Q-learning in the snake game

Adam Zaiter

***Abstract*—The goal of this paper is to train an AI agent to solve the classical snake game. The Q-learning algorithm was used to train that agent, however problems with the state space arised. A possible solution by using a fixed size state space was presented and tested.**

Experiments showed that training an agent using this approach worked better on smaller environments. This agent was also able to play in environments of much bigger sizes than those it was trained on, but the agent was not able to play perfect games in those environments.

I. Introduction

This paper will attempt to implement the Q-learning algorithm for a basic game of snake. A hypothesis about the expected learning outcome will be established and reviewed after implementing the Q-learning algorithm. The implementation will be done in Python and a custom snake game environment will be created. This environment will allow for training of the agent with different hyperparameters as well as visualizing a game played by the trained agent.

Basic knowledge of reinforcement learning is assumed, as this paper is implementing concepts from the Szepesvari paper [2].

II. Q-learning

Q-learning is the algorithm that will be explored in this paper. It will be used to train an agent that will be able to maximize the reward for it's environment.

Q-learning finds the optimal policy by learning the best Q values for each state-action pair.

The Q function accepts the current state and an action. It then returns the expected reward for taking said action for the given state.

The agent then plays episodes. For each step in the episode the Q values in the Q table are updated using the Bellman equation. This process goes on until the q function converges to the optimal function q*.

III. Reinforcement learning problems

With the algorithm to train the agent established, let us discuss the environment it will be placed in. Reinforcement learning is suitable for solving games, thus many games were considered as potential candidates.

One of the more interesting ones is the classical snake game [1]. This game is played on a grid, where a food item is randomly spawned on one of the tiles along with a snake, controlled by a player. The player's objective is to navigate the snake to collect as much food as possible. After collecting food, another is randomly spawned in one of the available tiles. This offers an interesting environment for the agent to be trained in, as it's random from turn to turn and game to game. Thus, the agent has to learn to generalize for the environment as a whole.

IV. Hypothesis

The goal is to train an agent that will play the snake game with any chosen grid size as best as possible using just Q-learning. Given the attributes of Q-learning, it should be possible to train an agent that plays the game of snake at a very good level. It is not expected to be perfect, but it should play as close to it as possible.

V. Environment

The environment mimics functionality provided by environments in OpenAI gyms [3]. This made the process of applying the Q-learning algorithm quite simple, as the OpenAI environments are designed with reinforcement learning in mind.

A. Snake game implementation

The snake game is programmed in python. It supports square grids of any size that is $>= 3$. The snake starts in the middle of the grid with a head of 1 square in size and attached body with 2 squares in size. There also is food spawned in a random square that is not occupied by the snake. The objective for the snake is to eat as much food as possible. If the food is eaten, the snake gains + 10 score and grows an additional body part of 1 square in size. If the snake collides with the borders of the grid or the head collides with it's own body, the game ends.

B. Action space

The environment provides three actions for the agent to take:

1) Turn the snake head left.
2) Turn the snake head right.
3) Continue heading straight.

These values are encoded as integers 0, 1, 2 and aliased as constants LEFT, RIGHT and STRAIGHT in the code.

C. State space

For this project, the computing resources are somewhat limited and as such, these limited resources have to be taken into consideration when designing the state and action spaces. The state space for a snake game with a variable size would be too big, if it were taken as is. That is, because the snake and it's body adds many additional states, as the snake can grow it's body to be the same size as all of the squares in the grid. Also, the head is distinct from the other body parts and would have to be accounted for in all of the states.

For these reasons the current environment does not provide the full state of the board. Thus, the environment is restricted to this basic information:

- Information about danger. It indicates if after taking an action from the action space in the current state the game will be over.
- Information about food position with respect to the snake's head. The values are left, right, up and down.
- Information about the snake's current direction. There are four values: left, right, up and down.

After gathering this information, it is encoded in a 1D array. For example, the array for food position is an array of four values. Each position of the array corresponds to a value (left, right, up, down). Thus if the food is left, the array on position left will have the value of 1 (true). Otherwise, the value will be 0 (false). This applies for all positions. In the end, three individual arrays for danger, food and direction are created and are returned concatenated as the current state.

This approach will have worse results for small environments than a full encoding of the grid, as it provides much less information to the agent. However, one of the criteria in the hypothesis (IV) is for the agent to play on grids of variable sizes. While encoding the whole environment for a 100x100 grid is simply infeasible, the approach described above has constant space size for any given grid and thus can still be used.

D. Extending the state space

The most obvious way to extend the state is to add more information about danger. For the most basic case, danger is supplied as an array of boolean values. From the current state a lookahead is performed on every action from the actions space. If the action would result in a game over, danger is set to true for that action. This array can be extended to looking ahead more than one action in the future. This however grows the danger array exponentially, so the comparison in the coming experiments will be made only between one and two future actions lookahead (11 size state space and 20 size state space respectively).

VI. Training

The python program supports a training mode, where training arguments can be provided. Some of those are the environment size, total episodes to train for, decay rate, depth (size of state space), learning rate and more.

The best results were observed in agents trained on the 3x3 grid environment.

All agents referenced below have been trained for 500 000 episodes with an exponential decay rate with a value of 0.01. Minimum epsilon with value 0.01 was used to keep some exploration throughout the whole training process.

Figure 1 contains training statistics for the agent that was trained with a limited information about danger. This agent performs well, however it is unable to play perfect games very often at all.

Figure 2 show statistics for the agent that was trained on a state space with information about danger for two turns ahead. This agent performed considerably better than the previous agent and is able to play a perfect game quite often on a 3x3 grid and on average outperforming the aforementioned agent.

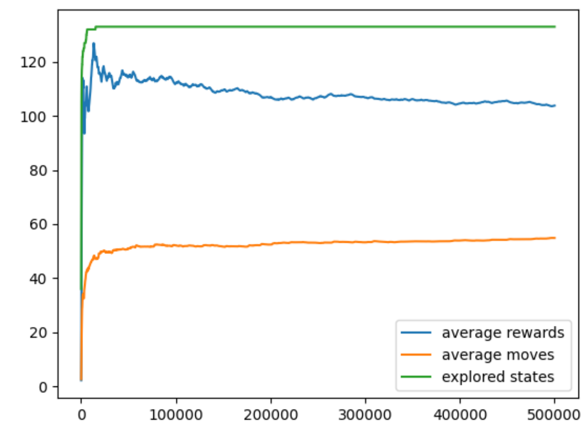

Fig. 1. 3x3 environment 11-size state space

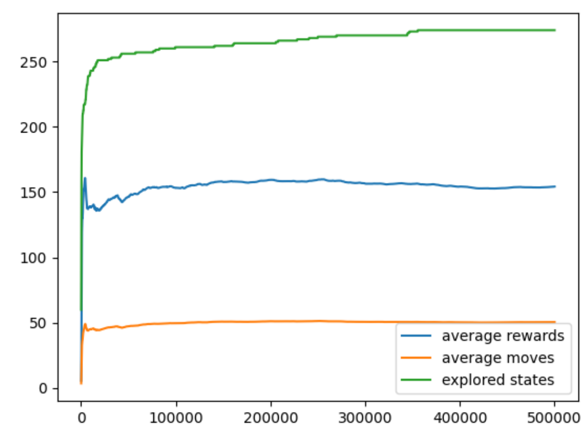

Fig. 2. 3x3 environment 20-size state space

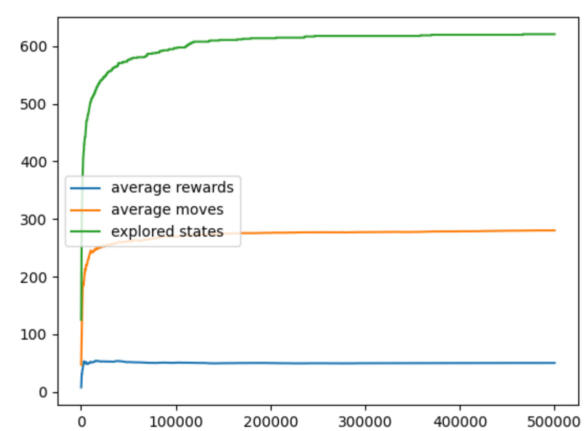

Fig. 3. 16x16 environment 20-size state space

Finally, figure 3 shows training of an agent on a 16x16 grid. Expectedly, the performance is much worse than the other presented agents.

Both of the mentioned agents that were trained on a 3x3 grid are capable of playing games in larger environments with reasonable performance. The agent trained on the larger state space continues to outperform the other agent even in the larger environments.

VII. Results

A. Learning

The agent learns much better in smaller environments as evident from the figures in section VI. One of the reason undoubtedly is the amount of moves the agent has to make. In smaller environments the number of moves made is much smaller, allowing the agent to explore states and learn much quicker. Another reason could be the amount of information the agent has about the environment. In smaller environments the state space looks much more similar to the real state of the game than in larger environments.

Even with limited information the snake learns the strategy of creating as much space as possible for it's own body. This results in the snake hugging the borders as much as possible, thus it takes longer paths to get to the reward (food) which might seem strange early in the game. However, when the snake's body is longer after collecting a few rewards, the benefits of this strategy are fully visible.

Another interesting fact that can be observed from the training plots is the dip in average rewards when new states are explored. This dip is more significant for the agent with limited state space.

B. Rewards

Since the environment is custom, rewards had to be assigned for performing certain actions. There are three types of reward

- Reward for eating food.
- Cost for achieving game over state.
- Survival reward for making a move that did not result in game over.

Survival reward did not seem to make any significant difference. Only when it was set too high the agent prefered to just move around safely without picking up food.

A lot of experiments were conducted to find the right combination of cost and reward. The one that seems to work most reasonably is to have the cost to be negative double of the amount of the reward.

C. Hyperparameters

The importance of hyperparameters was tested empirically through training the agent with various values. The hyperparameters available were:

- Learning rate α.
- Future reward discount γ.
- Exploration/exploitation ϵ.
- ϵ decay rate.

The values that gave the best results for α were around 0.75, for γ around 0.9. However, changing these values did not have as much effect as manipulating ϵ and it's decay rate.

First of all, exponential decay rate proved to be slightly better. Then, the seemingly best strategy was to explore quickly at the beginning and then let the agent learn the Q values. Still, a minimum ϵ value of 0.01 was kept to have at least a small element of exploration still present even later in the training.

VIII. Further work

A. Deep Q-learning

To solve some problems stemming from large state spaces as described earlier in the paper, deep Q-learning can be used. When value iteration becomes inefficent due to large state spaces, function approximation can be used to estimate the optimal Q function **??**. Deep neural networks are great candidates for this approximation.

B. Hamiltonian cycle

A hamiltonian cycle visits every vertex in a graph exactly once. The path starts and ends in the same vertex. If the grid of the environment would be of even width or height, a hamiltonian cycle would exist for that graph [4] and it could be used as the path for the snake. This way, it would be guaranteed to collect the food and not collide with it's own body or a border at any time.

IX. Conclusion

The Q-learning algorithm was implemented and used in a snake game environment. Issues of the environment's state space were discussed and a solution was proposed. It was then tested on multiple agents trained in different environments. It was found, that the agent learned better in smaller environments. Furthermore, the agent with extended state space outperformed the other agents. It was established via the experiments, that an agent that could play a perfect game on any grid size could not be created with the proposed methods. Thus, further work for improving performance in the snake game environment was discussed.

The code for the project can be found at https://github.com/adzai/Reinforcement-Learning.

References

[1] Snake (video game genre) https://en.wikipedia.org/wiki/Snake_(video_game_genre) [Accessed 18 January 2022].
[2] Csaba Szepesvari - Algorithms for Reinforcement Learning. [online] Available at: https://sites.ualberta.ca/~szepesva/papers/RLAlgsInMDPs.pdf [Accessed 15 January 2022].
[3] Open AI Gym. [online] Available at: https://gym.openai.com/ [Accessed 15 January 2022].
[4] Itai, Alon & Papadimitriou, Christos & Szwarcfiter, Jayme. (1982). Hamilton Paths in Grid Graphs. SIAM J. Comput.. 11. 676-686. 10.1137/0211056.