DRL_HW2

1070810128 常安彥

1. Model
   a. Dueling DQN

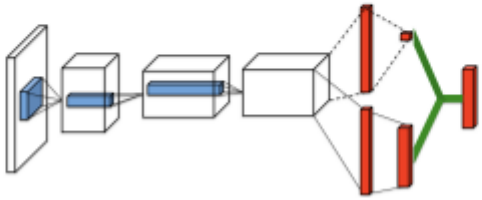
      Convolutional neural network to process the input(4 * 84 *168) observation image stack with network weight normalization and bias 0.

```
class DuelingDQN(nn.Module):
  def __init__(self, num_actions):
    super(DuelingDQN, self).__init__()
    self.conv1 = nn.Conv2d(4, 32, kernel_size=8, stride=4)
    std = math.sqrt(2.0 / (4 * 84 * 168))
    nn.init.normal_(self.conv1.weight, mean=0.0, std=std)
    self.conv1.bias.data.fill_(0.0)

    self.conv2 = nn.Conv2d(32, 64, kernel_size=4, stride=2)
    std = math.sqrt(2.0 / (32 * 3 * 8 * 8))
    nn.init.normal_(self.conv2.weight, mean=0.0, std=std)
    self.conv2.bias.data.fill_(0.0)

    self.conv3 = nn.Conv2d(64, 64, kernel_size=3, stride=1)
    std = math.sqrt(2.0 / (64 * 32 * 4 * 4))
    nn.init.normal_(self.conv3.weight, mean=0.0, std=std)
    self.conv3.bias.data.fill_(0.0)

    self.fc1 = nn.Linear(7616, 512)
    std = math.sqrt(2.0 / (64 * 64 * 3 * 3))
    nn.init.normal_(self.fc1.weight, mean=0.0, std=std)
    self.fc1.bias.data.fill_(0.0)
    self.V = nn.Linear(512, 1)
    self.A = nn.Linear(512, num_actions)

  def forward(self, states):
    x = F.relu(self.conv1(states))
    x = F.relu(self.conv2(x))
    x = F.relu(self.conv3(x))
    x = F.relu(self.fc1(x.view(x.size(0), -1)))
    V = self.V(x)
    A = self.A(x)
    Q = V + (A - A.mean(dim=1, keepdim=True))
    return Q
```

As the dueling DQN structure below, self.V function in my model is the value function as a linear layer outputting 1 dimensional state value, and self.A function in my model is the advantage function as linear layer outputting number of action dimensional, describing how much better an action is comparably.

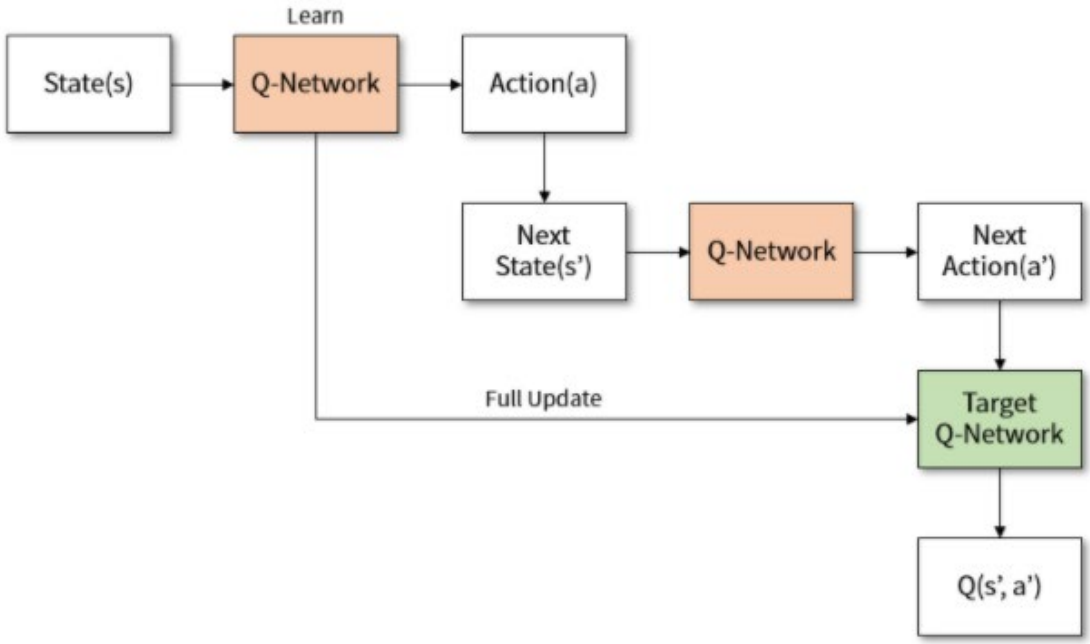
Dueling DQN

$$A_\pi(s,a) = Q_\pi(s,a) - V_\pi(s) \rightarrow Q_\pi(s,a) = V_\pi(s) + A_\pi(s,a)$$



Initialize a main neural network and a target network, separating the target and training main network, and only updating the target network after certain time steps (10000 frames in my implementation). While training, random sample form the buffer due to the relation.

```
main_nn = DuelingDQN(num_actions).to(device)
target_nn = DuelingDQN(num_actions).to(device)

optimizer = torch.optim.Adam(main_nn.parameters(), lr=1e-5)
loss_fn = nn.SmoothL1Loss()
```



Optimizer: Adam with hand adjusting learning rate

Loss function: Smooth L1 Loss which is smooth and doesn't explode the gradient

b. 4 Frame Stack Preprocessing

I stack the recent 4 frames as the input in order to get more information from the recent states, also I turned the observation into grayscale and append new observation, maintaining the maximum length as 4 with deque struct.

```
def reset(self):
    self.frameQue = deque([], maxlen=4)

def act(self, observation):
    observation = np.array(np.dot(observation, [0.299, 0.587, 0.114]), dtype=np.float32)
    if len(self.frameQue) == 0:
        for i in range(4):
            self.frameQue.append(observation)

    else:
        self.frameQue.append(observation)
    state = list(self.frameQue)
    state = np.array(state)
    state = torch.FloatTensor(np.expand_dims(state / 255., axis=0)).to(device)
    action = np.argmax(self.nn(state).cpu().data.numpy())
    return action
```

c. Uniform Buffer

I build a buffer to store an amount of (state, action, reward, next state, terminate flag) stacks, and randomly draw data out of the buffer during training steps with sample function. The buffer updates and substitute the least recent data with the incoming input when it is full with the add function.

```
class UniformBuffer(object):
    def __init__(self, size, device):
        self._size = size
        self.buffer = []
        self.device = device
        self._next_idx = 0

    def add(self, state, action, reward, next_state, done):
        if self._next_idx >= len(self.buffer):
            self.buffer.append((state, action, reward, next_state, done))
        else:
            self.buffer[self._next_idx] = (state, action, reward, next_state, done)
        self._next_idx = (self._next_idx + 1) % self._size

    def __len__(self):
        return len(self.buffer)
```

```
def sample(self, num_samples):
    states, actions, rewards, next_states, dones = [], [], [], [], []
    idx = np.random.choice(len(self.buffer), num_samples)
    for i in idx:
        elem = self.buffer[i]
        state, action, reward, next_state, done = elem
        states.append(np.array(state, copy=False))
        actions.append(np.array(action, copy=False))
        rewards.append(reward)
        next_states.append(np.array(next_state, copy=False))
        dones.append(done)
    states = torch.as_tensor(np.array(states), device=self.device)
    actions = torch.as_tensor(np.array(actions), device=self.device)
    rewards = torch.as_tensor(np.array(rewards, dtype=np.float32),
                              device=self.device)
    next_states = torch.as_tensor(np.array(next_states), device=self.device)
    dones = torch.as_tensor(np.array(dones, dtype=np.float32),
                            device=self.device)
    return states, actions, rewards, next_states, dones
```

d. Train Step

To achieve the algorithm shown below, the training step function implemented Huber loss on masked Q and target with discount factor, 0.99.

$$\delta = Q(s,a) - (r + \gamma \max_a Q(s',a))$$

The Huber loss acts like mean squared error when error is small, but like mean absolute error when error is large, making it more robust to outliers when estimates of Q are noisy.

```
def train_step(states, actions, rewards, next_states, dones):
    next_qs_argmax = main_nn(next_states).argmax(dim=-1, keepdim=True)
    masked_next_qs = target_nn(next_states).gather(1, next_qs_argmax.type(torch.int64)).squeeze()
    target = rewards + (1.0 - dones) * discount * masked_next_qs
    masked_qs = main_nn(states).gather(1, actions.type(torch.int64).unsqueeze(dim=-1)).squeeze()
    loss = loss_fn(masked_qs, target.detach())

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
    return loss
```

e. Main

Epsilon = 1.00

Batch size = 16

Discount = 0.99

Buffer size = 50000

For every episode, play the slime volleyball until one of the agents loses five lives or until the horizon of frames is reached. First, input the current

observation to the epsilon greedy function, and either returns a random sample of action space of the greedy action, action index with max value. Play the action on the environment and receives the next state, reward, terminate flag and the info. Accumulate the rewards, and append the next state to the frame queue, while adding the stack of information into the buffer. Finally, sample from buffer and run the train steps if buffer is up to the batch size.

```
my_frameQue = deque([], maxlen=4)
last_100_ep_rewards, cur_frame = [], 0
for episode in range(num_episodes+1):
  obs1 = env.reset()
  obs1 = to_gray(obs1)
  for i in range(4):
    my_frameQue.append(obs1)
  ep_reward, done = 0, False
  while not done:
    state = list(my_frameQue)
    state = np.array(state)
    state_in = torch.FloatTensor(np.expand_dims(state / 255., axis=0)).to(device)
    action = select_epsilon_greedy_action(state_in, epsilon)

    next_state, reward, done, info = env.step(action)
    ep_reward += reward
    reward = np.sign(reward)

    obs1 = to_gray(next_state)

    my_frameQue.append(obs1)
    buffer.add(state, action, reward, np.array(list(my_frameQue)), done)

    cur_frame += 1
    if epsilon > 0.01:
      epsilon -= 1.1e-6

    if len(buffer) >= batch_size:
      states, actions, rewards, next_states, dones = buffer.sample(batch_size)
      states = states.type(torch.FloatTensor).to(device) / 255.
      next_states = next_states.type(torch.FloatTensor).to(device) / 255.
      loss = train_step(states, actions, rewards, next_states, dones)
    if cur_frame % 10000 == 0:
      target_nn.load_state_dict(main_nn.state_dict())

  if len(last_100_ep_rewards) == 100:
    last_100_ep_rewards = last_100_ep_rewards[1:]
  last_100_ep_rewards.append(ep_reward)

  if episode % 25 == 0:
    print(f'Episode: {episode}/{num_episodes}, Epsilon: {epsilon:.3f}, '\
          f'Loss: {loss:.8f}, Return: {np.mean(last_100_ep_rewards):.4f}')
    if episode > 0:
      torch.save(target_nn, "dqn_weights.pt")
      torch.save(optimizer.state_dict(), "optimizer_weights.pt")
env.close()
```

2. Experimental Implementation
   a. Exploring with the default agent

      The default agent provided by the environment is very strong, barely making mistakes on serving or squelching. I trained my dueling DQN agent with the default with initial epsilon = 1.00 and decreases by 0.000001every frame.

```
Episode: 0/1200, Epsilon: 0.010, Loss: 0.0000, Return: -1.00
Episode: 25/1200, Epsilon: 0.010, Loss: 0.0000, Return: -1.08
Episode: 50/1200, Epsilon: 0.010, Loss: 0.0004, Return: -1.16
Episode: 75/1200, Epsilon: 0.010, Loss: 0.0000, Return: -1.11
Episode: 100/1200, Epsilon: 0.010, Loss: 0.0000, Return: -1.19
Episode: 125/1200, Epsilon: 0.010, Loss: 0.0000, Return: -1.19
Episode: 150/1200, Epsilon: 0.010, Loss: 0.0000, Return: -1.06
Episode: 175/1200, Epsilon: 0.010, Loss: 0.0000, Return: -0.93
Episode: 200/1200, Epsilon: 0.010, Loss: 0.0001, Return: -0.82
Episode: 225/1200, Epsilon: 0.010, Loss: 0.0000, Return: -0.79
Episode: 250/1200, Epsilon: 0.010, Loss: 0.0000, Return: -0.94
Episode: 275/1200, Epsilon: 0.010, Loss: 0.0000, Return: -1.08
Episode: 300/1200, Epsilon: 0.010, Loss: 0.0000, Return: -0.99
Episode: 325/1200, Epsilon: 0.010, Loss: 0.0000, Return: -0.89
Episode: 350/1200, Epsilon: 0.010, Loss: 0.0000, Return: -0.88
Episode: 375/1200, Epsilon: 0.010, Loss: 0.0000, Return: -0.82
Episode: 400/1200, Epsilon: 0.010, Loss: 0.0000, Return: -0.78
Episode: 425/1200, Epsilon: 0.010, Loss: 0.0000, Return: -0.91
```

      Above is the training rewards return after 4000 episodes, and finally reaches -1.00. When I tried the agent with render, I figured that the -1 match was my agent serving the ball, and later on neither the default agent nor my agent made mistakes until the horizon.

      Therefore, I had to solve the problem of serving the ball.

   b. Self-Competing

      I decided to train my agent with the agent I trained with the default agent in order to learn how to serve. Nevertheless, it learnt slowly and kept hitting the ball until horizon.

   c. Explore with random agent

      Next, to maximize the opportunity to learn how to serve, I train the agent with a random agent. Firstly, I discovered that the agent loved to run to the two edges of the court (as shown in picture below) because every time failing on serving is when the ball was served to the extremes. Therefore, I increased the epsilon to make the agent explore more.

d. Train with default agent and Itself one by one
   After the agent is familiar with serving the ball, it is not good at competing with the default agent; after training with the default agent, it becomes un familiar with serving. Finally, I decided to train with both strategies one by one.

3. References
   a. Professor's slides
   b. https://github.com/hardmaru/slimevolleygym
   c. https://github.com/MorvanZhou/Reinforcement-learning-with-tensorflow/blob/master/contents/5.3_Dueling_DQN/RL_brain.py
   d. https://www.youtube.com/watch?v=OZnhq1g9Om4&ab_channel=%E8%8E%AB%E7%83%A6Python