

Programming Assignment 3 of 10*

Description

In this assignment you will apply the hillclimber you developed in assignment 1 to the artificial neural network (ANN) you developed in assignment 2.

Tasks

1. Copy all your code for assignment 2 to the folder you will use for assignment 3. Make sure to keep a working backup of assignment 2.

2. Extract `hw_3_code`, and move `Fit_LastRow.hpp` and `Fit_Checkerboard.hpp` to your assignment 3 folder.

3. Comment out, or delete, all the assignment 2 code in `main.cpp` (but remember to keep a backup).

4. First you will apply the `Hillclimber` to evolve a fully connected neural network with 10 neurons to match a target pattern after 10 steps (where all neurons are initialized with 0.5). The target pattern will be a vector of length 10, where each even index holds a 1, and each odd index holds a -1. Open `Fit_LastRow.hpp` and implement a fitness function in the `evaluate` method, such that the fitness of a network is equal to one minus the normalized Manhattan distance between the target vector and the vector of activation values of the network after 10 steps (see below). Remember that you are allowed to add functions to your neural network, such as a function to initialize all neurons with a specific activation value.

Given two vectors, $\mathbf{a} = [a_1, a_2, ..., a_n]^t$ and $\mathbf{b} = [b_1, b_2, ..., b_n]^t$, the Manhattan distance between these vectors is defined as:

$$\text{Man}(\mathbf{a}, \mathbf{b}) = \sum_{i=1}^{n} (\text{Abs}(a_i - b_i)) \tag{1}$$

In the context of fitness functions, normalization refers to ensuring that the range of the fitness function becomes $[0, 1]$. This means that the highest possible value of the fitness function should be 1 and the lowest possible value should be 0. Clearly, the smallest possible Manhattan distance between two vectors is 0. Because the activation value of any neurons is guaranteed to lie between -1 and 1, and the most extreme values in our target vector are also -1 and 1, the maximum absolute difference between any two positions in

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

the target vector and the activation values is 2. This means that the maximum Manhattan distance between the target vector and the activation values, and thus our normalizing constant, is 20. As such, if x_{10} is a vector containing the activation values of individual x after 10 steps, and v is the target vector, then the fitness function is defined as:

$$f(x) = 1 - \frac{\text{Man}(x_{10}, v)}{20} \tag{2}$$

In summary, the following steps should be performed by your evaluate function:

- Set the activation values of all neurons to 0.5.
- Step the network 10 times.
- Calculate the fitness of the network according to equation 2.
- Set the fitness of the network by calling its `setFitness` method (see assignment 1 for details).

5. Before the neural network can be evolved, you will need to implement a `mutate` method for the neural network. To have finer control over the evolutionary process we also want to add mutation rate parameters to this class. Add the following members to the `NeuralNetwork` class, and initialize them in the constructor:

```
double _weightMutRate;
double _neuronMutRate;
```

Also add setter methods to the class so you can set these variables. Then, create a mutate function in `Ind_NeuralNetwork.hpp` as a member of the `NeuralNetwork` class, and make sure that each connection has a probability of `_weightMutRate` to be mutated, and each bias has a probability of `_neuronMutRate` to be mutated. When a weight or bias is mutated, add a random number drawn from a Gaussian distribution with a mean of 0 and a standard deviation of 1 (use `randGaussian()` in `Misc_Random.hpp`) to the current value. If the weight or bias would fall outside the [`_minWeight`, `_maxWeight`] range due to the mutation, clip the value so it becomes equal to `_minWeight` or `_maxWeight`, whichever is closer to the current value. For this assignment the mutation rate for both the weights and the bias will be 0.05.

6. Use the `Hillclimber` to evolve a neural network for 1000 generations (see assignment 1 for how to use the `Hillclimber`). Make sure that the initial neural network is fully connected and has all its weights and biases randomized. To see the effect of our hillclimber, run the network for 10 steps, with initial values of 0.5 for each neuron, before and after running the `Hillclimber`, and write the network activation for both cases to a file. Also write the fitness over time to a file. Plot the network activation files with `plotMatrix.py`, plot the fitness over time with `plotLine.py`, and add the resulting figures to your document. Your figures of the initial activation, final activation, and fitness over time should look like figures 1a, 1b and 1c, respectively. Note: the final activation may not be perfectly alternating if the fitness of that run did not get to a fitness of 1; that is fine if this is the case.

7. Second we will evolve a neural network to maximize the difference between neuron activations, both between 'adjacent' neurons, and between the current activation of a neuron, and the activation of that same neuron at the previous time-step. To maximize the difference between 'adjacent' neurons, calculate the following sum:

$$f_1 = \sum_{t=1}^{T} \sum_{i=1}^{n-1} \left| \frac{activation(neuron_i, t) - activation(neuron_{i+1}, t)}{2T \cdot (n-1)} \right| \tag{3}$$

where i indicates the neuron index, t the time-step, T the maximum number of time-steps, and n the total number of neurons. Note that, because this is a mathematical notation, indices start at 1 rather than 0, even though your code should start its indices at 0. Also note that you should skip this calculation for the last neuron, hence $n-1$. Lastly, note that $2T \cdot (n-1)$ is a normalizing term to bring these values between 0 and 1.

To maximize the difference between current activation and previous activation, calculate the following sum:

$$f_2 = \sum_{t=2}^{T} \sum_{i=1}^{n} \left| \frac{activation(neuron_i, t) - activation(neuron_i, t-1)}{2(T-1) \cdot n} \right| \tag{4}$$

where, once again, i indicates the neuron index, t the time-step, T the maximum number of time-steps, and n the total number of neurons. Note that you should not calculate this sum for the first time-step, hence $t = 2$.

The final fitness f should now be calculated as:

$$f = \frac{f_1 + f_2}{2} \tag{5}$$

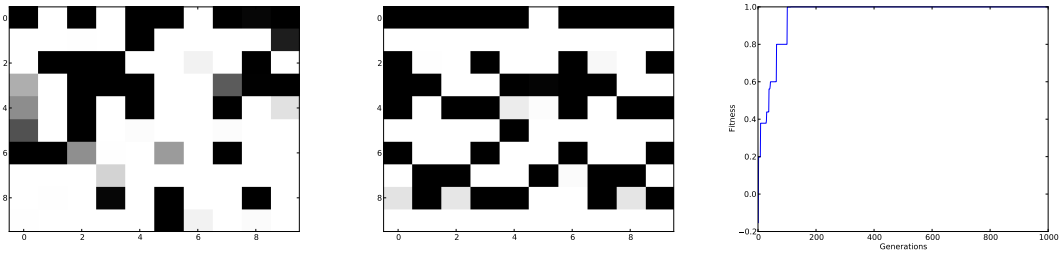
Implement this fitness function in `Fit_Checkerboard.hpp` in the `evaluate` method.

8. Run the `Hillclimber` with the `FitCheckerboard` fitness function for 1000 generations. Once again, write the activation over 10 steps with initial values of 0.5, both before and after running the hillclimber, to a file. Also write the fitness over time to a file. Plot these files using the appropriate plotting scripts and add the results to your document. The figures should look like figures 2a, 2b, and 2c. Note that the evolved neural network may not reach a perfect checkerboard configuration; that is fine if this is the case, but it should have large sections that look like a checkerboard. Run the program a few times until your network gains a fitness greater than 0.85, which should happen somewhere between 50% and 70% of the runs. If you are unable to generate a network with a fitness greater than 0.85, or if the final pattern does not resemble a checkerboard at all, you probably have a bug.

9. Things to think about: What other kinds of neural behavior could you select for? What visual pattern would it produce when plotted? If you like, try implementing these new fitness functions and see whether you get the pattern you were expecting. Answers to these questions need not be included in the document you hand in.

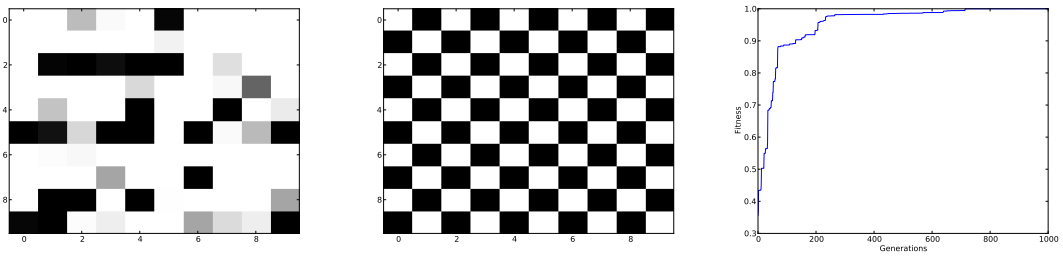
Deliverables

A pdf document containing the figures resembling figures 1 and 2, and a zip file containing your modified source and header files.



(a) Initial activation (b) Final activation (c) Fitness over time

Figure 1: Visualizations demonstrating the successful evolution of artificial neural networks on the task of reaching a specific activation after 10 time-steps. **(a)** Behavior of an initial, random ANN. **(b)** Behavior of an ANN evolved such that the neuron values, on the last update (bottom row), show alternating patterns (n1=0,n2=1,n3=0,...). **(c)** The fitness change of the best ANN over evolutionary time.



(a) Initial activation (b) Final activation (c) Fitness over time

Figure 2: Visualizations demonstrating the successful evolution of artificial neural networks on the task of maximizing the difference in activation between adjacent neurons, as well as maximizing the difference in activation between time-steps. **(a)** Behavior of an initial, random ANN from another evolutionary run. **(b)** Behavior of an ANN evolved such that neighboring neurons exhibit different values, and those values change from one time step to the next. **(c)** The fitness change of the best ANN over evolutionary time using this second fitness function.