Train RL Agent to Play Super-Mario

[Jiajun Ren, Yuen Zhou, Zhouyang Wang]

[Queen's University]

Fall 2022

Introduction and Problem Formulation

Reinforcement learning is a reward guidance behaviour obtained by agents through interaction with the environment through "trial and error". The goal is to make the agents get the maximum reward. The reward provided by the environment in reinforcement learning is a kind of evaluation of the quality of the action rather than telling the reinforcement learning system (RLS) how to produce the correct action. As the external environment provides little information, agents must rely on their experience to learn. In this way, agents gain knowledge in the action-evaluation environment and improve the action plan to adapt to the environment.

Super Mario Bros. is a horizontal board video game in which the player takes the role of Mario through trials to reach the finish line. The game environment was taken from the OpenAI Gym using the Nintendo Entertainment System (NES) python emulator (Kauten, 2022). We aim to train an agent to finish the game with reinforcement learning algorithms. Based on the SuperMarioBros-1-1-v0 game scene, Deep Q Learning (DQN) and Double Deep Q Learning (DDQN) will be used to compare the best algorithm to finish the game.

State/ Observation Space

The observation space for the Super Mario Bros game is a 240 x 256 x 3 RGB image. However, it is too big for training. To save training time, we decided to narrow the size. According to the instructions on the Pytorch tutorial, we applied wrappers to the environment so that the final state consists of 4 continuous 84 x 84 grayscale pixel frames, with a discrete high value of 255 and a low value of 0. Also, the pixel value is normalized from 0 to 1 (Feng et al.).

Action Space

We have two types of actions: SIMPLE_MOVEMENT and RIGHT_ONLY. The action space for SIMPLE_MOVEMENT is discrete and shown as [['NOOP'], ['right'], ['right', 'A'], ['right', 'B'], ['right', 'A', 'B'], ['A'], ['left']], where 'A' represents an attack, and 'B' represents jump. Similar to SIMPLE_MOVEMENT, the action space for RIGHT_ONLY movement is also discrete but only has 5 possible actions: [['NOOP'], ['right'], ['right', 'A'], ['right', 'B'], ['right', 'A', 'B']] (Kauten, 2022).

Reward Scheme

We wanted our agent to run as fast as possible to the right place. So, the reward consists of three variables: v for position difference, c for the time difference, and a death penalty d. So, r = v + c + d (Kauten, 2022).

Where v is: $v \begin{cases} =0 & no\ moves \\ <0 & move\ lef\ t \\ >0 & move\ right \end{cases}$ and d is: $d \begin{cases} =0 & \text{live} \\ =-15 & \text{death} \end{cases}$

c = t'- t, c starts at 0 and keeps decreasing with the time t goes on (the game is a countdown).

Methodology and Algorithm Description

Since the most primitive Q-learning algorithm always needs a Q table to record during the execution process, the Q table can still meet the demand when the dimension is not high. When it encounters exponential dimensions, the efficiency of the Q table is very limited and not feasible in terms of code complexity and our equipment. Thus, we consider a method of approximating the value function. We can obtain the corresponding Q value in real time if S or A is known in advance. Initially, the Q value was found through the state and action, but now the

neural network is used to obtain the approximate Q value (Hasselt et al., 2015). We directly use the game screen as a state S and input it into the neural network, and let each output of the network be the Q value corresponding to an action A so that we can obtain the Q value of each action in the state S. For this Mario game, we cannot tell from one frame whether Mario is rising or falling. So, we input several consecutive frames simultaneously, and the neural network can know whether Mario is rising or falling.

For the neural network, we build the convolutional layers first. Since the input is an 84*84*4 game scene, the stride of the input image is 4 (Kauten, 2022). After obtaining the output of the convolutional layers and expanding it into one dimension, the fully connected layer size is entered to build a fully connected layer.

DQN is a combination of Q-Learning and neural networks. It turns the Q-table of Q-Learning into Q-Network. In the fixed-capacity experience pool, a batch of data is randomly taken out from the experience pool each time for training, and the parameters are updated by gradient descent. For the algorithm description of DQN and DDQN, here are the formulas of these two algorithms' target functions:

DDQN: Q*(S, A) <- r + gamma * max_a Q_target(S', a) (1)

DQN: Q*(S, A) <- r + gamma * max_a Q(S', a) (2)

The DDQN algorithm is almost the same as the DQN algorithm, mainly for the latter's overestimation problem, changing the calculation method of the target value. Since DDQN is based on the parameters of the current Q network each time, it is not based on the parameters of the target-Q like DQN, so when calculating the target value, it will be a little smaller than the original. (Since the target value calculation must go through the target-Q network). DQN selected the action with the largest Q value according to the target-Q parameter. In DDQN, the Q

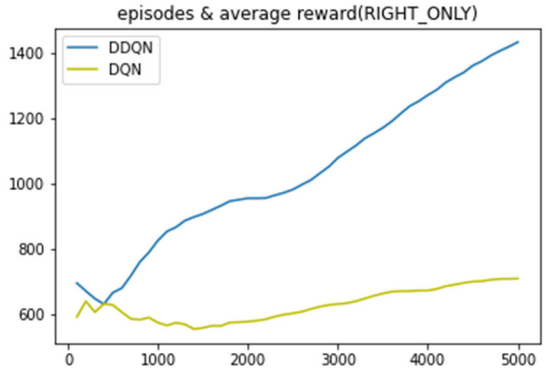
value is calculated after replacing the selection, and it must be less than or equal to the original Q value to reduce the overestimation to a certain extent and make the Q value closer to the actual value.

We built the DQN_agent class and included hyperparameters similar to DQN_network class. After setting aside the definition and usage environment of some DQN Layers, we use the DQN_network class to establish the implementation basis for the next DQN and DDQN. In the DQN algorithm, we only need to load one neural network. The DDQN algorithm uses two neural networks (the evaluation network and the target network). We use Adam as an optimizer. We also use an experience pool, which differs from traditional Q-Learning. Subsequent operations are putting data into the experience pool, obtaining data in batches, and obtaining actions in the current state through data experience. In DQN and DDQN, one of the most important steps is building a playback buffer in the experience playback function. When a policy interacts with the environment, the playback buffer collects data. We put all the data in a data buffer, which stores the experience of different policies. Each round of iteration has a corresponding batch_size, and then we use the sampling experience in this batch_size to update the target functions.

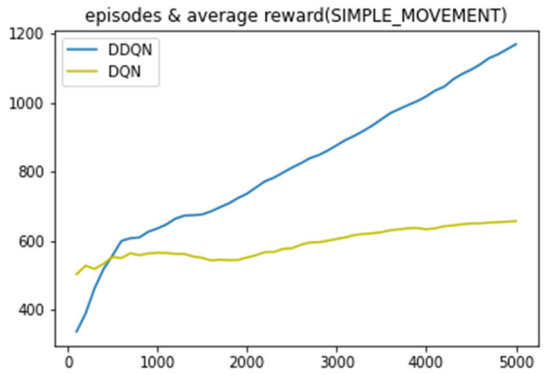
A start function is used to cover all the implementation steps. The preprocessing functions preprocess the states of the environment into 3-D arrays for DQN and DDQN. For the hyperparameters for the algorithms class, settings include 30000 experience pool storage length, 0.9 gamma, 32 batch_size and 0.00025 learning rate are selected. Save the outputs as charts after 5000-episode runs for each case.

Results

We train and test the agent on the RIGHT_ONLY actions and SIMPLE_MOVEMENT actions. Here is the RIGHT_ONLY's DQN & DDQN comparison chart:



and SIMPLE_MOVEMENT's DQN & DDQN comparison chart:



As mentioned above, DQN will overestimate the real Q value due to the maximization bias. The overestimated Q value is used again to update the weight of the Q network through backpropagation, which makes the overestimation more serious. This shortcoming leads to a very slow convergence of DQN. On the other hand, DDQN uses a model and a target model to minimize the mean squared error between the two. Therefore, DDQN can avoid maximization bias by separating updates from biased estimates. Furthermore, compared with the DQN, the DDQN training episodes are much fewer. Thus, the DDQN helps to eliminate the overestimation

problem found in the DQN network. After training, the gap between the two becomes significantly larger after about 1000 times of training. In the RIGHT_ONLY graph, when the average_reward value of DDQN reaches 1300, the average_reward value of DQN is only about 400. In SIMPLE_MOVEMENT, when DDQN comes to about 1000, DQN is only around 650. Thus, under RIGHT_ONLY and SIMPLE_MOVEMENT, DDQN can converge faster, so DDQN can find the optimal path and pass the Super Mario game faster than DQN.

The DQN and DDQN networks can train better on RIGHT_ONLY moves rather than simple and complex motion action spaces. According to the reward function we mentioned before (r = c+v+d, while going left, v >= 0), the reward attenuation of RIGHT_ONLY is less than the SIMPLE_MOVEMENT action's attenuation. Its number of steps required to reach the end is also less than the SIMPLE_MOVEMENT action's steps, so the slope of the chart of RIGHT_ONLY DQN and DDQN is larger than the slope of SIMPLE_MOVEMENT DQN and DDQN chart. So, DQN and DDQN under the RIGHT_ONLY actions can converge faster than DQN and DDQN under the SIMPLE_MOVEMENT actions.

Discussion

The three members of our group contributed equally to this project.

The first problem with this project is that the original observation space for Super Mario Bros is too big. Using it directly causes computing observations to take up too much space and requires memory optimization. We converted the original image to 84x84 size and the colour image to grayscale. [0,255] uint8 was stored in the replay buffer; the memory space was reduced to 1/4 of the original and then scaled to [0,1] when inputting the neural network.

Another challenge is equipment limitation. Even after compression, this project requires a lot of computation. Due to our low CPU and graphics configuration, we used Colab Pro+ to reduce training time costs.

For future enhancement, we chose 0.00025 as the learning rate (1/4 of 0.001), which is a very small value. We know that if the learning rate is large, the parameter update range will be huge, which is very effective when the number of samples is small. However, in our project, we do not want the agent to be affected too much due to the game's difficulty (it will fail many times, and there will be many samples). Therefore, a tiny learning rate is chosen. In the future, we can use different learning rates to gain a faster training convergence.

References

C. Kauten, “Gym-super-mario-bros,” *PyPI*, 21-Jun-2022. [Online]. Available: https://pypi.org/project/gym-super-mario-bros/. [Accessed: 07-Dec-2022].

H. van Hasselt, A. Guez, and D. Silver, “Deep reinforcement learning with double Q-learning,” *arXiv.org*, 08-Dec-2015. [Online]. Available: https://arxiv.org/abs/1509.06461. [Accessed: 07-Dec-2022].

Y. Feng, S. Subramanian, H. Wang, and S. Guo, “Train a mario-playing RL agent,” *Train a Mario-playing RL Agent - PyTorch Tutorials 1.13.0+cu117 documentation*. [Online]. Available: https://pytorch.org/tutorials/intermediate/mario_rl_tutorial.html#. [Accessed: 07-Dec-2022].