Snake with deep reinforcement learning

Jovan Prodanov
jp6957@student.uni-lj.si
FRI
University of Ljubljana
Večna pot 113
SI-1000 Ljubljana, Slovenia

Rok Nikolič
rn46656@student.uni-lj.si
FRI
University of Ljubljana
Večna pot 113
SI-1000 Ljubljana, Slovenia

Luka Šešet
ls90321@student.uni-lj.si
FMF
University of Ljubljana
Jadranska ulica 19
SI-1000 Ljubljana, Slovenia

ABSTRACT

The Snake game is a classic video game that poses a challenging problem for reinforcement learning algorithms. In this project, we developed a Snake game environment using Unity and implemented three reinforcement learning algorithms: Deep Q-Network (DQN), Double DQN, and Dueling DQN. Our goal was to train an agent to play Snake effectively and compare the performance of these algorithms. The Snake game was designed with four possible actions and a grid-based, Manhattan-style movement. We engineered the state space and reward functions to encourage the agent to navigate towards food while avoiding collisions. Our results demonstrate that Double DQN and Dueling Double DQN converge more quickly and achieve higher rewards compared to the standard DQN, highlighting the benefits of these advanced algorithms in reducing overestimation bias and improving learning efficiency.

KEY WORDS

Video game, Snake, DQN, DDQN, DDDQN, Hamiltonian cycle

1 INTRODUCTION

Snake is a well-known classic video game, originating in the 70s but gaining widespread popularity when it was pre-loaded on Nokia mobile phones in the 90s and 2000s, where we first came into contact with it. The objective of the game is to eat fruit, which makes the snake longer while avoiding its own tail and the walls. We decided to implement several *reinforcement learning* algorithms to teach an agent to play Snake as effectively as possible. This problem turned out to be quite challenging, as we will explore in this paper.

2 REINFORCEMENT LEARNING

Reinforcement learning is a type of machine learning where an agent learns to make decisions on its own by interacting with the environment. It requires minimal human intervention for learning as it aims to maximize cumulative reward over time.

The interaction with the environment is usually split into the *State space*, *Action space* and the *Rewards*.

State space represents the environment variables that the agent can detect. **Action space** comprises the actions the agent can perform. **Reward space** provides the positive or negative feedback the agent receives based on its actions.

The agent's goal is to learn a *policy*, which is a function that maps states to actions, maximizing the cumulative reward.

2.1 Deep Q-Network

A Deep Q-Network (DQN) [2] is an approach to Q-learning that utilizes a neural network to approximate the Q-value function, which estimates the maximum future rewards an agent can receive from each action in a given state. In traditional Q-learning, the Q-values are stored in a table, but this becomes infeasible for large state and action spaces.

It overcomes this limitation by using a neural network (policy network) to approximate the Q-value function, $Q(s, a)$, which predicts the best action a to take in a given state s. The Q-value function is updated using the Bellman equation:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left(r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right),$$

where r is the reward, γ is the discount factor, and s' is the next state. To improve stability, DQN uses experience replay, where transitions (s, a, r, s') are stored in a replay buffer and sampled randomly for training. Additionally, DQN employs a target network to provide stable target values for updating the Q-network.

2.2 Double Q-Learning

Double Q-Learning [3] is an extension of the Q-Learning algorithm designed to address the problem of overestimation bias in action-value estimates. This bias occurs because Q-Learning uses the same value estimates both to select and to evaluate an action. Double Q-Learning mitigates this by using two separate value functions, to decouple the selection and evaluation steps.

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left(r + \gamma Q_{\text{target}}(s', \arg\max_{a'} Q_{\text{policy}}(s',a')) - Q(s,a) \right),$$

where Q_{policy} is the learnt policy network and Q_{target} is the target network.

This decoupling of action selection and action evaluation helps to reduce the overestimation bias inherent in standard Q-Learning.

2.3 Dueling Q-Learning

Dueling Q-Learning [4] introduces a new neural network architecture for Q-value function approximation, separating the representation of state values and the advantages of each action. This architecture, known as the dueling network, is particularly useful in environments with many similar-valued actions, enhancing learning efficiency and policy robustness.

Dueling Q-Learning introduces a neural network architecture that separately estimates the state value function $V(s)$ and the advantage function $A(s, a)$. The Q-value function is then computed as:

$$Q(s,a) = V(s) + \left(A(s,a) - \frac{1}{|\mathcal{A}|} \sum_{a'} A(s,a') \right),$$

where $|\mathcal{A}|$ is the number of actions. This architecture helps in environments where it is useful to separately assess the quality of a state and the advantage of taking specific actions, thus improving learning efficiency and policy robustness.

The Dueling DQN architecture is implemented with two streams in the neural network: one for the state value function and one for the advantage function. The final Q-value is then computed by combining these two streams.

3 RELATED WORK

Many studies have successfully applied reinforcement learning techniques to classic video games. Mnih et al. (2013) demonstrated the capability of DQNs to play various Atari games [2], achieving human-level performance. Hasselt et al. (2016) introduced Double DQN to address the overestimation bias in Q-learning [3], showing improved performance in several benchmark tasks. Wang et al. (2016) proposed the Dueling DQN architecture [4], which further enhanced the performance of value-based reinforcement learning methods.

4 IMPLEMENTATION

In this project, we developed a Snake game environment using Unity and implemented three reinforcement learning algorithms: DQN, Double DQN, and Dueling DQN. The different algorithms were used to investigate varying levels of performance and learning efficiency [1]. We utilized the Unity ML-Agents toolkit to create a seamless connection between the game environment and our reinforcement learning models.

We developed a custom Snake game environment using Unity to facilitate the training and evaluation of reinforcement learning agents. The game consists of a grid where the snake moves in a Manhattan style, with four possible actions: up, down, left, and right. The snake grows in length each time it eats the food, and the game ends if the snake collides with itself or the walls.

To connect our Unity-based environment with the reinforcement learning algorithms, we used the Unity ML-Agents toolkit. This toolkit allowed us to define the agent's observations, actions, and rewards, and facilitated communication between the Unity environment and our Python-based training scripts.

This problem turned out to be quite hard for all the form of reinforcement learning that we tried. Our original state space and parameters didn't yield good results with the snake attaining mostly negative or neutral results forever. The snake loved to spin forever and run into it self while missing the fruit. After a lot of trial and error we got some better results which we will show.

4.1 Policy

We created the Snake game with four possible actions, corresponding to movements in the four cardinal directions. The snake moves in a grid-based, Manhattan style pattern (see Figure 1).

For DQN, and its extensions, to be effective, it is crucial to define a meaningful state representation and reward function. After experimenting with various state spaces and reward functions, we found the most effective state representation to be the normalized positions of the snake's head and the food, along with a binary indicator for obstacles in a 5x5 grid around the snake.

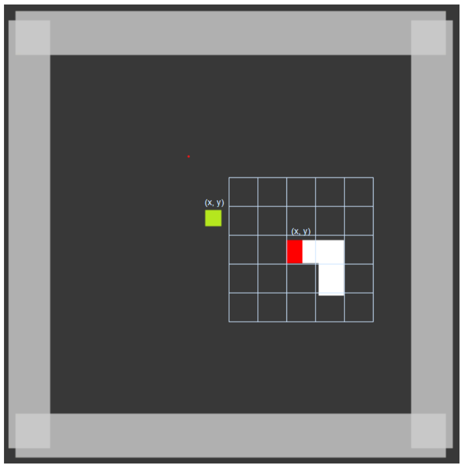

Figure 1: Snake environment

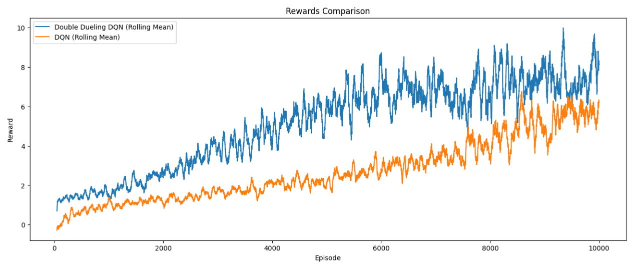

Figure 2: Training performance of DQN algorithms

The reward function was designed as follows:

- Negative reward for the snake hitting itself or a wall.
- Positive reward for the snake eating food.
- Small positive reward for each step the snake takes towards the food.

This setup encouraged the snake to navigate effectively towards the food while avoiding collisions.

4.2 Results

We trained the combination of DQN algorithms over 10,000 episodes, each with a maximum of 250 steps. Our experience memory buffer size was set to 1000 transitions. To balance exploration and exploitation, we used an epsilon-greedy strategy, starting with an epsilon value of 0.9 and gradually decaying it to 0.1 over 100,000 steps.

During training, the results indicated that Double DQN and Dueling Double DQN algorithms converged more quickly to optimal maximum rewards compared to the standard DQN (see Figure 2). This improved convergence can be attributed to the reduction of overestimation bias in Double DQN and the efficient separation of state values and action advantages in Dueling Double DQN.

The training process demonstrated that more sophisticated algorithms like Double DQN and Dueling Double DQN significantly enhance the learning efficiency and robustness of the agent in the Snake game environment.

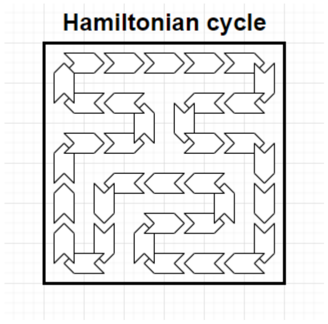

Figure 3: Hamiltonian cycle generated for a grid of 6x6

4.3 Hamiltonian cycle

After doing some research into different ways to beat the snake game effectively, we came across an obvious idea. The game of snake can be viewed as a graph and beating it as a *pathing problem*. As the snake must take a path that gets to the fruit but avoids its own body.

The simplest solution for this is to generate a Hamiltonian cycle. This is a path that visits every node in the graph exactly ones, and ends on the starting node. For our game this ensures that the snake doesn't run into it self and that it can keep traversing this same cycle until it has filled up the entire grid.

The Hamiltonian cycle on a general graph is an NP-Complete problem, this is not great as these are not easy problems to solve. Thankfully the Hamiltonian cycle on a grid of points is much easier with several knows solutions.

The one we picked is generating a minimum span tree on a grid and then walking it while sticking to one of the walls, left or right. This produces a Hamiltonian cycle that is twice the size of the MST. This is a limitation which limits us to even sides for our grid but for an NP-Complete problem we thing its a valid trade off.

The implementation of this algorithm was relatively straight forward and completely solves the snake game, allowing the snake to fill up the grid every time which is the largest score possible. While effective it's generally a bit boring as the snake just follows a predetermined path. We found interesting ways to improve this algorithm and make it more exciting and faster by having the snake cut of sections of the Hamiltonian cycle. This can be done with a search algorithm such as DFS or A*. Sadly we did not have time to implement these improvements for this paper.

5 CONCLUSION

To conclude, crafting the perfect policy for the Snake game is challenging. In our setup, the agent learned a policy that approached the optimal solution. However, we observed that the snake often found its way into dead ends. This issue arises from the limited view provided by the small grid surrounding the snake. While including the entire grid in the state space would provide more information, it would also introduce excessive complexity, making it impractical for larger grids.

Balancing the trade-off between the amount of information in the state space and the computational feasibility is a key challenge, that we also presented an alternative approach. Future work could explore more sophisticated state representations or multi-scale approaches to improve the agent's performance and avoid such pitfalls.

REFERENCES

[1] Matteo Hessel, Joseph Modayil, Hado van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. 2017. Rainbow: Combining Improvements in Deep Reinforcement Learning. arXiv:1710.02298 [cs.AI]

[2] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602* (2013).

[3] Hado Van Hasselt, Arthur Guez, and David Silver. 2016. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 30.

[4] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Hasselt, Marc Lanctot, and Nando Freitas. 2016. Dueling network architectures for deep reinforcement learning. In *International conference on machine learning*. PMLR, 1995–2003.