Deep Q-Learning to play Pong

AML Final project - Fall 2018
STAT5242

Xiaodi Sun
xs2315
Wenyi Lyu
wl2618
Chi Ma
cm3700
Xinyu Zang
xz2654

Contents

1. Introduction

Reinforcement learning (RL) is a field of machine learning, that is dedicated to agents acting in the environment in order to maximize some cumulative reward.

Actions of an agent are rewarded by the environment to reinforce correct behavior of the agent. In this case, the agent is able to automatically learn the optimal strategy. Methods of reinforcement learning appeared to be useful in many areas where AI is involved: robotics, industrial manufacturing, and video games. RL usually solves sequential decision-making problems. We follow the success of experiment in order to test the applicability of episodic control RL algorithm in Pong video game.

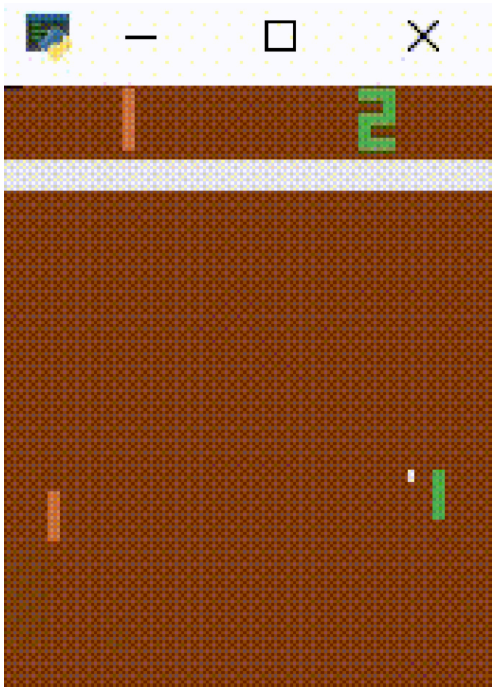
2. The Goal of the Project

The game of Pong, released in 1972, is a two-dimensional sports game that simulates table tennis. The player controls an in-game paddle by moving it vertically across the left side of the screen and can compete against either a computer controlled opponent or another player controlling a second paddle on the opposing side. Players use the paddles to hit a ball back and forth. The aim is for each player to reach twenty-one points before the opponent; points are earned when one fails to return the ball to the other.

In our project, we tried to use Q-Learning to create an agent which will learn to play pong and ultimately will play better than a human by developing some new strategy to win this game.

Our project goal is to create an agent which will use Q-Learning (online learning) to make smart decisions on paddle movement.

Fig 1. The screenshot of the Pong game



The game comes with the built-in AI controller that gets high-level features such as position and speed of the ball, the position of his and opponent paddle as an input, and uses a rule-based approach to control the paddle.

In the game, we control the green paddle which is on the right side. When the ball passes our paddle and goes to end right, we get a reward of -1 for losing. If the ball crosses the opponent and reaches the left, we get a reward of +1 for winning. The game finishes if one of the players reaches 21 scores.

From our code, we can find that although we have 6 discrete action values, there are only three real actions. 0&1 stands for doing nothing. 2&4 refers to going up. 3&5 refers to going down.

So the definition of the system in the reinforcement learning method is clear.
State is the screen of the game, which is a 210*160*3 image.
Action is going go up, go down and stay still.
Goal: maximize total reward.

3. Theory

3.1 Reinforcement Learning

Here are some definitions in the RL concept.

Agent: the one who is learning to win or gather benefits for himself (get more rewards)
Environment: the world which agent can interact with. Agent sees states and takes actions based on what he sees.
Reward: the agent gets rewards or punishments based on his actions. We can think of low reward or negative reward as punishment. Reward can be many things. Even the amount of time the agent stays alive could be considered a reward or score, usually, the reward is given to agent (agent does not think and decide what is a reward and what is not)

An RL agent interacts with an environment over time. At each time step t, the agent is situated in a state. The agent selects an action from some action space, following a policy. This policy is a probability distribution describing the agent behavior, i.e., a mapping from state to actions. Then the agent receives a scalar reward from the environment, and transitions to the next state according to the environment dynamics or model: reward function and state transition probability respectively. In an episodic RL problem, this process continues until the agent reaches a terminal state and then

restarts. The return is the discounted, accumulated reward with the discount factor. The agent aims to maximize the expectation of such long-term return from each state.

To determine agent preference over states, a value function is introduced as a prediction of the expected accumulated and discounted future reward. The action-value function Q(s, a) is the expected return for selecting action a in state s and following the policy afterward. An optimal action-value function Q(s, a) is the maximum action value achievable by any policy for state s and action a.

Temporal difference (TD) learning is the key idea in RL. It learns a value function Q(s) online directly from the experience with TD error, bootstrapping in a model-free and fully incremental way.

3.2 Epsilon-greedy

Epsilon-greedy is a simple mixture of exploration and exploitation.

In other words, we choose the action that can maximize the value function with the probability of 1-ϵ and randomly choose our action with the probability of ϵ.

3.3 Deep Q Networks

There are already many great works about reinforcement learning which make the computer learn how to play the game. One of the algorithms is Q-learning. Q-learning performs well in some games like Maze. The state-action space is not very large so that we can save them in memory. But here, the combination of the screen pixels and actions is so large that we can't save them in memory, and it takes too long to make the Q-table stable.

Fortunately, some talent people combine the deep learning and reinforcement learning to solve the problem, such as Deep Q Network method. DQN generalizes the approximation of the Q-value function rather than remembering the solutions.

4. Key Insight (Thought Process)

4.0 Original Model

At first, we use tensorflow to build the model based on the code we learned in class.

We build the model using the parameters as follows:

Environment: Pong-v0

Epsilon decay formula: $\epsilon = 0.01 + (1.0 - 0.01)e^{(-.001episode)}$

Batch size = 50

Gamma = 0.99
Learning rate =1e-4

The network we considered at first:

Table 1. The Structure of Original Network

Layer	Output Shape	Number of Parameters
Input layer	(None, 1, 84, 84)	0
flatten layer	(None, 7056)	0
Dense layer #1	(None, 512)	1411200
Dense layer #2	(None, 1)	512

When we test our network, we find that it is hard to debug the code. Then we use **pytorch**, which is easy to debug.

4.1 Model Increment 1—Network Construction

We have constructed a simple network model to evaluate Action-state value, which has only two fully connected layers with 1411712 parameters in total. We also defined MSE loss and used random numbers to test whether it could run on the computer. It is found that the MSE calculated by trained model is around 4.

Then we add three convolutional layers to construct a convolutional network. We calculated the MSE loss for the new model, which is 0.3. The result shows that the new model decreased MSE significantly compared to the previous one after training.

Table 2. The Structure of Convolutional Network

Layer	Output Shape	Number of Parameters
Input layer	(None, 1, 84, 84)	0
Conv2d layer #1	(None, 32, 20, 20)	2048
Conv2d layer #2	(None, 64, 9, 9)	32768
Conv2d layer #3	(None, 64, 7, 7)	36864
flatten layer	(None, 3136)	0
Dense layer #1	(None, 512)	1605632
Dense layer #2	(None, 1)	512

Thus, we used the new network model to construct DQN model.

4.2 Model Increment 2—Environment Wrapper

We used old Atari wrappers from OpenAI baselines project at the beginning, the running speed is dramatically slow. Then we searched online and found that the wrappers applied to the environment are very important for both speed and convergence. So we used another new environment, which is "PongNoFrameskip-v4" and the new method to **wrap the environment**, which can be found in env_wrapper.py.

4.3 Model Increment 3—Epsilon Decay Rate

After using the new environment wrapper, we think of another method to increase the learning rate. We used $\epsilon = 0.02 + (1 - 0.02)e^{(-episode/30000)}$ instead of $\epsilon = 0.01 + (1.0 - 0.01)e^{(-0.001episode)}$ to calculate epsilon value in each round. By using new epsilon decay formula, we decreased the epsilon decay rate and changed the epsilon decay from 1-0.01 to 1-0.02 to explore more possibilities during training process. As a result, the convergence speed increased a lot.

4.4 Model Increment 4—Batch Size

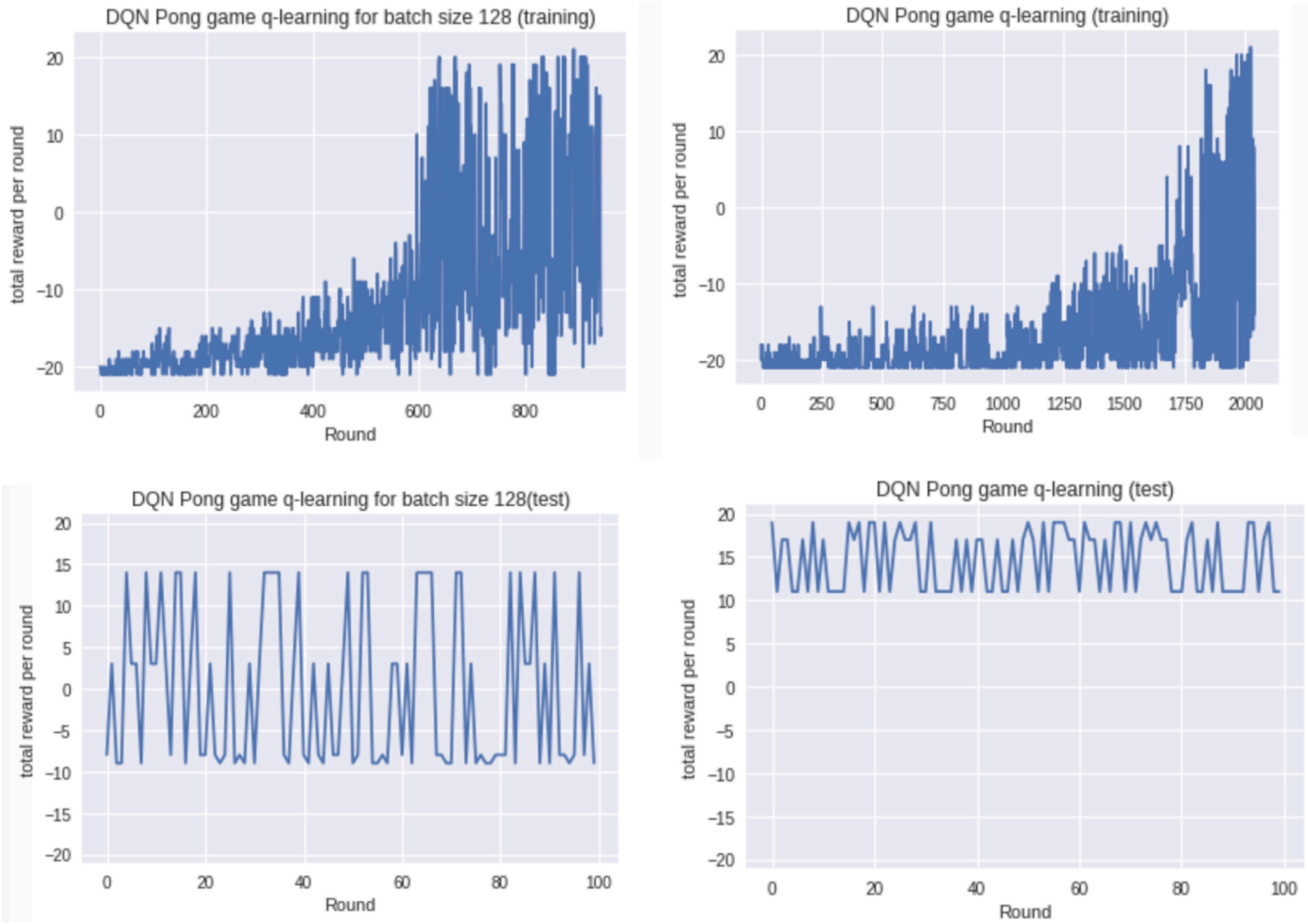
As for the batch size, we also tried many times to make the model perform well. Take batch size 128 and 32 as an example, we first tried to use 128 as our batch size. The log rewards plot of the training process shows that after about 600 rounds, the rewards become positive, which may give the proof that our model has converged. However, when we use this converged model to play the game, the model has a bad performance. The winning rate of the model is only 0.48. After we tweaked other parameters, the model did not perform well. Then we use batch size 32 instead. In the log rewards plot of the training process, the rewards become positive after about thousands of rounds, which indicates the training speed is much slower than the 128 batch size model. But when we use this 32 batch size model to play the game, winning rate becomes 1, which is significantly higher than 128 batch size model.

Increasing the batch size could increase the speed of the convergence, yet it can easily overfit to the current data.

Table 3. Comparison of models using different batch sizes

Batch size	Winning rate	Convergence speed
128	0.48	quick
32	1	slow

Fig 2. Comparison Plots of Different batch sizes (128 vs 32)



5. Model Structure

5.1 DQN Model

We use three convolutional layers and two fully connected layers to construct the DQN model. The filter sizes for the three convolutional layers are respectively 8*8,

4*4 and 3*3. By using the convolutional network model, we dramatically decrease the number of parameters we need to train.

The calculation steps for the convolutional network model can be concluded as the following two steps:
First, we compute the hidden layer values by simply finding the dot product of the filter matrix and the observation_matrix (or hidden layers).
Next, we apply a nonlinear thresholding function to those hidden layer values - in this case just a simple ReLU. At a high level, this introduces the nonlinearities that makes our network capable of computing nonlinear functions rather than just simple linear ones.

The **input layer** of the network is the observation, which is the preprocessed environment.
The **output layer** of this network is the Q values for the three different actions calculated from the network.

The loss function we use here is the squared loss between real Q values and expected Q values: $(Q\_real - Q\_expected)^2$. After defining the loss function, we can use the loss function to calculate gradient and optimize our DQN model.

Fig 3. The Structure of the DQN Model

```
Pong(
  (conv): Sequential(
    (0): Conv2d(1, 32, kernel_size=(8, 8), stride=(4, 4))
    (1): ReLU()
    (2): Conv2d(32, 64, kernel_size=(4, 4), stride=(2, 2))
    (3): ReLU()
    (4): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1))
    (5): ReLU()
  )
  (fc): Sequential(
    (0): Linear(in_features=3136, out_features=512, bias=True)
    (1): ReLU()
    (2): Linear(in_features=512, out_features=6, bias=True)
  )
)
```

5.2 Training Process

The first step of our algorithm is preprocessing the image of the game that OpenAI Gym passed to us.

In the env.wrapper code, we convert the image format from RGB to GREY because the color is not important to us and GREY format is easy to train. In addition, we resize the image from 210x160x3 to 1x84x84.

After preprocessing the environment, the agent can choose action based on epsilon greedy policy. After the agent takes the action, the environment will update and give us the reward. Then we will add the reward to the total rewards and record the information of the observation, action, reward and the new observation into the buffer. If the game is finished, we will reset the total rewards to 0 and reset the environment.

After the size of the buffer is larger than the **Replay_initial** parameter, we will begin to train the model. Each time the agent takes an action, we will train the model once. For every 10000 frame sequences, we will save the trained model once and record it in the log text.

5.3 Hyper Parameters

There are many hyper parameters we need to set before we begin to train the DQN model. Based on the experience from the previous papers and our own trails, we choose the values for the following hyper parameters.

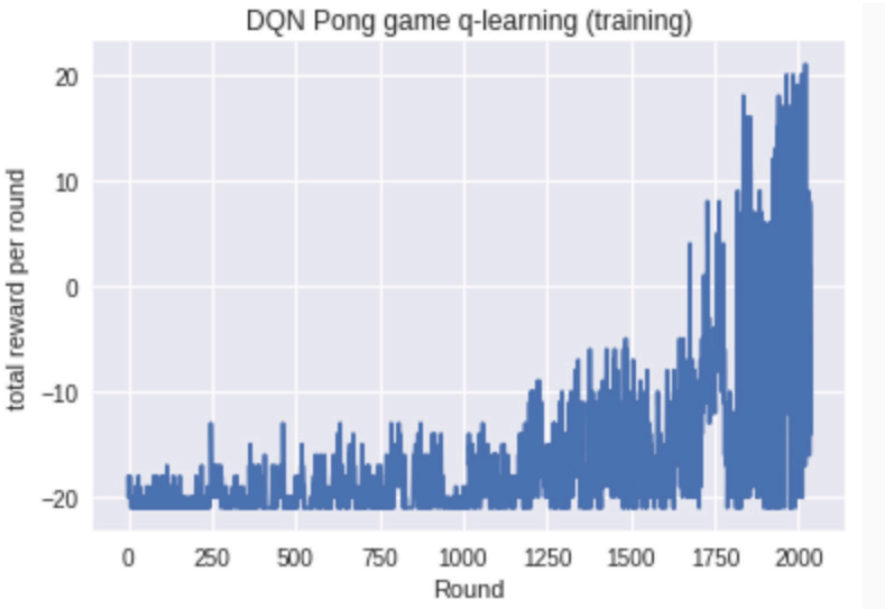
Batch_size: How many rounds we play before updating the weights of our network.
Gamma: The discount factor we use to discount the effect of old actions on the final result.
Epsilon_begin: Exploration rates at the beginning
Epsilon_end: Exploration rates at the end
Epsilon_decay: The speed of decreasing the exploration rates
Replay_buffer_size: The maximum number of data that our buffer can hold. If the length of the data is larger than this size, we should drop the oldest data point to make room for new data.
Replay_initial: The size of the first buffer before we begin to train from the buffer.
Learning_rate: The rate at which we learn from our results to compute the new weights. A higher rate means we react more to results and a lower rate means we don’t react as strongly to each result.

6. Result

6.1 Training model

We use batch size 32 to inspect the training process of the model. We write a script to save the model in two files and log rewards from time to time. Below are two plots based on those rewards.

Fig 4. DQN Pong game q-learning (training)



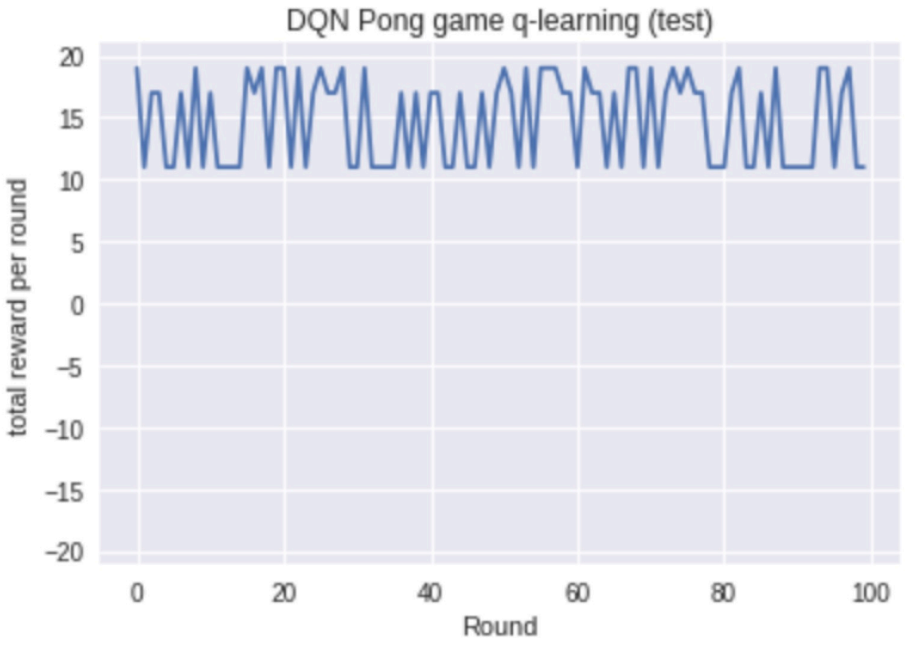
From the above figure, we can figure out that at the beginning, the total rewards of the agent in one episode are around -20, which means the agent is losing the game. The reason is that with the relatively large epsilon, the agent is likely to explore more possibilities. However, with the increase in the number of the game round, the rewards become larger. After around 1750 steps for batch size 32, the rewards turn positive and even reach 20, which means the agent begins to win. It can be explained that with the decay of the epsilon, the agent is prone to exploit rather than explore, leading to the higher rewards.

6.2 Test Result

We give 100 test rounds and plot the rewards for each round. In the end, we calculate the winning rate for each model, which can be calculated by winning time divided by test rounds.

The plot of test result shows that the rewards are all positive, which indicates the agent wins all the games for 100 rounds. The winning rate is 1, which confirms with the plots. Therefore, we can conclude that the agent **has learned some tricks to defend and even win.** However, the mean reward is around 15, which means we cannot hit the ball back every time.

Fig 5. DQN Pong game q-learning (test)



6.3 Key Success

Although there are many resources online to solve the Pong game problem, all of them need to train the model for at least 5000 rounds to get the converged model. After we conducted all the model increments described before, we get the model converged after 2000 rounds.

7. Limitations&Next Steps

7.1 Limitations

1. **The training process is too long.** With GPU, it takes us a whole day to train the model.
2. **The score result is not as high as we expected.** The mean score of our model does not reach 21, which means the trained agent does not always hit the ball back every time.

7.2 Next Steps

Here are some ideas to improve the training process and agents:

1. Generate simulations of batches in parallel. I tried using multithreading to generate and play batches of games at the same time to increase the speed of the training process. The problem was that the backend library for the game had issues creating environments in multiple threads. Maybe you can overcome this by implementing a multiprocess method.

2. Giving a reward to the agent for catching the ball could be good for not losing (for example a +0.5 reward). Now we are just giving a positive reward to agent for scoring. If it catches the ball and then gets a -1 reward, it will think it is probably bad to catch the ball.

3. Use a different gamma to train the model. In our model, we use 0.99 as gamma. We may try other values for the gamma parameter, for example, 0.9. This may improve the training speed.

8. References

1. Wikipedia. https://en.m.wikipedia.org/wiki/Pong

2. Jonathan Hui: Medium. https://medium.com/@jonathan_hui/rl-dqn-deep-q-network-e207751f7ae4

3. Xiaowen Shi: Jianshu. https://www.jianshu.com/p/10930c371cac

4. Andrej Karpathy: github. http://karpathy.github.io/2016/05/31/rl/

5. Timothy Lee: github. https://github.com/openai/baselines

6. higgsfield: github. https://github.com/higgsfield/RL-Adventure

7. Ilya Makarov, Andrej Kashin, and Alisa Korinevskaya: Learning to Play Pong Video Game via Deep Reinforcement Learning. Russian Science Foundation, 17-11-01294

8. Lior Motorin, Peleg Tuchman: Using Q-Learning to play Pong.

9. Soroush: github. https://github.com/thinkingparticle/deep_rl_pong_keras

10. Dhruv Parthasarathy, Medium. https://medium.com/@dhruvp/how-to-write-a-neural-network-to-play-pong-from-scratch-956b57d4f6e0

11. Max Lapan, Medium. https://medium.com/mlreview/speeding-up-dqn-on-pytorch-solving-pong-in-30-minutes-81a1bd2dff55