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 $[-10, 0.0]$. One alternative technique I tried was regarding the bit chunk as binary number in the range [0, 255]. I ended up not using that because bit flips could shake up the weights too much and the crossover operation didn't work so well with the representation.

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 earlier). 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. For more details see the code.

I've found that recurrence in the output layer generally made the agent performance worse. I've been told that it is okay to customize the topology of my CTRNN and that it is also 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 deviation from the suggested topology is that I have five hidden nodes instead of two, because that yields better agents more easily.

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 $[0, 4]$. 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 0.5.

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 the world. In some cases, when the agent discovers a large item late, it is only able to partially avoid it.

Fitness function: $1 * small\ captures + 1.5 * large\ misses - 3 * partial\ captures - 3 * small\ misses - 3 * large\ captures$

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 moving back and forth, trying to learn whether the item is large or not. Then, if the item is found to be large, the agent moves past the item. Otherwise it decides to pull 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 scanning behavior still has room for improvement.

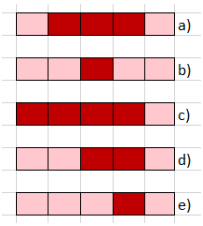

Figure 1: Some patterns that can trigger the pull action. Pattern c) is interesting, because the agent doesn't always pull in this situation. If the agent has memorized that it is fleeing from a large object, it does not pull when it senses this pattern. One other situation related to pattern c), which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

- The agent has understood and 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, I'll pull it" kind of state

Fitness function: $1 * small\ captures + 1.5 * large\ misses - 3 * partial\ captures - 3 * small\ misses - 3 * large\ captures + 1 * good\ pulls - 1 * bad\ pulls$

Where $good\ pulls$ is defined as $(small\ capture \,||\, large\ miss)$ and $bad\ pulls$ is defined as $(partial\ capture \,||\, small\ miss \,||\, large\ capture)$

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. When "bouncing" from the left wall, it keeps moving right for at least half of the world's width (if no item is seen), 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 on the rightmost side of the world. Luckily, it has learned the optimal technique for the opposite wall: When it bounces from the right wall, it correctly keeps moving left until it either hits the left wall or senses 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 previously mentioned jiggly technique to move back and forth to stay below the item. It has some difficulties 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.

Fitness function: $2 * small\ captures + 1.5 * large\ misses - 3 * partial\ captures - 3 * small\ misses - 3 * large\ captures + 1 * placement\ reward$

Where $placement\ reward$ is calculated as follows:

$$left\ proportion = x_l/(x_l + x_r)$$

$$deviation = abs(0.5 - left\ proportion)$$

$$placement\ reward = 15/(1 + 5 * deviation)$$

Where x_l is the number of timesteps the agent was in the left half of the world and x_r the number of timesteps the agent was in the right half of the world.

Analysis of an evolved CTRNN

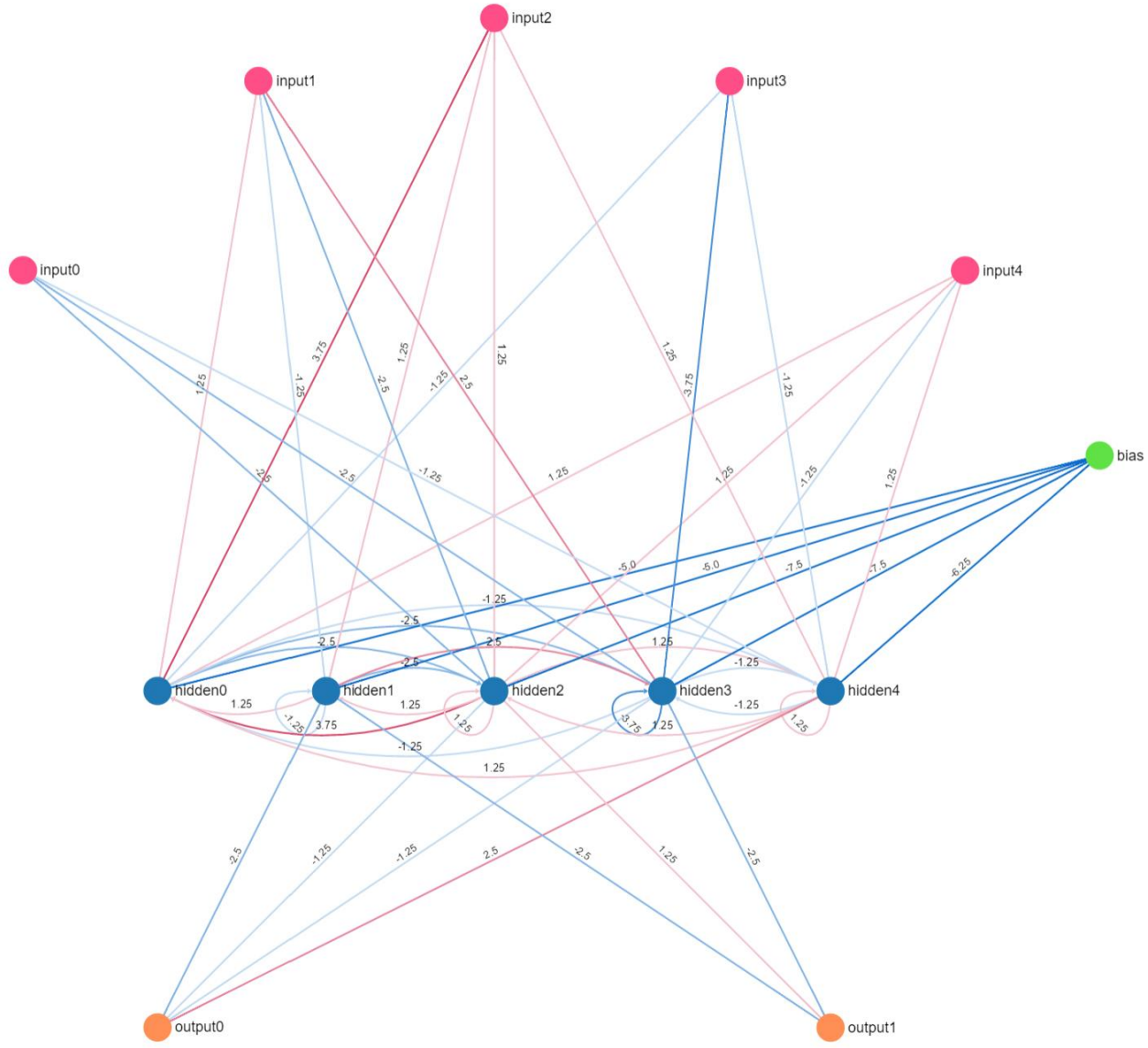

Figure 2: Visualization of an evolved CTRNN for the *standard* scenario. Blue edges: Negative weights. Red edges: Positive weights. Weak edge color means small magnitude. Weights with value 0 are not drawn.

Hidden0 doesn't send any signal directly to the output nodes. It acts as a memory node that strongly feeds a representation of its state to hidden2 (amongst others), 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. It is not immediately clear what the value of the node means, because it plays a role in a large context. It is natural to example suggest that this is one of the nodes that remembers if there is a large object or not. A good, real-time visualization of the activation levels of the nodes would help in understanding the role of the node.

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.

Example activations

Here are some consequent inputs and resulting output values from the CTRNN evolved in the *pull* scenario.

```
[0, 0, 0, 0, 0] -> [0.498805, 0.500812, 0.497859]  # move right by default
[0, 0, 0, 1, 0] -> [0.499881, 0.499664, 0.500003]  # pull that tiny object
[0, 0, 0, 0, 0] -> [0.499984, 0.500001, 0.499979]  # move right
[1, 1, 1, 1, 0] -> [0.499801, 0.499116, 0.499957]  # not sure if large, go left
[1, 1, 1, 1, 0] -> [0.499289, 0.492673, 0.501247]  # pull
```

My first thought is that these values are all surprisingly close to 0.5. This may be a problem because the agent doesn't harness the power of moving quickly (4 steps). It could probably be fixed by tweaking the CTRNN implementation. However, it doesn't seem to negatively affect how the agent determines which action to take (i.e. move *left*, move *right*, pull).

The two last activations have the same input values, but the last output has changed from the second last. Instead of wanting to *go left*, the agent now wants to *pull* the object, because it thinks that it has learned something about the size of the object. This shows how the memory of the network can give different output for the same input based on earlier input.