Reinforcement Learning Final Project

Chunlin Chen, 519021910463

June 7, 2023

1 Test Environments

1.1 Atari–Pong

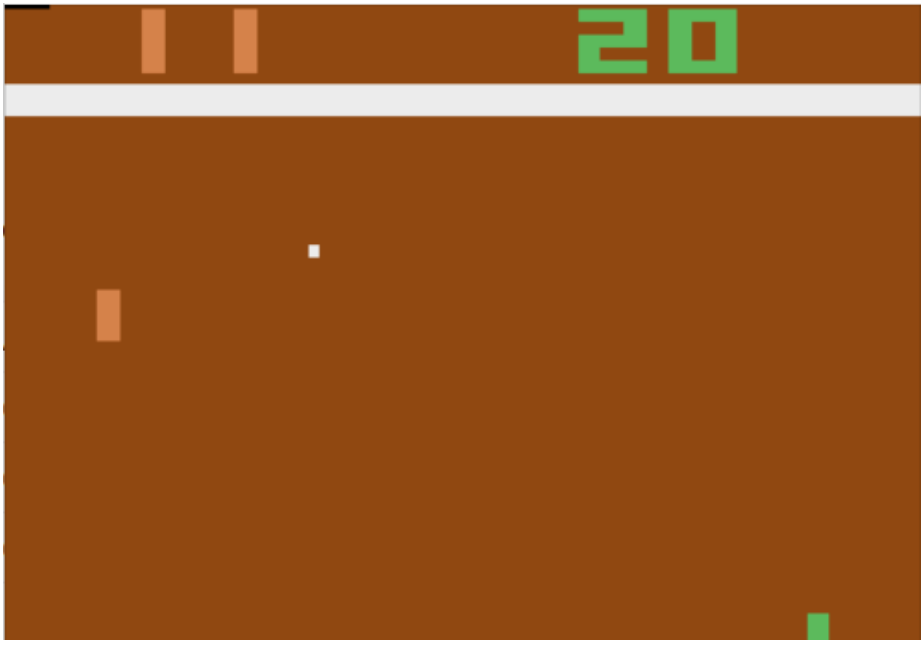

Figure 1: Pong

As shown in Figure 1, in the Pong game, the agent controls the right paddle, and competes against the left paddle controlled by the computer. The agent each tries to keep deflecting the ball away from its own goal and into the opponent's goal. The agent scores 1 point for getting the ball to pass the opponent's paddle and loses 1 point if the ball passes its own paddle. Whichever first reaches 21 points wins the game.

We keep 4 frames in each observation to be sure that we have all necessary information. Having the observation defined by the last 4 frames, we the preprocess the image, cropping and downsampling the 210*160 RGB arrays to 84*84 squares, which uses significantly less memory while still keeping all necessary information. Then we convert the colors to the grayscale, reducing the observation vector from 84*84*3 to 84*84*1. Assuming that for each observation we are going to store 4 last frames, the input shape will be 84*84*4.

1.2 Mujoco–Hopper

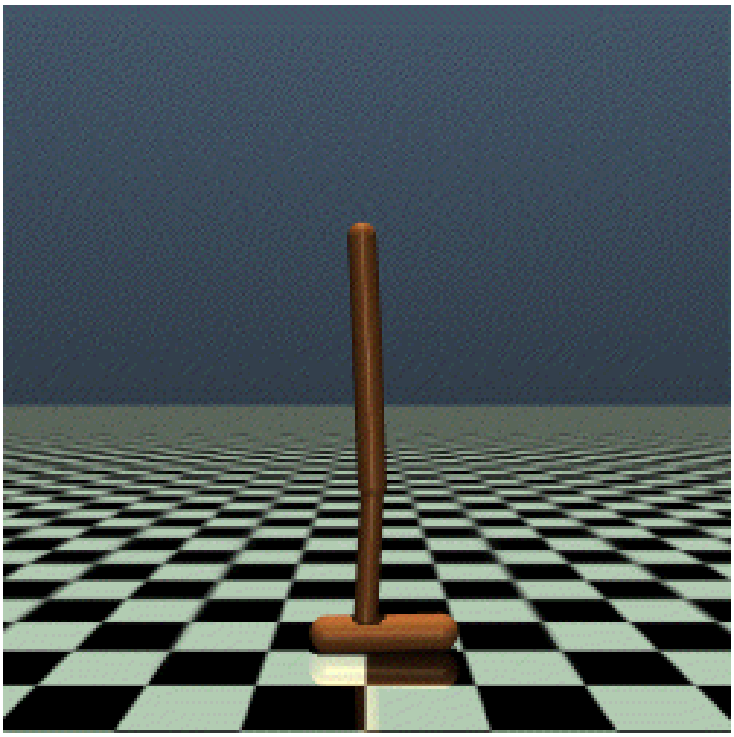

Figure 2: Hopper

As shown in Figure 2, the environment aims to increase the number of independent state and control variables as compared to the classic control environments. The hopper is a two-dimensional one-legged figure that consist of four main body parts - the torso at the top, the thigh in the middle, the leg in the bottom, and a single foot on which the entire body rests. The goal is to make hops that move in the forward (right) direction by applying torques on the three hinges connecting the four body parts.

The action space of hopper is defined as the torques applied to the thigh rotor, the leg rotor and the foot rotor, ranging from -1 to 1. And the observations

consist of positional values of different body parts of the hopper, followed by the velocities of those individual parts (their derivatives) with all the positions ordered before all the velocities.

The reward consists of three parts:

- healthy reward: Every timestep that the hopper is healthy, it gets a reward of fixed value.
- forward reward: A reward of hopping forward, this reward would be positive if the hopper hops forward (positive x direction).
- control cost: A cost for penalising the hopper if it takes actions that are too large.

The total reward returned is sum of the above three parts.

2 Algorithms

2.1 Value-based Algorithm: Double DQN

- Initialize Q-function Q, target Q-function $\hat{Q} = Q$
- In each episode
  - For each time step t
    - Given state s_t, take action a_t based on Q (epsilon greedy)
    - Obtain reward r_t, and reach new state s_{t+1}
    - Store (s_t, a_t, r_t, s_{t+1}) into buffer
    - Sample (s_i, a_i, r_i, s_{i+1}) from buffer (usually a batch)
    - Target $y = r_i + \max_a \hat{Q}(s_{i+1}, a)$
    - Update the parameters of Q to make $Q(s_i, a_i)$ close to y (regression)
    - Every C steps reset $\hat{Q} = Q$

Figure 3: DQN

DQN is an algorithm that combines the Q-learning algorithm with deep neural networks to handle high-dimensional state spaces. The key idea behind DQN is to use a deep neural network (commonly referred to as the Q-network) to approximate the Q-function, which estimates the expected future reward given a state and an action. The Q-network is trained to minimize the difference

between its predictions and the target Q-values, which are generated using the Bellman equation.

Key components of the DQN algorithm include:

- Experience Replay: To break the correlation between consecutive samples, DQN stores its past experiences in a replay buffer and samples a random mini-batch of experiences for training at each step.
- Target Network: To stabilize training, DQN uses a separate neural network (called the target network) to generate target Q-values. The target network's weights are periodically updated with the Q-network's weights.

The DQN algorithm is illustrated in Figure 3.

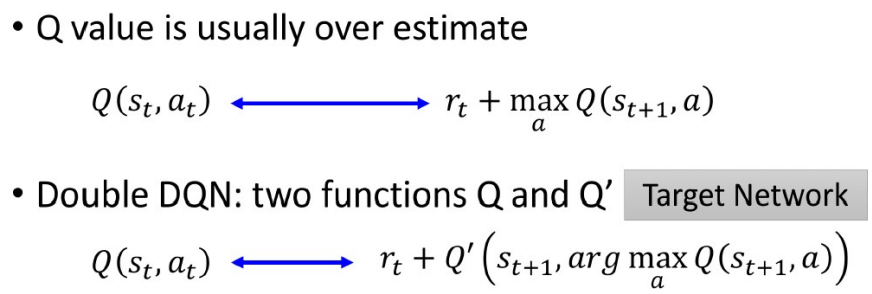

Figure 4: Double DQN

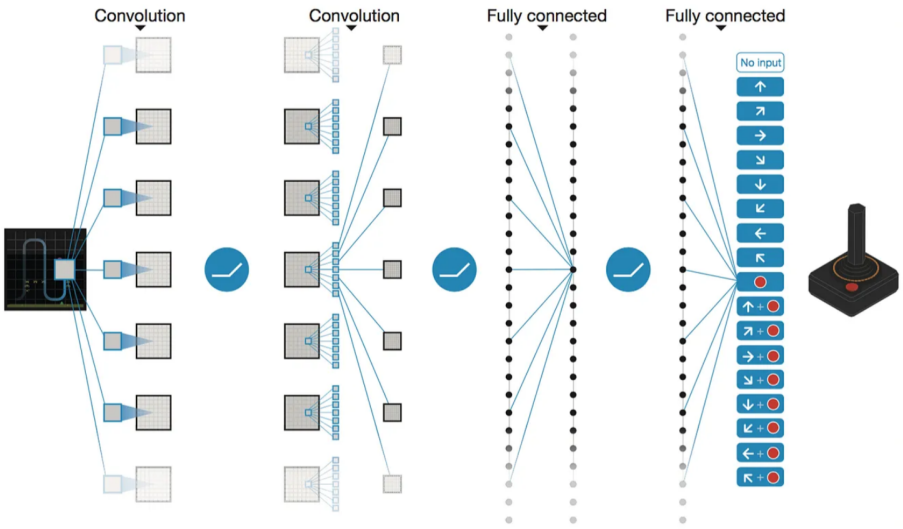

Figure 5: DDQN Network Architecture

DDQN (Double DQN) is an extension of the DQN algorithm that addresses the issue of overestimation of Q-values. This overestimation can lead to suboptimal policies and slower convergence. The key idea behind DDQN is to decouple the selection of the best action from the estimation of the Q-value for that action. As shown in Figure 4, in DDQN, the Q-network is used to select the best action, and the target network is used to estimate the Q-value of the selected action. This decoupling reduces the overestimation of Q-values and leads to better performance and stability.

As for the Atari Pong game, we use the network architecture as shown in Figure 5. The input to the neural network consists of an 84*84*4 image produced by the preprocessing map, The first hidden layer convolves 32 filters of 8*8 with stride 4 with the input image and applies a rectifier nonlinearity. The second hidden layer convolves 64 filters of 4*4 with stride 2, again followed by a rectifier nonlinearity. This is followed by a third convolutional layer that convolves 64 filters of 3*3 with stride 1 followed by a rectifier. The final hidden layer is fully-connected and consists of 512 rectifier units. The output layer is a fully-connected linear layer with a single output for each valid action.

2.2 Policy-based Algorithm

Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for t = 1, T **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in R
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$

    Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

  end for
end for

Figure 6: DDPG

Deep Deterministic Policy Gradient (DDPG) is a popular reinforcement learning (RL) algorithm that combines the strengths of deep learning and actor-

critic methods. DDPG is designed to solve continuous control tasks, where the action space is high-dimensional and continuous.

DDPG is an off-policy algorithm that employs an actor-critic architecture, where:

The actor is a neural network that learns a deterministic policy, mapping states to actions. The critic is another neural network that learns to estimate the Q-value function, evaluating the expected total reward for a given state-action pair. The primary components of the DDPG algorithm include:

- Experience Replay Buffer: This is a memory buffer that stores tuples of the form (state, action, reward, next state, done). By sampling random mini-batches from this buffer, DDPG can break the correlation between consecutive samples, stabilizing the training process.
- Soft Target Updates: DDPG maintains target networks for both the actor and critic, which are slowly updated with the weights of the main networks. This technique helps stabilize learning by preventing rapid changes in the networks' targets.
- Ornstein-Uhlenbeck Noise: To encourage exploration, DDPG introduces a temporally correlated noise process, the Ornstein-Uhlenbeck process, to the actions produced by the actor network. This noise helps the agent explore the environment more effectively.
- Deterministic Policy Gradient: DDPG uses the deterministic policy gradient theorem to update the actor network, making it suitable for continuous action spaces.

As shown in Figure 6, to train the DDPG agent, the algorithm performs the following steps:

1. Initialize the actor and critic networks.
2. Initialize the target networks with the same weights as the main networks.
3. Collect experience tuples by interacting with the environment using the current policy (actor network with added noise).
4. Store the collected experiences in the replay buffer.
5. Sample random mini-batches from the replay buffer and update the critic network by minimizing the loss (the difference between the target Q-values and the predicted Q-values).
6. Update the actor network using the deterministic policy gradient theorem. Softly update the target networks.

3 Algorithm Performance and Analysis

3.1 DDQN

We set the discouting factor $\gamma = 0.99$. The policy is ϵ-greedy with a starting $\epsilon = 1$ that decays at each step in a following rate:

$$\epsilon = 0.01 + 0.99 \times e^{-step/30000}$$

The learning rate of the Q-network is 1×10^{-4}. The target network updates every 1000 steps. The size of the replay buffer is 1×10^6 and the batch size is 32. And the networks start training when the buffer stores more than 32 transitions. We train the agent for 2000000 steps, which is approximately 1000 full episodes. The reward curve is shown in Figure 7.

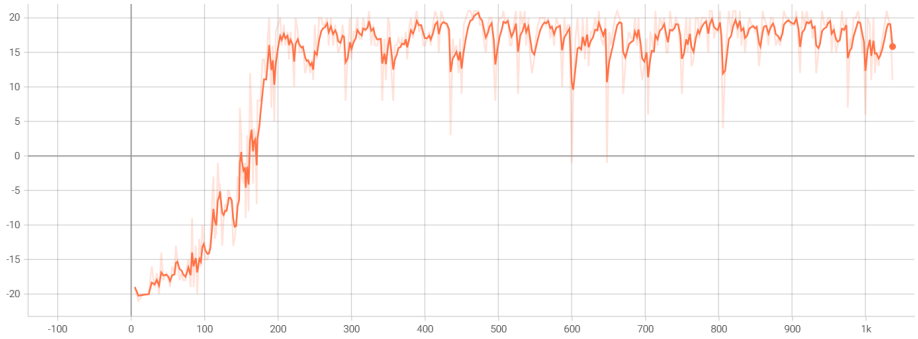

Figure 7: The reward curve of DDQN agent

```
[Test] episode:  36, episode_reward: 18.000000
[Test] episode:  37, episode_reward: 21.000000
[Test] episode:  38, episode_reward: 21.000000
[Test] episode:  39, episode_reward: 21.000000
[Test] episode:  40, episode_reward: 18.000000
[Test] episode:  41, episode_reward: 21.000000
[Test] episode:  42, episode_reward: 21.000000
[Test] episode:  43, episode_reward: 21.000000
[Test] episode:  44, episode_reward: 21.000000
[Test] episode:  45, episode_reward: 21.000000
[Test] episode:  46, episode_reward: 21.000000
[Test] episode:  47, episode_reward: 21.000000
[Test] episode:  48, episode_reward: 19.000000
[Test] episode:  49, episode_reward: 21.000000
avg reward: 19.460000
```

Figure 8: The average award of DDQN agent during testing

We can know from the curve that the reward converges to approximately 18 after 500 episodes. Then we test our model for another 50 episodes, and the average reward per episode is 19.46 as shown in Figure 8. From the results, we find that DDQN reduces the overestimation of action values, which can lead to better policy learning and improved performance. Due to the reduced overestimation bias, DDQN converges faster and to a better policy. However, DDQN introduces additional complexity and still suffers from some drawbacks, such as poor sample efficiency and reliance on epsilon-greedy exploration.

3.2 DDPG

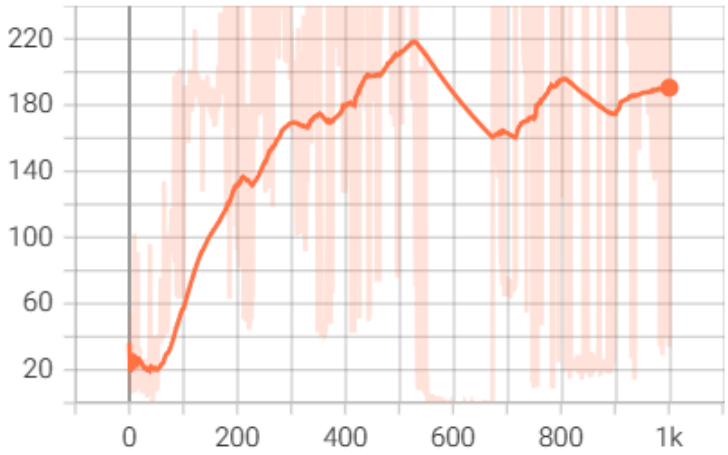

Figure 9: The reward curve of DDPG agent during training

```
episode :988, avg score : 273.1
episode :989, avg score : 35.9
episode :990, avg score : 179.3
episode :991, avg score : 235.5
episode :992, avg score : 203.1
episode :993, avg score : 201.5
episode :994, avg score : 225.1
episode :995, avg score : 377.6
episode :996, avg score : 177.4
episode :997, avg score : 177.5
episode :998, avg score : 248.0
episode :999, avg score : 210.2
```

Figure 10: The average score of DDPG agent during testing

We set the discouting factor $\gamma = 0.99$. The actor network and the critic network both consist of fully-connected layers with hidden layers of 256 neurons and their learning rate are both 3×10^{-4}. The soft update rate is 5×10^{-3}. The size of the replay buffer is 1×10^{6} and the batch size is 64. And the networks start training when the buffer stores more than 1000 transitions. We train the agent for 1000 episodes. And the reward curve is shown in Figure 9, the rewards converge to 180 after 500 episodes approximately.

Then we test our model for another 1000 episodes, and the average score is 210.2 as shown in Figure 10, which is higher than the convergence value of the trained agent.

From the reward curve in Figure 9, we find that DDPG converges fast, however, it is also arguably unstable since the reward curve has a strong oscillation. This instability is mainly due to the non-stationarity of the target policy and the value function, which are updated using the same weights for a given number of iterations before being updated again. Another limitation of DDPG is its sensitivity to hyperparameters, such as the learning rate, batch size, and discount factor. Finding the optimal set of hyperparameters can be a challenging task, and it often requires extensive experimentation and tuning to achieve a good performance.