Multi-snake game using Multi Agent Reinforcement Learning

Soumyadeep Thakur
193050033

Samrat Dutta
193050026

Shreyansh Jain
193050040

Jay Bansal
193050004

Abstract

The classical game of snake is a single-player game where the main aim of the player is to collect food as rewards and stay in the board alive without colliding with the walls or itself. This setup can be modelled as a Markov Decision Process and solved using reinforcement learning wherein a single agent acts upon the environment, collecting appropriate rewards for appropriate transitions. In this project we have tried to extend this setup to a two-player environment, where each snake is trying to compete against the other and collect the food while staying alive. This setting has been modelled as a zero-sum game. We have used Minimax-DQN to find the optimal policies for each snake.

1 Introduction

The reinforcement learning problem is focused on training an agent by interacting with the world or the environment by taking some actions. The agent tries to learn an optimal strategy following which it would reach its target goal efficiently. Actions, in return, yield rewards, which could be positive, negative or zero. The Agent's sole purpose is to maximize the total reward it collects over an episode, which is everything that happens between an initial state and a terminal state.

DeepMind introduced the first deep reinforcement learning method, Deep Q Network(DQN) in (1). Following this proposal, Reinforcement Learning (RL) has been applied in various gameplay learning research. DQN outperforms all the prior approaches and surpasses human level performance on various games. This breakthrough lit up many similar research.

Multi-Agent Reinforcement Learning (MARL) has received an increasing interest in the last few years. To control a single agent, Markov Decision Processes (MDPs) are generally used. Markov/Stochastic games are generally used as an extension to this for multi-agent systems.

In this work we aim at training two snake agent to play a modified version of the classic snake game. Both the agents are present on the board simultaneously, each trying to collect maximum number of food items possible, while avoiding the boundary walls. Since another snake is also present in the environment, it acts as an barrier. Thus if one snake hits the other, it gets killed and gets penalised while the other snake awarded with a reward and continues with the gameplay.

2 Related Work

The single agent reinforcement learning framework (2) is based on the model, where an agent interacts with the environment by selecting actions to take and then perceiving the effects of those actions, a new state and a reward signal indicating if it has reached some goal (or has been penalized, if the reward is negative). Deep Q Networks (DQNs) were applied to Atari games with much success

in (1). Ever since, single Reinforcement Learning algorithms have been used vastly in game playing, including video games as well as in board games like Chess and Go (8; 7). Mnih *et al.* (6) also have used deep reinforcement learning to accomplish Human-level control and used deep convolutional neural network to approximate the optimal action-value function. A brief review of game playing with reinforceent learning has been provided in (3)

In multi-agent settings (10), general decision making is intractable due to the exponential growth of the problem size with increasing number of agents. The big difference with single agent RL resides in the fact that each agent probably has some effect on the environment, so actions can have different outcomes depending on what the other agents are doing.

This is precisely the difference that poses problems when applying reinforcement learning techniques to the multi-agent domain. Usually, those are designed to solve stationary environments and, from the point of view of each agent, the environment is no longer stationary.

Littman (4) showed Markov games as a framework for MARL. He used the modification of general *Q-learning* formulation, called the minimax-Q learning algorithm using a simple two-player zero-sum Markov game. In his framework, one learner was trained against a random opponent, and the other against another learner of the same architecture. Yang *et al.* (5) have given a theoretical analysis for the Minimax Deep Q-network (DQN) algorithm for Markov Zero-Sum Games with 2 players.

Perolat et al. (9) carried the work forward and provided an analysis of error propagation in Approximate Dynamic Programming using three algorithms viz. Approximate Value Iteration, Approximate Policy Iteration and Approximate Generalized Policy Iteration applied to zero-sum two-player Stochastic Games.

In this work, we use Markov Zero-Sum Game to implement the multi-snake game, and use Minimax-DQN to train the snakes to play the game.

3 Proposed Solution

3.1 Modelling multi-snake game in MARL setting

- **State**: defined as a collection of frames
- **Actions**: Each snake can take 3 actions - *Turn Left*, *Turn Right* and *Maintain Direction*
- **Rewards**: A collision with the boundary wall or another agent, is awarded with a negative reward. Collecting a fruit, a positive reward is awarded, else 0 reward is obtained by the agent.

3.2 Markov Zero-Sum Game

A two-player zero-sum Markov game is defined as a 6-tuple (S, A, O, P, R, γ) where,

$S = \{s_1, s_2, ..., s_n\}$ a finite set of game states
$A = \{a_1, a_2, ..., a_m\}$ finite sets of actions for player 1
$B = \{b_1, b_2, ..., b_k\}$ finite sets of actions for player 2

P is a Markovian state transition model, $P(s, a, b, s')$ is the probability that s' will be the next state of the game when the players take actions a and b respectively in state s.
R is a reward function, $R(s, a, b)$ is the expected one-step reward for taking actions a and b in state s.
$\gamma \in (0, 1]$ is the discount factor for future rewards.

3.3 Multi-snake game as Zero-Sum Game

The multi-snake game can be modelled as a Zero-Sum Game. A zero-sum stochastic game can be used to model such situations where more than one players are acting on the environment and each trying to maximize an objective which is complementary to the other player's objective. In such a situation the gain of one is equally balanced by the net loss of the other agent. Equivalently it can be formalised as a situation where one agent is trying to maximize its own gain, when the behaviour of the other agent can be thought of as coming from the environment in an unpredictable manner.

In such situations, the worst-case condition may be considered, where the objective of the current player is to find a policy that guarantees the best performance under worst-case behaviour of the other player.

In this game, the rewards received by the snakes are made complementary to each other. The specific way in which rewards are received by each snake is elaborated in Section 4.3. When one snake receives a reward of r, the other snake is automatically given a $-r$ reward, such that the gain of one snake incurs an equivalent amount of loss to the other snake. Thus the net reward stays constant.

3.4 State and Action representation

A state in the game is represented as a stack of 4 *frames* **S** . A *frame* is a snapshot of the current position of the board. The idea behind using a stack of frames is to ensure that the approximation network understands both the position and the direction in which a particular snake is moving. Each snake is associated with a different color so that the model can well differentiate between them.

A is the set of actions corresponding to the first snake and **B** is the set of actions corresponding to the second snake. Each snake can choose one from the following actions : TURN_LEFT, TURN_RIGHT, STAY.

We have implemented 2 versions of the Minimax-DQN algorithm. In Algorithm 1 only 1 neural network is shared by both the snakes. In Algorithm 2, we modify the Minimax-DQN to include two neural networks are trained, one for each snake.

Algorithm 1: Minimax Deep Q-Network (Minimax-DQN)

Input: Zero-Sum Markov game (S, A, B, P, R, γ), replay memory M, number of iterations T, mini-batch size n, exploration probability $\epsilon \in (0, 1)$, a family of deep Q-networks $Q_\theta : S \times A \times B \to \mathbb{R}$ and a sequence of step sizes $\{\alpha_t\}_{t\geq 0}$.

Initialization:
Initialize the replay memory M to be empty.
Initialize the Q-network with random weights θ.
Initialize the initial state S_0.

for $t = 0, 1, 2, ..., T$ **do**

With probability ϵ, choose A_t uniformly at random from A. With probability $1 - \epsilon$, sample A_t greedily as $\max_{a\in A} \min_{b\in B} Q_\theta(S_t, a, b)$

Execute A_t and observe B_t (computed equivalently), reward R_t satisfying $-R_t \sim R(S_t, A_t, B_t)$, and the next state $S_{t+1} \sim P(\cdot|S_t, A_t, B_t)$.

Store transition $(S_t, A_t, B_t, R_t, S_{t+1})$ in M.

Experience replay: Sample random mini-batch of transitions $\{(s_i, a_i, b_i, r_i, s_i^{'})\}_{i\in[n]}$ from M.

for *each* $i \in [n]$ **do**

Compute the target:

$$Y_i = r_i + \gamma \cdot \max_{a\in A)} \min_{b\in B} Q_\theta(s_i^{'}, a, b)$$

end

Update the Q-network: Perform a gradient descent step

$$\theta \leftarrow \theta - \alpha_t \cdot \frac{1}{n} \sum_{i\in[n]} [Y_i - Q_\theta(s_i, a_i, b_i)] \cdot \nabla_\theta Q_\theta(s_i, a_i, b_i)$$

end

Output: Action-value function Q_θ.

.

Algorithm 2: Minimax-DQN with a neural network for each player.

Input: Zero-Sum Markov game (S, A, B, P, R, γ), replay memory M, number of iterations T, mini-batch size n, exploration probability $\epsilon \in (0, 1)$, 2 families of deep Q-networks $Q_\theta : S \times A \times B \to \mathbb{R}$, $Q_\phi : S \times B \times A \to \mathbb{R}$ and a sequence of step sizes $\{\alpha_t\}_{t\geq 0}$ and $\{\beta_t\}_{t\geq 0}$.

Initialization:

Initialize the replay memories M_1 and M_2 to be empty.
Initialize the Q-networks with random weights θ and ϕ.
Initialize the initial state S_0.

for $t = 0, 1, 2, ..., T$ **do**

With probability ϵ, choose A_t uniformly at random from A, B_t uniformly at random from B. With probability $1 - \epsilon$, sample A_t greedily as $\max_{a\in A} \min_{b\in B} Q_\theta(S_t, a, b) and \mathrm{B}_t$ greedily as $\max_{b\in B} \min_{a\in A} Q_\phi(S_t, b, a)$

Execute A_t, B_t and observe reward R_t satisfying $-R_t \sim R(S_t, A_t, B_t)$, and the next state $S_{t+1} \sim P(\cdot|S_t, A_t, B_t)$.

Store transition $(S_t, A_t, B_t, R_t, S_{t+1})$ in M_1, $(S_t, B_t, A_t, -R_t, S_{t+1})$ in M_2.

Experience replay: Sample random mini-batch of transitions $\{(s_i, a_i, b_i, r_i, s_i^{'})\}_{i\in[n]}$ from M_1.

for *each* $i \in [n]$ **do**

Compute the target:

$$Y_i = r_i + \gamma \cdot \max_{a\in A)} \min_{b\in B} Q_\theta(s_i^{'}, a, b)$$

end

Update the Q-network: Perform a gradient descent step

$$\theta \leftarrow \theta - \alpha_t \cdot \frac{1}{n} \sum_{i\in[n]} [Y_i - Q_\theta(s_i, a_i, b_i)] \cdot \nabla_\theta Q_\theta(s_i, a_i, b_i)$$

Experience replay: Sample random mini-batch of transitions $\{(s_i, b_i, a_i, r_i, s_i^{'})\}_{i\in[n]}$ from M_2.

for *each* $i \in [n]$ **do**

Compute the target:

$$Y_i = r_i + \gamma \cdot \max_{b\in B} \min_{a\in A} Q_{\phi^*}(s_i^{'}, b, a)$$

end

Update the Q-network: Perform a gradient descent step

$$\phi \leftarrow \phi - \beta_t \cdot \frac{1}{n} \sum_{i\in[n]} [Y_i - Q_\phi(s_i, b_i, a_i)] \cdot \nabla_\phi Q_\phi(s_i, b_i, a_i)$$

end

Output: Action-value function Q_θ and Q_ϕ.

4 Experiments

4.1 Game Environment

In this work, we used the snake game environment developed in the repository `https://github.com/YuriyGuts/snake-ai-reinforcement`. The snake environment developed in the above

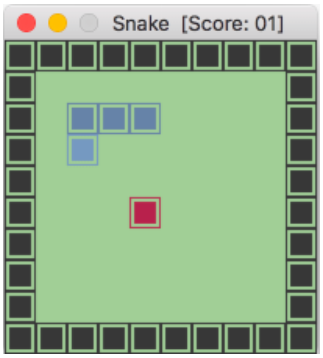

Figure 1: Single Player Snake

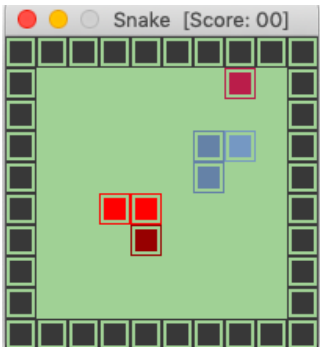

Figure 2: 2 Player Snake

mentioned repository is for a single snake. The environment was modified for 2 snakes and the termination rules and rewards were changed accordingly. Sample shots from the game environment is shown in 4.1. The game was developed using the *pygame* library for *python*.

4.2 Rules

Two snakes moves on a square 2D grid on which fruits appear randomly. For our experiment, the grid size was chosen to be 10 x 10. A snake grows by one cell on eating the fruit. A snake dies when its heads bumps into the wall or into its own body or into the body of the other snake.

Two variants of the game were implemented. In the first variant - **Variant 1**, the game ends as soon as one snake dies. In the second variant - **Variant 2**, the game continues even after 1 snake dies. The dead snake is purged from the environment, and the other snake continues to play and eat fruits.

4.3 Rewards

Each snake receives a reward of -1 when it collides with the wall or with the other snake. For an uneventful transition, where a snake neither collides nor eats a fruit, the snake gets a reward of 0. When a snake eats a fruit, it gets a reward equal to the current length of the snake. So, the reward a snake gets for eating a fruit increases with the length of the snake and hence on the number of fruits it has eaten in the past. Consequently, the other snake gets a negative reward with magnitude equal to the reward obtained by the other snake. Similarly, when one snake dies the other gets a $+1$ reward.

4.4 Results

The snakes (agents) were trained using Minimax-DQN as discussed above. Our deep neural network consists of 2 Convolutional Neural Network (CNN) layers, with ReLu activation, followed by a dense layer with linear activation. The output layer has 9 units (number of actions) with linear activation. Despite being non-linear ReLu activation function was chosen because of convergence bounds shown in (5). The code for the neural network was developed using *keras*, a *python* library.

Since an epsilon greedy behaviour policy was used by both snakes, the value of the epsilon was annealed from a maximum of 1.0 to a minimum of 0.05. Both the algorithms Algorithm 1 and Algorithm 2 were implemented, and the game was run for 50000 episodes.

We notice some interesting results after training the agents:

Table 1: Statistics over 100 episodes for each algorithm and game version

Algorithm	Version	S1 fruits	S2 fruits	Avg Episode Length	S1 dying first
Algorithm 1	1	75	2	5.25	7
Algorithm 1	2	60	6	7.00	10
Algorithm 2	1	40	37	9.04	44
Algorithm 2	2	56	48	74.39	47

Version 1: *Whenever one of the snakes die, the episode ends.*

For the algorithm 2, we notice that both snakes try to reach for the food initially. In some cases, one of the snake intentionally block the way of the other snake thus causing it to collide. In few cases one snake intentionally collides on the wall to prevent the other snake from eating the fruit.

Since the game ends every time one of the snakes dies, the rewards obtained by both the snakes are not very high at all. In fact, in an episode, it is common that a snake eats no fruit in an episode. Also, it follows that the episode lengths are not long at all.

We notice that on training using algorithm 1, one snake always collides with the wall quite early in the game.

Version 2: *If either of the snakes die, the other continues with the game.*

For the algorithm 2, we notice that snakes collide when chasing a fruit. Also sometimes a snake intentionally blocks the other causing it to die. However, we expected that the snake that survives will henceforth play out the game optimally, collecting more and more rewards after training. We notice that the snake that survives, doesn't live long enough, growing only a little and then either colliding with a wall, or looping continuously. We expect the surviving snake to capitalize on its survival if we train the network for long enough.

A brief summary of the total of fruits eaten both snakes (S1 and S2) and their survival status over 100 episodes are tabulated in Table 1. The number of times Snake 1 dies first is included.

5 Conclusion and Future Work

In this project we implemented two varieties of Minimax-DQN to play the multi-snake game. Interestingly, we note that the agents trained on Algorithm 2, in which we had separate neural networks for the two snakes plays the game more naturally. However, the results are partially what we expected and there are further improvements to be done. Training the agents for a sufficiently long time and proper reward shaping may give much better results, avoiding the continuous looping that we observe in the Version 2 of our game.

The Two Agent Snake game can be extended to a similar problem where multiple agents might act on the environment as a generalised Multi Agent Snake game. In such a situation, the reward function has to be appropriately modified so as to keep the game zero-sum. Another improvement could be that to refine the approximation function used, which could further improve the result, consequently the agents might perform better and learn better strategies. Other possibilities include that, each agents could be first trained alone, then use that trained model as a base and then proceed to train both the agents together in the Two-Agent environment. Given enough computation time, Monte Carlo Tree Search methods could be tried in this Two Snake situation.

References

[1] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D. and Riedmiller, M., 2013. Playing atari with deep reinforcement learning. arXiv preprint arXiv:1312.5602.

[2] Goncalo, N. (2005). From Single-Agent to Multi-Agent Reinforcement Learning: Foundational Concepts and Methods.

[3] Justesen, N., Bontrager, P., Togelius, J. and Risi, S., 2019. Deep learning for video game playing. *IEEE Transactions on Games*.

[4] Littman, M.L. (1994) Markov games as a framework for multi-agent reinforcement learning. *In Machine learning proceedings 1994* (pp. 157-163). Morgan Kaufmann.

[5] Yang, Z., Xie Y. & Wang Z. (2019) A Theoretical Analysis of Deep Q-Learning. *arXiv:1901.00137* [cs.LG]

[6] Mnih, V., Kavukcuoglu, K., Silver, D. *et al*. Human-level control through deep reinforcement learning. *Nature* 518, 529–533 (2015) doi:10.1038/nature14236

[7] Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A. and Chen, Y., 2017. Mastering the game of go without human knowledge. *Nature*, 550(7676), p.354.

[8] Silver, D., Huang, A., Maddison, C.J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M. and Dieleman, S., 2016. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587), p.484.

[9] Julien, P., Bruno, S., Bilal, P., Olivier Pietquin. Approximate Dynamic Programming for Two-Player Zero-Sum Markov Games; Proceedings of the 32nd International Conference on Machine Learning, PMLR 37:1321-1329, 2015.

[10] M Egorov - Multi-agent deep reinforcement learning; CS231n: Convolutional Neural Networks for Visual, 2016