Assignment 4

BY YUSEN ZHENG

April 13, 2023

Email: `zys0794@sjtu.edu.cn`

Student ID: 520021911173

1 Introduction

In this assignment, we implemented and compared the performance of three Deep Q-learning Network architecture: **DQN**, **Double DQN** and **Dueling DQN**. We tested these architectures in a classic reinforcement learning control environment: **MountainCar**. The test results demonstrate that the training outcomes of all three DQN architectures have **exceeded the benchmark defined by gym wiki**, with an average reward of -110 over a continuous period of 100 episodes.

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 $\theta$
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 $\epsilon$ 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 $\theta$
        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:

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

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 $\theta$
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 $\epsilon$ 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 $a^{\max}(s_{j+1}; \theta) = \text{argmax}_{a'} Q(s_{j+1}, a'; \theta)$
        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 $\theta$
        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. By default, the `MountainCar-v0` environment truncated an episode if the agent's step count exceeds 200 without reaching goal. But we have modified the maximum step count during training to 10,000. Because during the initial stages of training, the agent may require thousands of steps to reach the goal, and truncating the episode at 200 steps would make it difficult for the network parameters to be optimized.

During the training process, we sampled 1,000 episodes, with the agent taking at most 10,000 steps in each episode. If the episode did not end after 10,000 actions, it was truncated. For exploration, we employed an ϵ-greedy strategy with decay:

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

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

where η denotes the decay factor. This approach involves starting with a completely random exploration and multiplying the randomness of the exploration by the decay factor at each exploration step until the randomness reaches the set minimum value. For experience replay, we utilize a memory buffer with a capacity of 100,000. A batch of 64 samples is randomly sampled from the memory buffer, and the Adam optimization algorithm is employed to perform stochastic gradient descent on the loss function. In each experience replay, we perform 2 epochs (i.e., gradient descent iterations), as mentioned earlier. Additionally, after every 100 experience replays, we use the Q network to update the target network $\hat{Q}$.

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	100000

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. In the experiment, we recorded the reward of each episode during the training process (blue line), as well as the average reward of every 100 episodes (orange line). To compare with the benchmark, we also plotted the truncated (reward=-200, red line) and solved (average reward=-110, green line) standards. To ensure clarity, we separately plotted the rewards for the last 100 episodes (figure: Last 100 episodes) and all 1000 episodes, but limited the range of rewards to -200 to 0 (figure: Truncated). In addition, we tested the trained model with 100 episodes, excluding the ϵ-greedy exploration strategy during testing (figure: Test). The results are as follows.

3.3.1 DQN Results

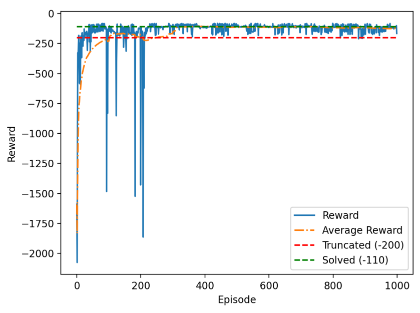

Figure 1. Train 1000 episode

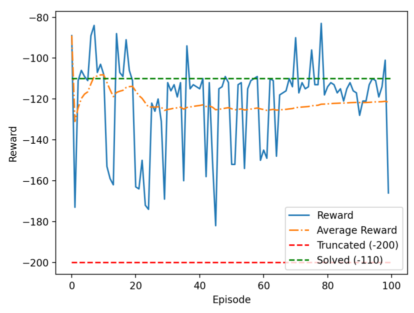

Figure 2. Last 100 episodes

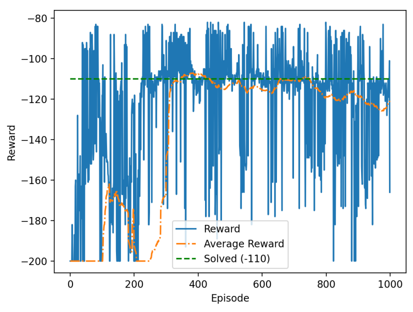

Figure 3. Truncated

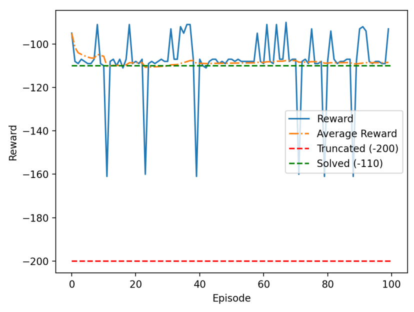

Figure 4. Test (without ϵ-greedy)

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

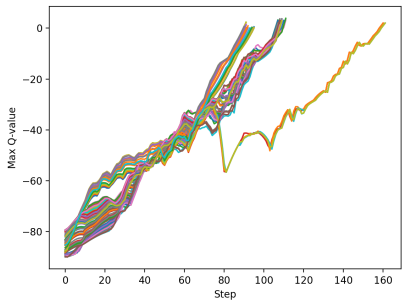

Figure 5. Max Q-value in test

3.3.2 Double DQN Results

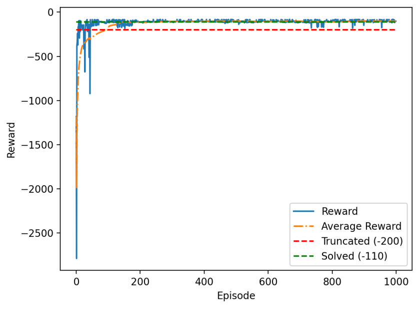

Figure 6. Train 1000 episode

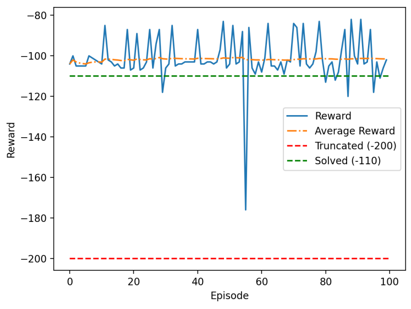

Figure 7. Last 100 episodes

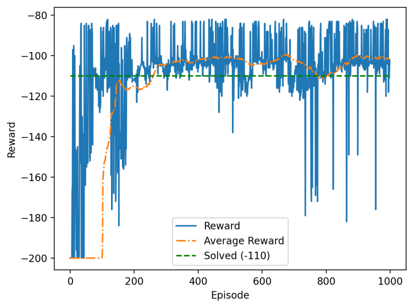

Figure 8. Truncated

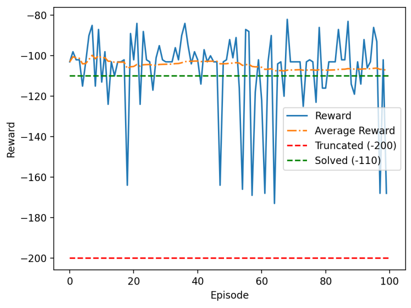

Figure 9. Test (without ϵ-greedy)

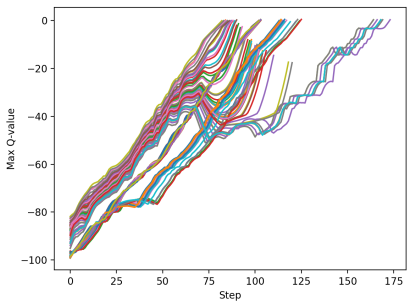

Figure 10. Max Q-value in test

3.3.3 Dueling Double DQN Results

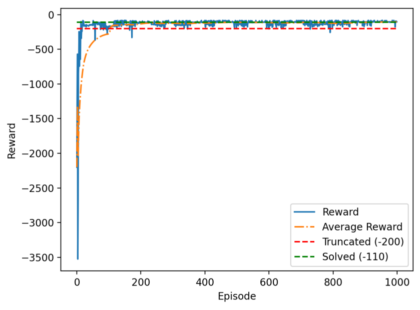

Figure 11. Train 1000 episode

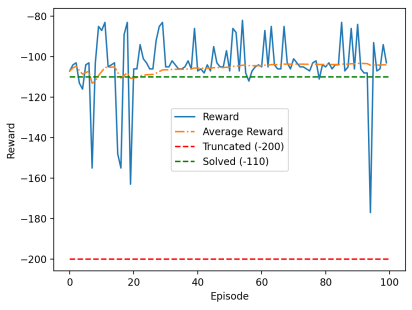

Figure 12. Last 100 episodes

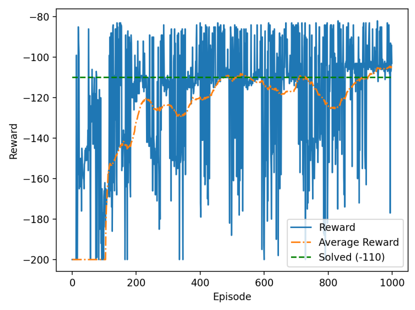

Figure 13. Truncated

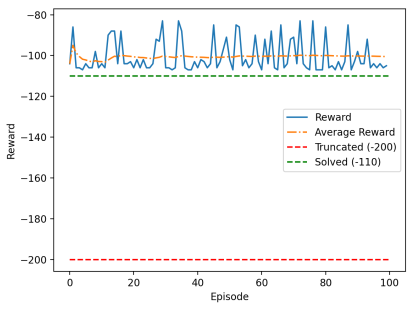

Figure 14. Test (without ϵ-greedy)

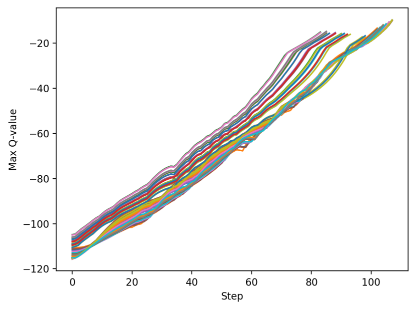

Figure 15. Max Q-value in test

4 Discussion and Conclusion

During the testing process, all three DQN architectures met the benchmark defined by the gym wiki for solving the mountaincar task, which requires the average reward over 100 consecutive episodes to be -110. Specifically, the reward achieved by the DQN architecture was around -108, slightly higher than the minimum benchmark. The rewards achieved by the DDQN and Dueling DDQN architectures were both around -100, but the Dueling DDQN architecture exhibited more stable rewards compared to the DDQN architecture, which showed larger fluctuations in rewards.

The findings suggest that the choice of DQN architecture can impact the performance of the RL agent in solving the mountaincar task. The DQN architecture, with its slightly higher rewards, may indicate that it is more effective in exploring and exploiting the environment to achieve higher rewards. On the other hand, the DDQN and Dueling DDQN architectures, with their rewards around -100, may suggest that they are capable of achieving a relatively stable performance in the task, albeit not surpassing the benchmark by a large margin.

The observation that the Dueling DDQN architecture demonstrated more stable rewards compared to the DDQN architecture may indicate that the former is better at reducing the variance in rewards during the training process. This could be attributed to the Dueling DDQN architecture's separate estimation of the value and advantage functions, which allows for a more refined estimation of the Q-values and potentially better decision-making. In contrast, the DDQN architecture, which uses a single Q-network, may be more prone to larger fluctuations in rewards due to the coupling of value and advantage estimations.

Further analysis and experimentation could be conducted to gain a deeper understanding of the underlying reasons for the observed differences in performance among the three DQN architectures. This could involve investigating the impact of hyperparameters, network architecture design, and training strategies on the performance of the RL agent. Additionally, other evaluation metrics such as convergence rate, learning stability, and sample efficiency could be considered to provide a more comprehensive assessment of the DQN architectures' performance in solving the mountaincar task.

A Codebase

A.1 Train

For training, use the following command:

```
python3 ./train.py [Model Name: DQN/DoubleDQN/DuelingDQN]
```

By default, the rewards obtained during training will be plotted and saved in the `images` folder, and the trained model will be saved with a `.pt` extension in the `models` folder.

A.2 Test

For testing, use the following command:

```
python3 ./test.py [Model Name: DQN/DoubleDQN/DuelingDQN]
```

By default, the rewards obtained during testing will be plotted and saved in the `images` folder.