IT3708 Project 4

Evolving Neural Networks for a Minimally-Cognitive Agent

Iver Jordal

# Implementation

## Genotype/phenotype

The genotype is a bit string. Each “chunk” of 8 bits represents an integer. The number of bits in the genotype is determined by the number of weights needed. For example, if 28 weights are needed, then the genotype will consist of 224 bits.

Conversion to phenotype: For each chunk of 8 bits, calculate the sum of 1s and scale it to the desired range. For example biases will be in the range . One alternative technique I tried was regarding the bit chunk as binary number in the range [0, 255]. I didn’t use that because bit flips could shake up the weights too much and the crossover operation didn’t work so well.

## CTRNN implementation

The CTRNN is implemented as a class in Python. It can be initialized with a given number of input nodes, hidden nodes and output nodes. Then its weights can be set (based on the phenotype described above). The CTRNN has an *activate* method that takes in sensor data, runs it through the network and returns the output values of the output layer.

In more detail: First the input buffer is populated with sensor data, then the internal states of the hidden layer is computed. Based on this the output buffer of the hidden layer is computed. Finally the values for the output nodes are computed. The math that is used to compute the internal states and output values is based on the explanation in part 3 of the assignment text.

I’ve found that recurrence in the output layer generally made the agent performance worse. I’ve been told that it was okay to customize the topology of my CTRNN and that it was okay to have recurrence only in the hidden layer. So, adhering to Occam’s razor, I’ve decided to have only have recurrence in the hidden layer. One other customization is that I have five hidden nodes instead of two, because that improves the agent.

The Agent class, which owns the CTRNN instance, is responsible for interpreting the output from the CTRNN. The maximum motor output value is chosen to be the current movement direction, and the magnitude of that value is quantified to a number of steps in the range . In the pull scenario case there is a separate output node for the pull action. The pull action is preferred to movement if the pull node output is over the pull threshold which is

# Performance of the EA

## Standard scenario

By default the agent is moving 2 steps to the left. Whenever it occurs an item of size 1, 2 or 3, it starts jiggling, but stays below the item until it is fully obtained. Whenever the item is of size 4 the agent becomes more sceptical because it looks more like those large objects that should be avoided. Sometimes it stops to catch the item, but on other occasions it misjudges the item and moves past it. For large items (size 5 or 6) the agent jiggly scans both ends of the objects for a few time steps until it decides to quickly flee to the left. After it has safely escaped the item, it moves a bit to the right again. This is good because it delays the agent so it won’t rediscover the same item after wrapping around. In some cases, when the agent discovers a large item late, it is only able to partially avoid it.

Fitness function:

## Pull scenario

By default the agent moves 3 steps to the right. Whenever it starts sensing shadow on its rightmost cells, it spends a few time steps trying to learn whether the item is large or not. Then, if the item is large, it moves past the item. Otherwise it simply pulls the object. Sometimes the agent does not stop when it senses a small agent. It depends on which of the cells are shadowed. In other words, the agent’s behavior scanning behavior still has room for improvement. However, in the pull scenario the most important thing is that the agent learns when it is smart to pull and when it is not.

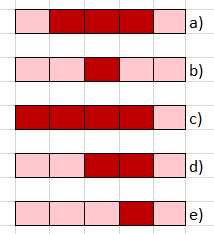


Figure 1: Some patterns that can trigger the pull action. Pattern c) is interesting, because the agent doesn’t always pull in this situation. If the agent has memorized that it is fleeing from a large object, it does not pull in when it senses this pattern. One other situation related to pattern c), which highlights a weakness:

* The agent has learned/memorized that the item is small (size 4) and pulls the item
* A new, large item appears and immediately shadows the same cells as the small item
* The agent pulls the large object, because it is still in a “this item is small, let’s pull it” kind of state

Fitness function:

Where is defined as and is defined as

## No-wrap scenario

This agent’s main purpose is to learn how to scan the entire world when there’s a “wall” on each side that stops it. It has a wall sensor on each side. Indeed, the agent has learned to move in the opposite direction when it hits a wall. If bouncing from the left wall, it keeps moving right for at least half of the world’s width, before it starts moving left again. Optimally, the agent should have remembered the left wall hit for a longer time and kept moving right until it’d find an item. However, it has learned the optimal technique for the opposite wall: When it bounces from the right wall, it can correctly keep moving left for the entire world or until it finds an item. When it finds an item, it reacts accordingly. It is able to catch most small objects (size 1-3). For these small items, the agent uses the well-known jiggly technique to move back and forth to stay in place below the item. It has problems with distinguishing between objects of size 4 and larger objects, though. The reason why it fears objects of size 4 is probably that it has learned to avoid large objects, and items of size 4 can look pretty much like larger objects.

# Analysis of an evolved CTRNN

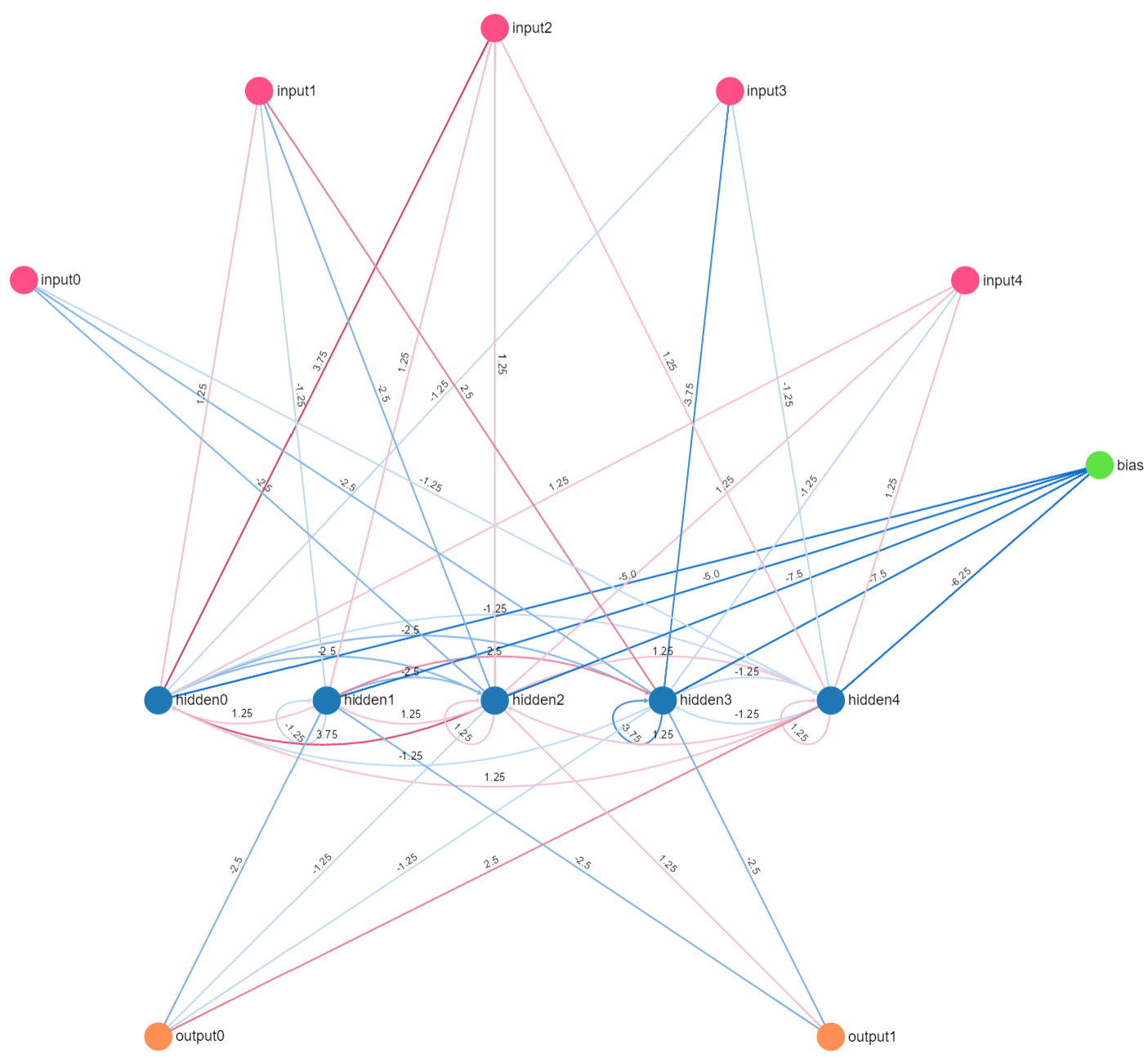


Figure 2: Visualization of an evolved CTRNN for the standard scenario. Blue edges: Negative weights. Red edges: Positive weights. Weak edge color means small magnitude. Weights with value 0 are not drawn.

Hidden0 doesn’t send any signal directly to the output nodes. It acts as a memory node that of feeds a representation of its state to hidden2, which is the only hidden node that sends positive signals to the right motor output node. The middle shadow sensor, input2, has a strong weight to hidden0. In other words, whenever a shadow is cast on the middle of the agent, the state of hidden0 is “charged”, and this affects the signal that the other hidden nodes receive in the current time step and in the future.

Hidden1, hidden2 and hidden3 act as inhibitors. Whenever they send out signals, leftwards movement is inhibited. In case of high activation levels in hidden3, the agent will move right rather than left.

Hidden4 has a strong weight to output0, which moves the agent to the left. This is probably the node that by default moves the agent to the left when it is looking for items.