Machine Learning Final Semester Project by:

Yaman Salman & Laith Asabeh

Evolutionary Algorithms –

Tic Tac Toe using Q-Learning Algorithm

Introduction

Evolutionary Algorithms are algorithms that evolve by themselves over generations (over time). They are a type of algorithms that go through a huge number of simulations (generations or testing epochs) and improve upon mistakes or rewards by following a certain set of rules. These algorithms basically "train" themselves into becoming the best version of themselves. One of these interesting algorithms is the popular Q-Learning Algorithm.

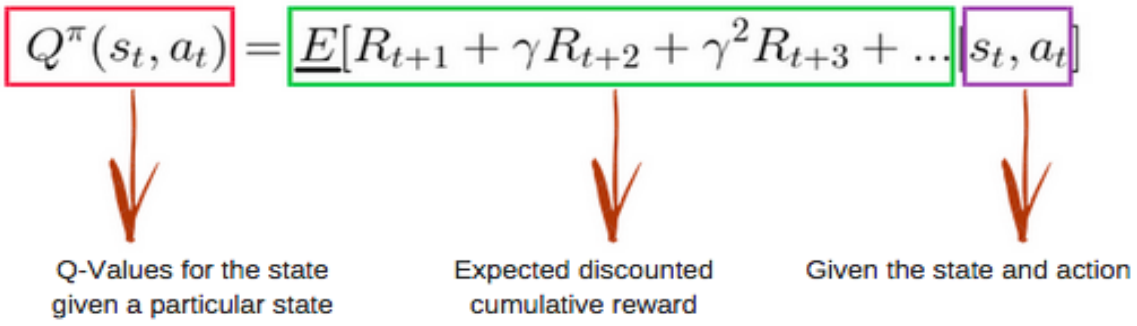
The Q-Learning Algorithm

The Q-Learning Algorithm is a model-free reinforcement learning algorithm that learns the value of an action in a particular state. It does not require a model of the environment to be able to adapt and learn. (Hence, model-free)

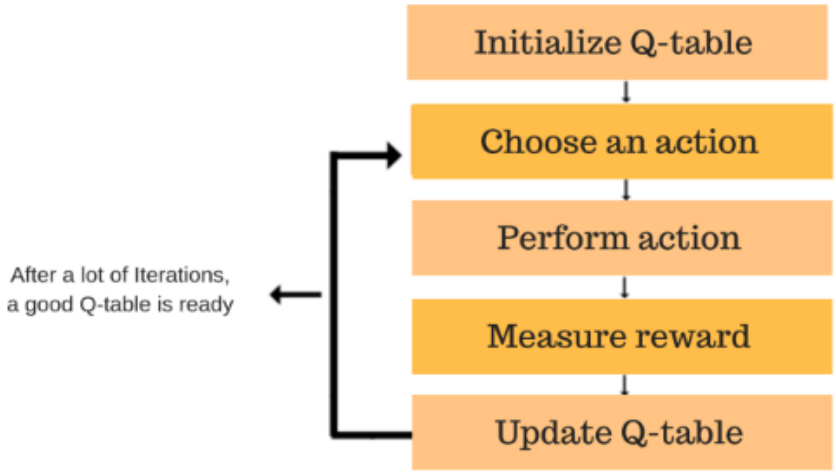
Concept & Principle

The Q-function is a Reinforcement Learning Algorithm. Reinforcement learning involves an agent, a set of states **(S)**, and a set A of actions per state. By performing an action in (**A)**, the agent successfully updates its current state to another. Depending on the state the agent is in, it either gets rewarded or punished. This is done by keeping track of a numerical score, between **0** and **1**, called The Epsilon Value. The goal of the agent is to maximize its total reward. It does this by adding the maximum reward attainable from future states to the reward for achieving its current state, effectively influencing the current action by the potential future reward. This potential reward is a weighted sum of expected values of the rewards of all future steps starting from the current state.

Here we can see how the Q-function uses the Bellman equation and takes two inputs: state **(S)** and action **(A)**.



Using the above function alongside a Q-Table, we get the values of Q for the cells in the Q-Table. When we start, all the values in the Q-table are zeros. There is an iterative process of updating the values. As we start to explore the environment, the Q-function gives us better and better approximations by continuously updating the Q-values in the table. After every move, the above equation is re-evaluated and the Q-Table is updated based on the newest reward.



Using the previous steps to perfect a q-table, the agent is assured to maximize its rewards and minimize its punishments. A perfect q-table can be achieved if enough iterations have been run, and the q-table constantly updated for the best values.

Approach

To demonstrate how the Q-Learning algorithm works, a simulation easy and simple enough was needed. The first idea that came to mind was to code a simple Tic-Tac-Toe game where the human plays against a self-learning A.I agent. Tic-Tac-Toe was chosen, because it follows the three criteria that were discussed in the previous proposal.

Condition 1: The neural network under study has to be in an environment where it is ever evolving.

Condition 2: The neural network must follow simple but advanced rules and conditions.

Condition 3: The neural network must be in an environment where it has to adapt to unexpected inputs.

Tic-Tac-Toe as simple as it is, it satisfies all three conditions. This will further help in the study of how the agent in the Q-Learning Algorithm works. The agent uses the q-learning algorithm, alongside the rules of the game, to learn through an iterative process how to play tic-tac-toe. The agent does this by playing around 20,000 games against itself, increasing and decreasing its Epsilon Value (the reward value), until it has reached a perfect state.

Tic-Tac-Toe is a game that a total of 3*3*3*3*3*3*3*3*3 = 3^9 = 19,683 different ways the 3x3 grid can be filled in. A little bit less than 20,000 simulations, which means the agent will reach a state in which it has perfected itself. It can never lose.

Code Implementation

The code was really simple to implement:

- Using python, a simple tic-tac-toe game was coded.
- The q-learning algorithm was implemented in a separate header file.
- The agent which uses the q-learning header file was implemented, given the rules of the game, and given the q-learning algorithm to predict the next moves for it.
- Everything was compiled into a separate main folder.

Results:

Some interesting results have shown after tweaking the parameters of the algorithm in the code. Some of them, were expected, while some others were completely astounding.

- Increasing the reward/punishment value, meant that the epsilon value permutates a lot less. If the permutation value is for example 0.5, and the maximum reward value for epsilon is 1 (which it has to be), then the simulation might run a maximum of two times before it hits 1, or 0. Resulting in an agent that has learned basically nothing.
- Increasing the number of simulations gives the agent more time to calculate the best possible predicted moves.
- Decreasing the number of simulations gives the agent less time to calculate the best possible predicted moves.
- Printing out all the games the agent plays against itself looks fascinating as we can see all the different combinations, iterations, and all levels of evolution of the agent.

Conclusion

Evolutionary Algorithms are one of the most interesting algorithms in the study of machine learning. The study of Evolutionary Algorithms is a field that contains a vast variety of different concepts. One of those interesting algorithms is the Reinforcement type algorithm: Q-Learning Algorithm. It is an algorithm that lets an agent use the Bellman equation. An equation which takes a certain set of parameters to evaluate the condition of the agent. The condition of the agent is evaluated based on a numerical reward value which can increase or decrease. The equation is an iterative equation that has to be repeated multiple times for the agent to perfect its way of evolving and adapting to its surroundings. (Due to a lack of an environment model)

The Q-Learning algorithm is such an interesting algorithm that can be implemented in so many different ways. It is also interesting how the Bellman equation mutates during any given set of simulations or generations. The results of tweaking certain parameters can be very interesting and sometimes unexpected.

Sources:

1. https://www.freecodecamp.org/news/an-introduction-to-q-learning-reinforcement-learning-14ac0b4493cc/
2. https://en.wikipedia.org/wiki/Bellman_equation#Derivation
3. https://en.wikipedia.org/wiki/Q-learning#Algorithm
4. https://www.baeldung.com/cs/epsilon-greedy-q-learning#:~:text=The%20epsilon-greedy%20approach%20selects,what%20we%20have%20already%20learned.
5. https://www.crows.org/blogpost/1685693/357431/The-Mathematics-of-Tic-Tac-Toe#:~:text=In%20actuality%2C%20tic-tac-,grid%20can%20be%20filled%20in.
6. https://towardsdatascience.com/simple-reinforcement-learning-q-learning-fcddc4b6fe56
7. https://www.geeksforgeeks.org/q-learning-in-python/
8. https://www.datacamp.com/tutorial/introduction-q-learning-beginner-tutorial