CS7309 Reinforcement Learning
Final Project Report

Hongjie Fang
(Student No. 022033910104)
Department of Computer Science and Engineering
Shanghai Jiao Tong University
galaxies@sjtu.edu.cn

1 Requirements

The goal of the final project is to implement two kinds of model-free reinforcement learning methods: **value-based** and **policy-based**. In this project, we are required to choose at least one reinforcement learning methods to solve two benchmark environments: **Atari** [1] and **Mujoco** [2].

2 Method

In this section, we introduce the methods used in this project, which can be roughly classified into two categories: value-based methods and policy-based methods.

2.1 Value-Based Reinforcement Learning

Value-based methods strive to fit action value function or state value function, and it is easy to implement off-policy training mode for value-based methods. However, there are also drawbacks for value-based reinforcement learning methods, especially if we are trying to solve problems in continuous action space, where we cannot enumerate all actions under a certain space. Moreover, non-linear value function described by neural network is unstable and brittle *w.r.t.* their hyper-parameters, which requires more fine-tuning.

2.1.1 Preliminary: Markov Decision Process (MDP), Q-Function and Q-Learning

Markov Decision Process (MDP) [3] is a powerful tool to formulate the reinforcement learning problems. An MDP is defined as a 5-tuple $(\mathcal{S}, \mathcal{A}, T, R, \mathcal{E})$, where

- $\mathcal{S}$ is the agent's **state space**;
- $\mathcal{A}$ is the agent's **action space**;
- $T : \mathcal{S} \times \mathcal{A} \to \text{Prob}(\mathcal{S})$ represents the **transition dynamics**, which returns the probability that taking action a in state s will resulting in state s', if we rewrite it as $T(s, a, s')$;
- $R : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is the **reward function**, which returns the reward received when taking action a in state s, if we rewrite it as $R(s, a)$; If we take the transition uncertainty into consideration, then it can also be expressed as $R(s, a, s') : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$, which returns the reward received when taking action a in state s and resulting in state s'.
- $\mathcal{E} \subseteq \mathcal{S}$ is the set of terminal states, which once reached prevent any future action or reward.

The goal of planning in MDP is to find a policy $\pi : \mathcal{S} \to \mathcal{A}$ (deterministic representation) or $\pi : \mathcal{S} \to \text{Prob}(\mathcal{A})$ (probability distribution representation) that maximizes the expected future discounted reward when the agent chooses actions according to π in the environment. The optimal policy that maximizes the expected future discounted reward is denoted as π^*.

Now we introduce **Q-function**, $Q^\pi : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$, that defines the expected future discounted reward for taking action a in state s and then following policy π thereafter. Specially, we denote the Q-function for the optimal policy as Q^*, which can be recursively expressed as follows.

$$Q^*(s,a) = \sum_{s' \in \mathcal{S}} T(s,a,s') \left[R(s,a,s') + \gamma \max_{a' \in \mathcal{A}} Q^*(s',a') \right] \tag{1}$$

where $\gamma \in [0,1]$ is the discount factor that defines the relative value between the near-term rewards and the long-term rewards.

Once we determine the optimal Q-function, we can recover the optimal policy trivially by greedily choosing the action in the current state with the highest Q-value, that is,

$$\pi^*(s) = \arg\max_{a \in \mathcal{A}} Q^*(s,a) \tag{2}$$

Now we consider how to estimate the optimal Q-function. One distinguished approach is called **Q-Learning** [4], which begins with an arbitrary estimate Q_0 and iteratively improves its estimate by taking arbitrary actions in the environment, observing the reward and next state, and updating its Q-function using the following formula.

$$Q_{t+1}(s_t,a_t) \leftarrow Q_t(s_t,a_t) + \alpha_t \left[r_{t+1} + \gamma \max_{a' \in \mathcal{A}} Q_t(s_{t+1},a') - Q_t(s_t,a_t) \right] \tag{3}$$

where s_t, a_t, r_t are state, action and reward at time step t, and $\alpha_t \in (0,1]$ is a step size smoothing parameter. Under the following conditions, Q-Learning method is guaranteed to converge at Q^*:

1. The Q-function estimate is represented tabularly, that is, a value is associated with each unique state-action pair,
2. the agent visits each state and action infinitely often,
3. and $\alpha_t \to 0$ as $t \to \infty$.

In practice, we usually encounters the infinite state-space problems. Under this circumstances, the Q-function is usually approximated by a certain function, *e.g.*, neural networks, instead of tabular representation. This trick may causes divergence of the learning, since it violates condition 1 stated before, but if we carefully use the function approximation, the performances are also well guaranteed.

2.1.2 Deep Q-Network (DQN)

Deep Q-Network (DQN) [5] is based on classic Q-Learning algorithm [4], and with the following innovations (or special techniques) [6]:

- **Neural Network for Q-Function**. It uses a deep convolutional neural network architecture for Q-function approximation.
- **Target Network**. It uses older network parameters to estimate the Q-values of the next state. According to Q-function,

  $$Q^\pi(s_t,a_t) = r_t + Q^\pi(s_{t+1}, \pi(s_{t+1})) \tag{4}$$

  where the left-hand-side is the output of the network, and the right-hand-side is the target. However, since the target also includes the network output $Q^\pi(\cdot,\cdot)$, it is varying during learning, which introduces training difficulties. Therefore, we can fix the right-hand-side network (target network) during several steps' learning, and copy the parameters of the left-hand-side network (policy network) to the target network every several steps to keep the target network up-to-date.
- **Experience Replay**. It uses mini-batches of random training data rather than the single-step updates on the last experience. In detail, a replay buffer is constructed, and the data collected from the interaction between the strategy and the environment (*i.e.* state $\to$ action $\to$ state $\to$ action $\to \cdots$) will be put into the replay buffer for future replay. When training the Q function, we will pick a random batch from the replay buffer, and use this mini-batch to update the Q function. It has the following advantages: first, the experiences from the replay buffer may come from different strategies, resulting in a variety of the training data; second, it reduces the interaction times between the strategy and the environment, which makes data usage efficient.

Now we consider the exploration problem, which is also known as the exploration-exploitation dilemma. When we use the Q-function to determine the action in the next step, the strategy is fully based on the Q-function, that is, we will take the action with the highest Q-value to execute. This strategy will result in taking the same action all the time. Therefore, we need exploration techniques, two common tricks are ϵ**-Greedy** and **Boltzmann Exploration**.

In ϵ**-Greedy** algorithm, we define the policy as follows.

$$\pi(a|s) = \begin{cases} \max\limits_{a \in \mathcal{A}} Q^{\pi}(s,a) & \text{with probability } 1-\epsilon \\ \text{random} & \text{with probability } \epsilon \end{cases} \tag{5}$$

We usually decrease ϵ as learning continues. Finally, ϵ will be decreased into a small number. Use the ϵ-Greedy algorithm, the DQN will explore more in the beginning of the training, and exploit more in the following learning process. The probability ϵ prevents the policy determined by the Q-function from running into the exploration problem.

In **Boltzmann Exploration**, we define the policy as the probability distribution, where

$$\pi : \text{Prob}(a|s) = \frac{\exp Q(s,a)}{\sum_{a' \in \mathcal{A}} \exp Q(s,a')} \tag{6}$$

Therefore, the Q-function determine the probability distribution of our policy. The probability here are strictly positive, which can also prevents the policy determined by the Q-function from running into the exploration problem.

In our implementation, we use the ϵ-Greedy algorithm and the linear decay method for ϵ. The pseudo-code for our implemented DQN (with ϵ-greedy) is shown as follows.

Algorithm 1 DQN with ϵ-Greedy

Initialize function $\varphi(\cdot)$ to process the observation o into state s, *i.e*, $s = \varphi(o)$.
Initialize replay buffer D of capacity N (**experience replay**).
Initialize action-value function Q with random weights θ (**neural network**).
Initialize target action-value function $\hat{Q}$ with weights $\hat{\theta} \leftarrow \theta$ (**target network**).
for episode from 1 to M **do**
  Initialize sequence head $s_1 \leftarrow \varphi(o_1)$.
  for time step from 1 to T' **do**
    With probability ϵ select random action a_t, otherwise $a_t \leftarrow \arg\max_a Q_\theta(s_t, a)$ (ϵ**-greedy**).
    Execute action a_t in the emulator and observe reward r_t and observation o_{t+1}.
    Process the next state $s_{t+1} \leftarrow \varphi(o_{t+1})$.
    Store experience (s_t, a_t, r_t, s_{t+1}) in replay buffer D (**experience replay**).
    Sample a random mini-batch of the experiences (s_j, a_j, r_j, s_{j+1}) from D (**experience replay**).

$$\text{Set } y_j \leftarrow \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max\limits_{a'} \hat{Q}_{\hat{\theta}}(s_{j+1}, a') & \text{otherwise} \end{cases} \quad \text{(target network)}.$$

    Perform a gradient descent step on loss of y_j and $Q_\theta(s_j, a_j)$ *w.r.t.* the weights θ.
    Every C steps set the target network to be the policy network, *i.e.*, $\hat{Q}_{\hat{\theta}} \leftarrow Q_\theta$ (**target network**).
    Every E steps linearly decay the ϵ, until reaching ϵ_{min} (ϵ**-greedy**).
  end for
end for

The action-value function Q is implemented with the convolutional neural network following [5].

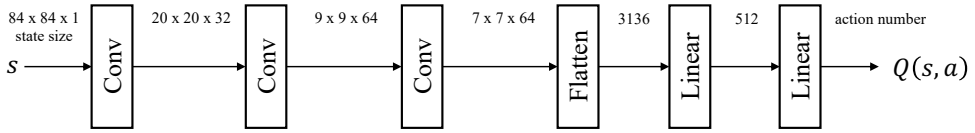

Figure 1: The Architecture of the Convolutional Neural Network in DQN

2.1.3 Double Deep Q-Network (Double-DQN)

Double Deep Q-Network (Double-DQN) [7, 8] is motivated by the fact that in original DQN [5], Q value is usually overestimated. The problem raises from the property of Q-function: the action that has a high Q-value will be selected frequently, which results in the higher Q-value. To solve the problem, we can use another function Q' parameterized by a neural network as the target network to compute the Q-value, that is,

$$Q(s_t, a_t) = r_t + Q'(s_{t+1}, \arg\max_{a \in \mathcal{A}} Q(s_{t+1}, a)) \tag{7}$$

Double-DQN can be regarded as the DQN with another implementation of the target network. As the result, the implementation of Double-DQN is similar with the implementation of DQN, except that we calculate the y_j according to Eqn. (7) instead of Eqn. (4). The architecture of the target network Q' is the same as the architecture of the policy network Q shown in Fig. 1.

2.1.4 Dueling Deep Q-Network (Dueling-DQN)

Dueling Deep Q-Network (Dueling-DQN) [9] further improves DQN simply in terms of the network architecture. Original DQN directly regress the Q-value using the convolutional neural network, however, this may cause unstable learning process. To solve the problem and improve the performance, Dueling-DQN proposes to decompose the Q-value $Q(s, a)$ into two parts: the value part $V(s)$ that only relates to the current state, and the advantage part $A(s, a)$ that relates to the current state and the current action. For example, in the Atari `BreakoutNoFrameskip-v4` game [1] (the game aims to use the reflective ball to break the bricks), the value part focus more on the brick status, which determines the difficulties of the current state, and the advantage part focus more on the ball to prevent missing the reflective ball.

Formally, besides the action-value function which is essentially the conventional Q-function $Q^\pi(s, a)$, we define the state-value function as follows.

$$V^\pi(s) = \mathbb{E}_{a \in \mathcal{A}}[Q^\pi(s, a)] \tag{8}$$

The optimal state-value function can be defined similarly with the optimal action-value function.

$$V^*(s) = \max_\pi V^\pi(s), \quad \textit{c.f. } Q^*(s, a) = \max_\pi Q^\pi(s, a) \tag{9}$$

Therefore, we can define the optimal advantage function as $A^*(s, a) = Q^*(s, a) - V^*(s)$, which gives us an interesting property that $\max_{a \in \mathcal{A}} A^*(s, a) = 0$ and therefore we can derive

$$Q^*(s, a) = V^*(s) + A^*(s, a) - \max_{a \in \mathcal{A}} A^*(s, a) \tag{10}$$

Therefore, we can parameterize $V^*(s)$ and $A^*(s, a)$ as neural networks respectively. In practice, we use the average operation to replace the maximum operation to further improve the performances and stablize the learning, *i.e.*,

$$Q_{\theta,\phi}(s, a) = V_\theta(s) + A_\phi(s, a) - \frac{1}{|\mathcal{A}|} \sum_{a \in \mathcal{A}} A_\phi(s, a) \tag{11}$$

Therefore, we only need to modify the network architecture of the DQN to get Dueling-DQN. We predict the value part $V_\theta(s)$ and the advantage part $A_\phi(s, a)$ respectively, and use Eqn. (11) to combine them into the Q-value $Q_{\theta,\phi}(s, a)$. The network architecture is illustrated in Fig. 2. The rest part remains the same with DQN.

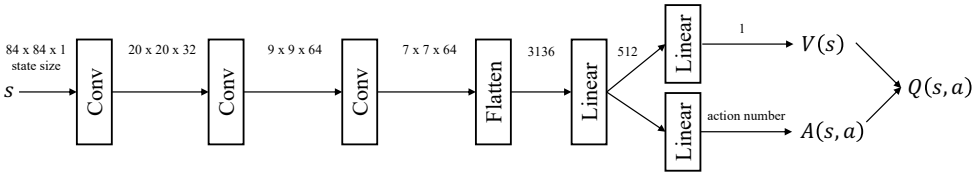

Figure 2: The Architecture of the Convolutional Neural Network in Dueling-DQN

2.2 Policy-Based Reinforcement Learning

On the contrary to value-based reinforcement learning methods, policy-based reinforcement learning methods usually solves problems in continuous action space. Current policy-based methods usually strive to introduce off-policy mode to Actor-Critic. However, it also has one biggest drawback: sample inefficiency.

2.2.1 Deep Deterministic Policy Gradient (DDPG)

Deep Deterministic Policy Gradient (DDPG) algorithms is essentially DQN algorithm under the Actor-Critic framework. Unlike DQN (with ϵ-greedy algorithm), DDPG outputs a deterministic policy π. Here "deterministic" refers to the fact that the output of continuous action space is a specific value (action). When the action space is discrete, the policy outputs the probability distribution of each action using Q-values based on the objective of maximizing long-term returns; while when the action space is continuous, the output can only be a specific value representing a specific action in the pursuit of the objective of maximizing long-term returns, thus becoming a deterministic strategy.

There are two networks in DDPG: actor network $\mu_\phi(s)$ with weight ϕ and critic network $Q_\theta(s,a)$ with weight θ, and two target networks of actor network and critic network respectively. We use the actor network $\mu_\phi(s)$ to replace the original policy. The updating of critic network remains the same, and the updating of the actor network is based on the following formula.

$$\nabla_\phi J = \nabla_a Q_\theta(s,a)|_{s=s_j, a=\mu(s_j)} \nabla_\phi \mu(s)|_{s=s_j} \tag{12}$$

Similar to DQN, we use replay buffer for experience replay. Also recall that DQN use ϵ-greedy or Boltzmann exploration to solve the exploration-exploitation dilemma. Here we can solve the problem simply by adding random noises $\mathcal{N}$ to the action sequences, *i.e.*, $a = \mu_\phi(s) + \mathcal{N}$. Another trick that DDPG uses is the soft updating method, *i.e.*, instead of update the target networks in a fixed frequency, DDPG use hyper-parameter τ to control the update fraction each time, and update the target networks by $\theta \leftarrow \tau\theta + (1-\tau)\hat{\theta}$.

The pseudo-code for our implemented DDPG is shown as follows.

Algorithm 2 DDPG

Initialize function $\varphi(\cdot)$ to process the observation o into state s, *i.e.*, $s = \phi(o)$.
Initialize replay buffer D of capacity N.
Initialize critic network $Q_\theta(s,a)$ and actor $\mu_\phi(s)$ with random weights θ and ϕ.
Initialize target network $\hat{Q}$ and $\hat{\mu}$ with weights $\hat{\theta} \leftarrow \theta$ and $\hat{\mu} \leftarrow \mu$.
for episode from 1 to M **do**
  Initialize a random process $\mathcal{N}$ for action exploration.
  Initialize sequence head $s_1 \rightarrow \phi(o_1)$.
  for time step from 1 to T' **do**
    Select action $a_t \leftarrow \mu_\phi(s_t) + \mathcal{N}_t$ according to current policy and exploration noise.
    Execute action a_t in the emulator and observe reward r_t and observation o_{t+1}.
    Process the next state $s_{t+1} \leftarrow \varphi(o_{t+1})$.
    Store experience (s_t, a_t, r_t, s_{t+1}) in replay buffer D.
    Sample a random mini-batch of the experiences (s_j, a_j, r_j, s_{j+1}) from D.
    Set $y_j \leftarrow \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \hat{Q}_{\hat{\theta}}(s_{j+1}, \hat{\mu}_{\hat{\phi}}(s_{i+1})) & \text{otherwise} \end{cases}$.
    Perform a gradient descent step on loss of y_j and $Q_\theta(s_j, a_j)$ *w.r.t.* the critic weights θ.
    Update the actor policy by the sampled policy gradient $\nabla_a Q_\theta(s,a)|_{s=s_j, a=\mu(s_j)} \nabla_\phi \mu(s)|_{s=s_j}$.

    Update the target networks using soft updating method, *i.e.*,

$$\theta \leftarrow \tau\theta + (1-\tau)\hat{\theta}, \quad \phi \leftarrow \tau\phi + (1-\tau)\hat{\phi}$$

  end for
end for

The architectures of actor network and critic network are illustrated in Fig. 3. Since experiment environment (Mujoco [2]) produces one-dimensional state, we use fully-connected neural network instead of convolutional neural network to implement the networks in DDPG.

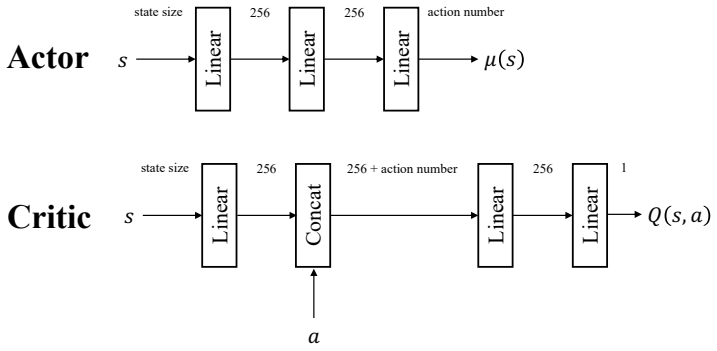

Figure 3: The Architecture of the Convolutional Neural Network in DDPG (and also TD3)

2.2.2 Twin Delayed Deterministic Policy Gradient (TD3)

Twin Delayed Deterministic Policy Gradient (TD3) [10] is actually the DDPG algorithm with the following critical optimizations.

- **Clipped Double-Q Learning**. The TD3 algorithm learns two Q-functions independently and uses the smaller Q-values to construct the target value for the learning process of the critic network, in order to reduce the critic's overestimation.
- **Delay Policy Update**. The update of actor network is slower than the update of critic network. Only when the temporal-different loss is relatively small, we update the actor network. This can reduce the training time and reduce the variance during learning.
- **Target Policy Smoothing**. TD3 adds noise to the target action when constructing the target value to aid critic learning. The inspiration for this approach is that there should not be much difference in the scores obtained by taking similar actions in the same state.

We use the same architectures of the actor and critic networks as those in DDPG, shown in Fig. 3.

3 Experiments

We use two different simulation environments to benchmark two different kinds of model-free reinforcement learning methods. Following previous works,

- For value-based methods, we benchmark them in Atari [1].
- For policy-based methods, we benchmark them in Mujoco [2].

3.1 Value-Based Experiments in Atari

3.1.1 Setup

Environments We use three environments in Atari [1] games to benchmark the value-based methods.

- `PongNoFrameskip-v4` (left, *abbrev.*, `Pong`): You control the right paddle, you compete against the left paddle controlled by the computer. You each try to keep deflecting the ball away from your goal and into your opponent's goal.
- `BreakoutNoFrameskip-v4` (middle, *abbrev.* `Breakout`): The dynamics are similar to `Pong` games described before. You move a paddle and hit the ball in a brick wall at the top of the screen. Your goal is to destroy the brick wall. You can try to break through the wall and let the ball wreak havoc on the other side, all on its own! You have 5 lives.

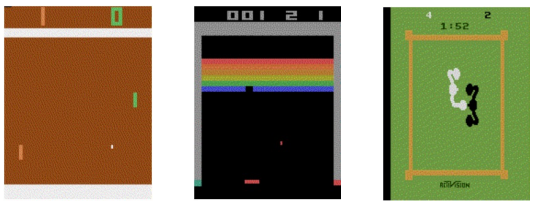

Figure 4: The Atari Games used in Experiments of Value-Based Methods

- `BoxingNoFrameskip-v4` (right, *abbrev.* `Boxing`): You fight an opponent in a boxing ring. You score points for hitting the opponent. Anyone who scores 100 points wins the game.

Hyper-Parameters Now we introduce the hyper-parameter settings for every experiment environment. Notice that it is a little bit different from the hyper-parameter settings in the original paper [5, 7, 8, 9], due to the limited computational resources. Therefore, the results might be inferior than the results in the original paper, but they are still enough for analyses and comparisons.

- For `Pong` game, we set the replay buffer capacity $N = 5 \times 10^3$, the total number of time steps $T = 10^6$ and the number of episode is approximately $M \approx 500$. The batch size $B = 32$, the learning rate is set to 10^{-4}, and the discount factor $\gamma = 0.99$. ϵ is set to $\epsilon_{\text{max}} = 1$ in the beginning and linearly decays to $\epsilon_{\text{min}} = 0.01$ in 10^5 steps ($\epsilon_{\text{frac}} = 0.1$ of total steps). Every step of training is followed by a step of optimization, and every 1000 steps of training is followed by a target network update.
- For `Breakout` game, we set the replay buffer capacity $N = 10^5$, the total number of time steps $T = 4 \times 10^6$ and the number of episode is approximately $M \approx 2.2 \times 10^4$. Every 4 steps of training is followed by a step of optimization. Other settings remains the same as the `Pong` game.
- For `Boxing` game, we set the replay buffer capacity $N = 10^4$, the total number of time steps $T = 3 \times 10^6$ and the number of episode is approximately $M \approx 3000$. Every 4 steps of training is followed by a step of optimization. Other settings remains the same as the `Pong` game.

Pre-processing We pre-process the $210 \times 160 \times 3$ RGB images (observations) into a $84 \times 84 \times 1$ gray-scale images (states) to reduce the calculation times and improves efficiency, following [5].

Loss We use L2 loss in the experiments. We conduct experiments about different types of loss in the following ablation section.

Optimizer We use RMSprop optimizers following original DQN paper [5]. We conduct experiments about different optimizers in the following ablation section.

3.1.2 Results and Analysis

We train 3 different agents (DQN [5], Double-DQN [7, 8] and Dueling-DQN [9]) in 3 different environments mentioned before and then evaluate the performances of the agents.

How well can the agents play? After training the agents, we evaluate the ability of the agents by letting them to play several episodes of the Atari games, and summarize the average score, minimum score and maximum score among these episodes. The results are shown in Tab. 1. From the results we can make the following analyses.

- For Atari `Pong` environment, **both Double-DQN and Dueling-DQN can improve the performance to a relatively large extent**, which is directly reflected by the average score in evaluation. Among these value-based methods, Dueling-DQN achieves the best performances. Furthermore, **both of them can improve the stability of the agents**, which is reflected by the standard deviation (as well as minimum and maximum) results in evaluation.

Table 1: Evaluation (100 Episode) of Different Methods in Atari Games

Atari Game	Methods	Score				
		Avg. ↑	Median ↑	Min. ↑	Max. ↑	Std. ↓
Pong	DQN	14.98	**20.0**	-1.0	20.0	8.52
	Double-DQN	17.62	17.0	17.0	19.0	0.80
	Dueling-DQN	**20.15**	**20.0**	**19.0**	**21.0**	**0.64**
Breakout	DQN	13.97	6.5	**1.0**	48.0	13.63
	Double-DQN	13.86	12.0	**1.0**	37.0	**9.53**
	Dueling-DQN	**24.66**	**20.0**	0.0	**55.0**	19.54
Boxing	DQN	61.77	61.0	27.0	83.0	11.87
	Double-DQN	**84.75**	**84.0**	**70.0**	**98.0**	**8.94**
	Dueling-DQN	82.10	81.0	64.0	96.0	9.44

- For Atari Breakout environment, **Dueling-DQN can improve the performances of DQN by a large margin**, according to the average score results in evaluation. Double-DQN achieves the relatively same results as DQN, but **it stablizes the performances of DQN**, which is reflected by the standard deviation results in evaluation.
- For Atari Boxing environment, **both Double-DQN and Dueling-DQN can improve the performances of DQN by a large extent**. Among these value-based methods, Double-DQN achieves the best performances. Furthermore, both of them can improve the stability of the agents, which is reflected by the standard deviation (as well as minimum and maximum) results in evaluation.

Generally speaking, we believe that **in most cases Double-DQN and Dueling-DQN can improve the performance of vanilla DQN**. Moreover, **they can also make the agent's ability more stable**.

Are the agents difficult to train? We also summarize the higest score and the highest 10-episode average score after different episodes' training of different approaches in Tab. 2 and Tab. 3. Both of them can reflects the training stabilities of the agents. We also visualize the reward curves during training of Pong and Boxing in Fig. 5, which reflects the training difficulties directly. From the results we can make the following analyses.

Table 2: Highest Score (↑) after Different Episodes' Training of Different Methods in Atari Games

Methods	Pong			Breakout			Boxing		
	100	300	500	5000	10000	20000	1000	2000	3000
DQN	-8.0	**21.0**	**21.0**	12.0	78.0	101.0	79.0	92.0	92.0
Double-DQN	-8.0	20.0	**21.0**	10.0	90.0	97.0	**97.0**	**98.0**	**98.0**
Dueling-DQN	**-2.0**	**21.0**	**21.0**	**15.0**	**92.0**	**102.0**	95.0	97.0	**98.0**

Table 3: Highest 10-Episode Average Score (↑) after Different Episodes' Training of Different Methods in Atari Games

Methods	Pong			Breakout			Boxing		
	100	300	500	5000	10000	20000	1000	2000	3000
DQN	-13.3	19.1	**20.6**	3.9	21.1	28.1	59.4	79.6	79.6
Double-DQN	-11.0	17.4	19.3	3.9	23.3	29.3	**86.9**	**92.9**	**93.5**
Dueling-DQN	**-8.7**	**19.8**	20.1	**4.8**	**29.3**	**29.4**	77.6	86.8	88.9

For Atari Pong environment,

- **Dueling-DQN can accelerate learning process effectively** (its reward curve goes up before other's reward curves in Fig. 5, and it has the biggest highest score and highest 10-episode average score in Tab. 2 and Tab. 3), however, **Double-DQN slows down the learning process**.
- **Dueling-DQN can help stablize the learning process** (its reward curve is less oscillating than the curve of DQN in Fig. 5), however, **Double-DQN may causes more training instability in this environment**.

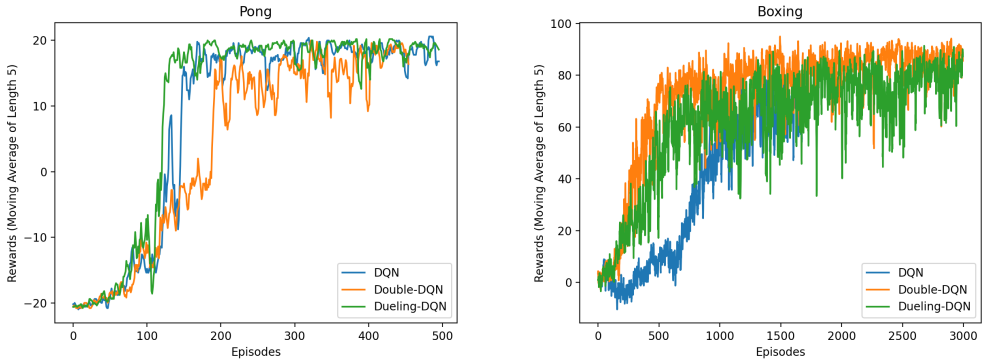

Figure 5: The Training Curves of Different Methods in `Pong` and `Boxing`

For Atari `Breakout` environment,

- **Double-DQN can accelerate the learning process of DQN by a small margin**, and **Dueling-DQN can further improve the learning speed and stability** (it has the biggest highest scores and the highest 10-episode average scores in Tab. 2 and Tab. 3 are similar).

For Atari `Boxing` environment,

- **Both Double-DQN and Dueling-DQN can accelerate learning process effectively** (their reward curves goes up before DQN reward curve in Fig. 5, and they have bigger highest score and highest 10-episode average score than DQN in Tab. 2 and Tab. 3).
- **Double-DQN can help stablize the learning process** (its reward curve is less oscillating than the curve of DQN in Fig. 5).

Generally speaking, we believe that **Double-DQN and Dueling-DQN can accelerate and stablize the learning process**, which means that the optimizations taken by Double-DQN and Dueling-DQN makes the agents easier to train than DQN.

3.1.3 Ablations and Analysis

Loss Type In the previous experiments, we use L2 loss. In this ablation we want to explore the results of using different types of loss, such as smooth L1 loss. We conduct the experiments only in environment `Pong` for convenience, and the results are shown in Tab. 4.

Table 4: Ablation of Loss Type

Methods	**Loss Type**	**Score** of Game `Pong` (100 episodes)				
		Avg. ↑	Median ↑	Min. ↑	Max ↑	Std. ↓
DQN	Smooth L1	4.79	**20.0**	-21.0	**21.0**	18.53
DQN	L2	**14.98**	**20.0**	**-1.0**	20.0	**8.52**
Double-DQN	Smooth L1	**18.94**	**19.0**	**18.0**	**20.0**	0.89
Double-DQN	L2	17.62	17.0	17.0	19.0	**0.80**
Dueling-DQN	Smooth L1	20.00	**20.0**	**20.0**	20.0	**0.00**
Dueling-DQN	L2	**20.15**	**20.0**	19.0	**21.0**	0.64

- From the standard deviation results we can see that **learning with L2 loss can stablize the ability of the agent**, because the curve of the L2 loss is smoother than that of the smooth L1 loss, making the agents easier and stabler to learn.
- We also observe that learning with **L2 loss can improve the performance of DQN significantly**, but **it might worsen the performance of Double-DQN**.

Generally speaking, **L2 loss is more appropriate than L1 loss during learning**, which is the reason that we use L2 loss in the experiments.

Optimizer Type In the previous experiments, we use the RMSprop optimizers following the original DQN paper [5]. However, recent years have witnessed several new kinds of optimizers. In this ablation we want to explore the results using different types of optimizers, such as Adam optimizers. For all optimizers in this experiment, we only change the learning rate while keeping all other hyper-parameters default. We conduct the experiments only in environment `Pong` for convenience, and the results are shown in Tab. 5. From the results we can see that **learning with modern optimizers like Adam further improve the performances of DQN and Double-DQN by an apparent margin, but it will degrade the performance of Dueling-DQN**. The DQN and Double-DQN agents trained with Adam performs stabler and better than the agents trained with RMSprop. Therefore, in the following policy-based experiments, since DDPG is more similar with DQN than Dueling-DQN, we will use Adam-based optimizer AdamW [11] as our default optimizer; but in the previous value-based experiments, we continue to use RMSprop in order to align with the original DQN paper [5].

Table 5: Ablation of Optimizer Type

Methods	**Optimizer Type**	**Score of Game** `Pong` (100 **episodes)**				
		Avg. ↑	Median ↑	Min. ↑	Max ↑	Std. ↓
DQN	RMSprop	14.98	**20.0**	-1.0	20.0	8.52
DQN	Adam	**19.38**	19.0	**17.0**	**21.0**	**1.54**
Double-DQN	RMSprop	17.62	17.0	17.0	19.0	0.80
Double-DQN	Adam	**20.58**	**21.0**	**20.0**	**21.0**	**0.49**
Dueling-DQN	RMSprop	**20.15**	**20.0**	**19.0**	**21.0**	**0.64**
Dueling-DQN	Adam	18.07	**20.0**	10.0	20.0	4.06

3.2 Policy-Based Experiments in Mujoco

3.2.1 Setup

Environments We use three environments in Mujoco [2] to benchmark the policy-based methods.

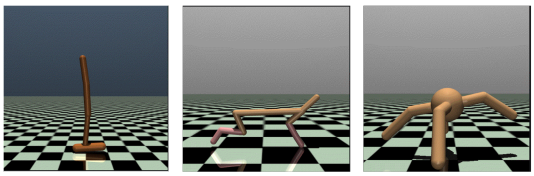

Figure 6: The Mujoco Environments used in Experiments of Policy-Based Methods

- `Hopper-v2` (left, *abbrev.*, `Hopper`) 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.
- `HalfCheetah-v2` (middle, *abbrev.*, `HalfCheetah`) The HalfCheetah is a 2-dimensional robot consisting of 9 links and 8 joints connecting them (including two paws). The goal is to apply a torque on the joints to make the cheetah run forward (right) as fast as possible, with a positive reward allocated based on the distance moved forward and a negative reward allocated for moving backward. The torso and head of the cheetah are fixed, and the torque can only be applied on the other 6 joints over the front and back thighs (connecting to the torso), shins (connecting to the thighs) and feet (connecting to the shins).
- `Ant-v2` (right, *abbrev.*, `Ant`) The ant is a 3D robot consisting of one torso (free rotational body) with four legs attached to it with each leg having two links. The goal is to coordinate the four legs to move in the forward (right) direction by applying torques on the eight hinges connecting the two links of each leg and the torso (nine parts and eight hinges).

Hyper-Parameters Now we introduce the hyper-parameter settings for every experiment environment. Notice that it is a little bit different from the hyper-parameter setting in the original paper [10], due to the limited computational resources. Therefore, the reesults might be inferior than the results in the original paper, but they are still enough for analyses and comparisons.

- For `Hopper` environment, we set the replay buffer capacity $N = 10^6$, the total number of time steps $T = 10^6$ and the number of episode is approximately $M \approx 3000$. The batch size $B = 64$, the learning rate is set to 3×10^{-4} for DDPG and 5×10^{-4} for TD3, and the discount factor $\gamma = 0.99$. The noise scale is set to 0.1. Every step of training is followed by a step of optimization, and the soft update parameter $\tau = 0.005$. For TD3, the policy noise scale is set to 0.2 and clipped into $[-0.5, 0.5]$, and every 2 optimizations is followed by a policy update.
- For `HalfCheetah` environment, the total number of time steps $T = 10^6$ and the number of episode is approximately $M \approx 1000$. The learning rate is set to 3×10^4 for both methods. Other hyper-parameter settings remains the same as `Hopper` environment.
- For `Ant` environment, the total number of time steps $T = 10^6$ and the number of episode is approximately $M \approx 1100$. The learning rate is set to 10^{-4} for both methods. Other hyper-parameter settings remains the same as `Hopper` environment.

Loss We use L2 loss for DDPG algorithm, and use the clipped double-Q loss for TD3 algorithm, which is introduced before.

Optimizer We use AdamW optimizers [11] for both algorithms as stated before.

3.2.2 Results and Analysis

We train 2 different agents (DDPG and TD3 [10]) in 4 different environments mentioned before and then evaluate the performances of the agents.

How well can the agents play? After training agents, we evaluate the ability of the agents by letting them to play several episodes in the Mujoco environment, and summarize the average score, minimum score and maximum score among these episodes. The results are shown in Tab. 6.

Table 6: Evaluation (100 Episode) of Different Methods in Mujoco Environments

Mujoco Environments	**Methods**	**Score**				
		Avg. ↑	Median ↑	Min. ↑	Max. ↑	Std. ↓
`Hopper`	DDPG	**3331.49**	**3561.41**	2094.54	**3681.84**	377.83
	TD3	3023.96	3032.27	**2930.62**	3090.99	**39.55**
`HalfCheetah`	DDPG	5483.22	5485.90	**5308.92**	5754.29	**78.18**
	TD3	**6398.91**	**6428.83**	3248.23	**6701.69**	338.98
`Ant`	DDPG	1093.41	1247.01	-867.71	1793.95	635.51
	TD3	**2511.51**	**2718.23**	**156.74**	**2859.59**	**588.64**

From the results we can make the following analyses.

- For Mujoco `Hopper` environment, the state dimension is 11 and the action dimension is 11. In this simple environment, TD3 performs a little worse than DDPG, according to the average score results in evaluation. However, **TD3 is more stable than DDPG**, according to the standard deviation (as well as minimum) score results in evaluation.
- For Mujoco `HalfCheetah` environment, the state dimension is 17 and the action dimension is 17. In this more complex environment, **TD3 can improve the performances of DDPG to a great extent**, according to the average score results in evaluation.
- For Mujoco `Ant` environment, the state dimension is 111 and the action dimension is 8. In this the most complicated environment, **TD3 outperforms DDPG by a large margin**, according to the average score results in evaluation. Moreover, **TD3 can stablize the ability of the agents compared to DDPG**, according to their standard deviation (as well as minimum and maximum) results in evaluation.

Generally speaking, we conclude that **TD3 performs similar with DDPG, but stabler than DDPG in simple environments** (such as `Hopper`), but **TD3 outperforms DDPG by a large margin in complex environments** (such as `HalfCheetah` and `Ant`). We also observe that **TD3 trains faster than DDPG from the perspective of training time**.

Are the agents difficult to train? We visualize the reward curves during training the agents in simple environment and complex environments respectively, shown in Fig. 7 and Fig. 8, which reflects the training difficulties directly. From the results we can make the following analyses.

- **The training stability of DDPG and TD3 are similar under simple environments** (like *Hopper*), which is reflected by the reward curves in Fig. 7.
- **The training stability of TD3 is better than DDPG under complex environments** (like *HalfCheetah* and `Ant`), which is reflected by the reward curves in Fig. 8. We also observe that **TD3 accelerate the learning process of DDPG in complex environments**.

Generally speaking, we believe that **TD3 can accelerate and stablize the learning process of DDPG, especially in complex environments**.

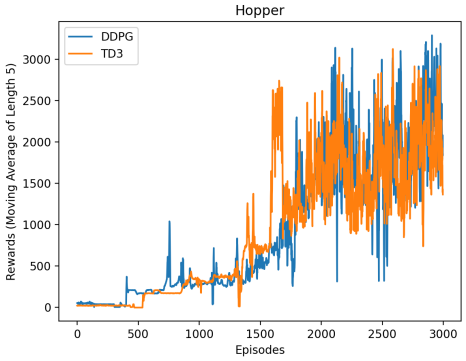

Figure 7: The Training Curves of DDPG and TD3 in Simple Environment `Hopper`

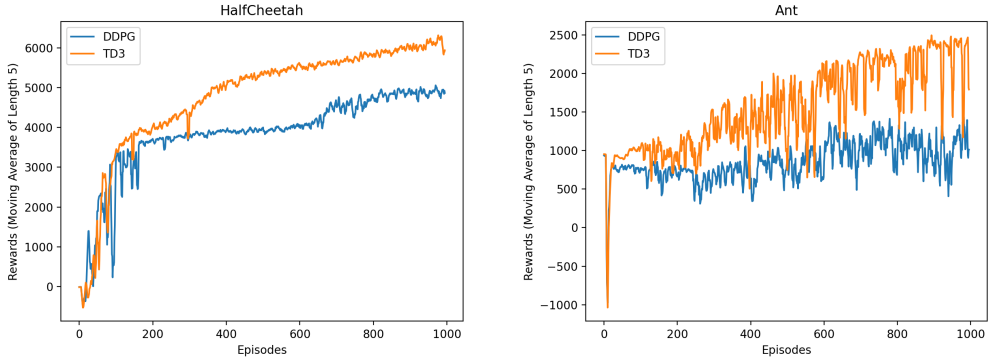

Figure 8: The Training Curves of DDPG and TD3 in Complex Environments `HalfCheetah` and `Ant`

4 Conclusion

In this project, we implement 3 value-based reinforcement learning algorithms (DQN [5], Double-DQN [7, 8] and Dueling-DQN [9]) for discrete-action environment like Atari games [1]. We conduct experiments on 3 Atari games: `Pong`, `Breakout` and `Boxing`. During experiments we find out in most cases Double-DQN and Dueling-DQN can improve the performance of vanilla DQN. Moreover, they can make the agent's ability more stable, and also accelerate and stablize the learning process of

the agents. Ablations shown that learning with L2 loss is more appropriate than L1 loss, and learning with modern optimizers can further improve the performance by an apparent margin.

We also implement 2 policy-based reinforcement learning algorithms (DDPG and TD3 [10]) for continuous-action environment like Mujoco [2]. We conduct experiments on 3 Mujoco environments: `Hopper`, `HalfCheetah` and `Ant`. During experiments we find out that TD3 performs similar with DDPG, but stabler than DDPG in simple environments, but TD3 outperforms DDPG by a large margin in complex environments. We also observe that TD3 trains faster than DDPG from the perspective of training time. Moreover, TD3 can accelerate and stablize the learning process of DDPG, especially in complex environments.

From the implementation process and the experiments, I gain more insight into various reinforcement learning algorithms, including value-based algorithms like DQN, Double-DQN and Dueling-DQn and policy-based algorithms like DDPG and TD3. Overall, I think this project benifits me a lot.

Acknowledgement

I would like to express my deepest gratitude to Prof. Junni Zou and Teacher Wenrui Dai for the detailed introduction and explanation on reinforcement learning algorithms.

I would like to thank the teaching assistants for answering our questions and providing us with supports during the course.

References

[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[2] E. Todorov, T. Erez, and Y. Tassa, "Mujoco: A physics engine for model-based control," in *IROS*, pp. 5026–5033, IEEE, 2012.

[3] R. Bellman, "A markovian decision process," *Journal of Mathematics and Mechanics*, pp. 679–684, 1957.

[4] C. J. C. H. Watkins, "Learning from delayed rewards," 1989.

[5] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[6] M. Roderick, J. MacGlashan, and S. Tellex, "Implementing the deep q-network," *arXiv preprint arXiv:1711.07478*, 2017.

[7] H. Hasselt, "Double q-learning," in *NeurIPS*, 2010.

[8] H. Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," in *AAAI*, 2016.

[9] Z. Wang, T. Schaul, M. Hessel, H. Hasselt, M. Lanctot, and N. Freitas, "Dueling network architectures for deep reinforcement learning," in *ICML*, 2016.

[10] S. Fujimoto, H. Hoof, and D. Meger, "Addressing function approximation error in actor-critic methods," in *ICML*, 2018.

[11] I. Loshchilov and F. Hutter, "Fixing weight decay regularization in adam," 2018.

Appendix

A Environment

For Atari games:

- **Operating System**. Ubuntu 20.04.3 LTS with NVIDIA A100 (CUDA 11.4).
- **Python**. Python 3.8
- **OpenAI Gym**. Gym 0.21 with atari.

For Mujoco environments:

- **Operating System**. MacOS Big Sur with CPU.
- **Python**. Python 3.8
- **OpenAI Gym**. Gym 0.21 with mujoco (mjpro150).

B Codebase

B.1 Preliminary

Our codebase relies on several Python packages, including: `pytorch`, `einops`, `tqdm`, `gym` (version 0.21), `opencv-python`, `easydict`, `numpy`. Please install the packages before running our codes.

B.2 Code

All the source codes in this project are available at

`https://github.com/Galaxies99/CS7309-Project`

Notice that it is a little bit different from our submitted version, since we put the checkpoints in the cloud drive instead of directly putting them in GitHub.

B.3 Checkpoints

For GitHub version, download checkpoints at

`https://pan.baidu.com/s/1CdFhlUiTs741v9yXq--U3A` (Extraction Code: onuk)

Then extract the zipped file into a folder `logs`, and put it in the root directory of our codebase.

For submitted version, checkpoints are in the folder `logs` under the root directory of our codebase.

B.4 Testing

For testing, use the following command:

```
python run.py --env_name [Environment Name]
              --method [Method Name]
              --test [Test Episodes]
```

where `[Environment Name]` is the environment name, `[Method Name]` is the method name and `[Test Episodes]` is the episodes during testing (100 in our experiments).

Currently we support the following combinations of `[Environment Name]` and `[Method Name]`:

- **Atari**. `[Environment Name]` $\in$ { `PongNoFrameskip-v4`, `BreakoutNoFrameskip-v4`, `BoxingNoFrameskip-v4`}, and `[Method Name]` $\in$ { `DQN`, `DoubleDQN`, `DuelingDQN` }.
- **Mujoco**. `[Environment Name]` $\in$ { `Hopper-v2`, `HalfCheetah-v2`, `Ant-v2`}, and `[Method Name]` $\in$ { `DDPG`, `TD3` }.

B.5 Training (Optional)

For training (optional, since we provide trained checkpoints), use the following command:

```
python run.py --env_name [Environment Name]
              --method [Method Name]
              (--cfg [Configuration File])
```

where `[Environment Name]` is the environment name, `[Method Name]` is the method name and `[Configuration File]` is the optional configuration files (we provide the configuration files for training in the `configs` folder, if you want to train by yourselves, please refer to the configuration settings we provided).