Deep Reinforcement Learning with Mario Games

Author: Shiqi Zheng

Univerisity of Florida, Gainesville, Florida, the United States

Corresponding author e-mail: zhengshiqi@ufl.edu

Abstract. Deep reinforcement learning (DQN) has multiple advantages in handling large-scale data and learning to play a task by trials and errors. In this paper, I evaluate this advantage and apply a DQN model to play Mario games. With the help of replay buffer and target model, the model gains about 2,000 reward scores after playing 7,183 times. I evaluated my model with different eps decay, disturbance, and action list and looked for improvements and limitations.

1. Introduction

As we know, deep reinforcement learning is suitable for gameplay by setting reward and the environment. The learning model would interact with the environment and updates itself by learning from errors. By providing appropriate Q-value functions and loss functions to solve gradient descent, I designed my DQN model that learned from the feedback loop. Also, in order to randomly shuffle the dataset and dynamically train the frames, I implement a replay buffer with push and sample functions. A target model is created alongside the current model, assisting the loss function usage. My DQN model reached a high reward score, and it can successfully pass the Mario game.

2. Related work

Mnih and his team design a deep reinforcement learning model and surpass human experts on multiple games. They create an algorithm about Deep Q-learning with Experience Replay, “which allows for greater data efficiency”[1]. Then they preprocess the dataset by cropping the images and train their models through a dedicated reward function. Their paper inspires me a lot, and the replay buffer idea of my project is based on that.

Another related work of DQN on playing games is TD-gammon, “a backgammon-playing program which learned entirely by reinforcement learning and self-play, and achieved a superhuman level of play [1].” Their model utilizes a multi-layer perceptron with one hidden layer. However, some reviews believe that the TD-gammon method is mainly for special cases and not appropriate for the large-scale dataset.

Besides, according to Tsitsiklis and his team's paper *An analysis of temporal-difference learning with function approximation*, non-linear approximators could cause Q-learning to diverge. To solve this issue, I introduce linear function approximators in the model to make sure it has better convergence guarantees.

3. DQN Model

My DQN model submission has three parts: training network, trained model, and playing module. "DQN_super_mario.ipynb" is my training network, and it requires jupter notebook installed. Also, since my model is for playing Mario, we also need to run

```
!pip install gym_super_mario_bros
```

to install gym mario module. Also, I saved my model as "Local_DQN_Mario_big_4", which can be used to play the Mario game by running "Mario_Play.ipynb".

3.1. Env Wrappers

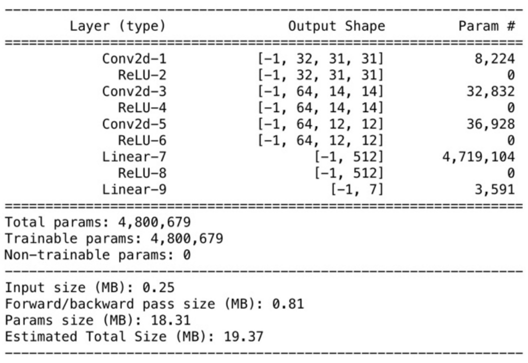
The wrappers bascially adjust the action space and the observation space for the DQN model. I created four wrappers: MaxAndSkipEnv, ProcessFrame128, BufferWrapper, and ImageToPyTorch. The MaxAndSkipEnv wrapper is to pick one image from two, saving the memory space. ProcessFrame128 would fit the image size to 128*128 and transform to greyscale, aiming to save memory as well. The BufferWrapper creates the basic function that aggregating four consequent frames as a state. The reason is that by combining the four frames, we can get a movement trend, which is helpful for our DQN model to train and find the action function. The final ImageToPyTorch wrapper is just for transforming data to tensor so that Torch can accept my dataset.

3.2. Replay Buffer

In the replay buffer, I will store the training as "Experience", which includes state, action, reward, new_state, and done. I created push and sample function in order to store and load my "Experience" into the buffer.

3.3. DQN Structure

My DQN network has three convolutional, two linear, and four ReLU layers. It's very straight forward, and here is a summary figure of my structure.



```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 32, 31, 31]           8,224
              ReLU-2           [-1, 32, 31, 31]               0
            Conv2d-3           [-1, 64, 14, 14]          32,832
              ReLU-4           [-1, 64, 14, 14]               0
            Conv2d-5           [-1, 64, 12, 12]          36,928
              ReLU-6           [-1, 64, 12, 12]               0
            Linear-7                  [-1, 512]       4,719,104
              ReLU-8                  [-1, 512]               0
            Linear-9                    [-1, 7]           3,591
================================================================
Total params: 4,800,679
Trainable params: 4,800,679
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.25
Forward/backward pass size (MB): 0.81
Params size (MB): 18.31
Estimated Total Size (MB): 19.37
----------------------------------------------------------------
```

Figure 1. DQN Structure

3.4. DQN Agent

The DQN Agent would generate two models. The first model is a real-time model that updates every step. The second one is a target_model, which updates every 10,000 steps and copies from the real-time model. The latency between these two models can be used to solve gradient descent by utilizing SmoothL1Loss loss function and Adam optimizer. In addition, I create an eps value, serving as a control gate to determine the next action is random or not. When we begin to train the model, we want our model to explore as many possibilities as it can by using random action. When the training time gets longer, we want our model start to use the neural network to predict the next action.

3.5. train_model

For each episode loop, the train_model will generate a mean reward. If the current reward is higher than the max mean rewards, the model will be updated and stored. For each step loop, the model will generate the episode, last_100_reward, and current reward information when a game is finished.

4. Result

After comparing all the models I generated from University of Florida HiPer Gator machine, I choose "Local_DQN_Mario_big_4" as my final model. Using this model, "Mario_Play.ipynb" can always pass the Mario game and get an average reward score of 3064.5. The 100 times rewards results are summarized in Table 1. The last 100 reward trend is shown in Figure 2.

Table 1. Reward score for running 100 times

Type	Reward Score
median	3072.0
mean	3064.5

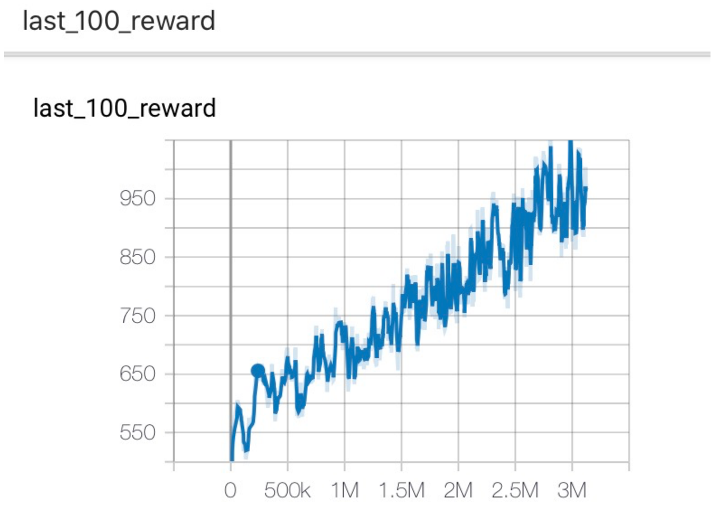


Figure 2. last 100 reward trend (from TenserBoard)

5. Evaluation

5.1. Small decay on eps vs. large decay on eps

As I mentioned in 3.4 DQN Agent, eps plays an important role in determining the next action is random or not. The model I provide in my submission actually uses a decay rate of 0.9999975. To be more specific, the model will generate a random number and compare it with the eps value. If the random number is smaller than eps, then the model will conduct random action. Therefore, small decay will cause the model to spend more time on conducting random actions. Here is a comparing graph about the decay rate of 0.9999975 vs. decay rate of 0.99999975 (ten times smaller).

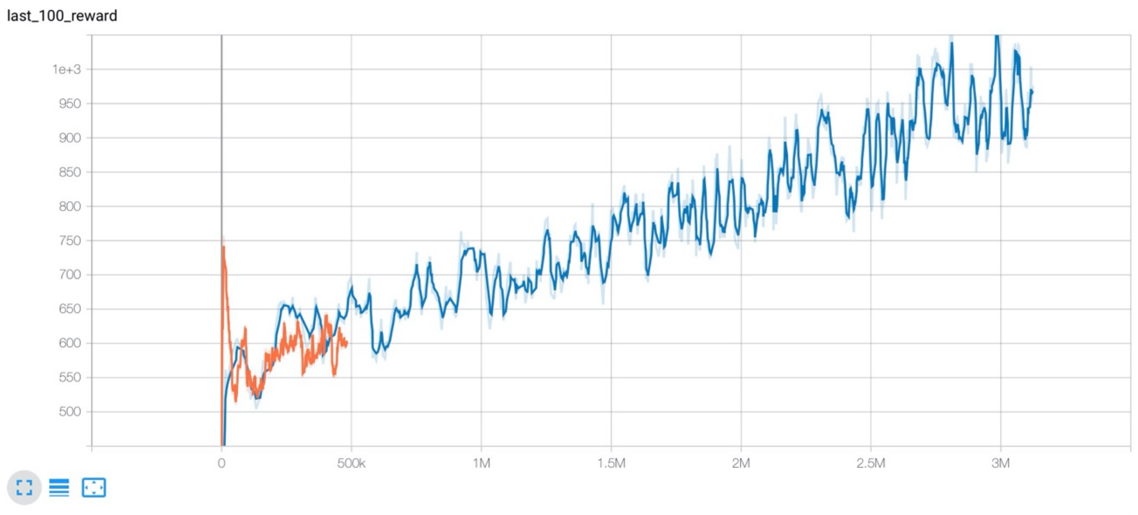


Figure 3. Comparing eps (from TensorBoard)

The orange line represents the small decay model, and the blue line represents the larger decay model. Due to the HiPer Gator traffic, I did not have the chance to finish my training on the small decay model. Theoretically, models with smaller decay will have a lower reward gaining slop because it needs more time to explore and train. And after a while, the model with smaller decay should perform a better reward slope since it includes more possibilities to reach the optimal rewards.

5.2. Stability and robustness

To make sure our model is stable, I introduce a random action possibility parameter to interfere with the model when playing the game. By changing agent_play.play(0.00) to agent_play.play(0.02), we add 2% random action possibility. And the table below summarizes the result under this condition.

Table 2. Reward score under interference

Model	Mean Reward Score for 100 Games
Without interference	3072.0
With 2% random	3067.4

According to the table above, the difference in mean reward between the two models is about 4.6, which is relatively small. As a result, I believe my model can survive in some extreme situations.

5.3. SIMPLE_MOVEMENT vs. COMPLEX_MOVEMENT

In gym-super-mario-bros pack, SIMPLE_MOVEMENT includes seven actions, while COMPLEX_MOVEMENT includes 12 actions. In my understanding, if the model uses complex movements, the reward slope would be shallower, and it will take a longer time to reach high rewards because complex movement could increase the workload of the model. However, by comparing the last_100 reward graph generated by TensorBoard, there's not much difference between the two models. Therefore, we can not conclude that the complexity of movements will affect the model in this case.

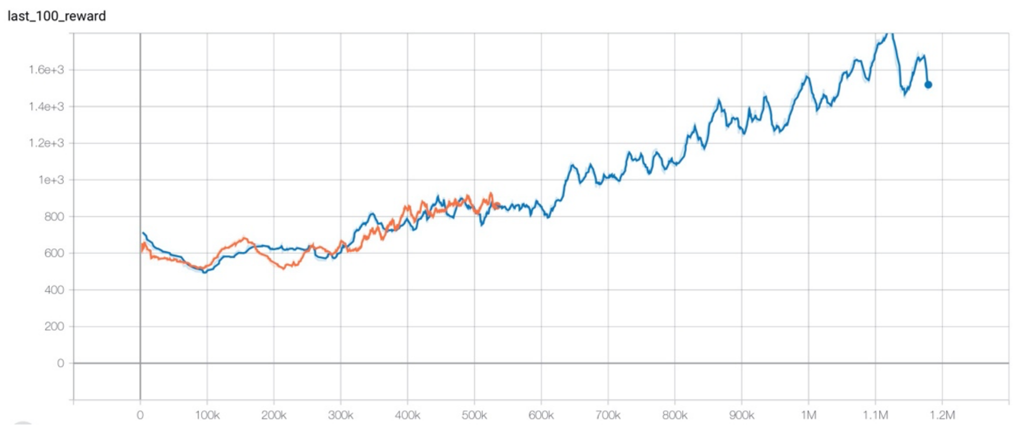


Figure 4. Comparing simple and complex movement (from TensorBoard)

6. Conclusion

By using DQN with replay buffer and DQN Agent, I successfully create a deep reinforcement model that can pass the Mario game. Also, I use different decay rates, random action possibilities, and complexity of movements to evaluate my model, seeking better performance and stability. I appreciate the deep learning course to let me have the chance to explore machine learning concepts and do hands-on experiments.

Reference

[1] Mnih, Volodymyr, et al. Playing Atari with Deep Reinforcement Learning. https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf.

[2] John N Tsitsiklis and Benjamin Van Roy. An analysis of temporal-difference learning with function approximation. Automatic Control, IEEE Transactions on, 42(5):674–690, 1997.

Code Reference

[1] https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html

[2] https://github.com/PacktPublishing/Deep-Reinforcement-Learning-Hands-On/blob/master/Chapter06/lib/wrappers.py