1 Introduction

In this final project, two kinds of model-free reinforcement learning methods, value-based RL and policy-based RL are implemented to solve some classic environments in Atari and MuJuCo.

Value-based methods strive to fit action value function or state value function and learn by on-policy or off-policy way. Deep Q-learning (DQN) is a representative value-based method which introduces deep neural network to Q-learning and achieves remarkable performance.

Policy-based methods try to learn a policy directly, by updating policy function with policy gradient. However, original policy gradient method suffers from high variance, then Actor-Critic method is proposed. To solve the drawback of policy gradient methods in sample inefficiency, off-policy methods like DDPG and Soft Actor-Critic are introduced.

For value-based method, I implement DQN on `PongNoFrameSkip-v4` and `BoxingNoFrameSkip-v4`. For policy-based method, I implement PPO and SAC on `Ant-v2`, `Hopper-v2` and `HalfCheetah-v2`.

2 Methods

2.1 Deep Q Network (DQN)

In original Q-learning, we use a lookup table to represent value function. However, for large MDPs like Atari games, there are too many states and actions to store in memory, which makes it hard to learn the value of each state and action with table-based methods. The solution is to estimate value function with function approximation. DQN [1] introduces deep neural network to Q-learning and shows a remarkable performance.

DQN algorithm introduces two key tricks, fixed target network and experience replay. Target network with older network parameters is used when estimating the Q-value for the next state in an experience and the target network is updated on discrete many-step intervals. It provides a stable training target for the network function to fit, and gives it reasonable time to do so, thus controlling the errors in the estimation.

DQN keeps a large replay memory of recent experience, and after each step in environment, the agent adds the experience tuple to the buffer. After some small number of steps, the agent randomly samples a mini-batch from the replay buffer on which to perform the network training. Experience replay helps to accelerate the backup of rewards and decorrelate the samples from the environment.

The detailed DQN algorithm is shown in Algorithm 1.

Algorithm 1: DQN

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
for episode=1 to M **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  for t=1 to T **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = arg\max_a Q(\phi(s_t), a; \theta)$
    Execute action a_t in the emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store experience $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
    sample random minibatch of experiences $(\phi_t, a_t, r_t, \phi_{t+1})$ from D
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the weights θ
    Every C steps reset $\hat{Q} = Q$
  end
end

2.2 Proximal Policy Optimization (PPO)

The original policy gradient methods work by computing an estimator of the policy gradient and learning the policy function with it. The most commonly used gradient estimator is formulated as

$$\hat{g} = \hat{\mathbb{E}}_t \left[\nabla_\theta \log \pi_\theta(a_t|s_t) \hat{A}_t \right]$$

where π_θ is a stochastic policy and $\hat{A}_t$ is an estimator of the advantage function at timestep t. The expectation is calculated by the empirical average over a minibatch of samples, in an algorithm that alternates between sampling and optimization. Normally, the algorithm needs an objective function whose gradient is the policy gradient estimator, as follows,

$$L^{PG}(\theta) = \hat{\mathbb{E}}_t \left[\log \pi_\theta(a_t|s_t) \hat{A}_t \right]$$

Sample inefficient is a problem for policy gradient and Actor-Critic methods. Specifically, π_θ is used to collect data, but when θ is updated, we have to sample training data again. Then importance sampling is used to improve the sample efficiency. It works by using the sample from older policy $\pi_{\theta_{old}}$ to train θ, while the policy parameters θ_{old} is fixed for reusing the sample data. Therefore we get the surrogate objective function

$$L^S(\theta) = \hat{\mathbb{E}}_t \left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \hat{A}_t \right]$$

Meanwhile, we need to restrict that the difference between policy π_θ and old policy $\pi_{\theta_{old}}$ cannot be too large. In Trust Region Policy Optimization (TRPO) [2], a constraint using KL divergence is added,

$$\text{subject to } \hat{\mathbb{E}}_t[\text{KL}[\pi_{\theta_{old}}(\cdot|s_t), \pi_\theta(\cdot|s_t)]] \leq \delta$$

This problem can efficiently be approximately solved using the conjugate gradient algorithm, after making a linear approximation to the surrogate objective and a quadratic approximation to the constraint.

Since it is complicated to solve the optimization problem with constraint in TRPO, an early version of PPO suggests using a penalty alternatively.

$$\underset{\theta}{\text{maxmize}}\, \hat{\mathbb{E}}_t \left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \hat{A}_t - \beta \text{KL}[\pi_{\theta_{old}}(\cdot|s_t), \pi_\theta(\cdot|s_t)] \right]$$

This follows from the fact that a certain surrogate objective forms a lower bound on the performance of the policy π. However, it is hard to choose a single value of the coefficient β, which is the reason that TRPO uses a hard constraint instead of the penalty. Therefore, additional modifications are needed.

PPO method [3] proposes the clipped surrogate objective

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t \left[\min \left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \hat{A}_t, \text{clip}(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}, 1-\epsilon, 1+\epsilon) \hat{A}_t \right) \right]$$

where ϵ is a hyper-parameter normally equals 0.2. The probability ratio is clipped to the interval $[1-\epsilon, 1+\epsilon]$, to make new policy distribution not move too far away from the older one.

With the minimum of the clipped and unclipped function, as shown in Figure 1, when the advantages are positive (left subfigure), it is more likely to take this action, which means the policy probability $\pi_\theta(a_t|s_t)$ will increase, and the minimum will limit how much it can increase. Inversely, when the advantages are negative (right subfigure), it is less likely to take the action, and the minimum restricts how much the policy probability can decrease.

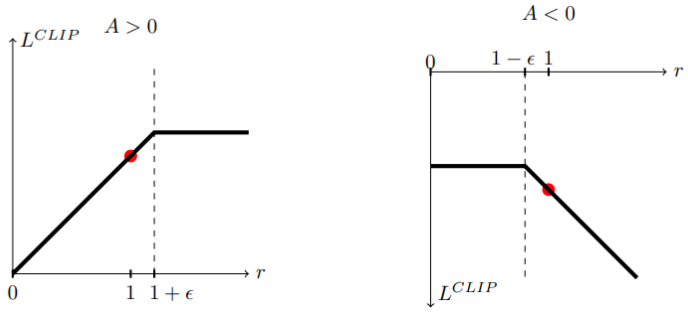

Figure 1: Clipped Surrogate Function for Advantages

The advantage estimator used in PPO is a truncated version of generalized advantage estimation (GAE), formulated as follows,

$$\hat{A}_t = \delta + (\gamma\lambda)\delta_{t+1} + \cdots + (\gamma\lambda)^{T-t+1}\delta_{T-1}$$
$$\text{where } \delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$$

where T is the trajectory length, γ is the discount ratio, and λ is a hyper-parameter.

The complete PPO algorithm is shown in Algorithm 2.

Algorithm 2: PPO

```
for iteration = 1, 2, ··· do
    for actor = 1 to N do
        Run policy π_{θ_old} in environment for T timesteps
        Compute advantage estimates Â_1, ..., Â_T
    end
    Optimize surrogate L wrt θ, with K epochs and minibatch size M ≤ NT
end
```

2.3 Soft Actor-Critic (SAC)

Soft Actor-Critic (SAC) algorithm [4] incorporates three key ingredients: an actor-critic architecture with separate policy and value function networks, an off-policy formulation that enables reuse of previously collected data for efficiency, and entropy maximization to encourage stability and exploration.

The standard reinforcement learning objective is the expected sum of rewards $\sum_t \mathbb{E}_{(s_t,a_t)\sim\rho_\pi}[r(s_t,a_t)]$, while in maximum entropy reinforcement learning, the maximum entropy objective augments it with an entropy term as follows,

$$\pi^* = \arg\max_\pi \sum_t \mathbb{E}_{(s_t,a_t)\sim\rho_\pi}[r(s_t,a_t) + \alpha\mathcal{H}(\pi(\cdot|s_t))]$$

where α is the temperature parameter that determines the relative importance of the entropy term versus the reward, and thus controls the stochasticity of the optimal policy.

The maximum entropy objective has a number of conceptual and practical advantages. First, the policy is motivated to explore more extensively, while giving up on clearly unpromising avenues. Second, the policy can capture multiple modes of near-optimal behavior. In problem settings where multiple actions seem equally attractive, the policy will commit equal probability mass to those actions.

Soft policy iteration is a general algorithm for learning optimal maximum entropy policies that alternates between policy evaluation and policy improvement in the maximum entropy framework. Adding entropy to the rewards, we can get the soft Q-function

$$Q_{soft}(s_t,a_t) = r(a_t,s_t) + \gamma\mathbb{E}_{s_{t+1},a_{t+1}}[Q_{soft}(s_{t+1},a_{t+1}) - \alpha\log(\pi(a_{t+1}|s_{t+1}))]$$

and soft state value V-function

$$V_{soft}(s_t) = \mathbb{E}_{a_t\sim\pi}[Q_{soft}(s_t,a_t) - \alpha\log\pi(a_t|s_t)]$$

And we can update the policy according to

$$\pi_{new} = arg\min_{\pi'\in\Pi} \mathrm{D}_{\mathrm{KL}}\left(\pi'(\cdot|s_t)\middle\|\frac{\exp(\frac{1}{\alpha}Q^{\pi_{old}}(s_t,\cdot))}{Z^{\pi_{old}}(s_t)}\right)$$

where $Z(\theta) = \int \exp(f_\theta(x))dx$ is the partition function.

In SAC algorithm, function approximators are used for both soft Q-function $Q_\theta(s_t,a_t)$ and policy $\pi_\phi(a_t|s_t)$, where θ and ϕ are the parameters of the networks, and both networks are optimized with stochastic gradient descent.

The soft Q-function parameters can be trained to minimize the soft Bellman residual

$$J_Q(\theta) = \mathbb{E}_{(s_t,a_t)\sim D}\left[\frac{1}{2}\left(Q_\theta(s_t,a_t) - (r(s_t,a_t) + \gamma \mathbb{E}_{s_{t+1}\sim p}[V_{\bar{\theta}}(s_{t+1})])\right)^2\right]$$

and it can be optimized with stochastic gradients

$$\hat{\nabla} J_Q(\theta) = \nabla_\theta Q_\theta(a_t,s_t)(Q_\theta(s_t,a_t) - (r(s_t,a_t) + \gamma(Q_{\bar{\theta}}(s_{t+1},a_{t+1}) - \alpha \log(\pi_\phi(a_{t+1}|s_{t+1})))))$$

where $Q_{\bar{\theta}}$ is the target soft Q-function.

And the policy parameters can be learned by minimizing the loss function

$$J_\pi(\phi) = \mathbb{E}_{s_t\sim\mathcal{D}}[\mathbb{E}_{a_t\sim\pi_\phi}[\alpha \log(\pi_\phi(a_t|s_t)) - Q_\theta(s_t,a_t)]]$$

In order to solve the problem that action is sampled from the distribution, which is not differentiable, the reparameterization trick is applied, thus getting action using a neural network transformation

$$a_t = f_\phi(\epsilon_t; s_t)$$

where ϵ_t is an input noise vector, sampled from some fixed distribution, such as a spherical Gaussian. So the objective function can be rewritten as

$$J_\pi(\phi) = \mathbb{E}_{s_t\sim\mathcal{D},\epsilon_t\sim\mathcal{N}}[\alpha \log \pi_\phi(f_\phi(\epsilon_t;s_t)|s_t) - Q_\theta(s_t, f_\phi(\epsilon_t;s_t))]$$

There is another problem that choosing the optimal temperature α is non-trivial and the temperature needs to be tuned for each task manually. Therefore, an automating temperature tuning method is proposed by formulating a constrained optimization problem where the average entropy of the policy is constrained, while the entropy at different states can vary. Specifically, it aims to find a stochastic policy with maximal expected return that satisfies a minimum expected entropy constraint.

$$\max_{\pi_{0:T}} \mathbb{E}_{\rho_\pi}\left[\sum_{t=0}^{T} r(s_t,a_t)\right] \text{ s.t. } \mathbb{E}_{(s_t,a_t\sim\rho_\pi)}[-\log(\pi_t(a_t|s_t))] \geq \mathcal{H}\ \forall t$$

where $\mathcal{H}$ is a desired minimum expected entropy. And we can solve the optimal α_t^* as follows,

$$\alpha_t^* = \arg\min_{\alpha_t} \mathbb{E}_{a_t\sim\pi_t^*}[-\alpha_t \log \pi_t^*(a_t|s_t;\alpha_t) - \alpha_t \bar{\mathcal{H}}]$$

But in practice, we compute gradients for α with the following objective:

$$J(\alpha) = \mathbb{E}_{a_t\sim\pi_t}[-\alpha \log \pi_t(a_t|s_t) - \alpha\bar{\mathcal{H}}]$$

The detailed algorithm of SAC is described in Algorithm 3.

Algorithm 3: SAC

Initial parameters θ_1, θ_2, ϕ
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 from the policy $a_t \sim \pi_\phi(a_t|s_t)$
        Sample transition from the environment $s_{t+1} \sim p(s_{t+1}|s_t, a_t)$
        Store the transition $(s_t, a_t, r(s_t, a_t), s_{t+1})$ in the replay pool $\mathcal{D}$
    end
    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 $\phi \leftarrow \lambda_\pi \hat{\nabla}_\phi J_\pi(\phi)$
        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
end

3 Experiments and Discussion

3.1 Atari

3.1.1 Environment Settings

Atari environment contains some classic games for the algorithm to play. The following environment and training settings are mainly from the original DQN implementation [1] and improved implementation [5], in order to acquire a better performance.

As the observation of Atari environment is the game machine's screen, it has an observation space with shape $210 \times 160 \times 3$. The observations require preprocessing to save memory and computation resource. First, the input is resized to 84×84 and converted from RGB to grayscale. Also, the state observations are stored into the replay buffer as uint8 type to save memory and converted back to float when sampled from the buffer.

In most Atari games, there is a notation of "lives" for the player, which means the number of times the player can fail in a game trial. A trick to increase performance is to count the loss of a life as a terminal state in the MDP, then no more reward can be got. This helps the agent learn that losing a life should be avoided at all costs, just like human players do. Moreover, the maximum number of steps in an episode is limited to 10000, and other tricks like frameskip and reward clipping are also used to stabilize training.

3.1.2 Implementation Details

When every episode starts, the agent performs a random number of "No-op" low-level Atari actions between 0 and 30 in order to offset which frames the agent sees, since the agent only sees every 4 Atari frames. Similarly, the m frames history used as the input to the Q network is the last m frames that the agent sees, not the last m Atari frames.

The network is updated every 4 environment steps which not only can greatly speed up the training process, and also cause the experience memory to more closely resemble the state distribution of the current policy with 4 new frames adding to the replay buffer and may prevent the network from over-fitting. The agent will start training after there are over 10000 experiences in the replay buffer.

The Q-network used for Atari is a convolutional neural network (CNN) with three convolution layers and two linear layers, as shown in Table 1. The observation state is normalized from $[0, 255]$ to $[0, 1]$ before inputting to the network. Another proposed trick is clipping the gradient of the error term to be between -1.0 and 1.0, which further improve the stability by not allowing any single mini-batch update to change the parameters drastically.

Table 1: CNN Network Details in DQN

Layer	Input_dim	Output_dim	Kernel Size	Stride	Activation
Conv2d	4	32	8×8	4	relu
Conv2d	32	64	4×4	2	relu
Conv2d	64	64	3×3	1	relu
Linear	448	512	/	/	relu
Linear	512	action size	/	/	none

The hyper-parameters used in DQN algorithm are listed in Table 2. Notice that in ϵ-greedy action selection, ϵ is updated by the equation

$$\epsilon \leftarrow \epsilon - \frac{\epsilon_{start} - \epsilon_{end}}{\epsilon \text{ decay rate}}$$

When $\epsilon = \epsilon_{start} = 1$, it is totally random selection, and with ϵ decreasing, the randomness goes down and the action selection is more deterministic.

Table 2: Hyper-parameters in DQN

Hyper-parameter	Value
batch size	32
γ	0.99
ϵ_{start}	1
ϵ_{end}	0.01
ϵ decay rate	1,000,000
policy update stride	4
target update stride C	1000
learning rate	0.0000625
replay buffer size	200,000

3.1.3 Results

I trained DQN on two Atari environments provided in `gym`, `PongNoFrameSkip-v4` (Pong) and `BoxingNoFrameSkip-v4` (Boxing). Pong is trained for 1500 episodes,

while Boxing is trained for 3000 episodes, and the results are shown in Figure 2 and 3.

For Pong, we can see that the training is quite fast, as it converges after just 800 episodes. And surprisingly, there is a huge rising of reward during 600 to 800 episode, which means the algorithm quickly learns a good policy to play this game. Considering Boxing, the training process is much slower and it converges after 2500 episodes. This is reasonable because the Boxing game is more difficult to learn a proper policy, and the information got from frames in Pong is clearer.

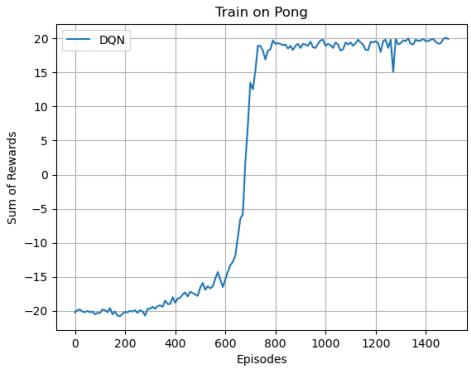

Figure 2: Training of Pong

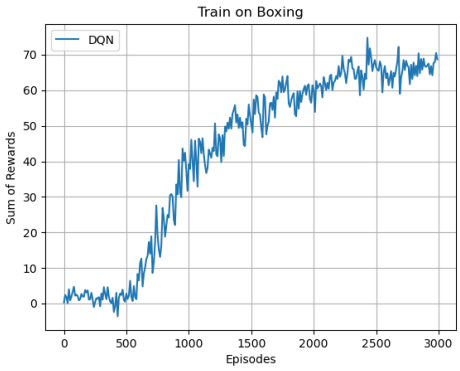

Figure 3: Training of Boxing

For evaluation of the trained models, with no reward clipping, the average score in Pong is 20.76($\pm$0.43), and in Boxing is 75.67($\pm$12.27). I found that the average score achieved in original DQN implementation [1] and improved implementation [5] are 18.9($\pm$1.3) and 19.7($\pm$1.1) respectively. It turns out DQN algorithm performs well in this environment.

I have tried to use a larger replay buffer for training, it shows that the performance is better in early stage. But it soon runs out of RAM on my computer, so I cannot acquire a complete result. I also tried to train DQN on another Atari environment `BreakoutNoFrameSkip-v4` and use Dueling DQN network alternatively in these environments. However, I find that the training process becomes much more slower than the previous experiments. Due to limitation of time and computation resource, I do not finish these extra experiments.

3.2 MuJoCo

MuJoCo stands for Multi-Joint dynamics with Contact, which is a physics engine for detailed, efficient rigid body simulations with contact, and it can be embedded into `gym` with `mujoco-py`. Compared with Atari games, there is not any specified environment settings in MuJoCo.

3.2.1 Implementation Details

The PPO algorithm is implemented in an Actor-Critic style, the policy function is estimated by the Actor, while the value function is estimated by the critic. Both actor and critic use a fully connected network with two hidden layers of 64 units and tanh activation function, but they do not share parameters like A2C or A3C. More hyper-parameters used in PPO are shown in Table 3.

Table 3: Hyper-parameters in PPO

Hyper-parameter	Value
batch size	64
γ	0.99
λ	0.95
clip ϵ	0.2
update stride	2048
actor learning rate	0.0003
critic learning rate	0.0003

The SAC algorithm is also implemented in an Actor-Critic style. The actor is modelled as a Gaussian distribution with mean and covariance given by a neural network with two hidden layers of 256 neurons per layer. SAC uses soft Q-function as the critic, and it makes use of two soft Q-functions to mitigate positive bias in the policy improvement step that is known to degrade performance of value based methods. The two critics each maintains a neural network with two 256-unit hidden layers, and they are trained independently. Then the minimum of the soft Q-functions is used in calculating gradients. Moreover, the actor will select action randomly for the first 10000 steps in order to sample more general experiences. The overall hyper-parameters of SAC are listed in Table 4.

Table 4: Hyper-parameters in SAC

Hyper-parameter	Value
batch size	256
γ	0.99
replay buffer size	1,000,000
entropy target	-dim($\mathcal{A}$)
target smoothing coefficient τ	0.005
target update interval	1
gradient steps	1
learning rate	0.0003

3.2.2 Results

I trained PPO and SAC on three MuJoCo environments provided in `gym`, `Ant-v2`, `Hopper-v2` and `HalfCheetah-v2`. The training step is 1,000,000 for both algorithms and all environments, the temperature α is fixed to 0.2 in SAC, and the results are shown in Figure 4, 5 and 6. Since the number of steps in a single episode varies among environments, the number of episodes shown in the results are different, though the total steps are the same.

Surprisingly, we can see that SAC overperforms PPO in all environments with rewards that are much times higher, especially in HalfCheetah, where SAC transcends PPO over 10 times. We can also notice that the learning speed and stability are different in these environments. For Hopper, there is a steep rise in SAC during 1000 to 1500 episodes, and then the algorithm converges. In HalfCheetah, the

reward first rises rapidly, then slows down and keeps going up, while the reward varies more smoothly in both methods. And in Ant, the score rises with a constant speed in SAC, but there are more fluctuations than the other two. For Ant and HalfCheetah environments, it seems that SAC does not completely converges within the step limitation, and maybe more training steps are required.

Figure 4: Training of Ant

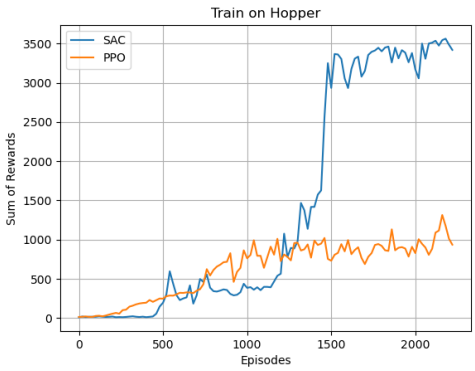

Figure 5: Training of Hopper

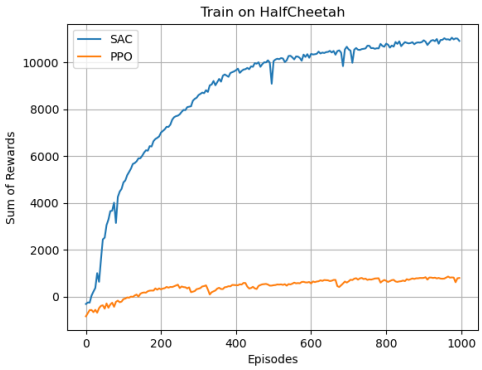

Figure 6: Training of HalfCheetah

For evaluation of the trained models, I run the methods for 100 episodes, and the average scores are shown in Table 5. The evaluation results are consistent with the training results, and we can find that the variance in SAC score is relatively smaller than PPO, which also proves that SAC is better than PPO in MuJoCo environments.

Table 5: Evaluation on MuJoCo

Environments	PPO Score	SAC Score
Ant-v2	542.84(±263.50)	5743.00(±670.14)
Hopper-v2	991.54(±305.39)	3595.18(±2.68)
HalfCheetah-v2	867.72(±218.35)	11515.29(±116.52)

Then I explore on the automatic temperature tuning trick in SAC. It is trained on HalfCheetah and the result is shown in Figure 7, while the variation of temperature is shown in Figure 8. We can see that there is a gap of performance

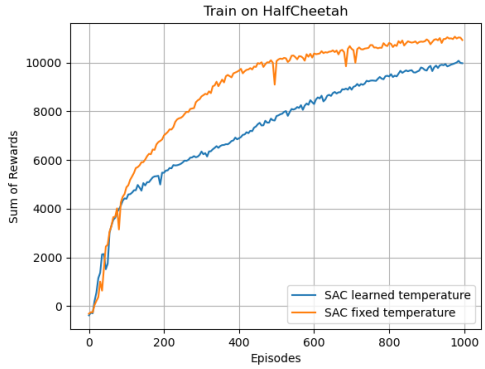

Figure 7: Training with Different Temperature Settings

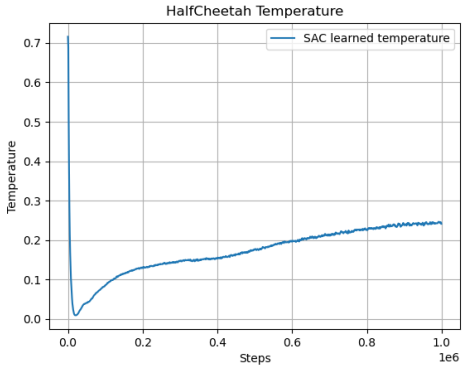

Figure 8: Temperature Variation in Training

between learned temperature and fixed temperature, and the learned temperature is approaching an optimal value near the fixed temperature 0.2.

In this case, the value of the fixed temperature is got from the previous work by researchers, however, when we are handling a brand new environment with SAC, the automatic temperature tuning method can save us a lot of time from looking for an optimal value manually. Therefore, this trick is very useful in SAC algorithm.

4 Conclusion

In this project, one value-based RL algorithm, DQN and two policy-based RL algorithms, PPO and SAC are implemented on several Atari and MuJoCo environments. We conclude that DQN can perform well in Atari games, but it suffers from sample inefficiency, which means it requires plenty of memory space and it takes a long time to train to get a good performance. PPO, as the default method in OpenAI, has a clear theory and uses importance sampling to solve the sample inefficient problem. However, PPO does not performs well in MuJoCo environments, though its training is relatively faster. SAC is an off-policy method for efficiently sampling and it introduces maximum entropy objective, which makes it harder to understand but shows a marvellous performance in MuJoCo environments. In general, I think that the effect of an algorithm may be greatly influenced by different environments and parameters, which adds difficulty to algorithm selection and implementation in RL problems.

References

[1] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[2] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International conference on machine learning*, pages 1889–1897. PMLR, 2015.

[3] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[4] 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.

[5] Melrose Roderick, James MacGlashan, and Stefanie Tellex. Implementing the deep q-network. *arXiv preprint arXiv:1711.07478*, 2017.

Code References

[1] https://github.com/jmichaux/dqn-pytorch

[2] https://github.com/reinforcement-learning-kr/pg_travel

[3] https://github.com/pranz24/pytorch-soft-actor-critic

[4] https://github.com/openai/baselines