# CS3316 Final Project

BY YUSEN ZHENG

Department of Information Security

*Email:* `zys0794@sjtu.edu.cn`

## 1 Introduction

In this task, I implemented three value-based reinforcement learning algorithms: **Deep Q-learning Network (DQN), Double Deep Q-learning Network (DDQN), Dueling Deep Q-learning Network (Dueling-DQN)**, and two policy-based reinforcement learning algorithms: **Deep Deterministic Policy Gradient (DDPG), 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 an introduction to the reinforcement learning algorithms used in this task, along with some details regarding their implementation.

### 2.1 Value-Based Reinforcement Learning

Value-Based Reinforcement Learning is an approach that combines principles from reinforcement learning and value functions to guide an agent in making optimal decisions in dynamic environments. By estimating value functions, which represent the expected cumulative rewards, the agent can learn to select actions that maximize long-term rewards. This approach has been successfully applied in various domains and enables intelligent decision-making in complex scenarios.

#### 2.1.1 Deep Q-learning Network

In [4], Mnih et al. proposed Deep Q-learning Network (DQN) algorithms. DQN is a model-free, off-policy 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 $\epsilon$-greedy exploration strategy, where it selects the action with the highest Q-value with probability $(1-\epsilon)$, and selects a random action with probability $\epsilon$, 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:

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

and the loss function is:

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

where $\theta$ and $\theta^-$ 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**

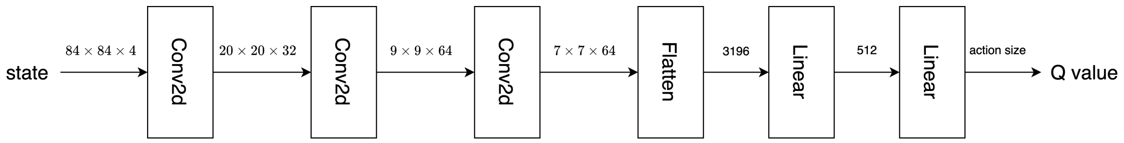


**Figure 1.** DQN

### 2.1.2 Double Deep Q-learning Network

Hasselt et al. propoesd an improved DQN algorithm — Double Deep Q-learning Network (DDQN) in [5]. 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 $\theta$
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 $a^{\max}(s_{j+1}|\,\theta) = \operatorname{argmax}_{a'} Q(s_{j+1}, a'|\,\theta)$
Set $y_j = \begin{cases} r_j & \text{, if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}\left(s_{j+1}, a^{\max}(s_{j+1}|\,\theta)|\,\theta^-\right) & \text{, otherwise} \end{cases}$
Perform a gradient descent step on $(y_j - Q\left(s_j, a_j|\,\theta\right))^2$ w.r.t. the networ parameter $\theta$
Every $C$ steps reset $\hat{Q} = Q$
**End For**
**End For**

### 2.1.3 Dueling Deep Q-learning Network

Wang et al. proposed another kind of improved DQN — Dueling Deep Q-learning Network (Dueling DQN) in [6]. 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.

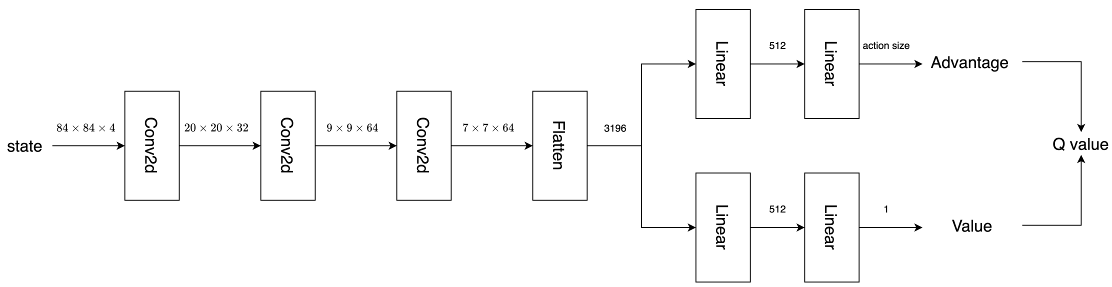


**Figure 2.** Dueling DQN

## 2.2 Policy-Based Reinforcement Learning

Policy-Based Reinforcement Learning is an approach in reinforcement learning that focuses on directly learning the optimal policy without estimating value functions. It uses a parameterized policy, often represented by a neural network, to map states to actions. By interacting with the environment and receiving rewards, the agent adjusts the policy parameters to maximize cumulative reward. Policy-based methods can handle continuous action spaces, exhibit good convergence properties, and learn stochastic policies.

### 2.2.1 Deep Deterministic Policy Gradient

Lillicrap et al. proposed Deep Deterministic Policy Gradient (DDPG) algorithm in [3], as 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 $\theta^\mu$ and $\theta^Q$ 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 $\mathrm{argmax}_a Q(s,a)$ 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 $\theta^{\mu}$, i.e. $\theta^{\mu} \leftarrow \theta^{\mu} + \alpha \cdot pg$. When updating the target network weight $\theta'$, 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 small random noise $\mathcal{N}$, i.e. $a = \mu(s|\,\theta^{\mu}) + \mathcal{N}$.

The formal description is shown in Algorithm 3.

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

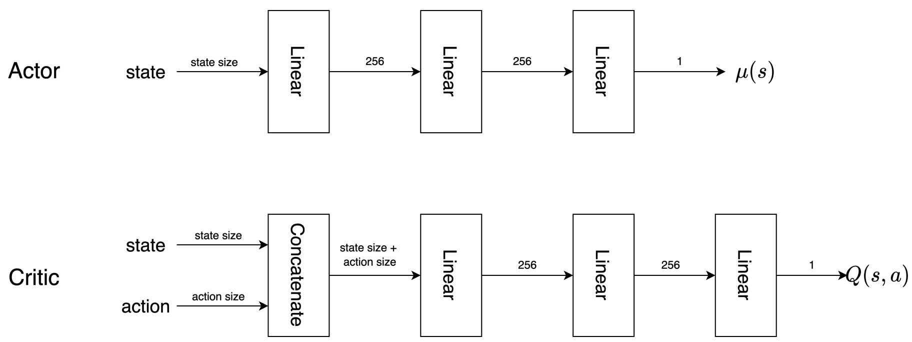


**Figure 3.** DDPG

### 2.2.2 Soft Actor-Critic

Haarnoja et al. introduced Soft Actor-Critic (SAC) in [1][2]. Unlike traditional policy-based methods, SAC maximizes a trade-off between maximizing the expected return and maximizing the **entropy of the policy**. This allows SAC to explore a wider range of actions and learn more robust and diverse policies. By explicitly considering the entropy of the policy, SAC achieves a good balance between exploration and exploitation.

The formal description is shown in Algorithm 4.

**Algorithm 4**

Initialize network parameters $\theta_1, \theta_2, \phi$
Initialize target network weights $\bar{\theta}_1 \leftarrow \theta_1, \bar{\theta}_2 \leftarrow \theta_2$
Initialize an empty replay pool $\mathcal{D} \leftarrow \emptyset$
**For** each iteration **do**
  **For** each environment step **do**
    Sample action $a_t \sim \pi_\phi(a_t | s_t)$ from the policy
    Sample transition $s_{t+1} \sim p(s_{t+1} | s_t, a_t)$ from the environment
    Store the transition in the replay pool $\mathcal{D} \leftarrow \mathcal{D} \cup \{s_t, a_t, r(s_t, a_t), s_{t+1}\}$
  **End For**
  **For** each gradient step **do**
    Update the Q-function parameters $\theta_i \leftarrow \theta_i - \lambda_Q \hat{\nabla}_{\theta_i} J_Q(\theta_i)$ for $i \in \{1, 2\}$
    Update policy weights $\phi \leftarrow \phi - \lambda_\pi \hat{\nabla}_\phi J_\pi(\phi)$
    Adjust temperature $\alpha \leftarrow \alpha - \lambda \hat{\nabla}_\alpha J(\alpha)$
    Update target network weights $\bar{\theta}_i \leftarrow \tau \theta_i + (1-\tau)\bar{\theta}_i$ for $i \in \{1, 2\}$
  **End For**
**End For**

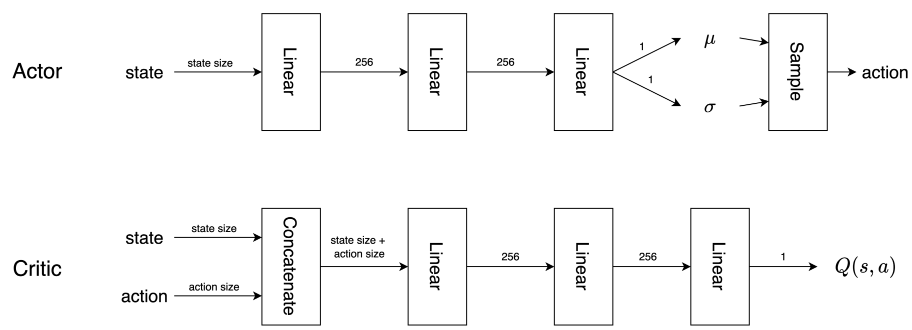


**Figure 4.** SAC

# 3 Experiments

## 3.1 Environment

In this task, we trained the aforementioned algorithms in four reinforcement learning environments provided by OpenAI Gym. These environments include the continuous action space environments `HalfCheetah-v2` and `Ant-v2`, and the discrete action space environments `PongNoFrameskip-v4` and `BoxingNoFrameskip-v4`. These environments are illustrated as follows:

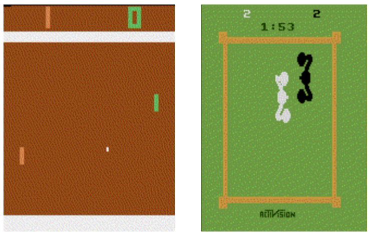

**Figure 5.** Atari

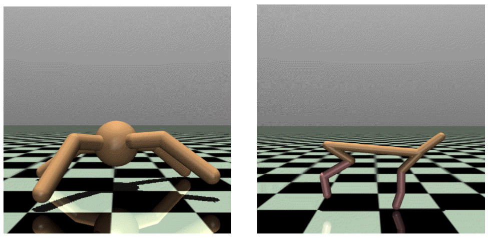

**Figure 6.** MuJoCo

## 3.2 Atari Game

### 3.2.1 Preprocessing

In the Atari Game environment, we employed the preprocessing method used in [4] to efficiently extract features from the game frames, thereby speeding up the training process and reducing memory consumption. The preprocessing process is as follows:

- **Image resize and grayscale conversion:** Convert the color image to grayscale and resize the grayscale image to change the image size from $210 \times 160$ to $84 \times 84$.

- **Frame skip:** To accelerate the training process and reduce the number of computations, a frame skip technique is employed. The agent only takes actions and receives rewards every fourth frame, while the intermediate frames are simply repeated. This effectively reduces the temporal resolution of the environment and speeds up the learning process without significantly sacrificing performance.

- **Frame stack:** To capture temporal information and provide the agent with a sense of motion, a technique called frame stacking is utilized. Four consecutive frames are stacked together to form a single observation. This allows the agent to perceive the dynamics of the environment over time and make informed decisions based on the aggregated information.

- **Reward clip:** In order to facilitate learning and stabilize the training process, the rewards obtained during gameplay are often clipped or bounded within a certain range. In this case, the rewards are clipped to the range of -1 to 1. This prevents extreme reward values from dominating the learning process and helps maintain a more consistent and manageable reward scale.

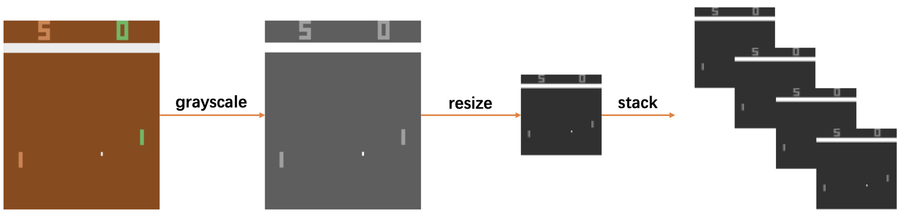


**Figure 7.** Atari Game preprocessing

### 3.2.2 Training

To compare the performance of different DQN algorithms, we utilized the same set of hyperparameters during the training process.

In the `PongNoFrameskip-v4` environment, we trained for 500 episodes. In the `BoxingNoFrameskip-v4` environment, we trained for 3000 episodes. The initial 100 steps were dedicated to the warm-up process, where the agent interacted with the environment without updating the network parameters. Afterward, the network parameters were updated every step using a batch size of 64 sampled from the replay buffer. The target network was updated with the updated parameters every 100 steps. We used a reward discount factor $\gamma$ of 0.9 and a learning rate of 0.0001. The replay buffer size was set to 1,000,000 to store past experiences. During training, we employed the epsilon decay technique. Initially, epsilon was set to 1, and after each step, it decayed by a factor of 0.995 until stabilizing at 0.01. This technique facilitated initial exploration during the learning process.

#### 3.2.3 Results

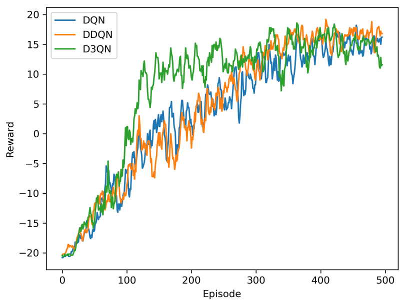


**Figure 8.** `PongNoFrameskip-v4`

**Figure 9.** `BoxingNoFrameskip-v4`

### 3.3 MuJoCo Control

#### 3.3.1 Training

We trained for 1000 episodes in both the `HalfCheetah-v2` and `Ant-v2` environments.

**DDPG.** During the training of the DDPG model, we set the warm-up steps to be 100. After that, we update the network parameters every step with a batch size of 64. The target network is updated with the updated parameters every 100 steps. In the update process, we use the soft update technique with a weight $\tau$ of 0.01. The reward discount factor $\gamma$ is 0.9, the learning rate is 0.0001, and the replay buffer size is 1000000. Gaussian noise with a mean of 0 and variance of 0.005 is added during the exploration process.

**SAC.** During the training process of SAC, most of the hyperparameters used are the same as in DDPG. The difference lies in the fact that in SAC, the network parameters are updated twice per step, the initial value of the temperature $\alpha$ is set to 0.05, and $\tau$ is set to 0.005.

#### 3.3.2 Results

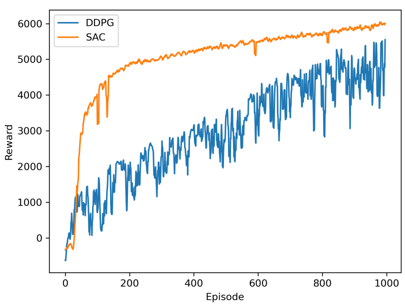


**Figure 10.** `HalfCheetah-v2`

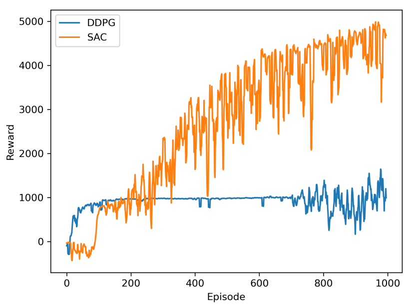


**Figure 11.** `Ant-v2`

# 4 Discussion and Conclusion

## Acknowledgments

At the end of this paper, I would like to express my sincere gratitude to Prof. Zou. Prof. Zou's comprehensive and detailed lectures on reinforcement learning have been highly beneficial to me. I would also like to thank the teaching assistants for the assistance in answering questions and providing necessary support throughout the course, which has greatly contributed to our learning experience. Lastly, I would like to thank you once again for reading this paper, and I hope it has provided you with some inspiration.

## Bibliography

**[1]** Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International conference on machine learning*, pages 1861-1870. PMLR, 2018.

**[2]** Tuomas Haarnoja, Aurick Zhou, Kristian Hartikainen, George Tucker, Sehoon Ha, Jie Tan, Vikash Kumar, Henry Zhu, Abhishek Gupta, Pieter Abbeel et al. Soft actor-critic algorithms and applications. *ArXiv preprint arXiv:1812.05905*, 2018.

**[3]** Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *ArXiv preprint arXiv:1509.02971*, 2015.

**[4]** Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529-533, 2015.

**[5]** Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30. 2016.

**[6]** Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Hasselt, Marc Lanctot, and Nando Freitas. Dueling network architectures for deep reinforcement learning. In *International conference on machine learning*, pages 1995-2003. PMLR, 2016.

# 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
```