Solving the pole-cart problem with dynamical systems
Project report - ACIT4610

Sebastian Testanière Overskott[a]

[a]*OsloMet TKD, Mathematical modelling and scientific computing*

Abstract

This report concludes the work done on the project **Solving tasks in Open AI Gym**. The project is a part of the course ACIT4610 Evolutionary artificial intelligence and robotics, a Master's course at OsloMet fall 2022. It shows that a 1-dimensional cellular automaton and a simple neural network can be trained by evolutionary algorithms to solve the cart in the pole-cart problem. Through this report I will describe in more detail how the pole cart problem is defined and tested, what evolutionary algorithms are and how they can aid us in solving complex problems, what models are explored and used to solve the problem, the challenges and solutions in implementing and developing the solutions, and discuss the results and findings the implemented dynamical systems gives. Link to source code in Appendix A.

Keywords: CartPole problem, Cellular automata, Neural networks, Evolutinary algorithms

Email address: s331402@oslomet.no (Sebastian Testanière Overskott)

Contents

1. The CartPole environment

The target of this project is to solve the cart pole problem by training a dynamic system using evolutionary algorithms. The cart pole problem is an environment in the OpenAI gym. The goal is to balance a upright pole that is hinged to a movable cart below it [1]. The pole is unstable and fall if left alone, but it can be balanced by applying one of two actions to the cart. Move left or move right. An *episode* is a run of the environment. The *observations* of the environment are the cart position, the cart velocity, the pole angle, and the pole angular velocity. See table 1 for details and ranges for the different observations. Controlling the cart is done by passing either

Table 1: Observations in the CartPole environment

Index	**Observation**	**Min**	**Max**
0	Cart Position	-4.8	4.8
1	Cart Velocity	-Inf	Inf
2	Pole Angle	∼-0.418 rad (-24°)	∼0.418 rad (24°)
3	Pole Angular Velocity	-Inf	Inf

0 (move left)or 1 (move right) as an *action* given to the environment. The observations and action are updated each *step*. Every step we will receive new observable values (based on the cart and pole positions and velocities, and the action given in the previous step), and pass an action. A *policy* is used to determine what the action is based on the observations. Finding policies that do well (balances the pole for many steps) is key to this project. The environment also returns a *score*. The score is the number of steps the policy manages to balance the pole. The longer the pole is balanced the better the

score. It is also an initial condition when initializing a new episode. The initial values for all the observations are taking uniformly random value in the range $\{-0.05, 0.05\}$. This will make the CartPole behave different at the beginning of each episode.

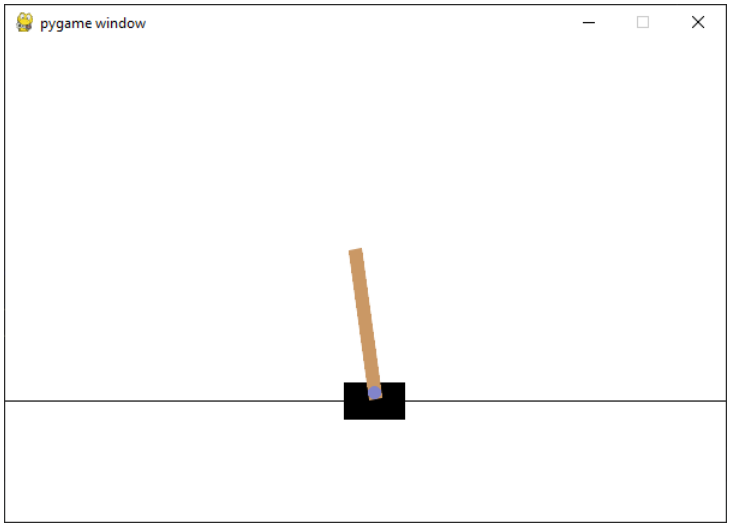


Figure 1: The visualization of the cartpole environment with the cart and pole in the center.

1.1. Termination of an episode

The episodes will terminate if one of three conditions is met. The termination ends the episode, and requires a reset of the environment to run a new episode. The first condition is the cart moving of screen i.e. the cart position exceeds ± 4.8. The other condition is the pole angle exceeding $\pm 12°$. The episode will also end if the environment reaches its step limit (set to 500 steps for v1 of the pole-cart).

1.2. Playing around with the environment

After getting the environment up and running I tried to make some simple policies to see how the environment responds. The very first policy was a simple "move right" policy. This of course was a very inefficient policy averaging a score of 9. For comparison a policy that does a completely random action on the cart gives an average score of 22.5. I then wanted to implement a simple policy that could beat random. Neither the moving right policy or the random movement policy takes any of the observations from the environment in consideration. By looking at the termination criteria that most often ends the episode, pole angle exceeding it's limit, i could try to make a policy that tries to minimize the pole angle. This policy simply checks the pole angle for each step and if it is negative (leaning to the left) it moves the cart left, and if it is positive (leaning right) it moves the cart right. This achieves average score of 42 which is an definite improvement of the two former policies, but far from the "perfect" score of 500. Now, with the basics of environment behaviour and a benchmark for lowest expected scores, I began the task with testing the dynamical systems to use to improve the results.

2. Cellular automata

Cellular automata (CA) is the name of a group of complex system models[2]. They are discrete in time and space, and the time evolution of the CA is dependent on previous states and a fixed set of rules. They have a rigid grid structure in on or more dimensions. Increasing the dimensionality also increases the CA's possible complexity. The CA consists of a set of cells that

can take on a finite number of values [3]. This project only uses type of CA, a 1-dimensional, synchronous CA, and explore how parameters affect the evolution and outcome in the space of the 1D CA.

Table 2: The attributes of the CellularAutomata1D class

Attribute	**Description**
fitness (float)	The current state of the CA
rule (bitarray)	The rule defining the evolution of the CA
size (int)	Number off cells in the CA
steps (int)	Number of steps the CA evolution should take before termination.
candidate_number (str)	An ID to identify different phenotypes
configuration (bitarray)	The current state of the CA
neighborhood size (int)	Number of neighbouring cells that will be evaluated when determinating next value for a cell
History (List)	A list of the CA's configurations (for visualizing)

2.1. Description of the CA

I'm using a 1-dimensional CA with binary valued states. Each instance can have different size i.e. number of cells and different number of time function steps it will take before termination. In the code, the CA is described by the `CellularAutomata1D` class. It inherits the `Genotype` abstract class. All class attributes and description can be seen in table [2].

2.2. Ca rules

The way a CA changes from one time step to the next is describes by its rule. The rules available for the time step function of the CA is generated by a chart of the possible neighbourhoods states for a cell and a the new cell value associated to each state. An example of how the rule 152 is applies to a cell in figure [2]. The neighborhood is tested with both size 3 and five,

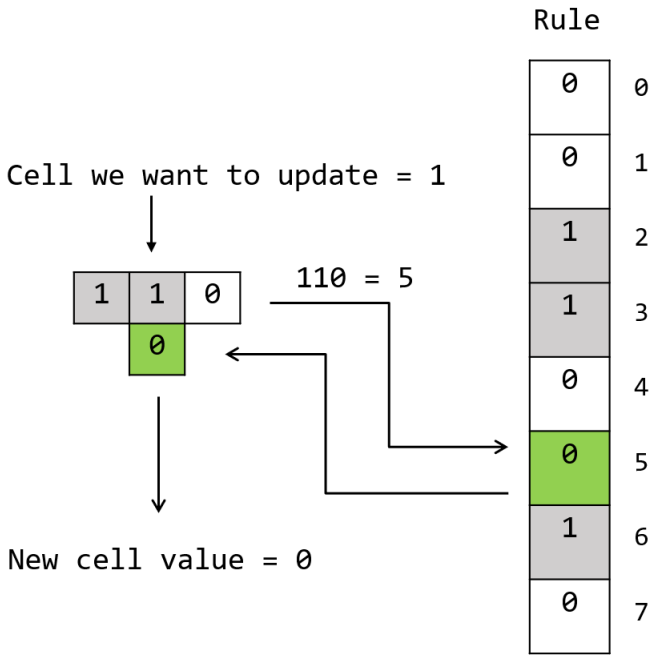


Figure 2: Chart explaining cell updating with 3-neighborhood rule

but size 5 is used as standard in the end. This is because the 2^8 rules you can have in the 3-neighborhood CA are are easy to brute force, and not very interesting to use in EA. Note, I did use the 3-neighborhood a lot for testing and troubleshooting.

2.3. Converting the observations to binary

One of the challenges of creating a CA to control the cart is translating the real valued observations received from the environment into the binary values of the CA. There are not any obvious way of encoding this. My first attempt was, inspired by the naive policy i created that looked at only the pole angle observable, a basic mapping from positive number to 1 and negative numbers to 0. This made intuitive sense since the 0 and 1 will give the CA some relevant information about the system. I chose the observations cart velocity, pole angle and angular velocity to implement in this first CA. A diagram of the encoding process can be seen in figure [3]. To my initial

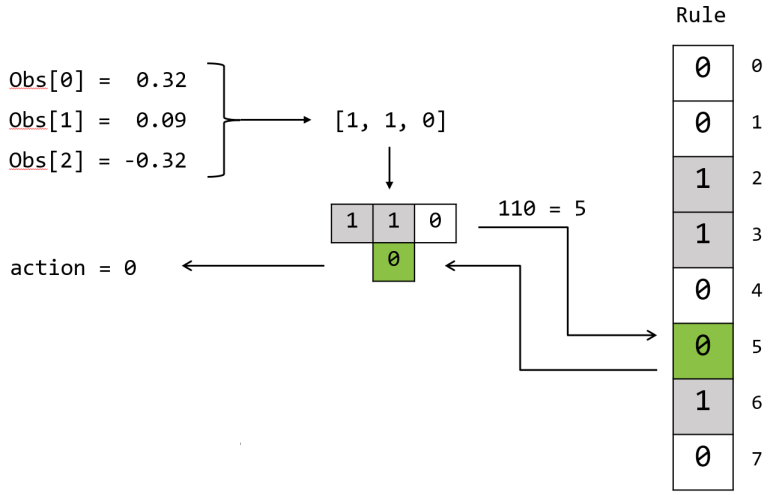
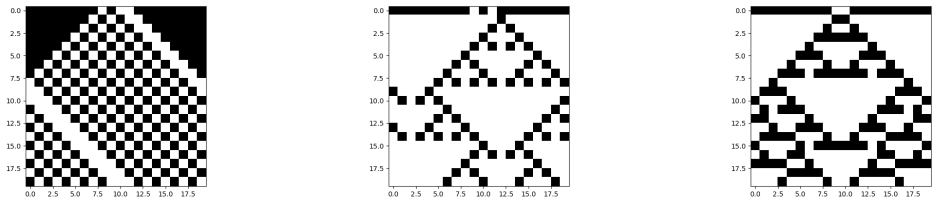


Figure 3: Flowchart on how observations are encoded into 3 cell CA and action returned

surprise, this did not perform any better than the naive implementation mentioned earlier, no matter which of the 256 rules was used. Some of the rules gave good results with one initial condition, but performed as move right policy for the rest. It also doesn't have any time evolution off the CA

state. This limits it's ability to develop and converge to something that is useful for controlling the cart.

The next attempt used the same "direct" translation from real valued observation value to a single cell state, but the CA was given some space to unfold. This gave better results than the previous encoding, but it didn't work in the evolution when it was tested in the space of 2^{32} rules of a 5-neighborhood. After some discussion with this course's teacher, it was clear that the encoding was too narrow. Both in the sense that the encoded states are being clumped too tight together, but also that the conversion form a real number to a singe binary value was truncating the observation too much. In figure [4] you can see the result of some of the encodings with a 3-neighborhood rule.



(a) $[1, 0, 1, 1]$ (b) $[1, 0, 1, 0]$ (c) $[0, 1, 1, 0$

Figure 4: Three CA evolution examples with the narrow encoding of observations

Finally, each of the observation values were encoded in a subspace of the CA trying to reflect the observable value range on the real axis around zero. The way this is done is by dividing the observable range into as many sub-ranges as the CA subspace length for each observable. The value of the observable is checked against each of the sub-ranges. If the value fits inside the sub-range the cell in CA subspace with the same index as the sub-range

is encoded as 1. A visual explanation of this can be seen in figure [5].

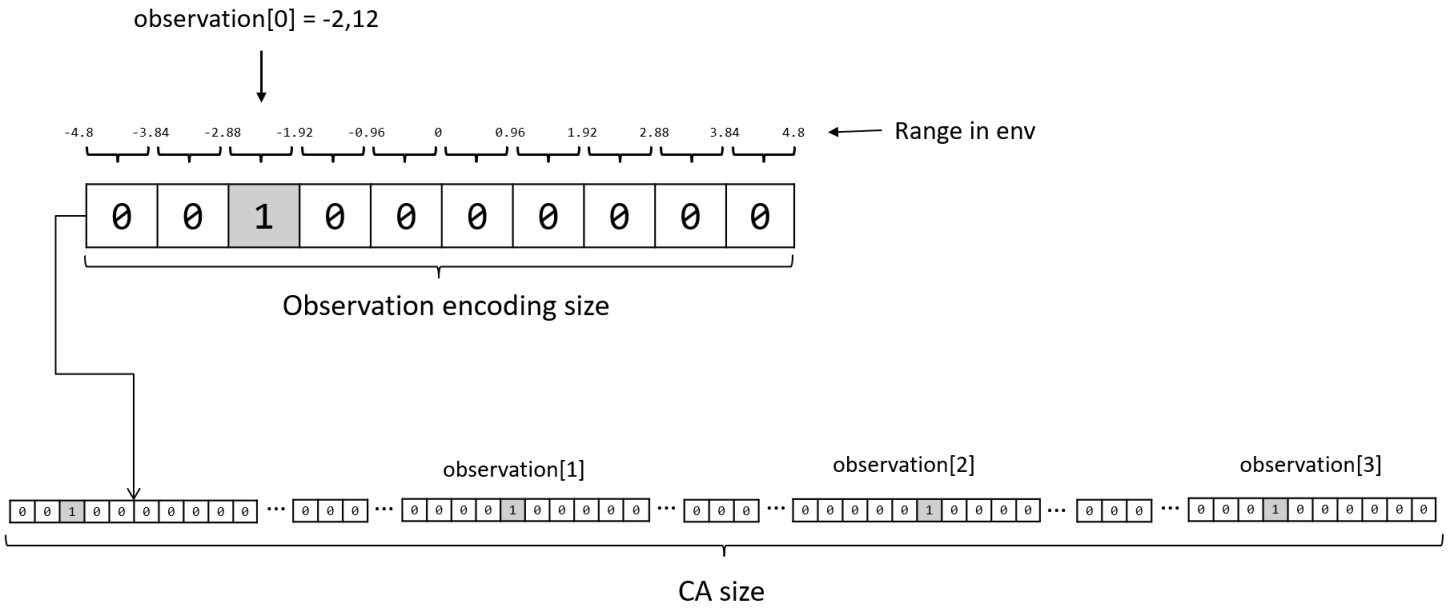


Figure 5: flow chart explaining how an observation value is encoded into the CA

The final encoding which allowed the CA to respond to the observation's "movements" and this yielded better results. One thing which took me some time to notice is that the value range from the pole angle is $\{-24°, 24°\}$, but the termination values are $\pm 12°$. After changing the encoding subspace to be in the range between the termination values instead of the given value range, the CA performed even better. This became the final encoding scheme.

2.4. The CA environment policy

The policy for retrieving an action from the CA was fortunately a simpler process than encoding the observations. I did some experimentation by selecting the middle, leftmost, or rightmost cell at the action value. This

didn't give any good results. Then i tried a simple majority vote of all the cells in the final CA state. This is done by summing up the value of each cell, divide on the number of cells and round it.

$$action = round\left(\frac{\sum_{i=0}^{n} C_i}{n}\right) \tag{1}$$

Where n is the number of cells, and C_i is the value of each cell in the final state. This turned out to be a good policy, and is the one used in the final implementation.

2.5. Evolving the CA

The initial generation of the CA is made out of individuals with a random rule, random size and step number in the range given in the `config.json`-file. The neighborhood size is fixed. All individuals are tested in the environment once before the parents are selected using FPS. Next generation is created by using one-point-crossover and mutation. The crossover only applies to the rule of the parents, while mutation generates new random values for size and steps in the same range as during initialization. The rate of crossover and mutation are set in the `config.json`-file. If crossover rate and mutation rate don't add up to 1, the difference is the rate of elitism.

3. Neural Network

The second dynamical system implemented as a genotype for the EA is a artificial neural network. Often called just Neural Networks(NN) are computing models that are inspired by the biological neurons in the brain [4]. NN is often represented as a graph of edges and nodes. The nodes are mimicking the neuron, and edges and weights the connections and connections

strengths between neurons. A NN has a input layer consisting of nodes that takes on the initial values. The input layer is connected to one or more hidden layers before ending at the output layer which is where result is read. It is also common to have a biases for each none in a layer to further tune the network. it is the weighs and biases that are trained to make the network achieve its goal. There are several ways to do this, e,g, back propagation, but in this project the network is trained by a genetic algorithm.

3.1. Description of project NN

The implemented NN has a quite simple structure. It is a feed-forward network with a input layer one hidden layer of adjustable size, and a output layer. The input layer is a four-node layer encoded with the four observations from the environment. Since a NN can handle floating numbers just fine, it is not needed to do any preparations of the observations as with the CA. The output layer is just one node e.g. the network only returns one answer. There's full connection between the layers, meaning all nodes form the input layer are connected to the hidden layer, and all nodes in the hidden layer are connected to the output node. These connections are described by their weights. Both the input layer and the hidden layer has bias values. The structure of the NN can be seen in fig 6.

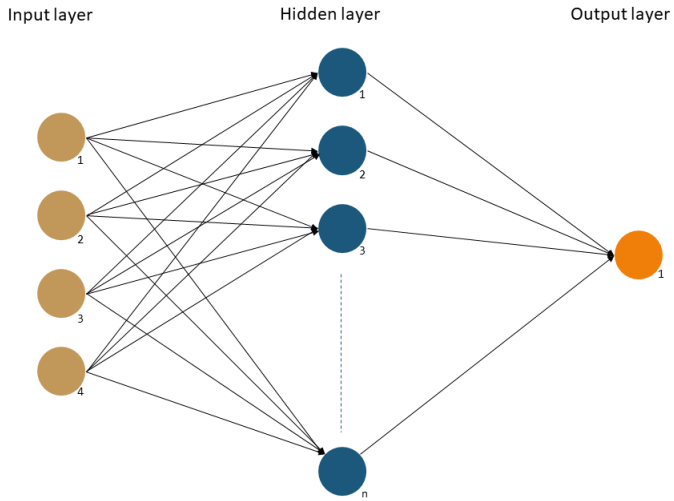


Figure 6: The structure of the NN used in this project

3.2. Implementation

Implementing the NN was an easier process than implementing the CA. A big reason for this was that many of the evolutionary functions already could be reused from the CA. Another reason is that the structure of a NN can be represented as matrices and vectors that are effortlessly calculated by libraries like Numpy. The NN in figure [6] can be written as folows.

$$(i + b) \cdot (W + h) \cdot o \tag{2}$$

Where i, b, h, o are the vectors input layer, input layer bias, hidden layer bias and output layer weights, and W is the input-to-hidden weights matrix. The result of this matrix multiplication is a scalar. I didn't implement a bias on the input layer in the first revisions of the NN code. I simply forgot. Without input bias, the solutions have very low score ($\tilde{1}0$) for some generations ($\tilde{2}0$)

before the algorithm picks up and increases the solution fitness. When i discovered the lacking input bias, I implemented it with random normal distributed values. Then the solutions are scoring a fitness of 120 right of the bat. My initial though was that this was a good thing given the results were better right away. I later decided to spilt the difference and initialize the input bias to 0 and let the algorithm find the bias. In the end i also ended up doing this with the hidden layer as well. This does in general not affect the end result of the evolution, but I got the impression from fitness curve that the evolution gets more time to explore before converging.

In the code,the `NeuralNetwork` class inherits the `Genotype` abstract class just as the CA. It's attributes with description can be seen in table [3]

Table 3: The attributes used by the NeuralNetwork class

Attribute	**Description**
input_values (array)	The four observations received from the environment
input_bias (array)	Bias for the input layer
input_weights (matrix)	Weights between input and hidden layer
hidden_bias (array)	The hidden layer bias
output_weights (array)	Weights between the hidden layer and output layer
candidate_number (string)	An ID to identify different phenotypes

3.3. NN policy

The policy of the NN takes the value calculated by the NN, feed it through the Sigmoid activation function, and then rounds the result for a binary action value.

3.4. Evolving the NN

An individual in the initial generation of NN has weights randomly generated in range -1, 1 from a Gaussian distribution. The biases are set to zero. All individuals are then tested in the pole cart environment with the observations for each step encoded in the input layer of the NN. Then FPS is used to select parents for the next generation. The crossover happens pairwise between the parents weights and biases swaps i.e. values of the input weights matrix of one parent are swaped with input matrix weights of the other. During mutation, 50% of the values are swapped. Mutation happens with equal probability on one of the NN attributes, and randomly generates new values for 30% of the attribute. The rate of crossover and mutation are set in the `config.json`-file. If crossover rate and mutation rate don't add up to 1, the difference is the rate of elitism.

4. Evolutinary algorithms

Evolutionary algorithms (EA) are heuristic methods to find solutions to many types of problems [5]. They are inspired by biological evolution, and usually does not have assumptions on the underlying problem space which makes them versatile. In short, the EA generate a initial generation of many candidate solutions (called candidates, individuals or solutions) which then is compared against a fitness function. The solutions that performs the best (i.e. best fitness score) are selected to generate a new generation. The idea is that the combination of two good solutions are more probable to be good, and possibly better.

A more concrete approach to EA are the genetic algorithms(GA). They

are inspired by the structure of genes and DNA [6]. The properties of an individual are encoded in a string of letters or numbers called a chromosome. This string/chromosome will represent the "genes" of the individual. More details on how the genetic structure of each individual is created will follow in the description of the solution types. the different procedures for moving candidates form one generation to the next is called *genetic operators*. Of the possible actions that can bring genes from one generation to the next, I have been utilizing three operations: *crossover*, *mutation,* and *elitism*

4.1. Crossover

The crossover operation consists of mixing the chromosomes of two individuals called parents. The parents are selected related to fitness, and the chromosomes are then divided into one or more parts. The part from the two parents are then combined to create new chromosomes that will be individuals in the next generation. An example can be seen in fig 7. This process preserves some of the desired properties of the parents (exploitation), while still adding some randomness to the solution search(exploration).

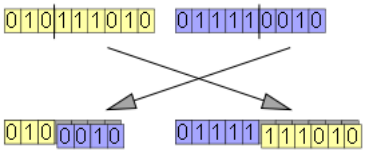


Figure 7: An example of crossover done on parents with a chromosome of binary values. Gathered from [6]

4.1.1. One-point crossover

One pint crossover is the type of crossover from the example in fig 7. I used one point crossover as the crossover scheme for the CA rule. The rules

in the CA are encoded as binary arrays, and are a convenient format for one point crossover. This is done by generating a random index, split to parent rules-array at the same point, swap the "pieces" between the parents, and the two new rules are passed on to two new children.

4.1.2. Uniform crossover

Uniform crossover uses a list of random indexes to select and swap values between two candidates genomes. Typically 50% of the values are swapped. This is used in the NN implementation where one-point crossover is harder to do with matrix representation.

4.2. Mutation

A mutation in the sense of genetic operators are a tiny random change on the chromosome of an individual. The simplest example is a flip of a random bit in a bit string. Mutation add more exploration to the pool of solutions, and is important to avoid getting stuck in local minima in the search space.

4.3. Elitism

Elitism operator is when an individual is added to the next generation without any changes (e.g. mutation). This is a strong restriction of the exploration of the solution space, but can be important to preserve a good solution. It also can be beneficial to test an individual more than once. I.e. in the cartpole environment, the initial conditions changes from episode to episode, and a solution that has a high fitness in one episode may score poorly in others.

4.4. Fitness functions

The cartpole environment returns a reward for each step the episode runs. A very natural fitness function is just summing these rewards.

$$F = \sum_{i=1}^{term} 1 \tag{3}$$

Where *term* is the number of steps on termination. The fitness score will range $\{1, 500\}$ since you are given a reward automatically the first round. Obviously, this means a higher score is better, so we want an individual to have as high a fitness as possible. Since the cartpole environment have a random initial condition, the same solution will yield different results based on chance. A solution that achieves full score (fitness of 500) will often be scoring very poorly in other initial conditions. I therefore tried to run each solution several times with different initial conditions and let the fitness be the average over that.

$$F_{avg} = \frac{\sum_{j=1}^{n} F_j}{n} \tag{4}$$

Where F_i are the fitness of run i, and n is the total number of runs. I ended up using the F_{avg} only on the NN because the increase in runs made the CA algorithm rum too slowly. The CA evolution is instead only terminated by max generations, or user stop of the evolution. This is to prevent the termination to occur if the solution happen to score higher than the termination fitness by luck very early in the evolution.

4.5. Selecting Parents

I used the Fitness proportionate selection(FPS) to choose the parents to create the next generation. FPS adds some randomness to the parent selecting process to retain a bit of the exploration of possible solutions, while still preferring candidates with highest fitness. The probability for a individual to be selected as parent is the candidate's fitness normalized over all fitnesses:

$$p_i = \frac{f_i}{\Sigma_{j=1}^{N} f_j} \tag{5}$$

Where p_i is the probability of the candidate i to be selected, f_i the candidates fitness. The selection procedure then draws a random number r in range $\{0, 1\}$. Then if $r < p_i$ then p_i is selected as parent. If not you check $r < p_1 + p_2$ to see if p_2 is the parent and this continues to a parent is selected. in pseudo code:

```
r = random(0,1)
probability = 0


for candidate in(candidates):
    probability += candidate
    if r < probability:
        return candidate
```

I also tested a modified FPS that uses the squared of the fitness to calculate probabilities. This punishes low fitness scores more, and reward high fitness scores. After some testing it seems that it preferred high fitness to a

degree where the randomness disappeared. This removes the point of FPS, and i kept using the "un-squared" selection.

$$p_i = \frac{f_i^2}{\Sigma_{j=1}^{N} f_j^2} \tag{6}$$

5. Implementation and code structure

The implementation for this project is done in Python. Code can be found at `https://github.com/Overskott/OpenAIgym-project`. Here i will briefly present the project and code structure. The main part of the code is divided in to separate files in the `src` modules. `genotypes.py` handles the classes for instantiating phenotypes of the `1DCellularAutomata` and `NeuralNetwork` classes. it also contain the `Genotype` abstract class to ensure function of finding and returning fitness is present i all genotypes. The module `generation.py` handles the instantiating and storage of a generation in the evolution such ass initializing a generation and getting the stored fitness for individuals. `evolution.py` is the collection of genetic operators and support functions for evolution of the genotypes. `utils.py` contain support functions that handler operation not relating to genotypes and evolution e.g. storage and conversion between binary and integer numbers. The `policies.py` file contains the different policies to be used in the cart pole environment. `config.json` contain the parameters and hyper parameters not involved in the evolution. While size and number of steps are subject to mutation in the CA, the neighborhood size is not. the same with hidden layer size in the NN. `config.py` loads and stores the parameters. Functions that are used in early or testing stages of the project, but do not have a purpose

in the final version of the code are annotated with `@DeprecationWarning` instead of deletion. This is for generating results with earlier versions of the code if needed. The folder `scripts` contain script for running the code and getting results and figures. `ca_environment.py` and `nn_environment.py` are for running the cartpole environment with respectively CA genotype or NN genotype. These script also saves details about the best individual at the end of a run, and figures with fitness evolution over time. `ca_visualization.py` is for generation the CA propagation figures and `visualize_env.py` is to visualise a solution steering the cartpole. The result are stored in the `results` folder found at root of the project.

6. Results, findings and conclusion

There are many parameters and hyper parameters to tweak when running the evolutionary algorithm. There are the evolution parameters e.g. generation size, number of generations and rates for mutation and crossover, the phenotype parameters e.g. CA size, CA neighbourhood size, neural network hidden layer size, and also some parameters regarding the environment e.g. randomly seeding or not and number of test to average fitness over. This makes it a very complex task optimize all parameters for best results. I have instead tested some selected parameters, compared the results to see how they make the EA behave. Some of the experimental result will be presented just as a conclusive text, with no graphs or numbers to support it. This is to increase the readability due to the sheer amount of tables and figures needed.

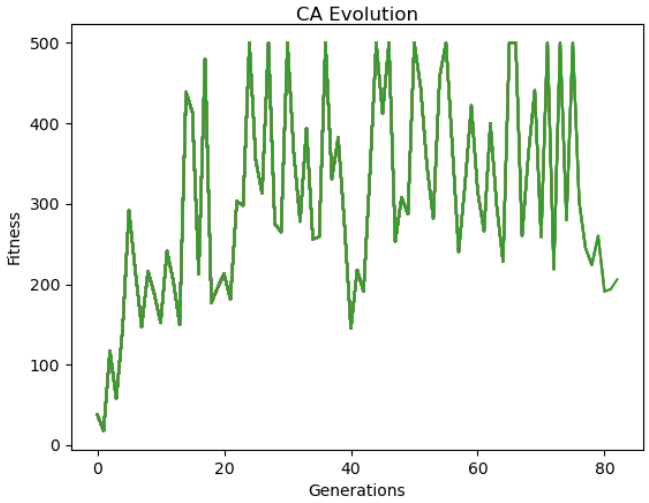


Figure 8: Example of a fitness score through a CA evolution

This solution had an average fitness over 50 runs of 210.82. It took about 80 generations and 70.25 minutes to evolve. it increases in the beginning, but converges after about 40 generations and an encoding size of 10. I think that over 200 is a good average score for the pole cart problem which can be challenging with the random initial conditions. I find it very surprising that a 1D CA is capable of do this given its simplistic nature. Not all parameters and evaluations evolved states resulting in good results. There is very varied range of solutions. Here i showcase some patterns generated by evolved CA with the parameters and fitness scores denoted.

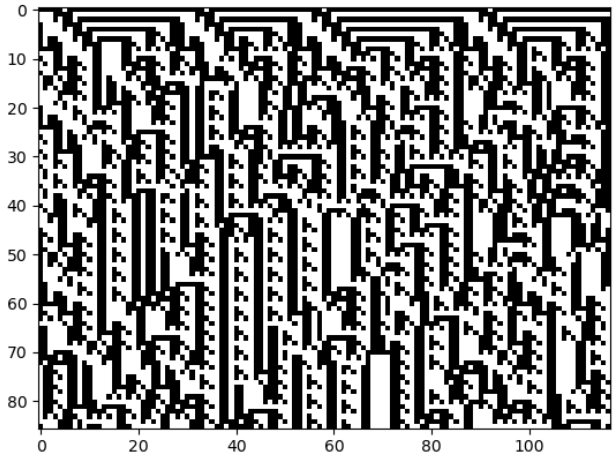


Figure 9: Encoding size: 11, crossover: 0.8, mutation: 0.1, size: 117, steps: 86, fitness: 25

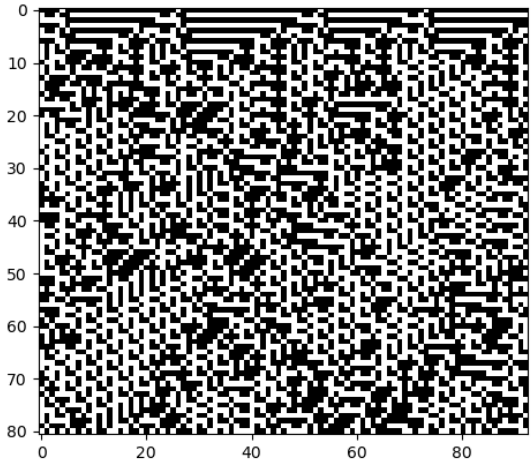


Figure 10: Encoding size: 8, crossover: 0.7, mutation: 0.2, size: 93, steps: 81, fitness: 52

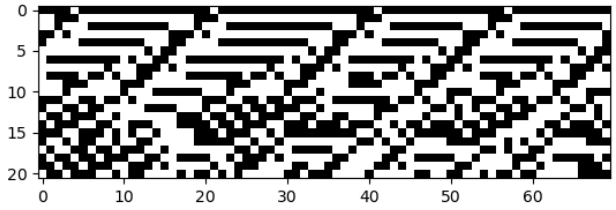


Figure 11: Encoding size: 6, crossover: 0.2, mutation: 0.6, size: 70, steps: 21, fitness: 182

Figure [9] shows that we can have a quite good solution (fitness 182 averaged over 50 runs), without a very deep CA i.e. many steps. This was a bit counter intuitive for me. I expected the CA needed more steps to converge in order to get good results.

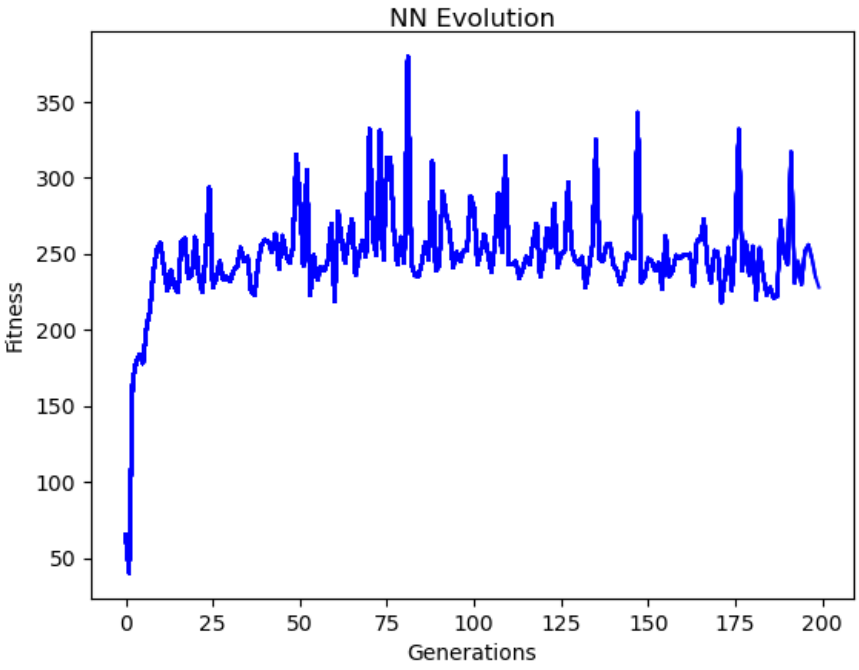


Figure 12: Example of a fitness score through a NN evolution

figure [12] displays the evolution of the best fitness in each generation. The evolution is done with a generation size of 100, 15 hidden nodes, and three runs for each candidate before finding fitness. The resulting solution has a fitness of 213.96 averaging over 100 runs. This took 6 min. The fitness seems to converge after only 25 generations. this might be of a big population size, or i could try to adjust the crossover and mutation ratios. I this case they were 0.3 and 0.6.

6.3. Hidden layer size importance

I wanted to invest the importance of the size of the hidden layer in the NN. since the algorithm has no access to change this during evolution, it was interesting to see if it improved the results in any way. In figure [13] we can see that the size of the hidden layer doesn't really change much in the fitness outcome.

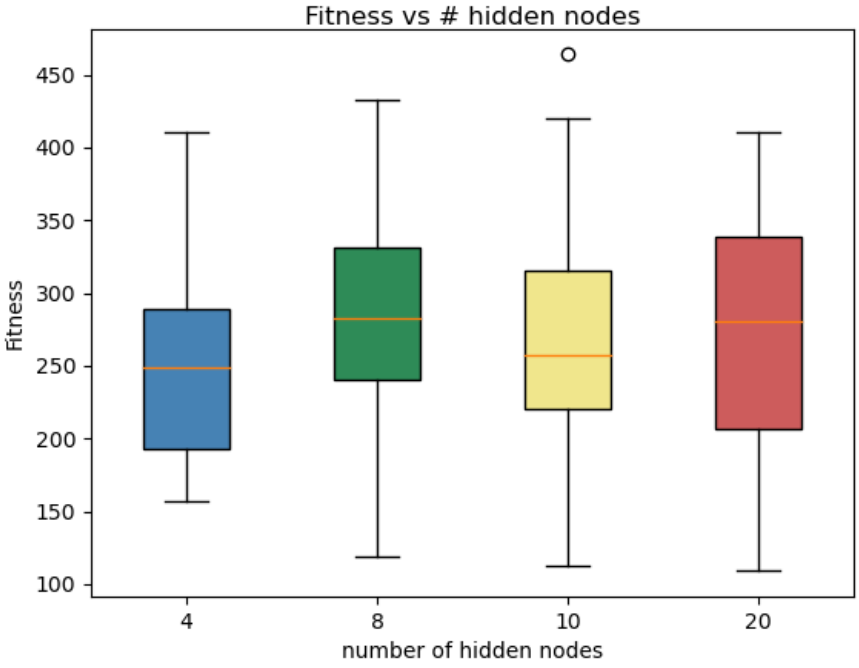


Figure 13: Fitness for 50 runs after evolution with different hidden layer size

Here i have collected 50 runs with the best candidate from each evolution. the average fitness is 250-300 and the spread is pretty similar as well. I suspect this lack of influence is because the problem is quite simple to the potential of NN, and the complexity a big hidden layer gives is not beneficial

for this problem.

6.4. Crossover vs mutation

The ratio between crossover and mutation is not always given, so i wanted to do some research in how the ratio between them affect the solution. I did some light tweaking to see if i could get wiser, but the heuristic nature of EA makes it difficult to see if small changes have effects. I instead tested four extremes. Full mutation, full crossover, 50/50 mutation and crossover, and only elitism. In figure [14] we see the fitness of 100 runs for each of the solutions found with the given parameters.

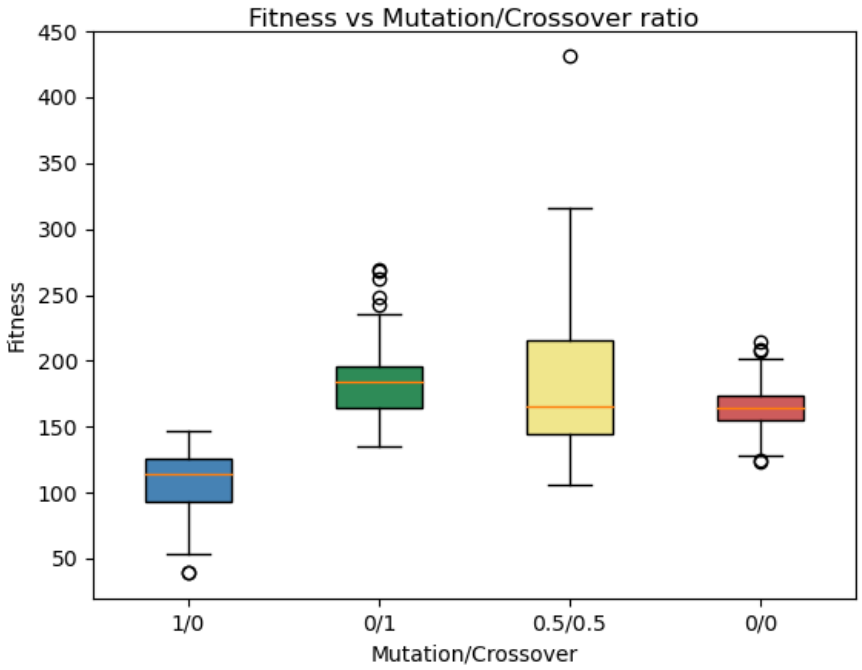


Figure 14: 4 different solutions with different M/C ratios over 100 runs.

Here we see that only using mutation doesn't really work. It has the

lowest score, both average and outlier. This is quite obvious since we are not following any trends in increasing the fitness, only randomly exploring new solutions. Only using crossover on the other hand, actually gives very good result in this case. This is not true in general according to literature, but with big enough generation size, you can evolve quite good solutions with only mixing the genes in the population. A mix on mutation and crossover gives, as expected, also good results. The last box is from only using elitism, which is basically taking the best individual from the first generation and a result of EA. It scored better than expected in this test, but this is due to chance.

6.5. Generation size

The number of individuals in each generation plays an important role exploring the solution space. This can been seen in figure [15]. Here we see that the average fitness and variety in solutions increases with the generation size. Of course this comes at the cost of increased run time.

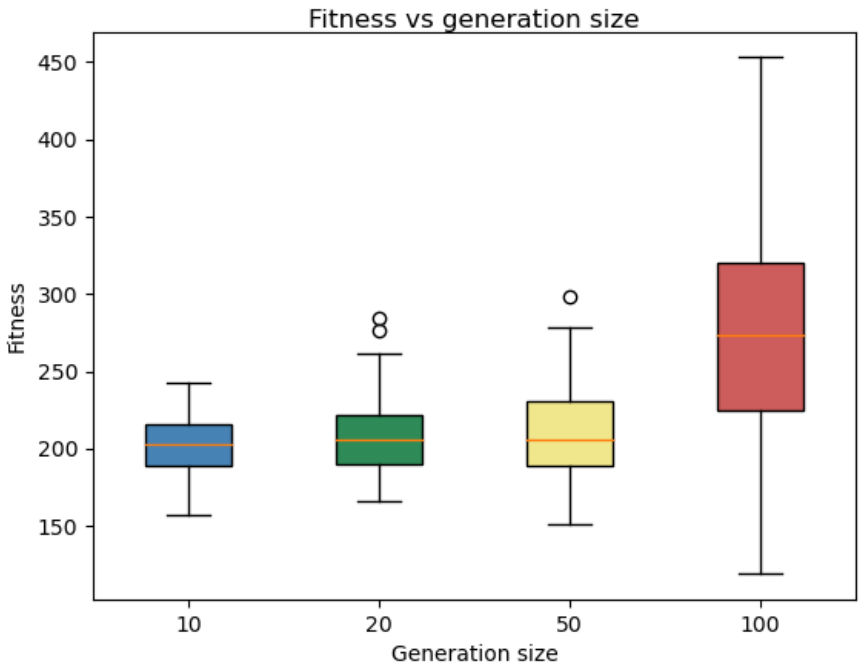


Figure 15: 50 runs with 4 different solutions with increasing population size.

6.6. Comparison CA vs. NN

There is a huge time difference between the CA implementation and the NN implementation. An evolution of a NN takes some minutes at max, but the CA evolution can take hours with a big search space and strict termination criteria. I believe it is many of the same reasons it was easier to implement the NN. Simple matrix structure, direct encoding of observations and libraries with efficient calculations. It is, of course, a substantial possibility that the code is inefficient in some ways as well.

The fitness seem to be quite similar for both the systems. I think this might related to the heuristic nature of the cartpole environment. I have not

been able to find a solution that scores an average fitness of above 250 over 100 runs. This still means the solution manages to balance the pole perfectly 50% of the time (or half perfect every time, you choose).

It is in the end clear that both these dynamic systems can be used to control the cartpole. Initial tests with my own solutions to this shows that it is not a trivial task to do this. I could easily have spend more time trying to create and tune my own solution than doing this project. This illustrates the power of EA and heuristic optimization, and I'm left impressed.

Appendix A.

`https://github.com/Overskott/OpenAIgym-project`

References

[1] Cart pole, `https://www.gymlibrary.dev/environments/classic_control/cart_pole/`, 2022. [Online; accessed 27-October-2022].

[2] H. Sayama, Introduction to the modeling and analysis of Complex Systems, 2015.

[3] Wikipedia, Cellular automaton — Wikipedia, the free encyclopedia, `http://en.wikipedia.org/w/index.php?title=Cellular%20automaton&oldid=1110162718`, 2022. [Online; accessed 27-October-2022].

[4] Wikipedia, Neural network — Wikipedia, the free encyclopedia, `http://en.wikipedia.org/w/index.php?title=Neural%20network&oldid=1117595892`, 2022. [Online; accessed 28-October-2022].

[5] Wikipedia, Evolutionary algorithm — Wikipedia, the free encyclopedia, http://en.wikipedia.org/w/index.php?title=Evolutionary%20algorithm&oldid=1119116995, 2022. [Online; accessed 31-October-2022].

[6] J. H. Holland, Genetic algorithms, Scholarpedia 7 (2012) 1482. doi:10.4249/scholarpedia.1482, revision #128222.