Reinforcement Learning and Game Theory

Mid-term Project: DQN-Breakout

谢小卉 19335225 伍海珊 19335215

0.Division of work

members	ideas	coding	writing
谢小卉	50%	55%	45%
伍海珊	50%	45%	55%

1.Code Analysis

Combining Q-learning and deep learning in reinforcement learning, DQN realizes a revolutionary end-to-end algorithm from perception to action, which is a good choice when the Q-table is too large to set up. The original code used DQN to train the Breakout game. Here's how to interpret each file.

- ***main.py***

In preparation, we set up and initialize the game environment, then create an agent.

```
torch.manual_seed(new_seed())
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
env = MyEnv(device)
agent = Agent(
    env.get_action_dim(),
    device,
    GAMMA,
    new_seed(),
    EPS_START,
    EPS_END,
    EPS_DECAY,
)
memory = ReplayMemory(STACK_SIZE + 1, MEM_SIZE, device)
```

After creating the environment, we start training. We created a Python progress bar on the terminal to output the training progress in real time. At the same time, we create an observation queue to store the state of the last 5 steps. In the current environment, the agent selects an action to execute, obtains state changes and rewards, and stores these information in the memory experience pool.

```
#### Training ####
obs_queue: deque = deque(maxlen=5)
```

```
done = True

progressive = tqdm(range(MAX_STEPS), total=MAX_STEPS,
                   ncols=50, leave=False, unit="b")  #python进度条
for step in progressive:
    if done:
        observations, _, _ = env.reset()
        for obs in observations:
            obs_queue.append(obs)

    training = len(memory) > WARM_STEPS
    state = env.make_state(obs_queue).to(device).float()
    action = agent.run(state, training)
    obs, reward, done = env.step(action)
    obs_queue.append(obs)
    memory.push(env.make_folded_state(obs_queue), action, reward, done)
```

After each *POLICY_UPDATE* steps, the agent extracts BATCH_SIZE samples from the experience pool for learning.

After each *TARGET_UPDATE* steps, the weight of the policy network is synchronized to the target network.

After each *EVALUATE_FREQ* steps, avg_reward is saved once and written to the target txt file.

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

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

    if step % EVALUATE_FREQ == 0:
        avg_reward, frames = env.evaluate(obs_queue, agent, render=RENDER)
        with open("rewards5.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***

This file defines and encapsulates agent variables and functions.The actions of each function are commented in the file.

run: Indicates the recommended action to be taken in a given state

learn: Learns the samples in the experience pool through TD-learning and train the value network

sunc: Synchronizes the weight of the policy network to the target network *save:* Saves the state_dict of the policy network

- ***utils_env.py***

Use the Atari Breakout environment integrated in gym for DQN training.

The suffix of *breakoutnoframeskip-V4* indicates that the probability of repeating the previous action, p, is 0.

reset: Resets the status of the environment and returns to the observation.

step: A physics engine that pushes forward one time step and returns Observation, reward, done

get_frame: A graphics engine that draws the current frame of the game.

evaluate: Runs several chapters of the game with a given agent, returning the average reward and captured frames.

- ***utils_memory.py***

ReplayMemory stores the most recent training samples that have been generated through interaction with the environment.

ReplayMemory has two benefits. First, the same sample can participate in training for many times to improve learning efficiency. Second, DQN works under the framework of supervised learning. Neural network requires samples to be independent, distributed and unrelated, while samples generated by interaction with the environment are often sequences, which are highly correlated before and after. Memory Replay can ease the correlation of samples.

- ***utils_model.py***

This is the DQN core algorithm module, which is applied to the utils_drl file. The structure consists of three convolution layers and two full connection layers, and the final output corresponds to the Q value of each action. The init_weights function initializes the weights, while the forward function evaluates the network.

2.Dueling DQN

Dueling DQN optimizes the algorithm by optimizing the structure of the neural network. It divides the Q network into two parts.

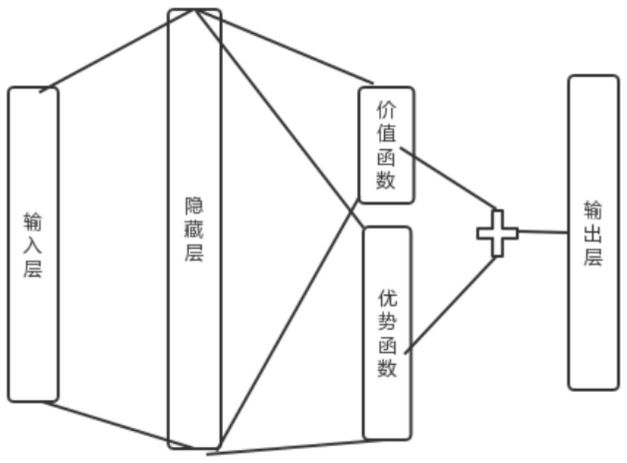
The first part is only concerned with the state S, and is independent of the specific action A to be taken. This part is called the value function part, which is called V(S, W, α). The second part is related to both state S and action A, which is called Advantage Function and is denoted as A(S, A, W, β).

Then our value Function can be reexpressed as:

$$Q(S, A, w, \alpha, \beta) = V(S, w, \alpha) + A(S, A, w, \beta)$$

Among them, **w** is the network parameter of the common part, **α** is the network parameter of the unique part of the value function, and **β** is the network parameter of the unique part of the advantage function.

Since the value function of Q network is divided into two parts, the network structure of Dueling DQN is different from that of DQN. In Dueling DQN, we add two sub-network structures, corresponding to the price function network part and the advantage function network part above respectively.



Corresponding to the figure above, the output of the final Q network is obtained by the linear combination of the output of the price function network and the output of the advantage function network.

However, this formula cannot identify the respective roles of V(S, W, α), V(S, W, α), A(S, A, W, β) and A(S, A, W, β) in the final output. In order to reflect this identifiability, the actual combination formula is as follows:

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

3.Code Improvement

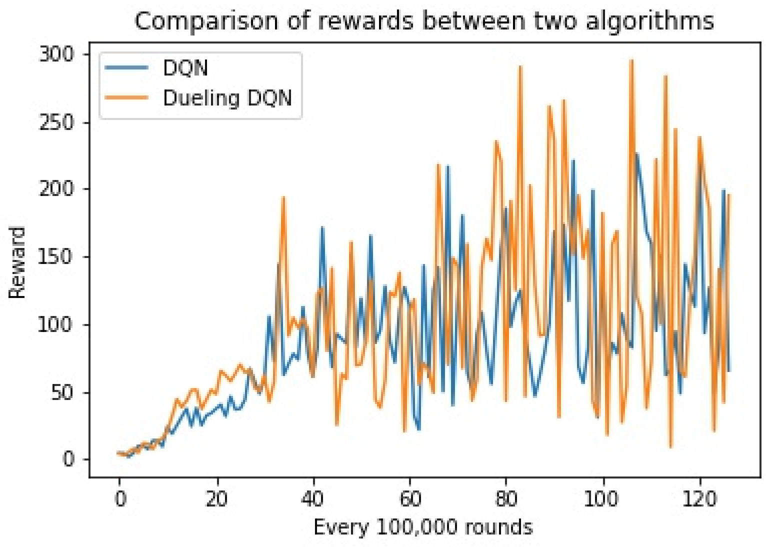
The utils_model.py file has been modified. The improved code using the Dueling DQN algorithm is shown below.

```
class DQN(nn.Module):

    # 将网络末端拆成值网络和优势网络，对应Dueling DQN中的基于状态的值函数和优势函数
    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

    @staticmethod
    def init_weights(module):
        if isinstance(module, nn.Linear):
            torch.nn.init.kaiming_normal_(module.weight, nonlinearity="relu")
            module.bias.data.fill_(0.0)
        elif isinstance(module, nn.Conv2d):
            torch.nn.init.kaiming_normal_(module.weight, nonlinearity="relu")
```

4.Experimental Results and Analysis

We used the DQN algorithm and the improved Dueling DQN algorithm to train agents 15 million rounds respectively to obtain two rewards files.

Use a simple visualization program to graph the two rewards files, as shown below:



Therefore, Dueling DQN can start convergence faster, and the training effect is better than that of traditional DQN.