Reinforcement Learning 2021 Practical Assignment 3: Actor-Critic implementations in *LunarLander*

Xinrui Shan(s2395738) Pengyu Liu(s2604531) Zhe Deng(s2696266)

1 Introduction

In assignment A2, we found that besides MCPG, there are many other policy gradient algorithms which have wide applications in reinforcement learning. So we decided to focus on the policy gradient algorithms in this assignment. In practice, we implemented three variants of Actor-Critic methods: Asynchronous Advantage Actor-Critic (A3C), Proximal Policy Optimization Algorithm (PPO) and Deep Deterministic Policy Gradient (DDPG). All of them are based on Actor-Critic method and are off-policy policy gradient algorithms. Both A3C and PPO can be used in both discrete and continuous action space, and DDPG is only available for continuous action space.

In order to implement and compare these three algorithms and also play with an interesting game, we used the LunarLander[1] environment from gym. It is a Box2D simulator that human players or AI navigate a lander to its landing pad as demonstrated in Figure 1. The landing pad is fixed, but the terrain outside of the landing pad changes between games. The observation space contains eight dimensions (Box(8)): horizontal coordinate, vertical coordinate, horizontal speed, vertical speed, angle, angular speed, if the first leg has contact the ground, if the second leg has contact the ground.

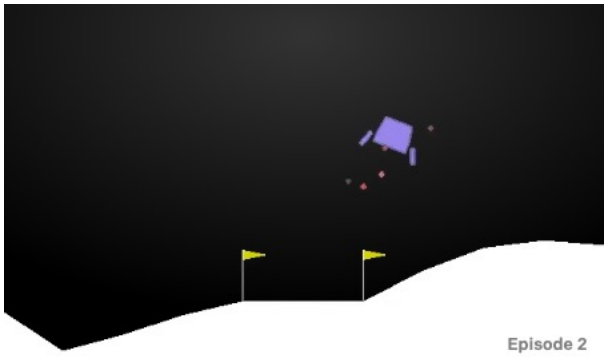

Figure 1: Lunar Lander environment

Depending the action space, LunarLander has two versions: discrete and continuous. In discrete version, four discrete actions are available: do nothing (0), fire left orientation engine (1), fire main engine (2) and fire right orientation engine (3). In continuous version, action is two real values vector from -1 to +1, where the first float number controls main engine (-1 - 0 is off and 0 - 1 fire from 50% to 100% power) and the second value controls left and right orientation engines (-1 - -0.5 fire left engine, -0.5 - 0.5 is off, and 0.5 - 1 fire right engine).

The reward for the discrete and continuous Lunar lander environment is quite complicated. If successfully landed to the landing pad with zero speed, there are 100 140 points to earn. If the lander falls or comes to rest, the episode ends, with the lander earning an extra -100 or +100 points. And each leg with ground contact is +10 points, firing the main engine costs -0.3 points each step, firing the side engine costs -0.03 points each step, solved is 200 points. And Landing outside the landing pad is possible. Due to the fact that fuel is unlimited, an agent can learn to fly and then land many attempts in one episode.

2 Actor-Critic

Actor-Critic methods[2] can be literally split into two parts to explain: actor and critic. The "actor" part develops from Policy Gradient methods, which updates the policy parameters θ for $\pi_\theta(a|s)$, in the direction suggested by the "critic". The "critic" part estimates the value function. This could be the value function parameters ω and the *action-value* $Q_\omega(a|s)$ or *state-value* $V_\omega(s)$ depending on the algorithm.

Value function methods work well in discrete environments when there is a finite set of actions. However, when working in continuous or stochastic environments, policy gradient methods are needed to compute the optimal policy directly, without first going through the intermediate step of calculating values. Direct policy evaluation can be more efficient in continuous or stochastic environments and when there are many different reward values. Converging to the value may then take a long time.[3]

Actor-Critic methods are are quite popular, and have been studied widely. Several actor-critic algorithms are well-known, such as Asynchronous Advantage Actor-Critic(A3C), Deep Deterministic Policy Gradient(DDPG) and Proximal Policy Optimization(PPO).

2.1 Policy Gradient

Policy Gradient methods are common used in model free reinforcement algorithms, and target at modeling and optimizing the policy directly.

Actor has its own *policy* π for a specific task. *Policy* π is usually represented by a neural network with a parameter θ. Starting from a specific state until the end of the task, it is called a complete episode. At each step, we can get a *reward* r, and the *cumulative reward* for a

complete task is called R. In this way, for an episode with T moments, the Actor continuously interacts with the environment to form the following *trajectory* τ. The gradient of the *reward function* $J(\theta)$ is defined below.

$$\nabla_\theta J(\theta) = \mathbb{E}_r[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) R_t] \tag{1}$$

Using gradient ascent, we can move θ toward the direction suggested by the gradient to find the best θ for π_θ that produces the highest return. One example of this method is REINFORCE family of algorithms.

Standard REINFORCE updates the policy parameters according to Equation(1). Updating the policy parameter by Monte Carlo updates can introduce inherent high variability in log probabilities and cumulative reward values, because each trajectories during training can deviate from each other at great degrees. Therefore, the high variability of logarithmic probability and cumulative reward value will produce noisy gradients, and lead to learning instability or policy distribution biased in a non-optimal direction.

Besides high variance of gradients, another problem with policy gradients occurs trajectories have a cumulative reward of 0.

Overall, these issues contribute to the instability and slow convergence of the policy gradient methods.

It is possible to reduce the variance of this estimate while keeping it unbiased by subtracting a learned function of the state $b(s_t)$, known as a baseline. Thus, the resulting gradient is $\nabla_\theta \log \pi_\theta(a_t|s_t)(R_t - b(s_t))$.[4]

2.2 Advantage Actor-Critic (A2C)

Based on the knowledge of Q-learning during the practical assignment 2 and previous section of Policy Gradient, we know that the expectation of Q-value is $Q(s_t, a_t) = \mathbb{E}_{r_{t+1}, s_{t+1}, \ldots, r_t, s_t}[R_t]$. As we know, the Q-value can be learned by parameterizing the Q function with a neural network (denoted by the parameter ω).

In this way, Equation(1) can be decomposed as follows:

$$\begin{aligned}\nabla_\theta J(\theta) &= \mathbb{E}_{s_0,a_0,\ldots,s_t,a_t}[\sum_{t=0}^{T-1}\nabla_\theta \log\pi_\theta(a_t|s_t)]\mathbb{E}_{r_{t+1},s_{t+1},\ldots,r_T,s_T}[R_t] \\ &= \mathbb{E}_{s_0,a_0,\ldots,s_t,a_t}[\sum_{t=0}^{T-1}\nabla_\theta \log\pi_\theta(a_t|s_t)]Q_\omega(s_t,a_t) \\ &= \mathbb{E}_\tau[\sum_{t=0}^{T-1}\nabla_\theta \log\pi_\theta(a_t|s_t)Q_\omega(s_t,a_t)]\end{aligned} \tag{2}$$

As is mentioned above, introducing a baseline function can reduce the variance of the estimation. It is commonly used to take the V function as the baseline function: $b(s_t) \approx V^\pi(s_t)$, that means the Q value term is subtracted with the V value. This value is defined as the advantage value: $A(a_t, s_t) = Q(a_t, s_t) - V(s_t)$. Intuitively, it means how much better it is to take a specific action compared to the average, general action at the given state. This approach can be viewed as an actor-critic architecture where the policy π is the actor and the baseline $b(s)$ is the critic.[5]

Instead of inefficiently constructing two neural networks for both the Q value and the V value, the relationship between them from the Bellman optimality equation is adopted: $A(a_t, s_t) = r_{t+1} + \gamma V_\omega(s_{t+1}) - V_\omega(s_t)$.

$$\nabla_\theta J(\theta) \sim \sum_{t=0}^{T-1}\nabla_\theta \log\pi_\theta(a_t|s_t)(r_{t+1} + \gamma V_\omega(s_{t+1}) - V_\omega(s_t)) = \sum_{t=0}^{T-1}\nabla_\theta \log\pi_\theta(a_t|s_t)A(s_t,a_t) \tag{3}$$

As is shown above, there is only one neural network needed for the V function (parameterized by ω). The Equation(1)&(2) can be updated as Equation(3). This is the main idea of the Advantage Actor Critic (A2C).

Advantage Actor-Critic (A2C) is a synchronous and deterministic version of A3C with "asynchronous" removed, which means a single-worker variant of the A3C. A2C has been shown to be able to utilize GPUs more efficiently and work better with large batch sizes while achieving same or better performance than A3C.[6]

2.3 Asynchronous Advantage Actor-Critic (A3C)

Asynchronous Advantage Actor-Critic (A3C) is a classic policy gradient method with a special focus on parallel training.

In A3C, the critics learn the value function while multiple actors are trained in parallel environments independently and get synced with global parameters from time to time. Hence, A3C is designed to work well for parallel training. One key benefit of having asynchronous actors is effective and efficient exploration of the state space.

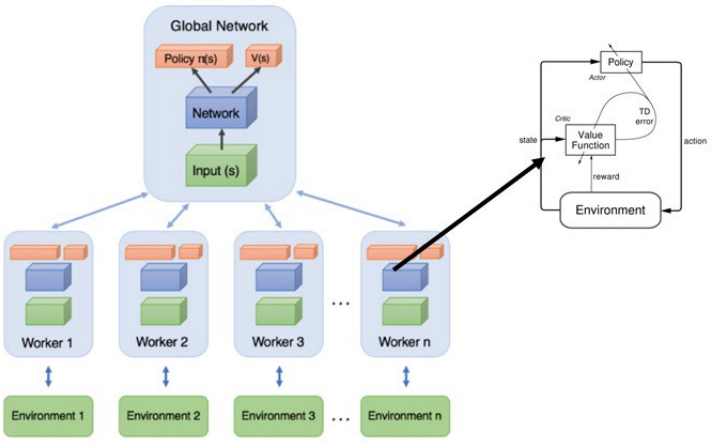

Figure 2: High Level Architecture of A3C[7]

In A3C, each agent communicates with global parameters independently, so it is possible sometimes the thread-specific agents would be playing with policies of different versions and therefore the aggregated update would not be optimal. Compared with A3C, a coordinator in A2C waits for all the parallel actors to finish their work before updating the global parameters and then in the next iteration parallel actors starts from the same policy to resolve the inconsistency,. The synchronized gradient update keeps the training more cohesive and potentially to make convergence faster.

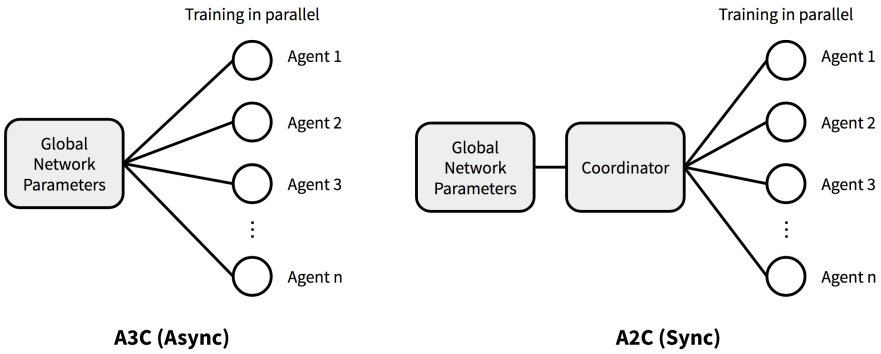

Figure 3: The comparison of the structure between A3C and A2C

Multi-threaded asynchronous variants of one-step Sarsa, one-step Q-learning, n-step Q-learning, and advantage actor-critic are presented to find RL algorithms that can train deep neural network policies reliably and without requiring large resource. Although the underlying RL method is very different, with actor-critic being an on-policy policy search method and Q-learning being an off-policy value-based method, these two main ideas are used to ensure these four algorithms practical.

Our implementation generally follows the algorithm outline of the Asynchronous one-step Q-learning for each actor-learner thread proposed in the paper[8]:

1. Initialize *thread step counter* $t \leftarrow 0$, *target network weights* $\theta^- \leftarrow \theta$, *network gradients* $d\theta \leftarrow 0$ and Get initial state s
   (Assume global shared *global parameters* θ, *similar thread-specific parameters* θ^-, and *counter* $T = 0$)

2. While $T \leq T_{max}$:

   2.1. Take *action* a with ϵ-greedy policy based on $Q(s, a; \theta)$

   2.2. Receive *new state* s' and *reward* r

   2.3. Set $y = r + \gamma max_{a'} Q(s', a'; \theta^-)$ for non-terminal s'; For terminal s', $y = r$

   2.4. Accumulate gradients w.r.t. θ: $d\theta \leftarrow d\theta + \frac{\partial (y - Q(s,a;\theta))^2}{\partial \theta}$

   2.5. $s = s'$, $T \leftarrow T + 1$ and $t \leftarrow t + 1$

   2.6. if (T mod $I_{target} == 0$): Update the target network $\theta^- \leftarrow \theta$

   2.7. if (t mod $I_{AsyncUpdate} == 0$ or s is terminal): Perform asynchronous update of θ using $d\theta$ and Clear gradients $d\theta \leftarrow 0$

2.3.1 Implementation

First, A net class is built that can be used for both actor and critic. For A3C, there are two types of networks: global network and local networks for every work. For global network, shape and parameters are defined. While for local ones, loss, gradient and synchronization(pull and push) are implemented. Building nets generally follows the way of one tutorial to build a basic RNN by using "BasicRNNCell". Actor net is used to output action and variance, and Critic net can output the state value to calculate TD. The class also includes the functions to apply local gradients to the global net and select action w.r.t the probability of actions.

Next, the important part of parallel working is implemented in the Worker class. In this class, the processes of initialization, buffer, state transition are included as well as reward accumulation.
Finally, we run the main function. Multiprocessing and threading are used. Every core of the CPU is assigned with a thread. Each thread works locally managed by COORD and updates the global network according to the frequency we set.

2.3.2 Experiment

Several experiments are carried out with the implemented A3C algorithm. The results of tuning the parameters and optimizing the code are illustrated below:

- **learning rate** α:
  Three values of learning rate are tested as the Figure 4. It is shown that a relatively smaller learning rate 0.0005(blue line) performs better that 0.001(orange line) and 0.005(green line). This is probably due to the number of the worker 8 and the global network is updated every 5 episode, so we need a long enough episode to guarantee the learning process to reach convergence.

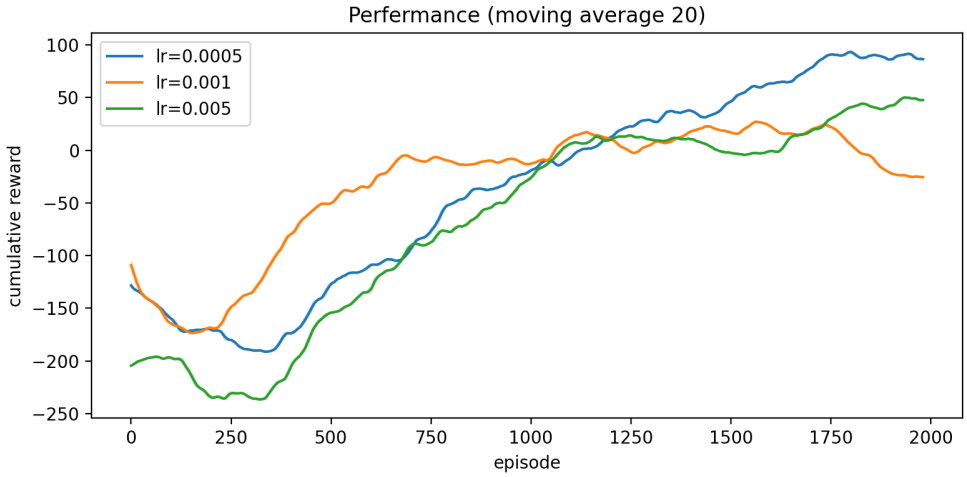

Figure 4: An experiment for three learning rate choices. Solid line is the moving average of recent 20 episodes, filled area is one time sigma uncertainty area calculated by recent 20 episodes to present the fluctuation of cumulative rewards (can represent the distribution of initial cumulative reward curve). Blue: $lr = 0.0005$; orange: $lr = 0.001$; green: $lr = 0.005$.

- **discount factor** γ:
  Due to the number of the workers and the the frequency that the local nets synchronize with the global network, We decide to test the value 0.99 and 0.999 after the experimenting with the value 0.9. The results are in the Figure 5. For episode greater than 1000, both the value 0.999(blue line) and 0.99(orange line) perform obviously better than 0.9(green line). Blue and orange lines are close, while orange line leave more space for improvement as the figure shows. Both 0.99 and 0.999 are suitable for the algorithm.

- **parameters associated with *Workers***:
  The parameters associated with *Workers* are the *number of workers* and the *frequency to update global network*. These parameters will not influence the result, but will determine the time span, speed and smoothness of carrying out the experiments. As the device in daily life is 8-core or 4-core CPU, it is reasonable to set the number of workers to be 4 or 8 to carry out experiments smoothly and fast enough. When the number of workers is 8, the frequency is not ought to be the same, because it would take longer episodes to converge and comparison with DDPG and PPO will be not that scientifically. In conclusion, it is better to fix the frequency to 4, and call the statement that detects the number of cpu cores to allocate a thread to each core. Examples are shown in Figure 6.

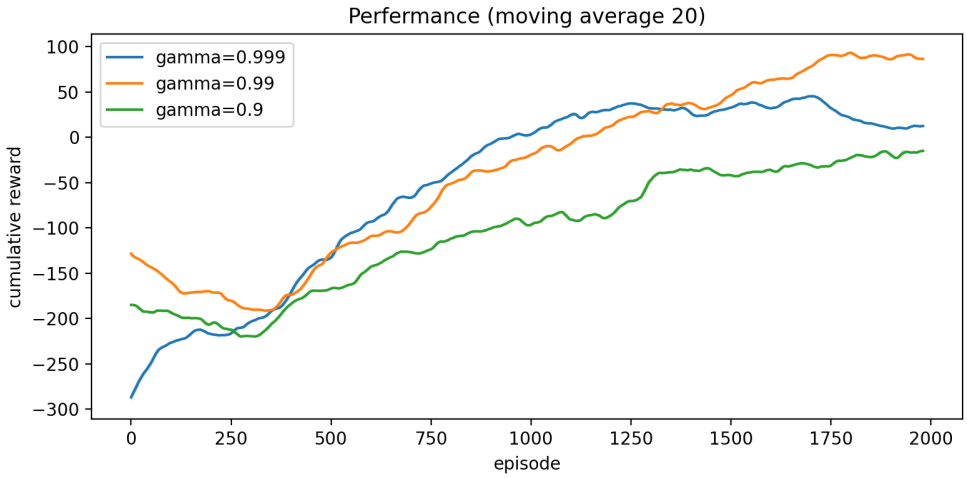

Figure 5: An experiment for three discount factor γ choices. Solid line is the moving average of recent 20 episodes, filled area is one time sigma uncertainty area calculated by recent 20 episodes to present the fluctuation of cumulative rewards (can represent the distribution of initial cumulative reward curve). Blue: $\gamma = 0.999$; orange: $\gamma = 0.99$; green: $\gamma = 0.9$.

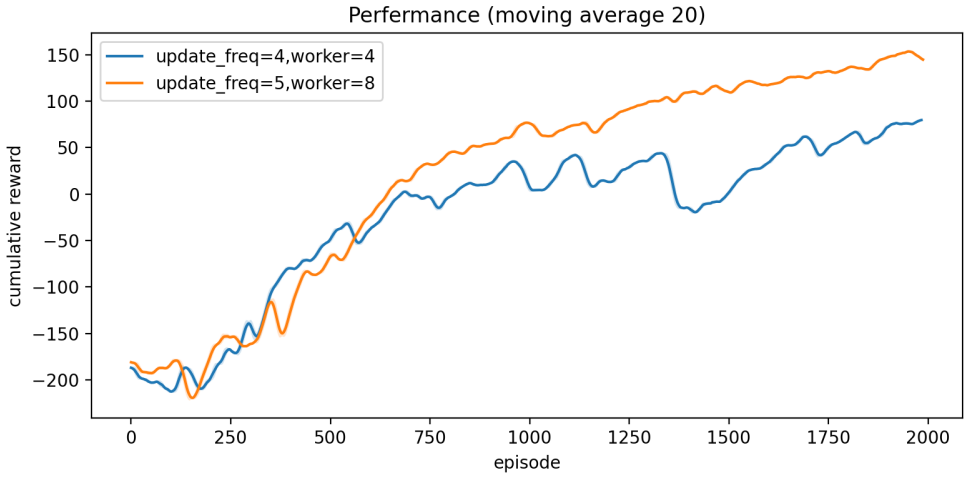

Figure 6: An experiment for number of workers and frequency to update global network.

3 Proximal Policy Optimization Algorithm (PPO)

3.1 Implementation

Proximal optimization algorithm (PPO) is also a policy gradient algorithm based on actor-critic method. Regular policy gradient usually has a fixed update step which can make the policy network deviate far away from the good one if a single roll-out trajectory is bad because of some randomness. This can make policy algorithm unstable and converge slowly. But instead of changing the update step, PPO adds clip operation to the optimization function and reduces the code complexity. It prevents excessively large policy update. The objective PPO needs to maximize is given by:

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t \left[\min \left(r_t(\theta)\hat{A}_t, \text{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right)\hat{A}_t \right) \right] \tag{4}$$

$r_t(\theta)$ is the probability ratio of transition (at,st) of new and old policy:$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$. $\hat{A}_t$ is the advantage function. Figure 7 shows how the surrogate function is clipped by a clipping factor ϵ.

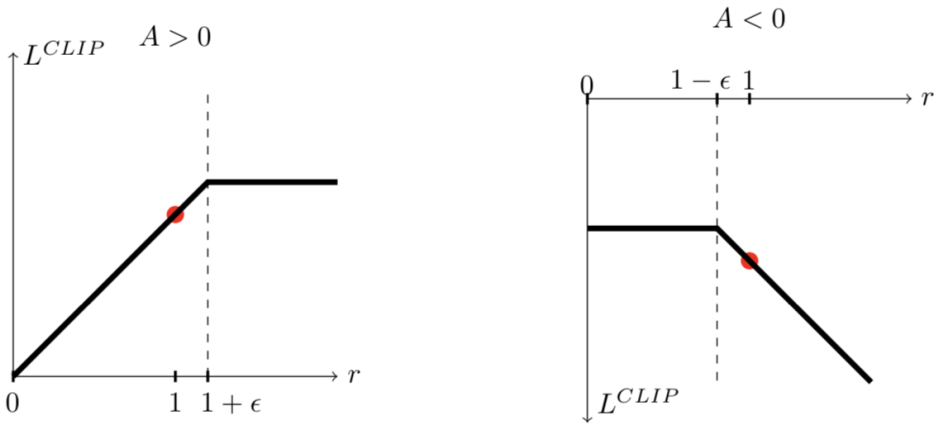

Figure 7: Surrogate function L^{CLIP} as a function of ratio [9]. Left panel is the advantage function A larger than zero and right panel is A smaller than zero. It shows how the surrogate function is clipped by a clipping factor ϵ. This method prevents extremely large policy update.

We implemented two versions of PPO agents: one is for discrete action space and the other is for continuous action space. Our implementation of PPO algorithm is described as below:

1. build the actor and critic networks separately using tensorflow: both actor and critic networks has two hidden layers with 64 units in each layer; the input of actor and critic networks is the environment state (note that we don't evaluate state and action pairs in critic agent, but just the state); the actor network outputs the probabilities of all possible actions(the action space of LunarLanderV2 is Discrete(4)) for the discrete version and actor

network outputs mean and sigma of a normal distribution (for LunnarLanderContinuousV2, the action space is two float numbers, so there are two pairs of mean and sigma to build two normal distributions of action space) for the continuous version; the critic network outputs one value which is the evaluation/baseline of the environment state. Table 1 demonstrates the architecture of our actor and critic networks. Actor and critic networks don't share hyperparamters. For the initialization of network parameters, we used the default initializer of each layer.

Table 1: Actor and Critic network of PPO built by tensorflow

layers	Actor	Critic
hidden layer	dense(64, activation='relu') dense(64, activation='relu')	dense(128, activation='relu') dense(64, activation='relu')
output layer(discrete)	dense(4, activation='softmax')	dense(1, activation='none'))
output layer(continuous)	mu=dense(2, activation='tanh') sigma=dense(2, activation='softplus')	dense(1, activation='none'))

2. roll-out and sample: we sample and store the full trajectory (state(s), action(a), reward(r) and done) with current policy during each game in training, but the length of trajectory used in each update is much shorter than the length of a full trajectory. We name this length used in update as batch size. In order to include some stochastic process and fully use the data we collect, we shuffle the full trajectory and divide it into K groups. We use the tensorflow_probability package to sample action given the probability and calculate the log probability. For continuous version, we treat the two action float numbers as two independent normal distributions and multiply them to obtain the probability of that action pair.

3. generalized advantage estimation A: for the full trajectory, we used the equations from [9] to calculate the advantage function which is used in loss calculation:

$$\begin{aligned} &\hat{A}_t = \delta_t + (\gamma\lambda)\delta_{t+1} + \cdots + \cdots + (\gamma\lambda)^{T-t+1}\delta_{T-1}, \\ &\text{where} \quad \delta_t = r_t + \gamma V\left(s_{t+1}\right) - V\left(s_t\right) \end{aligned} \tag{5}$$

r_t is the reward of the t-th step, γ is the discount factor and λ is a smoothing factor. If $\hat{A}_t$>0, the action is better than the average of all the actions in that state, so this action has a higher chance to adopted and the new policy will be increased. In step 2, when we shuffle and divide the full trajectory, we also do the sample operation to $\hat{A}_t$.

4. loss computation and policy update: Critic agent evaluates the value of a state to tell how good the state is. The target values V_target is the sum of advantage function A and evaluated values V_eva, so the loss of critic agent is mean squared error $\mathbb{E}_t(V_{target}(t) - V_{eva}(t)^2$, which is actually the mean squared $\hat{A}_t$. Because we need to apply gradients to critic network parameters, we calculate it in the former way under the gradient tape in tensorflow. For actor agent, because we want to maximize the surrogate function L^{CLIP}, the loss function

is -L^{CLIP}. Because we divide the trajectory into K groups, we compute the loss and update both critic and actor network in each group. In total, we update the policy K times in each game. The old probability is always the probability of the old policy when we sample the trajectory. In the first update/group, the new probability of a transition(at,st) is the same as the old probability; after we update the policy, the new probability of the same transition is calculated from the updated policy. When we calculate actor's loss, we normalize the advantage function $\hat{A}_t$, which gives us a much better and stable results. After calculating loss of the critic and actor networks, we use Adam to calculate gradients and update the networks.

5. training: we train the policy many times (until convergence) by repeating step 2, 3 and 4.

3.2 Experiments

As regular reinforcement learning tasks, tuning parameters is essential to make the algorithm have a good performance. In PPO, we have many parameters we can change: policy network structures, learning rate, discount factor, smoothing factor, clip factor, batch size and max step in each game. Because the observation space of LunarLander environment is Box(8) and the action space is either Discrete(4) or two float numbers, we don't need to have a complex network architecture and the current architecture with two hidden layers is enough to solve this problem. We fix the architecture, activation function, initialization function in the network. When discount factor is higher than 0.9, it usually has small impact on the results, so we fix discount factor to be 0.99. We explored the remaining parameters: learning rate of actor (default=3e-3), learning rate of critic (default=3e-3), batch size (default=32), clip factor (default=0.2), max step (default=300) and smoothing factor (default=1.0). If we don't mention these parameters, they are set to be the default values.

3.2.1 Learning rate

In this section, we used discrete PPO version to do experiments on two learning rate of actor and critic. Like usual reinforcement learning algorithms, PPO also has randomness during training. The training curve changes in same experiments: sometimes is good and sometimes is bad. So we did each experiment three times and averaged their rewards for all experiments in PPO. Figure 8 shows the results of different actor (lr_a) and critic (lr_c) learning rates. The training curves are smoothed over 20 episodes and the moving standard deviation is also over 20 episodes. We can see that compared with critic learning rate, actor learning rate has a bigger influence on the performance: converging speed and fluctuation. lr_a=3e-3 is better than lr_a=3e-4, while lr_c=3e-4, 1e-3, and 3e-3 has similar training curves. Because our critic just evaluates states and provides baselines, while actor is related to both actions and states, the learning ability is more sensitive to actor's learning rate. The three good training curves

reach convergence after 250 episodes and the final rewards are about 50.

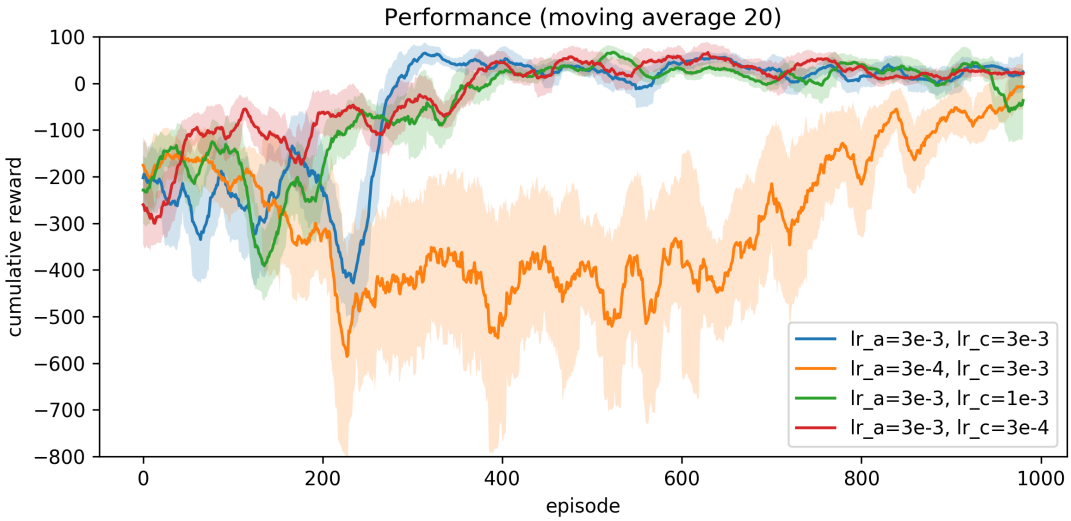

Figure 8: Training curves (moving average over 20 episodes) of different actor and critic learning rates in discrete PPO version. The shadow region of each curve is the moving standard over 20 episodes. lr_a is the learning rate of actor and lr_c is the learning rate of critic. Changing the actor learning rate has bigger influence on convergence speed, while changing the critic learning rate has smaller influence on the training curve. The convergence rewards are about 50.

3.2.2 Batch size

From this section, we used continuous PPO version to do experiments. We explored the influence of the batch size, the length of the trajectory used in each update. In Figure 9, the training curve of batch size=60 has bigger fluctuations and smaller rewards than batch size=32. This agrees with the requirement of PPO, batch size has to be much shorter the full trajectory. Because for the LunarLander environment, the landing states changes quickly with time and actions, calculating the advantage function in a shorter trajectory can help PPO learn quickly about the environment and update policy timely.

3.2.3 Clip factor

We continued to use continuous PPO version to explore the effect of clip factor ϵ in actor's loss function, which is the main difference of PPO between other algorithms, e.g., TRPO. Figure 7 demonstrates training curves of clip factor 0.1 and 0.2. Though the final rewards of clip=0.1 are close to clip=0.2, clip=0.1 has bigger fluctuations and smaller learning speed

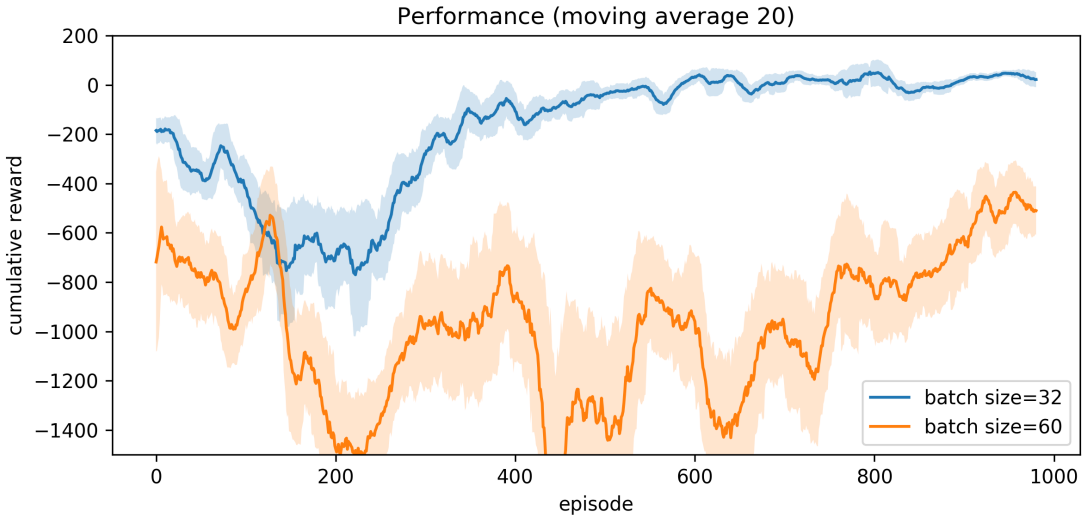

Figure 9: Training curves (moving average over 20 episodes) of two batch size: 32 and 60. The performance of batch size 32 is much better than batch size 60.

than clip=0.2 during training. This is becuase the clip factor ensure some random process cannot change the policy significantly and the policy is increased only by a clipped r_t , so smaller clip factor has better performance.

3.2.4 Maximum step

During training, we found that the spacecraft hovered in the air at the late stage of training and setting a maximum step for every game has better results than just letting the spacecraft hover. We did experiments to explore how the maximum step limit can influence results (still in continuous version). Figure 11 shows the performance of two maximum steps: 200 and 300. Maximum step=300 has better results than maximum step=200, though the final rewards are very close. Probably because 300 allows PPO have more chances to explore the environment.

3.2.5 Smoothing factor

In Equation 5, there is a smoothing factor λ that are used when calculating the advantage functions. We also tested the performance of two values of it: 0.95 and 1.0 (still in continuous version). 0.95 is the recommended value in [9] and 1 means we only consider the effect of discount factor γ. Figure 12 shows the similar results of these two values. So the smoothing factor doesn't have big impact on the results when it is very large (close to 1).

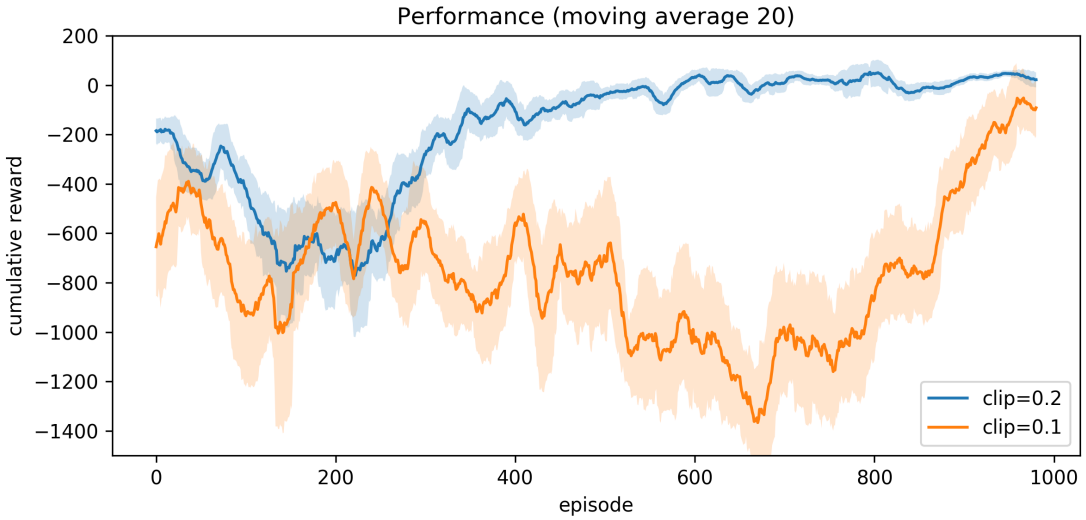

Figure 10: Training curves (moving average over 20 episodes) of two clip factors: 0.1 and 0.2. Clip=0.2 has a big convergence speed and higher reward than clip=0.1.

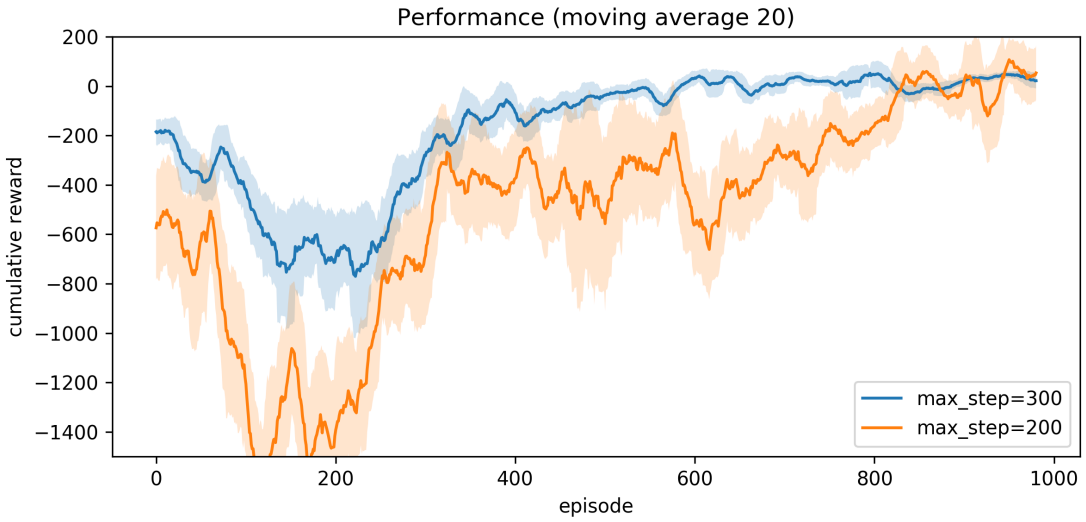

Figure 11: Training curves (moving average over 20 episodes) of two max step: 200 and 300. Maximum step constrains the maximum step in every game. The training curve of max_step=300 is more stable than the training curve of max_step=200.

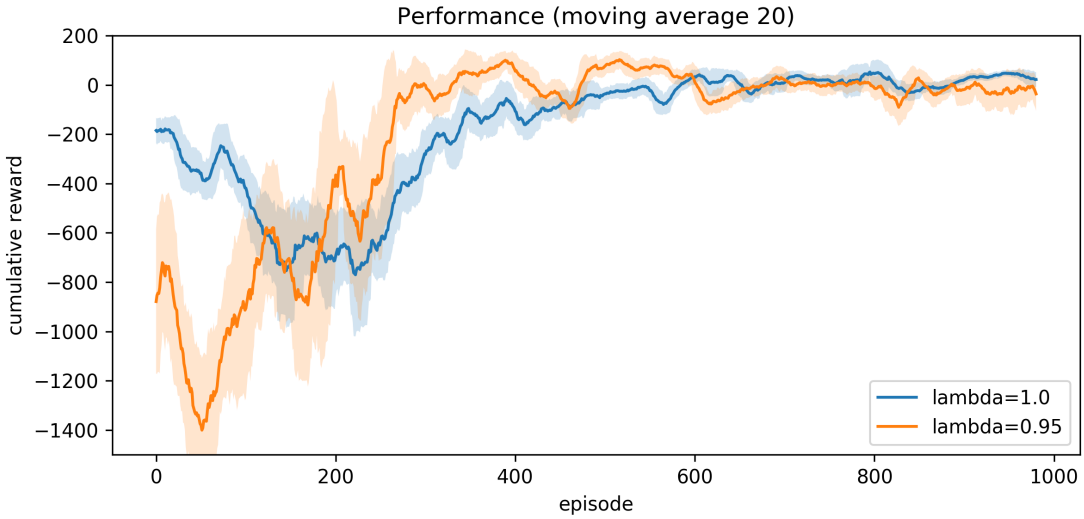

Figure 12: Training curves (moving average over 20 episodes) of two smoothing factor λ. They have similar results which means lambda above 0.95 is usually good in experiments.

4 Deep Deterministic Policy Gradient (DDPG)

Deep Deterministic Policy Gradient (DDPG) is an actor-critic, model-free algorithm based on the deterministic policy gradient that can run over continuous action spaces [10]. It inspires by 'Deep' (DQN) and 'Deterministic Policy Gradient'. DDPG refers to the target network and replay memory (also called trajectory) usages from DQN to stabilize the learning process. Its most advantage is the power for handling continuous action space type tasks which are the most cases in reality; while DQN is limited to discrete and low-dimensional action spaces. Considering its high preference in continuous space, we implement it to Lunar lander continuous environment only. And another side, policy gradient is randomly selected based on the learned action distribution, while deterministic outputs action values rather than outputting probabilities.

4.1 implementation

Here we first present the neural networks we implemented in DDPG. DDPG is a type of actor critic method, it has two distinct networks, one for the actor that will decide what to do based on the current state and one for the critic to evaluate state and action pairs. Besides the networks for action choice, we have to build the target networks as DQN to which are soft copies of the original two networks, so that DDPG learns slowly and improves stability. Overall, we have four networks: actor μ, critic Q, target μ', target critic Q'.

Here is our procedure for DDPG algorithm, we almost follow the same approach as in DDPG original paper [10].

1. Randomly initialize the critic network Q, actor μ, target networks Q', μ'.

2. Initialize *replay buffer R*. DDPG uses *replay buffer* to update neural network parameters as DQN,A3C does. We save the most recent 10,000 steps experience (*state, action, reward, next state*) to a numpy array.

3. Start the main training episode loop, initialize the observation state every episode.

    3.1. Start the step loop with maximum 150 steps (too much steps will let the lander tend to suspend in air and drop slowly) until the game is over (lander crashes or successfully lands)

        3.1.1. Select action according to current policy with extra *Gaussian noise* $\mathcal{N}$ with decaying variance to explore exploit dilemma.

        3.1.2. Execute *action a*, then get the *new state, reward* information.

        3.1.3. Store them to *memory trajectory R*, and sample random *batch size=32* from the R.

        3.1.4. Evaluate action with critic network $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$, and update critic by minimizing the mean-squared loss between memory and critic network $Loss = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))$.

        3.1.5. Update the actor policy using sampled policy gradient
        $\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu}\mu(s|\theta^\mu)|_{si}$
        where the first half of gradient is from Critic, the second half of gradient comes from the Actor to finally reach the goal: at what direction should actor to move that is more likely to obtain a larger reward.

        3.1.6. Update the target networks Q', μ' softly $\theta' \leftarrow \tau\theta + (1-\tau)\theta'$ with $\tau \ll 1$. So that the target network update is slowly and promotes greater stability.

4.2 experiment

Here we perform a series of experiments about (1)learning rate, (2) discount factor γ, (3) soft parameter, (4) batch size, **each experiment was repeated three times**. In all figures, the solid line is the average result of three runs with different random seeds of latest 20 episodes (i.e. moving average) cumulative rewards to clearly present the trend. The filled area is one time sigma uncertainty area calculated by latest 20 episodes to present the fluctuation of cumulative rewards.

4.2.1 learning rate

There are two types learning rate in DDPG: lr_{actor}, lr_{critic}. According to policy gradient theorem, if actor varies rapidly than critic, estimated Q-value will not truly reflect the value of the action, since it is dependent on previous policies. Hence we do not share the same learning rate hyper-parameter, allow critic learns a bit faster than the actor. Figure 13 is the result of three choice for learning rate. Our DDPG learns fast around episode 100 (we believe

it is because the memory trajector is filled around episode 100, and the algorithm benefit a lot from past memory), from negative reward ~ -300 to positive reward ~ 50 (this phenomenon also occurs in other experimental graphs). And the filled area becoming smaller as episode goes. However, there are still kind of randomness, for example the cumulative reward goes down at some episode suddenly, and needs a certain number of episodes to recover back. From figure 13, though the number for experiments is not sufficient(three), we can tell that the behaviors between three choice are not too much distinctive, finally we decide to choose the slightly better one curve for $lr_a = 0.001, lr_c = 0.002$ (blue curve).

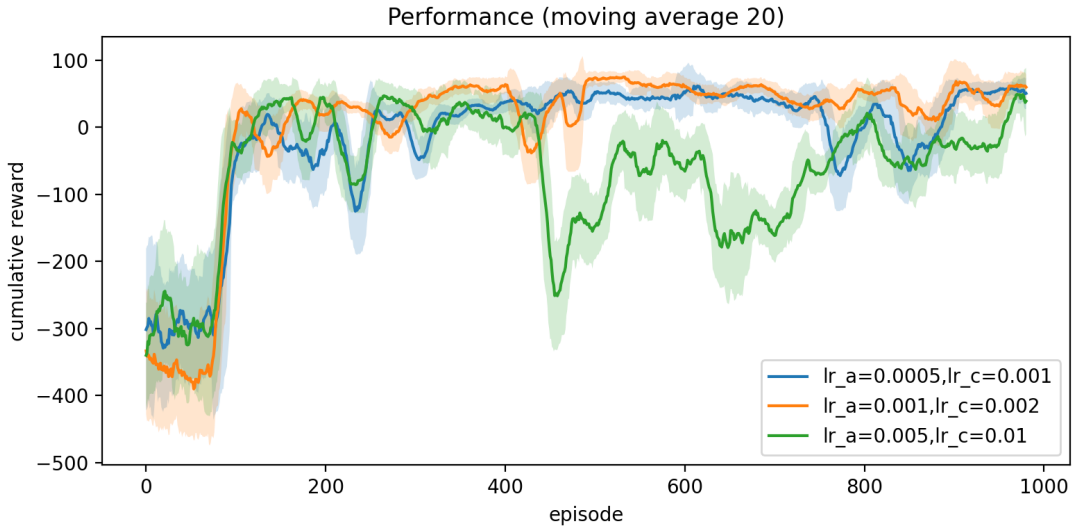

Figure 13: An experiment for three learning rate choices. Solid line is the moving average of recent 20 episodes, filled area is one time sigma uncertainty area calculated by recent 20 episodes to present the fluctuation of cumulative rewards (can represent the distribution of initial cumulative reward curve). Blue: $lr_a = 0.0005, lr_c = 0.001$; orange: $lr_a = 0.001, lr_c = 0.002$; green: $lr_a = 0.005, lr_c = 0.01$.

4.2.2 discount factor

Discount factor γ reflects how much the algorithm looks into the future. $\gamma = 0$ means the agent is short-sight only considering reward in the current iteration. In Reinforcement learning algorithms, people always use close to one value for γ. From figure 14, the behavior of three experiments are similar considering the randomness. Finally, we decide to choose the $\gamma = 0.99$ one (blue curve).

4.2.3 soft parameter

A soft update strategy consists of slowly updating target network weights with a small percent τ of evaluated network. Form figure 15, the conclusion is almost the same that there are not

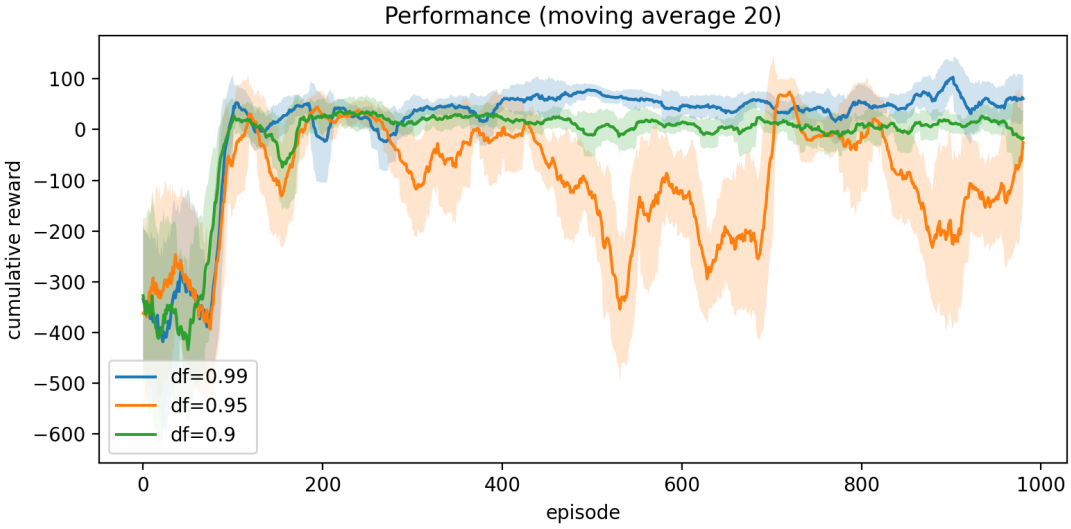

Figure 14: An experiment for three discount factor γ choices. Solid line is the moving average of recent 20 episodes, filled area is one time sigma uncertainty area calculated by recent 20 episodes to present the fluctuation of cumulative rewards (can represent the distribution of initial cumulative reward curve). Blue: $\gamma = 0.99$; orange: $\gamma = 0.95$; green: $\gamma = 0.9$.

much discrepancy between three choice. Since the uncertainty and average of orange curve is better than other two, we choose to use $\tau = 0.002$.

4.2.4 batch size

Batch size determines how many past transitions we are going to learn from memory trajectory. We explore two values: 32 and 64. From figure 16, for the same randomness reason, we find $batch\ size = 32$ may be a little stable and better.

4.3 Final parameters choice

We determine to take $lr_a = 0.001, lr_c = 0.002, \gamma = 0.99, \tau 0.002, batch\ size = 32$. where it is the blue curve in figure 17, and the cumulative reward can reach an average of ~ 80.

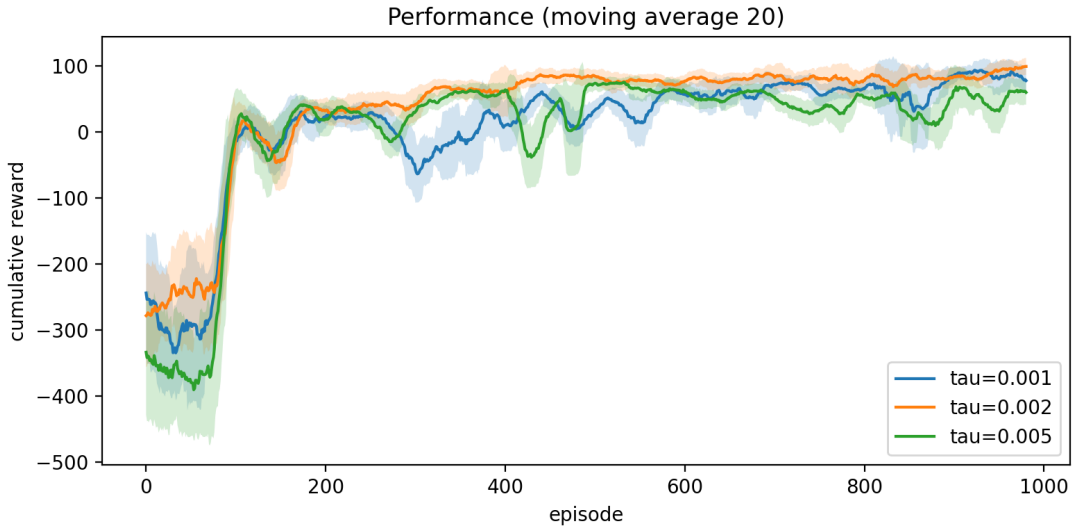

Figure 15: An experiment for three soft parameter τ choices. Solid line is the moving average of recent 20 episodes, filled area is one time sigma uncertainty area calculated by recent 20 episodes to present the fluctuation of cumulative rewards (can represent the distribution of initial cumulative reward curve). Blue: $\tau = 0.001$; orange: $\tau = 0.002$; green: $\tau = 0.005$.

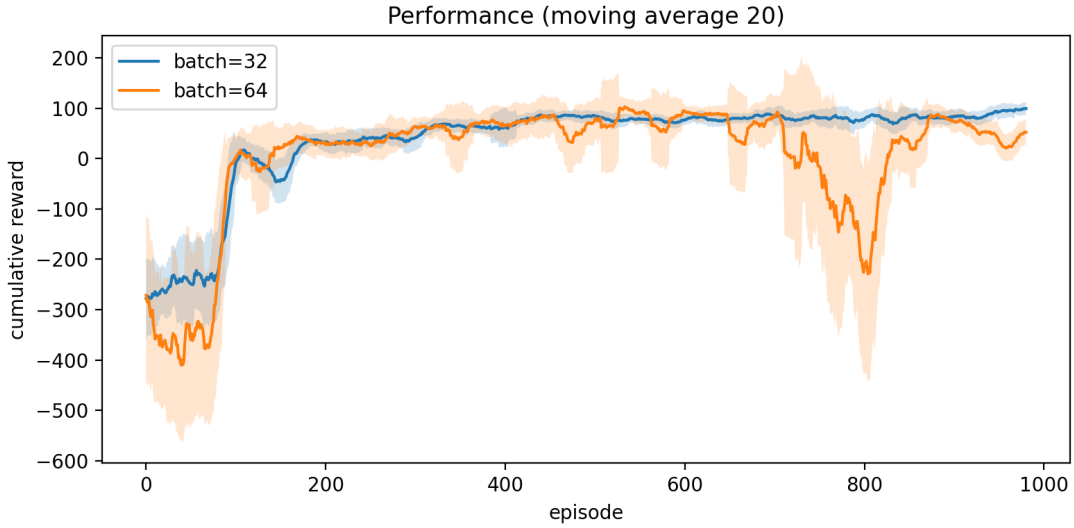

Figure 16: An experiment for two batch size choices. Solid line is the moving average of recent 20 episodes, filled area is one time sigma uncertainty area calculated by recent 20 episodes to present the fluctuation of cumulative rewards (can represent the distribution of initial cumulative reward curve). Blue: $batch\ size = 32$; orange: $batch\ size = 64$.

5 Results comparison

In this section, we compare the best results of our three algorithms in both discrete and continuous action space versions. Figure 17 shows the performance of A3C, PPO and DDPG. In discrete version, A3C has the best results with the smallest variance and highest rewards close to 200, although it needs more episodes for every individual worker to train in its local network and update the global networks. In continuous version, DDPG is better than PPO and has a final reward close to 100. PPO works robustly in both discrete and continuous versions with final rewards close to 50. PPO and DDPG have faster convergence speed than A3C and can reach convergence after about 300 episodes.

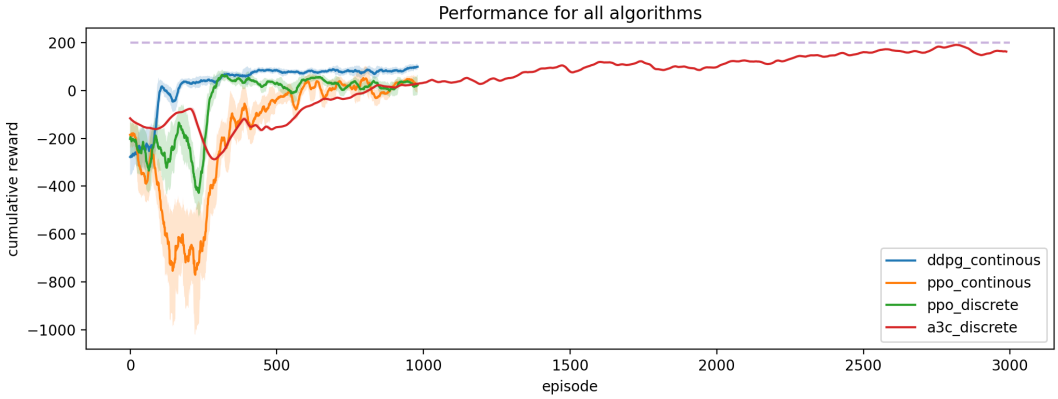

Figure 17: Best training curves of A3C, PPO and DDPG on both discrete and continuous LunarLander versions.

6 Conclusion and Discussion

In this assignment, we implemented three policy gradient algorithms: A3C, PPO and DDPG to solve LunarLander problems of both discrete and continuous action space versions. We tuned their parameters to find the best results and compared their performance: A3C has the best results in discrete version, DDPG has the best results in continuous version, and PPO works stably in both discrete and continuous versions.

- In our implementation of PPO, we only used single thread. We argue that PPO's performance can be improved by using multi-threads, that's simulating multiple trajectories at the same time. When we collect many trajectories for updating, we can beat some problems caused by random process and have more chances to explore the environment, e.g., exploring the correct landing state with the highest scores in the LunarLander problem. There are also other things we can study, for example, adding critic loss and entropy bonus to actor loss or using KLpenalty when calculating actor loss.

- For A3C, we not only implement it with the discrete environment (LunarLander-v2) , but also try the continuous one (LunarLanderContiuous-v2). However, performance in the continuous space is much worse than it in the discrete one. It is probably due to the synchronized gradient update keeps the training more cohesive and potentially to make convergence faster than the asynchronized one. We have read lost of blogs and posts on it. It is said that A2C has been shown to be able to utilize GPUs more efficiently and work better with large batch sizes while achieving same or better performance than A3C.

- When experimenting A3C algorithm implemented with TensorFlow, we found that the program runs slowly. Then we try to move that to pytorch, as working with multiprocessing, TensorFlow is not that great due to its low compatibility with multiprocessing. There is GIL in TensorFlow which limits the multi-thread processing. Pytorch is great for parallel training and can be used in both discrete and continuous action environments. In the end, we follow a tutorial and run the Pytorch code we try to implement. Although the program is not perfect, and it is not as suitable for the LunarLander environment as the TensorFlow version, but it makes good use of the device hardware resource to boost the calculation and greatly reduces the running time.

References

[1] OpenAI. https://gym.openai.com/envs/lunarlander-v2/, 2021-May-7.

[2] Richard S. Sutton, David Mcallester, Satinder Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *In Advances in Neural Information Processing Systems 12*, pages 1057–1063. MIT Press, 2000.

[3] Aske Plaat. Learning to play-reinforcement learning and games, 2020.

[4] R.J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning, 1992.

[5] Thomas Degris, Patrick M. Pilarski, and Richard S. Sutton. Model-free reinforcement learning with continuous action in practice. In *2012 American Control Conference (ACC)*, pages 2177–2182, 2012.

[6] Yuhuai Wu, Elman Mansimov, Shun Liao, Alec Radford, and John Schulman. Openai baselines: ACKTR and A2C, August 18, 2017.

[7] GroundAI's blog Schematic representation of Asynchronous Advantage Actor Critic algorithm (A3C) algorithm, 2018.

[8] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning, 2016.

[9] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017.

[10] 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, 2015.