Final Project

BY YUSEN ZHENG

May 23, 2023

Email: zys0794@sjtu.edu.cn

Student ID: 520021911173

1 Introduction

In this task, I implemented five classic reinforcement learning algorithms: **Deep Q-learning Network (DQN), Double Deep Q-learning Network (DDQN), Dueling Deep Q-learning Network (Dueling-DQN), Deep Deterministic Policy Gradient (DDPG), and Soft Actor-Critic (SAC)**. For the first three value-based algorithms, I tested them on **Atari** Games Environment `PongNoFrameskip-v4` and `BoxingNoFrameskip-v4`. For the latter two policy-based methods, I tested them on **MuJoCo** Continuous Control Environment `HalfCheetah-v2` and `Ant-v2`.

The training results showed that these algorithms achieved good convergence in their respective environments, demonstrating the correctness of our algorithm implementations. We also observed some differences in terms of convergence speed, stability, and other aspects among different algorithms in the same training environment. Consequently, we analyzed the possible causes of these differences and provided our insights on how to optimize algorithm performance.

2 Model Architecture

For the sake of clarity and completeness, in this section, we will provide a brief introduction to the 5 reinforcement learning algorithms used in this task, along with some details regarding their implementation.

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:

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

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.

2.4 Deep Deterministic Policy Gradient

The Deep Deterministic Policy Gradient (DDPG) algorithm is an extension of the deep Q-learning algorithm to continuous action spaces. The DDPG architecture consists of two neural networks: an actor network $\mu(s|\,\theta^\mu)$ and a critic network $Q(s,a|\,\theta^Q)$, where θ^μ and θ^Q denote the network weights, respectively.

In DDPG, the actor network is used to determine an optimal policy that maximizes the total rewards in a "deterministic" way. Unlike DQN, we use $\mu(s|\,\theta^\mu)$ to simulate the $\operatorname{argmax}_a Q(s,a)$ function. The critic network is used to calculate the Q-value. When updating the critic network, we minimize the mean squared error (MSE) loss function $\sum_i (y_i - Q(s_i, a_i|\,\theta^Q))^2$, where $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\,\theta^{\mu'})|\,\theta^{Q'})$. When updating the actor network, we calculate the policy gradient

$$pg = \frac{\partial Q(s, \mu(s|\,\theta^\mu)|\,\theta^Q)}{\partial \theta^\mu} = \frac{\partial Q(s,a|\,\theta^Q)}{\partial a} \cdot \frac{\partial \mu(s|\,\theta^\mu)}{\partial \theta^\mu}$$

and then use gradient ascent to update θ^μ, i.e. $\theta^\mu \leftarrow \theta^\mu + \alpha \cdot pg$. When updating the target network weight θ', we use a "soft update" method, i.e. $\theta \leftarrow \tau\theta + (1-\tau)\theta'$, with $\tau \ll 1$. We also incorporate exploration in DDPG. To select the policy action a, we add random noise $\mathcal{N}$, i.e. $a = \mu(s|\,\theta^\mu) + \mathcal{N}$.

Algorithm 3

```
Randomly initialize critic network $Q(s,a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights $\theta^Q$ and $\theta^\mu$.
Initialize target network $Q'$ and $\mu'$ with the 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
    Execute action $a_t$ and observe reward $r_t$ and observe new state $s_{t+1}$
    Store transtion $(s_t, a_t, r_t, s_{t+1})$ in $R$
    Sample a random minibatch of $N$ transitions $(s_i, a_i, r_i, s_{i+1})$ from $R$
```

```
        Set $y_i = \begin{cases} r_i & \text{for terminal } s_i \\ r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'}) & \text{for non-terminal } s_i \end{cases}$
        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
```

2.5 Soft Actor-Critic

3 Experiments

3.1 Environment

3.2 Training

DQN.

DDQN.

D3QN.

DDPG.

SAC.

3.3 Results

3.3.1 Atari Game

Figure 1. PongNoFrameskip-v4 **Figure 2.** BoxingNoFrameskip-v4

3.3.2 MuJoCo Contorl

Figure 3. HalfCheetah-v2 **Figure 4.** `Ant-v2`

4 Discussion and Conclusion

Acknowledgments

Bibliography

A Codebase

A.1 Train

For training, use the following command:

```
python3 ./run.py --env [Environment Name] --model [Model Name] --config [Config
Path] --mode train
```

By default, the rewards obtained during training will be saved in the `./out/datas/env_name` folder, and the trained model will be saved with a `.pt` extension in the `./out/models/env_name` folder.

A.2 Test

For testing, use the following command:

```
python3 ./run.py --env [Environment Name] --model [Model Name] --config [Config
Path] --mode test
```