IT3708 - Project 3

Evolving Neural Networks for a Minimally-Cognitive Agent

Written by: Thomas Almenningen, Halvor G. Bjørnstad

Introduction

The code is written in C++. In order to generate the plots you need to have matlab installed. We have used freeglut 2.8.0 which is an open source alternative to the OpenGL Utility Toolkit (GLUT) library to generate the graphics and need to be installed for the graphics to work. The dependencies are added in the dependencies folder in code.zip (freeglut & glew-1.9.0)

1. System Overview

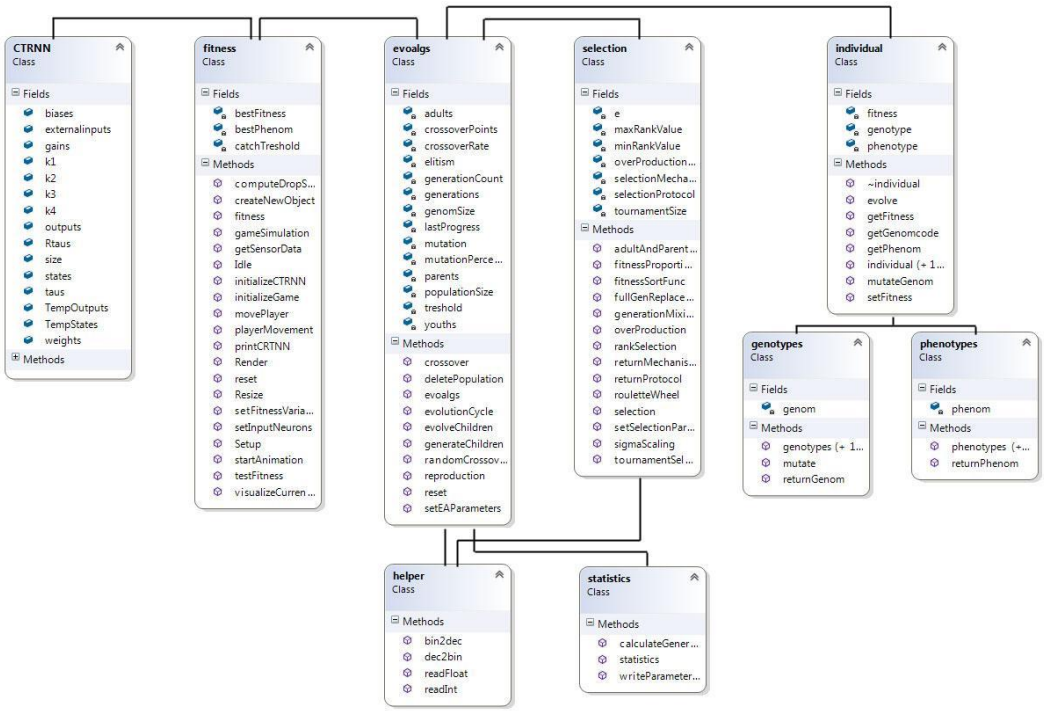

Figure 1: Class Diagram Generated In Microsoft Visual Studio 2012 Professional

We chose to use the CTRNN class found on Randall D. Beers homepage [1]. We made some small modifications to the eulerTime function so we got the same equations that were presented in the project description. Although making our own CTRNN class would be trivial, making it as fast as Beers' would not be.

Our genotype is represented by a vector of genes, where each gene is a bit set vector of size 8. The first 22 genes represent the connection weights of every neuron to neuron connection, the next 12 genes contain bias weights, gain parameters and time constants for the hidden and motor neurons.

The tracker environment is 15 blocks wide, 30 blocks high. The player width is set to 5 and the width of the object varies between 1 and 6. The vertical movement of the objects is set to one, and we have not implemented any sideways movement (But we have written the code for it).

To calculate the tracker's movement we use the following function, $X_{movement}$ = floor([(M_L - M_R) / C] + 0.5) where M_L and M_L are the motor neurons' outputs and C is a constant equal to 0.25. Thus resulting in an x-movement between [-4, 4].

2. Evolving Catching Behavior

To calculate the fitness of each individual we sum the score for each game instance. A successful catch is not defined for object sizes larger than the tracker agent. We elect to define the successful catch of this case to be whenever tracker agent is completely overlapped by the object. The score of each game instance is either *CATCH_BONUS* = 2.0 for catches or *NO_CATCH_PENALTY* = 0.0, the maximum fitness is then 80 when dropping 40 objects.

Figure 2 (Below) shows the number of generations that were needed to reach a perfect solution over 25 runs with the same parameters; except for 2 rare cases we were able to find the perfect solution in less than 150 generations. In the two cases where our EA failed we only missed 1 object, which we find acceptable. If you wish to view more detailed information we have written the run data to the file task2_phenoms.txt which can be found in the .zip file. The Structure in the file is: Run #, Generation #, Max fitness found, Parameters

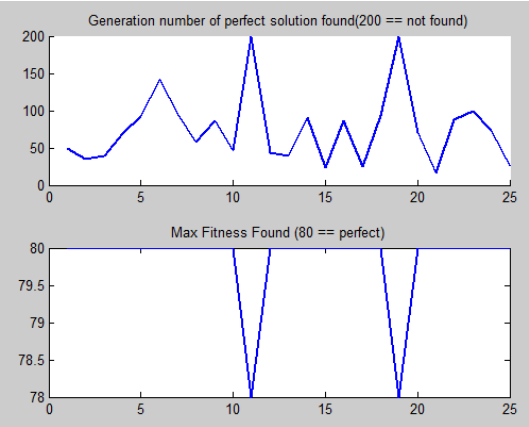

Figure 2: Fitness plots for Task 2

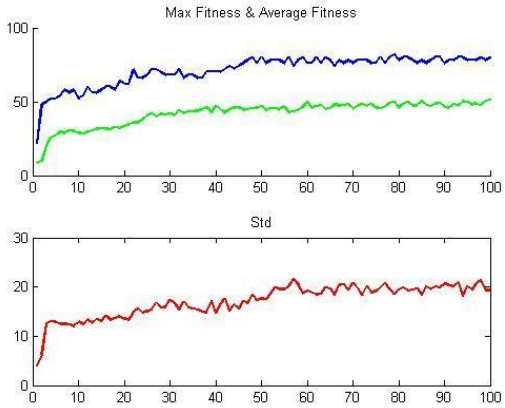

Figure 3: Fitness plot, Catch All

Figure 3 shows a pretty typical fitness plot for evolution of catching behavior. The tracker is able to capture all objects and achieve a maximum score of 80 at around 60 generations. A typical behavior for an agent for this scenario moves only one way and stops every time it gets under an object.

The parameter setting that seemed to give the best consistent solutions were as follows: Population Size: 100, Generations: 200, Elitism: 1, Mutation rate: [0.7-0.85], Mutation

Percentage: [0.04-0.1], Crossover rate: 0.2. For adult and parent selection we used Full Generation Replacement and Sigma Scaling. These parameters are more or less used for the rest of the project as well.

3. Evolving Catching and Dodging Behavior

To calculate the fitness of an individual we again sum up the score of each game instance. The score of each game instance for objects of size smaller than the player is *CATCH_BONUS* = 2.0 for catches or *NO_CATCH_PENALTY* = 0.0 for missing a catch, for objects of size greater than or equal to the player it is *AVOIDANCE_BONUS* = 2.0 for completely avoiding the object and *CRUSHED_PENALTY* = 0.0 for any overlap > 0, the maximum fitness is again 80 because the two situations are mutually exclusive.

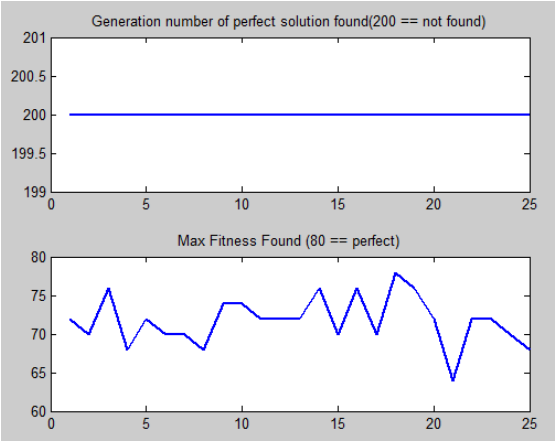

Figure 4: Performance of our EA over 25 generations, avoiding big objects and catching small objects, using the same parameters as in task 2. Our best found fitness values ranged from 64-78, which means that we were unable to evolve perfect behavior. But we are still fairly satisfied with the results. More data is available in task3_phenoms.txt

To successfully evolve better solutions we changed our block dropping function from generating an object of a random size to select values from our predetermined training set to assure a uniform distribution. This removed some occasional erratic behavior, and made us able to improve our agents slightly.

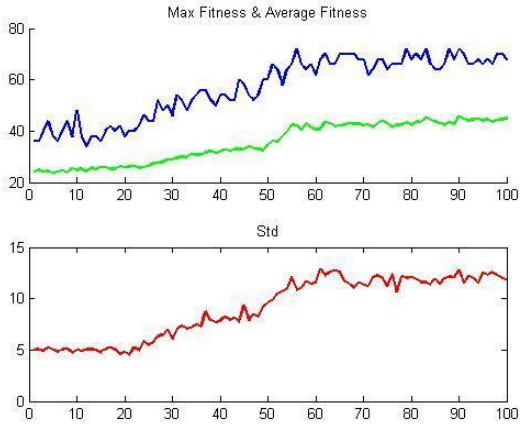

Figure 5: Fitness Plot, Catch and Avoid

Figure 6: Fitness Plot, Catch and Avoid

The agent evolved in figure 5 was kind of interesting in many ways. It moved constantly to the right all the time, but when it got under an object which covered all its sensors it jumped 8-10 steps to the left before continuing to the right. This made it able to dodge most blocks which it sensed fairly close to the ground, but failed on a few which it sensed so far up that it had time to move back under it again before it hit the ground. The best sound solution had a fitness of 72 of 84 possible.

The agent evolved in figure 6 clearly displayed intelligent behavior. It moved right with a constant velocity and stopped when it came under a smaller object and increased its speed to get away from larger objects. The best found solution had a fitness of 78 of 84 possible, which is still not perfect but pretty close.

4. Modifying the Tracker Environment

In this part we will play around with the vertical movement speed of the objects.

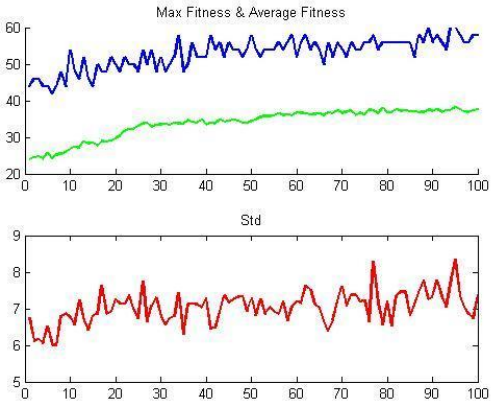

Figure 7: Fitness Plot, Object Drop Speed set to 1.5

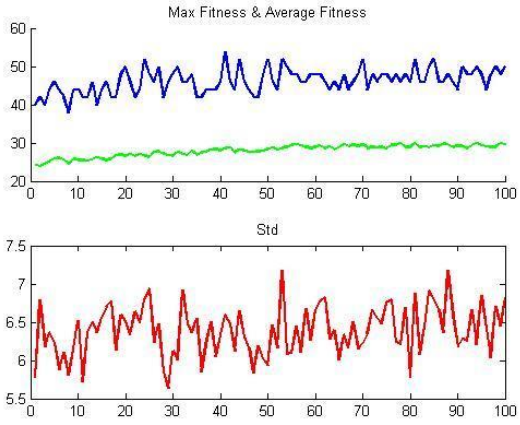

Figure 8: Fitness Plot, Object Drop Speed set to 2.0

Based on the two figures above we can conclude that a higher vertical drop speed increases the difficulty of evolving a good tracker significantly. When setting the vertical movement speed to 2.0 this means that our tracker has half as many moves to pick up or decide to dodge an object. The most obvious change was that the tracker seemed to focus less on dodging objects since it basically no longer had time to explore the size of an object before deciding to pick it up or dodge it. From the visualization of the tracker when the vertical movement speed was set to 1.5 the most significant changes from the standard setting of 1.0 was that the horizontal movement of the tracker seemed to be larger and more aggressive compared to the earlier visualizations. The tracker no longer has time to dodge as many objects and more often gets crushed by bigger objects.

When the vertical movement speed was set to 2.0 the tracker clearly mainly focused on picking up objects. Some runs we saw some small tendencies of dodging behavior but it still mainly focused on picking up as many objects as possible to maximize its score.

The last setting we tried was setting the vertical speed to 0.5. This gave a really good result. The resulting tracker moved sideways at a constant speed and stopped for smaller object and increased the speed to avoid larger objects (The same behavior evolved in figure 6).

5. Modifying the CTRNN Topology

For this task we chose to implement an additional neuron in the hidden layer. This means that the total amount of weights needed to set up the CTRNN increased from 22 to 34 and we also need one more bias, gain and time constant parameter. This gave us a total genotype size of 49.

We deviated from the topology shown in the problem text by adding an additional neuron in the hidden layer. As all other hidden neurons, this new neuron receives inputs from other hidden neurons (including itself), the bias neuron and the sensor neurons. It outputs to every hidden neuron (including itself) as well as to the motor neurons.

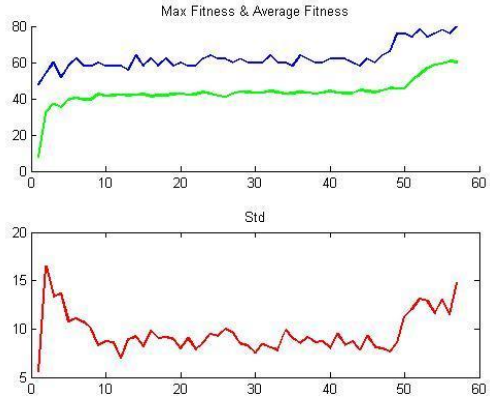

Figure 9: Fitness plot for catch all objects case

Figure 10: Fitness Plot for catch and avoid

For the catch all objects case we were quickly able to evolve a tracker which successfully could catch all objects (Between 21-49 Generations over 5 tries). On all the visualizations we watched the tracker was only able to move either only left or only right and stopped correctly to pick up more or less all the objects that were dropped.

For the catch and avoid case we seemed to be able to evolve good behavior more frequently compared to our original topology. The behavior evolved in figure 4 was pretty close to perfect and constantly moved right and jumped to the left every time it got under a large object. It still got crushed a few times when it didn't have time to move away after sizing up the object.

The general consensus in the literature seems to be that you should rather use too many than too few nodes in the hidden layer. And that the optimal size of the hidden layer usually lays between the size of the input and size of the output layers. What we experienced was that our new topology more often than not resulted in better behavior (And better average fitness) than our original topology.

6. Modifying Parameter Ranges

We decided to change the range of the time constants to range from [1, 10]. The time constant is there to model the membrane resistance. In our case this will affect how fast a neurons internal state can change.

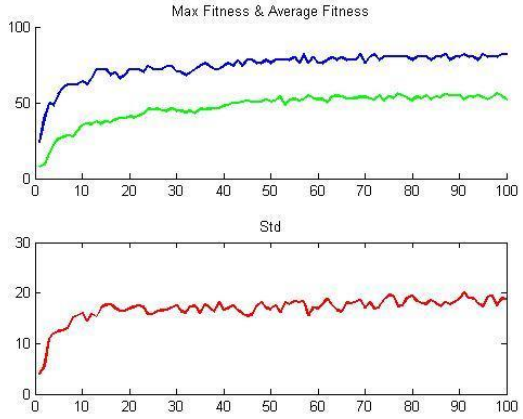

Figure 11: Fitness plot for Catch All

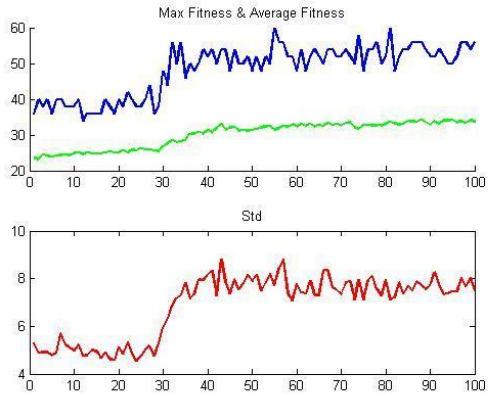

Figure 12: Fitness plot for Catch and Avoid

If we look closer at the parameters that were evolved for the catch all scenario we find that 1 out of 4 values are outside the range which was initially specified. Time Constants: Hidden neuron 1: 1, Hidden Neuron 2: 1, Left Motor Neuron: 6.68235, Right Motor Neuron: 1.24706. The best found parameters for the catch and avoid scenario are as follows: Hidden Neuron 1: 4.2, Hidden Neuron 2: 1, Left Motor Neuron: 1.28235, Right Motor Neuron: 4.04314

The above evidence shows that the new ranges actually are used in the best found solution and therefore are viable. We were not able to generate as good results as with our standard

parameter ranges. Looking at the data we see that the time constants more often than not seem to end up fairly low, below 3 for the best solutions. We both agree that a lower time constant would be better for these kinds of problems, since the tracker should be able to alter its internal state fairly quick in order to perform well in an environment such as this.

The way the behavior was affected by this increase was that some of the trackers that had multiple high time constants often ended up staying still for many timeframes without moving at all. We also performed a few trials where the range was changed to [5, 10] which resulted in trackers that performed very poorly.

7. Analysis of Evolved CTRNN

The CTRNN we will study in this task consists of the following parameters which scored 80 of 84 possible in a catch and avoid simulation consisting of 42 object drops. The Catch and avoid bonuses were set to 2. Both penalties were 0.

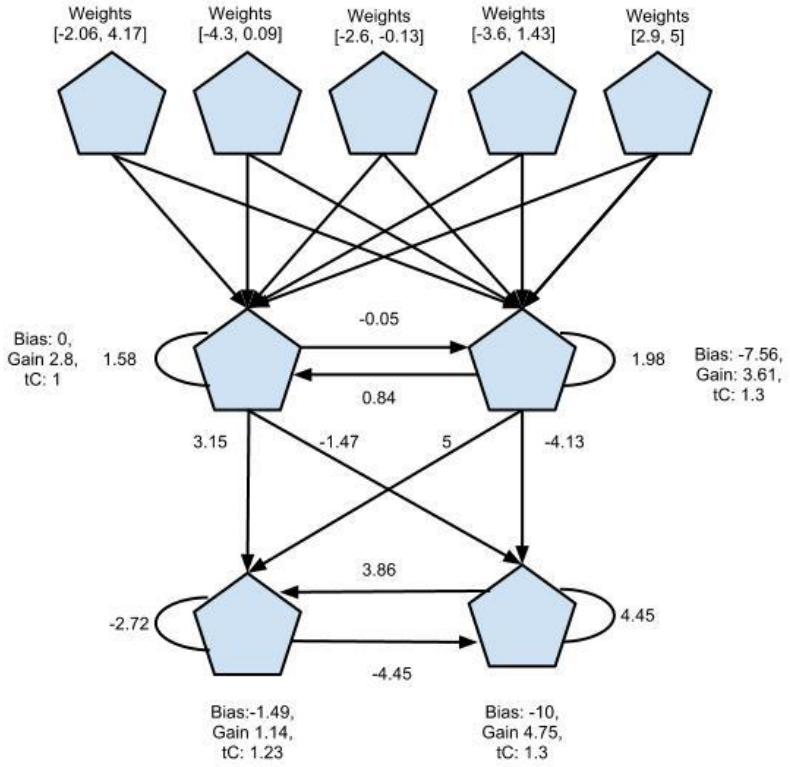

Figure 13: The evolved parameters

7.1 Input/Output Behavior

The following table shows how the CTRNN reacts to new input over 2 time steps

Sensor Data	Movement Timestep 2	Movement Timestep 3
10000	0	0
11100	0	0
11111	4	3
00111	1	1
00001	3	2
01110	0	0
00000	1	2

When no sensors are activated the agent will start moving right until it achieves a speed of 2 blocks pr. time step. When a few of the right sensors are activated and the left ones are not, the tracker will size up the object by moving under it so the object ends up on the left side of the tracker. We then have two possible scenarios:

- If the whole tracker is then covered by an object it will speed up its right movement and dodge it.
- If some of the right sensors are not activated it will stop and pick up the object.

When the left sensors are activated but the right ones are not, the agent will stop and wait for the object. This will result in some failure when an object spawns over the left side of the agent since the agent will perceive this as a smaller object.

If the object is of a smaller size (3 or less) we will end up with a case where the center sensors are activated but the outer ones are not. In this case the agent will stop and wait for the object to fall down.

The main reason why this agent cannot obtain a perfect score is that it will fail when it gets under a larger object that it so close to the ground that the agent is not able to move away from it on time. The agent will also fail when an object spawns above the left side of the agent activating only the sensors on the left side. The agents' movement speed when no sensors are activated could also preferably have been higher.

7.2 Weight analysis

The general pattern we can see in the connection between the input layer and the hidden layer is that all the weights going to the right hidden is greater than those going to the left hidden neuron. Most connection going to the left hidden neuron is negative in 4 of 5 connections.

The connections from the hidden layer to the motor neurons are the opposite from what we see from the input neurons to the hidden layer. Here, the weights going to the right motor neuron are smaller (and negative), while the connections to the left motor neuron are positive and fairly high values.

The connection weights between the motor neurons are pretty interesting. All weights going from the left motor neuron are negative, while the connection going from the right are on the opposite side of the spectrum and are positive.

The following table shows how the neurons internal states update as the same input is fed into the CTRNN over 3 time steps.

Timestep	LH N	RH N	LM N	RM N	LM Output	RM Output
#1 [10000]	-2.05	1.94	-1.206	-7.72	0.2	0
#2	-2.05	-2.77	-1.86	-10.2	0.1	0
#3	-2.05	-3.13	-1.78	-10.4	0.115	0
#1 [11111]	-10.5	1.7	-1.2	-7.7	0.2	0
#2	-9.6	3.58	2.15	-13.4	0.92	0
#3	-9.64	4.38	1.21	-17.1	0.8	0
#1 [00001]	2.09	-1.47	-1.2	-7.72	0.2	0
#2	3.68	-2.1	0.68	-11.3	0.68	0
#3	3.68	-2.4	-0.03	-13.8	0.49	0

Looking at the table it is really hard to really see any general pattern. Both times the input resulted in movement the values of the nodes in the hidden layer are completely different. We can also see that the state of the left motor neuron is what really controls the agent, since the output of our right motor neurons always is close to zero due to high negative value and gain parameter. This makes all the weights related to the right motor neuron uninteresting to study further.

The only obvious pattern we could see was that activation of input nodes with high weights towards the right hidden node, results in a higher state value for our left motor node due to the high weight values which connect these nodes in addition to the right hidden neurons high gain factor. This means that inputs that gives the right hidden neuron a value as high as possible without reducing the value of our left hidden neuron to much will result in the most movement (Input such as [10001]). This means that when the right sensor fires the agent is likely to increase its movement speed to the left. While activation of nodes on the right side leads to lower values of the hidden nodes thus no movement.

References

[1] - http://mypage.iu.edu/~rdbeer/