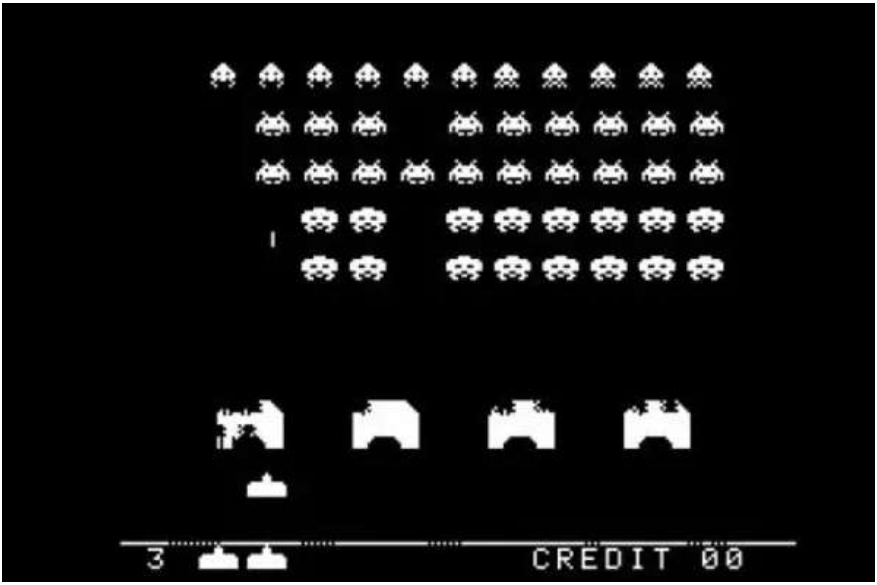
Deep Q-Learning with simple atari game

Using DQN method to teach a computer to play classic atari game—Space Invaders



Space Invaders

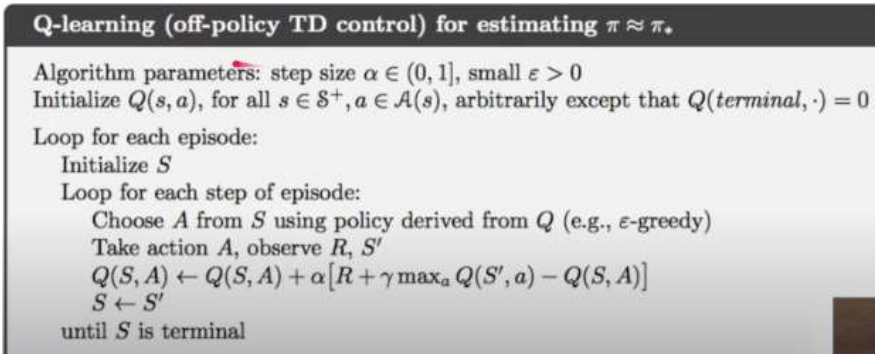
Reinforcement Learning and Game

As one of the three major machine learning methods, reinforcement learning has an obvious feature that it has biological and

psychological basis, and is based on control theory and statistics.

Reinforcement learning regards learning as a trial and evaluation process. The agent chooses an action for the environment, environment accepts the action and then generates a reinforcement signal (reward or punishment) to feed back to the agent. After receiving the signal, agent adjusts its strategy and chooses the next action.This process will repeat continuously, and finally the agent can always output correct action.

Obviously, because of this model of reinforcement learning, the application of reinforcement learning to games is very natural. An important reason is that game can quickly generate a large amount of naturally labeled (state-action-reward) data, which are high-quality training materials for reinforcement learning.



Reinfocement method : Q-Learning

Why DQN ?

As the game becomes more and more complex, more and more information is required to describe the current situation of the game, the corresponding state space becomes very large and most states are rarely observed, the estimation of Q table will take a lot of time and difficult to converge. Moreover, for a large number of unobserved possible states, we also hope to be able to estimate Q value of them. This is what DQN solves， Neural Network is very good at extracting good features from structured data, therefore, we can use NN to approximate Q function.

At the same time, states defined by humans may miss elements which can also affect rewards.Obviously, it is a better choice to use all pixel information of the picture as the state, which contains all information in this scene.

Environment

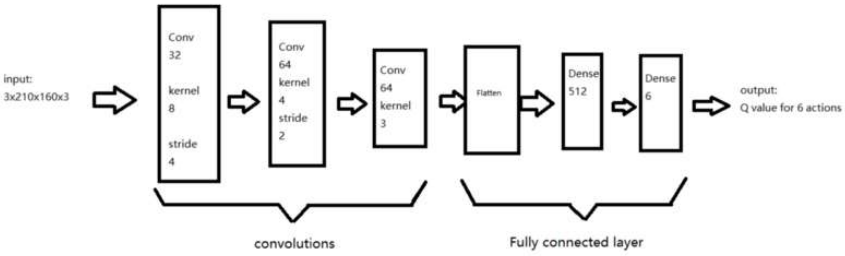
Game Name : Space Invaders

Game Version : gym.atari.SpaceInvaders-v4 (will not repeat the previous action, only execute the action given by the agent)

State : 3 channels pixel information 3 frames (3 x 210 x 160 x 3)

Action : ['NOOP', 'FIRE', 'RIGHT', 'LEFT', 'RIGHTFIRE', 'LEFTFIRE']

Q-Network



Q-Network Model structure

The main body of this Q-Network consists of three Conv2D convolutional layers and two fully connected layers, they are transitioned through a Flatten layer. Their basic information is shown in the figure above, and the relevant codes are as follows

```
def build_model(height, width, channels, actions):
    model = Sequential()
    model.add(Convolution2D(32, (8,8), strides=(4,4), activation='relu',
                            input_shape=(3,height, width, channels)))
    model.add(Convolution2D(64, (4,4), strides=(2,2), activation='relu'))
    model.add(Convolution2D(64, (3,3),  activation='relu'))
    model.add(Flatten())
    model.add(Dense(512, activation='relu'))
    model.add(Dense(actions, activation='linear'))
    return model
```

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_6 (Conv2D)            (None, 3, 51, 39, 32)     6176
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 3, 24, 18, 64)     32832
_________________________________________________________________
conv2d_8 (Conv2D)            (None, 3, 22, 16, 64)     36928
_________________________________________________________________
flatten_2 (Flatten)          (None, 67584)             0
_________________________________________________________________
dense_9 (Dense)              (None, 512)               34603520
_________________________________________________________________
dense_10 (Dense)             (None, 6)                 3078
=================================================================
Total params: 34,682,534
Trainable params: 34,682,534
Non-trainable params: 0
_________________________________________________________________
```

Network Summary

This network receives the pixel image information and outputs the Q value corresponding to each action. At the same time, this network also has enough variables to allow various factors related to the Q value to be considered.

DQN Agent and Hyperparameters

The agent in this experiment is directly constructed by the DQNAgent in the rl.agent package.

```
from rl.agents import DQNAgent
def build_agent(model, actions):
    policy = LinearAnnealedPolicy(EpsGreedyQPolicy(), attr='eps',
                                  value_max=epsilon, value_min=min_epsilon,
                                  value_test=.2, nb_steps=max_steps)
    memory = SequentialMemory(limit=1000, window_length=3)
    dqn = DQNAgent(model=model, memory=memory, policy=policy,
                   enable_dueling_network=True, dueling_type='avg',
                    nb_actions=actions, nb_steps_warmup=2000,
                    gamma=Gamma
                   )
    return dqn
dqn = build_agent(model, actions)
dqn.compile(Adam(lr=learning_rate))
```

Policy : Epsilon Greedy Policy

Randomly select all actions with epsilon probability, and greedily select actions with 1-epsilon probability.Basic but effective.

Memory : size=1000, window length = 3

The hyperparameters related to this model are as follows.

total_episodes : 10000 (due to my computer, it is not large, but also can make some progress)

total_test_episodes : 10 (test 10 times after training)

max_steps : 10000 (max steps)

learning_rate : 0.01(using a low learning rate ensures we don't miss any local minima, but also means we will take longer to converge)

Gamma : 0.99(high discount rate)

Max_epsilon : 1

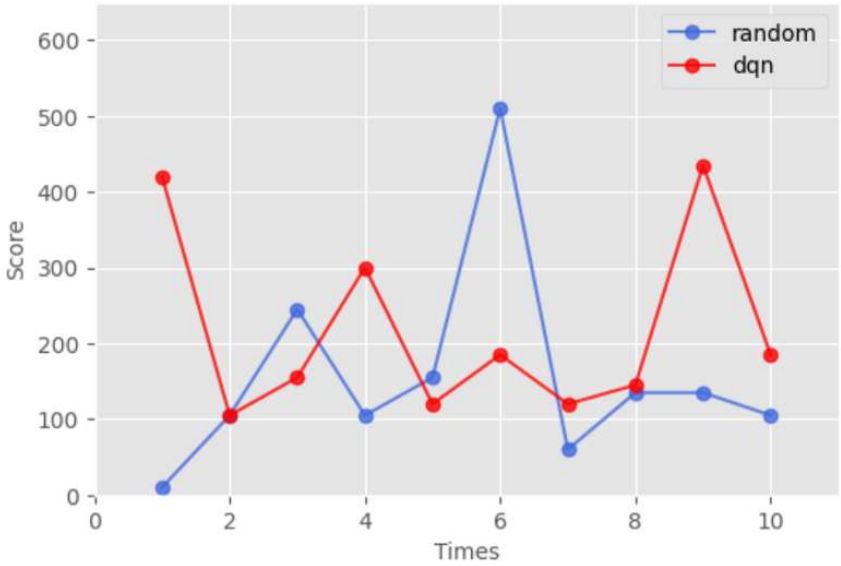
Min_epsilon : 0.1

Result

Because of the small number of training steps, it is difficult to evaluate this model in terms of high scores and stability.

At the same time, because this game has six actions, it is difficult to achieve a good score if you choose action randomly.

So I'm going to prove that the computer did learn something in this experiment by comparing it with the random selection strategy. The result is as follows.



It can be seen that most of the time the trained agent is better than random.

This is further evidenced by the average score.

DQN : 217

Random: 146.5

Future Work

1 : More training steps

Restricted by the experimental conditions, I only performed 10,000 steps of training this time. I believe that after more steps of training, we can get better result.

2 : More reasonable hyperparameter values

Because the number of training steps is not enough, we cannot clearly see the effect of hyperparameters, but when the number of training steps increases, the impact of each hyperparameter on the results will be more significant, and we should make corresponding adjustments at that time.

3 : More personalized agents and environments

In this experiment, I fully used the agent and environment in the package, only changed some hyperparameters, which may cause the result to be not perfect. Maybe this can be solved by customizing the agent and environment in the next experiment.

Reference

[1] : *Deep learning guide* *https://zhuanlan.zhihu.com/p/498713060*

[2]: ***D****ense layer in Keras*
https://blog.csdn.net/weixin_44551646/article/details/112911215

[3]: Deep Reinforcement Learning for Atari Games Python Tutorial | AI Plays Space Invaders https://www.youtube.com/watch?v=hCeJeq8U0lo

[4]: About installing gym.Atari at windows 10
https://zhuanlan.zhihu.com/p/523895071

MIT License