Deep Learning Lab7 Report

陳以瑄 (ID: 313551047)

1 Introduction

In this lab, I implement the A2C and PPO algorithms to solve the Pendulum and Walker2d tasks. In Task 1, the A2C agent achieves a return above -150 within approximately 100k steps. In Task 2, the PPO agent reaches the same performance in just 40k steps. In Task 3, the PPO agent obtains a return of 2,500 within 1M steps.
I also conduct an empirical study to evaluate several techniques for improving PPO performance. The results show that entropy regularization encourages exploration and leads to faster learning. Additionally, using a small clipping value for the surrogate objective makes the policy overly conservative and slows down training.
Other effective modifications include:

1. adjusting the learning rate at every step
2. using Smooth L1 loss instead of MSE loss for value estimation
3. normalizing the advantage estimates
4. applying the `tanh` activation function instead of `ReLU`

These adjustments significantly enhance the PPO agent's performance on the Walker2d task.

2 Implementation

2.1 Task 1: A2C in Pendulum-v1

2.1.1 Network Architecture

The Pendulum-v1 environment is a continuous control task with a low-dimensional observation space; therefore, both the actor and critic networks are kept simple.
Actor Network
The actor network consists of two fully connected hidden layers, each with 64 units and ReLU activations, followed by two separate output layers: *mu_layer* and *log_std_layer*.

- Line 6: *mu_layer*: outputs the mean (μ) of the Gaussian policy.
- Line 16: Since the action range of Pendulum-v1 is [-2, 2], the output mean is scaled using tanh and multiplied by 2.

- Line 7: *log_std_layer*: to make training more stable, we let the model learn the log standard deviation, rather than learning the standard deviation directly.
- Line 17: The log σ is transformed via softplus to ensure it is positive.
- Line 18: Convert log σ to the standard deviation (std).

```python
1  class Actor(nn.Module):
2      def __init__(self, in_dim: int, out_dim: int, hidden_dim: int = 64):
3          super(Actor, self).__init__()
4          self.layer1 = nn.Linear(in_dim, hidden_dim)
5          self.layer2 = nn.Linear(hidden_dim, hidden_dim)
6          self.mu_layer = nn.Linear(hidden_dim, out_dim)
7          self.log_std_layer = nn.Linear(hidden_dim, out_dim)
8          initialize_uniformly(self.mu_layer)
9          initialize_uniformly(self.log_std_layer)
10
11     def forward(self, state: torch.Tensor) -> torch.Tensor:
12         x = self.layer1(state)
13         x = F.relu(x)
14         x = self.layer2(x)
15         x = F.relu(x)
16         mu = torch.tanh(self.mu_layer(x)) * 2 # action space [-2,2]
17         log_std = F.softplus(self.log_std_layer(x)) # softplus >=0
18         std = torch.exp(log_std)
19         dist = Normal(mu, std)
20         action = dist.sample()
21         return action, dist
```

Critic Network

Similar to the actor, the critic consists of two hidden layers with 64 units and ReLU activations, followed by a linear output layer that predicts the value function estimate V(s).

```python
class Critic(nn.Module):
    def __init__(self, in_dim: int, hidden_dim: int = 64):
        super(Critic, self).__init__()
        self.layer1 = nn.Linear(in_dim, hidden_dim)
        self.layer2 = nn.Linear(hidden_dim, hidden_dim)
        self.out = nn.Linear(hidden_dim, 1)
        initialize_uniformly(self.out)

    def forward(self, state: torch.Tensor) -> torch.Tensor:
        x = self.layer1(state)
        x = F.relu(x)
        x = self.layer2(x)
        x = F.relu(x)
        value = self.out(x)
        return value
```

2.1.2 Stochastic Policy Gradient & the TD Error

After collecting a single transition $(s, a, r, s', \text{done})$ from the environment, we compute the temporal-difference target and use it to estimate the TD error for both value function learning and policy optimization.

Critic loss

The definition of value loss is: $\mathcal{L}_{\text{critic}}(w) = (r + \gamma V_w(s') - V(s))^2$

The implementation is:

```
td_target = reward + self.gamma* self.critic(next_state) * mask
state_value = self.critic(state)
value_loss = F.mse_loss(state_value, td_target.detach())

self.critic_optimizer.zero_grad()
value_loss.backward()
self.critic_optimizer.step()
```

Actor loss

The definition of policy loss is: $\mathcal{L}_{\text{actor}}(\theta) = -\log \pi_\theta(a \mid s) \cdot (r + \gamma V(s') - V(s))$

The implementation is :

```
advantage = (td_target - state_value).detach()
policy_loss = -(log_prob * advantage) - self.entropy_weight * log_prob

self.actor_optimizer.zero_grad()
policy_loss.backward()
self.actor_optimizer.step()
```

2.1.3 Enforce Exploration

Although A2C is an on-policy algorithm, it incorporates exploration explicitly through two mechanisms:

Stochastic Policy

The policy is probabilistic, modeled as a Normal distribution:

```
dist = Normal(mu, std)
action = dist.sample()
```

This ensures that even for the same state, different actions may be selected, allowing the agent to explore the environment.

Entropy Regularization

A soft penalty is added to the actor’ s loss to discourage the policy from becoming too certain, thereby promoting exploration:

```
policy_loss = -(log_prob * advantage) - self.entropy_weight * log_prob
```

2.2 Task 2: PPO in Pendulum-v1

2.2.1 Network Architecture

Although in task 2 we also attempt to solve the Pendulum-v1 problem as in task 1, I observe that PPO is more powerful and stable. Therefore, this time I only use a single hidden layer of dimension 64 for both the actor and the critic. Another adjustment is that, to ensure the standard deviation remains within a reasonable range and avoids numerical instability, I normalize log_std using a scaled tanh transformation to ensure it lies within [-20,0]. This is implemented as follows:

```python
class Actor(nn.Module):
    def __init__(self, in_dim: int, out_dim: int, log_std_min: int = -20,
 log_std_max: int = 0, hidden_dim: int = 64):
        super(Actor, self).__init__()
        self.log_std_min = log_std_min
        self.log_std_max = log_std_max

        self.layer1 = nn.Linear(in_dim, hidden_dim)
        self.mu_layer = init_layer_uniform(nn.Linear(hidden_dim, out_dim))
        self.log_std_layer = init_layer_uniform(nn.Linear(hidden_dim,
 out_dim))

    def forward(self, state: torch.Tensor) -> torch.Tensor:
        x = self.layer1(state)
        x = F.relu(x)
        mu = torch.tanh(self.mu_layer(x)) * 2 # action space [-2,2]
        log_std = torch.tanh(self.log_std_layer(x)) # softplus >=0
        log_std = self.log_std_min + (log_std + 1) *\
                  (self.log_std_max - self.log_std_min)/2
        std = torch.exp(log_std)

        dist = Normal(mu, std)
        action = dist.sample()
        return action, dist

class Critic(nn.Module):
    def __init__(self, in_dim: int, hidden_dim: int = 64):
        super(Critic, self).__init__()
        self.layer1 = nn.Linear(in_dim, hidden_dim)
        self.out = init_layer_uniform(nn.Linear(hidden_dim, 1))

    def forward(self, state: torch.Tensor) -> torch.Tensor:
        x = self.layer1(state)
        x = F.relu(x)
        value = self.out(x)
        return value
```

2.2.2 Clipped Objective

In PPO, the actor objective is defined as the clipped surrogate objective:

$$L_{\text{CLIP}}(\theta) = \min\left(\rho_{s,a}(\theta)A^{GAE}(s,a),\ \text{clip}\left(\rho_{s,a}(\theta),\ 1-\epsilon,\ 1+\epsilon\right)A^{GAE}(s,a)\right)$$

where $\rho_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$ is the probability ratio, and A^{GAE} is the advantage estimate.
The implementation is:

```python
_, dist = self.actor(state)
log_prob = dist.log_prob(action)
ratio = (log_prob - old_log_prob).exp()

# actor_loss
surrogate = ratio * adv
clipped_surrogate = torch.clamp(ratio, 1-self.epsilon, 1+self.epsilon)* adv

entropy = dist.entropy().mean()
actor_loss = (-torch.min(surrogate, clipped_surrogate)).mean()\
             - self.entropy_weight*entropy

self.actor_optimizer.zero_grad()
actor_loss.backward()
self.actor_optimizer.step()
```

As for the critic loss, it is the same as in A2C, and is computed using mean squared error between the predicted value and the return:

```python
# critic_loss
state_value = self.critic(state)
critic_loss = F.mse_loss(state_value, return_)

self.critic_optimizer.zero_grad()
critic_loss.backward(retain_graph=True)
self.critic_optimizer.step()
```

2.2.3 Estimator of GAE

Instead of computing the advantage function using a single-step return, GAE provides a more flexible and stable estimate by using a weighted sum of temporal differences.
Because this is a weighted sum, we can compute the advantage iteratively in reverse order (from the final time step to the beginning).
The advantage at time step t: $\hat{A}t = \delta_t + \gamma\lambda\delta t + 1 + \gamma^2\lambda^2\delta_{t+2} + \cdots$
,where the TD residual is: $\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$
This formulation can be rewritten recursively as: $\hat{A}_t = \delta_t + \gamma\lambda\hat{A}_{t+1}$
Implementation Details:

- Line 5: Initializes the advantage accumulator and begins reverse iteration.
- Line 7: Computes the TD residual δ_t
- Line 8: Recursively computes $\hat{A}_t$ and adds it to $V(s_t)$ to form the return.

```python
1  def compute_gae(next_value: list, rewards: list, masks: list, values: list,
      gamma: float, tau: float) -> List:
2      gae = 0
3      gae_returns = []
4      next_v = next_value
5      for reward,value,mask in zip(reversed(rewards),reversed(values),\
6                                   reversed(masks)):
7          delta = reward + gamma * next_v * mask - value
8          gae = delta + gamma * tau * gae * mask
9          gae_returns.insert(0,gae+value)
10         next_v = value
11     return gae_returns
```

2.2.4 Collect samples from the environment

Samples are collected by running the current policy in the environment over a fixed number of steps (rollout_len).

2.2.5 Enforce Exploration

Similar to A2C, we use a stochastic policy and entropy regularization to encourage exploration, as described in Section 2.1.3.

2.3 Task 3: PPO in Walker2d-v4

2.3.1 Network Architecture

To apply PPO to the Walker2d-v4 environment, I use a slightly deeper network than in Task 2. While the PPO algorithm remains unchanged, the increased complexity and higher-dimensional state-action space in Walker2d require a deeper architecture to improve representation capacity and training stability.
Specifically, I use two hidden layers instead of one for both the actor and critic networks. Each hidden layer has 64 units. Furthermore, we use the tanh activation function rather than ReLU. Empirically, I observe that tanh provides more stable training in this continuous control environment. Another difference is that the action space is bounded within $[-1, 1]$, so I apply a `tanh` activation on the output layer

```python
class Actor(nn.Module):
    def __init__(self, in_dim, out_dim, log_std_min=-20, log_std_max=0,
   hidden_dim=64):
        super(Actor, self).__init__()
        self.log_std_min = log_std_min
        self.log_std_max = log_std_max

        self.layer1 = nn.Linear(in_dim, hidden_dim)
        self.layer2 = nn.Linear(hidden_dim, hidden_dim)
        self.mu_layer = init_layer_uniform(nn.Linear(hidden_dim, out_dim))
        self.log_std_layer = init_layer_uniform(nn.Linear(hidden_dim,
    out_dim))

    def forward(self, state):
        x = self.layer1(state)
        x = F.tanh(x)
        x = self.layer2(x)
        x = F.tanh(x)
        mu = torch.tanh(self.mu_layer(x))  # Output bounded to [-1, 1]
        log_std = torch.tanh(self.log_std_layer(x))
        log_std = self.log_std_min + (log_std + 1) * (self.log_std_max -
    self.log_std_min) / 2
        std = torch.exp(log_std)

        dist = Normal(mu, std)
        action = dist.sample()
        return action, dist

class Critic(nn.Module):
    def __init__(self, in_dim, hidden_dim=64):
        super(Critic, self).__init__()
        self.layer1 = nn.Linear(in_dim, hidden_dim)
        self.layer2 = nn.Linear(hidden_dim, hidden_dim)
        self.out = init_layer_uniform(nn.Linear(hidden_dim, 1))

    def forward(self, state):
        x = self.layer1(state)
        x = F.tanh(x)
        x = self.layer2(x)
        x = F.tanh(x)
        value = self.out(x)
        return value
```

2.4 Weight & Bias

I just followed the sample code and logged the current step count, actor loss, and critic loss after each model update (i.e., every environment step). I also logged the episode number and total reward (return) after each episode finished.

3 Additional Training Strategy

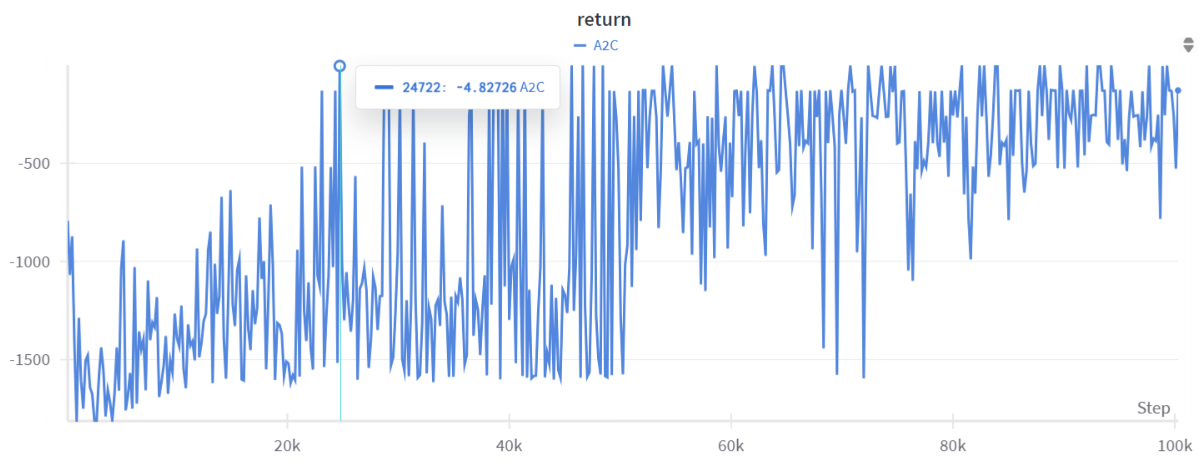
3.1 Training Curves

3.1.1 Task 1: A2C in Pendulum-v1

Hyperparameters:

- Actor learning rate (–actor-lr): 1e-4
- Critic learning rate (–critic-lr): 1e-3
- Discount factor (–discount-factor): 0.9
- Entropy weight (–entropy-weight): 1e-2
- Number of hidden layers: 2
- Hidden layer dimension (–hidden-dim): 64

Training result:
From the training curve, the A2C agent reaches approximately -4 around 20,000 steps and stabilizes around 100,000 steps.



The seed in testing that reach -150 point:

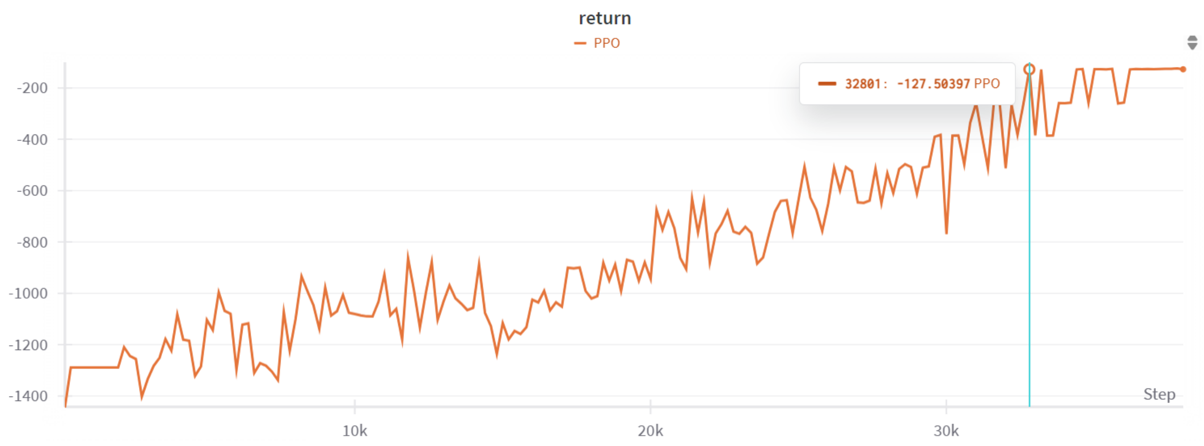
Seed	0	1	2	6	7	8	12	15
Reward	-132.11	-0.59	-127.61	-0.65	-129.30	-132.13	-134.88	-128.88
Seed	16	18	19	20	22	23	24	26
Reward	-2.68	-131.65	-2.13	-129.00	-128.01	-130.00	-128.38	-0.27
Seed	27	30	33	35				
Reward	-132.90	-129.86	-1.37	-128.89				

3.1.2 Task 2: PPO in Pendulum-v1

Hyperparameters:

- Actor learning rate (`--actor-lr`): 1e-3
- Critic learning rate (`--critic-lr`): 5e-3
- Discount factor (`--discount-factor`): 0.9
- Entropy weight (`--entropy-weight`): 1e-2
- Soft update coefficient (`--tau`): 0.8
- Batch size (`--batch-size`): 64
- Clipping parameter (`--epsilon`): 0.2
- Number of hidden layers: 1
- Hidden layer dimension (`--hidden-dim`): 64

Training result:
From the training curve, we can see that the PPO agent reaches approximately -127 at around 32,000 steps and stabilizes around 40,000 steps.



The seed in testing that reach -150 point:

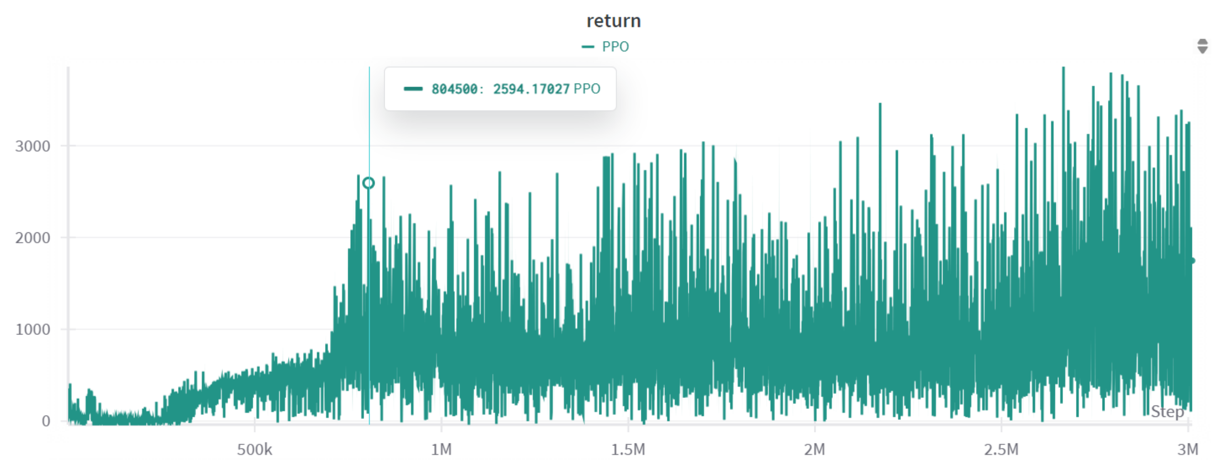
Seed	0	1	5	6	7	8	15	16
Reward	-130.18	-1.12	-121.35	-0.51	-127.34	-131.61	-125.79	-132.14
Seed	18	19	20	21	22	23	24	26
Reward	-130.25	-2.15	-132.15	-124.87	-126.92	-126.44	-128.12	-0.98
Seed	27	33	35	37				
Reward	-128.94	-2.22	-128.01	-126.29				

3.1.3 Task 3: PPO in Walker2d-v4

Hyperparameters:

- learning rate (`--actor-lr` and `--critic-lr`): 3e-4
- Discount factor (`--discount-factor`): 0.99
- Entropy weight (`--entropy-weight`): 1e-2
- Soft update coefficient (`--tau`): 0.95
- Batch size (`--batch-size`): 64
- Clipping parameter (`--epsilon`): 0.2
- Rollout length (`--rollout-len`): 2048
- Number of hidden layers: 2
- Hidden layer dimension (`--hidden-dim`): 64

Training result:
From the training curve, we can see that the PPO agent reaches a score higher than 2500 at around 800k steps.



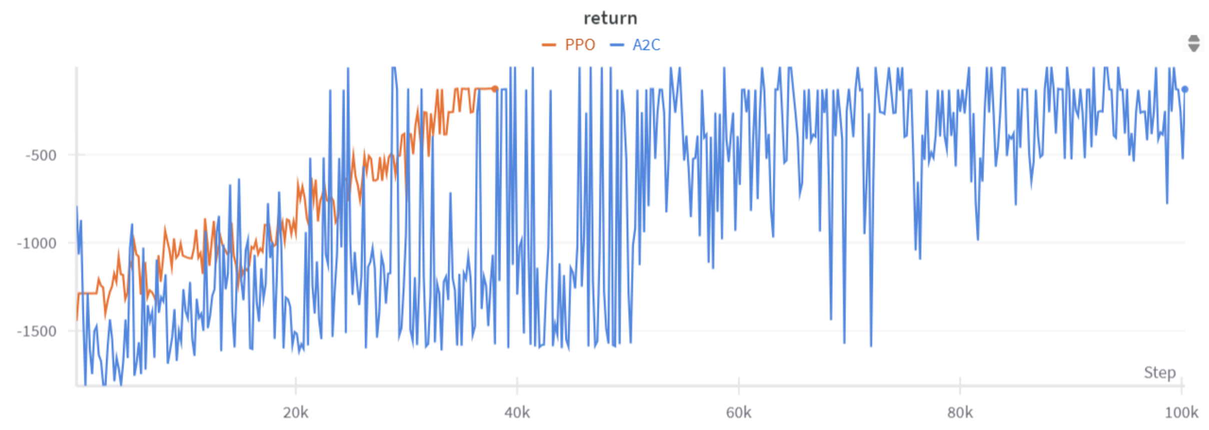
The seed in testing that reach 2500 point:

Seed	0	7	8	14	17	28	31	32
Reward	2587.16	2528.43	2595.67	2681.26	2594.50	2625.21	2524.58	2536.73
Seed	33	34	36	37	39	44	49	50
Reward	2505.21	2507.77	2505.87	2676.75	2584.56	2543.42	2518.51	2834.80
Seed	51	55	56	57				
Reward	2530.07	2606.91	2566.19	2546.88				

3.2 Compare the sample efficiency and training stability

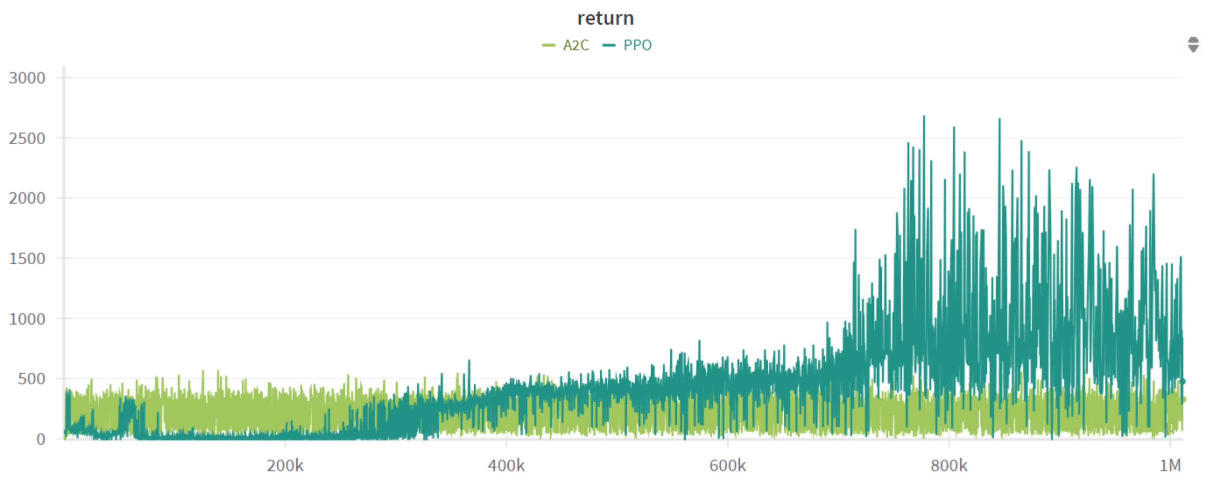
3.2.1 Pendulum-v1

By comparing the training curves of A2C and PPO in the Pendulum environment, we observe that although both methods eventually reach the target return of -150 within 200k timesteps, their learning behaviors differ significantly. A2C shows high instability, with returns fluctuating considerably up to around 100k timesteps. In contrast, PPO exhibits a stable and consistent improvement, converging to the target return in approximately 40k timesteps.

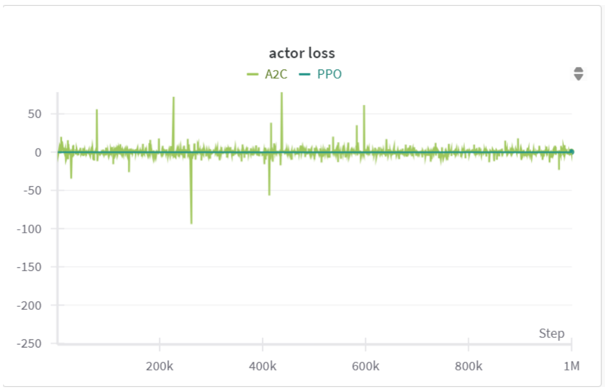


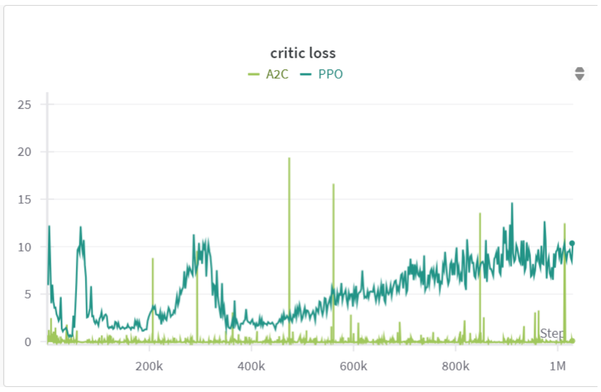
3.2.2 Walker2d-v4

In the Walker2d environment, despite experimenting with several hyperparameter settings for A2C, none were able to achieve the target return of 2,500. The plotted results use the same hyperparameters as PPO (listed in Section 3.1.3.), under which A2C consistently fails to reach the goal—its returns fluctuate between 0 and 500 throughout training.



When examining the actor and critic losses, we observe that the variance of the actor loss in A2C is significantly higher than in PPO, suggesting that A2C's policy updates are less stable and may lead to inconsistent learning. Additionally, while the critic loss in PPO some trend over training; the critic loss in A2C remains highly volatile, further highlighting the instability of the training process.

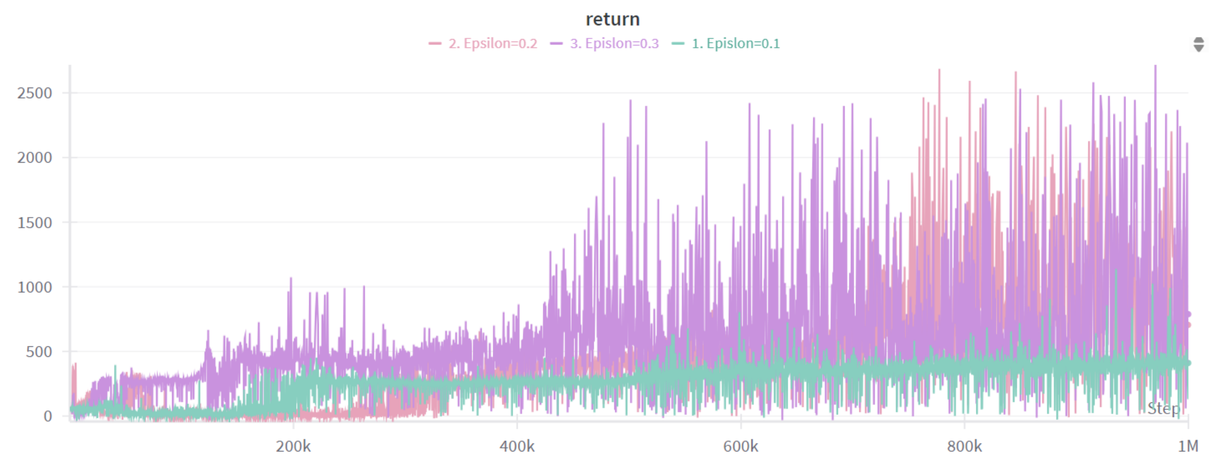




3.3 Empirical Study

3.3.1 Clipping Parameter

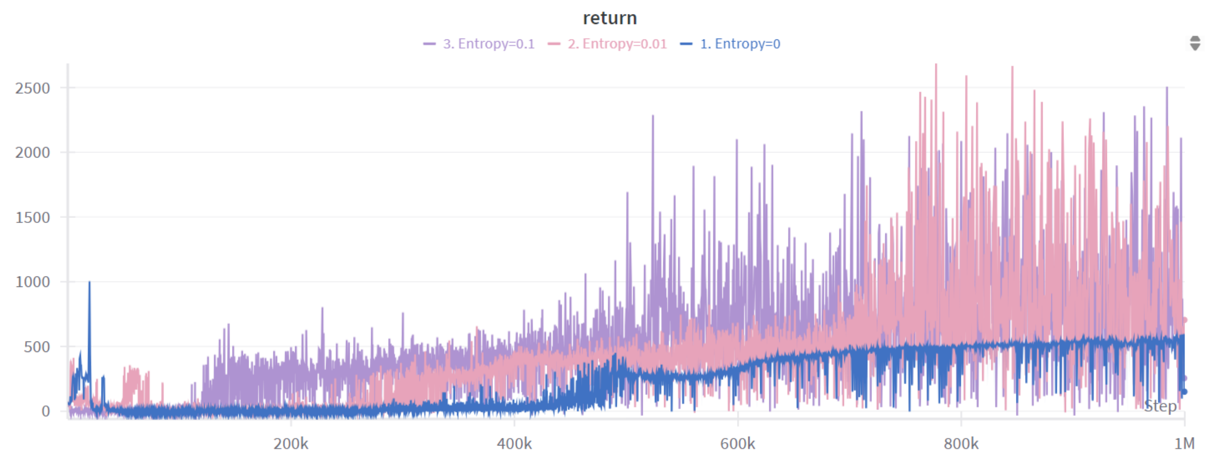
The plot compares PPO performance on Walker2d using different clipping values: 0.1 (turquoise line), 0.2 (pink line), and 0.3 (purple line). The smallest value, = 0.1, shows no



clear improvement. The largest, = 0.3, learns faster and nearly reaches 2,500 by 400,000 steps. However, = 0.2 performs best overall, balancing learning speed and stability. This suggests that = 0.1 is too small, making the policy updates overly conservative and potentially causing the agent to get stuck in suboptimal behaviors. In contrast, = 0.2 and = 0.3 allow more flexible updates, striking a better balance between exploration and stability.

3.3.2 Entropy Coefficient

In the Walker2d PPO experiments, we tested different entropy coefficients: 0 (blue line), 0.01 (pink line), and 0.1 (purple line). Without entropy regularization (coefficient = 0),



the learning progress is minimal, indicating a lack of exploration. With a higher entropy coefficient of 0.1, the agent begins to improve earlier, suggesting that stronger exploration helps escape poor initial policies. However, the best final performance is achieved with an entropy coefficient of 0.01, which balances early exploration and long-term stability.

4 Additional analysis on other training strategies

4.1 Learning Rate Scheduler in PPO Walker2d-v4

Initially, I used `MultiStepLR`, which updates the learning rate every few epochs. However, this approach failed to reach the target return of 2,500 within 1M steps. I then referred to the PPO implementation provided by OpenAI [1], where a schedule function is used to update the learning rate at every timestep. This method indeed help me to solve the task 3. The learning rate scheduler is implemented as follows:

```
class PPOAgent:
    def __init__(self, env: gym.Env, args):
        lr_lambda=lambda step:max((1.0-float(step)/float(1_000_000 + 1))\
            ,0.00005/3e-4)
        self.A_scheduler = LambdaLR(self.A_optimizer, lr_lambda=lr_lambda)
        self.C_scheduler = LambdaLR(self.C_optimizer, lr_lambda=lr_lambda)
    def train(self):
        for ep in tqdm(range(1, self.num_episodes)):
            for _ in range(self.rollout_len):
                action = self.select_action(state)
                action = action.reshape(self.action_dim,)
```

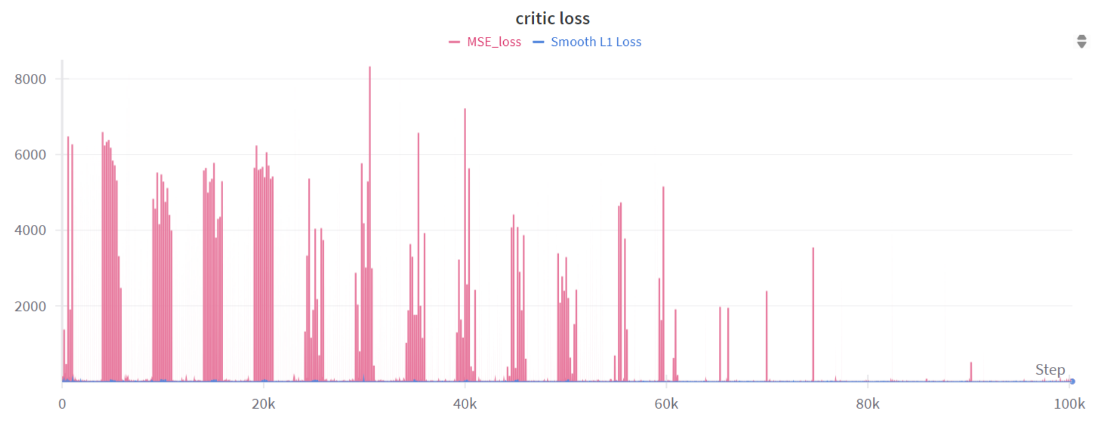
[1] https://github.com/openai/baselines/blob/master/baselines/ppo2/ppo2.py

```
                next_state, reward, done = self.step(action)

                state = next_state
                score += reward[0][0]
                self.A_scheduler.step()
                self.C_scheduler.step()
```

4.2 Smooth L1

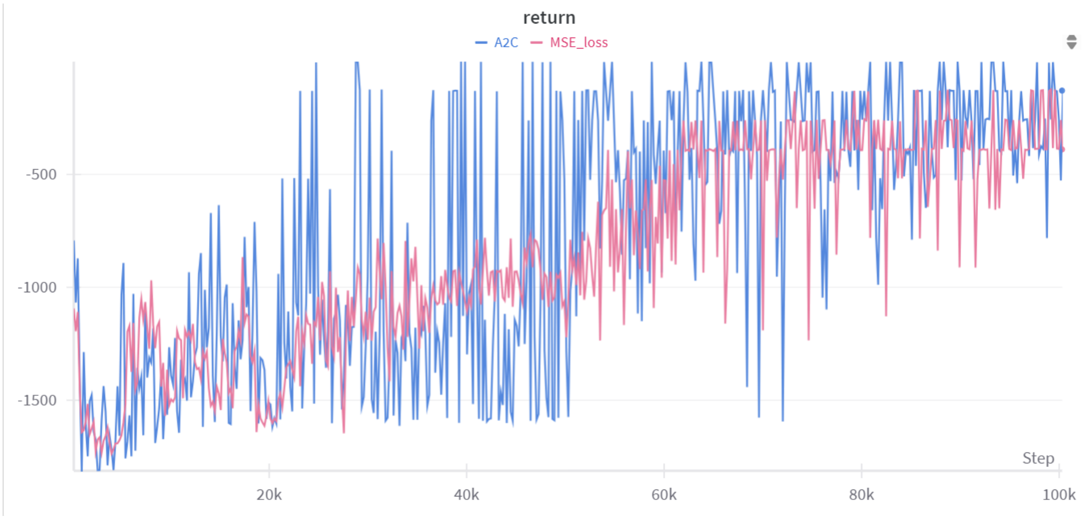
In Task 1, I observed that using `MSELoss` to compute the critic loss resulted in high variance during training.



To address this, I replaced it with `SmoothL1Loss`. The implementation:

```
td_target = reward + self.gamma* self.critic(next_state) * mask
state_value = self.critic(state)
#value_loss = F.mse_loss(state_value, td_target.detach())
value_loss = F.smooth_l1_loss(state_value, td_target.detach())
```

Although the training curve with `SmoothL1Loss` appears more fluctuating, it enables the agent to reach higher returns more quickly. This suggests that `SmoothL1Loss` provides a better trade-off between stability and learning speed in this setting.



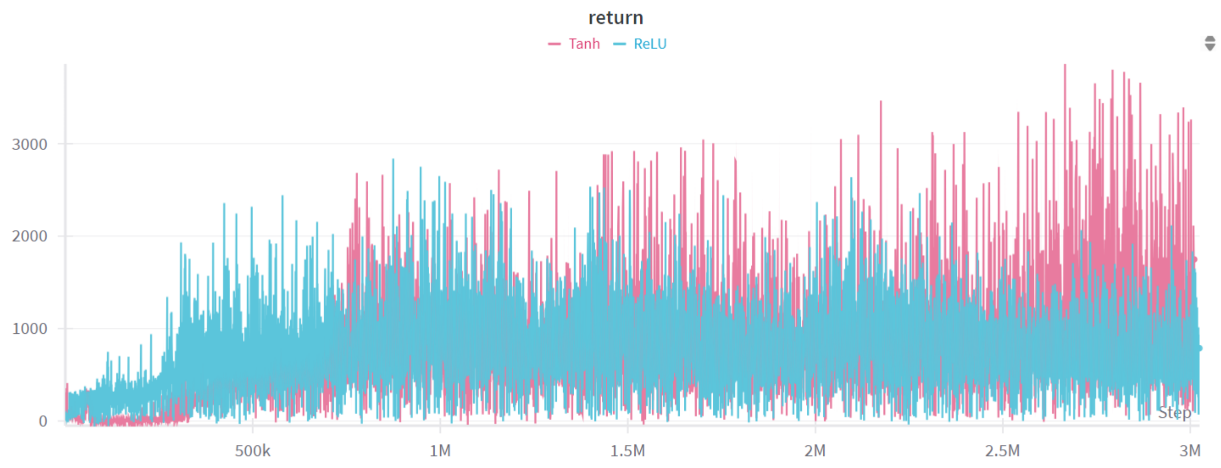
4.3 Normalize Advantages

Based on the article [2], which presents 10 practical tricks to improve PPO performance, I found that advantage normalization has a significant impact. To implement this technique, I modified the `update_model` function by adding the following line to normalize the advantages:

```python
def update_model(self, next_state: np.ndarray) -> Tuple[float, float]:
    advantages = returns - values
    advantages = (advantages-advantages.mean())/(advantages.std()+1e-8)
```

4.4 Activation Function: ReLU v.s. Tanh

Initially, I used the ReLU activation function for the Walker2d environment, following the same setup as in the Pendulum task. However, I observed that with ReLU, the agent's performance failed to improve over the long term. To address this, I switched to the `tanh` activation function. Although `tanh` led to a slower initial improvement, the training curve continued to increase steadily over time. This suggests that `tanh` provides smoother gradients and better long-term learning stability for the Walker2d environment.



[2]`https://zhuanlan.zhihu.com/p/512327050`