Assignment 4

BY YUSEN ZHENG

April 11, 2023

Email: zys0794@sjtu.edu.cn

Student ID: 520021911173

1 Introduction

In this assignment, we implemented and compared the performance of **DQN, Double DQN and Dueling-DQN** and test them in a classical RL control environment — MountainCar.

2 Model Architecture

2.1 Deep Q-learning Network

In this study, we employed Deep Q-learning Network (DQN) algorithms to train an agent for solving the MountainCar problem, which is a classic reinforcement learning task. DQN is a model-free, and value-based deep reinforcement learning algorithm. It combines Q-learning with deep neural networks to approximate the action-value function, which maps states to action values.

The DQN algorithm follows a Q-learning approach, where the agent learns an action-value function, denoted as $Q(s, a)$, that maps states s to action values a. The agent uses an ϵ-greedy exploration strategy, where it selects the action with the highest Q-value with probability $(1-\epsilon)$, and selects a random action with probability ϵ, in order to balance exploration and exploitation. The DQN algorithm uses a replay buffer to store and sample experiences, and updates the neural network weights using an optimizer to minimize the mean squared error (MSE) loss between the predicted Q-values and the target Q-values. The predicted Q-values and target Q-value at iteration i is:

$$y_i^{\text{predict}} = Q(s_i, a; \theta)$$
$$y_i^{\text{target}} = r_i + \gamma \max_{a'} \hat{Q}\left(s_{i+1}, a'; \theta^-\right)$$

and the loss function is:

$$L(\theta) = \mathbb{E}[(y_i^{\text{target}} - y_i^{\text{predict}})^2]$$

where θ and θ^- denoted the parameters of Q network and target $\hat{Q}$ network. While training, using Q-network to update the target Q-network every C step. The formal description is shown in Algotirhm 1.

Algorithm 1

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 begining state 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 at in emulator and observe reward r_t and next state s_{t+1}
Store transition (s_t, a_t, r_t, s_{t+1}) in D
Sample random minibatch of transition (s_j, a_j, r_j, s_{j+1}) from D
Set $y_j = \begin{cases} r_j & \text{, if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(s_{j+1}, a'; \theta^-) & \text{, otherwise} \end{cases}$
Perform a gradient descent step on $(y_j - Q(s_j, a_j; \theta))^2$ w.r.t. the networ parameter θ
Every C steps reset $\hat{Q} = Q$
End For
End For

2.2 Double Deep Q-learning Network

We also employed a kind of improved DQN — Double Deep Q-learning Network (DDQN). In Q-learning and DQN, the max operator uses the same values to both select and evaluate an action. This can therefore lead to overoptimistic value estimates. To mitigate this problem, DDQN first finds the optimal action by applying Q network:

$$a^{\max}(s_{i+1}; \theta) = \text{argmax}_{a'} Q(s_{i+1}, a'; \theta)$$

and then calculates target Q-value by applying target $\hat{Q}$ network:

$$y_i^{\text{target}} = r_i + \gamma \hat{Q}(s_{i+1}, a^{\max}(s_{i+1}; \theta); \theta^-)$$

The formal description is shown in Algorithm 2.

Algorithm 2

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 begining state 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 at in emulator and observe reward r_t and next state s_{t+1}
Store transition (s_t, a_t, r_t, s_{t+1}) in D
Sample random minibatch of transition (s_j, a_j, r_j, s_{j+1}) from D
Define $a^{\max}(s_{j+1}; \theta) = \text{argmax}_{a'}(s_{j+1}, a'; \theta)$
Set $y_j = \begin{cases} r_j & \text{, if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(s_{j+1}, a^{\max}(s_{j+1}; \theta); \theta^-) & \text{, otherwise} \end{cases}$
Perform a gradient descent step on $(y_j - Q(s_j, a_j; \theta))^2$ w.r.t. the networ parameter θ
Every C steps reset $\hat{Q} = Q$
End For
End For

2.3 Dueling Deep Q-learning Network

We also employed another kind of improved DQN — Dueling Deep Q-learning Network (Dueling DQN) to solving this problem. Dueling DQN is an extension of the original DQN algorithm. It introduces a modification to the DQN architecture, separating the estimation of the state-value and the advantage-value, which allows the agent to learn the value of each action independently from the state.

Instead of estimating the Q-value for each action directly, the Dueling DQN separates the estimation of the state-value $V(s)$ and the advantage-value $A(s,a)$, where $V(s)$ represents the value of the state regardless of the action taken, and $A(s,a)$ represents the advantage of taking a certain action in a certain state. The Dueling DQN uses two separate streams in the neural network to estimate $V(s)$ and $A(s,a)$, and combines them to obtain the final action-value function, $Q(s,a)$, as the sum of $V(s)$ and $A(s,a)$ minus the mean of $A(s,a)$ across all actions, i.e.:

$$Q(s,a) = V(s) + \left(A(s,a) - \frac{1}{|\mathcal{A}|} \sum_{a'} A(s,a') \right)$$

where $|\mathcal{A}|$ denoted the size of action space. In this assignment, we use DDQN algorithm to train Dueling DQN architecture, i.e. Dueling Double Deep Q-learning Network.

3 Experiments

3.1 Environment

We use the `MountainCar-v0` environment provided by OpenAI gym[1]. It consists of a car placed stochastically at the bottom of a sinusoidal valley, with the only possible actions being the accelerations that can be applied to the car in either direction.

The observation space contains of two elements x and v, representing position of the car along the x-axis and velocity of the car, respectively. x is clipped to the range $[-1.2, 0.6]$ and v is clipped to the range $[-0.07, 0.07]$.

The action space contains three elements $\{0, 1, 2\}$, representing accelerate to the left, don't accelerate and accelerate to the right, respectively.

Given an action, the mountain car follows the following transition dynamics:

$$v_{t+1} = v_t + (a-1) \cdot f - \cos(3 \cdot x_t) \cdot g$$
$$x_{t+1} = x_t + v_{t+1}$$

where a denoted an action in the action space, f denoted the acceleration force and g denoted the gravity. By default, $f = 0.001$ and $g = 0.0025$. The collisions at either end are inelastic with the velocity set to 0 upon collision with the wall.

In the beginning, the position of the car is assigned a uniform random value in $[-0.6, 0.4]$. The starting velocity of the car is always assigned to 0. The goal is to reach the flag placed on top of the right hill as quickly as possible, as such the agent is penalised with a reward of -1 for each timestep. If the position of the car is greater than or equal to 0.5, then the episode end.

3.2 Training

We now experimentally compare the performance of these three DQN architectures on the MountainCar task. During the training process, we sampled 1,000 episodes, and the agent took 10,000 steps at each episode (that is, when the agent took 10,000 actions, if the episode has not ended, the episode will be truncated). During training, the exploration strategy we use is ϵ-greedy with decay:

$$\epsilon_{i+1} = \max\{\eta \epsilon_i, \epsilon_{\min}\}, \epsilon_0 = 1$$

where η denoted decay factor. This strategy means starting with a completely random exploration and multiplying the randomness of the exploration by a decay factor at each exploration step until the randomness reaches the set minimum value. In experience replay, we use a memory buffer with a capacity of 1,000,000. A batch of 64 samples is randomly sampled from memory buffer, and the Adam optimization algorithm is used to perform stochastic gradient descent on the loss function. After every 100 experience replays, use the Q network to update the target network $\hat{Q}$.

1. https://gymnasium.farama.org/environments/classic_control/mountain_car/

The hyperparameters we used are shown in the table 1.

epsidoe	max_steps	batch_size	epoch	gamma
1000	10000	64	2	0.999
epsilon_decay	epsilon_min	learning_rate	target_update	memory_size
0.999	0.001	0.001	100	1000000

Table 1. Hyperparameters

3.3 Results

In gym wiki[2], MountainCar-v0 defines "solving" as getting average reward of -110.0 over 100 consecutive trials.

3.3.1 DQN Results

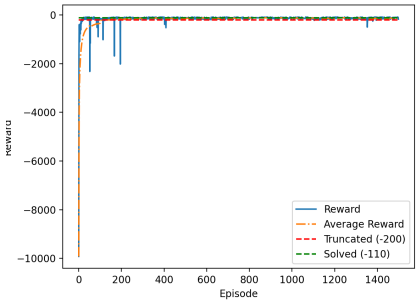

Figure 1. 1000 episode

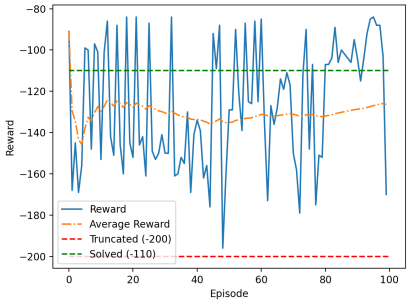

Figure 2. Last 100 episodes

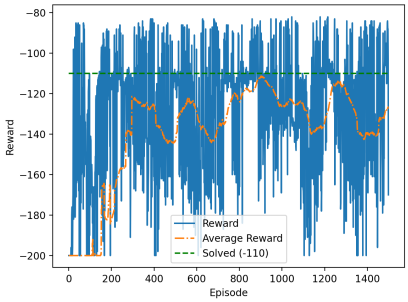

Figure 3. Truncated

Figure 4. Test

3.3.2 Double DQN Results

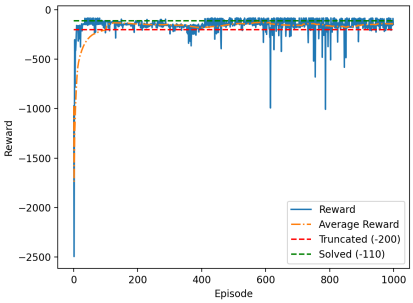

Figure 5. 1000 episode

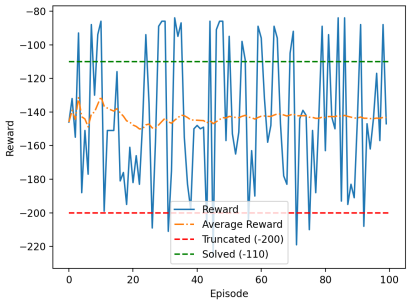

Figure 6. Last 100 episodes

2. https://github.com/openai/gym/wiki/MountainCar-v0

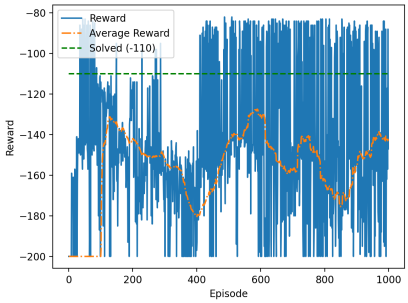

Figure 7. Truncated

Figure 8. Test

3.3.3 Dueling DQN Results

Figure 9. 1000 episode**Figure 10.** Last 100 episodes

Figure 11. Truncated**Figure 12.** Test

4 Discussion and Conclusion