DLP
Lab 7: Policy-Based Reinforcement Learning Report

林睿騰(Rui-Teng Lin)
313551105

May 20, 2025

Contents

1 Introduction

In this lab, we implement a A2C agent and a PPO agent to solve on Pendulum and Walker2d environments.

1.1 A2C

A2C is a policy-based reinforcement learning algorithm. It uses a policy network to select actions and a value network to estimate the value of the current state. It uses the advantage function to update the policy network, so it is named as Advantage Actor-Critic. Although it is better than the original actor-critic algorithm, it is still not the best policy-based reinforcement learning algorithm.

It requires a lot of samples to train the model, so it is not efficient; and also it is not stable, it needs explicit exploration to enforce the policy to explore the environment.

1.2 PPO

PPO is a policy-based reinforcement learning algorithm. It uses a policy network to select actions and a value network to estimate the value of the current state. It uses the clipped surrogate objective function to update the policy network, so it is named as Proximal Policy Optimization.

Unlike A2C, PPO is more stable and efficient, and exploration can be done implicitly, so it is more popular in the research community.

2 Implementation Details

2.1 Policy Gradient and TD Error for A2C

When calculating the policy gradient, we firstly calculate the advantage $A(s,a)$:

$$A(s,a) = r + \gamma V(s') - V(s) \tag{1}$$

where r is the reward, γ is the discount factor, $V(s)$ is the value function, and $V(s')$ is the value function of the next state.

Then we use the following formula to update the policy network:

$$\nabla_\theta J(\theta) = \mathbb{E}_{s\sim\rho^\theta}[\nabla_\theta \log \pi_\theta(a|s) A(s,a)] \tag{2}$$

To calculate the TD error, we use the following formula, which is just the same as the advantage function:

$$\delta = r + \gamma V(s') - V(s) \tag{3}$$

After that, we can write them into forms of loss functions:

$$L_A(\theta) = \mathbb{E}_{s\sim\rho^\theta}[\nabla_\theta \log \pi_\theta(a|s) A(s,a)] \tag{4}$$
$$L_C(\theta) = \mathbb{E}_{s\sim\rho^\theta}[\delta^2] \tag{5}$$

which is better for us to implement.

For implementation, given the reward r, the value function $V(s)$, and the next state s', we can calculate the advantage $A(s,a)$ and the TD error δ using the above formulas.

```
s, a, r, d, n_s = self.transitions

next_value = self.get_value(n_s)
values = self.critic(states)

value_target = torch.tensor(r + self.gamma * next_value * (1 - d)).float().to(self.device).reshape(-1, 1)
value_loss = F.mse_loss(values, value_target)

advantage = value_target - values
policy_loss = -(log_probs * advantage.detach()).mean() - self.entropy_weight * dist.entropy().mean()

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

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

2.2 Clipped Surrogate Objective for PPO

To implement the clipped surrogate objective for PPO, we can use the following formula:

$$L_C(\theta) = \mathbb{E}_{s\sim\rho^\theta}[\min(r(\theta)A(s,a), \text{clip}(r(\theta), 1-\epsilon, 1+\epsilon)A(s,a))] \tag{6}$$

where $r(\theta)$ is the ratio of the new policy to the old policy, ϵ is the clipping parameter.

And its implementation is really straightforward, as shown in the following code:

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

actor_loss = -torch.min(ratio * adv, torch.clamp(ratio, 1 - self.epsilon, 1 + self.epsilon) * adv).mean()
```

2.3 Generalized Advantage Estimation for PPO

For GAEs, we can use the following formula:

$$A(s,a) = \sum_{t=0}^{T-1} \gamma^t (\prod_{i=1}^{t} \gamma\lambda)\delta_{t+1} \tag{7}$$

where γ is the discount factor, λ is the GAE parameter.

For simpler implementation, we can calaulate GAEs in a reverse order:

$$A(s_{t-1}, a_{t-1}) = \delta_t + \gamma\lambda A(s_t, a_t) \tag{8}$$

where $\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$.

And its implementation is shown in the following code:

```python
values = values + [next_value]
gae = 0
advantages = []

for step in reversed(range(len(rewards))):
    delta = rewards[step] + gamma * values[step + 1] * masks[step] - values[step]
    gae = delta + gamma * tau * masks[step] * gae
    advantages.insert(0, gae)
```

2.4 Collect Samples for Training

To collect samples for training, we sample actions from the policy network and store them in the memory, so that we can explore the environment and update the policy network.

And its implementation is shown in the following code:

```python
dist = self.get_dist(state)
action = dist.mean if self.is_test else dist.sample()

value = self.get_value(state)
self.states.append(state)
self.actions.append(action)
self.values.append(value)
self.log_probs.append(dist.log_prob(action))

next_state, reward, terminated, truncated, _ = self.env.step(action)

done = terminated or truncated

self.rewards.append(reward)
self.masks.append(1 - done)
```

2.5 Enforce Exploration

To enforce exploration, we can use entropy to encourage the policy to explore the environment.

And its implementation is shown in the following code:

```python
entropy_loss = -self.entropy_weight * dist.entropy().mean()
```

when the entropy of action distribution is high, the policy will explore the environment more, and the loss will be lower.

And for A2C, we further clip the log standard deviation of the action distribution into a range to ensure it explores the environment more, but also not too much.

And its implementation is shown in the following code:

```python
log_std = torch.clamp(log_std, min=self.log_std_min, max=self.log_std_max)
std = torch.exp(log_std)
dist = Normal(mean, std)
```

2.6 Weights and Biases

To record the training process, we log the training loss, training reward, and testing reward using Weights and Biases. And also for reproducibility, we also log the code, the random seed, and the hyperparameters on W&B.

The implementation is really simple, as shown in the following code:

```python
wandb.init(project="DLP-Lab7-PPO-Walker", name=args.wandb_run_name,
save_code=True, config=vars(args))

wandb.log({
    "train/step": self.total_step,
    "train/episode": episode_count,
    "train/return": score
})

wandb.log({
    "update/step": self.total_step,
    "update/actor_loss": actor_loss,
    "update/critic_loss": critic_loss
})

wandb.log({
    "test/avg_score": avg_score,
    "test/step": self.total_step
})
```

3 Analysis and Discussion

3.1 Training Curves

The training curves can be shown by the testing return curves, since the training process contains more noise.

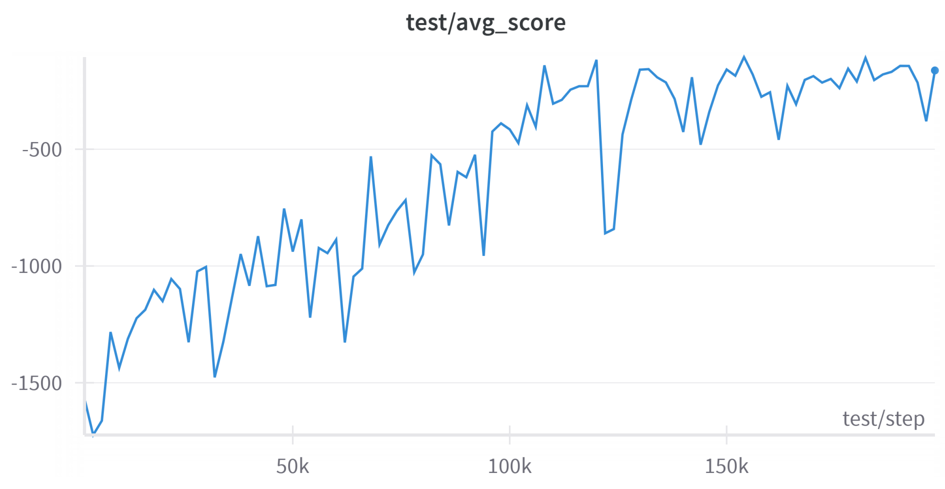

Figure 1: Testing return curve for A2C on Pendulum environment

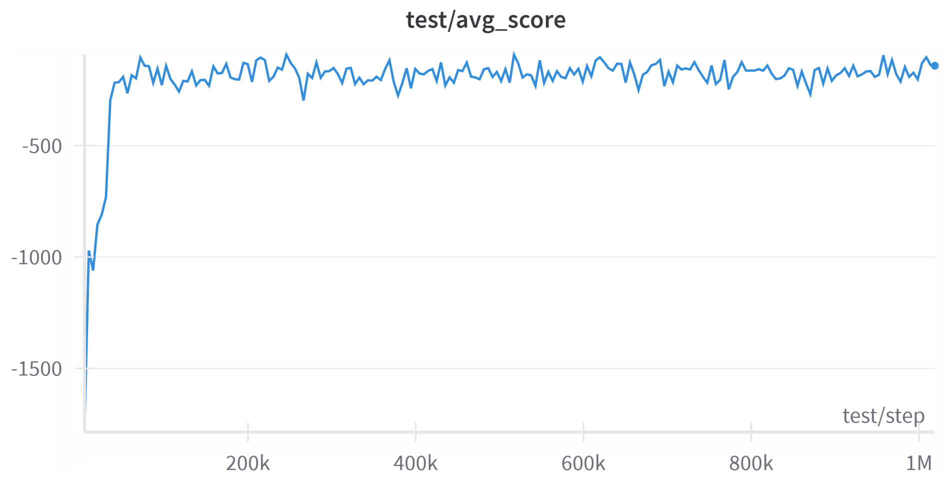

Figure 2: Testing return curve for PPO on Pendulum environment

3.2 Sample Efficiency

Sample efficiency is the number of samples required to train the model to a certain level of performance. Since A2C and PPO are both on-policy algorithms, so the sample efficiency is a critical factor to consider. From the testing return curves, we can see that PPO is much more sample efficient than A2C.

3.3 Training Stability

Training stability is the stability of the training process. From the testing return curves, we can see that PPO is more stable than A2C, thanks to its clipped surrogate objective function.

3.4 Key Parameters

3.4.1 Entropy Weight

The entropy weight is a key parameter to consider for A2C, since it dont have a self-clipped surrogate objective function.

From lots of experiments, we find that the entropy weight for A2C should be a small value, like 0.01, but can't be set to 0, otherwise the training will not converge.

As for PPO, the entropy weight to train on pendulum environment is also set to 0.01, since the agent need to explore more in the beginning of training to reach the optimal policy. But for Walker environment, the agent need to explore less, since the environment is much more complex and might need more exploitation. Thus the entropy weight is set to 0.

3.4.2 Clipping Parameter

The clipping parameter is a key parameter to consider for PPO, since it has a self-clipped surrogate objective function. We've found that the clipping parameter equal to 0.2 is really good for both environments, it can strike a good balance between exploration needed in pendulum environment and exploitation needed in walker environment.

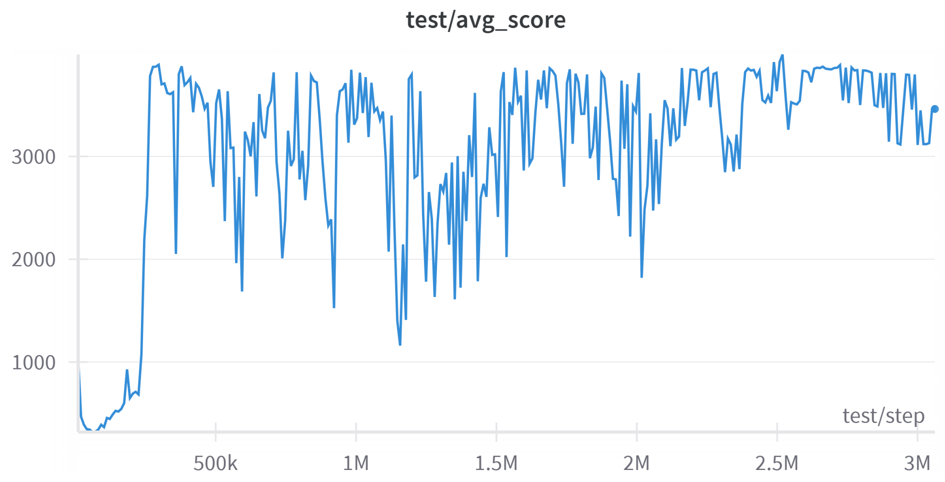

Figure 3: Testing return curve for PPO on Walker environment

3.4.3 Action Distribution

To achieve a good performance for A2C, I've tried two ways to get the action distribution.

Firstly, I've tried to use a normal distribution whose mean and variance are both depends on the state.

$$\pi(a|s) = \mathcal{N}(\mu(s), \sigma^2(s) \cdot I) \tag{9}$$

The performance of this policy on PPO is already really good, but it seems like the agent can't learn anything, this setting might be too complex for A2C to handle.

Then I've tried to use a normal distribution whose mean is depends on the state and variance is based on the action dimension, which can be shown in the following equation.

$$\pi(a|s) = \mathcal{N}(\mu(s), [\sigma_0^2, \sigma_1^2, \cdots, \sigma_{a_{dim}-1}^2]) \tag{10}$$

where a_{dim} is the dimension of the action space, and σ_i are learnable parameters.

This setting can achieve a good performance for A2C since it is more simple and stable, and it also improves the performance of PPO after using this setting.

A Appendix

A.1 Seeds

- Task1: 89, 90, 91, 92, 94, 95, 96, 97, 99, 103, 105, 107, 109, 114, 115, 121, 123, 124, 128, 129
- Task2: 89, 90, 91, 92, 94, 95, 96, 97, 99, 103, 105, 107, 109, 114, 115, 116, 121, 122, 123, 124
- Task3: 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101, 102, 103, 104, 105, 106, 107, 108