IT3708 | The Flatland-Agent

Kristoffer Hagen, MTDT, spring 2014
Mads Wilthil, MTIØT, spring 2014

1. Evolutionary Parameters

The values we used for the final testing is summarized in Table 1. An eliteism count of 5 were based on our gut feeling, thoguh raising or dropping this number significantly did not give any clear improvements. We found the parameters by keeping all but one parameter fixed, while tweaking the target parameter. After all the parameters were established this way, we tried a drastic change of the mutation rate and crossover rate to test them again.

Mutation rate	Crossover rate	Population size	Max generation count
0.2	0.8	30	50

The selection strategies were tested with less accuracy. We found that most combinations of parent- and adult selection strategies gave us adequate solutions after about 20 runs (in the static case). For the runs described below, we used a fitness porportionate parent seleciton strategy, and used genertional mixing for the adult selection strategy, sorting on the best individuals.

2. The Fitness Evaluation

Starting things off we kept the fitness evaluation simple. For a run r, having timesteps $t = 1, ..., T$ let the food if the board at time t be f_t and the poison on the board at time t be p_t. The fitness of the run is then defined as:

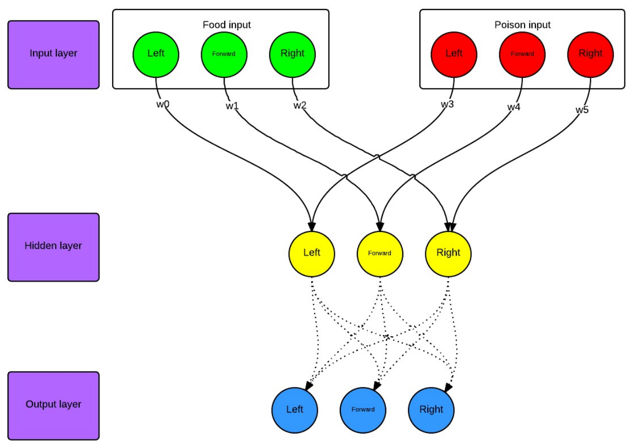
$$F_r = (f_0 - f_T) - 5(p_0 - p_T)$$

In simpler terms, we count how much food and poison the agent eats in a single run, and award 1 point for each food eaten and deduct 5 point for each poison eaten. The number 5 was taken from experiments; we tried 1, 2, 5, 10 and 50, the latter essentially killing the agent. 5 proved a good balance between avoiding poison and exploring the search space.

Averaging the fitness for all 5 runs give the agents fitness meassure.

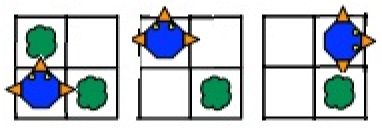
3. ANN Design

We started out with a simple ANN with a hidden layer consisting of three nodes.



We decided to make a change in the connectivity between the input and the hidden layer, instead of n-to-n connectivity, we went for a one-to-one, where both inputs dealing with sensors from the same location are sent to the same hidden layer node. Even though it is assumed that every node in layer A connects to every node in layer B, we decided to use this one-to-one connectivity, we thought that this architecture would enable us to see the emergence of rational behavior at a faster rate and thus we could run more simulations with interesting results without having to run for hundreds of generations.

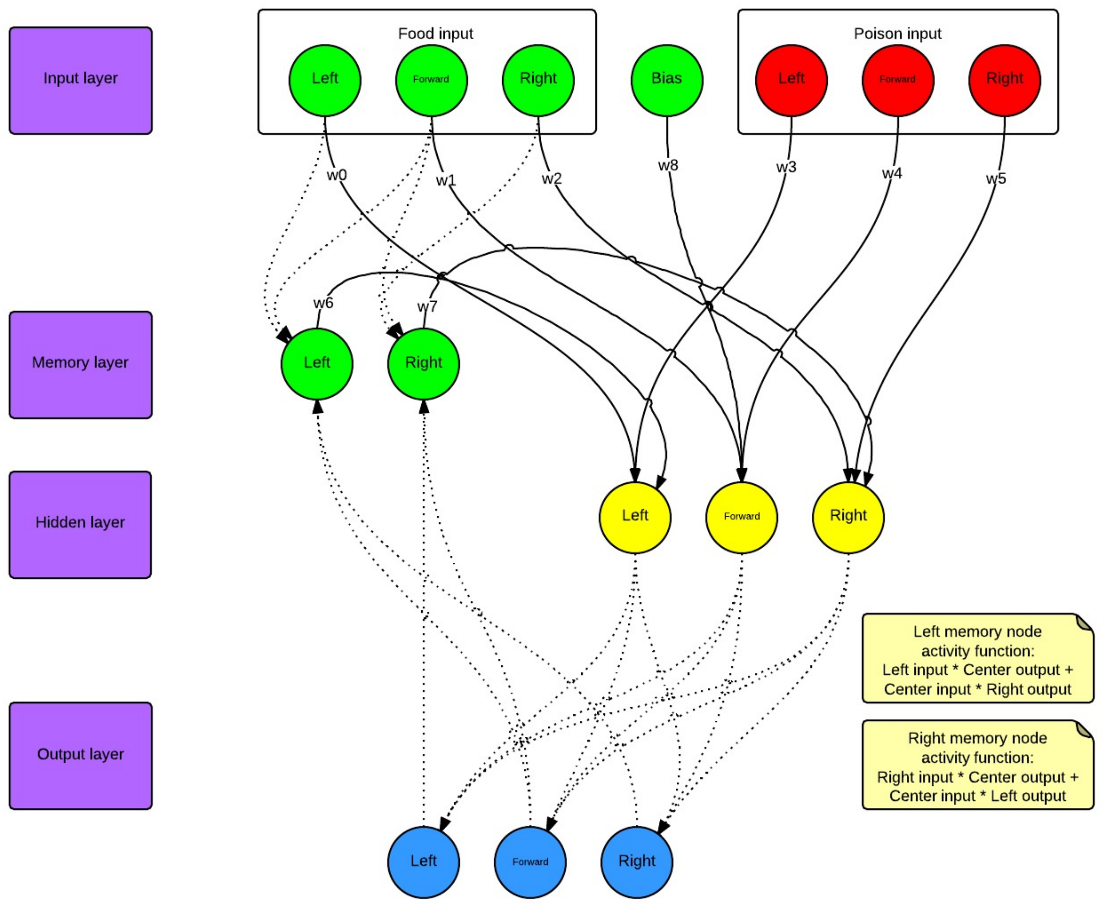
The next step was adding some sort of memory to the ANN. We wanted the individuals to have a very short term memory where they could store information that could be of use in the next step. Let us use an example:



In the first picture the robot will move forward, in the second one the robot has no input to the six sensor units so it will move randomly, what we want to implement with the memory part is the the robot in the second picture will move right so that it is in position to eat the food it saw in picture one, on the next move. When the robots spots food that it does not end up eating we want it to remember it and then have to option to let that memory affect the next move.

The last part we wanted was a bias node. We did decide to let the robot move in a random direction when no input was discovered, but quickly discovered that this might not be the best behavior. Therefore we added a bias node only linked to the center node in the hidden layer. This would enable the individual to move forward by default should no inputs be registered and the weight from the bias node be high enough.

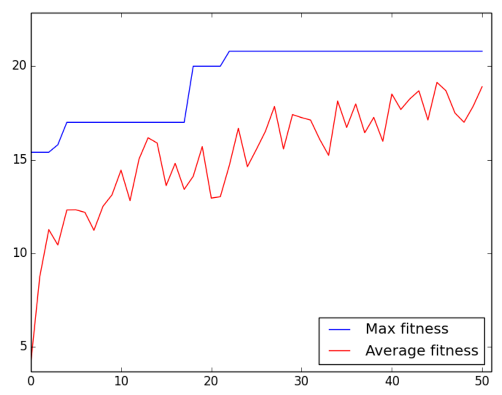
The final ANN looks like this:


Input layer
Memory layer
Hidden layer
Output layer
Food input
Left
Forward
Right
Bias
Poison input
Left
Forward
Right
w0
w1
w2
w8
w3
w4
w5
w6
w7
Left
Right
Left
Forward
Right
Left
Forward
Right
Left memory node activity function: Left input * Center output + Center input * Right output
Right memory node activity function: Right input * Center output + Center input * Left output

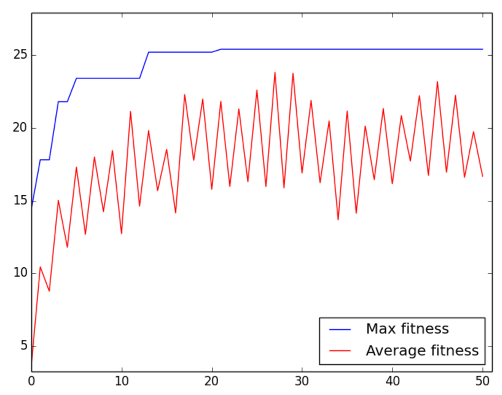
The output layer will end up selecting the highest hidden layer node as output. The memort layer works as follows. The activity function of the left node is, Left food input * Forward output + Forward food input * Right output. This will cause the memory nodes to fire only if it is beneficial to move in that direction the next move. Have we had enough time and computing power we would ofcourse liked to use a all-to-all connectivity and let evolution adjust the weights to accomplish the same behavior, but given the situation we decided to make our ANN this way.

4. Static VS. Dynamic Runs

One of the core challenges in a static environment is to avoid overfitting. Five maps being used troughout the enitre evolutionary run should be of no problem if they are diverse enough, and prove a good generalization of the flatland concept. After running several simulation, both static and dynamic, we did discover that the final fitness of the best individual was not better nor worse in any of the cases. We simply could not see any indications of overfitting in the static runs wich should lead to a higher fitness in that case. There was a slightly higher variance for the fitness in the dynamic runs wich we attributed to the randomness of the maps. The largest and most interesting differance however was the very volatile average fitness of the population during the dynamic runs. This can be seen in the picture below, the average fitness of the individuals alternates between high and low, and with a really big difference, for every single generation.



Static run



Dynamic run

We were not able to figure out with certainty what caused this. What is really curios is the strictness of the change, where every single generation following a rise in average fitness has decreasing fitness. It almost looks like the population finds a really good solution to the current map, and then next generation is kind of overfitted to the last map and performs poorly, and somehow corrects the overfitness again, but overfitting in just one generation doesnt sound that realistic, and it also wouldnt explain the strictness of the phenomena and how it jumps back up after a “poor” generation. It really just seems that somewhere there is an attribute, maybe a weight in the ANN, that at some threshold is beneficial for the individual, but if the attribute is changed past the threshold it becomes very detrimental.

5. A Working demo of Static Runs

This part is shown at the demo day, or ran from main.py without arguments.

6. A Working demo of Dynamic Runs

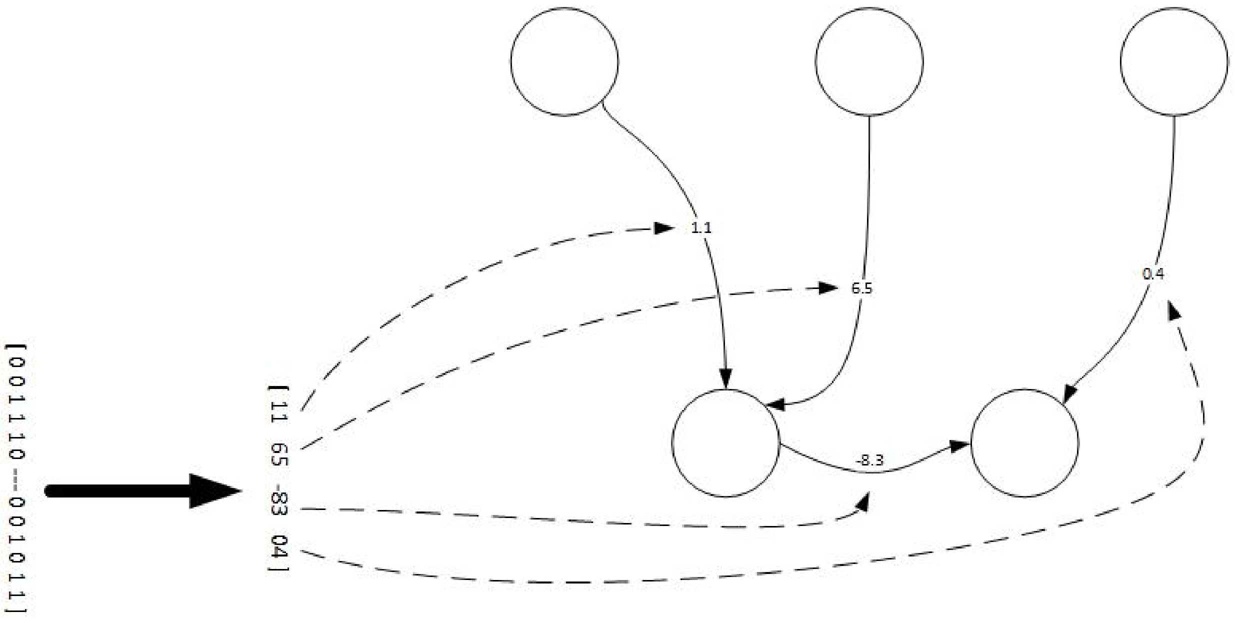
This part is shown at the demo day, or ran from main.py without arguments.

IT3708 | The Beer-Agent

Kristoffer Hagen, MTDT, spring 2014
Mads Wilthil, MTIØT, spring 2014

1. System overview

Our agent phenotypes are simply lists of integers, based on a binary string. To apply these as weights, they are scaled down before being sent to the neural net. One phenotype can evolve an arbitrary number of weights, but they need to have the same range. Hence, one phenotype is needed for each of (1) bias weigths, (2) other weights, (3) time constants and (4) gains.



We based our CTRNN on the topology suggested in the assignment text, and no modifications were made for the base case agent runs. Upon evolving, we settled on Generational Mixing for the adult selection strategy. As with the Flatland agent, a high crossover rate of 0.8 and a low mutation rate of 0.2 evolved satisfiable results. We believe the low mutation rate is needed due to our binary representation; when a bit is flipped, a mutation can change a weigth quite drastically if it flips the most significant bit. This frequency should be kept low, and mainly use the crossover operator to explore the weights in the population.

Our tracker scenarios were also constructed in accordance with the assignment text. The object size distribution was deterministic - 7 objects of sizes 1 trough 4 and 6 objects of sizes 5 and 6 - but their order was random. In the base case the objects were dropped from a height of 15, leaving the agent with 15 moves to handle it. The arena was 30 blocks wide.

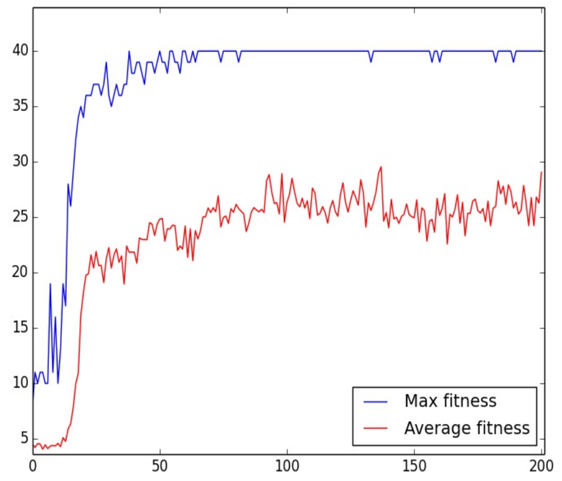
2. Catching all objects

To test the catching of all objects, we did 20 runs and logged the the number of objects caught by the best individual after 200 generations. On average, our best-of-run agents managed to catch 36.5 objects. Some degree of eliteism seems crucial to evolve the wanted behavior; we used Generational Mixing as our Adult Selection strategy and sorted the parents and children on their fitness value.

Our fitness function is quite simple. It awards 1 point for each catch, and no points for “hits” (only some overlapping pieces) or misses. This seems to be encouragement enough for the agent to evolve a “catch all”-behavior.

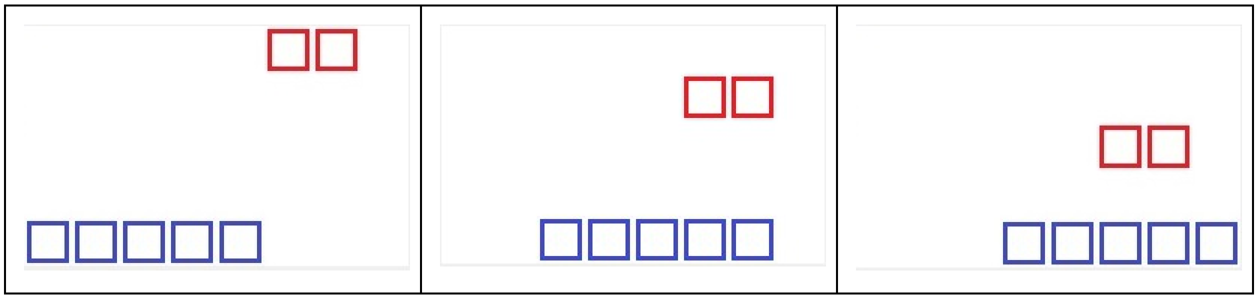
3. Catching the small and avoiding the large objects

The catching of objects only smaller than the agent were tested the same way as when catching all objects. The best-of-run agents correctly handled about 70 % of the objects encountered. We had some issues in evolving this controller: in some cases, the agent wold evolve a “catch all”-behavior. We estimate this happens about 50 % of the time. In the other half of the run, the agent has evolved some escaping behavior, though not perfect in any way. A fitness plot in the case of a successful is shown to the right. We would like to stress that this is a particulary fortunate case, and this behavior arise rarely. The average case run flattens out at a lower fitness value, usually around the 30-point mark, though the graph generally has the same structure.



The final fitness evaluation is 1 point for correctly handling an object, while deducting 1 point in the case of hitting an object of size 5 or 6. Our reasoning for not giving a penalty on the small objects is that a catching behavior was not hard to evolve, so no penalty was really needed to punish incorrect behavior. We also applied eliteism when searching for this individual.

One characteristic move we saw when evolving the smarter agents was that they would a little past the far edge of the dropped object, to check its length. The agent would then stop if the object was shorter, or keep moving if the object was longer. This is illustrated below.



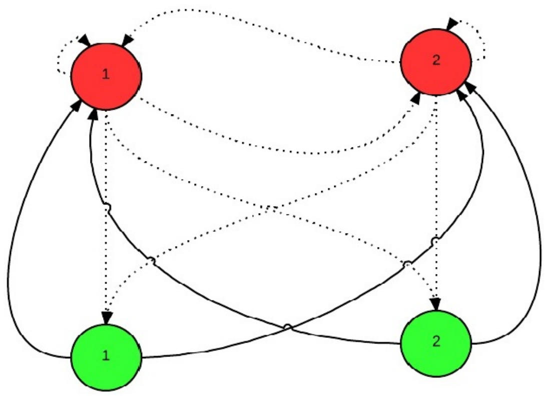
4. Modifying the tracker scenario

First, we tried decreasing the arena height from 15 to 10, essentially increasing the speed of the falling objects by 33 %. The benchmark were the same as the above; 20 runs noting the best-of-run agent performance. On average, the agents correctly handled about 40 % of the objects encountered, quite the fall from the case in 3. We believe the way we move our agent may be to blame; the motor outputs has to be high in order to move 4 squares, and they may not be fine-tuned enough for this case.

Second, we doubled the arena height from 15 to 30. We expected an increase in the agent performance this way and this was an attempt to measure that increase. One observed fact is that the agent slowed down it's search speed; with a height of 15, agents would move up to 4 squares. In the later scenario agents would usually move 2 steps, though 3 was also observed at times. On average, the best-of-run agents acted correctly in 75 % of the scenarios. Only marginally better performance were observed. After some thoughts, we concluded the agent behavior is unaffected by the time it has to react to an object, given that is has enough time to discover the size and move accordingly. The agent disregards if it must wait for 20 timesteps or 5 timesteps for the object to land.

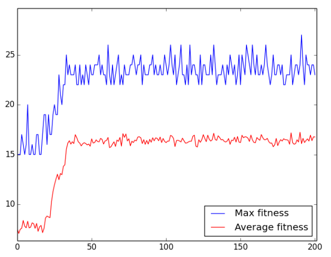
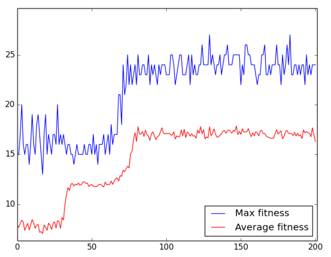
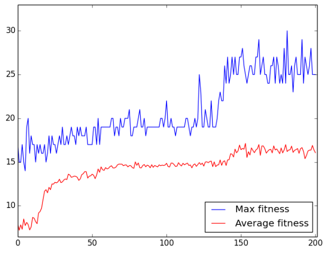
5. Modifying the CTRNN topology

To modify the CTRNN topology we decided to create a new connection between the output layer and the hidden layer. The connection we created goes from the output layer to the hidden layer. We decided to try the new connectivity hoping that the output of the output layer could prove useful for the hidden layer in the next timestep.



The old connectivity is shown with dashed lines, with the new addition shown by the filled lines. Note that the old connectivity was not removed, but the new one was added.

The average fitness after 5 runs with the new topology was 24 which is worse then what it was before the modification. The new connectivity did not find a good solution to the complete tracker, and runs ended up with catch all, or a strategy that avoided most big blocks, caught some of the medium blocks, and ignored the small ones. With the data we collected, it was not a noticable difference between the old and new topology. There might be a difference in the time it took the network to get to a decent stable strategy, the new topology seemed to do that faster by some generations, but without simulating many more runs, we cannot conclude with anything yet.

		
Fitness graph with the modified topology	Fitness graph with the modified topology #2	Fitness graph with the old topology

6. Modifying the weight range

We made two changes to the range of the evolveable weights. First, we tried including positive bias weights, evolving weights with a range of [-10 , 5]. Second, we dropped the lower bound of the gains to 0, essentially allowing agents to activate neural net nodes regardless of its internal state. These modifications were applied simultaniously to the agents. Increasing the weight range means the agent has to explore a larger search space, so we set the generation cap at 250. Best-of-run performance ended at 70 % of objects handled correctly. As expected when **increasing** the allowed range, the bias weights remained mostly unchanged; the hightest we saw trough our tests was 0.4. Some gains did drop below 1, though the lowest ended up at 0.86.

No significant change of the agent performance is due to the increase in weight range. A relaxation on the agent constraints should result in performance no worse than in the unrelaxed case. Sadly, we did not have time to restrict the agent constraints / decrease the range of the evolved parameters.

7. Analyzing the CTRNN

Seeing that some of our shorter simulations provided less then optimal results when it came to solving the full tracker task, we decided to do a large scale run overnight, with 400 in the population and over 2000 generations, the best individual had a fitness of 38 fitness and was chosen to be analyzed. In the interest of full disclosure we did not have time to run more then one of these simulations so whether this is a once-in-a-lifetime individual is uncertain. (We have since replicated this result)

The evolved network weights and attributes (continues on next page):

To From	1	2	3	4	5	6	7	8	9
6	**3,8**	**-4,8**	**2,1**	**-2,4**	**3,6**	**0,2**	**-3,9**		
7	**-3,8**	**1,3**	**4,5**	**2,2**	**-4,1**	**0,1**	**-3,3**		
8						**-4,5**	**4,7**	**-2,9**	**-4,7**
9						**1,8**	**-4,9**	**-0,4**	**-1,2**
0						**-7,3**	**-1,8**	**-8,1**	**-0,3**

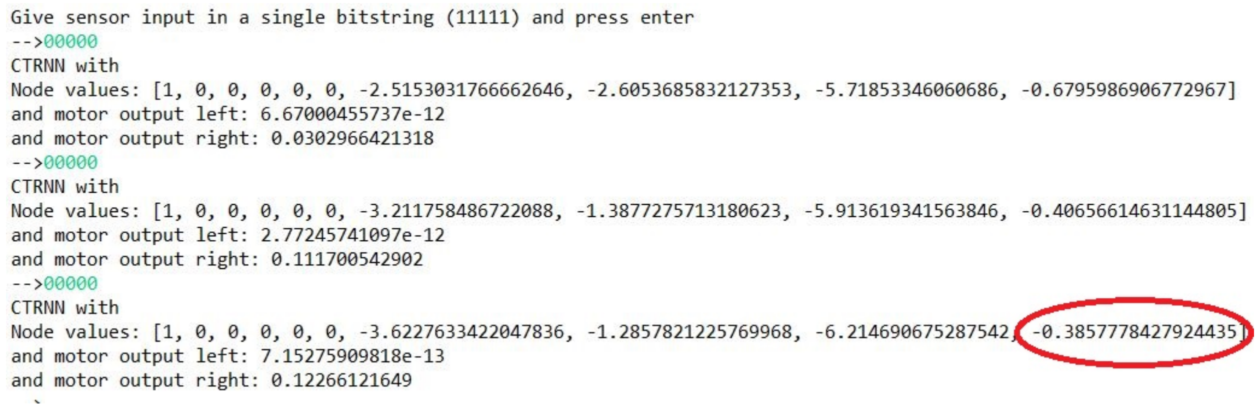
Attributes:	6	7	8	9
Gains	**5,0**	**4,2**	**4,5**	**5,1**
Time constant	**2,4**	**1,1**	**1,7**	**1,5**

By observing the behaviour, in combination with the high fitness of this individual, we saw that the performance of this one was better then most results we got in the shorter simulations where most induviduals evolved either a catch all strategy, or avoid most and catch some.

By looking at the weights and seeing the network in action, we can suggest a few key features that caused this individual to perform well. For starters, the large difference in the weight from the bias node to the right side of the network (node 6 and 8), compared to the left side (node 7 and 9), seems to enable the robot to search when no input is given. This individual is a “right searcher” while on other runs the dominating strategy was to search to the left. In addition the large difference in this case (-7,3 and -8,1 vs. -1,8 and -0,3) might determine the speed of the searching, i.e. makes the robot move “fast” when no input is given. Worse performing individuals somtimes had next to no difference in left/right bias weights which cased them to stand still, or little difference which caused them to search too slowly.

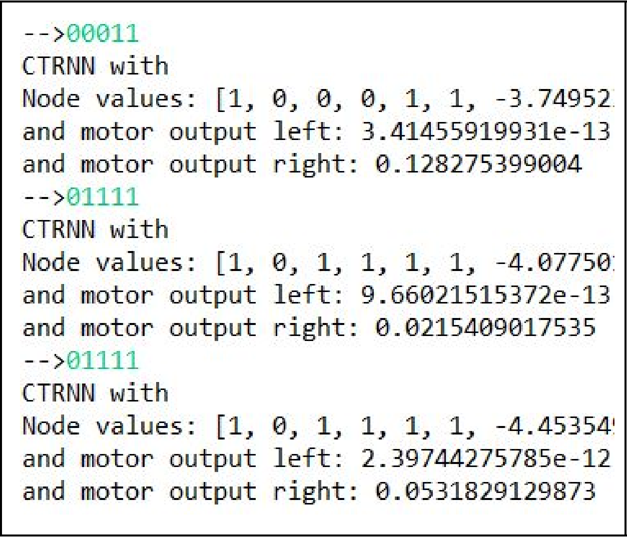
It is hard to make any strong statements regarding the weights of the network. We can observe some tendencies but its hard to draw conclusions about causality in a network this complex (and unfamiliar). It seemed, by observation, that a high average gain values was prevalent in many of the good solutions, in addition to a time constant that was not too low. Too low of a time constant seemed to make the robot unable to stop to catch the small blocks. As mentioned we had trouble making any conclusions regarding which weights do what in the network, many of the better evolved networks has very different weight values and it would take a very long time to fully analyze and understand the network.

Now we will isolate the network and feed it input manually and watch what happens with the values and outputs of the internal nodes and see if we can identify some key features that leads to a good fitness value. We are using the evolved networkmentioned above.

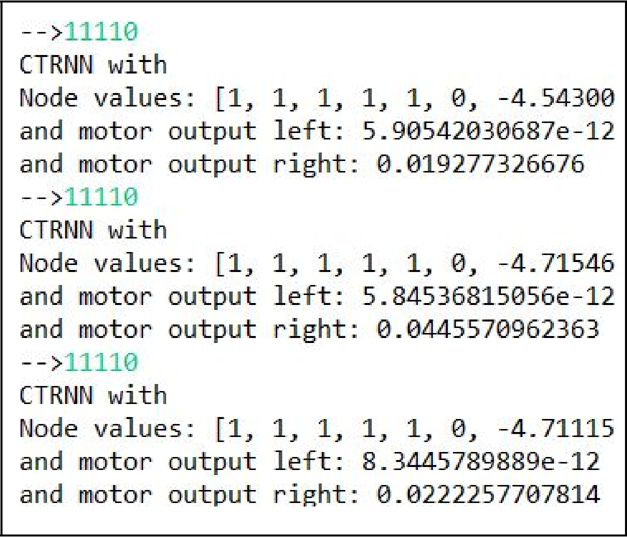


```
Give sensor input in a single bitstring (11111) and press enter
-->00000
CTRNN with
Node values: [1, 0, 0, 0, 0, 0, -2.5153031766662646, -2.6053685832127353, -5.71853346060686, -0.6795986906772967]
and motor output left: 6.67000455737e-12
and motor output right: 0.0302966421318
-->00000
CTRNN with
Node values: [1, 0, 0, 0, 0, 0, -3.211758486722088, -1.3877275713180623, -5.913619341563846, -0.40656614631144805]
and motor output left: 2.77245741097e-12
and motor output right: 0.111700542902
-->00000
CTRNN with
Node values: [1, 0, 0, 0, 0, 0, -3.6227633422047836, -1.2857821225769968, -6.214690675287542, -0.3857778427924435]
and motor output left: 7.15275909818e-13
and motor output right: 0.12266121649
```

Here we feed the network 00000 (no blocks above) input for 3 timesteps. We can see that the right output node stabalized at about -0.38, thisleads to a output of 0.12 to the right which translates to about two blocks every step. This searching behavior is key to attain a good fitness.

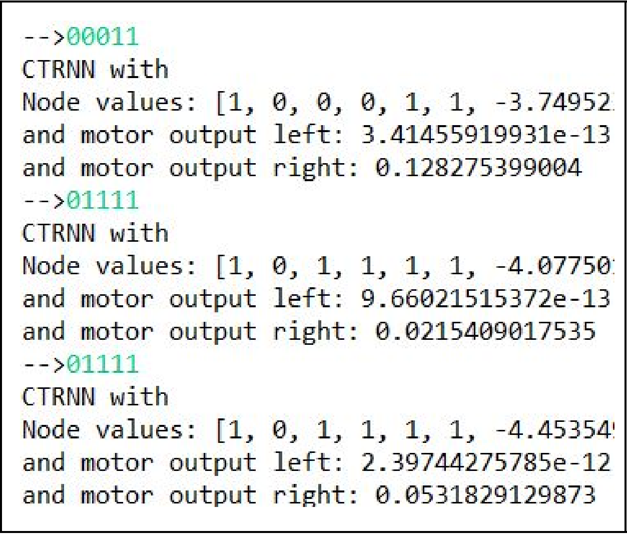


```
-->00011
CTRNN with
Node values: [1, 0, 0, 0, 1, 1, -3.74952
and motor output left: 3.41455919931e-13
and motor output right: 0.128275399004
-->01111
CTRNN with
Node values: [1, 0, 1, 1, 1, 1, -4.07750
and motor output left: 9.66021515372e-13
and motor output right: 0.0215409017535
-->01111
CTRNN with
Node values: [1, 0, 1, 1, 1, 1, -4.45354
and motor output left: 2.39744275785e-12
and motor output right: 0.0531829129873
```

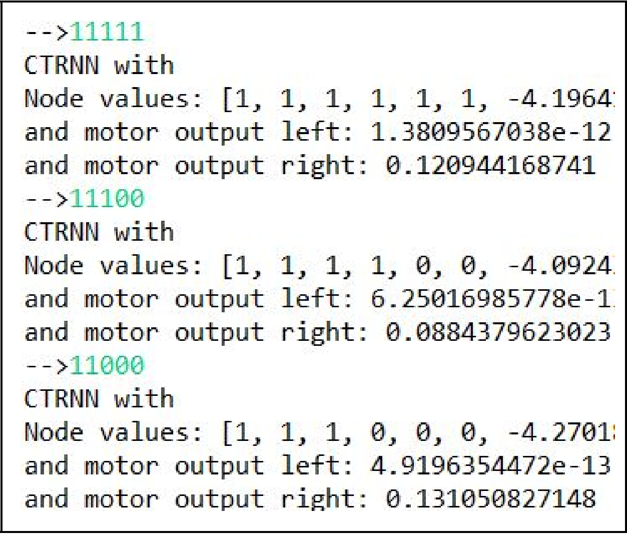


```
-->11110
CTRNN with
Node values: [1, 1, 1, 1, 1, 0, -4.54300
and motor output left: 5.90542030687e-12
and motor output right: 0.019277326676
-->11110
CTRNN with
Node values: [1, 1, 1, 1, 1, 0, -4.71546
and motor output left: 5.84536815056e-12
and motor output right: 0.0445570962363
-->11110
CTRNN with
Node values: [1, 1, 1, 1, 1, 0, -4.71115
and motor output left: 8.3445789889e-12
and motor output right: 0.0222257707814
```

Next we approach a 4-size block. In these 6 timesteps we can see the network pause abit in the 2nd frame, move it one forward in the 3rd and then keeping it above the robot in frame 4 to 6 as it is ready to catch it.The key here is the ability to slow down when a block is found, identify the size and then go to a full stop if it is a small catchable block.



```
-->00011
CTRNN with
Node values: [1, 0, 0, 0, 1, 1, -3.74952
and motor output left: 3.41455919931e-13
and motor output right: 0.128275399004
-->01111
CTRNN with
Node values: [1, 0, 1, 1, 1, 1, -4.07750
and motor output left: 9.66021515372e-13
and motor output right: 0.0215409017535
-->01111
CTRNN with
Node values: [1, 0, 1, 1, 1, 1, -4.45354
and motor output left: 2.39744275785e-12
and motor output right: 0.0531829129873
```



```
-->11111
CTRNN with
Node values: [1, 1, 1, 1, 1, 1, -4.1964
and motor output left: 1.3809567038e-12
and motor output right: 0.120944168741
-->11100
CTRNN with
Node values: [1, 1, 1, 1, 0, 0, -4.0924
and motor output left: 6.25016985778e-1
and motor output right: 0.0884379623023
-->11000
CTRNN with
Node values: [1, 1, 1, 0, 0, 0, -4.2701
and motor output left: 4.9196354472e-13
and motor output right: 0.131050827148
```

This last example start similar to the 2nd one with two shades showin up on the right in frame 1, the frame 1 to 3 is identical to the 2nd example. In frame 4 however it is now a 5-sized block showing up, so instead of stopping like in example 2, the robot now speeds up and gets out of the way moving 2,1 and 2 steps to the right in frame 4,5 and 6.