Interim Project Report

19335125 廖剑雄，19335155 麦子丰

Introduction of our work

In the previous work, we finish the implementation of sarsa and Q-learning, which store the value Q in the memory. However, because of the high complexity of the atari game, it's impossible to store all value in the memory. Therefore, we regard the Q as a function of state and compute the present value of the state. DQN is a good tool to help us achieve the assumption, which may help us compute the value and train the agent when we process the program. In this project, we get a good understanding of the implementation of the atari game and explain it in our report. In addition, we use the prioritized experience replay to improve the basic implementation and get a faster efficiency. Finally, we make an assumption of the way to stabilize the AI agent and make it more like a human player

Explaination of the basic implementation

The whole implementation of the code is real difficult to understand since it use a great amount of the functions defined in the python library, which we can only get part of their definition in head files. Therefore, we just try to explain the main frame of the code and get a brief understanding of the basic implementation.

We'll follow the implementation in main.py and explain in detail what each component is responsible for and how the components are connected together. We may ignore the parameters defined at the beginning of main.py since they will be used in the following code and we can explain their usage together with the function which call the parameters.

1. First of all, we need to initialize the environment of the game in order to ensure our training can exactly run. By checking the device used in our computer we can decide the runtime environment of the game. To initialize the environment of the atari game, we create an instance of MyEnv to initialize and return the environment

   ```
   torch.manual_seed(new_seed())
   #检查运行环境
   device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
   #初始化游戏环境
   env = MyEnv(device)
   ```

   After having a brief understanding of the function MyEnv, we shall know that the atari game is encapsulated in the python library and we can choose the type of atari game by calling the according parameter. In this game, we call the parameter "BreakoutNoFrameskip-v4" to get the environment we need. The init function mainly use the library function and we cannot actually know how it create the environment.

```
#BreakoutNoFrameskip-v4为打转块游戏环境
        env_raw = make_atari("BreakoutNoFrameskip-v4")
        self.__env_train = wrap_deepmind(env_raw, episode_life=True)
        env_raw = make_atari("BreakoutNoFrameskip-v4")
        """Configure environment for DeepMind-style Atari."""
        self.__env_eval = wrap_deepmind(env_raw, episode_life=True)
        self.__env = self.__env_train
        self.__device = device
```

2. After creating the environment, we should initialize the AI player and the memory used to record the path. AI player is represented by the agent, which is initialized by the constructer of the class Agent. The constructer initialize most of the parameter used in the running and training such as gamma, device. It also initialize two DQN network for training, from which we can get the value of Q to decide the further action.

```
self.__action_dim = action_dim
self.__device = device
self.__gamma = gamma

self.__eps_start = eps_start
self.__eps_final = eps_final
self.__eps_decay = eps_decay

self.__eps = eps_start
self.__r = random.Random()
self.__r.seed(seed)

self.__policy = DQN(action_dim, device).to(device)
self.__target = DQN(action_dim, device).to(device)
```

The DQN network has a framework as follow:

The first three convolution layers increase the number of channel from 4 to 64 while subsampling the resolution from 84*84 to 7*7. These convolution layer extract the high-level feature from the input image and decrease the number of parameter to train at the mean time. The following two full connect layers map the 64*7*7 feature graph to one of the 3 action as general Neural Network does. In addition, each layer use the ReLU function as activation function and has no Batch Normalization operation.

DQN initialize the parameter obeying Kaiming Distribution and uses Adam optimizer for training.

```
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)
```

3. The ReplayMemory is also an important data structure used in the program. We initialize the dimension of the memory and the whole capacity of it. The dimension of the memory represents the length of path recorded in the memory. From the definition of memory, we know that every path recorded in the memory is formed by the map of the game.

```
def __init__(
        self,
        channels: int,
        capacity: int,
        device: TorchDevice,
) -> None:
self.__device = device
self.__capacity = capacity
self.__size = 0
self.__pos = 0

self.__m_states = torch.zeros(
    (capacity, channels, 84, 84), dtype=torch.uint8)
self.__m_actions = torch.zeros((capacity, 1), dtype=torch.long)
self.__m_rewards = torch.zeros((capacity, 1), dtype=torch.int8)
self.__m_dones = torch.zeros((capacity, 1), dtype=torch.bool)
```

4. After the initialization of the environment, we can start processing the game and training our AI player. According to the meaning of the parameter training, we know that after getting WARM_STEPS(50000) path, we shall start training our agent. The function make_state and agent.action are used to get the next action according to the present environment. And calling the function env.step, we update the environment due to the action and get the next observation, reward of the game. Then we push the observation into our deque to get a new path which will be recorded in the memory. If the parameter done is true, it shows that the game comes to end and we should reset the environment to start a new game.

```
if done:
    observations, _, _ = env.reset()
    """reset resets and initializes the underlying gym environment.
    maybe observation represent the environment"""
    for obs in observations:
        obs_queue.append(obs)

training = len(memory) > WARM_STEPS
state = env.make_state(obs_queue).to(device).float()
"""run suggests an action for the given state.
if memory is bigger than warm_steps then reduce eps"""
action = agent.run(state, training)
"""step forwards an action to the environment and returns the newest
        observation, the reward, and an bool value indicating whether the
        episode is terminated."""
obs, reward, done = env.step(action)
obs_queue.append(obs)
"""make_folded_state将队列列表化并变成1*n大小的二维数组，这样一整个状态可以作为一个成员
存储在memory"""
memory.push(env.make_folded_state(obs_queue), action, reward, done)
```

5. If the parameter training turns to true, we can start training our agent. Firstly, we get 32 samples from previous path recorded in memory. The action_batch, reward_batch and done_batch are easy to understand. But it's necessary for us to get a full understanding of the function sample to learn about the production of the state_batch and next_batch. State_batch gets the first four observations of the path and next_bath gets the last four observations of the path. We may know that after taking the action, the state observations will change to the next observation, for which we can get four last-next pair from one path. After getting 32*4 last-next pair from the sample, we compute the values and next_values by using the policy DQN and target DQN. We should compute the expected value using the parameter gamma and reward so that we can then compute the loss of the DQN. We use the smooth_l1_loss to train our DQN in order to minimize the loss. According to the following picture, we learn about that policy DQN represents the Q and target DQN represents the $\widehat Q$. In addition, smooth_l1_loss represents the Δw.

$$\Delta \mathbf{w} = \alpha\left(r + \gamma \max_{a'} \hat{Q}(s', a', \mathbf{w}^{-}) - Q(s, a, \mathbf{w})\right)\nabla_{\mathbf{w}} Q(s, a, \mathbf{w})$$

```
def learn(self, memory: ReplayMemory, batch_size: int) -> float:
"""learn trains the value network via TD-learning."""
state_batch, action_batch, reward_batch, next_batch, done_batch = \
    memory.sample(batch_size)

values = self.__policy(state_batch.float()).gather(1, action_batch)
values_next = self.__target(next_batch.float()).max(1).values.detach()
expected = (self.__gamma * values_next.unsqueeze(1)) * \
    (1. - done_batch) + reward_batch
"""计算梯度Δw"""
loss = F.smooth_l1_loss(values, expected)

self.__optimizer.zero_grad()
loss.backward()
for param in self.__policy.parameters():
    param.grad.data.clamp_(-1, 1)
self.__optimizer.step()

return loss.item()
```

6. Finally, the function sync is used to synchronize the target DQN and policy DQN, which may make the training more precise. We may record the rewards every EVALUATE_FREQ steps to get the final reward curve.

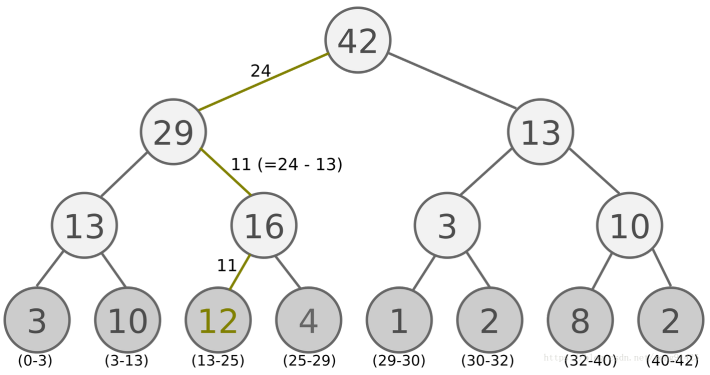
Implementation of Prioritized Experience Replay

In the classic DQN network, we choose completely randomly a batch from the ReplayMemory every time to train our network. However, it seems to be inefficient to sample a training batch in that way since the importance of these data is not the same for the AI agent. Prioritized Experience Replay is an improvement based on this idea.

The basic idea of Prioritized Experience Replay is to break uniform sampling and assign the samples with higher learning efficiency more sampling weight so that they are more likely to be sampled. A ideal measurement of the learning efficiency is the TD deviation donated by

$$\delta = \left| R + \gamma \, max_{a'} \, Q(s', a', w^{-}) - Q(s, a, w) \right|$$

In order to sampling and storing data efficiently, we implement a data structure named SumTree as following. SumTree is a Complete Binary Tree. Suppose there are n sample stored in ReplayMemory, the SumTree has n leaf nodes representing each sample by one leaf. Each internal node has a value equal to the sum of its children node.



Here's the way to sample a data in the SumTree. Donating the total value of each leaf nodes as m， firstly choose a number s from 0 to m randomly and then access the SumTree from root all way down to a leaf by the following principle:

- If the value of left child is greater than s, then go to the left child.
- Else, subtract s by the value of the left child, then go to the right child.

Since the SumTree is a Complete Binary Tree, we can implement it by using an array. It can be proved that if there're n leaves, there're n-1 internal nodes in the tree. For the node with index i , its children are 2i and 2i+1, its parent are i//2. From above, we can easily access the parent node or child node using the index.

We create a tensor named __m_error in ReplayMemory worked as the SumTree. When pushing a record into ReplayMemory, we also provide the expected value and real value by accessing the DQN network policy and target. Then we insert the TD error to the SumTree by modifying the value from the inserted leaf up to the root.

```
def push(
        self,
        folded_state: TensorStack5,
        action: int,
        reward: int,
        done: bool,
        values: float,
        expected: float
) -> None:
    diff = torch.zeros((1,1),dtype = torch.float)
    diff[0][0] = abs(expected-values)
    if self.__size == self.__capacity:
        pre = self.__m_error[self.__pos, 0]
        pre_pos = self.__pos
        while pre_pos != 0:
            self.__m_error[pre_pos, 0] -= pre
            pre_pos = pre_pos//2
    self.__m_states[self.__pos] = folded_state
    self.__m_actions[self.__pos, 0] = action
    self.__m_rewards[self.__pos, 0] = reward
    self.__m_dones[self.__pos, 0] = done
    pre_pos = self.__pos
    while pre_pos != 0:
        self.__m_error[pre_pos,0] += diff[0][0]
        pre_pos = pre_pos // 2
    self.__pos = 100001 if self.__pos == 2*self.__capacity else self.__pos+1
    self.__size = min(self.__capacity, self.__size+1)
```

When Sampling a batch from memory, we used the method described above for batch_size time.

```
def sample(self, batch_size: int) -> Tuple[
        BatchState,
        BatchAction,
        BatchReward,
        BatchNext,
        BatchDone,
]:
    dice = [random.uniform(0, self.__m_error[1, 0]) for i in range(batch_size)]
    indices = [0 for i in range(batch_size)]
    num = 0
    for i in dice:
        pos = 1
        while pos <= 100000:
            if i < self.__m_error[2 * pos, 0]:
                pos = 2*pos
            else:
                i -= self.__m_error[2*pos, 0]
                pos = 2*pos+1
        indices[num] = i
        num += 1
    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
```

We've tried to use our GPU to test our method and get the result. But we fail to finish the job since the video memory in our computer isn't large enough to finish running the program, while using the given colony is even slower than our CPU. Because of this, we couldn't have enough time to finish running this job. But we've test that the code of improvement can actually run in our computer, for which we consider it a correct implementation.

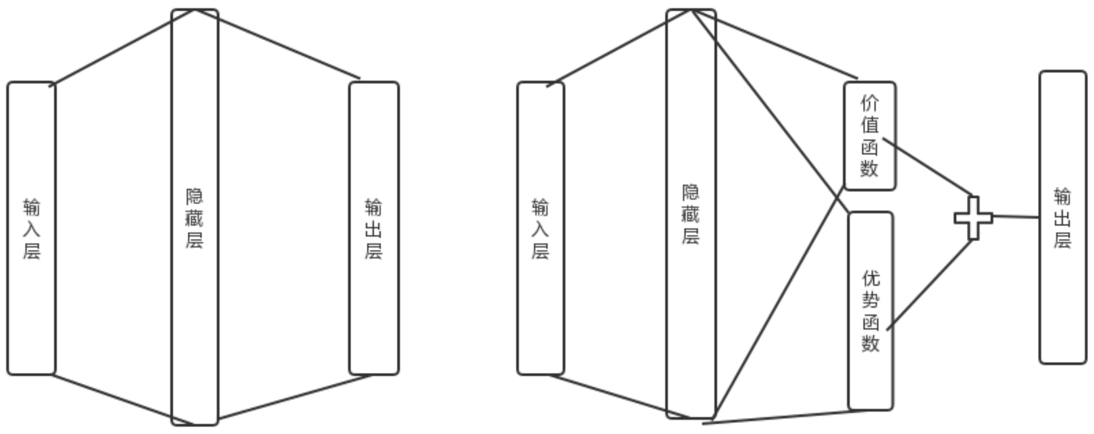
But What is puzzling is that we can successfully run the basic program in our computer, but unable to run the improved code due to the memory size. By analysing the implementation in the basic code, we find that the code use the backward() function except using the forward() function, which may cause that in every learn step, the torch clear its memory and make the next step have enough memory to finish updating. However, in our code, we just use the forward() function and then the program may create a new net to compute next value without releasing the previous one. This action may continuously malloc memory from our view memory and finally it can't afford the expenses. For this reason, we just add with torch.no_grad() before the computation and solve the problem. The updated code is:

```
with torch.no_grad():
    folded_state = env.make_folded_state(obs_queue)
    indices = [0 for i in range(32)]
    state = folded_state[indices, :4].to(device).float()
    state_next = folded_state[indices, 1:].to(device).float()
    pre_action = torch.zeros((1, 1), dtype=torch.long)
    pre_action[0][0] = action
    #计算对应的值，从而能够存储比较差值大小
    values = agent.policy(state.to(device).float()).gather(1,
pre_action[indices].to(device))
    values_next =
agent.target(state_next.to(device).float()).max(1).values.detach()
    expected = (agent.gamma * values_next.unsqueeze(1)) * \
           (1. - done) + reward
    values = sum(values[0])
    expected = sum(expected[0])
    memory.push(folded_state, action, reward, done,values,expected)
```

Using Dueling DQN to improve the performance

We know that Dueling DQN is a way to improve our DQN network. It decomposes Q(s,a) into V(s) and A(s,a), which represent different values. V(s) represents the value of being at the state and A(s,a) represents the advantage of taking action a in state s versus all other possible actions at that state. Therefore, we use two streams to finish the work: one that estimates the state value V(s) and one that estimates the advantage for each action A(s,a).

The main idea of implementing the Dueling DQN is to modify the DQN network and compute V(s) and A(s,a) separately. We know that the basic DQN network consists of three layers: input layer, hidden layer and output layer. In our Dueling DQN, we add two layer to one level of the network, which represent the value and advantage. The modified network is



In addition, if we just add V(s) and A(s,a) together, it may cause some problems. Therefore, we use the following formula, which centralizes the advantage function

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + (A(s,a;\theta,\alpha) - \sum_{a'} A(s,a';\theta,\alpha)/\left|A\right|) \quad (3)$$

The final implementation is as follows:

```
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)
        #初始化网络，将Q分为v和a
        self.__fc_v = nn.Linear(64 * 7 * 7, 512)
        self.__v = nn.Linear(512, 1)
        self.__fc_a = nn.Linear(64 * 7 * 7, 512)
        self.__a = 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))
        #计算当前的v值
        v = F.relu(self.__fc_v(x.view(x.size(0), -1)))
        v = self.__v(v)
        #计算当前的a值
        a = F.relu(self.__fc_a(x.view(x.size(0), -1)))
        a = self.__a(a)
        return v + (a-a.mean(dim = 1,keepdim = True))
```

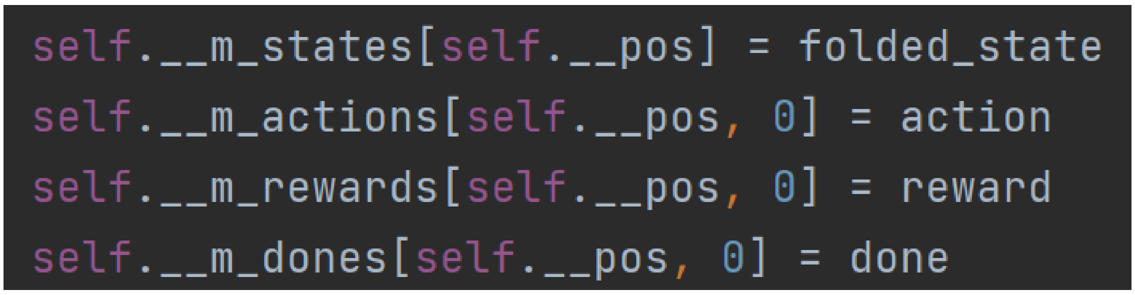
A possible method to stabilize the movement of the paddle

We can learn from the result of the training that the AI agent may keep shaking when playing the game. One of the reason is that the AI agent has three actions in total : left moving, right moving and noop. However, the reward of the three actions may be the same, for which choosing the left or right moving is the same as choosing the noop action. If we decide that the reward decrease when we choose the left or right moving, the agent will choose to stay in the previous state rather than move to left or right. In this way, the AI agent may be stabilized and avoid keeping shaking.

Unfortunately, the whole game and the reward in the atari are defined by the python library, for which we don't know how to change the action and modify the reward. So we can't actually try this method. It's necessary for us to learn more about the concrete definition of the game in the further study.

Some of the problems we find in the code

1. In the replay memory, we know that the memory restore five observations in one position. However, the memory only restore the fifth action in the same position, which may cause some inaccuracy. In the push function, we find that it only restore one action in the previous position, where the folded_state restore five observations in it.

   ```
   self.__m_states[self.__pos] = folded_state
   self.__m_actions[self.__pos, 0] = action
   self.__m_rewards[self.__pos, 0] = reward
   self.__m_dones[self.__pos, 0] = done
   ```

   We know that the diff is computed as follows

$$\alpha\left(R_{t+1} + \gamma \max_a \hat{q}(s_{t+1}, a, \mathbf{w}) - \hat{q}(s_t, a_t, \mathbf{w})\right)\nabla_{\mathbf{w}}\hat{q}(s_t, a_t, \mathbf{w})$$

If we only restore one action, we view all the action as the fifth action. This may be inaccurate since the first three state may not do this action. Therefore, we think that we need to restore four action in each position, which will correctly reflect the real situation.

2. In the last "if" in the main function, we find that the evaluate function use the present agent to evaluate the present reward in our DQN. However, the eps of the agent isn't 0, for which it may choose the action randomly instead of using the policy of q-learning. Therefore, we can't get a precise value from our DQN and the result can't exactly reflect the real performance. If we change the eps to 0, it may be more precise.

```
avg_reward, frames = env.evaluate(obs_queue, agent, render=RENDER)
```

```
state = self.make_state(obs_queue).to(self.__device).float()
action = agent.run(state)
obs, reward, done = self.step(action)
```

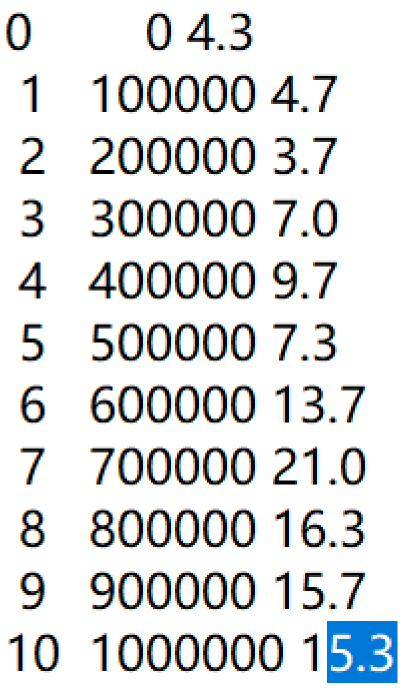
3. A very small problem in the code is that if we have already create a folder named models in our present folder, the program can not start. We can try modifying the name of folder if the folder models has already existed.

Result of our work

We compare the reward of the basic implementation and improved implementation. Since the number of steps in training is to large, we just compare the result of the first 1000000th steps.

basic implementation

```
Avg. Reward: 16.0
```



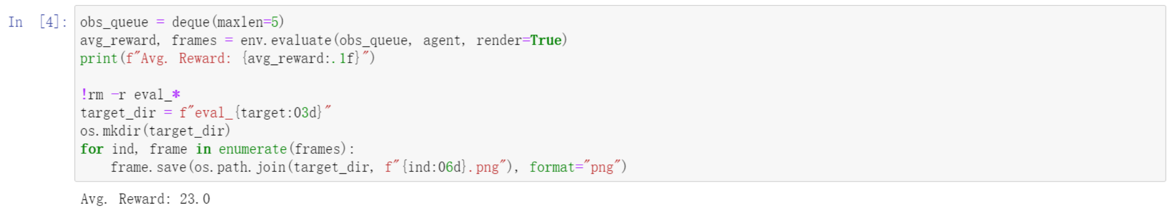
```
0        0 4.3
 1   100000 4.7
 2   200000 3.7
 3   300000 7.0
 4   400000 9.7
 5   500000 7.3
 6   600000 13.7
 7   700000 21.0
 8   800000 16.3
 9   900000 15.7
10  1000000 15.3
```

improved implementation： Priority Replay

```
Avg. Reward: 24.0
```

0	0	4.3
1	100000	2.3
2	200000	3.3
3	300000	2.7
4	400000	2.7
5	500000	4.3
6	600000	4.7
7	700000	9.0
8	800000	12.3
9	900000	16.7
10	1000000	22.4

improved implementation： Dueling DQN



```
In  [4]: obs_queue = deque(maxlen=5)
         avg_reward, frames = env.evaluate(obs_queue, agent, render=True)
         print(f"Avg. Reward: {avg_reward:.1f}")

         !rm -r eval_*
         target_dir = f"eval_{target:03d}"
         os.mkdir(target_dir)
         for ind, frame in enumerate(frames):
             frame.save(os.path.join(target_dir, f"{ind:06d}.png"), format="png")

         Avg. Reward: 23.0
```

1	100000	1.7
2	200000	2.7
3	300000	6.0
4	400000	9.7
5	500000	10.0
6	600000	11.3
7	700000	15.7
8	800000	16.0
9	900000	20.0
10	1000000	17.0
11	1100000	32.0
12	1200000	29.0
13	1300000	38.3
14	1400000	38.3
15	1500000	44.0

According to the result, we can infer that if we continue train the DQN network, the improved method will finally convergence faster than the basic implementation since the improved implementation gets more reward in the first 1000000th steps.

However, after running more steps, we find that the Dueling DQN increases faster than the other two method since it divides the Q into V(s) and A(s,a) and fully uses the advantage value. The Priority Replay method may not obviously speed up the speed of convergence and need to have more times to run the same number of steps since it needs to compute the diff value in every step. The reason is that the Priority Replay method doesn't update the diff restored in the memory in time, which causes the situation that the policy function may give the wrong priority and wrongly choose the samples to train. This causes the training inefficient.

Division of our work

Member	Ideas	Coding	Writing
廖剑雄	50%	60%	40%
麦子丰	50%	40%	60%