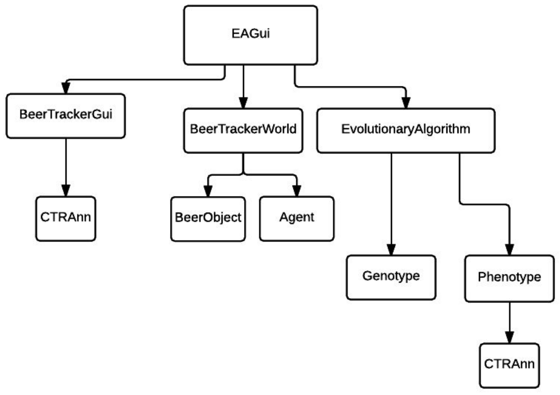
# Project 4

Subsymbolic Methods in AI
Håkon Gimse

## Architecture and implementation



Class diagram

The class diagram and general implementation remains roughly the same as it was in exercise 3.

**Genotype/Phenotype representation and conversion between them.**
The genotype representation also remains the same. An array of integers. The range of each of these integers are configurable in the GUI. For this assignment 256 different integers are used while testing as the assignment hints that this should be enough to solve the problem. The specific phenotype implementation for this problem has some different characteristics than the last exercise had. The development function, developing a genotype array to a phenotype array has to consider that each integer in the genotype represents different things. It starts with all the standard weights, then comes neuron-specific bias weights, gains and lastly time constants. Each of these types are transformed from a integer between 0 and N to the given ranges in the assignment. The result is a phenotype with an array of values, each type in the specified range.

**Implementation of CTRNN**
The weights are still held in a 3-dimensional array. In this array I append all the recursive weights. Each hidden node has a connection with a weight to all the other hidden nodes in that layer. I also keep an 2d array of all the neurons states. This is initially set to 0. Two corresponding arrays are made up of the neurons gains and time-constant. When a move should be predicted, I pass the input signal into the input layer. A product of the next layers neurons states is also sent through this layer of weights. This product is calculated using a sigmoid function, where the gains term is included as shown in the assignment paper. The last signal that is being sent through this layer is a 1 from the bias node. These values are then dotted with the weights and then passed through an activation function, specified in the GUI. This appending of recurrent and bias signals are done in each layer. Each time a neuron fires it changes its internal state. This is done by adding a derivative of the previous state to the state. This derivative is found using the formula:

$$\frac{dy_i}{dt} = \frac{s_i - y_i}{\tau_i}$$

Where si is the signal after activation, yi is the state and ti is the time constant.

**Prediction to maneuver**
To determine what to do I use two output nodes. The first one is to determine to go left or right. If it is above 0.5 you go right. The second is how far to go in one step. Less than 0.2 is to stand still, less than 0.4 is one step, less than 0.6 is 2 steps etc. When the pull move is introduced i add a third output node to the network. When this is above a threshold, the pull move is executed. When the agent

can not wrap around the sides of the board, two extra input nodes are introduced as proposed. I have also implemented an option to use a so called one-hot output scheme. Here you have one node for each action, so 9 in the standard scenario. The action with the highest value is executed. This proved to be less effective than the topology i ended up with.

**Performance of EA**
**Standard**
The agent moves in a steady pace one way, until it finds an object. If it is a large object it moves on, and if it is a small object it starts going back and forth underneath it. It the object is under 3 slots long it stops and waits on the spot. The agent sometimes walks into a large object just as it is hitting the ground and receives a penalty.
**No-wrap**
It is very good at stopping under small objects. It goes one step past and the to a full stop. When the agent hits a wall, it changes its general direction until it hits the opposite wall. Unfortunately it often mistakes large objects for small, and catches them as well.
**Pull**
It moves very fast through the board, and it does not always pull the small beers down, but given the high pace, it often does it the second time it sees an object. Also it sometimes pull down objects with length of 5, or does not pull those with a length of 4. This is dependant on the training.
The fitness function used are a composite of different fitness functions and their weight. The most intuitive are one for success rate in catching and avoidance. The formula is as follows:
$f_{capture} = nr\_of\_small\_taken / total\_nr\_of\_small$
$f_{avoidance} = nr\_of\_large\_avoided / total\_nr\_of\_large$
These two are used in combination on the standard and no-wrap problems. In the pull problem I introduce one fitness function for speed as well to make it utilize the pull option. This formula is as follows
$f_{speed} = f = nr\_of\_beers\_spawned / max\_nr\_of\_spawns$
Each of these count for a percentage of the total fitness. The total fitness is calculated as follows for the scenario:
$f_{standard} = f_{capture} * 0.65 + f_{avoidance} * 0.35$
$f_{no-wrap} = f_{capture} * 0.7 + f_{avoidance} * 0.30$
$f_{standard} = f_{capture} * 0.5 + f_{avoidance} * 0.1 + f_{speed} * 0.4$

**Analyse evolved ANN**
This is a printout of the weight configuration in a evolved standard nett.
```
  [[[ 4.33333333, -4.60784314, -3.62745098],
    [-3.50980392, -3.66666667,  0.80392157],
    [ 0.96078431, -3.58823529,  3.74509804],
    [ 1.70588235, -3.39215686,  4.76470588],
    [ 1.43137255,  4.05882353, -3.8627451 ],
    [ 2.17647059,  4.7254902 ,  3.66666667],
    [ 3.50980392,  2.84313725,  1.23529412],
    [ 1.47058824, -1.62745098, -1.8627451 ],
    [-8.2745098 , -4.47058824, -2.70588235]],
   [[-4.76470588,  0.96078431], [ 0.96078431,  3.90196078], [ 4.25490196, -0.41176471],
    [-0.21568627,  2.45098039],[ 0.21568627,  2.33333333], [-3.09803922, -3.80392157]]]
```
Analysing this is very hard and some of the reason we find neural nets so fascinating. Although you can say something about the values leading to the output nodes. The weights are generally more positive than negative. This results in more movement to the right than the left. It is also somewhat noticeable that weight leading into the speed node are not very high. This result in low speed. This is also reflected in the simulation as it in this run slowly moves right almost every timestep.

The same input values are often used here, e.i [0, 0, 0, 0, 0] are the most common sensor reading. The behaviour can vary even if the sensor reading does not. Here we see that the states affect the network and gives it a much broader spectrum of configurations and memory.