

Atari Breakout Reinforcement Learning Environment

Haider Sajjad
1004076251

HAIDER.SAJJAD@MAIL.UTORONTO.CA

Weiyu Li
1003765981

WEIYU.LI@MAIL.UTORONTO.CA

Abstract

Atari Breakout environment implementation and training an agent using multiple algorithms over a generated environment (changing brick layouts)

1. Introduction

Our project is implementing an Atari breakout environment and training an agent to play it across general levels. The project repository can be found here: <https://github.com/duoduocai-dot/csc498-project> To play the original game, after cloning the project, just run the file `run_Breakout.py`¹.

Our environment is here². We made a made our environment embedded into a pygame class for Breakout, adding environment functions and variables.

We made 6 algorithms which play Breakout, and compare their performance in this report. The algorithms we made are: DQN, double-DQN, policy gradient REINFORCE, tabular Q-Learning, tabular double Q-Learning, and tabular SARSA.

2. Environment

Our environment is here: <https://github.com/duoduocai-dot/csc498-project/blob/main/breakout.py>.

2.1 Rewards

We experimented with various reward mechanisms. Our original reward schema was:

- +10 for ball hitting paddle
- -10 ball missing paddle
- +1 ball hits brick
- -0.2 movement penalty
- +1000 ball destroys all bricks

However, with this model we noticed it was difficult for the agent to actually learn, as it doesn't get direct feedback after making an action. So we altered the reward schema to be:

1. https://github.com/duoduocai-dot/csc498-project/blob/main/run_Breakout.py

2. <https://github.com/duoduocai-dot/csc498-project/blob/main/breakout.py>

- +10 for ball hitting paddle
- -10 ball missing paddle
- Distance between paddle and ball in the x-Axis³
- +1000 ball destroys all bricks

2.2 Environment variables, functions

Regarding the rest of the environment, the step function is here⁴, where the paddle moves depending on the action passed in, and updates rewards for that step. It then returns the game state which is [paddle x-location, ball x-location, ball y-location, ball x-speed, ball y-speed, bricks left]⁵.

The reset function resets the environment, setting ball, bricks, paddle to their original position.

Render function, renders the game. The make function creates the brick layout, and main function allows a player to play the game.

Agent actions are to move left (0), move right (1), stay still (2).

There is more about the environment including the brickLayout which we used in testing and comparing the tabular algorithms which will be talked about in it's own section.

3. Algorithms

3.1 Tabular Algorithms

We made 3 algorithms which utilize a tabular setup. To discretize the problem, we reduced the state sizes such that they could fit inside a table.

How we created the tabular setup is by first getting every possible paddle and ball locations using this discretizeStateSpaceAllStates function⁶ which splits the paddle x-locations into 20 possible locations, the ball x-locations into 80 possible locations, and ball y-locations into 30 possible locations. The game screen is 800x600 and paddle is 80 wide and ball is 5x5.

We experimented with 3 different schemas to discretize the state space. These included:

- 10 paddle locations, 40 ball x-locations, 20 ball y-locations
- 10 paddle locations, 80 ball x-locations, 30 ball y-locations
- 20 paddle locations, 80 ball x-locations, 30 ball y-locations

The last one with the most paddle and ball locations worked best:

3. <https://github.com/duoduocai-dot/csc498-project/blob/main/breakout.py#L179>

4. <https://github.com/duoduocai-dot/csc498-project/blob/f46268a6c9241b77efd6c96a6dd8ecc3ec5bae1e/breakout.py#L266>

5. <https://github.com/duoduocai-dot/csc498-project/blob/f46268a6c9241b77efd6c96a6dd8ecc3ec5bae1e/breakout.py#L246>

6. https://github.com/duoduocai-dot/csc498-project/blob/main/tabular_Q_learning.py#L223

Rewards from the last 10 episodes using fewest number of states (first bullet):

```
episode = 9990, epsilon = 0.0009000000000938177, rewards = -41954
episode = 9991, epsilon = 0.0008000000000938177, rewards = -33707
episode = 9992, epsilon = 0.0007000000000938176, rewards = -41622
episode = 9993, epsilon = 0.0006000000000938176, rewards = -48913
episode = 9994, epsilon = 0.0005000000000938175, rewards = -42440
episode = 9995, epsilon = 0.0004000000000938175, rewards = -269565
episode = 9996, epsilon = 0.0003000000000938175, rewards = -31498
episode = 9997, epsilon = 0.0002000000000938175, rewards = -59294
episode = 9998, epsilon = 0.0001000000000938175, rewards = -26515
episode = 9999, epsilon = 9.381755897326649e-14, rewards = -36514
```

Rewards from the last 10 episodes using largest number of states (last bullet):

```
episode = 9990, epsilon = 0.0009000000000938177, rewards = -29335
episode = 9991, epsilon = 0.0008000000000938177, rewards = -37168
episode = 9992, epsilon = 0.0007000000000938176, rewards = -38860
episode = 9993, epsilon = 0.0006000000000938176, rewards = -33860
episode = 9994, epsilon = 0.0005000000000938175, rewards = -62025
episode = 9995, epsilon = 0.0004000000000938175, rewards = -34979
episode = 9996, epsilon = 0.0003000000000938175, rewards = -35564
episode = 9997, epsilon = 0.0002000000000938175, rewards = -35623
episode = 9998, epsilon = 0.0001000000000938175, rewards = -31513
episode = 9999, epsilon = 9.381755897326649e-14, rewards = -35037
```

It's clear that the larger number of states gives more rewards.

After, we create dictionaries to store the policy and Q-values, with every possible state using the schema above.

3.1.1 TABULAR Q-LEARNING

For tabular Q-Learning, now with a table of all possible states, we can just apply the Q-Learning epsilon greedy algorithm. Where the Q-Learning step is:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)) \quad (1)$$

This exact Q-Learning update can be found here. Where we get the Q-value at each action for the next state, and plug in the max Q-value for the next state into the equation.

The other important aspect of Q-Learning is the epsilon-greedy exploration. We first initialize our epsilon in the class, then it is updated after every episode.

Regarding epsilon decay, to increase exploitation rather than exploration, instead of the usual $\epsilon = \epsilon/k$ approach, we use $\epsilon = \epsilon + \text{decay}$ where decay is some negative number between 0 and 1.

3.1.2 TABULAR DOUBLE Q-LEARNING

In tabular double Q-Learning, our algorithm is nearly the exact same as tabular Q-Learning, except we maintain two Q-value dictionaries and each one is updated with 0.5 probability. Pseudocode for double Q-Learning:

Select a_t using epsilon greedy $\pi(s) = \text{argmax}_a Q_1(s_t, a) + Q_2(s_t, a)$
with 0.5 probability:

$$Q_1(s_t, a_t) \leftarrow Q_1(s_t, a_t) + \alpha(r_t + \gamma Q_1(s_{t+1}, \text{argmax}_a Q_2(s_{t+1}, a)) - Q_1(s_t, a_t)) \quad (2)$$

else:

$$Q_2(s_t, a_t) \leftarrow Q_2(s_t, a_t) + \alpha(r_t + \gamma Q_2(s_{t+1}, \text{argmax}_a Q_1(s_{t+1}, a)) - Q_2(s_t, a_t)) \quad (3)$$

3.1.3 TABULAR SARSA

The only thing different from the tabular SARSA implementation from the above two, is that the Q-value update is done using a bootstrapped one-step look ahead⁷, like in the equation:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)) \quad (4)$$

3.1.4 TRAINING

To train each of the algorithms, there is a training function which creates an instance of the algorithm then loops through a number of episodes collecting states, actions and rewards and using them for training. The code to run the training is in `tabular_experiments.py`⁸

3.2 Tabular algorithms Analysis

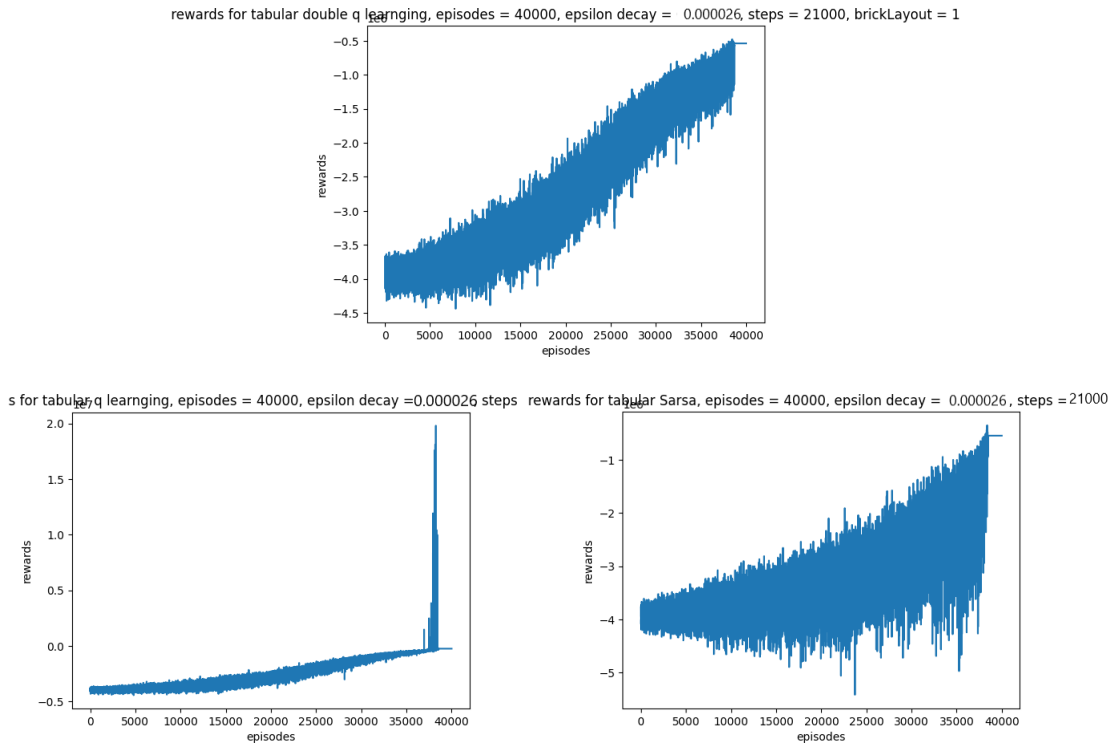
3.2.1 PERFORMANCE

All 3 tabular algorithms do achieve good performance and are able to play the game effectively.

Here are graphs for all 3 algorithms, it's clear that as episodes increase and epsilon grows smaller, all 3 algorithms achieve higher rewards. These graphs were made using the `experiment2` function in `tabular_experiments.py`. All three algorithms were trained for 40000 episodes 21000 steps each, with a decay value of -0.000026 as specified in the `experiment` function. They were all trained on brick-layout 1 (the one with the spaces between bricks, check brick-layout section) The bottom left graph is Q-Learning, top is double Q-Learning, and bottom right is SARSA. Each algorithm took approximately 5-6 hours to train with the above episodes and steps. Also the y-axis is on a scale of 10^6 for DQL and SARSA and 10^7 for QL.

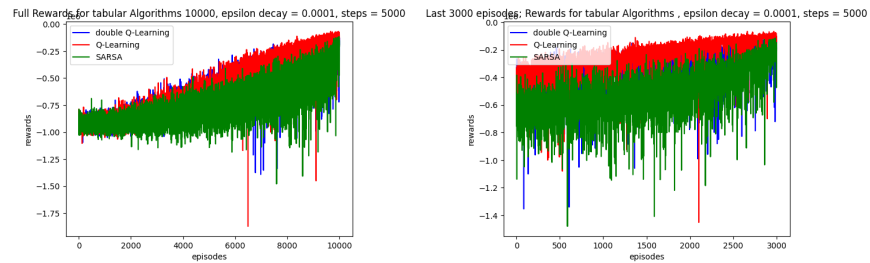
7. <https://github.com/duoduocai-dot/csc498-project/blob/main/Sarsa.py#L80>

8. https://github.com/duoduocai-dot/csc498-project/blob/main/tabular_experiments.py#L130



3.2.2 COMPARING TABULAR ALGORITHMS

Here is a direct comparison of all 3 algorithms trained over 10000 episodes for 5000 steps per. It's clear from the pictures that Q-Learning and double Q-Learning outperform SARSA. First graph is full rewards comparison over all episodes, second is over the last 3000 episodes:



3.2.3 PARAMETER AND HYPERPARAMETER CHOICE

In the tabular algorithms there were many parameters that we used: epsilon and decay-value, episodes, step size, learning rate (α), discount (γ).

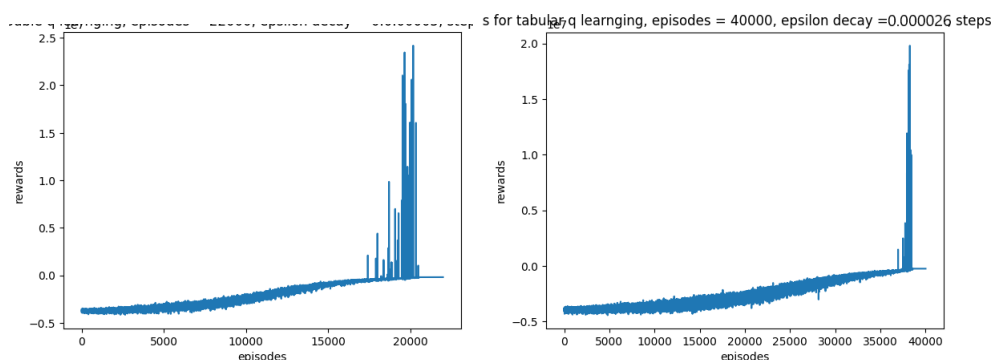
The choice of epsilon was simply just 1. We wanted lots of exploration in the beginning, for the agent to learn the environment. The epsilon decay was implemented such that: $\epsilon = \epsilon + decay$. The reason for this is because we wanted a linear decrease of epsilon to account for lots of exploration.

Our choice of step size to fully complete the game is 20000. The reason for the large step size is that the game overall takes a long time to complete. For example, this video example⁹ of a double Q-Learning agent playing the game takes more than 30000 steps.

Our choice of γ was 0.9 because we wanted the agent to count future states heavily because it takes many steps for the agent to win the game, and we wanted to account for this with our discount.

3.2.4 Q-LEARNING MAXIMIZATION PROBLEMS

We had maximization bias occur in our Q-Learning agents:



This is an example video¹⁰ of the agent playing the game, from the first graph. As it is seen from the end of the video, the agent gets stuck in an infinite loop of bouncing the ball in the same pattern which happens to avoid the bricks. From the graph, it is clear that the agent was able to win the game in some episodes (from 17000-22000), but after, the rewards converged incorrectly. What ended up happening is that at some point in an episode the agent picks an action that gives it a higher reward now, but not in the future. One reason for this is because the agent is essentially just trying to get as close to the ball as possible, from our reward model, we don't reward the agent when the ball hits a brick, and there is no brick representation in the state, so it's a partially observed state. So the agent is trying to center itself with the ball, and that doesn't guarantee that it will hit the ball in such a way that a brick will break.

To fix this, we can add a more complex form of exploration, such that the agent can explore more when it gets close to the end or gets stuck in a loop of avoid bricks.

Or we can change our entire environment to go from a partially observed MDP to a fully observed one, modelling brick-layout, score, ball location/speed, paddle location, etc. But for this, the algorithms also get more complex, and we won't be able to use our tabular setup.

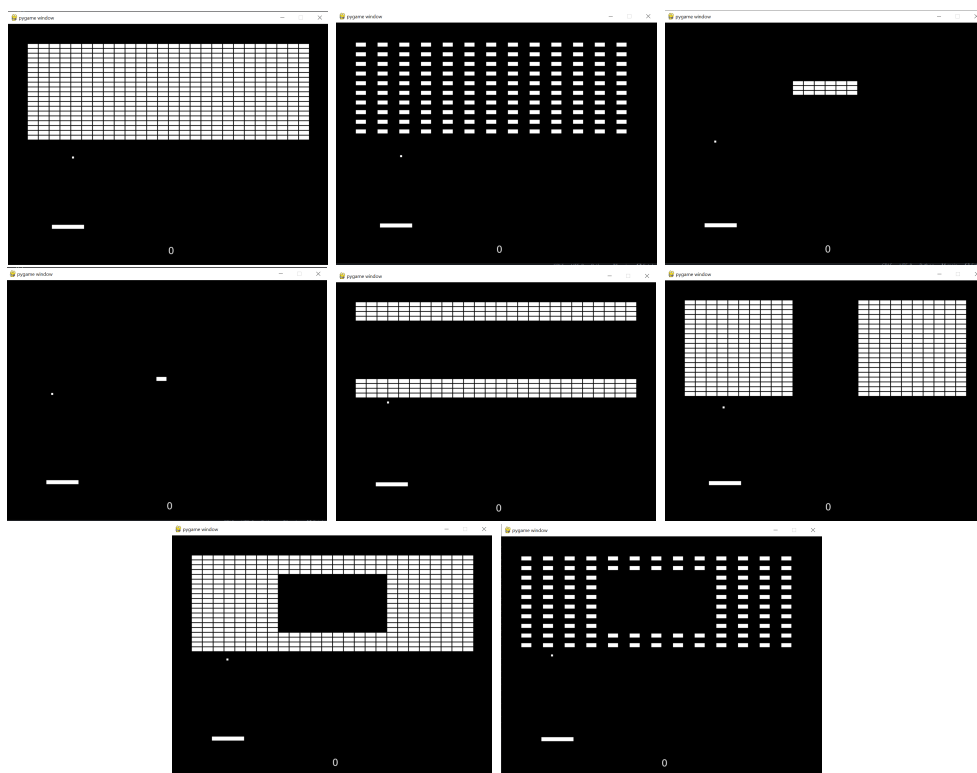
9. https://github.com/duoduocai-dot/csc498-project/blob/main/pictures_and_videos/tabular_double_q_learning_example.mkv

10. https://github.com/duoduocai-dot/csc498-project/blob/main/pictures_and_videos/incorrect_double_q_learning_maximization_bias_example.mkv

Another possible fix is to alter reward mechanics such that hitting a brick awards rewards. The reason we didn't do this initially is because the ball hitting a brick is something that happens much later after the paddle hits the ball that it was harming our training in the initial stages. I.e, the agent would get random rewards, but wouldn't be able to recreate it.

3.2.5 GENERALIZATION

Using the same agents trained in 3.2.1 section (all the algorithms trained on the same number of episodes and steps), we made generalization tests. In our environment we inbuilt several brick-layouts that the game can play on¹¹. These are:

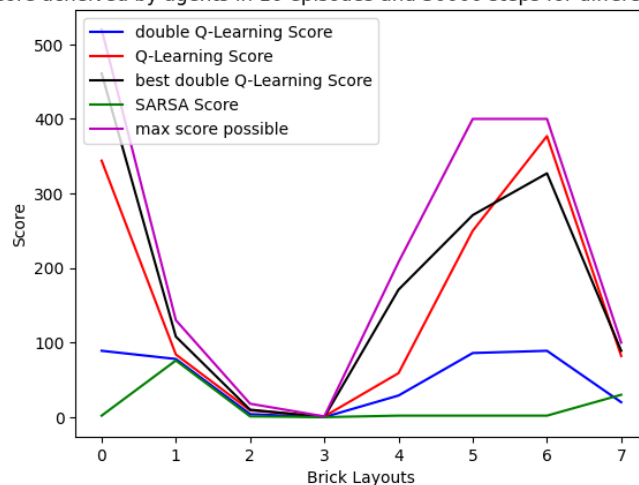


The experiment we made to test generalization is here¹². Essentially what it's doing is getting the trained agents from all 3 algorithms and running them across all layouts and measuring their score. Those algorithms were trained on layout1 (second layout in the top row). And along with them we included one more trained double Q-Learning agent which was trained for 100,000 episodes, and graphed the results. On the graph is also the max possible score achievable per layout as a reference to how well the algorithm does.

11. <https://github.com/duoduocai-dot/csc498-project/blob/main/breakout.py#L64>

12. https://github.com/duoduocai-dot/csc498-project/blob/main/generalization_tests.py#L10

Best Score achieved by agents in 10 episodes and 30000 steps for different brick layouts

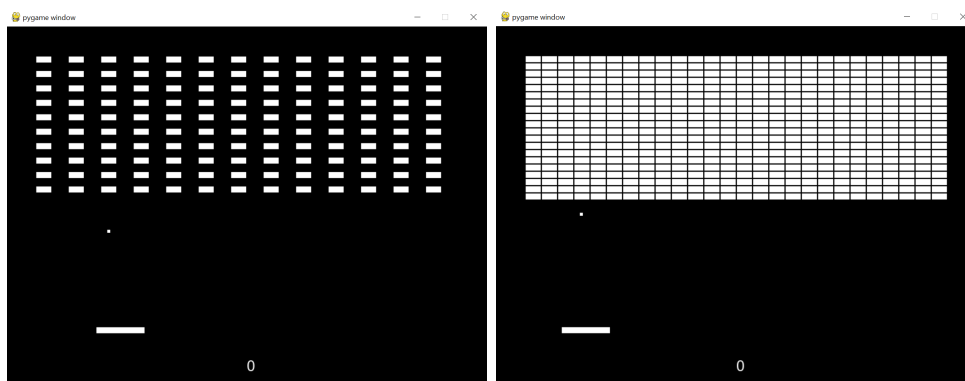


We can see both the Q-Learning and the best double Q-Learning algorithm do very well in generalizing across different layouts, both their scores are close the the maximum possible score.

Here are example videos of the best double Q-Learning agent generalizing across brick layouts.¹³

3.2.6 WHICH LAYOUT IS BEST TO TRAIN ON?

We think that the layout best to train on is brick-layout number 1 (left). It is better than brick-layout 0 (right) because it allows for the ball to go in between bricks in a much earlier part of the episode than layout 0 allows. Thus the agent is able to learn more variations of ball locations and velocities.



13. https://github.com/duoduocai-dot/csc498-project/tree/main/pictures_and_videos/generalization_over_layouts

3.3 Policy Gradient

Our policy gradient implementation is here¹⁴.

3.3.1 REWARD SCHEMA PROBLEMS

We had many issues with the initial reward schema in section 2.1. What was happening was that the agent was not moving. Essentially, the agent saw that any movement would result in lower reward, and our learning rate was too high, such that the agent would get stuck in a gradient minimum. When we changed the reward schema to get close to the ball, the agent began learning.

3.3.2 GRADIENTS GETTING TRAPPED

Another problem with policy gradients that we were running into was the gradients getting stuck in a local min.

We experimented a lot with the hyperparameters. Specifically, with the learning rate, we had some success by lowering it to be very small¹⁵, especially because our episode size was so large and as a result loss was very large. Another thing that helped with this problem was dividing loss by a large constant number¹⁶. However, even with all these changes the gradient still got stuck at random local minimas. Here is a video of a policy gradient agent playing the game¹⁷.

Other things we tried to solve this problem was by including exploration. We added a greedy exploration in the algorithm, so the agent can get a feel for the environment, although it was unclear if this helped as the same problems above kept occurring.

Some other things we could try to fix this is by using momentum in the gradient update (although Adam optimizer used¹⁸ should already be doing this). Another potential fix is to use the soft-actor critic algorithm¹⁹ which maintains stochasticity by having ample exploration.

3.4 Deep Recurrent Q-Network

The deep recurrent Q-network has most of the ideas from the paper ..., to calculate the target function, we utilize a gated recurrent unit with one layer. Architecture: unidirectional, input size: 6 (observation space), one hidden layer of size 10, output size 3 (action space). In the forward pass, the computed value is then passed into a linear model. In the experience replay buffer, we sample sequences instead of single points.

14. https://github.com/duoduocai-dot/csc498-project/blob/main/policy_gradient.py

15. https://github.com/duoduocai-dot/csc498-project/blob/main/policy_gradient.py#L55

16. https://github.com/duoduocai-dot/csc498-project/blob/main/policy_gradient.py#L93

17. https://github.com/duoduocai-dot/csc498-project/blob/main/pictures_and_videos/best_policy_gradient_agent.mkv

18. https://github.com/duoduocai-dot/csc498-project/blob/main/policy_gradient.py#L55

19. <https://arxiv.org/abs/1801.01290>

The reason we use GRU:

- Given that the problem is relatively simple, using LSTM is inefficient.
- The forget gate of LSTM is not useful in our case, because we don't need to skip any weak past frame and reuse them in later stages.

3.5 Conclusion

In conclusion, If a problem's state space can be discretized, then it should be. The tabular algorithms that performed the best on this problem were Q-Learning and double Q-Learning. When discretizing the problem, more states is usually better, however it takes longer to train. And if the problem can be solved with fewer states it is better to stick with that. When deciding the best game environment to train over for a general agent, it is better to pick one where the agent sees a variety of game states early, so it can learn more efficiently using fewer samples.

Regarding Policy Gradient methods for this problem, they are overall not good. The action space is discrete and the state-space can be made discrete. So using Policy Gradient just over complicates the task. If you were to use policy gradient for this sort of problem, then use something like soft-actor critic with ample exploration, or implement an extensive exploration method. Also use entropy loss, to penalize the selection of actions with high-likelihood.

Furthermore, always pick a good reward schema, one such that the agent can directly measure how good an action is. Not a schema where an agent makes an action, and has to wait until later to learn if it was a good action or not.