ECE 517 - Reinforcement Learning - Project 2

Andrei Cozma [*1] **Hunter Price** [*1]

1. Introduction

In this project, we explore using two popular Reinforcement Learning algorithms, Q-Learning and SARSA, to play the game of Pong; To accomplish this, we use a provided Pong game simulator, a Python-based simulation using the PyGame library. The simulator offers a simple interface for interacting with the game and allows us to easily change the game state and observe the results.

We implement a Q-Learning and SARSA agent to play the game. This is accomplished by starting an episode within the simulator, which acts as the environment. Each step of the episode is a single frame of the game. The agent observes the state of the game, chooses an action, and gives the action to the simulator. The simulator then updates the game state and returns the reward to the agent. The agent then updates its action-value function and chooses the next action. This process continues until the episode ends, at which point the agent updates its action-value function one last time. The agent then starts a new episode and repeats the process. In order to cope with the large state space, we quantize the state space into a smaller number of states. We also use a variety of different hyper-parameters to test the performance of the agents.

We compare the performance of these algorithms and analyze their behavior in different learning scenarios through a series of experiments. We also discuss the effect of hyper-parameters, specifically alpha, on the algorithms' performance. Finally, we analyze the results and discuss the algorithms' performance through various metrics collected during the experiments.

2. Background

2.1. Pong Game

The Pong game environment is a two-player game in which players control paddles by moving them vertically across the left and right sides of the screen. The goal of each player is to prevent the ball from hitting their side of the screen.

The game environment is implemented in Python using the PyGame library. The game environment is shown in Figure 1.

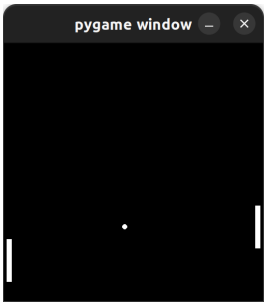

Figure 1. The Pong game environment.

As the game progresses, the ball's speed increases, making it more difficult for the players to keep the ball from hitting their side of the screen. The game ends when a player fails to prevent the ball from hitting their side of the screen.

3. Methodology

3.1. MDP Problem Formulation

The game environment provides the following information to the agent as the state of the environment:

- The Y-coordinate of the two players' paddles.
- The X and Y coordinates of the ball.
- The X and Y velocities of the ball.

The reward structure is as follows:

- The agent receives a reward of +100 if it wins the game.
- The agent receives a reward of -100 if it loses the game.
- The agent receives a reward of 1 if it hits the ball with its paddle.
- The agent receives a reward of 0 otherwise.

[*]Equal contribution [1]University of Tennessee, Knoxville, TN, USA. Correspondence to: Andrei Cozma <acozma@vols.utk.edu>, Hunter Price <hprice7@vols.utk.edu>.

The action space is a discrete set of actions, represented as integers 0-2 that the agent can take at each step in the game. The action space is as follows:

- The agent can move its paddle up [Action 0].
- The agent can move its paddle down [Action 1].
- The agent can do nothing [Action 2].

3.2. Code Design

Regarding the code structure employed, we have two main Python files called `Q_Learning.py` and `runEpisode.py`.

The other files are found in the `rlenv` and `rllib` folders. The `rlenv` folder contains the code for the game environment in `pongclass.py`, while the `rllib` folder contains `agent. py`, which contains the code for the Q-Learning and SARSA agents, as well as `utils.py`, which contains utility functions for parsing command line arguments.

3.3. Q_Learning.py

The `QLearning` class takes in the arguments provided by the user when running the script and initializes the game environment and the agent. We define the set of states we use out of the six possible states the environment provides at each step. Further, we keep track of histories for various metrics we later use to evaluate the performance of the agent. More specifically, we keep track of the Q-table, the number of visited states, and the rewards and wins.

The `getQState` function takes in the number of bins we want to use to discretize the state space and returns the discretized state. The representation of the discretized state is a tuple of integers, where each integer represents the bin the corresponding state variable falls into.

The `runEpisode` function takes in a boolean that indicates whether we want to learn or not and runs a single episode of the game. First, the function resets the environment and the agent, grabs the initial discretized state, and initializes variables to keep track of the total episode reward and whether the agent has won or not. The episode runs until the agent either wins or loses (i.e., the last reward is either -100 or 100). At each step, the agent takes an action based on the current state. If the learning flag is set to true, the agent takes an action based on the epsilon-greedy policy, where the probability of taking a random action is epsilon, and also updates the Q-table using the agent's update function. If the learning flag is set to false, the agent takes the greedy action but does not update the Q-table. Finally, the function returns the total episode reward and whether the agent has won or not.

The `checkpoint` function takes in the number of episodes to run and returns the average reward and number of wins over the specified number of episodes. This function is used in the runEpisodes function described below.

The `runEpisodes` function takes in the number of episodes to run, a boolean that indicates whether we want to learn or not, the number of episodes between checkpoints, and the path to save the Q-table at each checkpoint. We initialize variables to keep track of the rewards and wins over the course of the training, and run the specified number of episodes. At each episode, we run the runEpisode function and update the rewards and wins variables. During training, at each checkpoint we run the checkpoint function, save the Q-table, and update the histories. Finally, the function returns the rewards and wins over the course of the training.

The `train` function simply calls the runEpisodes function with the learning flag set to true, and also providing the number of episodes between checkpoints and the path to save the Q-table at each checkpoint.

The `test` function calls the runEpisodes function with the learning flag set to false. We also chose to reload the Q-table from the file at the beginning of each episode, such that we can run a separate script to visualize the agent's improvement while another script trains the agent in the background.

The minimum set of required parameters to run the script is as follows:

- `q` - The quantization factor for the state space.
- `a` - Alpha, The learning rate.
- `e` - Epsilon, The probability of taking a random action.
- `n` - The number of episodes to train the agent.
- `c` - The number of episodes between checkpoints.
- `f` - The file to save the results to.

An example of how to run the script is as follows:

```
python3 Q_Learning.py q a e n c f
```

Additional parameters can be provided to specify the algorithm to use (Q-learning or SARSA), as well as various other convenience parameters:

- `--alg` - Learning Algorithm (Q-LEARNING or SARSA) [Default: Q-LEARNING]
- `-d` or `--draw` - Draw the game to the screen. This is useful in combination with the `--test` flag, but rendering will slow down the training process. [Default: False]

- `-s` or `--speed` - Speed factor for frame rendering when `--draw` is enabled. [Default: 1.0]

- `-t` or `--test` - Test the model from the given file instead of training [Default: False]

3.4. runEpisode.py

The `runEpisode.py` script is used to run a single episode of the game by simply loading the action-value function from the specified file and running the game.

The script can be run with the following command:

```
python3 runEpisode.py q f
```

Similar to the previous script, the `q` parameter is the quantization factor for the state space, and the `f` parameter is the file containing the action-value function.

3.5. agent.py

The `agent.py` file contains the implementation of the Q-learning and SARSA agents.

The agent class takes in parameters for the number of states, number of bins (for discretization), number of actions, alpha parameter (learning rate), epsilon parameter (exploration rate), and gamma parameter (discount factor). The agent also takes in a parameter for the learning type, which can be either Q-learning or SARSA.

The constructor for the agent class initializes the Q-table with random values and keeps track of the number of times each state-action pair has been visited.

Note that even though the agent class supports changing the gamma parameter, we kept the value set to 1.0 for this project.

The `getAction` method returns the action to take at a given state. If the deterministic parameter is set to True, then the greedy action is returned. Otherwise, an epsilon-greedy action is returned.

The `update` method handles both the Q-learning and SARSA algorithms. The method takes the current state, action, reward, and next state. For Q-learning, the method uses the max action-value estimate at the next state, and for SARSA, the method uses the action-value estimate at the next state, using the next action.

There are also two helper methods for saving and loading the table and storing the action-value function in a file.

Table 1. Example Setup Parameters.

PARAMETER	VALUE
Q	8
A	0.01
E	0.1
N	20,000
C	500

4. Q-Learning Results

4.1. Example Setup

In this section, we present the results of our Q-Learning algorithm for the example setup. The example setup Parameters are shown in Table 1.

We used 20,000 episodes for training as we found this sufficient to achieve comparable results to the example setup in the writeup. We found that after training the model, the agent won 38/1000 games, and the average reward was -83.978. These results surpass the example setup in the writeup, where the agent won 20/1000 games, and the average reward was -86.

In Figure 2, we show the online (epsilon greedy) and offline (deterministic) average return as a function of the number of episodes trained. We can see that as training progresses, the average reward increases for both the online and offline methods. However, the offline method begins to divert from the online method after episode 7,500 and performs better on average. This makes sense as the offline method can use the optimal deterministic policy, whereas the epsilon-greedy policy limits the online approach (epsilon-soft policy). This is also reflected in the cumulative number of wins per episode, as shown in Figure 3. The offline method also begins to diverge from the online method after episode 7,500 and performs vastly better.

The total time taken to train is shown in Table 2. These results were gathered on a 12th Gen Intel® Core™ i9-12900K. In this setup, no checkpoints were used, and the training was done on a single thread. We can see that the training took just under 10 minutes (580 seconds) to complete 20,000 episodes. This is 0.029 seconds per episode. This training time is exceptionally fast and can be further optimized by limiting any file io and logging statements. It makes sense that the training time is so fast as the Q-table is relatively small (8,8,8,3), and the environment is relatively simple.

This brings to light the importance of software-in-the-loop simulation, as it allows for faster training times and more experimentation. On this simple problem, we can simulate the environment very quickly, but on more complex problems, this may not be the case. This may also be an issue when the environment is in the physical world, as it may not be

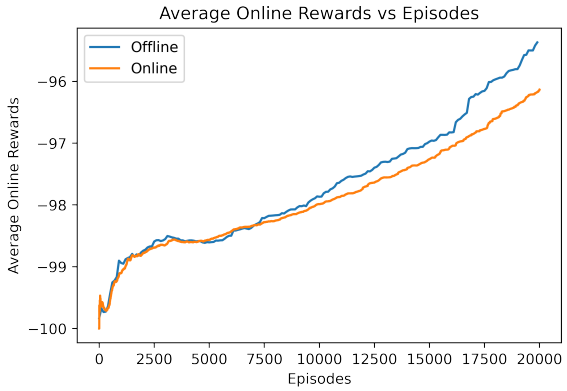

Figure 2. Example Setup average rewards - Online vs. Offline Learning.

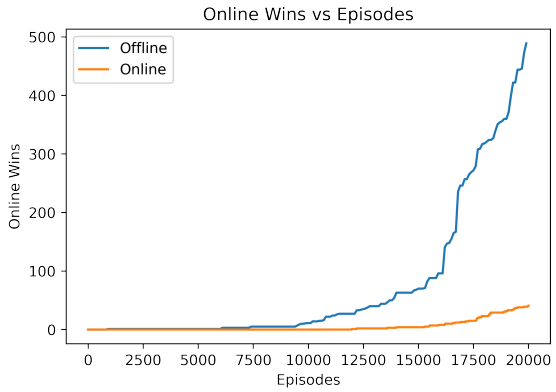

Figure 3. Example Setup cumulative wins - Online vs. Offline Learning.

possible to simulate the environment as quickly as we can in software. In this case, more efficient training methods may be required.

5. Other Experiments

5.1. Changing Alpha Parameter

In this section, we present the results of altering the alpha parameter. In these experiments, we use the same parameters as the example setup in 1, except we change the alpha parameter. The values used for alpha were: 0.1, 0.01, and 0.001. The larger the alpha value, the larger the update to the Q-table.

The average reward and cumulative wins are shown in Figure 4 and Figure 5, respectively. In both figures, we can see that the larger the alpha value, the faster the model improves. Conversely, the smaller the alpha value, the slower the model improves. This makes sense as the larger the alpha value, the larger the update to the Q-table, and the smaller the alpha value, the smaller the update to the Q-table. This can be a double-edged sword, as with a larger alpha value, the model will converge faster, but it may also overshoot the optimal solution. With a smaller alpha value, the model will converge slower, but it will be more likely to find the optimal solution.

Table 2. Training time.

# EPISODES	DURATION(S)
1	0.029044
20,000	580.888799

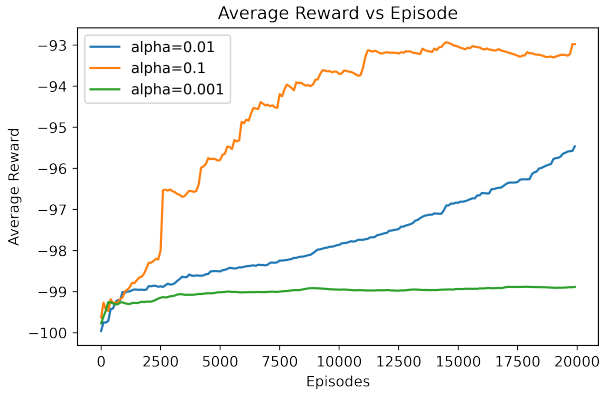

Figure 4. Alpha - Average Offline Rewards.

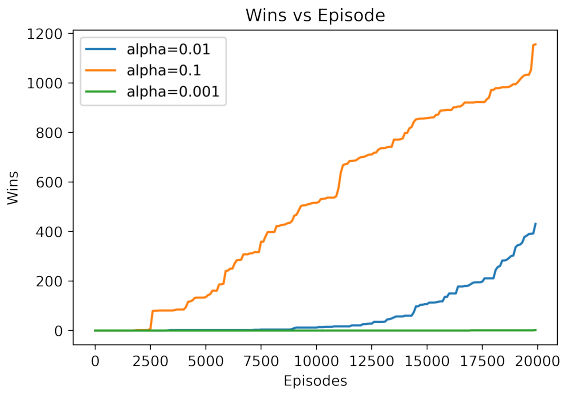

Figure 5. Alpha - Cumulative Offline Wins.

5.2. Q-Learning vs. SARSA

In this section, we present the results of comparing Q-Learning and SARSA. In these experiments, we use the same parameters as the example setup in 1, except we change the learning algorithm. The only difference between these two methods is that Q-Learning is an off-policy approach because it uses the max action from the next state, whereas SARSA is an on-policy approach because it uses an epsilon-greedy policy to select the next action.

The average offline reward and cumulative offline wins are shown in Figure 6 and Figure 7, respectively. In both figures, we can see that Q-Learning performs better than SARSA. This makes sense as Q-Learning was trained on the optimal policy, whereas SARSA was not. This means that Q-Learning is run offline it is able to take advantage of the optimal policy that was trained. This is not the case for SARSA, as it is not trained on the optimal policy. So, it is expected that Q-Learning would perform better than SARSA on offline performance.

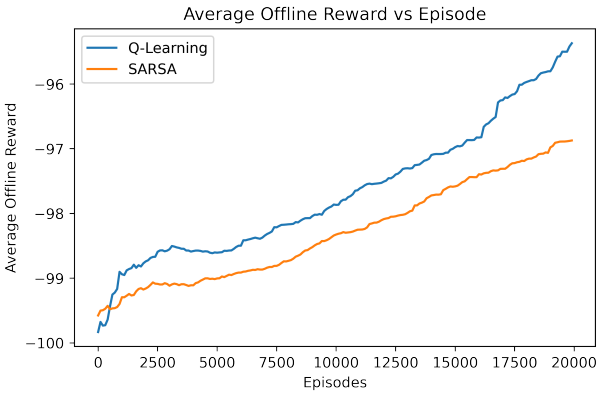

Figure 6. Q-learning vs. SARSA average offline reward.

The average online reward and cumulative online wins are shown in Figure 8 and Figure 9, respectively. In both figures, we can see that Q-Learning performs better than SARSA. This is interesting because one would expect that SARSA would perform better than Q-Learning on online performance. However, when we examine the reward structure, we can see that the only positive or negative reward that is given is at the end of the episode. This means that there is not necessarily a safe path or trajectory to follow to avoid a random action that results in a negative reward. Therefore, SARSA will not show any improvement over Q-Learning in online performance.

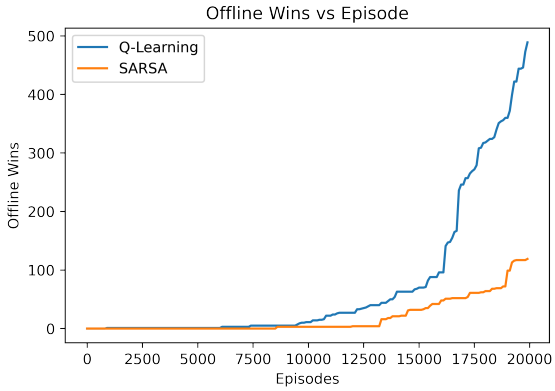

Figure 7. Q-learning vs. SARSA cumulative offline wins.

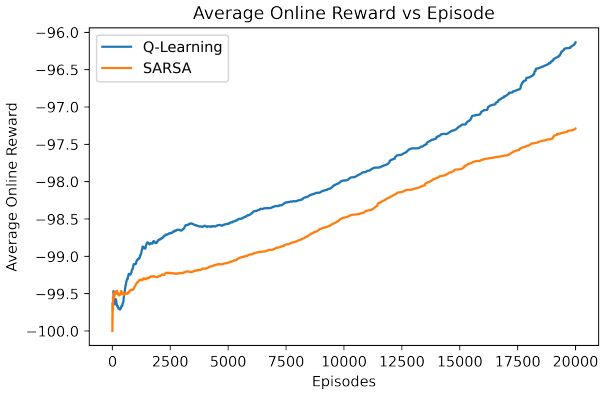

Figure 8. Q-learning vs. SARSA average online reward.

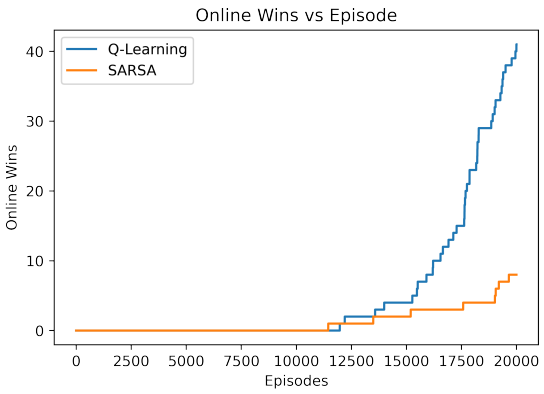

Figure 9. Q-learning vs. SARSA cumulative online wins.

6. Conclusion

In this project, we have implemented the Q-Learning and SARSA algorithms to play the game of Pong. We have learned that Q-Learning is a robust algorithm that can be used to solve complex problems. We have shown that Q-Learning has a better offline performance than online in both average return and cumulative wins. Additionally, we have demonstrated that Q-Learning can learn the optimal policy in a reasonable amount of training time.

We have experimented with the alpha parameter (learning rate) to show that the larger the value, the faster the model improves. However, the larger the alpha value, the more likely it is to overshoot the optimal solution. We have also implemented the SARSA algorithm to compare it to Q-Learning. We have shown that Q-Learning can learn the optimal policy and perform better than SARSA on both online and offline performance.