Lab7 313605019 方敏

1. Introduction

This lab explores two policy-based reinforcement learning algorithms: Advantage Actor Critic (A2C) and Proximal Policy Optimization (PPO). The main objective is to evaluate how these algorithms perform in continuous control tasks, in this lab we use Pendulum-v1 environment and MuJoCo Walker2d-v4 to evaluate.

Throughout the implementation, we need to complete A2C and PPO key components such as the actor-critic neural networks, Generalized Advantage Estimation (GAE), and the PPO clipping objective. The experimental results show that compared to A2C, PPO with clipping and GAE significantly improves training stability and final performance.

2. Implementation

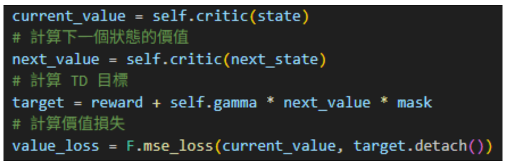
- How do you obtain stochastic policy gradient and TD error for A2C?

TD Error:

The temporal difference (TD) error is used to measure how good the critic's value estimate is. It is defined as: $\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$

This error indicates how much the current value prediction deviates from the bootstrapped target.

My implementation:



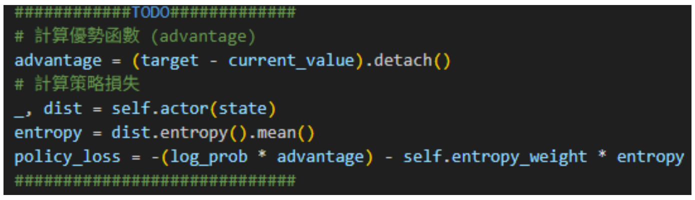
```
current_value = self.critic(state)
# 計算下一個狀態的價值
next_value = self.critic(next_state)
# 計算 TD 目標
target = reward + self.gamma * next_value * mask
# 計算價值損失
value_loss = F.mse_loss(current_value, target.detach())
```

Stochastic Policy Gradient:

The stochastic policy gradient is computed by taking the gradient of the log-probability of the selected action with respect to the policy parameters, scaled by the advantage (TD error): $\nabla_\theta J(\theta) = \mathbb{E}\left[\nabla_\theta \log \pi_\theta(a_t|s_t) \cdot \delta_t\right]$

This guides the policy to assign higher probability to actions that lead to higher returns.

My implementation:



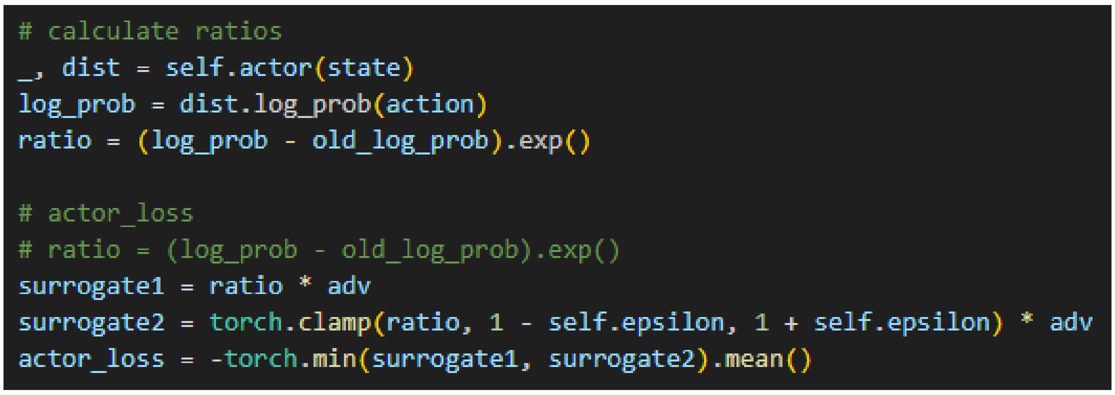
```
############TODO#############
# 計算優勢函數 (advantage)
advantage = (target - current_value).detach()
# 計算策略損失
_, dist = self.actor(state)
entropy = dist.entropy().mean()
policy_loss = -(log_prob * advantage) - self.entropy_weight * entropy
##############################
```

- How do you implement the clipped objective in PPO?

  In Proximal Policy Optimization (PPO), a clipping mechanism is introduced to prevent the updated policy from deviating too much from the old policy, which can lead to unstable during training.

  The clipped surrogate objective is defined as: $L^{CLIP}(\theta) = \mathbb{E}_t \left[\min\left(r_t(\theta)A_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)A_t\right)\right]$

  In my implementation, I calculate the policy ratio using the current and previous log probabilities, then apply clipping to ensure stable updates. The clipped surrogate loss is computed by taking the minimum between the unclipped and clipped objectives, following the PPO formulation.



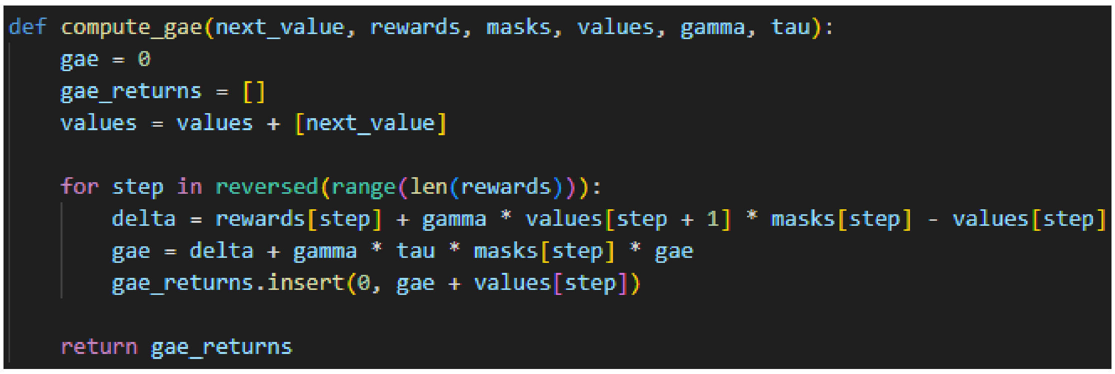
```
# calculate ratios
_, dist = self.actor(state)
log_prob = dist.log_prob(action)
ratio = (log_prob - old_log_prob).exp()

# actor_loss
# ratio = (log_prob - old_log_prob).exp()
surrogate1 = ratio * adv
surrogate2 = torch.clamp(ratio, 1 - self.epsilon, 1 + self.epsilon) * adv
actor_loss = -torch.min(surrogate1, surrogate2).mean()
```

- How do you obtain the estimator of GAE?

  GAE (Generalized Advantage Estimation) is a technique used to compute advantage values that balances bias and variance. It accumulates the temporal difference (TD) errors over future time steps using both the discount factor γ and the smoothing factor λ (tau in the code).

  The formula is: $A_t^{GAE(\gamma,\lambda)} = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l}$

  In my implementation, I iterate backward over the trajectory to accumulate the TD errors using the formula for GAE. The result combines both the advantage and the value function, which is then used to update the policy.

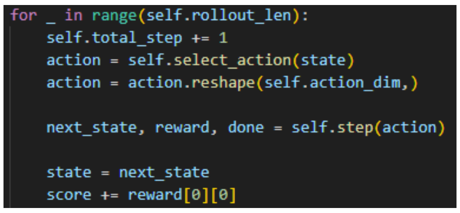


```
def compute_gae(next_value, rewards, masks, values, gamma, tau):
    gae = 0
    gae_returns = []
    values = values + [next_value]

    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
        gae_returns.insert(0, gae + values[step])

    return gae_returns
```

- How do you collect samples from the environment?

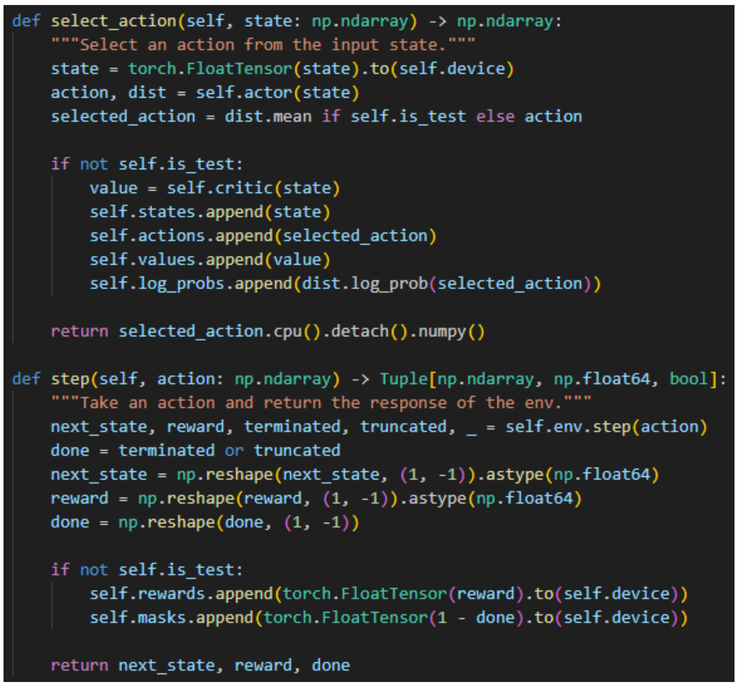
  In PPO implementation, I collect samples by interacting with the environment step-by-step. Each step the agent observes the current state, selects an action using the Actor network, and then executes the action in the environment to receive the next_state, reward, and done signal. All these values that along with value estimates and log-probabilities are stored in temporary memory buffers (self.states, self.actions, self.rewards, self.values, self.masks, self.log_probs) until a full rollout of steps (I set it as 2048) is collected. These collected samples are later used to update the model.

  Train:



```python
for _ in range(self.rollout_len):
    self.total_step += 1
    action = self.select_action(state)
    action = action.reshape(self.action_dim,)

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

    state = next_state
    score += reward[0][0]
```

  Select_action and Step:



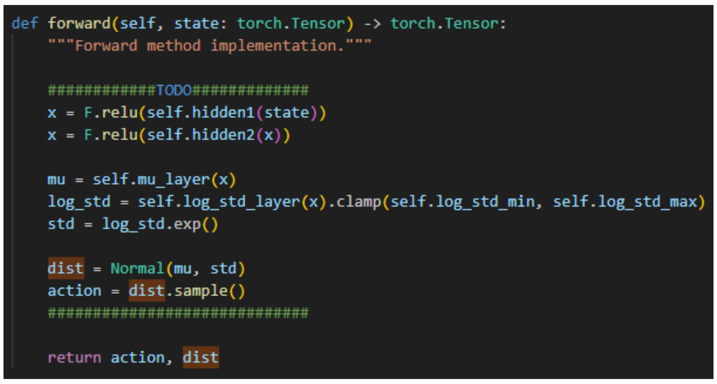
```python
def select_action(self, state: np.ndarray) -> np.ndarray:
    """Select an action from the input state."""
    state = torch.FloatTensor(state).to(self.device)
    action, dist = self.actor(state)
    selected_action = dist.mean if self.is_test else action

    if not self.is_test:
        value = self.critic(state)
        self.states.append(state)
        self.actions.append(selected_action)
        self.values.append(value)
        self.log_probs.append(dist.log_prob(selected_action))

    return selected_action.cpu().detach().numpy()

def step(self, action: np.ndarray) -> Tuple[np.ndarray, np.float64, bool]:
    """Take an action and return the response of the env."""
    next_state, reward, terminated, truncated, _ = self.env.step(action)
    done = terminated or truncated
    next_state = np.reshape(next_state, (1, -1)).astype(np.float64)
    reward = np.reshape(reward, (1, -1)).astype(np.float64)
    done = np.reshape(done, (1, -1))

    if not self.is_test:
        self.rewards.append(torch.FloatTensor(reward).to(self.device))
        self.masks.append(torch.FloatTensor(1 - done).to(self.device))

    return next_state, reward, done
```

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

  In both A2C and PPO, I enforce exploration by using a stochastic policy during training. This is done by sampling actions from a probability distribution rather than choosing deterministic actions. The actor network outputs a mean (mu) and standard deviation (std) for a Gaussian distribution, and actions are sampled from this distribution.

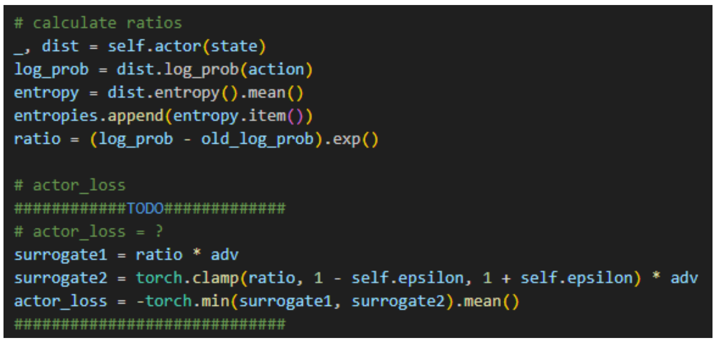
In the implementation, exploration is encouraged via entropy regularization in the loss function. Higher entropy leads to more randomness in action selection, which promotes exploration. The entropy term is added to the actor loss with a small weight to balance exploration and exploitation.

Function forward in actor:



```python
def forward(self, state: torch.Tensor) -> torch.Tensor:
    """Forward method implementation."""

    ############TODO#############
    x = F.relu(self.hidden1(state))
    x = F.relu(self.hidden2(x))

    mu = self.mu_layer(x)
    log_std = self.log_std_layer(x).clamp(self.log_std_min, self.log_std_max)
    std = log_std.exp()

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

    return action, dist
```

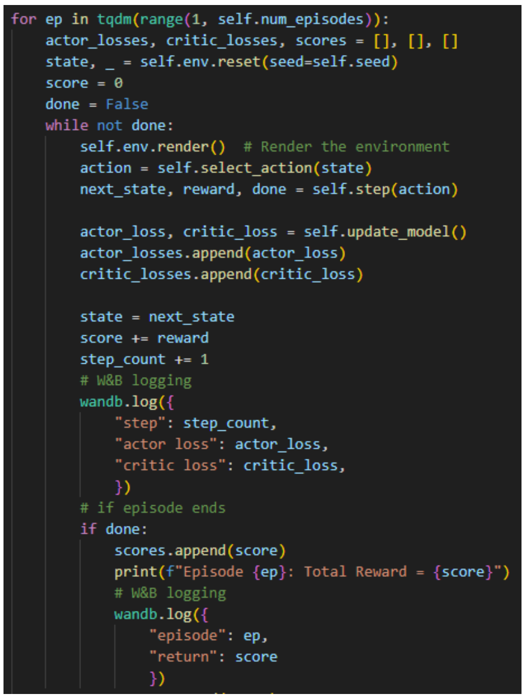
Update:



```python
# calculate ratios
_, dist = self.actor(state)
log_prob = dist.log_prob(action)
entropy = dist.entropy().mean()
entropies.append(entropy.item())
ratio = (log_prob - old_log_prob).exp()

# actor_loss
############TODO#############
# actor_loss = ?
surrogate1 = ratio * adv
surrogate2 = torch.clamp(ratio, 1 - self.epsilon, 1 + self.epsilon) * adv
actor_loss = -torch.min(surrogate1, surrogate2).mean()
#############################
```

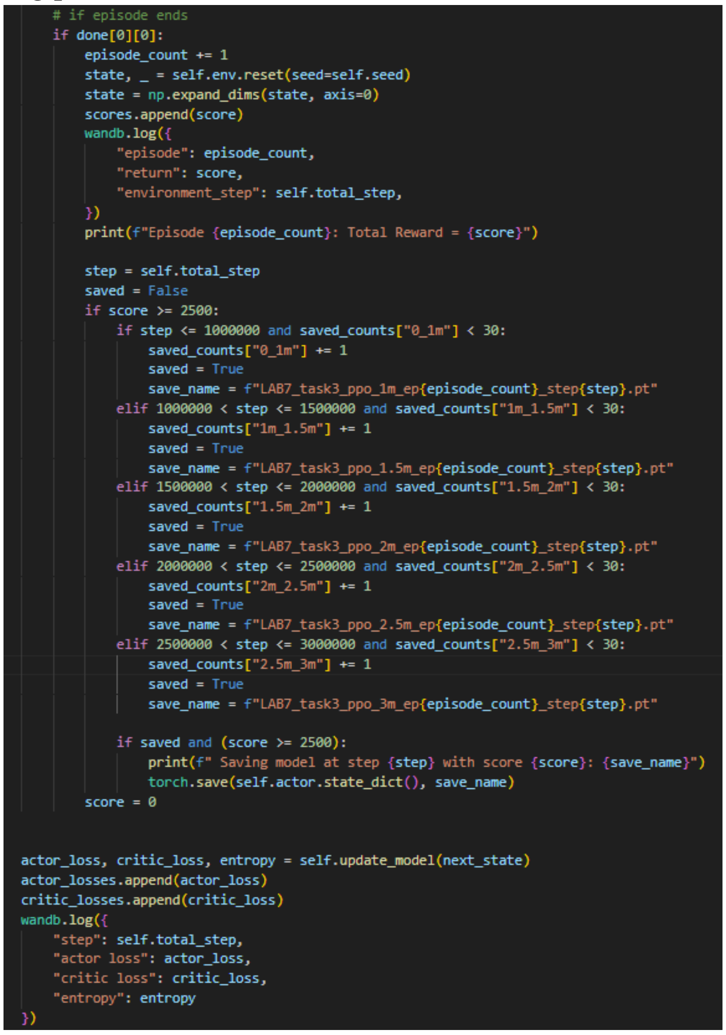
- Explain how you use Weight \& Bias to track model performance and the loss values (including actor loss, critic loss, and the entropy)

  In A2C, I did not change the wandb code TA gave.



```python
for ep in tqdm(range(1, self.num_episodes)):
    actor_losses, critic_losses, scores = [], [], []
    state, _ = self.env.reset(seed=self.seed)
    score = 0
    done = False
    while not done:
        self.env.render()  # Render the environment
        action = self.select_action(state)
        next_state, reward, done = self.step(action)

        actor_loss, critic_loss = self.update_model()
        actor_losses.append(actor_loss)
        critic_losses.append(critic_loss)

        state = next_state
        score += reward
        step_count += 1
        # W&B logging
        wandb.log({
            "step": step_count,
            "actor loss": actor_loss,
            "critic loss": critic_loss,
            })
        # if episode ends
        if done:
            scores.append(score)
            print(f"Episode {ep}: Total Reward = {score}")
            # W&B logging
            wandb.log({
                "episode": ep,
                "return": score
                })
```

In PPO, I record the reward at the end of each episode, and after every rollout, I record the actor loss, critic loss, and entropy to Weights & Biases. This allows me to effectively track the training performance and stability throughout the learning process.



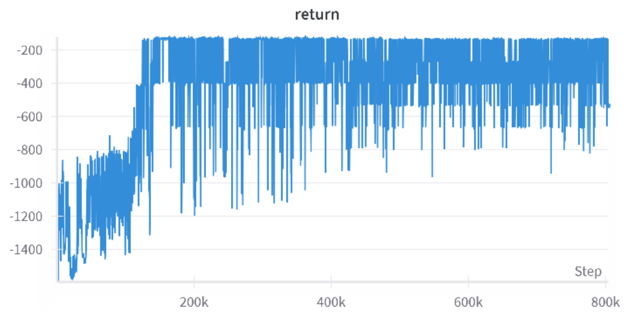
```python
    # if episode ends
    if done[0][0]:
        episode_count += 1
        state, _ = self.env.reset(seed=self.seed)
        state = np.expand_dims(state, axis=0)
        scores.append(score)
        wandb.log({
            "episode": episode_count,
            "return": score,
            "environment_step": self.total_step,
        })
        print(f"Episode {episode_count}: Total Reward = {score}")

        step = self.total_step
        saved = False
        if score >= 2500:
            if step <= 1000000 and saved_counts["0_1m"] < 30:
                saved_counts["0_1m"] += 1
                saved = True
                save_name = f"LAB7_task3_ppo_1m_ep{episode_count}_step{step}.pt"
            elif 1000000 < step <= 1500000 and saved_counts["1m_1.5m"] < 30:
                saved_counts["1m_1.5m"] += 1
                saved = True
                save_name = f"LAB7_task3_ppo_1.5m_ep{episode_count}_step{step}.pt"
            elif 1500000 < step <= 2000000 and saved_counts["1.5m_2m"] < 30:
                saved_counts["1.5m_2m"] += 1
                saved = True
                save_name = f"LAB7_task3_ppo_2m_ep{episode_count}_step{step}.pt"
            elif 2000000 < step <= 2500000 and saved_counts["2m_2.5m"] < 30:
                saved_counts["2m_2.5m"] += 1
                saved = True
                save_name = f"LAB7_task3_ppo_2.5m_ep{episode_count}_step{step}.pt"
            elif 2500000 < step <= 3000000 and saved_counts["2.5m_3m"] < 30:
                saved_counts["2.5m_3m"] += 1
                saved = True
                save_name = f"LAB7_task3_ppo_3m_ep{episode_count}_step{step}.pt"

            if saved and (score >= 2500):
                print(f" Saving model at step {step} with score {score}: {save_name}")
                torch.save(self.actor.state_dict(), save_name)
        score = 0


actor_loss, critic_loss, entropy = self.update_model(next_state)
actor_losses.append(actor_loss)
critic_losses.append(critic_loss)
wandb.log({
    "step": self.total_step,
    "actor loss": actor_loss,
    "critic loss": critic_loss,
    "entropy": entropy
})
```

3. Analysis and discussions

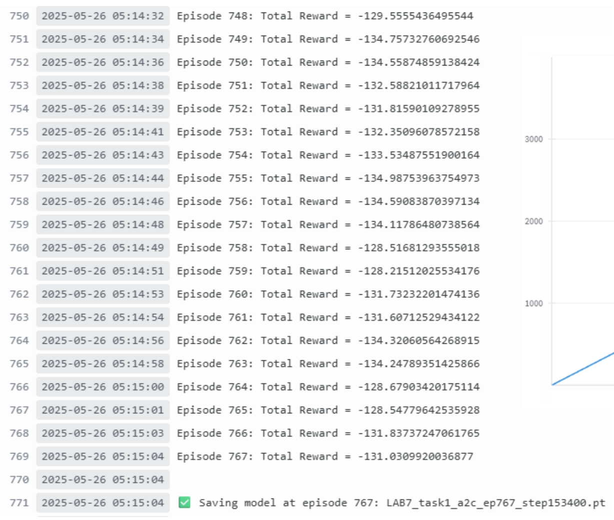
- Plot the training curves Task 1, Task 2, and Task 3 separately.

  Task1

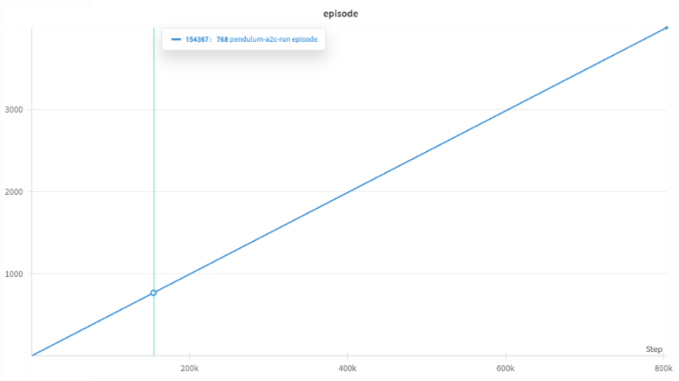
  (Since A2C training is relatively unstable, I chose to save the model only when the average reward exceeds -150 for 20 consecutive episodes, ensuring that only sufficiently stable policies are stored.)



The model I saved is in episode = 767, step < 200K



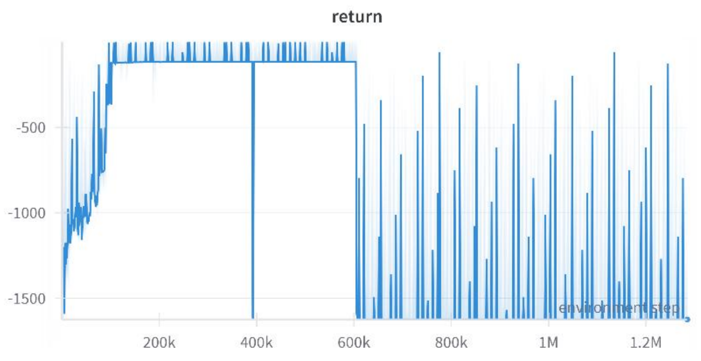
```
2025-05-26 05:14:32  Episode 748: Total Reward = -129.5555436495544
2025-05-26 05:14:34  Episode 749: Total Reward = -134.75732760692546
2025-05-26 05:14:36  Episode 750: Total Reward = -134.55874859138424
2025-05-26 05:14:38  Episode 751: Total Reward = -132.58821011717964
2025-05-26 05:14:39  Episode 752: Total Reward = -131.81590109278955
2025-05-26 05:14:41  Episode 753: Total Reward = -132.35096078572158
2025-05-26 05:14:43  Episode 754: Total Reward = -133.53487551900164
2025-05-26 05:14:44  Episode 755: Total Reward = -134.98753963754973
2025-05-26 05:14:46  Episode 756: Total Reward = -134.59083870397134
2025-05-26 05:14:48  Episode 757: Total Reward = -134.11786480738564
2025-05-26 05:14:49  Episode 758: Total Reward = -128.51681293555018
2025-05-26 05:14:51  Episode 759: Total Reward = -128.21512025534176
2025-05-26 05:14:53  Episode 760: Total Reward = -131.73232201474136
2025-05-26 05:14:54  Episode 761: Total Reward = -131.60712529434122
2025-05-26 05:14:56  Episode 762: Total Reward = -134.32060564268915
2025-05-26 05:14:58  Episode 763: Total Reward = -134.24789351425866
2025-05-26 05:15:00  Episode 764: Total Reward = -128.67903420175114
2025-05-26 05:15:01  Episode 765: Total Reward = -128.54779642535928
2025-05-26 05:15:03  Episode 766: Total Reward = -131.83737247061765
2025-05-26 05:15:04  Episode 767: Total Reward = -131.0309920036877
2025-05-26 05:15:04
2025-05-26 05:15:04  ✅ Saving model at episode 767: LAB7_task1_a2c_ep767_step153400.pt
```



Command:

--actor-lr 6e-4 --critic-lr 3e-3 --discount-factor 0.9 --num-episodes 4000 --seed 77 --entropy-weight 0.015

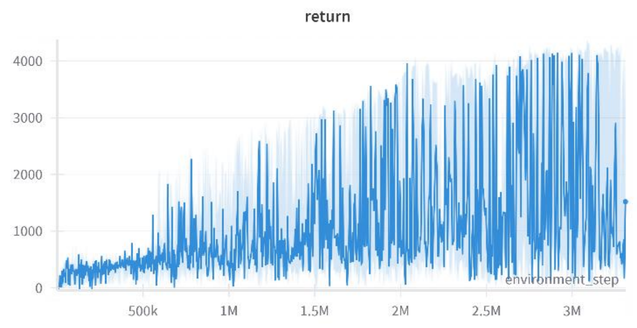
Task2



Command:

--actor-lr 2e-4 --critic-lr 1e-3 --discount-factor 0.99 --num-episodes 2000 --seed 77 --entropy-weight 0.01 --tau 0.95 --batch-size 64 --epsilon 0.2 --rollout-len 2048 --update-epoch 10

Task3



I reach on 1M~1.5M steps.

Command:

```
--actor-lr 2e-4 --critic-lr 2e-3 --discount-factor 0.99 --num-episodes 2000 --seed 77 --entropy-weight 0.01 --tau 0.95 --batch-size 256 --epsilon 0.1 --rollout-len 2048 --update-epoch 5
```

- Compare the sample efficiency and training stability of A2C and PPO.

  From the charts, PPO shows much faster convergence in the early stages, reward rises and stabilizes before 400k steps, indicating high sample efficiency. However, it later collapses sharply (after 600k steps), showing a lack of training stability.

  In contrast, A2C converges more slowly and exhibits large oscillations throughout whole training. Even when it appears stable, the reward fluctuates frequently, suggesting lower stability and higher sensitivity to noise.

  Summary:

  Sample Efficiency: PPO > A2C (PPO learns faster)

  Stability: Neither is fully stable, but PPO's late-stage collapse is more critical.

- Additional analysis on other training strategies (Bonus)

  1.

     During my experiments, I observed that changing the random seed had a significant effect on training performance. Specifically, when using seed = 7, the agent achieved faster convergence and higher returns compared to seed = 77, under the exact same hyperparameter settings.

     This shows that random initialization and the initial environment sampling can greatly influence the training dynamics in on-policy methods like PPO and A2C, which are sensitive to early exploration.

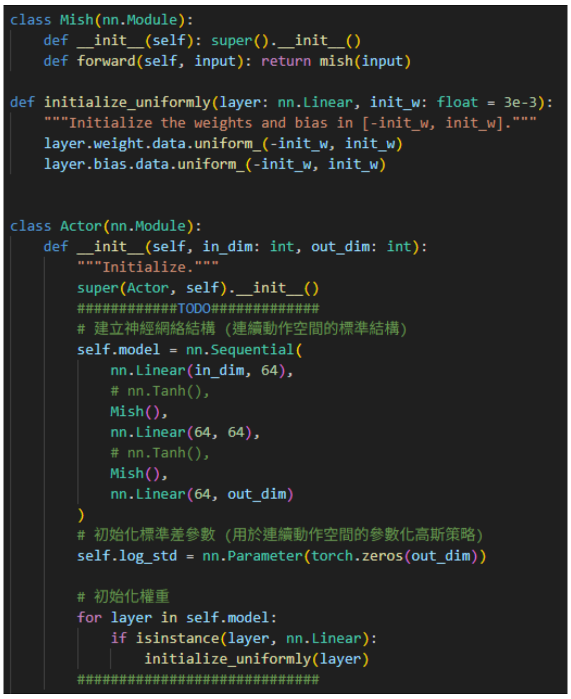
  2.

     During training with A2C, I found that no matter how I tuned the hyperparameters, the model consistently failed to converge and unstable.

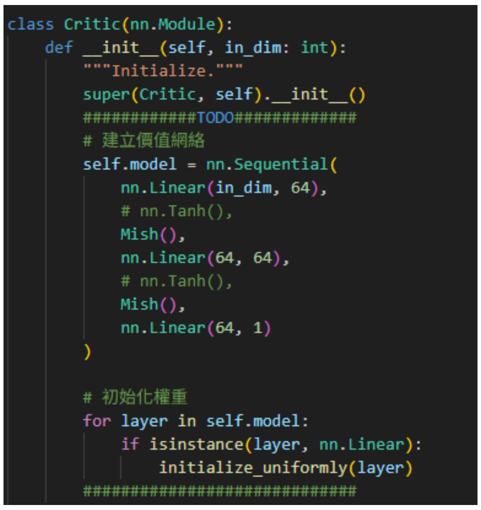
     To address this, I replaced the default activation functions (typically ReLU or Tanh) in both the Actor and Critic networks with Mish, a smooth and non-monotonic activation function.

I see the Mish function from the GitHub link below:

https://github.com/hermesdt/reinforcement-learning/blob/master/a2c/pendulum_a2c_online.ipynb



```python
class Mish(nn.Module):
    def __init__(self): super().__init__()
    def forward(self, input): return mish(input)

def initialize_uniformly(layer: nn.Linear, init_w: float = 3e-3):
    """Initialize the weights and bias in [-init_w, init_w]."""
    layer.weight.data.uniform_(-init_w, init_w)
    layer.bias.data.uniform_(-init_w, init_w)


class Actor(nn.Module):
    def __init__(self, in_dim: int, out_dim: int):
        """Initialize."""
        super(Actor, self).__init__()
        ############TODO#############
        # 建立神經網絡結構 (連續動作空間的標準結構)
        self.model = nn.Sequential(
            nn.Linear(in_dim, 64),
            # nn.Tanh(),
            Mish(),
            nn.Linear(64, 64),
            # nn.Tanh(),
            Mish(),
            nn.Linear(64, out_dim)
        )
        # 初始化標準差參數 (用於連續動作空間的參數化高斯策略)
        self.log_std = nn.Parameter(torch.zeros(out_dim))

        # 初始化權重
        for layer in self.model:
            if isinstance(layer, nn.Linear):
                initialize_uniformly(layer)
        #############################
```



```python
class Critic(nn.Module):
    def __init__(self, in_dim: int):
        """Initialize."""
        super(Critic, self).__init__()
        ############TODO#############
        # 建立價值網絡
        self.model = nn.Sequential(
            nn.Linear(in_dim, 64),
            # nn.Tanh(),
            Mish(),
            nn.Linear(64, 64),
            # nn.Tanh(),
            Mish(),
            nn.Linear(64, 1)
        )

        # 初始化權重
        for layer in self.model:
            if isinstance(layer, nn.Linear):
                initialize_uniformly(layer)
        #############################
```