CS295/CS395/CSYS395: Evolutionary Robotics

Programming Assignment 3 of 10

Assigned: Friday, September 16, 2011

Due: Friday, September 23, 2011 by midnight

Description: In this assignment you will apply the hillclimber you developed in assignment 1 to the artificial neural networks (ANNs) you developed in assignment 2. The program you develop in this assignment works as follows:

1. Create a (numNeurons=)10×10 (numNeurons=)matrix, parent, to hold the synapse values for an ANN with numNeurons=10 neurons.
2. Randomize the parent synapse matrix.
3. Create a (numUpdates=)$10 \times$ (numNeurons=)10 matrix neuronValues to hold the values of the neurons as the network is updated. The first row stores the initial values of the neurons, which for this assignment will all initially be set to 0.5. The second row will store the new values of each neuron, and so on.
4. Create a vector of length 10, desiredNeuronValues, that holds the output that we want the ANN to create. We'll compare the final values of the neurons—the last row of neuronValues—to this vector. The closer the match, the higher the fitness of parent synapse matrix.
5. Update neuronValues nine times (thus filling in rows two through 10) using the parent synapse values.
6. The program will then loop through 1000 generations. For each loop:
   . Create the child synapse matrix by copying and perturbing the parent synapse matrix.
   . Set each element in the first row of neuronValues to 0.5.
   . Update the neuronValues of the ANN nine times using the child synapse values.
   . Calculate and save the fitness of the child synapses as childFitness.
   . If the childFitness is better than parentFitness, replace the parent synapse matrix with the child synapse matrix and set parentFitness to the childFitness value.

Here are the steps to implement the program.

1. **Back up your Python code from assignment 2.** Encapsulate your code from assignment 2 in a single file, Assignment_2.py, such that when you run it you can reproduce all of the visualizations from that assignment. This will prove to you that that code is working fine, as you will use it in this and subsequent assignments.
2. Create a blank Python file called Assignment_3.py. As you implement this assignment, copy and paste functions from assignments 1 and 2 as they are needed.

3. Copy and paste the main function that you created in assignment 1 at step 8, comment out all but the first two lines and change it so that instead of creating a vector of random numbers, you create a matrix of random numbers. This matrix contains the synaptic weights of the parent neural network. (You'll have to copy across the **`MatrixCreate`** and **`MatrixRandomize`** functions from that assignment as well.) Note that in Python, putting a hash symbol (#) at the front of the line comments it out:

```
parent = MatrixCreate(10,10)
parent = MatrixRandomize(parent)
# parentFitness = Fitness(parent)
# for currentGeneration in range(0,1000):
#     print currentGeneration, parentFitness
#     child = MatrixPerturb(parent,0.05)
#     childFitness = Fitness(child)
#     if ( childFitness > parentFitness ):
#         parent = child
#         parentFitness = childFitness
```

4. Insert a **`print parent`** statement after the two lines to make sure the matrix was randomized correctly.

5. Modify the MatrixRandomize function so that it returns values in [-1,1] rather than [0,1]. **`parent`** now encodes synaptic weights for a neural network with 10 neurons. **`print parent`** to make sure this was done correctly.

6. Uncomment the third line, and copy across the **`Fitness`** function you created in assignment 1.

7. You must now modify the **`Fitness`** function so that it returns a single value that indicates the fitness, or quality of the neural network specified by the synaptic weights stored in **`parent`**. Steps 7 through 14 will help you to do this. First, delete all the code currently within the **`Fitness`** function, and add code into the now-empty function that creates an all-zero (numUpdates=)$10 \times$ (numNeurons=)10 matrix **`neuronValues`** like you did in assignment 2. Print the matrix to make sure it is correct.

8. Fill each element in the first row of **`neuronValues`** with 0.5: each of the 10 neurons will start with this value. **`print neuronValues`** to make sure this was done correctly.

9. Copy across the **`Update`** function you developed in assignment 2, step 12. Use it to fill in the second row of **`neuronValues`** using the synaptic weights stored in parent. Apply it again to fill in the third row. Apply it nine times in total so that **`neuronValues`** is completely filled (**`numUpdates=10`** in this assignment). Print **`neuronValues`** periodically to make sure it is being updated correctly (i.e. each row of zeros are replaced by non-zero values).

10. After the nine calls to **`Update`**, use the plotting function you developed in assignment 2, step 13 to print out how the neuron values change over time. You should get a picture like Fig. 1a or d.

11. Now we are going to calculate how close the final set of neuron values is to some desired set of neuron values. After **`neuronValues`** has been filled, extract the last row as follows: **`actualNeuronValues = neuronValues[9,:]`**. This copies the last row of **`neuronValues`**.

12. Create a vector that stores the desired values for the neurons. Let's select for odd-numbered neurons that are on, and even-numbered neurons that are off:

```
desiredNeuronValues = VectorCreate(10)
for j in range(0,10,2):
    desiredNeuronValues[j]=1
```

13. **`VectorCreate`** returns a row vector of length 10. To create this function, copy, rename and modify **`MatrixCreate`**. To create a vector using NumPy: **`v = zeros((width), dtype='f')`**. The for loop counts from 0 to 9 in steps of 2: 0,2,4,... This code should produce a vector of the form [1,0,1,0,1,0,1,0,1,0].

14. Now, create a function **`MeanDistance(v1,v2)`** that returns the normalized distance between two vectors with elements between 0 and 1: The function should return 0 if the vectors are the same, and 1 if they are maximally different. This function should be used to compute the distance d between **`actualNeuronValues`** and **`desiredNeuronValues`**. **`Fitness`** should then return $f = 1 - d$: the lower the distance between the vectors, the closer the fitness approaches 1.

15. Now, uncomment the remaining lines of the main function, and ensure that your hillclimber is working correctly: **`child`** should be a slightly different matrix compared to **`parent`**, and **`parentFitness`** should increase toward 1 as the loop runs.

16. Add a fitness vector to the main function, and record **`parentFitness`** into this vector after each pass through the loop.

17. Add in a call to your neural network plotting function just before the main loop begins, to show the behavior of the initial random network. Copy and paste the resulting image into your document, which should look like Fig. 1a or d. Note that the image should show a band of gray at the top, which correspond to the initial 0.5 settings of all the neurons.

18. Add another call to the plotting function after the main loop has finished, to show the behavior of the final, evolved neural network. Copy and paste this figure into your document. It should show an alternating pattern of black and white pixels in the bottom row, like Fig. 1b. It may not be perfectly alternating if the fitness of that run did not get to a fitness of 1; this is fine if this is the case.

19. Store the fitness of the parents as the main loop iterates in a vector, and plot that vector. It should look like Fig. 1c. Copy it into your document. Run the program a few times to see what kinds of patterns you get.

20. Now copy the **`Fitness`** function, rename it **`Fitness2`**, and change the two calls to **`Fitness`** in your main function to calls to **`Fitness2`**.

21. Leave the internals of **`Fitness2`** as they are, but change how the fitness of the neural network's behavior is calculated. Remove the mean squared error calculation, and instead compute the average difference between neighboring elements in the matrix:

    ```
    diff=0.0
    for i in range(0,9):
        for j in range(0,9):
            diff=diff + abs(neuronValues[i,j]-neuronValues[i,j+1])
            diff=diff + abs(neuronValues[i+1,j]-neuronValues[i,j])
    diff=diff/(2*9*9)
    ```

22. Re-run the hillclimber, and save out the initial random neural network's behavior (as in Fig. 1d), the behavior of the final, evolved neural network (as in Fig. 1e), and the fitness increase during the hillclimber (as in Fig. 1f). Note that the evolved neural network may not reach a perfect checkerboard configuration; that is fine if this is the case. Run the program a few times to see what kinds of patterns you get.

23. **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.

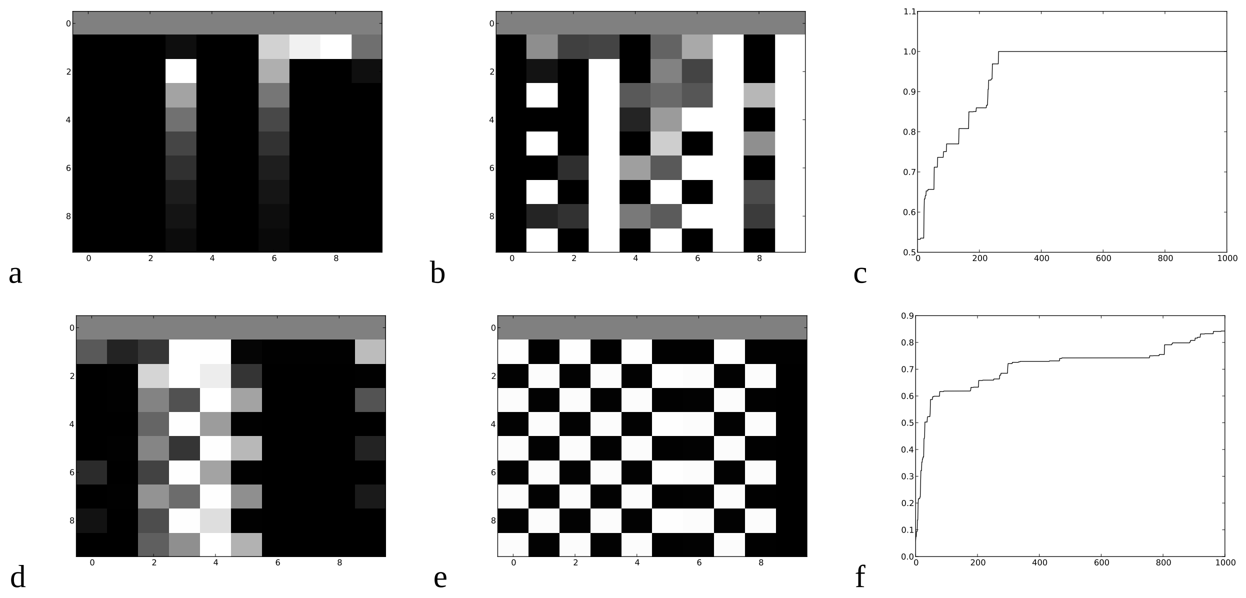

Figure 1: Visualizations demonstrating the successful evolution of artificial neural networks. a: Behavior of an initial, random ANN. b: Behavior of an ANN evolved such that the neuron values, on the last update (bottom row), show alternating patterns (n_1=0,n_2=1,n_3=0,...). c: The fitness change of the best ANN over evolutionary time. d: Behavior of an initial, random ANN from another evolutionary run. e: Behavior of an ANN evolved such that neighboring neurons exhibit different values, and those values change from one time step to the next. f: The fitness change of the best ANN over evolutionary time using this second fitness function.