Atari Breakout Reinforcement Learning Environment

Haider Sajjad haider.sajjad@mail.utoronto.ca
1004076251

Weiyu Li weiyu.li@mail.utoronto.ca
1003765981

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 this file: `https://github.com/duoduocai-dot/csc498-project/blob/main/run_Breakout.py`.
Our environment is here. 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 if making an action is good or not. So we altered the reward schema to be:

- +10 for ball hitting paddle
- -10 ball missing paddle
- Distance between paddle and ball in the x-Axis `https://github.com/duoduocai-dot/csc498-project/blob/main/breakout.py#L179`
- +1000 ball destroys all bricks

2.2 Environment variables, functions

Regarding the rest of the environment, the step function is here, 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].
The reset function resets the environment, setting ball, bricks, paddle to their original position.
Render functio, renders the game. Make function creates the brick layout, and main function allows the user 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 I will talk about in that section.

3. Algorithims

3.1 Tabular Algorithms

We made 3 algorithms which utilize a tabular setup. We reduced the state sizes such that they could be discretized, and fit inside a table.

How we created the tabular setup is by first getting every possible paddle and ball locations using this discretizeStateSpaceAllStates function 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.

I 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:

Here are the rewards from the last 10 episodes of training using the 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 = 0.0003000000000938176, rewards =  -31498
episode = 9997, epsilon = 0.0002000000000938176, rewards =  -59294
episode = 9998, epsilon = 0.0001000000000938176, rewards =  -26515
episode = 9999, epsilon = 9.381755897326649e-14, rewards =  -36514
```

Compared the rewards from the last 10 episodes of training using the most 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 = 0.0003000000000938176, rewards =  -35564
episode = 9997, epsilon = 0.0002000000000938176, rewards =  -35623
episode = 9998, epsilon = 0.0001000000000938176, rewards =  -31513
episode = 9999, epsilon = 9.381755897326649e-14, rewards =  -35037
```

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

After, we (`https://github.com/duoduocai-dot/csc498-project/blob/main/tabular_Q_learning.py#L37`) just create a dictionary 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:

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

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. The policy is updated for every state inside the q_learning function using the same epsilon-greedy approach.

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 update with 0.5 probability.

Pseudocode for double Q-Learning:
Select a_t using epsilon greedy $\pi(s) = argmax_a Q_1(s_t, a) + Q_2(s_t, a)$
with 0.5 probability:

3.1.4 Comparing tabular algorithms

3.2 Policy Gradient