Reinforcement Learning and Game Theory, Fall, 2022

> Based on the Atari game breakout.
>
> All Source Code AND Parameters are based on: https://gitee.com/goluke/dqn-breakout

Team Information

- 任铭
- 20337231

- 范云骢
- 20337191

Introduction

In some cases, Q-learning algorithm cannot solve problems effectively. For example, Q-table can be used to store the Q value of each state action pair when the state and action space is discrete and the dimension is not high, while Q-table is unrealistic when the state and action space is high-dimensional continuous. Therefore, in order to deal with problems effectively, people found the DQN algorithm.

DQN (Deep Q-Network) is the groundbreaking work of Deep Reinforcement Learning. It introduces deep learning into reinforcement learning and builds the End-to-end architecture of Perception to Decision. DQN was originally published by DeepMind in NIPS 2013, and an improved version was published in Nature 2015.

NIPS DQN

- The following figure shows the pseudocode of DQN in NIP 2013:

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

Nature DQN

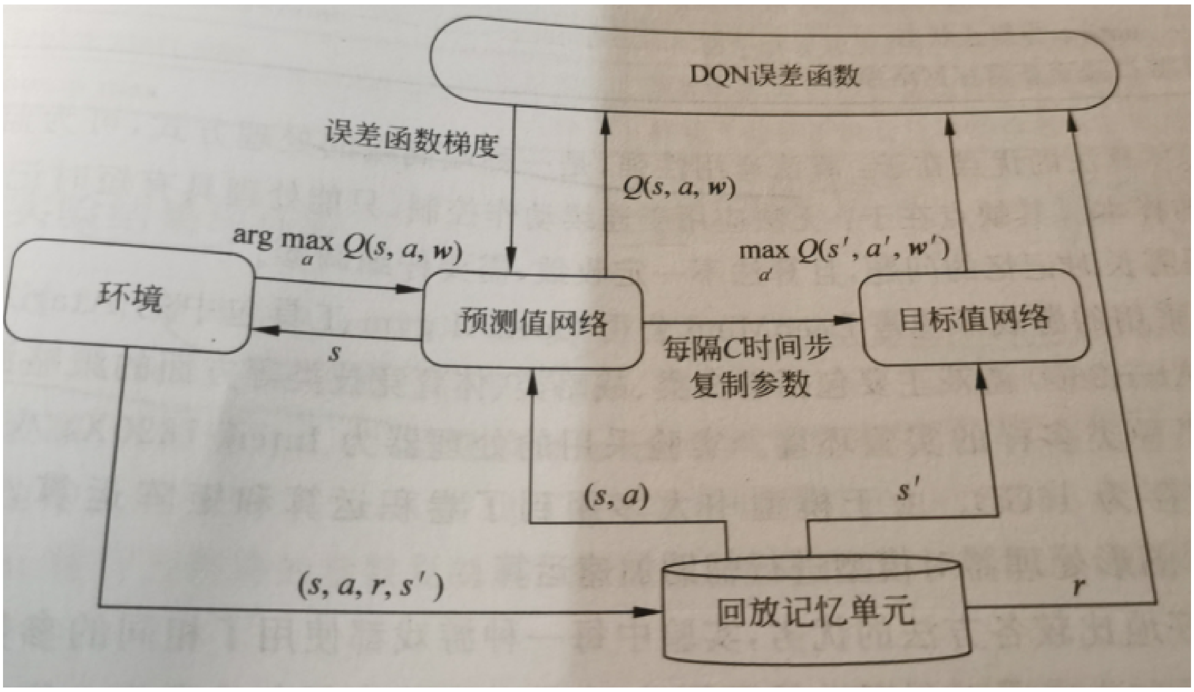
- In Nature 2015, DQN was improved, and a target network was proposed. The pseudocode is shown below:

Algorithm 1: deep Q-learning with experience replay.
Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
For episode $= 1, M$ **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For $t = 1$,T **do**
    With probability ε select a random action a_t
    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
    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 D
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  End For
End For

In fact, when we calculate the value y_i , we use the target network, and then we are actually trained to be main network, and then every C steps, the target network parameter will be "assigned" to the main network parameter. So it's going to be updated once.

Source Code

How DQN trains



main.py

> Required Parameters.

```
GAMMA = 0.99  # discount factor
GLOBAL_SEED = 0
MEM_SIZE = 100_000  # total memory size
RENDER = False
SAVE_PREFIX = "./models"
STACK_SIZE = 4

EPS_START = 1.
EPS_END = 0.1
EPS_DECAY = 1000000

BATCH_SIZE = 32
POLICY_UPDATE = 4  # the frequency of updating policy network
TARGET_UPDATE = 10_000  # the frequency of updating target network
WARM_STEPS = 50_000  # the number of warming steps
MAX_STEPS = 50_000_000  # total learning steps
EVALUATE_FREQ = 100_000
```

> Check and modify the game

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

> Observe the current state

```
state = env.make_state(obs_queue).to(device).float()
```

> Select an action according to the current state using $\epsilon - greedy$ algorithm

```
action = agent.run(state, training)
```

> Execute the action and get the information of observation, reward and done

```
obs, reward, done = env.step(action)
```

> Add observation to the observation queue and push the current state, action, reward and done into memory replay

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

> Check if it is time to update the policy network and target network

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

if step % TARGET_UPDATE == 0:
    agent.sync()
```

> Check if it is time to record the reward information and the performance of the network, and save them to local disk

```
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
```

utils_drl.py

- The purpose of this file is to define an agent, which has four behaviors.

1. `run()`: Decide the next action based on the current state
2. `learn()`: Information obtained from the experience pool by random sampling is used to update parameters in the policy grid
3. `sync()`: Example Synchronize the weight from the policy network to the target network
4. `save()`: Save the structure and parameters of the policy network to the local disk

utils_env.py

- The purpose of this file is to create an execution environment for the breakout and various definitions to get and manipulate the current state. The main functions are as follows.

1. `reset()`: Initialize the game, set the agent to the initial state, and keep the same 5 steps, observe the initial environment
2. `step()`: Receive the action sequence number and execute it, returning the next state reward and information about whether the game is complete
3. `evaluate()`: The performance is evaluated by running the game set and the average reward is returned

utils_memory.py

- This file gives the definition of the replay memory pool, which structure is a simple deque.

```
def sample(self, batch_size: int) -> Tuple[
        BatchState,
        BatchAction,
        BatchReward,
        BatchNext,
        BatchDone,
]:
    indices = torch.randint(0, high=self.__size, size=(batch_size,))
    b_state = self.__m_states[indices, :4].to(self.__device).float()
    b_next = self.__m_states[indices, 1:].to(self.__device).float()
    b_action = self.__m_actions[indices].to(self.__device)
    b_reward = self.__m_rewards[indices].to(self.__device).float()
    b_done = self.__m_dones[indices].to(self.__device).float()
    return b_state, b_action, b_reward, b_next, b_done
```

> it can store the last n experiences at most where n is the capacity of the pool. As for the sampling process, it use the random sampling to avoid the data dependencies. Like above.

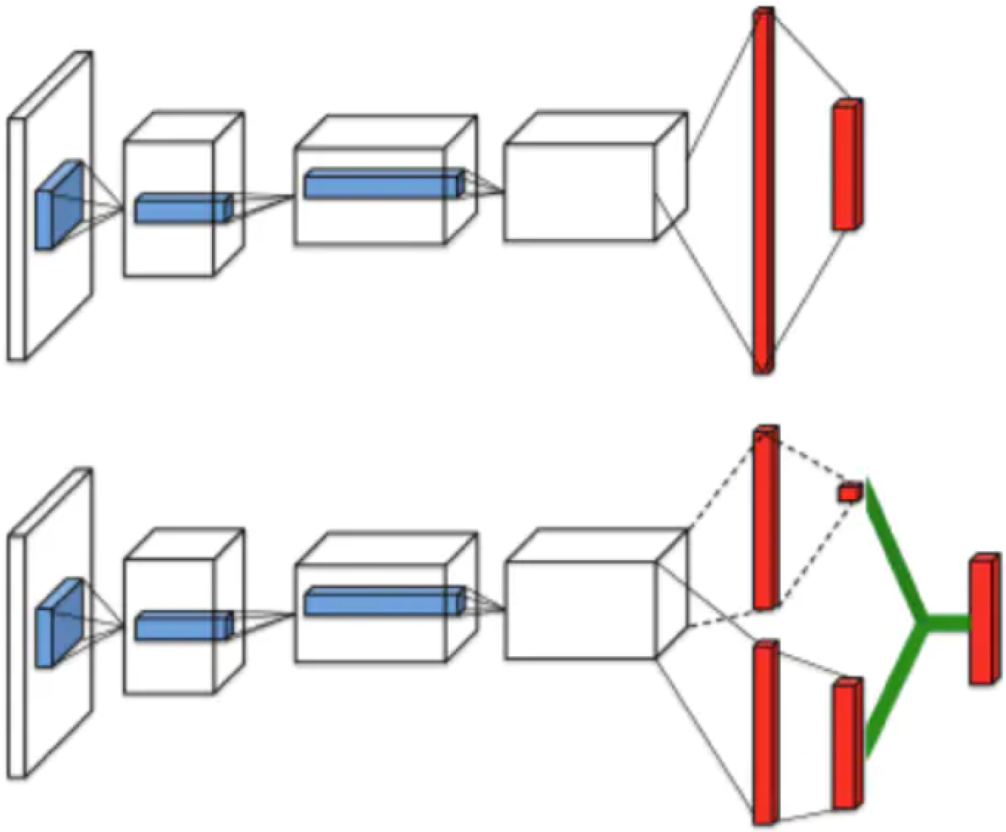
utils_model.py

- The definition of this file is to create a neural network.

1. `__init__()`: It consists of three convolution layers and two fully connected layers, the output of the last layer corresponds to the Q value of each action.
2. `forward()`: Method function of network computation.Which is followed by function `relu()`.
3. `relu()`: Activation function to increase the nonlinearity of the network model.

Dueling DQN

> Here is how Dueling DQN works. We can clearly see that the normal DQN above has only one output, which is the Q value of each action; The Dueling DQN breaks down the Value of the state and the Advantage of each action.



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

This formula mainly centralizes the advantage function, aiming to embody the respective influence of value function and advantage function. The other part is the same as the nature DQN. The main changes of codes are shown below. It breaks down the value of the state, plus the advantage of each action on that state. Because sometimes, no matter what you do in one state, it doesn't have much of an impact on the next state.

> DQN

```
class 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 = nn.Linear(64*7*7, 512)
        self.__fc2 = nn.Linear(512, 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 = F.relu(self.__fc1(x.view(x.size(0), -1)))
        return self.__fc2(x)
```

> Dueling-DQN

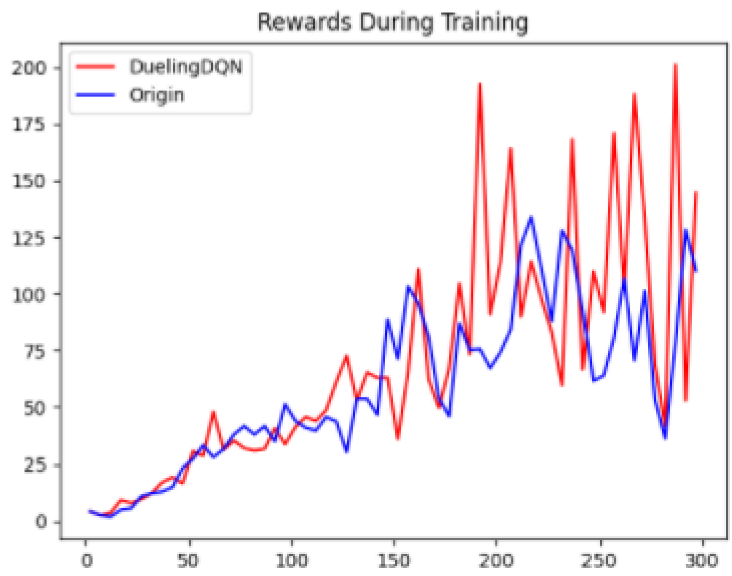
```
def __init__(self, action_dim, device):
    super(DQN, self).__init__()
    self.__conv1 = nn.Conv2d(4, 32, kernel_size=8, stride=4, bias=False)
    self.__conv2 = nn.Conv2d(32, 64, kernel_size=4, stride=2, bias=False)
    self.__conv3 = nn.Conv2d(64, 64, kernel_size=3, stride=1, bias=False)
    self.__fc1_a = nn.Linear(64*7*7, 512)
    self.__fc1_v = nn.Linear(64*7*7, 512)
    self.__fc2_a = nn.Linear(512, action_dim)
    self.__fc2_v = nn.Linear(512, 1)
    self.__device = device
    self.actionsDim = action_dim

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))
    a = F.relu(self.__fc1_a(x.view(x.size(0), -1)))
    a = self.__fc2_a(a)
    v = F.relu(self.__fc1_v(x.view(x.size(0), -1)))
    v = self.__fc2_v(v).expand(x.size(0), self.actionsDim)
    res = v + a - a.mean(1).unsqueeze(1).expand(x.size(0), self.actionsDim)
    return res
```

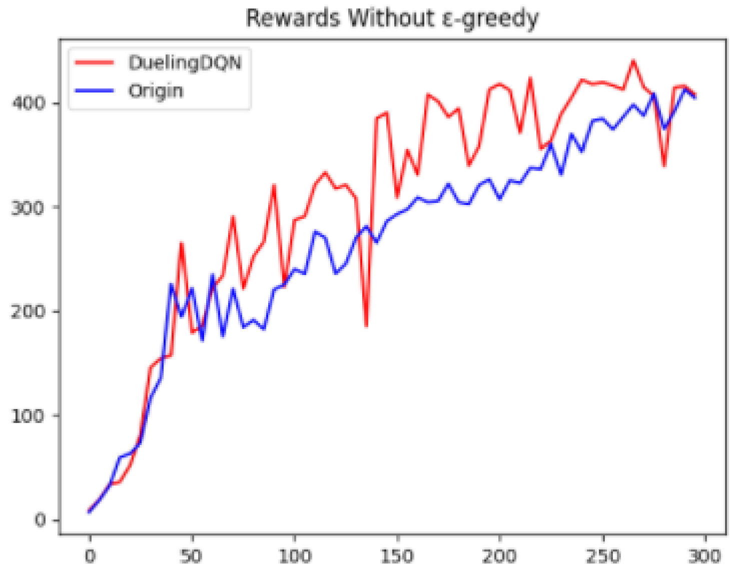
Results

> Rewards During Training

We can find out that DuelingDQN works better than OriginDQN while training



> Rewards Without $\epsilon - greedy$



Summary

We use .Ipynb for the model performance. Similar to Qleanring, DQN is an algorithm based on value iteration. However, in ordinary Q-learning, Q-table can be used to store the Q value of each state action pair when the state and action space is discrete and the dimension is not high. However, when the state and action space is high-dimensional continuous, It is difficult to use Q-Table without too much action space and state.Therefore, Q-table can be updated into a function fitting problem here, and Q-Table can be replaced by a function fitting to generate Q values, so that similar states can get similar output actions. Therefore, it can be thought that deep neural network has a good effect on the extraction of complex features, so DeepLearning can be combined with Reinforcement Learning. This becomes DQN.

Distribution

Member	Ideas (%)	Coding (%)	Writing (%)
任铭	60%	60%	40%
范云骢	40%	40%	60%

References

【强化学习】Deep Q-Network (DQN) - 知乎 (zhihu.com)

DQN从入门到放弃5 深度解读DQN算法 - 知乎 (zhihu.com)

深度强化学习中深度Q网络（Q-Learning+CNN）的讲解以及在Atari游戏中的实战（超详细 附源码）_showswoller的博客-CSDN博客

•https://arxiv.org/pdf/1511.06581.pdf