Exploration of Reinforcement Learning to SNAKE

Bowei Ma, Meng Tang, Jun Zhang

***Abstract*— In this project, we explored the application of reinforcement learning in the problem not amenable to closed form analysis. By combining convolutional neural network and reinforcement learning, an agent of game Snake is trained to play the revised Snake game. The challenge is that the size of state space is extremely huge due to the fact that position of the snake affects the training results directly while its changing all the time. By training the agent in a reduced state space, we showed the comparisons among different reinforcement learning algorithms and approximation optimal solution, and analyzed the difference between two major reinforcement learning method.**

I. INTRODUCTION

Since video games are challenging while easy to formalize, it has been a popular area of artificial intelligence research for a long time. For decades, game developer have attempted to create the agent with simulate intelligence, specifically, to build the AI player who can learn the game based on its gaming experience, rather than merely following one fixed strategy. The dynamic programming could solve the problem with relative small number of states and simple underlying random structure, but not the complex one.

The Reinforcement learning is one of the most intriguing technology in Machine Learning, which learns the optimal action in any state by trial and error, and it could be useful in the problem not amenable to closed form analysis. Therefore, we selected and modified the Snake game to investigate the performance of reinforcement learning. The goal of this project is to train an AI agent to perform well in a revised Snake game.

Snake is the game popularizing over the world with Nokia mobile phones. It is played by sole player who controls moving direction of a snake and tries to eat items by running into them with the head of the snake, while the foods would emerge in random place of bounded board. Each eaten item makes the snake longer, so maneuvering is progressively more difficult. In our game, we slightly changed the games rule into scoring as many points as possible in fixed time, instead of counting points in unlimited period.

Q-Learning is an example of a reinforcement learning technique used to train an agent to develop an optimal strategy for solving a task, in which an agent tries to learn the optimal policy from its history of interaction with the environment, and we call the agent's knowledge base as "Q-Factor". However, it is not feasible to store every Q-factor separately, when the game need a large number of action state pairs, so we could introduce Neural Network to store Q-factor of each state.

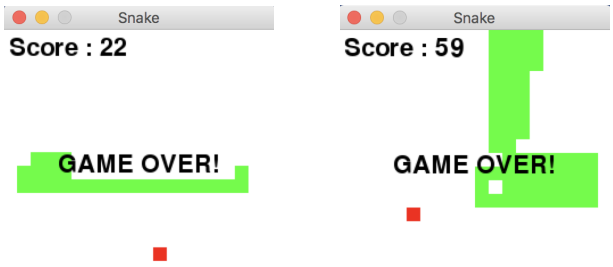

Fig. 1: Game Snake

The SARSA algorithm is an On-Policy algorithm for Temporal Difference learning. Compared to Q-learning, the major difference of SARSA is that the maximum reward for the next state is not necessarily used for updating the Q-values, instead, a new action, and therefore reward, is selected by using the same policy that determined the original action.

II. WORK REVIEW

There are abundant works about the artificial intelligence research for game agent. In [1], Miikkulainen showed that soft computational intelligence (CI) techniques such as neural network, have been excelling in some challenging field, where the relatively standard, labor-intensive scripting and authoring methods failed. At the same time, recent research focuses not only on games that can be described in a compact form using symbolic representations, such as board and card games, but on more complex video games.

TD-gammon, a backgammon-playing program developed in 1990s, is one of the most successful example in reinforcement learning area. In [2], TD-gammon used a model-free reinforcement learning algorithm similar to Q-learning, and approximated the value function using a multi-layer perceptron with one hidden layer1.

Tsitsiklis and Van Roy [3] showed that combining model-free reinforcement learning algorithms such as Q-learning with non-linear function approximators, or indeed with off-policy learning could cause the Q-network to diverge. Subsequently, the majority of work in reinforcement learning focused on linear function approximators with better convergence guarantees.

With the revival of interest in combining deep learning with reinforcement learning, Sallans and Hinton [4] illustrated that deep neural networks could be used to estimate the environment, while restricted Boltzmann machines could benefit the estimation of the value function.

III. METHOD

A. Mathematical Analysis of Snake

- Abstractly, each step of the Snake game could be considered as finding a self-avoiding walk (SAW) of a lattice in $\mathbb{R}^2$. A realization of Snake game can be mathematically represented as
  - The $m \times n$ game board is represented as $G = (V, E)$, where $V = \{v_i\}$ is the set of vertices, where each vertex corresponding to a square in the board, and $E = \{e_{ij} | v_i \text{ is adjacent to } v_j\}$.
  - The snake is represented as a path $\{u_1, \cdots, u_k\}$, where u_1, u_k are the head and tail, respectively.
- **NP-hardness**: Viglietta(2013) [6] proved that any game involving collectible item, location traversal and one-way path is NP-hard. Snake is a game involving traversing a one-way path, i.e., the snake cannot cross itself. Thus, if we have no prior information about the sequence of food appearance, picking every single shortest SAW for each step in an episode is NP-hard.

B. Approximation Algorithm for Snake

Due to the NP-hardness of finding optimal solution for Snake, we developed a heuristic algorithm that does considerably well for Snake problem and used it as benchmark for the reinforcement learning algorithms we will discuss later. The algorithm is shown in $Algorithm1$.

Algorithm 1: The deterministic heuristic algorithm for Snake

```
Build the graph G = (V, E), each square v_ij ∈ V
while Snake S= {s_1, ··· , s_k} ∈ V and Food F= {f} ∈ V do
    build {v_ij, v_kl} ∈ E, where v_ij, v_kl ∉ S with (i = ±k, j = l) or (i = k, j = ±l)
    initialize m × n array D
    assign D[i][j] = length of path between v_ij and f or MAX_NUM if no path exists
    find the min{D[i][j]} with {v_ij, s_1} ∈ E
    if D[i][j] ≠ MAX_NUM then
        find Path P = {v_ij, p_2, ··· , p_{n-1}, f}
        assign s̄_1 = f, s̄_2 = p_{n-2}, ··· s̄_k = p_{n-k}
        if s̄_1 and s̄_k is connected then
            assign s_1 = v_ij, s_2 = s_1, ···
            continue to step 2
        end
        else
            Find the longest Path between s_1 and s_k,
            P = {s_1, p_2, p_3 ··· , p_{n-1}, s_k}
            assign s_1 = p_2, s_2 = s_1, ···
            continue to step 2
        end
    end
end
```

C. Reduction of State Space for Reinforcement Learning

To conduct and implement successful reinforcement learning algorithms to play the game, one of the fundamental obstacles is the enormous size of state space. For instance, denote the size of the game board as $n \times n$. Naively, using the exact position of the snake and food, each cell could be parameterized by one of the four conditions: {contains food, contains the head of the snake, contains the body of the snake, blank}. By simple counting principle the size of the state space $|S|$ satisfies $|S| > n^8$, which is immense when n is large and learning in state space of such size is infeasible. Hence, to accelerate the learning rate, one simple reduction technique is to record only the relative position of the snake and the food with the current condition of the snake. Precisely, each state in the reduced space is of the form

$$\{w_s, w_l, w_r, q_f, q_t\}$$

In the above expression, w_s, w_l and w_r are indicator functions whether there is a wall adjacent to the head in straight, left and right directions respectively, and q_f, q_t are the relative position of the food and tail with respect to the head. Using such mapping the total size of state space is only $2^3 \cdot 4^2 = 128$, and the performance is would be improved prominently in the learning process.

D. Reinforcement Learning - Q-Learning

In Q-learning, an agent tries to learn the optimal policy from its history of interaction with the environment. A history is defined as a sequence of state-action-rewards:

$$< s_0, a_0, r_1, s_1, a_1, r_2, s_2, a_2, r_3, \cdots > \quad (1)$$

For the Snake game, the process is indeed a Markov Decision Process, and the agent only needs to remember the last state information. We define the experience as a tuple of $< s, a, r, s_{next} >$. These experiences will be the data from which the agent can learn what to do. As in decision-theoretic planning, the aim is for the agent to maximize the value of the total payoff $Q(s, a)$, which in our case is the discounted reward.

In Q-learning, which is off-policy, we use the Bellman equation as an iterative update

$$Q_{i+1}(s, a) = E_{s' \sim \epsilon}\{r + \gamma \max_{a'} Q_i(s', a' | s, a)\} \quad (2)$$

In the above equation, s, s' are the current and next state, r is the reward, γ is the discount factor and ϵ is the environment. And it could be proven that the iterative update will converge to the optimal Q-function. Since the distribution and transition probability is unknown to the agent, in our approach we use a neural network to approximate the value of the

Q-function. This is done by using the temporal difference formula to update each iteration's estimate as

$$Q(s,a) \leftarrow Q(s,a) + \alpha(r + \max_{a'} Q(s',a') - Q(s,a)) \quad (3)$$

The action set includes three possible outcomes :{turn left, turn right, go straight}.

An ϵ greedy approach is implemented here. The exploration probability is ϵ is changed from 0.5 to 0.1 with a constant density 0.01 during training. Once it reaches 0.1, it holds constant. This propels the agent to explore a lot of possibilities in the beginning of the game when it doesnt know how to play the game. This leads to a great amount of random actions thus enable the agent to exploit much enough to narrow down the optimal actions.

E. Reinforcement Learning - State-Action-Reward-State-Action(SARSA)

State-Action-Reward-State-Action(SARSA) is an on-policy reinforcement learning algorithm which estimates the value of the policy being followed. We could describe an experience in SARSA in the form of:

$$< s, a, r, s', a' > \quad (4)$$

This means that the agent was in state s, did action a, received reward r, and ended up in state s', from which is decided to do action a'. This provides a new experience to update $Q(s,a)$ and the new value which this experience provides is $r + \gamma Q(s',a')$. So SARSA could be described as below:

$$Q(s_t,a_t) \leftarrow Q(s,a) + \alpha(r' + \gamma Q(s',a') - Q(s,a)) \quad (5)$$

So SARSA agent will interact with the environment and update the policy based on the actions taken, and that's why it's an on-policy learning algorithm. Q value for a state-action is updated by an error, adjusted by the learning rate α. Q values represent the possible reward received in the next time step for taking action a in state s, plus the discounted future reward received from the next state-action observation. The algorithm of SARSA goes in $Algorithm3$

Algorithm 2: The algorithm for SARSA

```
Start the game snake
while do
    Initialize Q(s, a) arbitrarily
    Choose a from s using ε-greedy policy
    while do
        Take action a
        Choose a from s using ε-greedy policy
        Q(s_t, a_t) ←
        Q(s, a) + α(r' + γQ(s', a') − Q(s, a))
        s ← s
        a ← a
    end
    Update game parameter
end
```

IV. RESULTS

A. Tuning Parameters

The discount factor γ was set to be 0.95, moderately decreasing learning rate starting from $\alpha = 0.1$ and the rewards were set as shown in Table 1.

We want the agent to control the snake to go to the food quickly, and the last column in table 1 is a punishment for taking for one movement, which encourages the agent to traverse shorter walk to the food. This negative rewards acts as similar function as the discount factor γ. We performed trial and error on different combinations of reward for different cases to get current combination of reward values as one optimal combination among all the trials.

TABLE I: Rewards Table

case	eat food	hit wall	hit snake	else
reward	+500	-100	-100	-10

B. Learning Curve

The learning curves for Q-learning (fig. 2.) and SARSA (fig. 3.) are shown below respectively. The red dash lines represent the average learning curves.

We could easily observe that the performance of agent with Q-learning get improved faster than that of agent with SARSA in the beginning, i.e. in a short run, agent with Q-learning algorithm outperforms the agent with SARSA. But as the number of training iterations increases, the performance of agent with Q-learning doesn't improve much, while the performance of agent with SARSA still gets improved comparatively significantly. Q-learning does well(compared to SARSA), when the training period is short. But in the long period, SARSA wins.

The reason why the agent of Q-learning doesn't perform well in some cases in the long period training is that Q-learning algorithm would reinforce its self-righteous Q-value even with a very small learning rate. This would lead to considerably volatile performance. Although the agent with SARSA seems to outperform the agent with Q-learning algorithm in a long-period training, it also has a comparatively sluggish learning curve in our cases.

C. Performance Comparison

We took the performance of approximated optimal solution algorithm as the benchmark. As we mentioned before, due to the fact that Snake is a NP-hard problem, the best benchmark we could get here is the approximated optimal solution. And it could be shown later that our agents with reinforcement learning algorithms could not beat this approximated solution even with a considerable long training period.

Then we compared the performance of the agents based on two reinforcement learning algorithms with the benchmark - the performance of our approximated optimal solution. At the

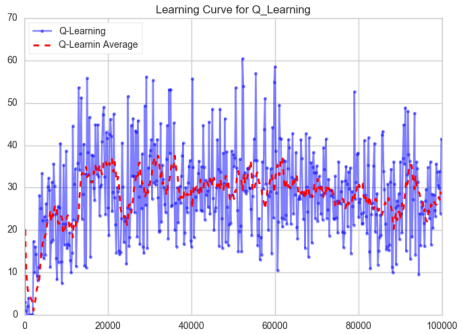

Fig. 2: Learning curves from Q-Learning

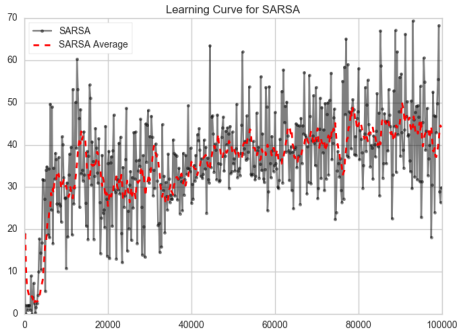

Fig. 3: Learning curves from SARSA

same time, we consider the effects of training period on the performances of SNAKE agents. We performed 10^4,10^5,10^6 training iterations on agents based on different reinforcement learning algorithms and evaluate their performances respectively.

We set the time limit to be 1 minute for the game, run 1000 tests for agents with different algorithms and compute the average score different agents could achieve. The reason why we choose 1 minute to be the time limit is that within 1 minute, the agents with those reinforcement learning algorithms could control the snake survive so we could make sure that the difference lies only on whether different agents could find shorter path to the food. 1 minute is a comparatively long time period to show the significant performance difference among different algorithms while it is a relatively short time period that 1000 tests could be handled with our PC in a reasonable running time.

The results are shown in figure 3 and Table 2.

TABLE II: Performance Comparison

iterations \ method	Optimal	SARSA	Q-learning
10^4	77.504	36.858	18.023
10^5	77.504	51.994	25.789
10^6	77.504	61.830	36.567

It's worth mentioning here that the performance of our approximated optimal solution algorithm is not related to the number of training iterations. Only the performances of agents with SARSA and Q-learning algorithm are directly

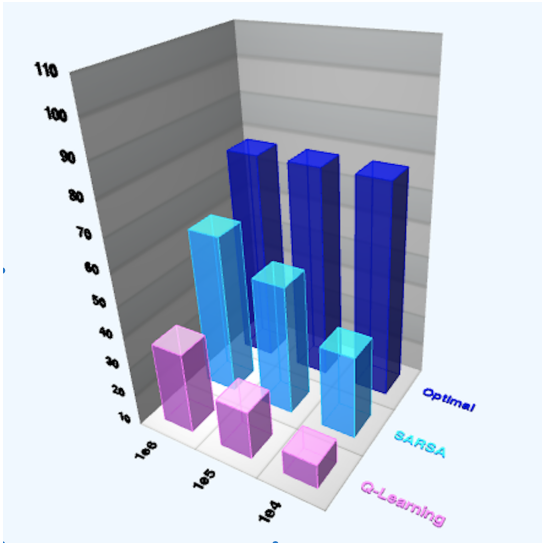

Fig. 4: performance comparison among 3 different algorithms

related to the number of training iterations. The performance of the approximated optimal solution is shown to be a benchmark in the same table and figure as performances of agents with different reinforcement learning algorithms.

Our results show that within the range of 10^4 to 10^6 training iterations, a larger number of training iterations would lead to better average scores for both Q-learning and SARSA algorithm. Given the same number of training iterations, agent trained in SARSA has better performance than that trained in Q-learning.

V. FUTURE WORK

A. Explore the stability of Q-learning algorithm

We found that in some training cases, even with a decreasing exploration probability, the performance of Q-learning algorithm is not very stable. So it's worth exploring principles and methods to improve the stability of the Q-learning algorithm. For example, one intuitive way to address this problem is to add various tuning parameters to improve the probability of convergence for Q-learning algorithm.

B. Study the other state space approximation methods

Other approximation of the state space could be explored for better performance. Currently, we use a quadrant view state mapping technique as discussed in the previous section. Using such means, the snake does not have a good sense of precaution for hitting himself. Hence, a more rigorous state mapping technique should be developed. Such reduction mapping must not only approximate the relative position of the head and the food, but also obtain a concise sense of the position of the body without tremendously enlarging the size of the state space.

C. Expected SARSA

In order to further improve the learning rate of the Snake agent, Expected SARSA could be used. van Seijen et al.

(2009) [7] provide a theoretical and empirical analysis of Expected SARSA, and found it to have significant advantages over more commonly used methods like SARSA and Q-learning.

VI. CONCLUSIONS

In this project, we have shown an implementation of both Q-learning and SARSA, by approximation of state space, in neural network. We anticipated that the difference between performances of Q-Learning and SARSA might become apparent after long training period, and the outcome verified our expectation. Also, we compared this two methods with an approximated optimal solution and found that neither of the two agents could achieve the performance of the approximated optimal solution, while they exhibited prominent learning. Also, We observed the instability of Q-learning algorithm in some cases and it's worth exploring feasible solutions for future work.

REFERENCES

[1] Risto Miikkulainen, Bobby Bryant, Ryan Cornelius, Igor Karpov, Kenneth Stanley, and Chern Han Yong. *Computational Intelligence in Games.*
URL:ftp://ftp.cs.utexas.edu/pub/neural-nets/papers/miikkulainen.wcci06.pdf

[2] Gerald Tesauro. *Temporal difference learning and td-gammon.* Communications of the ACM, 38(3):5868, 1995.

[3] John N Tsitsiklis and Benjamin Van Roy. *analysis of temporal-difference learning with function approximation.* Automatic Control, IEEE Transactions on, 42(5):674690, 1997.

[4] Brian Sallans and Geoffrey E. Hinton. *Reinforcement learning with factored states and actions.* Journal of Machine Learning Research, 5:10631088, 2004.

[5] Lucas Jen *An application of SARSA temporal difference learning to Super Mario*
URL: http://x3ro.de/downloads/MarioSarsa.pdf

[6] Giovanni Viglietta. 2013. *Gaming is a hard job, but someone has to do it!*

[7] van Seijen, H., van Hasselt, H., Whiteson, S. and Wiering, M. (2009). *A theoretical and empirical analysis of Expected Sarsa, 2009 IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning pp. 177184.*
URL: http://goo.gl/Oo1lu