Reinforcement Learning in Snake Game

Ailin Zhang, ailin@utexas.edu, az8287

Qinqin Zhang, zhangqinqin1204@gmail.com, qz3253

1. SOURCE CODE AND VIDEO LINK

Our code has been uploaded via GradeScope. We were using source code from GitHub as our framework. Their code implemented DQN with Tensorflow for the snake game. The link to the reference code is: https://github.com/YuriyGuts/snake-ai-reinforcement. We were using the framework's game environment, but built our DQN, Double DQN, Dueling DQN, N-step DQN, and DQN with CER models with Pytorch by ourselves. We kept using the framework's experience replay code, but also changed that part when implementing our DQN extensions.
Video link:
https://www.youtube.com/watch?v=ug_1qvB0bGM

2. ABSTRACT

In this project, we trained several different agents to play the Snake game. We firstly implemented Deep Q-Learning, which is also well-known as DQN, as our benchmark. Then, we extended the DQN to other reinforcement learning algorithms and tested those models' performance. Some of them learnt faster than the original one, while some did not perform well in our game. We compared those experiment results and also discussed about possible explanations for their better or worse performance.

3. INTRODUCTION

In recent years, Reinforcement learning algorithms have been shown to be effective for the AI games. Specifically, we focus on the Snake game in our work. In Section 4, we briefly summarized some previous algorithms related with reinforcement learning and the Snake game. In Section 5, we clarified our state and action space as well as reward settings of the environment. Then in Section 6, we introduced our DQN with experience replay, Double DQN, Dueling DQN, n-step DQN and DQN with combined experience replay models. We tested the aforementioned model and experiment results are discussed in Section 7.

4. RELATED WORK

Reinforcement learning is a goal-oriented technique for performing decision-making tasks. The core idea is to train agents to learn from an interactive environment and try to maximize the rewards by either exploring or exploiting. Reinforcement learning algorithms have been applied to playing games for years and have been proved to be practically effective. Classic reinforcement learning algorithms such as SARSA and Q-learning has shown their comparable results with heuristic benchmarks (Ma et al., 2016[1]). The best-known breakthrough would be the TD-gammon (Tesauro, 1995[2]), which combined neural networks with TD approaches and greatly outperformed previous computer programs with regard to the ability to win the backgammon game. Researchers such as Abbeel, P. et al., 2004[3] also explored ways to use expert judgement for training, which have also been applied for game AI in recent years.

Recently, motivated by the success of deep neural networks in computer vision and speech recognition, reinforcement learning has been combined with deep neural network to directly process raw inputs and effectively train data (Minh et al., 2013[4]). Deep Q-Networks (DQN) has been widely used for games. Diallo et al.[5] used DQN on doubles pong games and the agents were able to jointly learn to divide their area of responsibility and modify their behaviors, showing the ability of coordinating and cooperating to take optimal actions. With the success of vision-based DQN in Atari games, Yoon et al.[6] showed the potential of DQN approach for the real-time fighting game with large number of actions for the two players.

However, DQN also has many limitations. Firstly, DQN produces sparse and delayed rewards that may lead to ineffective learning of correct policies. Wei et al.[7] developed a refined DQN model with a carefully designed reward mechanism to solve the sparse and delay problem, enabling the agent to play the classi-

cal Snake Game, whose constraint gets stricter as the game progresses. Secondly, DQN needs to remember previous states and learn from these states. In practice, DQN agent is trained with four previous states, and thus cannot deal with games that require more than four previous screens. In that case, the game would become a Partially Observable Markov Decision Process (POMDP). Hausknecht et al.[8] presented Deep Recurrent Q-Network (DRQN), combining DQN with LSTM. DRQN is able to deal with POMDPs and better handles the loss of information than DQN does. Lample and Chaplot[9] augmented a DRQN model with high-level game information and incorporation of independent networks responsible for different phases, showing great performance in more complex games like FPS with 3D environment. Thirdly, DQN sometimes suffers from the problem of overestimation, tending to learn unrealistic high action values. To tackle this problem, Van Hasselt et al.[10] introduced a Double DQN algorithm, which extended Double Q-learning algorithm to a semi-gradient method, allowing the use of Deep Neural Network.

In our project, we focus on the Snake Game. The Snake Game is a classic game where a snake aims to achieve the highest score by eating apples and avoiding colliding with walls or its own body. Algorithms inspired by different AI concepts for the Snake Game have been developed over years. Yeh, Jia-Fong et al.[11] built their controller with a set of rating functions, followed by an evolutionary algorithm to calculate weights for the aggregated weighted sum of the rating functions corresponding to each action selection. Crevier, F. et al.[12]introduced adversarial-based strategies for the snake game and compared them with reinforcement learning algorithms.

While DQN has been proved to be one of the most successful reinforcement learning algorithms in the Snake game, we would like to consider improvements of DQN such as applying Double Q-learning concepts to it, as introduced by Van Hasselt et al.[10] Another option is to use proper eligibility traces instead of implementing TD(0). As experimented by Crevier, F. et al,[12] setting lambda from 0.1 to 0.2 can yield better results than 0 in their game environment. In this project, we plan to implement Double DQN and DQN with proper eligibility traces and evaluate their performance in the Snake game comparing with the original DQN algorithm.

5. PROBLEM DESCRIPTION

Snake game is a popular single-player game. The player, which in our project is the agent trained with reinforcement learning algorithms, controls the direction of the snake and tries to eat more fruits without hitting into the wall or its own body. We suppose that at any time, only one fruit appears in the playground, and a fruit appears immediately after the previous one has been eaten with its position randomly selected.

5.1 State Space

States are represented by cell types. For instance, cell [x,y] can be empty/fruit/snake head/snake body/wall. If the size of the playground is $m \times n$, then state space would have $5^{(m*n)}$ different possible states. In our experiments, we concatenated the 2-dimensional playground of 4 continuous frames to form an input to the neural network.

5.2 Action Space

Action space contains three actions: maintain direction, turn left and turn right.

5.3 Reward

For each timestep movement, the agent gets a reward of 0. When having eaten a fruit, the agent gets a reward with value of its length. When died (hitting walls or body), the agent receives a reward of -1.

6. METHODS

6.1 DQN

In Q-Learning, action values update following the rule of Bellman equation. In Deep Q-Learning, our loss function is the mean square of the TD error of Q-Learning:

$$Loss = MSE[Q(S_n, A_n), R_{n+1} + \gamma * \max_a Q(S_{n+1}, a)]$$

6.1.1 Network Architecture and Hyper-parameters

The DQN network serves as a function approximation for input states' state-action pairs. The inputs to our deep neural network were the concatenated single-frame states of the last 4 frames, and the outputs would be the value of each of the three actions, maintain direction, turn left and turn right, given the input states. We tested Deep Q-Learning with different architectures (number of hidden layers, hidden dimension, activate function, exploration rate, etc.) and hyper-parameters (learning rate and discounting), and found that using the architecture as shown in Table 1 with learning rate

set to 0.001 led to good results.

As Table 1 shows, Our network consists two convolutional layers and two fully-connected layers. It receives a batch of $frame_num \times gridwidth \times gridheight$ input data, then processes the input with two continuous convolutional layers, each followed by a ReLU activation layer. The outputs of the second convolutional layer were flattened and fed into the first fully-connected layer, followed by a ReLU activator, and finally put into the last fully-connected layer with output size the same as number of actions.

6.1.2 Experience Replay

To use previous experience more efficiently and obtain better convergence for the neural network function approximator, we applied experience replay to our learning process, where we stored transitions in a replay buffer and sampled batch data from that buffer during training, instead of updating the value function immediately. Using experience with mean square error, we were able to learn with previous data multiple times. By uniformly sampling from buffer data, experience replay also provides uncorrelated data for training.[13]

6.2 DQN with Combined Experience Replay (CER)

When we set the buffer size to a fixed value, we found that the agent's performance was not robust to the changing of buffer size. Small buffer size would result in bad convergence performance. That might be explained considering that a small replay buffer was not enough to contain a wide range of experiences. However, a large buffer not only slows training, but also has a negative influence on the agent's performance.[14]
Zhang, Shangtong et al. have presented a simple O(1) method to remedy that sensitivity. They proposed a combined experience replay (CER) method, which corrected the sampled transitions by adding the latest transition at each time step. In our experiment, we also implemented CER and compared that with our previous model, using unlimited buffer size, to see if CER could lead to a better result.

6.3 Double DQN

Since DQN selects the maximum action value as an approximation for the maximum expected action value, the estimations of action values are subject to a positive bias.[13] To eliminate that bias, we tried to combine double learning with our Deep Q-Learning model. We defined two independent DQN networks and randomly selected one of the two networks to update, with the other one used to get estimations of action values at each training step. It turned out that Double-DQN has led to a significant improvement to our agent's performance.

6.4 n-step DQN

DQN uses TD error to calculate loss of the neural network, which is based on just the one next reward. [13] To get a trade-off between TD and Monte Carlo methods, which update each state basing on the entire sequence of rewards until the end of the episode, we also tried to implement multi-step DQN and compared its performance with one-step DQN benchmark. We were using a n-step buffer similar to the replay buffer in one-step experience replay, but the experiment got rather poor results. The reason might be that trajectories introduced more sampling bias than single transitions.

6.5 Dueling DQN

Another improvement of DQN is Dueling DQN. Dueling DQN only presents a change in the network architecture compared with DQN. Action values in Dueling DQN are calculated with the outputs of two estimators, state values and state-dependent action advantages. Using two streams instead of a single state-action value stream allows us to update state-action values actions for not chosen, which would result in faster learning.[15] In the snake game, it also happens that in some states, the values of the different actions are very similar, and it is less important which action to take, so it is possible that Dueling DQN could perform better. But considering that our action space only contains only three elements, the advantage might not be very apparent. We implemented Dueling DQN using the same encoding convolutional layers as our original DQN, and experiments showed that our Dueling DQN agent just gained similar rewards as DQN.

7. EXPERIMENTS

7.1 Benchmark

Firstly we built our DQN network and agent basing on the github framework states at the beginning. The framework has implemented experience replay, set up the training environment, and built their DQN model with Tensorflow. We made use of their experience replay and training environment, built our own models using Pytorch, and also modified the experience replay part when experimenting on other models. With the performance of our DQN model trained on a 10×10 play ground as a benchmark, we trained possible improvements of DQN in the same environment and compared them with the benchmark.

Table 1. Network Architecture

	input	kernel	stride	output
conv1	frames=4	3×3	1×1	16
ReLU				
conv2	16	3×3	1×1	32
ReLU				
fc3	32*flatten	1×1	1×1	256
ReLU				
fc4	256	1×1	1×1	actions=3

7.2 Evaluation Standard

As the initialization of the snake environment is subject to stochasticity, we computed the average rewards of 100 independent trials and used the average reward to evaluate the performance of our trained agents.

7.3 DQN Network Architecture and Hyper-parameters

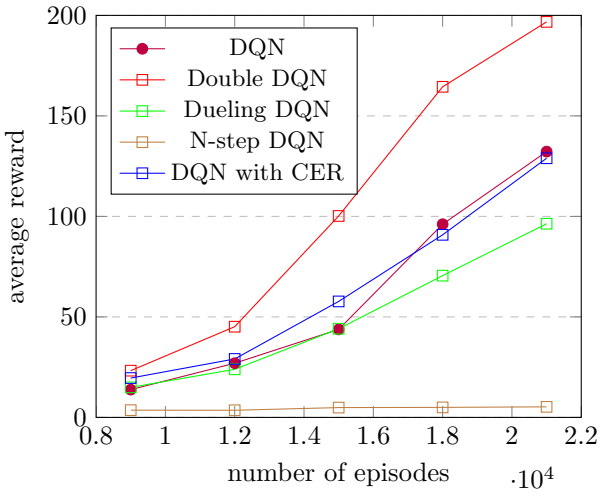
We tested DQN with different numbers of convolutional layers, fully-connected layers and different kinds of activation functions, and chose the best combination as shown in Table 1. We also found that RMSprop optimizer with a learning rate of 0.001 and a discount factor of 0.95 were proper for training, which we kept using in the following experiments. Furthermore, Our DQN started with an exploration rate of 1, corresponding to acting uniformly at random; it annealed the amount of exploration with a given decay until meeting the specified minimum value. In our experiments, we were using a exploration decay of 0.01 and a minimum rate of 0.1.

7.4 Model Comparison

We trained and tested our 5 different models on a 10×10 play ground without any obstacles. As shown in Graph 1, compared with our benchmark DQN, Double DQN was an evidently superior method. That was because using two independent DQN networks, Double DQN could effectively eliminate positive bias in sample data and thus won a gain in rewards.

Although with two separate streams trained, Dueling DQN agent was supposed to act better than the benchmark agent, in our experiments, the Dueling DQN model just performed similarly to the benchmark DQN model. That was probably because our action space, which only has three actions to select, was too small for the dueling neural network to show its strengths.

Graph 1. Comparison of Different Models



Originally, our DQN was using experience replay without a limitation of replay buffer size. That was a relatively preferred setting in our game environment, as it turned out that snake agents with smaller buffers did not save too much training time, yet agents with buffer sizes as small as 1000 to 10000 were hardly learning at all. We think that the fail of learning of small replay buffer was caused by the inadequate experiences for training.

The paper of Zhang, Shangtong et al.[14] mentioned that DQN could not learning robustly with the change of replay buffer size, which motivated us to try the DQN with CER model. However, in our game, with unlimited buffer size, Combined Experience Replay did not show much advantage over the benchmark. Probably that was because that our maximum number of training experience of 21000 was not large enough for CER to show its strengths.

For N-step DQN, we tested with a step number of 2 and 3 separately. Their results were quite similar, so in Graph 1 we only presented the results of 3-step DQN. We see that N-step DQN was learning a little

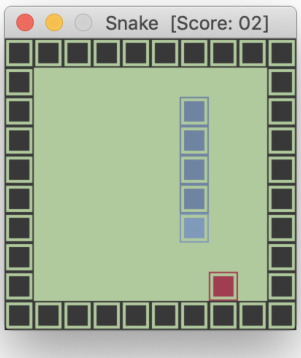


Figure 1. 10×10 blank playground

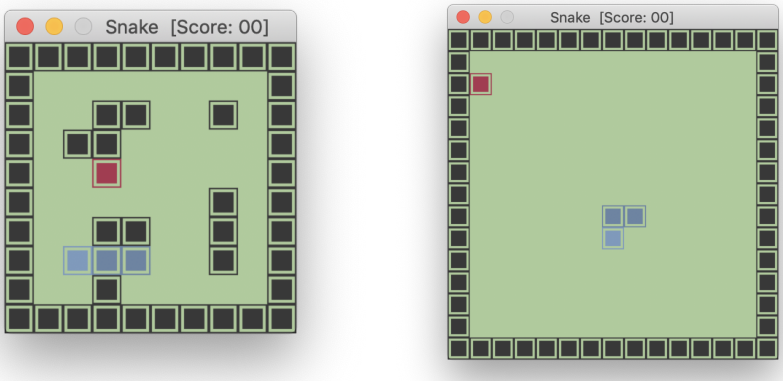


Figure 2. Other Environments: 10×10 with obstacles and 15×15 blank playground

but did not converge to an optimal policy. That might be caused by the extra bias introduced by longer trajectories.

7.5 Other Environments

After testing our models in 10×10 blank playground, we also tried other environment settings, including a larger playground of 15×15 and a 10×10 playground with obstacles. The relative performance of different models in those experiments was generally the same as 10×10 blank.

8. CONCLUSION

In this project, we implemented DQN agent to play the Snake game and compared its results with several extensions of the standard DQN model, including Double DQN, Dueling DQN, n-step DQN and DQN with Combined Experience Replay.

Double DQN apparently showed its advantage over the original benchmark DQN, as with two independent networks randomly selected to update, it efficiently eliminated the positive sample bias of training experiences. Dueling DQN and DQN with Combined Experience Replay theoretically should have their strengths over DQN, but just showed similar performance as the benchmark in our experiments. Dueling DQN was not presenting its superiority probably because we do not have a large action space, as its advantage is its ability to update actions not chosen at each training step. Combined Experience Replay did learn better than buffer experience replay possibly because that our buffer size did not reach a threshold value to to harm experience replay's performance. Finally, N-step DQN performed worst. We think that was a result of new

biases brought by longer sample trajectories.

REFERENCES

[1] Ma, B., Tang, M., and Zhang, J., "Exploration of reinforcement learning to snake," (2016).

[2] Tesauro, G., "Temporal difference learning and td-gammon," *Communications of the ACM* **38**(3), 58–68 (1995).

[3] Abbeel, P. and Ng, A. Y., "Apprenticeship learning via inverse reinforcement learning," in [*Proceedings of the twenty-first international conference on Machine learning*], 1, ACM (2004).

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

[5] Diallo, E. A. O., Sugiyama, A., and Sugawara, T., "Learning to coordinate with deep reinforcement learning in doubles pong game," in [*2017 16th IEEE International Conference on Machine Learning and Applications (ICMLA)*], 14–19, IEEE (2017).

[6] Yoon, S. and Kim, K.-J., "Deep q networks for visual fighting game ai," in [*2017 IEEE Conference on Computational Intelligence and Games (CIG)*], 306–308, IEEE (2017).

[7] Wei, Z., Wang, D., Zhang, M., Tan, A.-H., Miao, C., and Zhou, Y., "Autonomous agents in snake game via deep reinforcement learning," in [*2018 IEEE International Conference on Agents (ICA)*], 20–25, IEEE (2018).

[8] Hausknecht, M. and Stone, P., "Deep recurrent q-learning for partially observable mdps," in [*2015 AAAI Fall Symposium Series*], (2015).

[9] Lample, G. and Chaplot, D. S., "Playing fps games with deep reinforcement learning," in [*Thirty-First AAAI Conference on Artificial Intelligence*], (2017).

[10] Van Hasselt, H., Guez, A., and Silver, D., "Deep reinforcement learning with double q-learning," in [*Thirtieth AAAI conference on artificial intelligence*], (2016).

[11] Yeh, J.-F., Su, P.-H., Huang, S.-H., and Chiang, T.-C., "Snake game ai: Movement rating functions and evolutionary algorithm-based optimization," in [*2016 Conference on Technologies and Applications of Artificial Intelligence (TAAI)*], 256–261, IEEE (2016).

[12] Crevier, F., Dubois, S., and Levy, S., "Apprenticeship learning via inverse reinforcement learning," in [*Multiplayer snake AI*], 1, cs221 project final report (2016).

[13] Sutton, R. S. and Barto, A. G., "Reinforcement learning: An introduction," (2011).

[14] Zhang, S. and Sutton, R. S., "A deeper look at experience replay," (2017).

[15] Wang, Z., Schaul, T., Hessel, M., Van Hasselt, H., Lanctot, M., and De Freitas, N., "Dueling network architectures for deep reinforcement learning," *arXiv preprint arXiv:1511.06581* (2015).