ECE239AS Final Report

Sangjoon Lee
Department of Computer Science
University of California, Los Angeles
Los Angeles, CA 90024
aaccjjt@g.ucla.edu

Sangmin Lim
Department of Mechanical Engineering
University of California, Los Angeles
Los Angeles, CA 90024
limsm3@g.ucla.edu

Donghun Noh
Department of Mechanical Engineering
University of California, Los Angeles
Los Angeles, CA 90024
donghun.noh@ucla.edu

Abstract

Slither.io is a mass multiplayer online game with a simple game mechanism. With continuously changing environment with random actions imposed by numerous different players, multiplayer online games are non-deterministic, which further pushes the boundary of the performance of RL algorithm. In this project, we performed comparative analysis of different RL algorithms. We examined and applied Deep Q- Network (DQN), Advantage Actor Critic method (A2C), and Proximal Policy optimization (PPO), with Slither.io as a benchmark for each algorithms performance measure. Through comparative analysis, we could determine and compare the efficacy of algorithm. From our findings, we report that the PPO outperformed DQN and A2C algorithm in Slther.io

1 Introduction

Since the development of Deep Q-learning, Atari games served as a benchmark for comparing the performances of different reinforcement learning (RL) algorithms [1]. The reason for Atari games being such a strong benchmark for RL roots from its complexity due to high dimensional visual input and versatile objectives, which resemble tasks that robots or humans would do. However, multiplayer online games poses an additional challenge on the RL performance. With continuously changing environment with random actions imposed by numerous different players, multiplayer online games are non-deterministic, which further pushes the boundary of the performance of RL algorithm.

Slither.io is an online game where a player controls a snake using a mouse. The basic objective of the game is to record high scores by eating color pallets to maximize the snake's body length. The game ends when the player's head collides into another snake's body. Once the player is eliminated then the snake turns into pellets for other players to feast on.

We chose Slither.io [2] due to its three key characteristics. First, Slither.io provides simple mechanism for control (controllable through mouse position). Second, it has high randomness in the environment. Slither.io constantly changes distribution of environment. All players and pellets are spawned in random position, in random time, and the players behaviors are random strategic behavior conducted by humans. This feature of Slither.io is a challenging topic to tackle due to high variance of the visual input and therefore interesting. Lastly, Slither.io is an online game that does not have a specific terminal state. Constructing a learning algorithm from an online environment requires real-time or close to real-time data extraction from the web which poses another challenge for the learning to

Preprint. Under review.

extrapolate the information if the information is not real time. Also, without a specific terminal state, in other words, non-episodic. Therefore, the RL algorithm will always have to update the policy or values without reaching a terminal state.

With aforementioned characteristics, Slither.io makes a good benchmark for an online game performance analysis. In this project, we performed comparative analysis of different RL algorithms. We examined and applied Deep Q- Network (DQN) [1], Advantage Actor Critic method (A2C) [3], and Proximal Policy optimization (PPO) [4], with Slither.io as a benchmark for each algorithms performance measures.

2 Related works

Several efforts were made to train Slither.io using RL. During our research, we discovered there were numerous efforts in implementing RL algorithm for Slither.io. We could find several github repositories with codes. However, only a few written reports were found from 2017 and 2019 [5, 6, 7]. Students at Stanford attempted to train agents to play Slither.io using the Q-learning algorithm by incorporating human demonstrations and prioritized replay. However, median final score of the trained agent was only able to acheive 1/3 of the score for the human palyers even after incorporation of human demonstration [5]. Another attempt from a student from California Polytechnic State University was made through deep Q-learning implementation with carefully conducted computer vision algorithm that does the image segmentation through making dictionary of the segmented images as pseudo-labeled features. However, due to the time taken for the computer vision processing, the network was inefficient and slow to make decision. The time discrepancy between the action and the image lead to unsuccessful training, averaging 10 hours to show positive return [6]. Lastly, a different type of unsupervised learning, Neuroevolution of Augmenting Topologies (NEAT), implementation on Slither.io was reported [7]. This paper showed great performance of an agent reaching an average score of 502, three times higher than the score of average human players reported in Ref [5]. However, NEAT was out of scope of our analysis on reinforcement learning since it implements the neuroevolution. None of the written documents we could find provided any comparative analysis on existing methods performance on Slither.io. Several github repositories had multiple algorithms implemented for Slither.io, yet they did not provide thorough analysis or comparison of each methods.

3 RL algorithms

We now introduce the algorithms that were applied for training of the Slither.io.

3.1 Deep Q Network

Deep Q Network (DQN) is the first deep reinforcement learning method published in 2013 and 2015 by DeepMind [1, 8]. Unlike Q-learning, DQN uses a neural network to approximate the Q-value function as shown in figure 2. There are 4 techniques that make DQN performs much better than Q-learning as follows: experience replay, target network, clipping rewards, skipping frames. Experience replay is used to prevent the overfitting problem of deep neural networks. All data required for Q learning such as state transition, rewards, and actions are stored in experience replay and update neural networks using mini-batches. This technique helps to increase learning speed with mini-batches and reduce the correlation between experiences in updating DNN. Target network makes parameters of target function fixed and replaces them with the latest network every thousands steps. Clipping rewards is a technique to limit rewards to values between -1 and 2, which

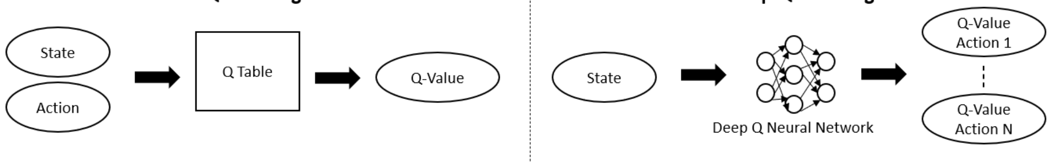

Figure 1: Comparison between Q-learning and DQN

makes training more stable. Skipping frames is used to reduce computational cost and gathers more experiences by skipping frames. Usually, DQN calculates Q values every 4 frames. In the DQN paper, DQN shows great performance in many Atari games because of the superior performance of the deep neural network to deal with high dimensional states.

3.2 A2C

Advantage Actor Critic (A2C) is a synchronous version of the Asynchronous Advantage Actor Critic (A3C), which is one of the state-of-the-art policy-based methods that is different from value-based methods in terms of the fact that policy-based methods directly optimize the policy without using a value function. Since A2C and A3c are policy-based methods, they show great performance on the environment having high dimensional and stochastic continuous action spaces. Furthermore, A2C and A3C have successfully decrease its gradient variance by using an advantage function as

$$A(s_t, a_t; \theta, \theta_v) = \sum_{i=0}^{k-1} \gamma^i r_{t+i} + \gamma^k V(s_{t_k}; theta_v) - V(s_t; \theta_v) \tag{1}$$

, where k can vary from sate to state and is upper-bounded by t_max [3]. However, A3C has a delay issue to diminish the advantage of its asynchrony because even within an asynchronous system, synchronization for shared resources causes delay. By using synchronous implementation, A2C could more effectively use GPUs due to larger batch sizes and it shows better performance than A3C especially using larger policies [9].

3.3 Proximal policy optimization

Proximal policy optimization (PPO) stems from actor critic methods and considered one of the state of the art policy gradient based RL methods. Similar to A2C, PPO can learn "on-line" by bootstrapping. However, the actor critic methods does not prevent the new updated policy moving too far away from the previous policy, which in turn can result in bad update of policy which can be not recoverable. Trust region policy optimization (TRPO) was first introduced in order to deal with this unrecoverable malicious update [10]. The main equation for the optimization that was also used for PPO is,

$$\max_x \sum_{n=1}^{N} \frac{\pi_\theta(a_n|s_n)}{\pi_{\theta_{old}}(a_n|s_n)} \hat{A_n} \tag{2}$$

, where $\pi_\theta(a_n|s_n)$ refers to current policy,$\pi_{\theta_{old}}$ refers to old policy and $\hat{A_n}$ refers to the advantage function, which is the difference between the baseline value estimate and the bootstrapped Q function. This optimization was used along with KL-constraint as boundary on the policy update so that the updated policy does not move too far away from the previous policy. However, the implementation of KL divergence is relatively hard. Therefore, PPO implemented a clip function to optimize for the loss using the following loss function,

$$L_{\text{PPO}} = \hat{E}[min(r_t(\theta)\hat{(A_t)}, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)(\hat{A_t})] - c_1 L_t^{VF}(\theta) + c_2 S[\pi_\theta](s_t) \tag{3}$$

,

where the first part of the equation refers to the surrogate loss with $r_t(\theta)$ representing $\frac{\pi_\theta(a_n|s_n)}{\pi_{\theta_{old}}(a_n|s_n)}$, ϵ, representing the clipped boundary, $c_1 L_t^{VF}(\theta)$, represents the loss for the value function, and an entropy term for exploration also introduced in A2C, $c_2 S[\pi_\theta](s_t)$. Algorithm 1, shows the pseudo code for the update procedure of PPO. Notice that the algorithm goes over same data multiple times, however, the author of Ref [4] mentions that PPO still works due to the proximal trust region constrained by the clipped function.

Algorithm 1 PPO

```
1: for $iteration = 1, 2, \ldots$ do
2:     for $actor = 1, 2, \ldots, N$ do
3:         Run policy $\pi_{\theta_{old}}$ in environment for $T$ time steps
4:         Compute advantage estimates $\hat{A}_1, \ldots, \hat{A}_T$
5:     end for
6:     Optimize surrogate $L$ wrt. $\theta$, with $K$ epochs and minibatch size $M \leq NT$
7:     $\theta_{old} \leftarrow \theta$
8: end for
```

4 Methods

4.1 Environment setup

For environment setup, our team initially investigated OpenAI gym and OpenAI Universe. However, due to innate latency that arises from Virtual Network Connection (VNC) and OpenAI's shift of gear towards newly developed integrated platform (thereby, lack of maintenance on Universe), we decided to search for direct connection with the internet for the system integration. For this purpose, We relied on custom python libraries for mouse cursor control, keyboard control, and screen capture. Then, for web browser's textual data, we utilized Selenium Library to extract it into python script.

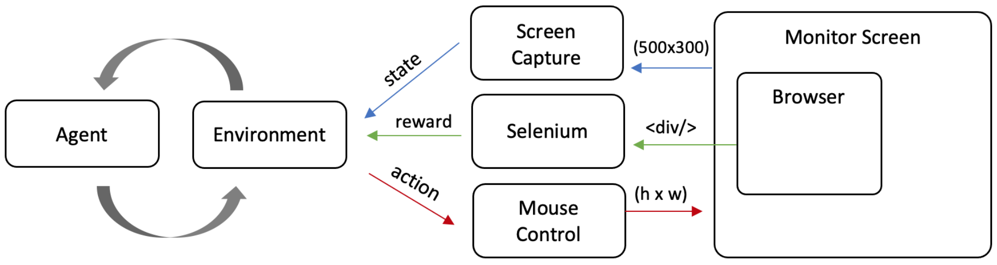

Figure 2: Environment Pipeline

The reinforcement learning environment is setup with following elements: action, observation, and reward. When the python environment consumes an action, it converts that action into moving mouse on the web. Then the environment returns the next state observation by taking a screen capture of the web browser. The reward is returned with the observation by connecting to the web and reading the snake's length using Selenium.

The observation space is a fixed rgb channel screen. To reduce the latency of the system, The environment opens the browser to 500 pixel by 300 pixel wide instead of using full screen. This effectively reduces the network weights for reinforcement learning agents while also reducing screen capture latency 4x times. The action space was defined to be discrete space of 8 cardinal and ordinal direction. We decided to use 8 because these 8 direction allowed worm to travel anywhere while also providing more flexibility in its locomotion then using just 4 cardinal directions. Finally, the reward was defined after theorizing and testing several different reward system. we decided to use increase or decrease of snake's length for a given timestep to determine negative or positive reward. Hence, the snake gains zero reward when it doesn't do anything. when the snake dies, reward of -20 is returned. Then, the environment restarts the game and the agent is disabled to have interaction with python environment until the python environment perceives that the browser has correctly and fully reloaded the game. The total latency of the environment differed per computer system, but they roughly measured 250-400ms per step. we were able to gain 3-4 frames per second. Reset time takes roughly 5 second.

4.2 Agent setup

The reinforcement learning agent was set up with pre-existing reinforcement learning python library called Stable Baselines. Within the Stable Baselines, we utilized DQN, A2C, and PPO library to create our agents.

DQN was set up with hyper parameter similar to original nature paper. However replay buffer size had to be limited due to PC's limited memory capacity. We used the following parameters. learning rate=0.0001, replay buffer size=5000, learning starts=2000, batch size=32, gamma=0.99, train freq=4, gradient steps=1, target update interval=5000, exploration fraction=0.1, exploration initial eps=1.0, exploration final eps=0.05.

A2C was set up with hyper parameter of the following: gamma=0.99, steps per update=5, value function coef=0.5, entropy coef=0.0, learning rate=0.0007

PPO was set up with hyper parameter of: learning rate=0.0003, steps per update=2048, batch size=64, epochs=10, gamma=0.99, clip range=0.2, entropy coef=0.0, value function coef=0.5

5 Results

Each of the Reinforcement Learning Algorithm had to be tested with different parameters to generate maximum result. In our case, given the general methods and environment described above, we gave variation to the settings and parameters to make our model perform better. For instance, we would try to modify the reward signal to better describe the goal of the agent.

We tried changing the reward by give varying value of penalty for dying. The penalty ranged from -10 to -10000. we also tried giving negative reward for failing to achieve increased length within a time step. Likewise we also tried resetting after certain timestep of idleness to encourage faster search for point. Finally, we tried changing observation state from using time series channel to single frame rgb channel.

5.1 Algorithm specific analysis

For DQN, it seems the best results were gathered when we used the environment that was described in the method. -20 penalty for death, and +1 for every increase in length. rgb channel observation state with zero reward for idleness. reset only after death.

During the training, the mean reward of the A2C was not stable along with consistently high variance throughout the training phase. Neither variance nor the mean reward had a stable increasing behavior even after 200K iteration even for the best model (-20 penalty for death, reward of difference in length for each frame). It is known through previous findings in the reinforcement learning community that conventional actor critic methods are prone to overcorrection and this trend is reflected in the training results.

PPO showed better performance on training Slither.io than DQN and A2C. We verified that PPO could train Slither.io by training the game several times with the same parameters as used for the successfully trained model. All the training curves obtained from the verification process tend to show increasing curves after 80-100k iterations as shown in Figure 3.

For all the cases, there were a limitation in that the latency between the update in javascript data feed from the web and the actual moment that the snake dies had discrepancy of at least 5 steps. We contribute this to be one of the limiting factor for the training.

In the finality of our evaluation, we had three models with best of their own environment and agent parameters. They were each trained 200,000 timesteps. These fully trained models conducted 5 episodes. Their average scores were measured: DQN: 125.6 A2C: 70.6 PPO: 137.4

As can be seen in Figure 3, the graph shows faster and greater convergence with PPO then either A2C or DQN. After training for 200,000 steps we each tested the trained algorithm N number of episodes and found that PPO still dominated in overall score average. The performance of A2C was compromised due to the update mechanism of policy for A2C, where the "over correction" of policy will update the global policy and negatively affect the training process. This drawback of A2C is complemented by introducing trust region based policy updates algorithm such as PPO.

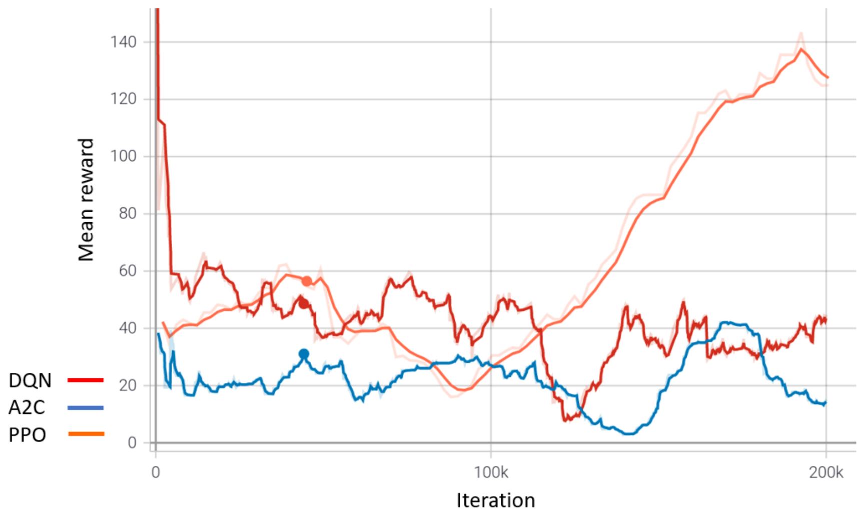

Figure 3: Training curves of three different RL algorithms: DQN, A2C, and PPO.

6 Conclusion

By performing comparative analysis on the 3 reinforcement learning algorithm, it is shown that while A2C performs poorly on online gaming system, PPO remains viable option that outperforms DQN. Given the recent advancement PPO, actor critic model may become viable reinforcement model that shows robust performance with flexibility to include continuous actions space for gaming system.

References

[1] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing Atari with Deep Reinforcement Learning. pages 1–9, 2013.

[2] Steve Howse. Slither.io. `http://www.slither.io`, 2016.

[3] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937. PMLR, 2016.

[4] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[5] Joan Creus-Costa and Zhanpei Fang. Learning to play SLITHER.IO with deep reinforcement learning CS229 project report. Category: Theory and Reinforcement Learning. pages 1–6, 2019.

[6] James Caudill. Slither.io Deep Learning Bot. (June), 2017.

[7] Mitchell Miller, Megan Washburn, and Foaad Khosmood. Evolving unsupervised neural networks for Slither.io. *ACM International Conference Proceeding Series*, 2019.

[8] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[9] OpenAI. Openai universe official blog. `https://openai.com/blog/baselines-acktr-a2c/`, 2017.

[10] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. Trust region policy optimization, 2017.