Reinforcement learning: Custom multi-agent snake environment

David Gichev 196034

Abstract

Reinforcement learning is a very active field in machine learning research. This paper demonstrates the methods that are most commonly used and the challenges that come with them. A special custom environment is specifically developed for this task, serving as an experimental playground. The game is a multiplayer version of the classical snake game, resembling a discrete version of sliter.io. First, a brief introduction to reinforcement learning is given. It is then followed by a summary of previous work done in similar environments. The game is then explained along with the numerous experiments done to better understand reinforcement learning in this context. Finally there is a results section coupled with data supporting the conclusions and reasoning.

1 Introduction

Reinforcement learning is a branch of machine learning that focuses on learning guided by reward maximization. It has been applied to a multitude of environments in diverse ways. It stems from the notion that biological organisms learn to interact with the world by trying different actions and seeing how the situation plays out. An agent (living creature) learns the world around it by developing intuition about the world's inner workings. It does so by observing its environment via sensory information and using that information to take actions and make decisions. The agent is intermittently rewarded or punished for the choice it makes. Reinforcement learning centers around the idea that these situations (states) each have an optimal strategy (action) in which the agent receives the maximum possible reward. The goal is to map each state to a value function which models the expected reward for the given state (s). We can then use this function to reason about the outcomes (s') that come with taking different actions (a) along the way.

$$Value(s) = max_a(Reward(s, a) + Value(s'))$$

The Bellman equations are a recurring part of reinforcement learning because of their ability to model the field's approach. The environment in which the agent operates can be set to mimic characteristics from the real world by employing nondeterminism or purposeful masking of its state. Meaning that the agent can only see parts of the world and that taking a certain action doesn't lead to a consistent outcome. This resembles the real world more closely and is often incorporated in environments used when researching this type of learning. This paper focuses on the snake game playground. This game dates back to the earliest ages of computing and is familiar to almost everyone. We consider a custom version of this game that more closely resembles sliter.io. This version is discretized in order to be an easier fit for existing models.

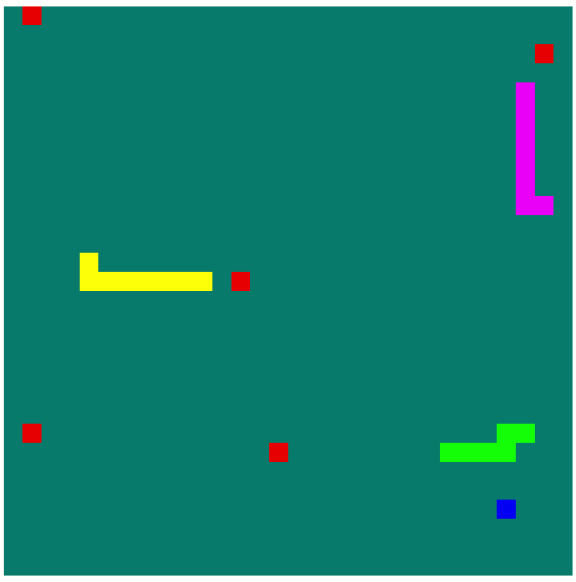

Figure 1: Overview of the proposed algorithm

The state space of the game itself is a grid of cells that can either be empty, a fruit, a snake body-part or a wall. Every snake has a direction it is heading in and it can decide to either continue moving forward, or turn to the right or left. There has been work done on the original snake game and on sliter.io as well. Some of the approaches are discussed in the following section. Then the main work behind this paper is laid out in the "Trials and tribulations" section which contains a compressed summary detailing the experiments that have been carried out and interpretations of the corresponding outcomes. In addition, the "Results" section visualizes and compares the learning process of a subset of the models. Finally, I discuss the results and methods along with potential future work in the "Conclusions" section.

2 Related work

Considering that the original snake game dates back to 1997, it's no surprise that a large body of work has been done in applying reinforcement learning to it. The state space is discrete and the game itself is relatively simple, hence its popularity in this domain. There are two common approaches to tackling this problem. The first one is serving the agent an incomplete state representation (data about the nearest apple, whether the snake is next to a wall) and letting it figure out the rest whilst guiding it with rewards promoting movement directed at food and negative rewards when hitting a wall or itself. A very informational instance of this type of approach can be seen in [1]. The author describes variations of the state representation and shows how tweaking some of the parameters inhibits learning. My method takes these results into consideration and plays with similar state representation and reward systems. The other approach lets the agent have a complete view of the environment [5] [6]. This can be done by sending the whole grid to the agent as a matrix, which is practically equivalent to the way modern Atari game research is done (the agent views the game the same way as we do - by observing the pixels on the screen). Letting the agent have access to the whole state has the most potential for a training process that results in a policy that is optimal. However this comes at the cost of training time and processing resources. The models used to train on this sort of data are almost always based on deep learning and on networks that have hundreds of neurons and millions of parameters. In addition, transfer learning is almost certainly used as an initial starting point for the network, as solving basic computer vision via reinforcement learning is far from feasible in this context. This second approach is very popular, if not universal, in the world of slither.io reinforcement learning research. The game screen is passed to the agent which maps it to the appropriate keyboard controls and plays the game just as a human would. I decided to stick with the first approach. My reasoning boils down to two things: performance and experimentation. I'm interested in reinforcement learning because of its ties to cognition and I find that playing with different state spaces and reward mechanisms serves to deepen my understanding of this field. The state of the art models are usually run on powerful machines with computational capability miles ahead of most personal computers. This work serves as a venture in the world of reinforcement learning.

3 Trials and tribulations

Most of the attempts first start with a single snake and then graduate to multiple snakes. All the snakes share the same model. Each snake receives its own state and uses the shared policy to decide its next action, independent of what state the other snakes have. Most of the experiments along with their environments, testing and training code are available online [3].

3.1 Experimenting with natural state representations

The first attempt at deriving a state representation was building a window of the game grid that is invariant to the direction of the snake. This would serve to shrink the number of possible states which would ease the learning curve. The state is taken by extracting the cells that surround the snake's head in the direction the snake is facing. The end result is a point of view of the snake itself - a natural representation analogous to the way vision works in biological snakes. Each of the cells is represented by an integer corresponding to the type of object currently residing in the cell: 0 stands for nothing, 1 for the snake's own body, 2 for food, 3 for walls, and 4 for body parts of other snakes. The first method I attempted was classical Q-learning. This method is characterized by a large table that stores the expected values for every state and every action. The

agent chooses which action to take by the payoff it expects according to the table. Exploration is a very important topic in reinforcement learning and it's almost always used in Q-learning. A constant (epsilon) is defined as a measure of randomness. When considering the next action to make, the agent chooses to take a random action part of the time (the probability of it choosing a random action is epsilon). This facilitates exploration into unfamiliar states and results in a richer model. I quickly realized that Q-learning is not applicable to this problem due to the sheer size of the state space. Just a 2x3 grid results in 5^6 possible states (around 15k states), while a 4x5 grid has around 10^{13} states. No matter how much time is spent on training or how large epsilon is, populating a state space of this scale is impossible.

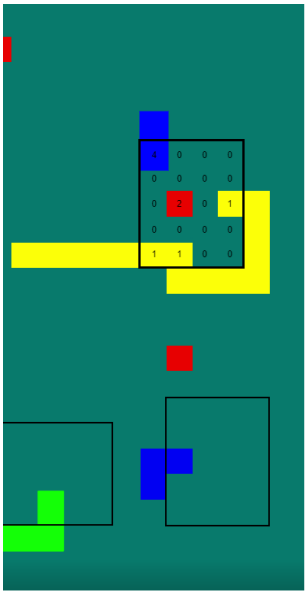

Figure 2: Proposed natural point-of-view state

The next method I turned to was deep Q-learning [4]. This operates in a similar notion to classical Q-learning with the crucial difference being that it doesn't store an exhaustive state mapping. Rather, it uses a deep neural network to predict the outcomes of taking each action in a given state. Those values then get used to pick the action with the most favorable prediction. I experimented with different network sizes and learning parameters. The discount rate was almost always set to 0.99 or 1. The training resulted in agents that were semi-successful. When faced with a near-death situation they avoided the obstacles at hand, but they didn't seem to care for fruit rewards. The main problem however, was that they had a tendency to start spinning in a circle and never stop. This was most pronounced when testing (epsilon is set to 0 then). This behavior turned out to be a symptom of a larger problem that plagues these types of state representations.

3.2 The problem with window based state representations

Q-learning and a lot of reinforcement learning methods in general make the assumption that a state has a corresponding optimal action. This in conjunction with a poor state representation can lead to an agent that makes confusing choices. Since a state can have only one optimal action, the agent will have to choose that action every single time. Take the empty state as an example (every cell around the snake's head is empty). The agent effectively has to pick an action for this state. Since going forward can eventually lead to obstacles, the agent chooses to change direction either to the left or right. This results in an agent that constantly turns clockwise or counterclockwise in order to avoid potential damage. Increasing the window size only works up to a point - the agent will almost certainly find itself being surrounded by empty cells most of the time. The conclusion is clear - in order to make better decisions the agent needs to know more about its surroundings.

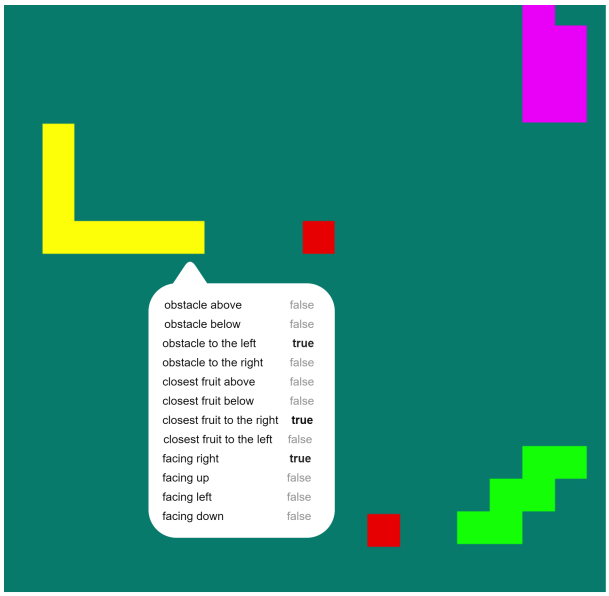

Figure 3: Proximity based state representation

3.3 Proximity based representations

This class of representations is often used when researching reinforcement learning on the classical snake game. The agent receives information about whether or not the cells around it are dangerous

and its relation to the nearest fruit. More specifically, the state space can be represented by 12 values wherein the first 4 correspond to an obstacle being present in the above/below/right/left cell (be it a wall or snake body part), the next 4 describe the position of the closest apple (whether it's to the left/right/above/below the snake's head). And the last 4 are a vector mapping of the direction the snake is facing ($< 0, 0, 0, 1 >$ means it's facing down, $< right, up, left, down >$).

This sort of simplified state representation leads to faster learning. The agent was trained for around 500 episodes (DQN, training occurred every 3 episodes) using memory recall. It successfully managed to operate correctly and developed maneuvers similar to ones displayed by human players. It learned to go towards the fruit (the reward system - explained after this) and it didn't always take the shortest path since that sometimes resulted in it bumping itself. A different version of the state was also tested - the second quadruple was modified to check whether there was a snake body part 3 or fewer blocks left/right/below/above the snake's head. Yet another state variant was tested in which the first three quadruples remained unchanged and a fourth one was added with the aforementioned proximity information. This representation yielded the same results albeit requiring a longer training time.

3.4 Reward systems

Building reward configurations is a very delicate process that sometimes gives output to counter-intuitive results. The initial reward system consisted of a few simple rules: eating a fruit resulted in a reward of 1, dying resulted in -10, and everything else had a default reward of -0.1. This was enough for the natural state representation, but further modifications were needed for the proximity based version. An additional reward for movement towards the nearest fruit was included and a corresponding negative reward for distancing. This helped with training times and resulted in better guided learning. Variations of this reward system are explained in [1]. For example, setting a negative reward for living and a small reward for eating can result in the snake finding the nearest wall to run into, to minimize the negative reward.

3.5 Memory sampling techniques

While training I experimented with the memory sampling procedure. Memory replay is commonplace in deep reinforcement learning as it helps with bias and quick learning. The most common way of implementing sampling is by storing a memory buffer of past experience (in the form of state/action/reward data) and later uniformly sampling it. This is very effective for reducing bias as the most frequent memories have the largest effect on the network weights. I tried defining a different way of sampling in order to steer the agent decision process in a more effective manner. The motivation for doing so was the relative rarity of death events. I increased the odds of a death event being remembered as opposed to a normal event. This worked in the short term, but later resulted in agents that are too wary of their surroundings which gave way to looping tendencies. Curiously, the loops were far more complicated when compared to the single-action loops present in the natural state representation case. They could perform up to 30 actions, tracing intricate patterns, and return to the same starting state.

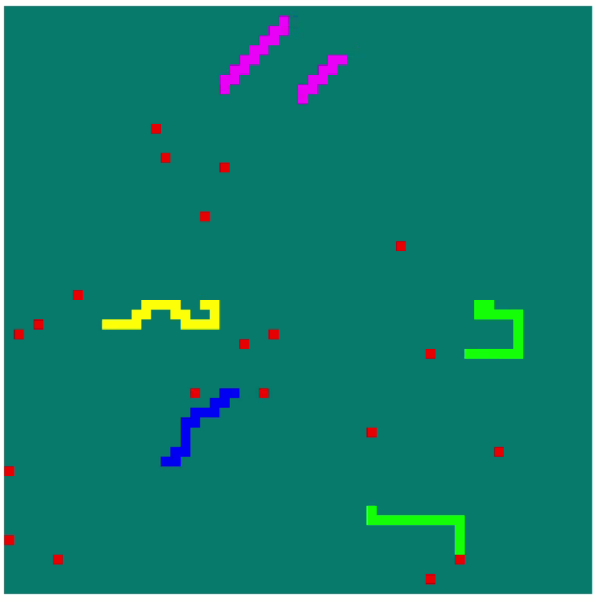

Figure 4: Large grid space

3.6 Multiagent behavior and larger grid sizes

Most of the initial training was done with only a single action in the system. If the agent showed promising results, I tried the same model with multiple agents and carried on with the learning by using the same weights and parameters (some sort of manual transfer learning). For example, I trained a single-agent with the original proximity based representation and later added a quadruple containing information about the other agents. The learning was nearly seamless and intuitively it makes sense why (only some of the network weights had to be modified). I increased the grid size in a similar manner. I trained the agent/s on a smaller grid and then continued with the training on a larger grid. Nothing really changed in the state representations and perceptions that the agents had, which is why it's no surprise that they were well equipped to deal with a sparser grid. However, after sufficient training a pattern started to emerge - the snakes began to take the shortest path to the fruit. This was not the case in smaller grids since the chance for collision outweighed any time concerns. But when faced with larger distances the snakes realized that getting to the fruit quicker led to a bigger reward.

4 Results

The results were gathered with a custom training component that monitored agent performance over the course of training. The data points are calculated by running a benchmark every 20 episodes of training. The benchmark is a test version of the environment where the agent doesn't make random decisions (epsilon is set to 0). It consists of running 10 test episodes and taking note of the rewards they come up with. These 10 reward values are then aggregated by taking the reward with the highest value and later mapping it to a data point. This minimizes the large reward fluctuations that happen between episodes and allows for easier visualization of the model's progress.

The model in question is trained on an average sized grid (30x30) with 3 agents. The model's performance hits a noticeable plateau around the 1000th episode. This shows that the agent's capabilities given the state it uses are bounded and further training would probably be to no avail.

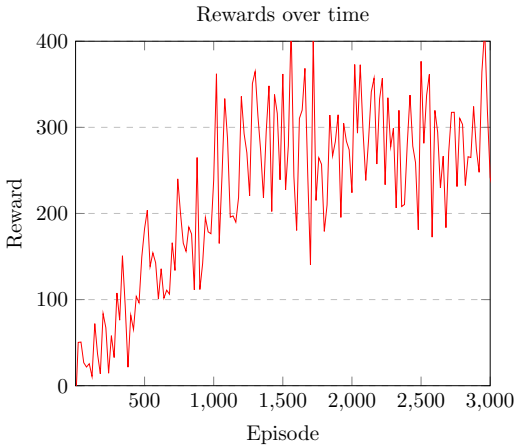

Figure 5: Agent performance over large period of training

The following experiment shines light on the effect of the reward system. In the "not rewarded" case, the agent fails to learn that eating fruit leads to a positive outcome. The agent has access to information that can guide it toward the apple but that same information doesn't get utilized and the agent resorts to looping and moving along walls.

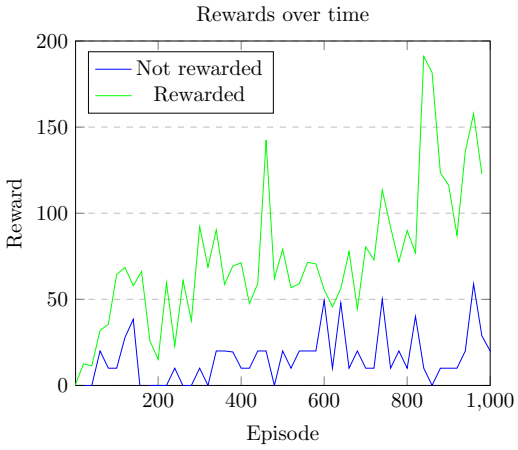

Figure 6: The effect of giving a reward for moving towards the apple

I wanted to explore how transfer learning could be incorporated into the equation, so I tried training a model on a large grid with multiple agents from scratch. Then, the same experiment was repeated, only this time the agent had a head-start in form of another agent's experience.

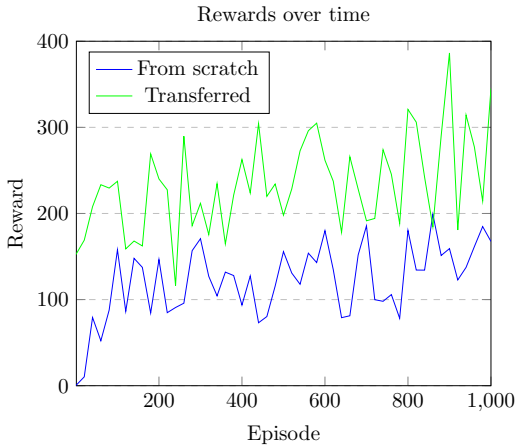

Figure 7: The benefit of transfer learning

The experience in question was in the form of a trained network from an agent raised in a smaller single-agent environment. This is just one example of multiple configurations that allow for model reuse.

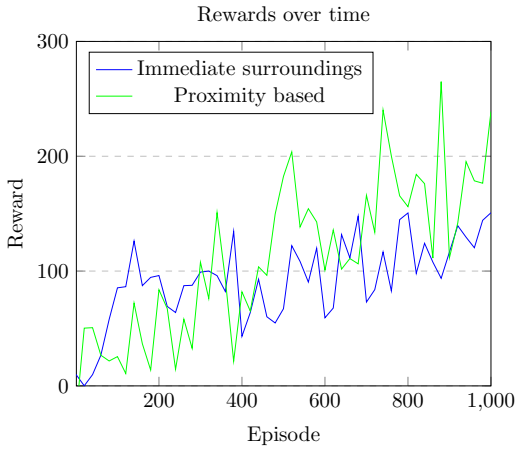

Figure 8: Experimenting with obstacle distance threshold

Registering nearby snake bodies that are further away (2-3 cells) results in a slight improvement on agent performance.

5 Conclusion

This paper proves that reinforcement learning is applicable to the discussed environment in many different ways. It shows that the process of system tuning is very delicate and consists of an infinite array of dials and switches. Multiple state and reward formulations were suggested and pitted against each other. Apart from feeling like an agent myself, I've gained a better understanding of the discipline and noted multiple interesting questions and ideas for future work. Striking a balance between the rewards and deriving a suitable state representation is not straightforward and this paper serves as a collection of experiments that help to better understand systems like these. Transfer learning is a very relevant topic in reinforcement learning research and having a simplified system helps with building intuition around the way knowledge can be transferred and shaped. Future work can be done in finding more creative ways to represent the state and in testing new methods. Further experimentation with DDQNs, Dueling DQNs and DDPGs can be performed and compared. Finally, an altogether different approach that focuses on competition can be looked into.

References

[1] Hennie de Harder. Snake played by a deep reinforcement learning agent. https://towardsdatascience.com/snake-played-by-a-deep-reinforcement-learning-agent-53f2c4331d36, 2020.

[2] Arthur Dujardin. Ai plays snake game. https://github.com/arthurdjn/snake-reinforcement-learning, 2020.

[3] David Gichev. (source code) reinforcement learning for custom snake environment. https://github.com/davidgicev/ABS, 2022.

[4] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. 2013.

[5] Alessandro Sebastianelli, Massimo Tipaldi, Silvia Liberata Ullo, and Luigi Glielmo. A deep q-learning based approach applied to the snake game. jun 2021.

[6] Zhepei Wei, Di Wang, Ming Zhang, Ah-Hwee Tan, Chunyan Miao, and You Zhou. Autonomous agents in snake game via deep reinforcement learning. jul 2018.