REINFORCEMENT LEARNING ON ATARI GAMES

Kevin Leonardo, Li Jiayao
$\{k.l.a, jiayao.li\}$@u.nus.edu
School of Computing
National University of Singapore

ABSTRACT

The purpose of this project is to code reinforcement learning (RL) algorithms, compare the performance of them on playing Atari games, and draw insights on the characteristics of each different algorithm. The three RL algorithms we have selected to study in this project are SARSA, naive Q-learning, and monte-carlo tree search(MCTS) with upper confidence bound (UCB). The Atari game environments are available on Open AI Gym. We are focusing on the following three environments: Taxi-v2 (deterministic environment), Centipede-ram-v0 (stochastic, non-markovian environment), and BreakoutDeterministic-v4 (deterministc, non-markovian environment). Finally, we would like to compare the performance of naive Q-learning with deep Q network implemented by *Deepmind* to shed light on how neural network and *experience reply* helps in non-markovian game environments.

***Index Terms*—** reinforcement learning, SARSA, Q-learning, MCT with UCB, deep Q network

1. INTRODUCTION

In this paper, we consider the performance of several RL algorithm on different Atari games, and analyze them in order to identify factors which might influence the performance of these algorithms on different types of environments. Our motivation is to draw further insights into the characteristics of different RL algorithms, so that we can identify which algorithms may be better suited for certain environments. RL can be considered as learning which actions to take in a particular situation, which we call *state*, in order to maximize a accumulated reward, which we call *utility* [1]. The game player, which we call *agent* is not told what to do, but to learn the *transition function, T* and *reward function, R* of the game environment by taking actions thus interacting with the environment. In this project, we will first present three different popular reinforcement learning algorithms, namely, SARSA, naive Q-learning, and MCT with UCB, and their performance on different environments, then we will compare these three algorithms and analyse their performance with respect to different game environment. Finally, we will show the performance of deep Q network playing a typical Atari game – BreakoutDeterministic-v4, and through its performance, we will see how its unique implementation guides the agent to search the game playing strategy.

Originally, our goal was to try and maximize the score in the game Breakout-ram-v0 using Q-Learning and Deep Q-Learning. While this would allow us to dive deeper into a particular environment, However, due to time constraints and lack of manpower, we decided to instead do an analysis and comparison on various RL algorithms and how they perform on different Atari games. This allowed us to perform several experiments using different algorithms on different games simultaneously, thus saving time. We also opted to adopt an existing Deep Q implementation for similar reasons.

2. ENVIRONMENTS

We explore three different environments :

- Taxi-v2 as a deterministic environment with limited observation space.
- Centipede-ram-v0 as a stochastic, non-markovian environment with a relatively big observation space.
- BreakoutDeterministic-v4 as a deterministc, non-markovian environment with big observation space.

2.1. Taxi-v2

Taxi-v2 is a deterministic, single-agent game where the agent's job is simply to pick up the passenger at one location and drop him off in another. Agent receive +20 points for a successful dropoff, and lose 1 point for every timestep it takes. There is also a 10 point penalty for illegal pick-up and drop-off actions. [2] There is no randomness that is out of agent's control, as there are no moving parts in this game other than the agent.

The input the agent receive consist of a single integer which represent one of the 500 possible state of the environment. The number of possible actions are 6, one for each direction of movement and two for picking up and dropping off passengers. This is the least complex environment that we have tried, and is used to measure the performance of algorithms on a simple environment.

2.2. Centipede-ram-v0

In this game, to maximize the utility, the agent's goal is to shoot the centipede for a small reward and the spider to gain a bigger reward, while avoiding being hit by them for negative reward. The centipede moves in a predictable downwards fashion at first. However, the spider moves and appear in a random fashion, and additional, shorter centipedes may appear randomly from the agent's sides. Due to this randomness, it is a stochastic environment whereby In addition, the agent is only able to shoot upwards, and once a centipede reaches the bottom of the screen, it will begin to move upwards until a certain point, before moving downwards once more. Once the agent is hit 3 times, the game is reset.

Index	Action
0	NOOP
1	FIRE
2	UP
3	RIGHT
4	LEFT
5	DOWN
6	UPRIGHT
7	UPLEFT
8	DOWNRIGHT
9	DOWNLEFT
10	UPFIRE
11	RIGHTFIRE
12	LEFTFIRE
13	DOWNFIRE
14	UPRIGHTFIRE
15	UPLEFTFIRE
16	DOWNRIGHTFIRE
17	DOWNLEFTFIRE

Table 1. Action space in Centipede-ram-v0

2.3. BreakoutDeterministic-v4

Breakout is a deterministic, single-agent, non-markovian environment where the agent has to block the path of a ball as it's coming towards them and bounce it back to hit blocks above, whereby it will bounce again in some direction based on its previous trajectory. The agent gains -1 reward if the ball passes them and +1 whenever a block is hit. Once the ball passes the agent a set amount of time, the game is over and environment is reset.

The observation received by agent is the RGB image of the screen. This environment is non-markovian as, given a state (RGB image), the transition to the next state depends on the previous series of states, particularly to determine the direction in which the ball will transition.

3. REINFORCEMENT LEARNING ALGORITHM

3.1. Naive Q learning

Naive Q learning is an algorithm which learns the value of a state (s) - action (a) pair, which we can represent by $Q(s, a)$, and store it in a table, which we call Q table. Q learning algorithm learns $Q(s, a)$ by updating $\arg\max_a Q(s, a)$ for any state s the agent is in. $Q(s, a)$ is updated after each action the agent takes, and we assume there is a reward signal after each action. Therefore, the update equation for naive Q-learning is: $Q(s, a) \leftarrow Q(s, a) + \alpha(R(s) + \gamma \arg\max_a Q(s', a') - Q(s, a))$ [3]
The behaviour policy that our Q learning follows is $\epsilon - greedy$ which choose the greedy action with probability (1 - ϵ), and an action from the policy with probability ϵ. In our implementation, we have a high initial ϵ value, which is 0.99 to make sure that the agent explore enough actions so that good action strategies would not be missed. ϵ is annealed over 10000 steps, with a lower bound of 0.01. The discount rate $\gamma = 0.99$ and $\alpha = 0.6$.

We have applied naive Q learning algorithm to a stochastic game environment Centipede-ram-v0.The naive Q-learning result is shown in figure 1. As we can see from figure 1, naive Q-learning does not seem to improve over time on Centipede-ram-v0. We suspect the reason being that the underlying assumption for naive Q-learning is *Bellman equation* which follows *Markov assumption. Markov*

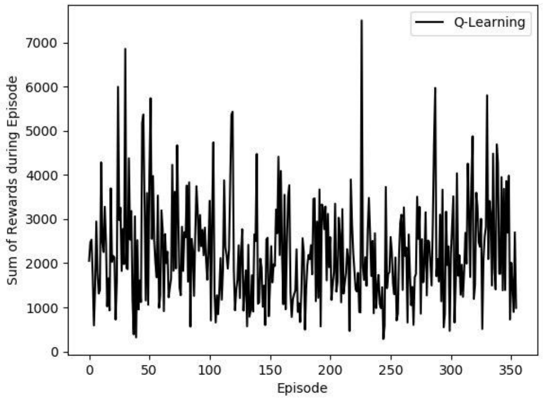


Fig. 1. Naive Q learning performance on Centipede-ram-v0

assumption states that state transition probability only depends on the current state s, not the history of earlier states. However, this assumption does not fit perfectly in the game environment of Centipede-ram-v0. To assert our hypothesis, we looked for a deterministic game environment where the history of earlier states does not determine the state transition, Taxi-v2. As we anticipated, naive Q learning improves the agent's game playing strategy. This can be shown in figure 2 taxi. As we can see from the figure, the agent does learn to maximize its utility value in general under naive Q-learning

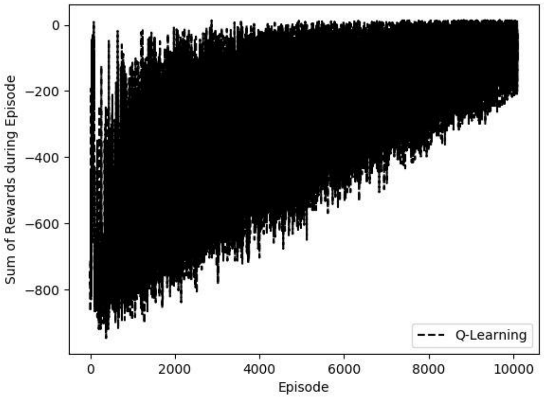


Fig. 2. Naive Q learning performance on Taxi-v2

3.2. SARSA

Similar to naive Q-learning, SARSA simulate *Bellman update* closely, but with a slightly different update equation, which is $Q(s, a) \leftarrow Q(s, a) + \alpha(R(s) + \gamma Q(s', a') - Q(s, a))$ [3]. With much similarity to naive Q-learning, the difference between these two is the removal of max operator in SARSA because SARSA is a *on-policy algorithm*. SARSA keeps updating its Q table which is used for action selection to interact with the environment it is in. However, naive Q-learning is *off-policy algorithm*, meaning the action selection algorithm is not necessarily updated, but Q table is updated with each action taken with respect to the update equation. As both algorithms, naive Q-learning, and SARSA are based on *Markov assumption*, we are expecting the similar results produced by SARSA playing Centipede-ram-v0 and Taxi-v2 as that of naive Q-learning. As we anticipated, performance on SARSA playing Centipede-ram-v0 can be found in figure 3, and performance on SARSA playing Taxi-v2 can be shown in 4. As we can see SARSA does not perform well on Centipede-ram-v0, but reasonably well on Taxi-v2. Similarly to naive Q-learning, we have a high initial ϵ value, which is 0.99 to make sure that the agent explore enough actions so that good action strategies would not be missed. ϵ is annealed over 10000 steps, with a lower bound of 0.01. The discount rate $\gamma = 0.99$ and $\alpha = 0.6$.

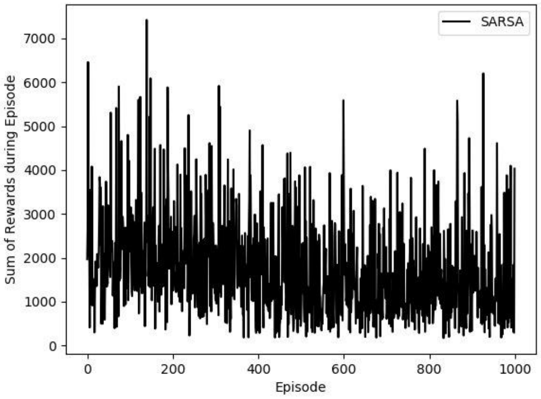


Fig. 3. SARSA performance on Centipede-ram-v0

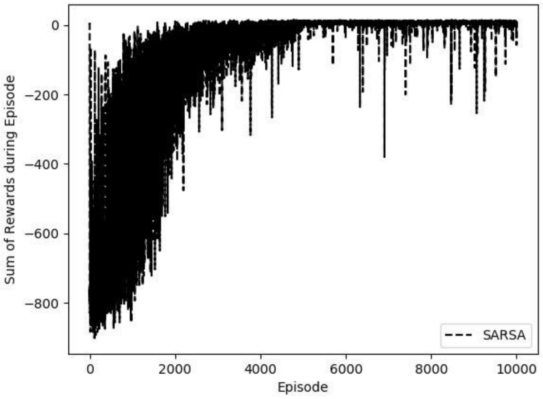


Fig. 4. SARSA performance on Taxi-v2

3.3. Monte-Carlo Tree Search

MCTS is a method for making optimal decisions in artificial intelligence problems, typically move planning in combinatorial games. It combines the generality of random simulation with the precision of tree search. MCTS can theoretically be applied to any domain that can be described in terms of state, action pairs and simulation used to forecast outcomes [4]. The basic algorithm of MCTS comprises of four different parts. There are explained as follows:

- Selection: starting at root node R, recursively select optimal child nodes (explained below) until a leaf node L is reached.

- Expansion: if L is a not a terminal node (i.e. it does not end the game) then create one or more child nodes and select one C.
- Simulation: run a simulated playout from C until a result is achieved.
- Backpropagation: update the current move sequence with the simulation result.

In our implementation, we make use of *upper confidence bound* in the selection of child node. In *upper confidence bound*, a child node n is chosen according to the policy π: $\pi_{UCT}(n) = \arg\max_a \overline{Q}(n,a) + c\sqrt{\frac{ln(N(n))}{N(n,a)}}$, where $N(n)$ is the number of times the node n has been visited, $N(n,a)$ is the number of trials through n with action a, and c is a constant. $\overline{Q}(n,a)$ is the average utility return from running simulations at node n with action a. From our experiment of playing Atari game, due to the simulation in MCTS, the program runs extremely slow. We have tried to use python parallel implementation to speed up the progress but due to the simulation in nature, the game plays still very slow. We would think that Q-learning and SARSA are more efficient than MCTS.

4. RESULTS & DISCUSSION

4.1. Results

Based on the performance of Naive Q (Figure 2) and SARSA (Figure 4) on Taxi-v2, we can see that while there index exists a trend where the sum of rewards over an episode increases as the algorithm is trained, and both seem to be converging to a similar policy, SARSA seems to converge much faster than Naive Q in this case. This difference in convergence rate can be explained by the effect of epsilon-greedy implementation, where SARSA converges faster as it takes the epsilon probability of random action into account when updating its policy.

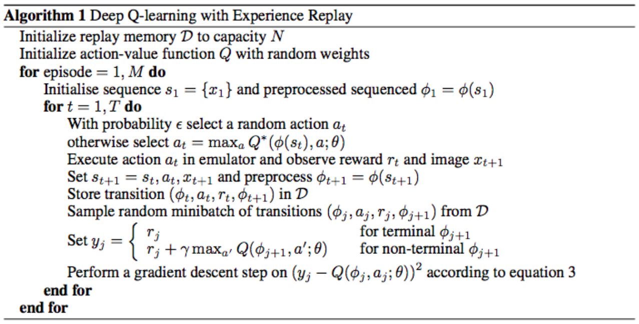
In contrast, performance in Centipede (Figure 1, Figure 3) does not seem to indicate any learning, let alone convergence. With either algorithm, there does not seem to be any difference between their initial performance and final performance. Our hypothesis is that the problem may lie in the non-markovian nature of this environment, as our implementation only takes into account the current state when deciding which action to take. With a non-markovian environment, this leads to a situation where value of an action given a state cannot be learned properly, as the value actually depends on the history of past states and not the current state. Another possibility is that the state space is too big for either algorithm to show any trends within the episodes we trained it for. This seems less likely as we've trained it for 1000 episodes. However, this possibility can be further explored simply by training for a longer amount of time.

Performance of Naive Q in Breakout (Figure **??**) seems in line with our hypothesis. Although Breakout is deterministic and single-agent, just like Taxi-v2, Naive Q does not show any improvement in performance after training. This means that the non-markovian property that Breakout has in common with Centipede is likely the reason why neither algorithm can converge for these games.

Improvement in performance by Deep Q-Learning (Figure 6 using Experience Replay further supports our hypothesis, as using Experience Replay allows it to use a history of states to determine action, solving the problem of non-markovian property.

5. RELATED WORK

From our research, many Atari games utilize *Deep Q Network*. *Deep Q Network* makes the training of Q-learning more stable and more data-efficient, when the Q value is approximated with a nonlinear function. Two major ingredients are *experience replay* and a separately updated target network to predicate the next state [5]. *Experience replay* is to make up the disadvantage of *Markov assumption* where the agent treats the current state independent from the history. in other words, the agent remebers the history through experience reply. *Deep Q learning* with *experience reply* algorithm used by *Deepmind* [6] is as figure 6.



```
Algorithm 1 Deep Q-learning with Experience Replay
  Initialize replay memory $\mathcal{D}$ to capacity $N$
  Initialize action-value function $Q$ with random weights
  for episode $= 1, M$ do
      Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
      for $t = 1, T$ do
          With probability $\epsilon$ select a random action $a_t$
          otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
          Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
          Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
          Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
          Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
          Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
          Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
      end for
  end for
```

Fig. 5. Deep Q-learning with Experience Replay

From *Deepmind* their result on playing BreakoutDeterministic-v4 presented as below, which shows a very good tendency of learning.

In comparison, naive Q-learning plays poorly on Breakout, which we have generated the result as shown in figure 7.

6. CONCLUSION

In conclusion, we have analyzed the performance of naive Q-Learning, SARSA, and MCTS on several games of differing types, and drawn several insights such as :

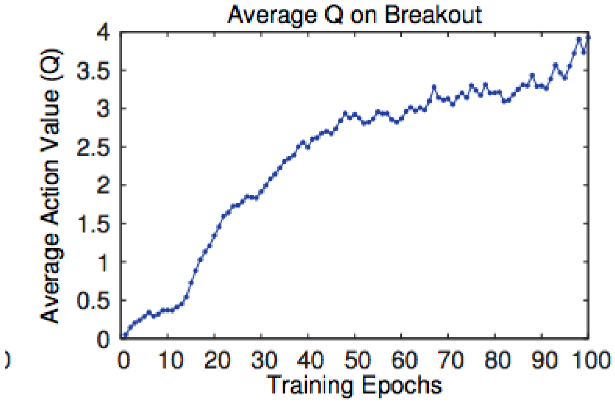


Fig. 6. Deep Q-learning with Experience Replay on BreakoutDeterministic-v4
[6]

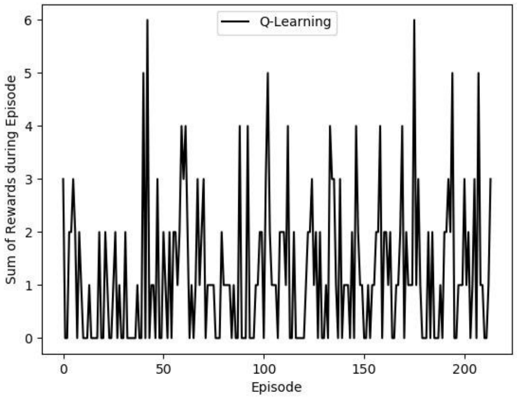


Fig. 7. Naive Q-learning on BreakoutDeterministic-v4

- Naive Q and SARSA converges on markovian environments, but not so on non-markovian environments.
- Deep Q-Learning using Experience Replay can solve the problem with non-markovian property.
- SARSA seems to converge faster when using epsilon-greedy.

In the future, we can extract more insights out of these algorithms by training them on different games and for a greater period of time. We might also be able to use further variants of RL algorithms to better understand their characteristics.

7. REFERENCES

[1] Andrew G.Barto Richard S.Sutton, *Reinforcement Learning An Introduction*, The MIT Press, 2018.

[2] “Taxi-v2,” 2011.

[3] Peter Norvig Stuart Russel, *Artificial Intelligence A Modern Approach*, Pearson, 2010.

[4] “Monte carlo tree search,” 2010.

[5] “Implementing deep reinforcement learning models with tensorflow + openai gym,” 2018.

[6] “Playing atari with deep reinforcement learning,” 2013.