Project 4 IT3708: Evolving Neural Networks for a Minimally-Cognitive Agent

Purpose: Learn to implement an evolving neural network to use as a controller for a simulated agent that performs a task requiring a small amount of intelligence and memory.

1 Assignment

You will use your evolutionary algorithm (EA) to evolve the weights and and other parameters for a small artificial neural network (ANN) whose fitness will depend upon the performance of an agent controlled by the ANN. This agent will do a minimally cognitive task similar to those used in the field of artificial life (ALife) to illustrate the emergence of rudimentary intelligence. Specifically, the agent will sense falling objects of various sizes and must evolve the ability to a) intercept objects smaller than itself, but b) avoid objects of its own size or larger.

2 Beer Tracker: The Testing Environment

The basic testing environment is a modified version of a simple game used by Randal Beer in his research on ANNs. The game consists of a *tracker agent* and objects falling down from the sky in a grid-world.

The world is 30 units wide and 15 units high. A game lasts 600 *timesteps* in this world.

Objects spawn from the top of the world, and fall straight down with a speed of 1 unit per timestep. The horizontal spawn position of the object should be random, as well as the size. The size can be from 1 to 6 units wide. There is only one falling object in the world at the same time. When an object hits the tracker or the bottom of the world, a new one appears on the top of the world the next timestep.

The tracker should be 5 units wide, and can only move horizontally at the bottom of the world. The tracker consists of 5 shadow sensors (one per unit) that can see directly above whether there is an object there or not. So the tracker only knows what's above it, and can't see the whole world. It can move *up to* 4 steps in either direction per timestep. You will need to figure out how to use the outputs of the two motor neurons (more about this later) to determine the direction and magnitude of each move.

A *capture* is when the *whole* falling object lands on the tracker. *Avoidance* is when no part of the falling object lands on the tracker. Note that avoidance is more than failing to capture, for instance would a falling object where only half of it lands on the tracker neither count as a capture nor an avoidance.

The goal of the game is for the tracker to capture *small objects* (of size 1, 2, 3 and 4) and avoid *big objects* (of size 5 and 6).

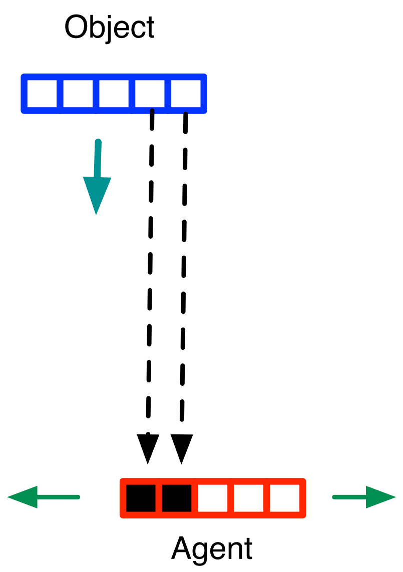

Figure 1: A simplified version of Randall Beer's classic video-game scenario. Here, objects cast a shadow straight down (dashed vertical arrows), and the tracker agent consists of a row of shadow sensors (red squares). Two of the sensors are sensing something above. The agent can only move horizontally, while the objects fall straight down. The tracker captures the object if it is completely below it, and the object is not too big.

3 Continuous Time Recurrent Neural Networks

Via his minimally-cognitive behavior research, Randall Beer has popularized an interesting new style of ANN: Continuous-Time Recurrent Neural Networks (CTRNNs). These supplement standard ANNs with two neural properties: a time constant and a gain. Values of both properties normally vary among neurons. Beer often describes the CTRNN neural model in a dynamic-systems format, while the following reformulation adheres more closely to the standard integrate-and-fire perspective.

First, integrating the inputs from all upstream neighbors involves the standard equation (1):

$$s_i = \sum_{j=1}^{n} o_j w_{i,j} + I_i \tag{1}$$

Here, o_j is the output (or, equivalently, the activation level) of neuron j, while $w_{i,j}$ is the weight on the arc from neuron j to neuron i. I_i is the sum of all external inputs to neuron i. A typical external input is the value of the single sensor associated with an input neuron. Non-input neurons typically have no external inputs.

In Beer's CTRNN descriptions, y_i denotes the internal state of neuron i. Equation 2 shows that the derivative of this internal state, $\frac{dy_i}{dt}$, is a combination of s_i and a *leak term*, wherein a portion ($\frac{y_i}{\tau_i}$) of the internal state from the previous time step drains out. In addition, Beer incorporates a neuron-specific *bias* term, θ_i. As shown later, this is easily modeled as an input from a *bias neuron* and is thus incorporated into the s_i term.

$$\frac{dy_i}{dt} = \frac{1}{\tau_i}[-y_i + s_i + \theta_i] \tag{2}$$

The time constant, τ_i, determines how fast the neuron changes internal state. A low value entails fast change, with the new state being predominantly determined by current conditions; whereas a high τ_i produces more *memory* in the neuron for its previous state(s) (due to less leak and less influence of s_i), which are reflections of earlier conditions. It is this memory, and the ability of neurons to vary in their levels of it (via different τ values) that enables CTRNNs to exhibit extremely rich dynamics, and hence, convincingly sophisticated cognition. This assignment only scratches the surface of those cognitive abilities.

As shown in equation 2, Beer typically employs a logistic (i.e. sigmoidal) activation function to convert the internal state to an output, o_i. However, y_i is multiplied by the neuron-specific gain term, g_i.

$$o_i = \frac{1}{1 + e^{-g_i y_i}} \tag{3}$$

On each timestep, a CTRNN neuron must recompute s_i and then $\frac{dy_i}{dt}$, from which new values of y_i and o_i are then calculated. The new value of y_i is simply the old value plus $\frac{dy_i}{dt}$, while o_i stems from equation 3.

The CTRNN model is used in all of Beer's minimally-cognitive simulations, spread across dozens of research papers; but the ranges of key parameters (gains, biases, weights and time constants) vary according to the demands of the different minimally-cognitive tasks. The network topologies typically consist of an input layer - with each neuron attached to one sensor and simply outputting the value of that sensor - a hidden layer, and an output (motor) layer, which controls the left and right movement of the agent. The *recurrence* in CTRNNs normally stems from a) connections among the neurons of the hidden layer, b) backward links from the motor neurons to the hidden layer, c) connections between the two motor neurons, and/or d) connections from individual hidden (or motor) neurons to themselves. Recurrence adds a second form of memory to the CTRNNs, since the states of neurons are fed back into the system

4 Evolving the CTRNN for a Tracker Agent

Figure 2 portrays a CTRNN topology that suffices to solve simple tracker problems. The 5 input neurons correspond to the 5 shadow sensors of the agent. These simply output a 1 or 0, depending up on whether or not a portion of the falling object is directly above that sensor. The hidden and output/motor layers each consist of 2 neurons, all of which abide by the CTRNN integrate-and-fire rules of equations 1, 2, and 3. The input layer is fully connected to the hidden layer, which is fully connected to the output layer. Recurrence stems from intra-connections in the hidden and output layers.

To account for the bias terms, θ_i in equation 2, a standard technique is to include a single bias node (labeled B in Figure 2. This node always outputs a 1, but the weights on the connections from B to each of the hidden and output nodes can vary. In this case, they are evolved. Hence, evolution determines the biasing input to each node.

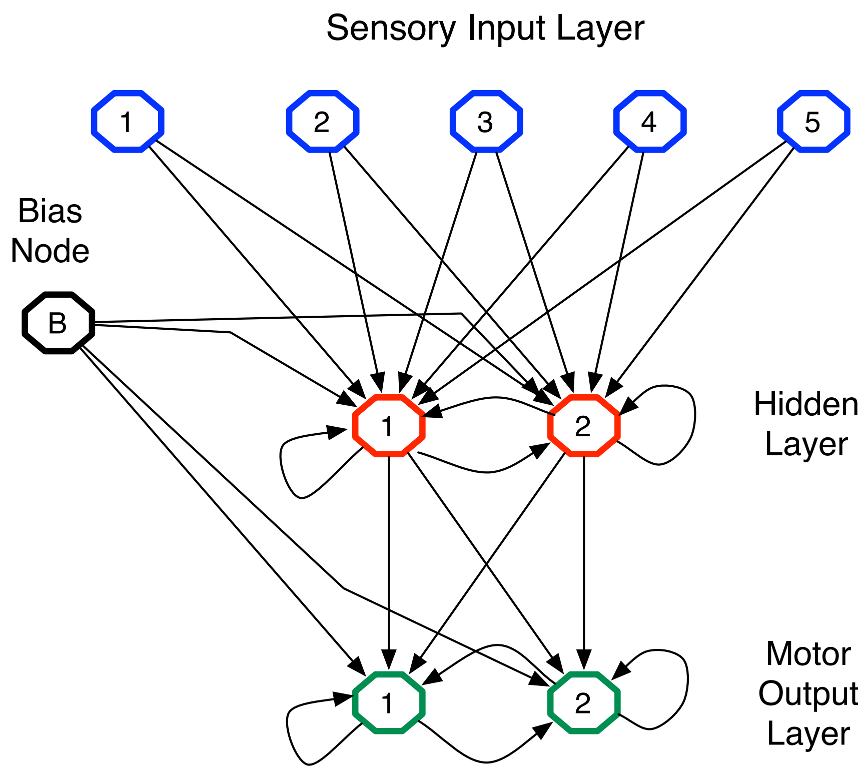

Figure 2: A CTRNN topology for evolving a tracker. All connections are shown (thin arrows). The looping connections imply that a node feeds its own output value back to itself on the next timestep. Each connection in the diagram has an evolvable weight: none are hard-wired by the user.

4.1 ANN Parameters to Evolve

Your EA will need to evolve the following parameters:

1. Weights for each connection in the CTRNN, including those on arcs out of the bias node.
2. Gains, g_i, for each of the hidden and output neurons.
3. Time constants, τ_i, for each of the hidden and output neurons.

The following ranges of these parameters (all real numbers) are known to work for the topology of Figure 2:

1. Weights (emanating from all nodes except the bias node): [-5.0, +5.0]

2. Biases (i.e., weights on the arcs emanating from the bias node): [-10.0 , 0.0]
3. Gains: [+1.0 , +5.0]
4. Time constants: [+1.0, +2.0]

Binary genes of 8 bits per parameter suffice to evolve CTRNNs that can solve the Tracker problem. So each parameter is encoded using 8 bits, and your converter converts those 8 bits to a value in the correct range given above.

4.2 Fitness function

Your fitness function will take the phenotype, which is all the different weights etc., and use them for an agent's CTRNN. Then run the agent through the Beer game and record what happens.

You will need to devise a scoring function that rewards the desired behaviors: catching of small objects and avoidance of big ones. How much you reward capturing small objects and avoiding big objects, and how much you punish avoiding small objects and capturing big objects is something with which you must experiment. *You will probably spend a lot of time tweaking this function to get the desired behavior.*

You may need different scoring functions for the various scenarios below, but how the phenotype is fed to a CTRNN and then used in the Beer game should be the same.

5 Scenarios

You will use your EA to evolve agents that can handle various modifications to the game. The standard scenario is the most important one, which the two other scenarios are based upon, so spend time getting that one to work first.

For all of the scenarios, it can be smart to start "easy" by having the agent try to capture everything. When that works, you can then extend it to avoid the big objects.

When your EA is done, you should visualize the best evolved agent play the game. The visualization can be very simple, but it should be easy to see the behavior of the agent. Have different colors for the tracker, small objects and big objects to make them easier to distinguish. The delay between visualized timesteps must be adjustable so that we can slow down the run to see exactly what the agent is doing.

5.1 Standard

This uses the CTRNN and values discussed earlier to play the game. There should be *wrap around*, so that an agent exiting the world on the right (left) comes back to the left (right). This makes it a bit easier for the CTRNN to only focus on capturing/avoiding objects.

Expected performance for the standard scenario is an agent that search the world and can distinguish between small objects to capture and big objects to avoid.

5.2 Pull

The tracker gets a new *pull* action that it can use. This action makes the current falling object instantly drop all the way down to the bottom of the world. The rules for avoidance/capturing are the same, so if the tracker is below the object when using the pull action it may capture it. When an object has been pulled down, a new one will spawn the next timestep as normal.

Pulling an object takes 1 timestep in the game. So smart agents will use this to quickly capture/avoid more objects during the game and thus get a higher score, instead of waiting for objects to fall down.

To implement this, you will need to modify your CTRNN to output this new action. Visualize the action in some way, for instance by having the tracker momentarily change color.

Expected performance here is an agent that uses this new superpower in a somewhat smart way to capture more objects during the game. For instance by pulling objects directly above that it wants to capture.

5.3 No-wrap

This scenario is the same game as in the standard scenario, but there is no wrap around. So an agent moving to the left (right) will eventually hit the left (right) edge of the world and stop.

Your agent will need to learn how to then change direction. How you do this is up to you, but the easiest way may be to add two new sensors to the tracker. One sensor on the left of the tracker will say whether it is touching the left edge of the world or not, the same for a sensor on the right.

Your CTRNN will then need two new input neurons, corresponding to the values of these sensors. The range of weights from these sensors can be the same as for the other input neurons, but seeing as moving away from the wall now becomes the number one priority, experimenting with a bit bigger range for these weights is a good idea.

Expected performance here is an agent that has learned to turn when hitting the edge of the world. If it at the same time can capture/avoid the correct objects, that is very good!

6 Report

You should write a report answering the points below. The report can give a maximum score of 8 points. Your report must not exceed 2 pages in total. Overlength reports will result in a point being deducted from your final score. Print on both sides of the sheets, preferably. Bring a hard copy of your report to the demo session.

a) Document your implementation (2p)

- Document your genotype/phenotype representations and the conversion between them.
- Describe how you have implemented the CTRNN. Include how it's used and how it determines which action to take.

b) Performance of the EA (3p)

- For the *standard scenario*, describe the behavior of an evolved agent in detail. For example, does the agent wait under objects for a period, sizing them up, before making a move; or does it move continuously and only stop when it senses a small object; or does it do something else? Also include the fitness function used.
- Do the same for the *pull scenario.*
- Do the same for the *no-wrap scenario.*

c) Analyze an evolved CTRNN (3p)

- Show the evolved weights for an agent for the *standard scenario.*
- Analyze how these weights determine the behavior of the agent.
- Try different inputs to the network and analyze the outputs. Also try to show how the "memory" of the network can give different output for the same input based on earlier input.

7 Demo

There will be a demo session where you will show us the running code and we will verify that it works. This demonstration can give a maximum of 12 points.

For this demo, you will use your EA to evolve agents and show us that they behave reasonably intelligent. Your EA should be able to produce these good agents in just a few minutes during the demo. We will ask you to demonstrate all of the three scenarios. If you have troubles getting some of them to work properly, you can show us them with an agent that catches all objects (instead of distinguishing between big/small ones) for partial credit.

You should be able to choose which scenario your EA should evolve an agent for without having to modify any code during the demo.

8 Delivery

You should deliver your report + a zip file of your code on It's Learning. The deadline is given on the assignment on It's Learning, and it is **before the demo sessions start in the morning**. The 20 points total for this project is 20 of the 100 points available for this class. For this project you can work alone or in groups of two.