Reversing Game of Life

Year: 1
Group: 246
Specialization: ICA
Subject: Numerical Modelling in Data Analysis

Team:
- Marian Andrada
- Mardaloescu Ana
- Nechitoaea Daniel

This documentation contains information about Conway's Reverse Game of Life. It is structured in the following sections: (i) the problem definition which is an introduction in the topic, (ii) reasons for the problem to be solved, (iii) the solution we propose, (iv) the design of the learning task and the whole application, (v) an analysis on the obtained results and (vi) several conclusions of our trial in solving the Reverse Game of Life.

1. Problem definition

Game of Life is a zero-player game which takes place infinitely in a two-dimensional board. It is an example of cellular automaton and it is characterized by cells with two states: "alive" and "dead". This game is defined by 4 rules:

- Birth: a cell that is dead at time t will be alive at time t+1 if exactly 3 of its eight neighbors were alive at time t;
- Death: a cell can die by:
  - Overcrowding: if a cell is alive at time t+ 1 and 4 or more of its neighbors are also alive at time t, the cell will be dead at time t+1
  - Exposure: If a live cell at time t has only 1 live neighbor or no live neighbors, it will be dead at time t+1
- Survival: a cell survives from time t to time t+1 if and only if 2 or 3 of its neighbors are alive at time t.

There are several types of "life-forms":
- still life: block, beehive, boat, ship, loaf;
- periodic life-forms/oscillators: blinker, toad, glider, spaceship, light weight space ship, medium weight space ship, heavy weight space ship.

This game is very useful for modelling complicated, non-linear systems in physics, chemistry, biology, meteorology, cosmology, computational science, engineering and many other fields. Studying the patterns of Life can result in discoveries in other areas of math and science.

- The behavior of cells or animals can be better understood using simple rules. Behavior that seems intelligent, such as we see in ant colonies might just be simple rules that we don't understand yet.
- Traffic problems might be solved by analyzing them with the mathematical tools learned from these types of simulations.
- Computer viruses are also examples of cellular automata. Finding the cure for computer viruses could be hidden in the patterns of this simple game.
- Human diseases might be cured if we could better understand why cells live and die.
- Exploring the galaxies would be easier if machines could be invented that could build themselves. Imagine sending a probe to Mars that could build a copy of itself. Although this is theoretically possible, it has not been invented yet.

With all this raised interest in so many fields, a more interesting thought appeared: reversing the Game of Life. Using machine learning in order to reverse time in this famous game became a subject of interest for many people, but also a challenge.

2. Reasons for the problem to be solved

We see this experiment as a challenge to test our limits and compare our results with others. Just like Game of Life, the reversed one is a math modelling problem.

3. Solution

3.1. Other approaches

In his research paper Jonathan Goetz[1] describes a solution that is using a Multi-Layer Perceptron. The main reason for using this approach is the fact that for the forward application of the game each output is a function of its own and its neighbors input values, which is a principle that a MLP can use to determine approximate weights through training. Another reason is that due to the iterative application of these rules it is possible to break the problem down into multiple steps using a properly structured MLP to allow its internal hidden states to generate an approximation for each successive predecessor based on training data of the same number of back iterations.

The dataset that was used is provided by Kaggle as a part of their Reverse Game of Life Contest. Further a distinct set of data was produced with the same approach to generate the final error values for comparison.

Two implementations were tested, one using a recursive application of a trained neural network for each iteration to step back in time and another using a single application and training a neural network to associate the final outputs with the input. After training and validation on a divided data set of 500 samples expanded into 1500 step samples, both runs of recursive application with different input scaling showed maximum improvements over halfway approximation (assigning the midpoint of the range as a guess) of no more than 5.10% in terms of the sum squared errors for each data point, with no optimal number of hidden nodes across all layer counts or input

ranges, though there were generally local minimums around 5-10 nodes, 25 nodes, and 40 nodes per hidden layer. Using the second approach of applying a single approximator to multiple stages at once proved to be completely ineffective at an input set of 500 samples with the final trained approximator always predicting an output around1 regardless of the input with both input formats across multiple training and testing sets.

3.2. Our solution -> Genetic Algorithm

In computer science and operations research, a **genetic algorithm** (**GA**) is a metaheuristic inspired by the process of natural selection that belongs to the larger class of evolutionary algorithms (EA). Genetic algorithms are commonly used to generate high-quality solutions to optimization and search problems by relying on biologically inspired operators such as mutation, crossover and selection. Genetic Algorithms are method for solving both constrained and unconstrained optimization problems based on a natural selection process that mimics biological evolution

In a genetic algorithm, a **population** of candidate solutions to an optimization problem is evolved toward better solutions.

The optimization is done by the fitness functions, which chooses the best 80% of all the candidates, delegating them based on Number_Of_Matching_Cells/ Total_Number_Of_Cells.

The crucial parameter for this optimization is "delta" which considers each candidate after "delta" cycles of running the game and afterward doing the fitness selection. This way, only the candidates with the "best future(evolution)" are chosen.

The above mentions do not take into consideration the "mutation" part of the algorithm, which is done afterwards based also on a random chance and a binomial distribution.

As of **parametrization**, we have the crucial "delta", size of candidate population, total number of cycles the solver runs and the percentages on which the mutation and crossover are being done (the GA part of the algorithm).

A typical genetic algorithm requires:

a. a genetic representation of the solution domain
b. a fitness function to evaluate the solution domain

4. Design documentation

4.1. Dataset

The input for our solver is a game of life starting board (we made a default config but it also has customizable features as well) which we "run" for a "*warm_up_steps*" number of times resulting in the "*target_board*". The **target_board** is the input itself for the problem-solving.

The input can be given as a set of points (Y,X), or it can be edited inside the Fitness ->generate_default_problem method as a matrix dataset.

One important thing to mention is that the board was limited to a 20x20 matrix, so that the computation is relatively limited, the focus of this project is on the method and not the scalability.

4.2. Design of the learning task

Steps

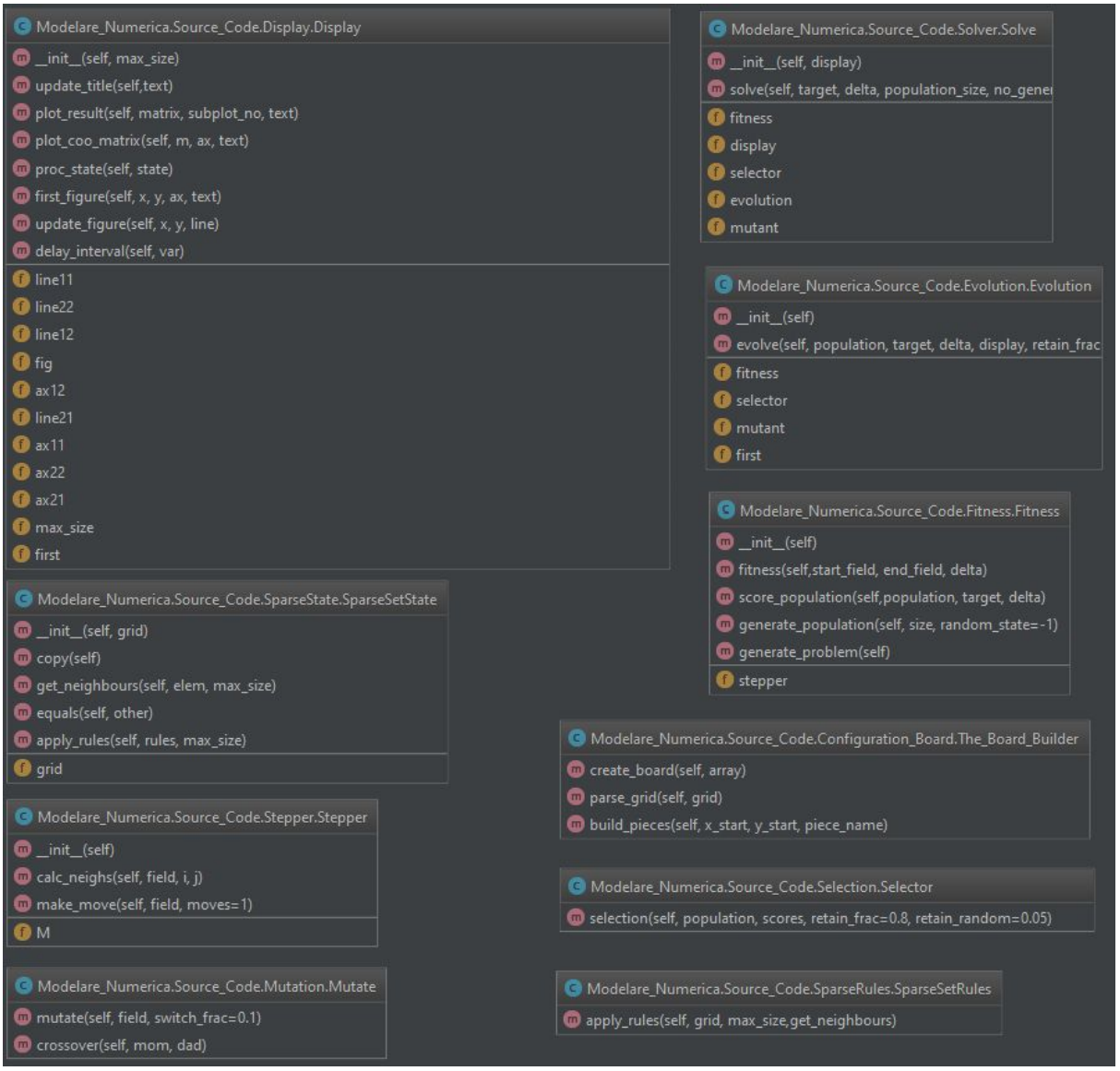
1. Initialize population
   Each individual is a 20x20 board that represents candidate start configuration. Each cell has equal chance of being "alive", and we're advancing the whole board by five iterations (known in challenge documentation as "warming steps") to match competition's distribution. The end of the "warm up" is the target to be solved. Afterwards we create a population of random samples that are candidates for the solution.
2. Define - Fitness Function
   Next we define fitness function to evaluate individual solution and population scoring function to evaluate population as a whole. The fitness functions evaluate each candidate by applying the game rules for "delta" which afterwards is compared with the target by the following:
   a. A truth table is computed by comparing each cell of the candidate with the target and the total number of True Positives is added up
   b. The resulting sum is divided by the total number of cell, which in our case is 400, the result being a matching factor (0,1].
3. Selection
   Selection operator is responsible for picking the best individuals to proceed into the next generation. In order to resemble natural selection more closely, we still allow for a small chance for sub-optimal individuals to live on and mate.
4. Mutation
   Mutation operator will introduce random variability into our population. It does this by randomly selecting cells and switching their state. Based on a binomial distribution with a 0.1 probability we create a matrix which will decide the cell on which XOR operation will be performed. Combined with the selection operator, this will make our algorithm perform a hill-climbing optimization.
5. Crossover
   Crossover operator will recombine two genes, simulating mating of two individuals. The result is two children, each with half of the genes (cell states) from father and other half from mother.

6. Solve

   Now that we have everything we need to perform an evolution step, let's collect it into a single function. It will take a population, perform a selection, mutation, and crossover, and return the next population.

   As an example, we have chosen a random configuration and played it for 5 generations. The population chosen was of 200 candidates, each being evaluated based on a "delta" equal to 2, so 2 steps forwards. The solver played for 300 cycles, outputting the best candidate with a performance of 91%.

Design of the application



5. Analysis of the results

- Metric used for evaluation is Mean Absolute Error: for each board, the number of cell states correctly guessed is divided by the total number of cells.
- Depending on the configuration, a bigger population size is always preferred, to have more chances of correctly guessing (Who knows, maybe a Gen 1 random perfect match!)
- The second important factor is the number of cycles the game runs. Depending on this, and the mutations/genetic factors, the longer the game runs, more chances are that a better configuration will appear.
- The most dependent factor and the one on which the performance is evaluated is "delta" which is how much "in the future" the candidate is evaluated. Despite a larger value for delta performs better, it results in a huge computational downgrade.
- The last factor is the probabilities based on which the best X% from the initial population is chosen, Y% of the remaining population is "mutated" and the remaining gap between initial population and best population is filled by randomly chosen "parents" and performing crossover.

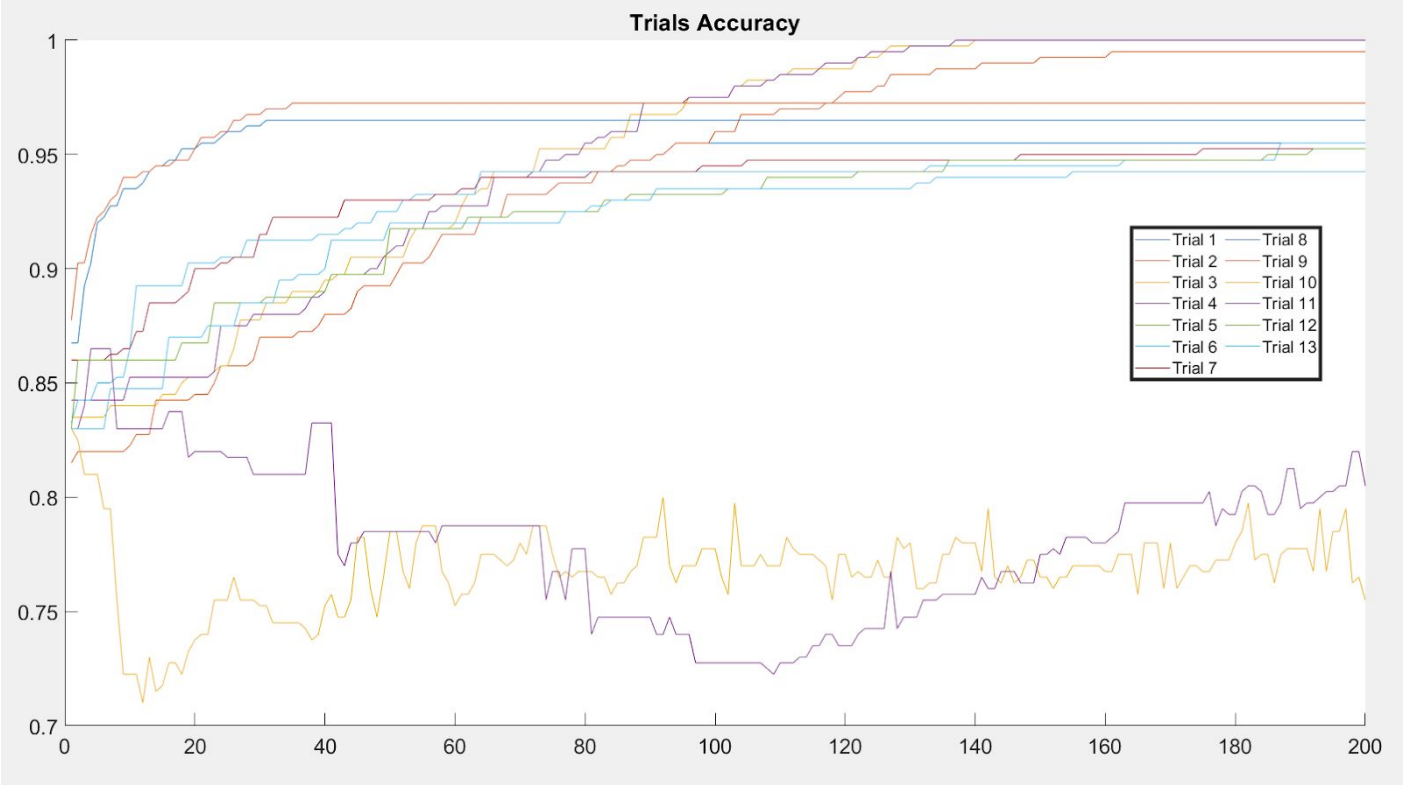


Fig.1: Trials Accuracy for each of the trials

Analysis on parameters:

# trial	Population size	Number of generations	Delta	Retain fraction	Retain random	Mutate chance	Accuracy
1	200	100	0	0.8	0.05	0.05	0.96
2	200	200	0	0.8	0.05	0.05	0.995
3	500	200	0	0.8	0.05	0.05	1
4	1000	200	0	0.8	0.05	0.05	1
5	200	200	1	0.8	0.05	0.05	0.955
6	200	500	1	0.8	0.05	0.05	0.965
7	500	200	1	0.8	0.05	0.05	0.9525
8	200	200	1	0.2	0.05	0.05	0.965
9	500	200	1	0.2	0.05	0.05	0.9725
10	200	200	1	0.8	0.05	0.5	0.83
11	200	200	1	0.8	0.05	0.2	0.865
12	200	200	1	0.8	0.2	0.05	0.9525
13	200	200	1	0.8	0.05	0.1	0.9425

Given the default values for the parameters at the first trial resulted in an accuracy of 0.96 (trial [1]).
After doubling the number of generations, the accuracy increased with 0.395 points, but did not reach the maximum accuracy and probably got stuck in a local maxima, unable to overcome it.
Increasing the population size consisted in a consistent improvement, as the maximum accuracy was achieved only after 140 generations. Having a population size of 1000 increased the reach of the maximum accuracy, but only 4 epochs earlier than the previous trial, so the tradeoff is not good as it consumes more resources, time and memory. The population size is an important parameter. A bigger number of populations leads to bigger chances to a better population.
Changing the delta parameter in trial [5] brings a dramatic increase in the time for the training, from several seconds to several minutes or hours, depending on the rest of the parameters. This is because the delta parameter is the number of steps that have to be reverted, so the computational calculus is much more bigger. According to the next trial, [6], raising the number of generations does not significantly affect the accuracy and neither does the retain fraction parameter (trial [8]), although the accuracy raises with almost 0.1 after choosing a population size of 500 (trial [9]).

The mutate chance parameter represents the probability of random change and its aim is to help the algorithm escape from a local maxima or minima of a function. After several trials of varying the mutate chance, the conclusion is that although it is an important parameter, it should be just like in the real world genetics, a very small number. It should not be the main influence on the population, but only a shade. Even a mutated chance of 0.1 (trial [13]) which is double than the one in the trial [1] results in a worse accuracy.
The best accuracy so far, with delta = 1, is the one in the trial [9] and this is due to the numerous populations and, surprisingly, the retain fraction which is a percentage of how much of the old population is retained in the new one.

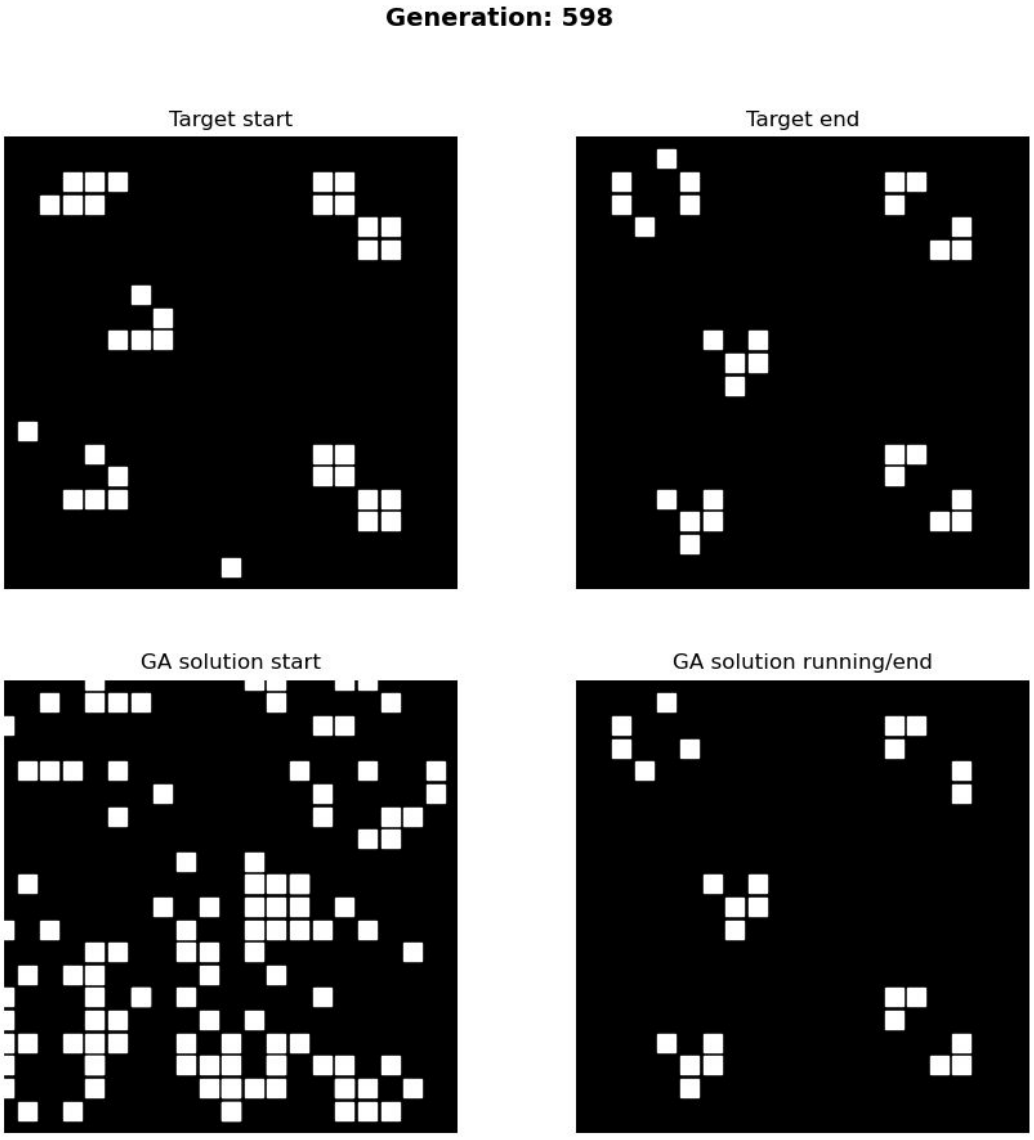


Fig.2: Example results (default board, delta = 0, population = 200, generations = 600)

6. Conclusion

Conway's Reverse Game of Life is a many-to-one problem, meaning that many starting states can lead to the same stopping state. After running the algorithm for a population of 200 candidates, the best candidate had a fitness score of 0.91

Multiple algorithms are used for solving this problem such as: Random Forest, CNN, SVM. Genetic Algorithms are currently on the second position in the Kaggle contest, which shows us that further improvements of the algorithm is possible. Solving The Game of Life has many applications, some of which reflect also on the reverse of it.