Report

Introduction

This is the final lab aimed at implementing Advantage Actor Critic (A2C) and Proximal Policy Optimization (PPO), the A2C is used in pendulum environment while the PPO is used in both pendulum and walker environments. Both of the architecture use actor network and critic network, but the critic network here is used to estimate the state value which is required in calculating the advantage value for training more stable. As for my implementation, I find out that the A2C is more difficult to train than PPO since I encounter several issues that will be discussed later.

The commands of inference of these three codes are listed below, note that the fixed random seeds have been hard-coded in the testing function.
For task1:
‘python3 a2c_pendulum.py –test –model-path ./LAB7_313552052_task1_a2c_pendulum.pt’

For task2:
‘python3 ppo_pendulum.py –test –model-path ./LAB7_313552052_task2_ppo_pendulum.pt’

For task3:
‘python3 ppo_walker.py –test –model-path ./LAB7_313552052_task3_ppo_1m.pt’
‘python3 ppo_walker.py –test –model-path ./LAB7_313552052_task3_ppo_1p5m.pt’
‘python3 ppo_walker.py –test –model-path ./LAB7_313552052_task3_ppo_2m.pt’
‘python3 ppo_walker.py –test –model-path ./LAB7_313552052_task3_ppo_2p5m.pt’
‘python3 ppo_walker.py –test –model-path ./LAB7_313552052_task3_ppo_3m.pt’

Implementation

1. How do you obtain the stochastic policy gradient and the TD error for A2C?

   For policy gradient, the policy parameters are updated in this direction.

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(a|s) \cdot A(s,a)]$$

In code implementation, first we have to get the action distribution of the current state through the actor network, then calculate the log probability of the action we take in this state, finally compute the policy loss with the log probability multiplied by the advantage value. Here is the code segment.

```
# Normalize advantages for stability
if advantages.shape[0] > 1:  # Only if batch size > 1
    advantages = (advantages - advantages.mean()) / (advantages.std() + 1e-8)

_, dist = self.actor(states_norm)
log_probs = dist.log_prob(actions).sum(dim=-1, keepdim=True)

# Make sure log_probs has correct shape
if len(log_probs.shape) == 1:
    log_probs = log_probs.unsqueeze(1)
policy_loss = -(log_probs * advantages).mean()

# Entropy
entropy = dist.entropy().mean()
policy_loss = policy_loss - self.entropy_weight * entropy
```

Besides, for avoiding trapping into local optima, the entropy term is also considered here for exploration purpose.

For the TD error, it's the same as what we do in the DQN lab, first use the current state and next state to estimate the Q-value respectively through critic network, then calculate the target Q-value by the Bellman equation,

$$target = r + gamma * next\ Q\ value$$

Finally, the subtract the target Q-value from estimated current Q-value with the mean square error to get the critic loss. There is an advanced technique used here called advantage value which indicate how good an action is compared to the average Q-value of the state. The advantage calculation in A2C is simpler, just subtract the target Q-value from current Q-value, which is similar to the TD error. There is the code segment.

```
# Compute target values
target_values = rewards + self.gamma * next_values * masks

# Compute value loss
value_loss = F.mse_loss(values, target_values.detach())

# Compute advantages
advantages = (target_values.detach() - values.detach())
```

2. How do you implement the clipped objective in PPO?

PPO has a special technique that ensures the update within a trust region, which is achieved by narrowing the policy gradient in a range to avoid aggressive update for stability. The range is determined by a hyperparameter ε , the probability ratio between new and old policy below 1-ε or above 1+ε will be clipped to the boundary respectively. It has an excellent effect in preventing too large increase or decrease in action probability. Here is the code segment.

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

# actor_loss
############TODO#############
# actor_loss = ?
# PPO clip objective: min(ratio * advantage, clip(ratio, 1-epsilon, 1+e
# We use negative because we're minimizing the loss (and want to maximi
actor_loss = -torch.min(
    ratio * adv,
    torch.clamp(ratio, 1.0 - self.epsilon, 1.0 + self.epsilon) * adv
).mean()
# Add entropy regularization to encourage exploration
entropy = dist.entropy().mean()
actor_loss -= self.entropy_weight * entropy
```

After the actor network calculated the action distribution, we can get the specific action log probability denoted as log_prob, then calculate the ratio between log_prob and old log probability, finally, when estimating actor loss, we have to consider whether the ratio is out of bound, if it does, clip the ratio first then multiplied by the advantage value. Same to the A2C implementation, the entropy term is applied here for exploration purpose.

3. How do you obtain the estimator of GAE?

The implementation of GAE is a little different from the traditional GAE which is defined as

$$\hat{A}_t = \sum_{i=0}^{\infty} (\gamma\lambda)^i \delta_{t+i}$$

Where the δ is defined as

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

But in my implementation, the GAE is implemented in a simpler version called GAE return for training value function. The main difference is instead of returning the advantage value, the GAE return produces the value

function target, which is used for training the critic network. In code implementation, the calculation of delta (TD error) is the same as the traditional GAE, but it returns the GAE + current state value, as shown below.

```python
def compute_gae(
    next_value: list, rewards: list, masks: list, values: list, gamma: float, tau: float) -> List:
    """Compute gae."""

    ###########TODO############
    # Initialize storage
    values = values + [next_value]
    gae = 0
    gae_returns = []

    # Calculate returns backwards
    for step in reversed(range(len(rewards))):
        # Calculate TD error: r_t + gamma * V(s_{t+1}) - V(s_t)
        delta = rewards[step] + gamma * values[step + 1] * masks[step] - values[step]

        # GAE calculation: sum_i (gamma * lambda)^i * delta_{t+i}
        gae = delta + gamma * tau * masks[step] * gae

        # Prepend to get the returns in the correct order
        gae_returns.insert(0, gae + values[step])
    ############################
    return gae_returns
```

Note that instead of returning GAE directly, it returns the GAE + current state value for training the critic network.

4. How do you collect samples from the environment?

In all my implementations, the experiences are stored with buffer for training stability, the state, action, reward, next state, done are stored separately, and only when the buffer size is greater than n_steps (an hyperparameter), the model will be updated. Instead of updating the model at every environment step which may cause unstable issue, the implementation only updates the model after collecting a number of experiences, and the buffer will be cleared out once the updating complete. In A2C, the code segment will look like this.

```python
# Perform mini-batch updates when enough transitions are collected
if self.buffer_size >= self.n_steps:
    # Update model with batch
    actor_loss, critic_loss = self.update_model_batch()
```

It checks whether the current buffer size is greater than the n steps or not to update the model.

In PPO, the code segment looks like this.

```
# Process collected trajectory in batches during update
actor_loss, critic_loss = self.update_model(next_state)
actor_losses.append(actor_loss)
critic_losses.append(critic_loss)
wandb.log({
    "env_steps": self.total_step,
    "Critic loss": critic_loss,
    "Actor loss": actor_loss,
})
```

Instead of collecting a specific number of experiences, it only updates the model after the number of rollout lengths, which is the same way as A2C implementation.

5. How do you enforce exploration (despite that A2C and PPO are on-policy RL methods)?

   The exploration term lies in the actor loss in both A2C and PPO implementation, it's achieved by introducing the entropy term, the higher entropy means the more exploration, this will be added into the actor loss with entropy weight. The code segment is below.

```
# Add entropy regularization to encourage exploration
entropy = dist.entropy().mean()
actor_loss -= self.entropy_weight * entropy
```

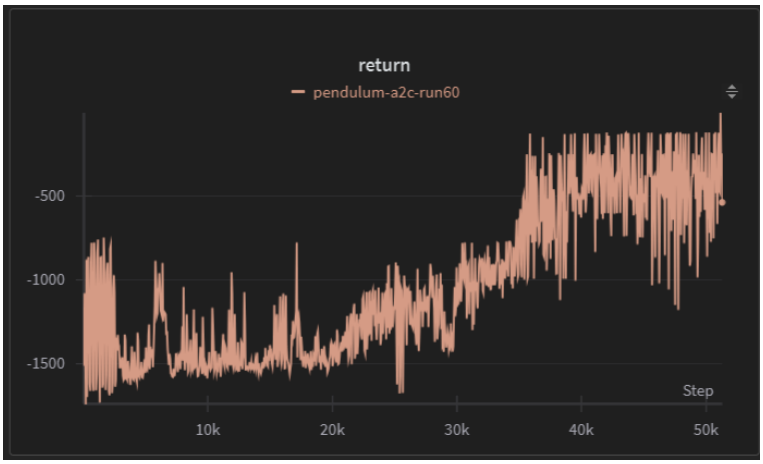
   As you can see, the entropy is calculated via the action distribution, and the exploration is controlled by the entropy weight, the higher weight means it encourages more exploration to avoid trapping into local optima.

6. Explain how you use Weight & Bias to track model performance and the loss values (including actor loss, critic loss and the entropy).

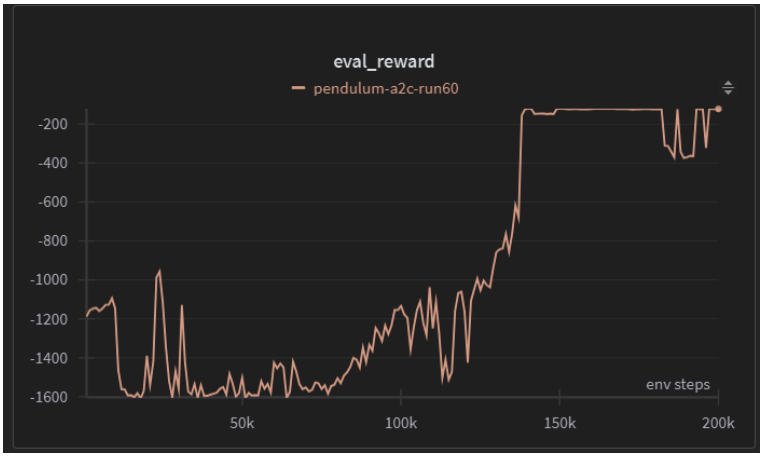
   In my implementation, there are several metrics tracked via W&B such as environment step, actor loss, critic loss, episode reward, evaluation reward, and other information for auxiliary like Q value mean and Q value standard deviation.

Analysis and discussion

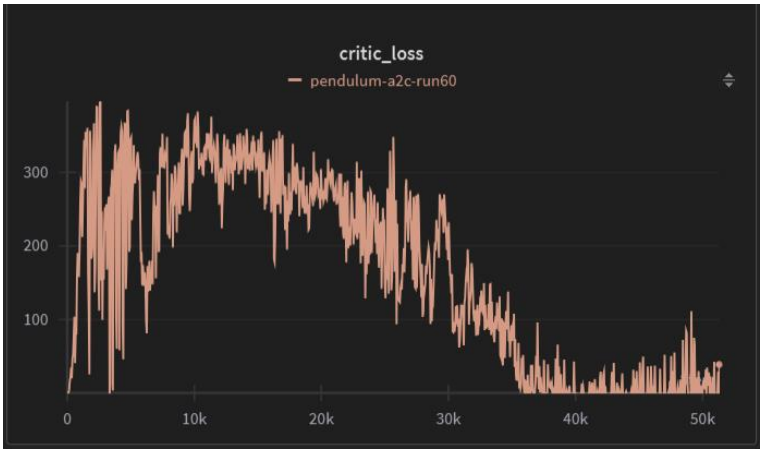
1. Plot the training curves for task1, task2 and task3.
   Task1

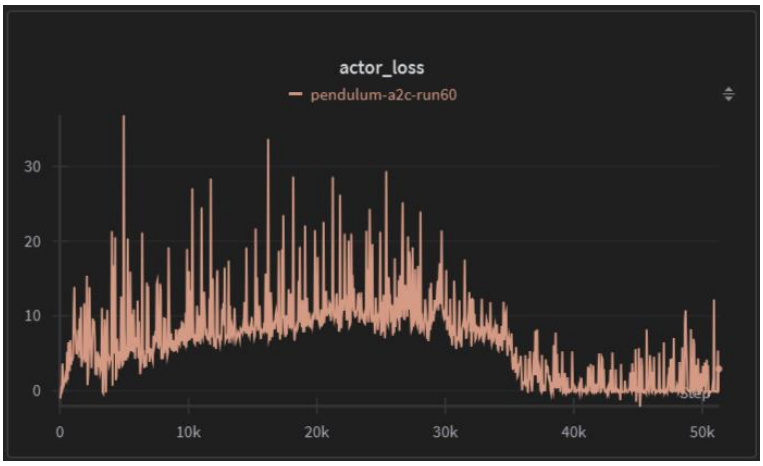


Here is the evaluation reward with respect to environment steps.



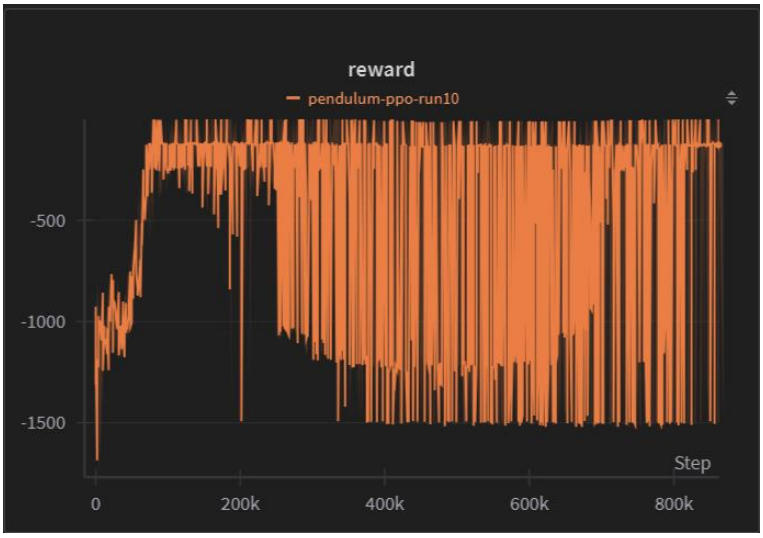
The critic loss and actor loss.



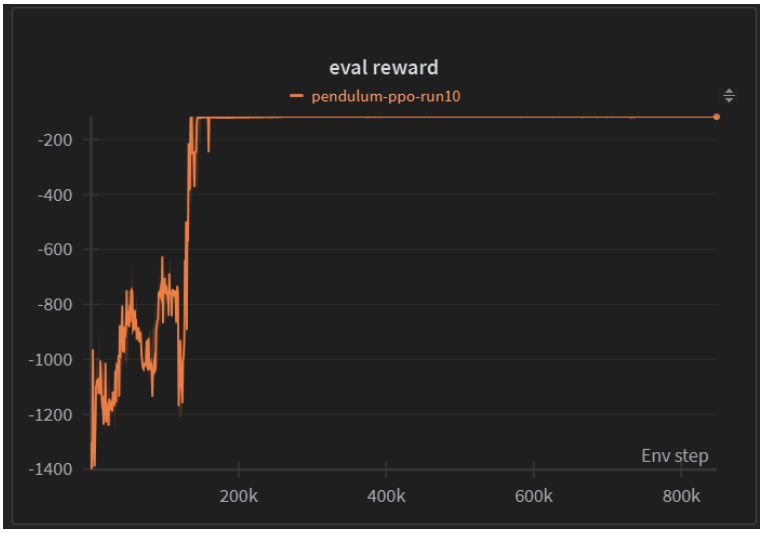


It shows that the network converges successfully.

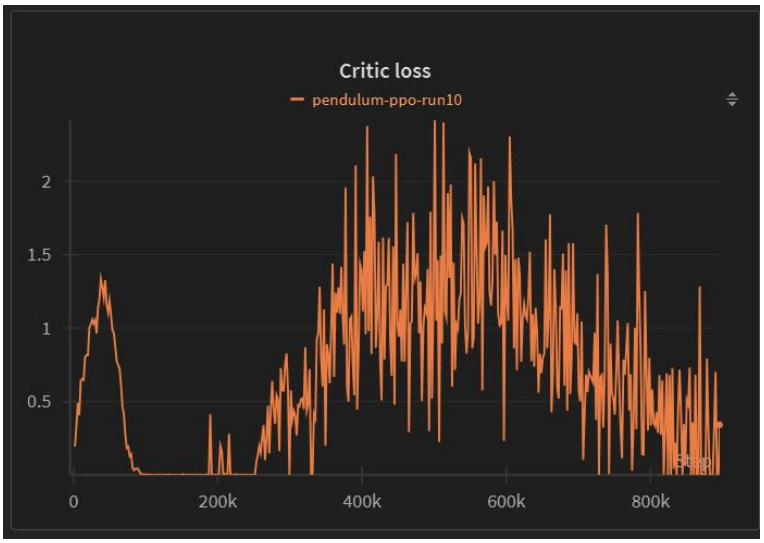
Task2

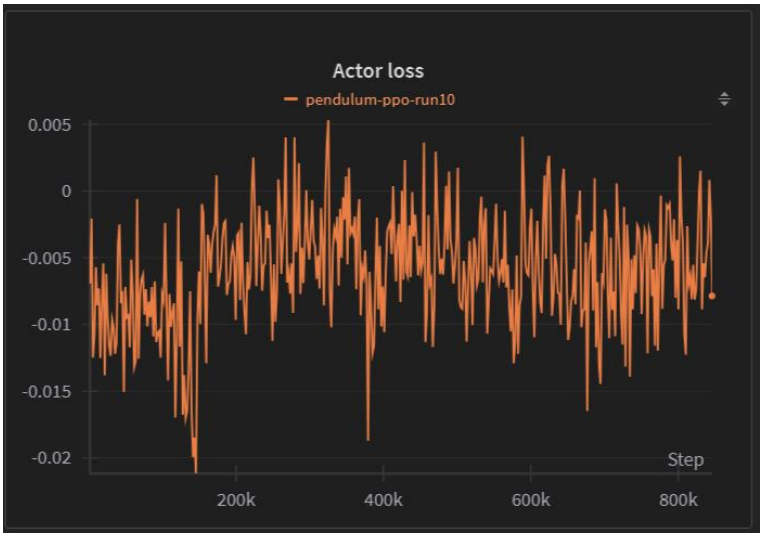


Here is the evaluation reward with respect to the environment step.

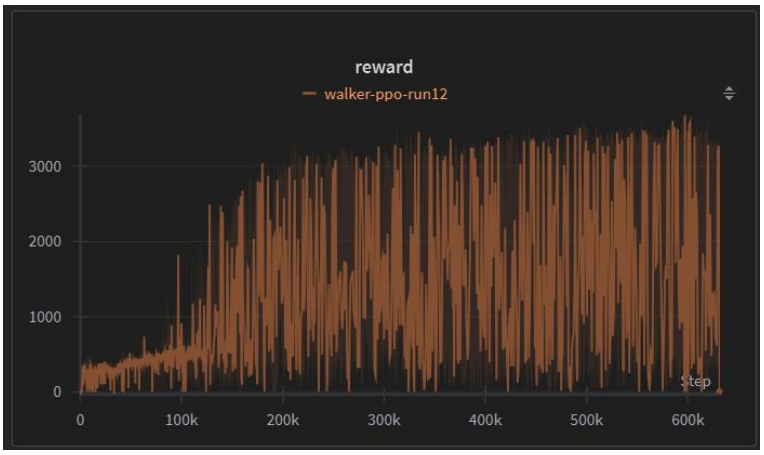


The critic loss and the actor loss.

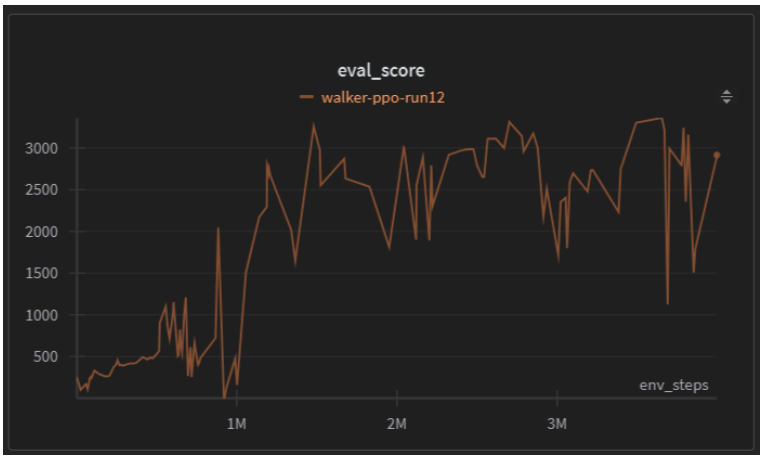




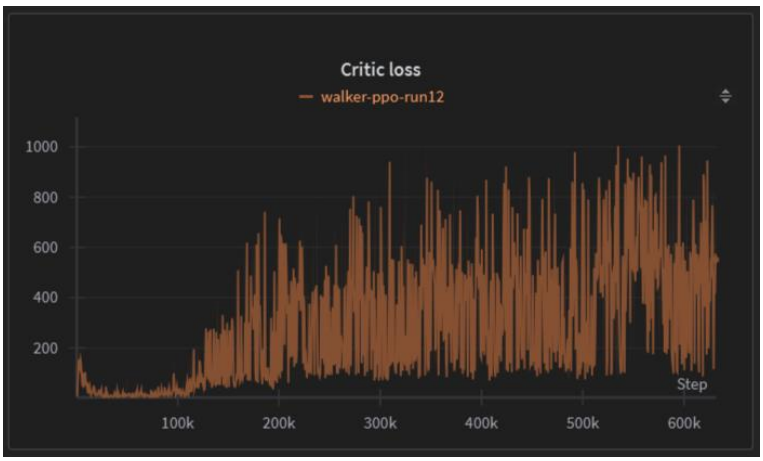
Task3

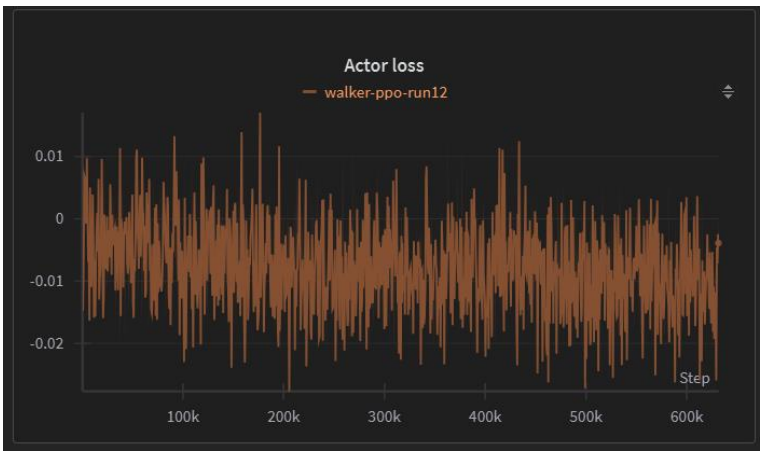


Here is the evaluation reward with respect to environment step.



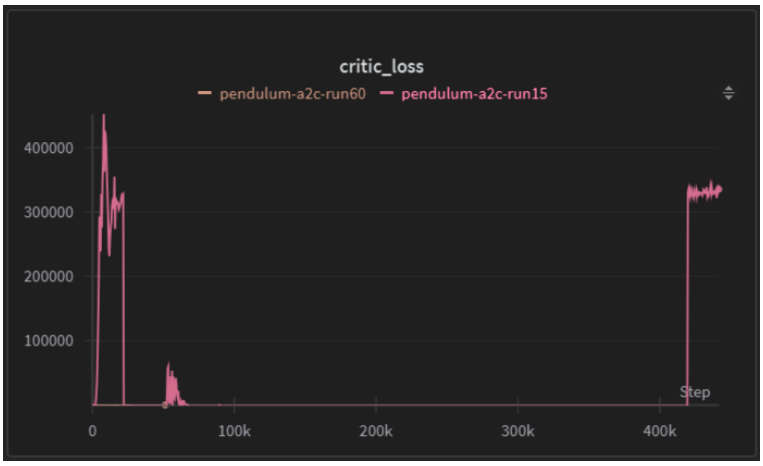
The critic loss and actor loss.



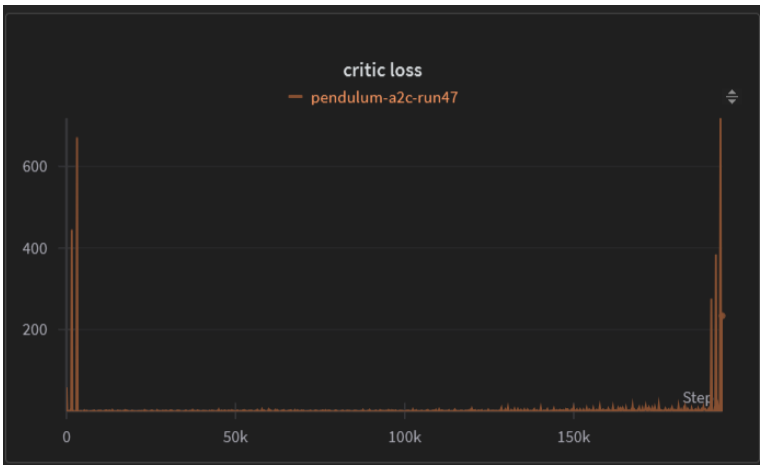


Though the model seems not converging yet, it still achieves 2500+ reward with 1.5M environment step.

2. Compare the sample efficiency and training stability of A2C and PPO.
   In general, A2C is more unstable than PPO. As my implementation, A2C can easily encounter gradient exploding or vanishing, like the illustration here.

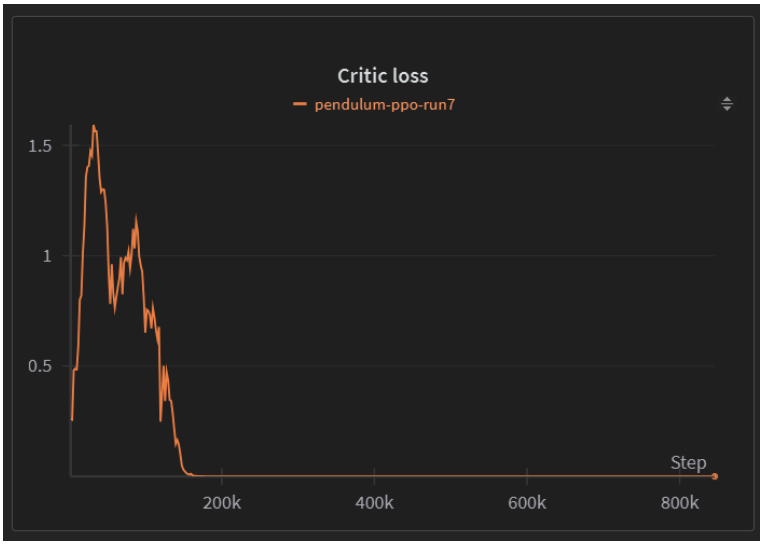


The loss value may surge suddenly.



Or the loss value may converge at very beginning, both cases make A2C hard to train, these show the critic network learns in an inefficient way. Besides, since the actor is learned from the critic network, if the critic doesn't learn anything, the actor will perform poorly as well.

On the other hand, the PPO network shows more stable training.



The PPO architecture shows more stable training, it can converge easily. There are some reasons that why PPO is more stable than A2C.

- Trust region enforcement
  PPO network use a technique called trust region, which ensures the update won't deviate from the old policy too far to increase the stability of training. A2C has no mechanism to prevent harmful large policy steps, allowing updates of any magnitude.
- Sample reuse with safety
  PPO reuses the same trajectory data for multiple update epochs while protecting against overoptimization through clipping. This extracts more learning from each sample while maintaining policy stability. In A2C, if it reuses samples, can rapidly diverge without protective mechanisms.
- GAE usage
  PPO uses GAE estimation to calculate the advantage value, it produces more temporally consistent advantage estimates by exponentially weighting TD errors. In general, the combination of GAE with PPO's clipping mechanism creates a highly stable, sample-efficient algorithm

3. Perform an empirical study on the key parameters, such as clipping parameter and entropy coefficient.

   In terms of clipping parameter, the higher clipping parameter means the wider range the network can update, this will cause faster convergence but destabilize the training process, on the contrary, the lower clipping parameter the slower the convergence, but it introduces more stable training. As a result, this issue is a kind of trade-off between convergence speed and stability, for the most of the work, the clipping parameter often sets to 0.2 to provide the best balance for most environments.

   In terms of the entropy coefficient, the higher the entropy coefficient the more exploration it encourages, this will help not to trap into the local optima, but taking more time to converge is the by-product. Except for using the same value of entropy coefficient during the training process, the decay schedule of entropy coefficient usually introduces to use higher coefficient at the very beginning of the training to explore more about the states, and decay the coefficient through time, the later episodes can use a lower coefficient to utilize the exploitation.

   Parameter interaction analysis:

Generally, the higher clipping parameter requires higher entropy coefficient for stability, on the other hand, the higher learning rate needs more conservative clipping parameter to avoid updating too aggressively. In my observation, the complex environments (e.g. walker) benefit from lower clipping and higher entropy coefficient.

Additional analysis on other training strategy

Since A2C is more unstable than PPO, most of the training strategies are applied to A2C implementation to try to stabilize the training.

- Calculate running mean and standard deviation:
  This is a class which implements online normalization of observations, it maintains running statistics (mean and variance) without storing all previous data, making it memory-efficient. It dramatically improves training stability by ensuring inputs to neural networks remain in a consistent range. Besides, the critic network learns more efficiently with normalized inputs to avoid gradient exploding I mentioned above. Without normalization, different observation scales cause gradient updates to vary wildly in magnitude.
- Use target network:
  I try to add another critic network works as target network to stabilize the training. A2C uses the same critic network to compute both current Q-value and target Q-value, this creates a harmful feedback loop where the target constantly shifts during learning, if adding another target critic network, the target Q-value changes more slowly and prevents rapid oscillations in value estimates. This shows that advantage estimation can have high variance due to unstable value predictions, so the more stable advantage estimation of the target critic network lead to more consistent policy updates
- Different activation function:
  Since there are different activation functions used in A2C architecture, such as ReLU, Tanh, ELU. ReLU function is the easiest one, but due to the environment we used here (e.g. pendulum), the state, action and the reward usually have negative values, this makes ReLU function inefficient here, the network may have zero gradient easily. To solve the problem of ReLU, Exponential Linear Unit (ELU) may be a good solution, it preserves negative information unlike ReLU, smoother

gradients near zero compared to ReLU, but it has more computation. However, it still has saturation if the model input is too negative. As a result, the final activation function I use is tanh which output range is bounded within [-1, 1], the most important property is that it's differentiable across entire range, no discontinuities, this can solve the issue which ReLU and ELU encounter. Some modification also used here to map output range to bounded action spaces like Pendulum.

- Different weight initializing:
  The initial weight initialization is uniform initialization, the init_w parameter can easily set up the controlled scale of the weight, another initialization method is orthogonal initialization, it creates weights where W^TW = identity matrix and helps maintain consistent gradient flow. Besides, it can provide more stable value estimation from the start and usually faster initial learning and better final performance.
- Buffering the experience:
  Instead of storing one transition in A2C, there are several lists to store state, action, reward, next state and done for every step. Once collecting a number of experiences, the model is updated. In doing so, the model won't deviate too much since updating at every environment step is too aggressive. It uses a similar approach in PPO architecture which updating the model after collecting a number of rollout length experiences. This method stabilizes the model, and the evaluation reward progress successfully.
- Warm-up in early training:
  In early episode, use a specially assign initial state, that is, the pendulum shows up at the top right corner, to make the model easily learn the process since it can get a large reward at the beginning and has faster training speed. Like the code segment shows below.

```
# Special for pendulum: set initial pendulum position upright with small noise
# This makes learning much faster (curriculum learning)
if ep < 100:  # First 100 episodes start near upright
    self.env.unwrapped.state = np.array([np.pi + np.random.normal(0, 0.1), 0])
```