Mid-term Assignments

Authorship

`宋丽铭 20337234`

`李亚茹 20337066`

开源仓库链接

【Brief Introduction】

Atari Breakout is a brick playing arcade game developed and released by Atari. It controls the left and right movement of the board and bounces the ball onto it to eliminate bricks.

There are 8 rows of bricks in total, and each two rows of bricks have different colors. The bottom to top colors are yellow, green, orange, and red. After the game starts, the player must control a platform to move left and right to rebound a ball. When the ball hits the brick, the brick will disappear and the ball will bounce back. When the player fails to use the platform rebound ball, the player loses that round. When the player loses 3 times in a row (5 times used here), the player will lose the whole game. The player's goal in the game is to remove all bricks.

【Principle】

1 Reinforcement Learning

1.1 What is reinforcement learning?

> Reinforcement Learning (RL) is one of the paradigms and methodologies of machine learning, which is used to describe and solve the problem that agents use learning strategies to maximize returns or achieve specific goals in the interaction process with the environment.

1.2 Components of reinforcement learning

- Agent(智能体、机器人、代理): The main body of reinforcement learning is agent, which is sometimes translated as "代理", and is collectively referred to as "智能体" here.
- Environment(环境)： The overall background of the game is the environment
- State(状态): state of the current environment and agent
- Action(行动): Based on the current state, what actions can the agent take, such as left or right, up or down; Action is strongly linked to the state. For example, many positions in the above figure have partitions. It is obvious that the agent cannot go left or right under this state, but only up and down;
- Reward(奖励): When the agent takes a specific action in the current state, it will get some feedback from the environment, that is, Reward. This is generally referred to as "Reward". Although Reward is translated into Chinese to mean "reward", in fact, in reinforcement learning, Reward only represents "feedback" given by the environment, which may be a reward or a punishment.

1.3 Main characteristics of reinforcement learning

Trial and error learning: Reinforcement learning requires the training object to constantly interact with the environment and summarize the best behavior decision at each step through trial and error. The whole process has no guidance, only cold feedback. All learning is based on environmental feedback, and the training objects adjust their behavior decisions.

Delayed feedback: In the process of reinforcement learning training, the "trial and error" behavior of the training object gets feedback from the environment. Sometimes, it may be necessary to wait until the whole training is completed to get a feedback, such as Game Over or Win. Certainly, in this case, we usually break down the feedback during training and try to break it down to each step.

Time is an important factor in reinforcement learning: A series of environmental state changes and environmental feedback of reinforcement learning are strongly linked with time. The whole training process of reinforcement learning changes with time, and the state feedback is also changing constantly, so time is an important factor of reinforcement learning.

The current behavior affects the data received subsequently: The reason why this feature is proposed separately is also to distinguish it from supervised learning and semi-supervised learning. In supervised learning and semi-supervised learning, each training data is independent and has no relationship with each other. But this is not the case in reinforcement learning. The current state and the actions taken will affect the state received in the next step. There is a certain correlation between data and data.

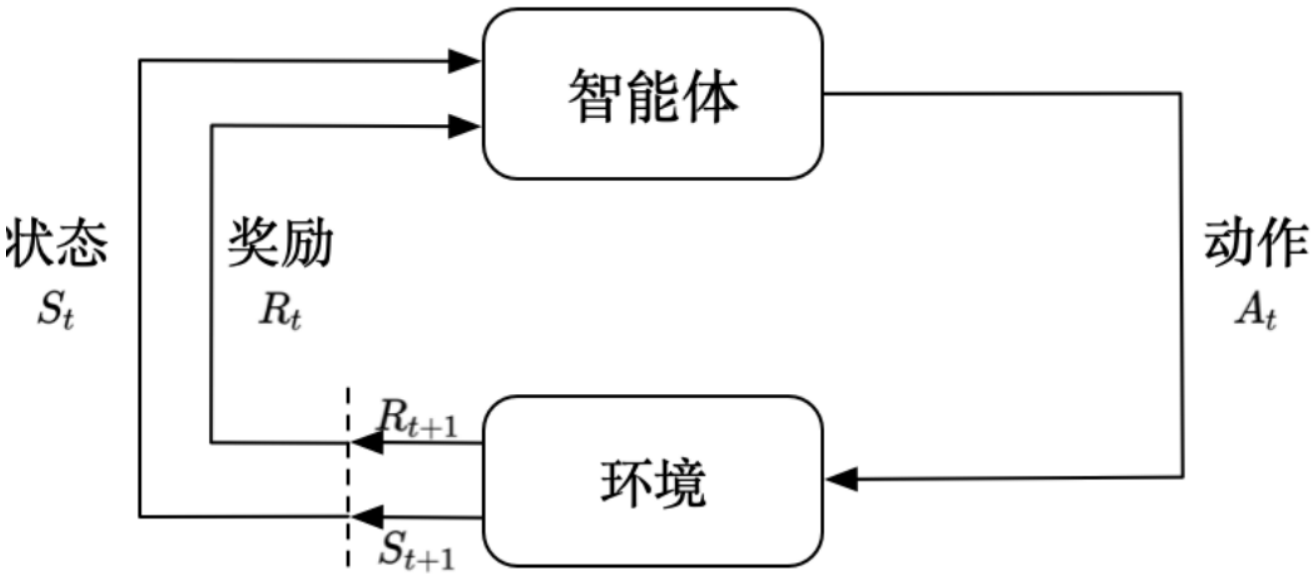


图 1.1　强化学习示意图

2 Q-learning

Q-learning algorithm is a value based reinforcement learning algorithm. Q is the abbreviation of quality. The Q function Q (state, action) indicates the quality of executing action under the state state, that is, how much Q-value can be obtained. The goal of the algorithm is to maximize the Q value, and maximize the expected forward by selecting the best action among all possible actions under the state state.

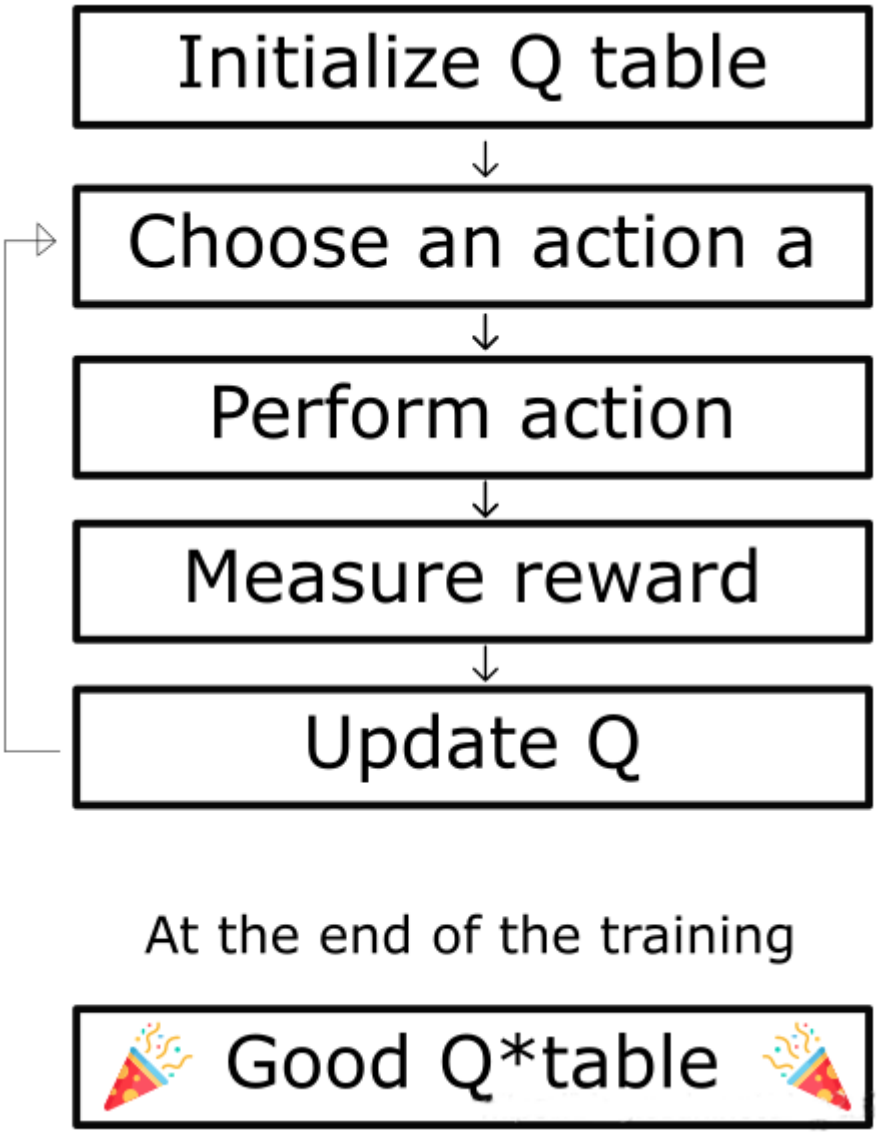
The Q-learning algorithm uses Q-table to record the estimated Q value of different actions under different states. Before exploring the environment, Q-table will be initialized randomly. When the agent explores in the environment, it will use the Bellman equation to update Q (s, a) iteratively. With the number of iterations increasing, the agent will understand the environment more and more, and the Q function can be fitted better and better, until it converges or reaches the set number of iterations.

Update formula:

α is the learning rate, γ is the discount factor, it is updated by time difference method.

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma max_{a'} Q(s',a') - Q(s,a)]$$

Multiply the maximum $Q(s',a')$value in the next state s' by the decay γ In addition, the true return value is the most realistic for Q, and the $Q(s,a)$in the past Q table is used as the Q estimate. Q-learning's algorithm process:



3 DQN (Deep Q Network)

3.1 Why DQN is needed?

The core of Q-learning is the Q table, which provides guidance for actions by establishing Q tables. However, this is applicable when the state and action spaces are discrete and the dimensions are not high. When the state and action spaces are high-dimensional continuous, the Q table will become very huge, which is unrealistic for maintaining the Q table and searching. Imagine what the scenario would be if AlphaGo used Q-learning. The probability level of Go is 10 ^ 170. Such a huge Q table has lost its value.

Q table cannot be solved. Therefore, consider a method of value function approximation to realize that only knowing S or A in advance each time can obtain its corresponding Q value in real time. Depth learning works well in complex feature extraction, so DQN uses depth neural network as a tool for value function approximation, and combines RL and DL to obtain DQN. This brings two benefits:

1. Only the network structure and parameters of DL need to be stored
2. Similar inputs will get similar outputs, with stronger generalization ability

3.2 DQN (NIPS 2013)

Q-learning algorithm has existed for a long time, but its combination with deep learning was realized in the paper "Playing Atari with Deep Reinformation Learning" published by DeepMind in 2013. This paper creatively integrates RL and DL, proposes the Experience Replay mechanism and the Fixed Q-Target, and realizes part of the Atari game control, even surpassing the human level. The innovation of this paper is the Experience Replay and Fixed Q-Target just mentioned.

Experience Replay

Experience replay: we store the agent's experience at each time step in the data, and gather many rounds into one replay memory, with the dataset $D = e_1, .., e_N$, where $e_t = (s_t, a_t, r_t, s_{t+1})$。 In the internal loop of the algorithm, we will randomly sample some data and use the samples as the input of the neural network to update the parameters of the neural network. The advantages of using experience replay are:

- Each step of experience may be used in many weight updates, which will improve the efficiency of data use;
- In the game, the correlation between each sample is strong, and adjacent samples do not meet the independent premise. What the machine learns from continuous samples is invalid. Using experience playback is equivalent to adding randomness to the sample, but randomness will destroy these correlations, thereby reducing the variance of update.

Fixed-Q-Target

Another innovation of this paper is that instead of using only one neural network for training, two neural networks with identical structures are designed, namely, the target network (Q target) and the evaluation network (Q predicate). But the parameters used in Q-target network are old ones, while those used in Q-predicate network are new ones. In the training process, the parameters of Q-predicate are constantly updated and optimized, and the parameters of the evaluation network are only assigned to Q-target at a certain interval. Q predicate gives the predicted q_{value} according to the input state s. Q target gives q_{next} according to the input of s_{next}. Then substitute q_{next} into the core recursive formula of Q-learning to get q_{target}. Finally, $loss(q_{value}, q_{target})$ back-propagates and optimizes the parameters to continuously get a better neural network, so after repeated training and optimization, the agent can basically master the action decisions under various states.

Pseudocode:

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

3.3 Nature DQN

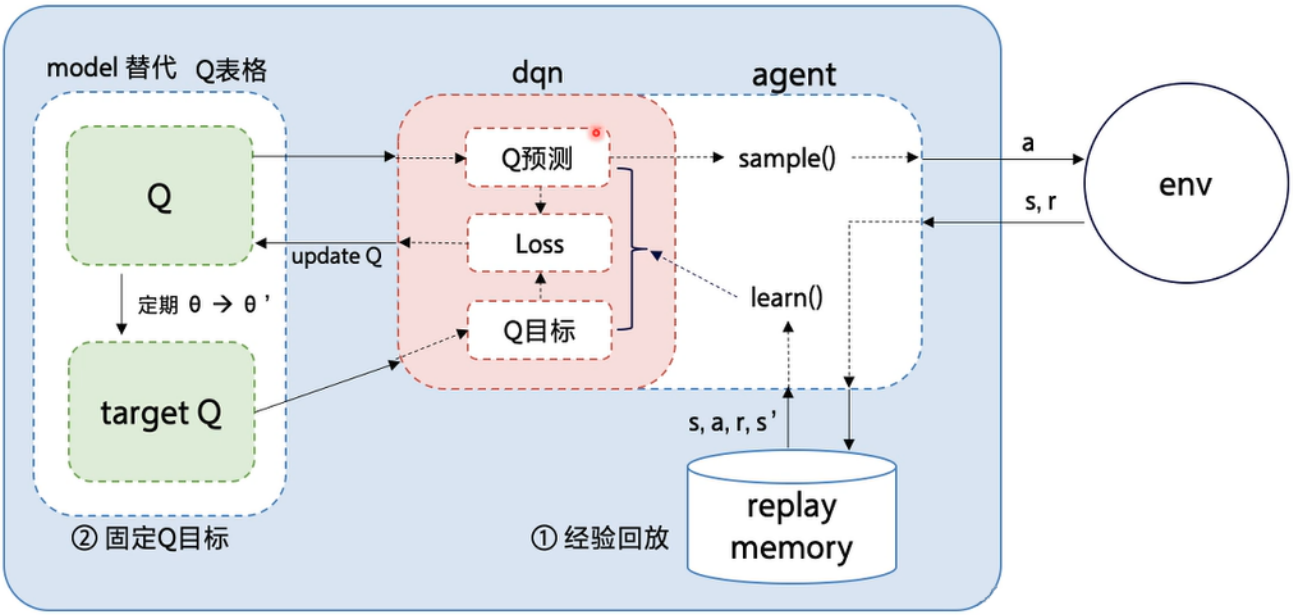
From the above DQN, we can see that the same network Q is used to calculate the target value y_j and the current value, so the network Q we need to train is used to calculate the target value y_j, and then we use the target value y_j to update the parameters of the network Q. In this way, the dependence of the two is too strong, which is not conducive to the convergence of the algorithm.

Therefore, in Nature DQN, it is proposed to use two networks. One original network Q is used to select actions and update parameters, and the other target network Q' is only used to calculate the target value y_j . Here, the parameters of the target network Q' will not be updated iteratively, but copied from the original network Q at regular intervals. Therefore, the structures of the two networks need to be completely consistent, otherwise, parameter replication cannot be performed. We simply introduce the difference between DQN algorithm and DQN algorithm from the formula. There are two networks Q and Q', and their corresponding parameters are $theta$ and $theta'$, respectively. The update frequency here can be set by yourself, and the update parameter is $theta' = theta$ directly.

In addition, we also need to change the calculation formula of target value y_j as follows:

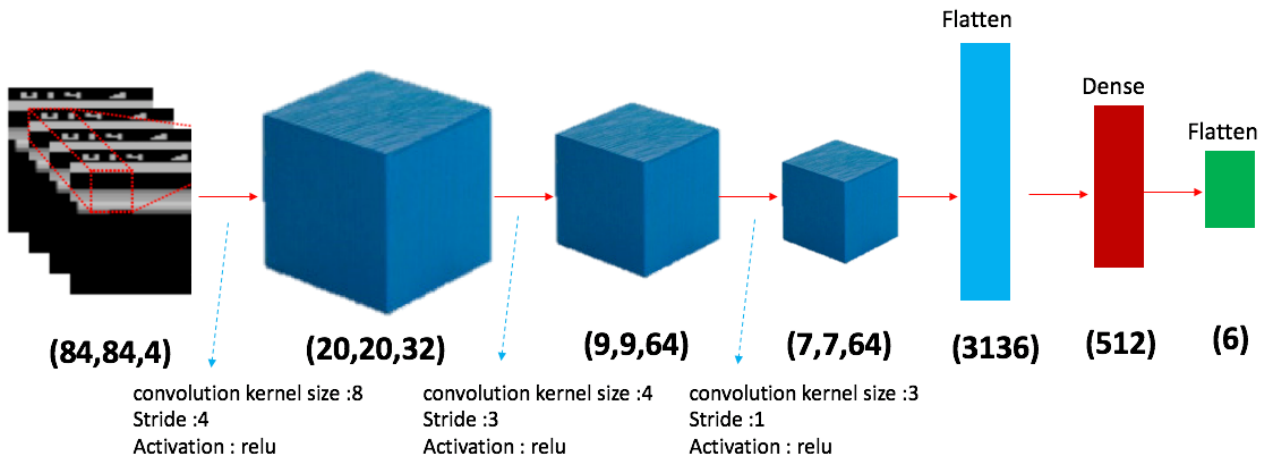
$$y_j = \begin{cases} r_j & \text{for terminal } \varnothing_{j+1} \\ r_j + \gamma max_{a'} Q(\varnothing_j, a_j; \theta') & \text{for non-terminal } \varnothing_{j+1} \end{cases}$$

The rest are identical to DQN.



In the Breakout game of this question, the input image is 210 * 160 * 3. We will process it slightly. After removing the unnecessary pixels on the edge, we will reduce the order of sampling to grayscale, and take the 84 * 84 * 4 image as the input of the algorithm.

The input of the neural network is not a single frame image, but the latest four consecutive frames. This is also easy to understand, because it adds time series. For the brick playing game, if only one frame is used as input, although the bricks are in the same position, they may move in several directions, and the agent cannot judge its value. But if we add the last few frames, the agent can judge which direction it is moving according to the time before and after, and the state is complete. The following figure shows the processing structure of graph using convolutional neural network:



4 Code Structure

Base implementation: https://gitee.com/goluke/dqn-breakout

4.1 `main.py`

`main.py` defines the required constant

```
GAMMA = 0.99            # discount factor
```

```
GLOBAL_SEED = 0          # global seed initialize
MEM_SIZE = 100_000       # memory size
RENDER = False           # if true, render gameplay frames
STACK_SIZE = 4           # stack size

EPS_START = 1            # starting epsilon for epsilon-greedy alogrithm
EPS_END = 0.1            # after decay steps, spsilon will reach this and keep
EPS_DECAY = 1000000      # steps for epsilon to decay

BATCH_SIZE = 32          # batch size of TD-learning traning value network
POLICY_UPDATE = 4        # policy network update frequency
TARGET_UPDATE = 10_000   # target network update frequency
WARM_STEPS = 50_000      # warming steps before training
MAX_STEPS = 50_000_000   # max training steps
EVALUATE_FREQ = 100_000 # evaluate frequency
```

- `GAMMA` is the discount factor γ, set to `0.99`;
- `MEM_ SIZE` is the `capacity` in `ReplayMemory`;
- When `RENDER` is "TRUE", the game screen will be rendered every time the evaluation is made;
- `STACK_ SIZE` is `channels` in `ReplayMemory`;
- `EPS_ START` and `EPS_ END` is in `EPS_ Start` and end values of ϵ attenuation in DECAY step, and then ϵ Always in `EPS_ END`, it is worth mentioning that `EPS_ Start` will be `1`, but it is necessary to change it to a smaller value when the loaded model continues training, otherwise the performance of the loaded model cannot be well performed;
- `BATCH_ SIZE` is the number of samples when sampling from `ReplayMemory`;
- `POLICY_ UPDATE` is the update frequency of the policy network;
- `TARGET_ UPDATE` is the update frequency of the target network;
- `WARM_ STEPS` is to wait until there are enough records in `ReplayMemory` before lowering ϵ ;
- `MAX_ STEPS` is the number of training steps;
- `EVALUATE_ FREQ` is the frequency of evaluation.
  Then initialize the random number, computing device and environment `MyEnv` 、 agent `Agent` and `ReplayMemory`.

Note that setting `done` to `True` here is to initialize the environment and record the initial observation when starting training.

Then start to implement the **Nature DQN** algorithm mentioned above. In the loop, first judge whether a round has ended. If it is over, reset the environment state and queue the observation data for storage:

```
if done:
    observations, _, _ = env.reset()
    for obs in observations:
        obs_queue.append(obs)
```

Then judge whether the `Warming steps` have been passed. If so, set `training` to `True`, and attenuation will start ϵ:

```
training = len(memory) > WARM_STEPS
```

Observe the current state `state` and select the action `action` according to the state. Then get the new information `obs`, feedback `reward` and whether to end the game `done`:

```
state = env.make_state(obs_queue).to(device).float()
action = agent.run(state, training)
obs, reward, done = env.step(action)
```

Enter the observation into the team, and record the current status, action, feedback and whether it is over into `MemoryReplace`:

```
obs_queue.append(obs)
memory.push(env.make_folded_state(obs_queue), action, reward, done)
```

Update the policy network and the synchronization target network. The synchronization target network is to update the parameters of the target network to the parameters of the policy network:

```
if step % POLICY_UPDATE == 0 and training:
    agent.learn(memory, BATCH_SIZE)
if step % TARGET_UPDATE == 0:
    agent.sync()
```

Evaluate the current network, save the average feedback and trained strategy network, and end the game. If `RENDER` is `True`, the game screen will be rendered:

```
 if step % EVALUATE_FREQ == 0:
        avg_reward, frames = env.evaluate(obs_queue, agent, render=RENDER)
        with open("rewards.txt", "a") as fp:
            fp.write(f"{step//EVALUATE_FREQ:3d} {step:8d} {avg_reward:.1f}\n")
        if RENDER:
            prefix = f"eval_{step//EVALUATE_FREQ:03d}"
            os.mkdir(prefix)
            for ind, frame in enumerate(frames):
                with open(os.path.join(prefix, f"{ind:06d}.png"), "wb") as fp:
                    frame.save(fp, format="png")
        agent.save(os.path.join(
            SAVE_PREFIX, f"model_{step//EVALUATE_FREQ:03d}"))
        done = True
```

4.2 `utils_drl.py`

The agent `Agent` class is implemented in `utils_drl.py`, and the incoming parameters and two models are initialized. When the trained model is not imported, the model parameters are initialized. Load model parameters when passing in the trained model.

```
if restore is None:
            self.__policy.apply(DQN.init_weights)
        else:
            self.__policy.load_state_dict(torch.load(restore))
        self.__target.load_state_dict(self.__policy.state_dict())
        self.__optimizer = optim.Adam(
            self.__policy.parameters(),
            lr=0.0000625,
            eps=1.5e-4,
        )
        self.__target.eval()
```

There are four functions defined in the class `Agent`, as follows:

- `run()` selects an action according to ϵ−greedy policy;
- `learn()` updates the parameters of the neural network:

```
def learn(self, memory: ReplayMemory, batch_size: int) -> float:
    """learn trains the value network via TD-learning."""
    # 从memory中随机取样本
    state_batch, action_batch, reward_batch, next_batch, done_batch = \
            memory.sample(batch_size)
    # state预期的values
    values = self.__policy(state_batch.float()).gather(1, action_batch)

    # max_action = self.__policy(next_batch.float()).max(1)
    # values_next = self.__target(next_batch.float()).gather(1, max_action)

    # 下一个state预期的values
    values_next = self.__target(next_batch.float()).max(1).values.detach()
    # state现实的values=衰减因子gamma*values_next+reward
    expected = (self.__gamma * values_next.unsqueeze(1)) * \
            (1. - done_batch) + reward_batch
    # 损失
    loss = F.smooth_l1_loss(values, expected)

    self.__optimizer.zero_grad()
    # 求导
    loss.backward()
    for param in self.__policy.parameters():
      param.grad.data.clamp_(-1, 1)
      #更新policy神经网络的参数
    self.__optimizer.step()

    return loss.item()
```

- `sync()` updates the target network to the policy network in time delay;
- `save()` saves the current policy network parameters.

4.3 `utiles_env.py`

`utils_env.py` mainly implements the call package and configures the running game environment `MyEnv`, the main functions are as follows:

- `reset()` initializes the game and allow the agent to observe the environment in five steps;
- `step()` executes a step action and return the latest observation, feedback and Boolean value of whether the game is over;
- `evaluate()` uses the given agent model to run the game and return the average feedback value and record the frame of the game.

4.4 `utils_model.py`

`utils_model.py` uses `pytorch` to implement the Nature DQN model.

4.5 `utils_memory.py`

In `utils_memory.py`, we mainly implement `ReplayMemory`, which realizes data storage and random sampling. `ReplayMemory` is a bounded circular buffer used to store the most recently observed transformations. It also implements the `sample()` method, which is used to select random transition batches for training.

5 Double DQN Improve performance

5.1 Calculation of Target Q Value of DQN

Before DDQN, basically all the target Q values were directly obtained through the greedy method, whether it was Q-Learning, DQN (NIPS 2013) or Nature DQN. For example, for Nature DQN, although two Q networks are used and the target Q network is used to calculate the Q value, the target Q value of the jth sample is still calculated by the greedy method, which is calculated into the following formula:

$$y_j = \begin{cases} R_j & \text{is\_end}_j \text{ is true} \\ R_j + \gamma max_{a'} Q'(\phi(S'_j), A'_j, w') & \text{is\_end}_j \text{ is false} \end{cases}$$

Although using max can quickly bring the Q value closer to the possible optimization goal, it will lead to over estimation. The overestimation means that the algorithm model we finally get has a large deviation. To solve this problem, DDQN decouples the selection of target Q value action and the calculation of target Q value to eliminate the problem of over estimation.

5.2 Algorithm Modeling of Double DQN

Like Nature DQN, DDQN has the same two Q network structures. On the basis of Nature DQN, the problem of over estimation is eliminated by decoupling the two steps of target Q value action selection and target Q value calculation.

In the above, for the non termination state, the calculation formula of the target Q value of Nature DQN is:

$$y_j = R_j + \gamma \max_{a'} Q'(\phi(S'_j), A'_j, w')$$

In DDQN, instead of directly finding the maximum Q value of each action in the target Q network, first find the action corresponding to the maximum Q value in the current Q network, that is:

$$a^{max}(S'_j, w) = \arg\max_{a'} Q(\phi(S_j'), a, w)$$

Then use the selected action $a^{max}(S'_j, w)$ to calculate the target Q value in the target network, that is:

$$y_j = R_j + \gamma Q'(\phi(S'_j), a^{max}(S'_j, w), w')$$

To sum up:

$$y_j = R_j + \gamma Q'(\phi(S'_j), \arg\max_{a'} Q(\phi(S'_j), a, w), w')$$

Except for the calculation method of the target Q value, the DDQN algorithm and the Nature DQN algorithm process are identical.

5.3 DDQN Code Modification

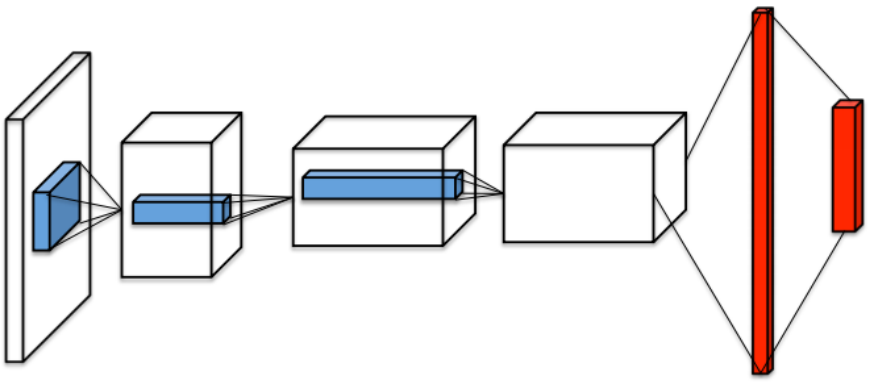
The only difference in the code is that when calculating the target Q value, the `learn()` function is modified in `utils_drl.py`

```
values = self.__policy(state_batch.float()).gather(1, action_batch)
# DDQN
max_action = self.__policy(next_batch.float()).max(1)[1]
values_next = self.__target(next_batch.float()).gather(1, max_action.unsqueeze(1)).squeeze(1)
# DQN
# values_next = self.__target(next_batch.float()).max(1).values.detach()
expected = (self.__gamma * values_next.unsqueeze(1)) * \
(1. - done_batch) + reward_batch
```

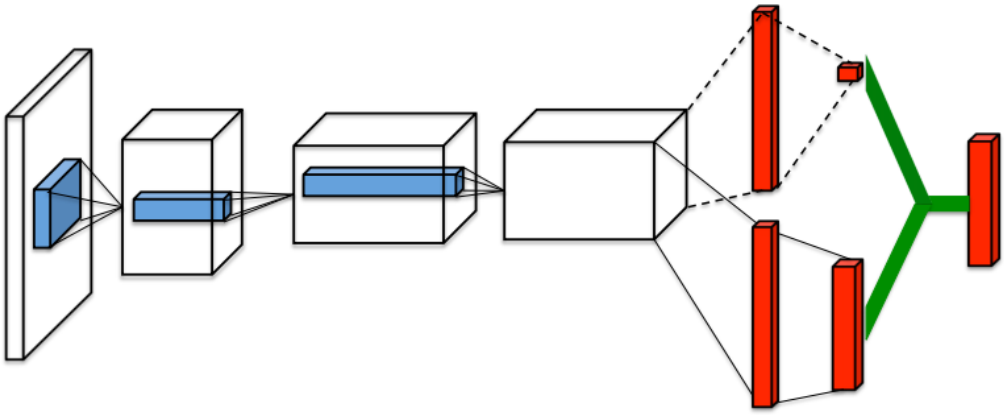
6 Dueling DQN

6.1 Dueling DQN Algorithm Modeling

Dueling DQN algorithm proposes a new neural network structure - duel network. The input of the network is the same as the input of DQN and DDQN algorithms, which are state information, but the output is different. The output of Dueling DQN algorithm includes two branches, namely, the state value V (scalar) of the state and the dominant value A (vector with the same dimension as the action space) of each action. The output of DQN and DDQN algorithms has only one branch, which is the action value of each action in this state (a vector with the same dimension as the action space). The specific network structure is shown in the figure below:



Single branch network structure (DQN and DDQN)



Dueling DQN

In the network structure of DQN algorithm, the input is one or more photos, the image features are extracted by convolution network, and then the action value of each action is output through full connection layer; In the network structure of Dueling DQN algorithm, the input is also one or more photos, and then the convolution network is used to extract image features to obtain feature vectors. The output will go through two full connection layer branches, corresponding to the state value and dominant value respectively. Finally, the action value of each action can be obtained by adding the state value and dominant value (i.e. green connection operation).

Dueling DQN is also an improvement based on DQN. The change is the last layer of the DQN neural network. The original last layer is a full connection layer, through which n Q values are output (n represents the number of selectable actions). Dueling DQN does not directly train to obtain the n Q values. It obtains two indirect variables V (state value) and A (action advantage) through training, and then represents the Q value by their sum.

$$V^{\pi}(s) = E_{a\ \pi(s)}[Q^{\pi}(s,a)]$$

$$A^{\pi}(s,a) = Q^{\pi}(s,a) - V^{\pi}(s)$$

V represents the average expectation of Q value under the current state s (all optional actions are comprehensively considered). A represents how much Q exceeds the expected value when action a is selected. The actual Q (s, a) is the sum of the two. For example, when you see a game screen, you can roughly judge the current situation, and then see which action will gain the most from the current situation according to each choice. V is the current situation, and A is the gain to the current situation. Therefore, the model is designed so that the neural network can have a basic judgment on the given s, and then modify it according to different actions.

However, there is a problem with direct training according to the above idea. The problem is that if the neural network trains V to a fixed value of 0, it is equivalent to an ordinary DQN network, because the A value at this time is the Q value. So we need to add a constraint condition to our neural network, so that the sum of A values corresponding to all actions is zero, so that the trained V value is the average of all n Q values (n represents the number of optional actions). The specific approach in this paper is to use the formula in the following figure to give constraints.

$$Q(s,a;w) = V(S;w,a) + (A(S,A;w,\beta) - \frac{1}{\mathcal{A}}\sum_{a' \in |\mathcal{A}|} A(a,a';w,\beta))$$

The part in parentheses in the formula is actually the value A previously mentioned, and the | A | in the formula represents the number of selectable actions (that is, n in the previous paragraph). It is obvious that if you add the parts in brackets corresponding to | A | actions, their sum is zero. So the problem is converted into using neural network to find $V(s;theta,beta)$ and $A(s,a;theta,alpha)$ in the above formula. Where, $V(s;theta,beta)$ is the V value mentioned above, while $A(s,a;theta,alpha)$ is actually different from the generalized A value mentioned above, but the A value can be calculated by $A(s,a;theta,alpha)$.

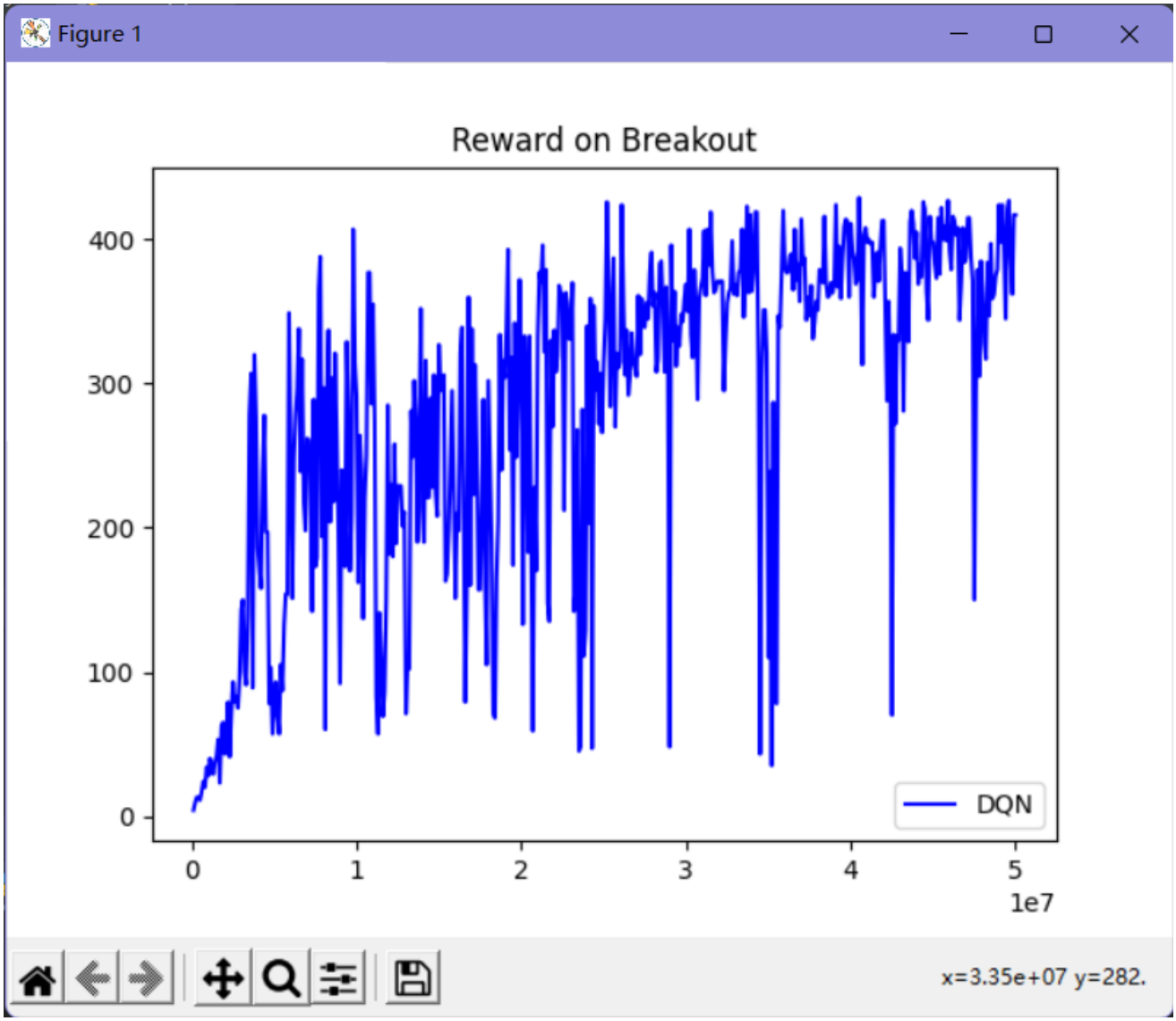
6.2 Dueling DQN Code

Modify in `utils_model.py`

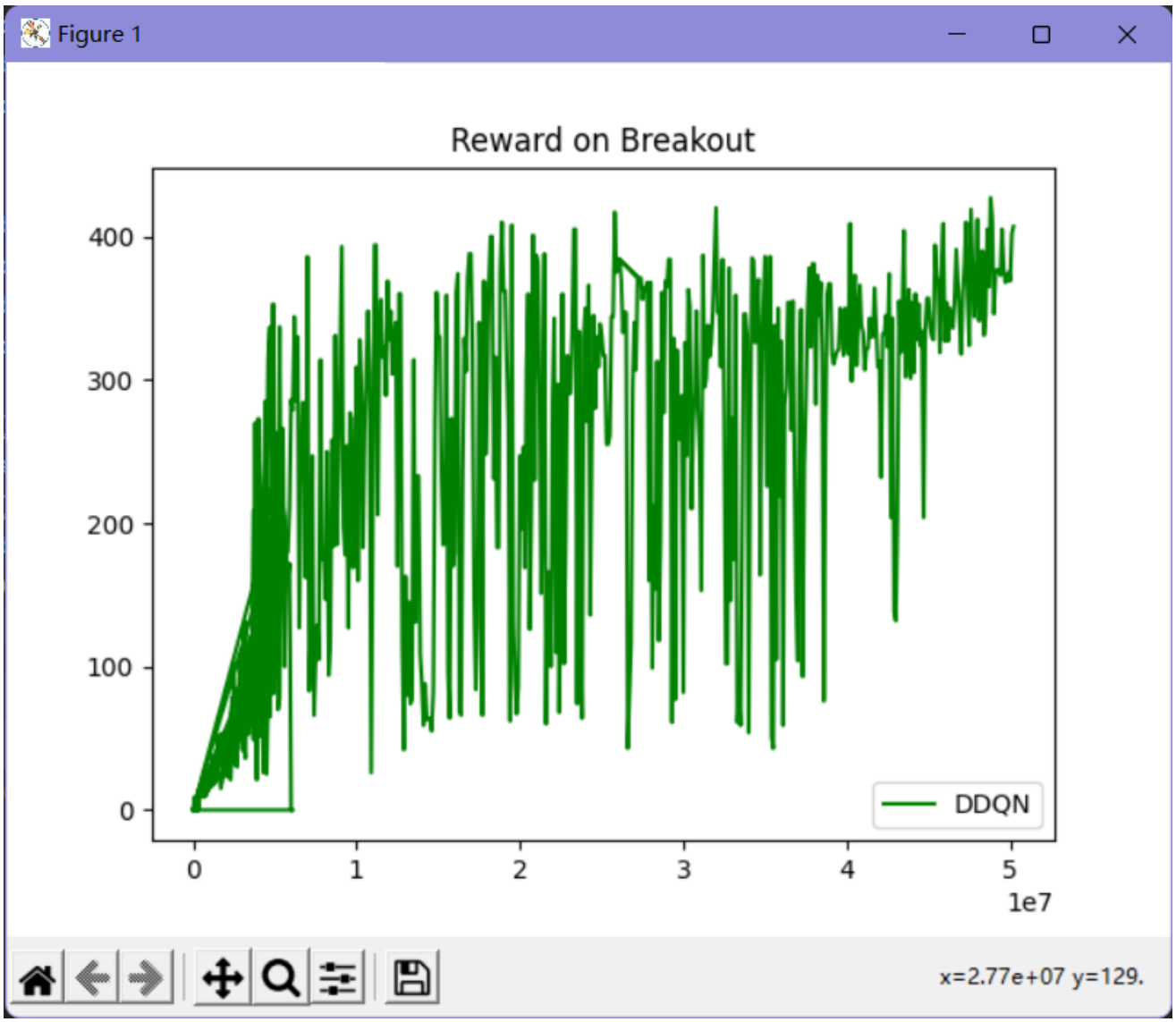
```
class Dueling-DQN(nn.Module):
    def __init__(self, action_dim, device):
        super(DQN, self).__init__()
        self.__conv1 = nn.Conv2d(4, 32, kernel_size=8, stride=4, bias=False)
        self.__conv2 = nn.Conv2d(32, 64, kernel_size=4, stride=2, bias=False)
        self.__conv3 = nn.Conv2d(64, 64, kernel_size=3, stride=1, bias=False)
        self.__fc1_advantage = nn.Linear(64*7*7, 512)
        self.__fc1_value = nn.Linear(64*7*7, 512)
        self.__fc2_advantage = nn.Linear(512, action_dim)
        self.__fc2_value = nn.Linear(512, 1)
        self.__act_dim = action_dim
        self.__device = device

    def forward(self, x):
        x = x / 255.
        x = F.relu(self.__conv1(x))
        x = F.relu(self.__conv2(x))
        x = F.relu(self.__conv3(x))
        x_v = x.view(x.size(0), -1)
        v_s = self.__fc2_value(F.relu(self.__fc1_value(x_v))).expand(x.size(0), self.__act_dim)
        asa = self.__fc2_advantage(F.relu(self.__fc1_advantage(x_v)))
        return v_s + asa - asa.mean(1).unsqueeze(1).expand(x.size(0),self.__act_dim)
```

7 Result

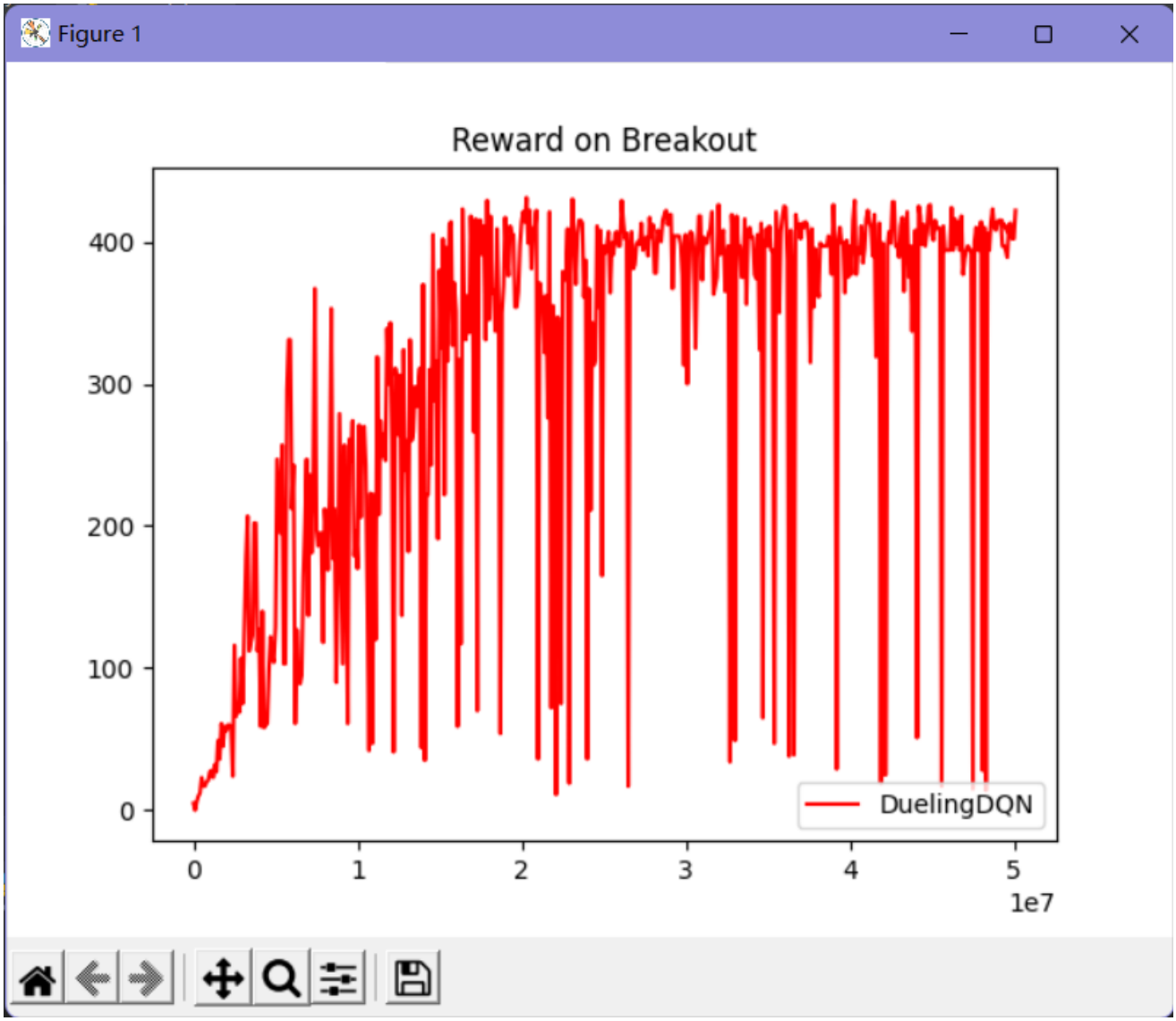
I did 50 million iterations of each of the three methods and plotted the results in a line graph.



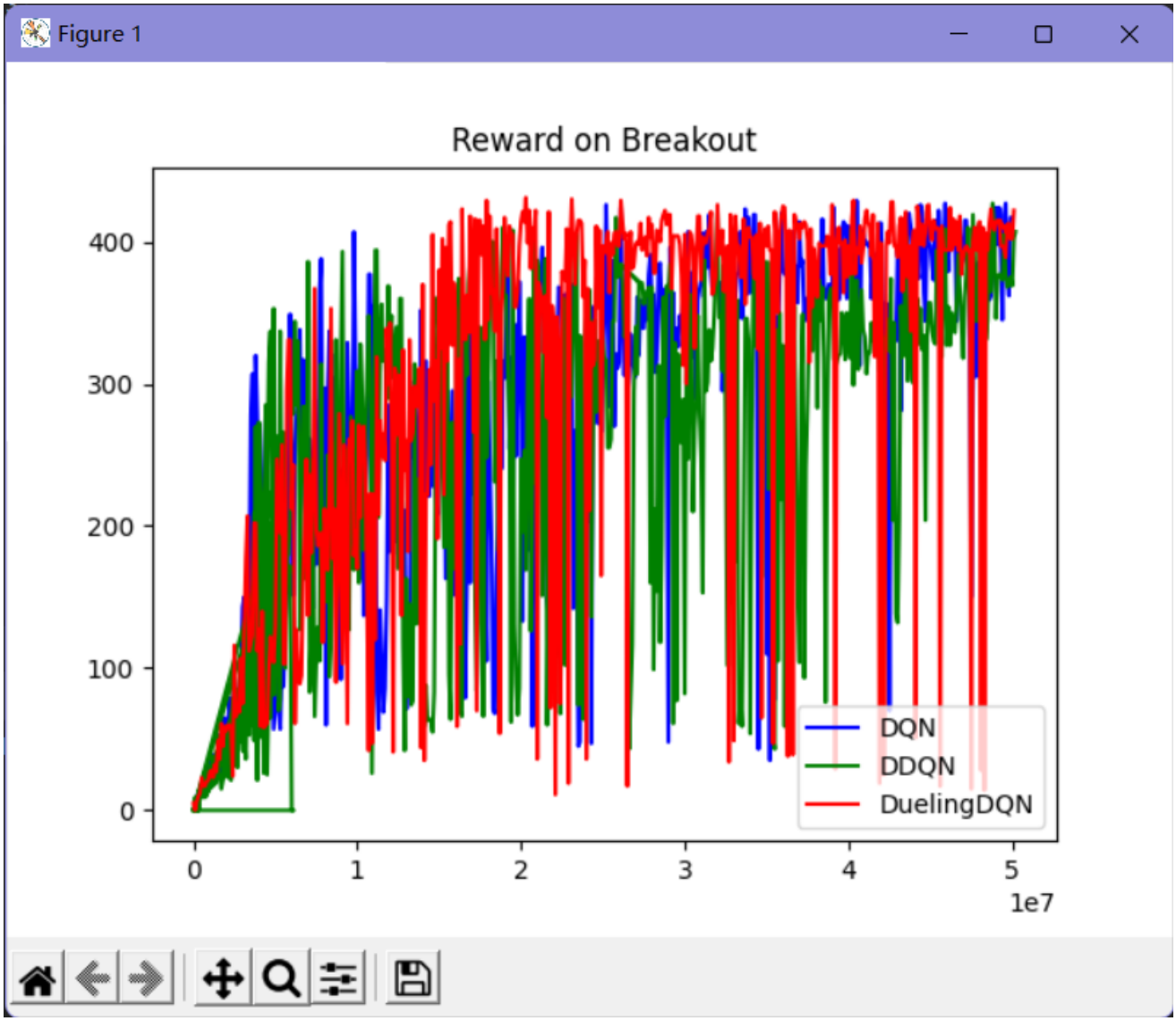
The data after 30 million times is relatively stable. Compared with DQN, the reward of DDQN is more stable, and can almost reach more than 400 when approaching 50 million times.



Dueling-DQN has gradually stabilized at around 400 as it approaches 20 million times, and you can see that it is significantly better than the first two methods.



When the results of the three methods are plotted on a graph, it can be seen that the results of Dueling-DQN are better.



【Reference】

[Deep Learning for Video Game Playing]

[Playing Atari with Deep Reinforcement Learning]

[Human-level Control Through Deep Reinforcement Learning]

[Deep Reinforcement Learning with Double Q-learning]

[Dueling Network Architectures for Deep Reinforcement Learning]