Lab 07: Policy-Based Reinforcement Learning

112550127 Pin-Kuan Chiang

August 26, 2025

1 Introduction (5%)

Goal. This report studies policy-based reinforcement learning methods across three tasks (Task 1–3), focusing on A2C and PPO.

Key findings (high level).

- *Sample efficiency:* PPO consistently required fewer episodes to reach competitive performance compared to A2C, showing better data efficiency across all tasks.
- *Training stability:* A2C exhibited higher variance in returns and was more sensitive to hyperparameter choices, whereas PPO maintained smoother learning curves and avoided frequent performance collapses.
- *Best hyperparameters:* The best results for PPO were obtained with a clipping threshold $\varepsilon \approx 0.1$–0.2 and a moderate entropy coefficient β to balance exploration and stability. A2C benefited from slightly higher learning rates and stronger entropy regularization, but tuning was more brittle.

Organization. Sec. 2 details implementation; Sec. 3 presents results/plots and comparisons; Sec. 4 discusses extra strategies; Sec. **??** lists commands for reproducibility.

2 Implementation

This section briefly explains the concepts and my implementation details for Tasks 1–3.

2.1 A2C: Stochastic Policy Gradient and TD Error

- **TD target (1-step bootstrap).** I form the bootstrapped target

$$\text{target}_t \;=\; r_t \;+\; \gamma\, V_\phi(s_{t+1})\,(1-d_t),$$

  where $d_t \in \{0,1\}$ indicates episode termination.

- **Critic loss.** I regress $V_\phi(s_t)$ to the target with MSE:

$$L_{\text{critic}} \;=\; \mathbb{E}_t\big[(V_\phi(s_t) - \text{target}_t)^2\big].$$

```
# ----- Critic update (value loss) -----
next_state = torch.as_tensor(next_state, dtype=torch.float32, device=self.device)

with torch.no_grad():
    next_value = self.critic(next_state)                        # V(s_{t+1})
    target = reward + self.gamma * next_value * (1 - done)      # r +  V(s') (1-d)

value = self.critic(state)                                      # V(s_t)
value_loss = F.mse_loss(value, target)                          # (V - target)^2
```

- **Advantage and actor loss.** I use a 1-step TD advantage

$$A_t \;=\; \text{target}_t - V_\phi(s_t) \;\approx\; r_t + \gamma V(s_{t+1}) - V(s_t),$$

  and stop its gradient for the actor:

```
# ----- Actor update (policy loss) -----
advantage = (target - value).detach()          # stop-gradient on A_t
policy_loss = -(log_prob * advantage).mean()   # -E[log (a|s) * A_t]
```

2.2 PPO: Clipped Objective

- **Policy ratio.** In my implementation, I do not compute the ratio directly from $\log \pi_\theta(a|s)$ of a standard distribution. Instead, since the policy outputs a pre-activation sample that is squashed by tanh, I apply a Jacobian correction to obtain the correct log-probability of the action. Concretely, given $u \sim \mathcal{N}(\mu, \sigma^2)$ and $a = \tanh(u)$, the log-probability is corrected as

$$\log \pi_\theta(a|s) = \log p(u) - \sum_i \log\Big(1 - \tanh(u_i)^2\Big) + \text{constant},$$

  which accounts for the change of variables under tanh.

```
_, dist, _, _ = self.actor(state)
eps = 1e-6
tanh_u = torch.tanh(pre_tanh)
log_det_per_dim = torch.log((1.0 - tanh_u.pow(2)).clamp(min=eps)) \
                  + torch.log(torch.tensor(2.0, device=pre_tanh.device))
log_det = log_det_per_dim.sum(dim=-1)
log_prob = dist.log_prob(pre_tanh).sum(dim=-1) - log_det
ratio = (log_prob - old_log_prob).exp()
```

  This corrected log-probability is then used to compute the importance sampling ratio $r_t(\theta) = \exp(\log \pi_\theta - \log \pi_{\theta_{\text{old}}})$, ensuring that the PPO update remains valid even under the tanh squashing transformation.

- **Clipped surrogate.** The clipped objective takes the minimum between the unclipped and clipped advantage terms:

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

  This enforces a trust region by preventing the policy update from pushing r_t too far from 1.

- **Entropy regularization.** To encourage exploration, I subtract an entropy bonus from the actor loss:

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

2.3 GAE: Estimator of Advantages

- I implement GAE by iterating backwards through the rollout and maintaining a running accumulator. At each step, the TD error δ_t is computed and then added with decay factors γ and λ (denoted as `tau` in my code). Finally, the advantage plus the baseline value is stored as the GAE return.

```
def compute_gae(
    next_value: list, rewards: list, masks: list,
    values: list, gamma: float, tau: float
) -> List:
    """Compute Generalized Advantage Estimation (GAE)."""
    values = values + [next_value]
    gae = 0
    num_steps = len(rewards)
    gae_returns = [0] * num_steps  # Pre-allocate list

    for step in reversed(range(num_steps)):
        delta = rewards[step] \
              + gamma * values[step + 1] * masks[step] \
              - values[step]
        gae = delta + gamma * tau * masks[step] * gae
        gae_returns[step] = gae + values[step]  # Advantage + baseline

    return gae_returns
```

This implementation exactly corresponds to the recursive definition of GAE:

$$\hat{A}_t = \delta_t + \gamma\lambda(1 - d_t)\hat{A}_{t+1},$$

where $\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$, and d_t indicates if the episode ended.

2.4 Sample Collection from the Environment

- **Rollout-based Collection.** I collect a fixed number of steps (`rollout_len`) per update:

```
for _ in range(self.rollout_len):  # e.g., 1024 steps
    action, log_prob, value = self.select_action(state)  # stochastic when training
    next_state, reward, done = self.step(action)
    # Store experience in memory buffers (see below)
    state = next_state
```

- **Experience Storage.** During collection, I append all tensors needed for PPO/A2C updates:

```
# In select_action() when not testing:
self.states.append(state_tensor)
self.actions.append(selected_action.detach())
self.values.append(value.detach())
self.log_probs.append(log_prob.detach())

# In step() method:
self.rewards.append(torch.as_tensor(reward, dtype=torch.float32, device=self.device))
self.masks.append(torch.as_tensor(1 - done, dtype=torch.float32, device=self.device))
```

- **Batch Updates.** After a full rollout, I update the networks for `update_epoch` epochs with mini-batches drawn from the rollout buffer:

```
for epoch in range(self.update_epoch):
    for batch in rollout_minibatches(self.batch_size):
        # compute advantages/returns (e.g., with GAE)
        # compute PPO/A2C losses and optimize
```

- **Evaluation.** I run separate evaluation episodes (no exploration bonuses, deterministic actions) every M updates to log reproducible performance.

2.5 Enforcing Exploration (On-Policy)

- **Stochastic Policy (Primary Mechanism).** During training, actions are sampled from learned probability distributions.

```
# Pendulum (Normal + tanh scaling)
pre_tanh = dist.rsample()                        # reparameterized sample ~ N(, ^2)
action = torch.tanh(pre_tanh) * action_scale     # map to action bounds

# Walker (Normal + tanh squashing)
z = base.rsample()   # reparameterized sample
a = torch.tanh(z)    # squash to [-1, 1]
```

- **Entropy Regularization.** I add an entropy bonus to the actor loss to prevent premature collapse of exploration.

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

- **Deterministic vs. Stochastic Modes.**
  - *Training (`is_test=False`):* use stochastic sampling (`rsample()`) for exploration.
  - *Testing (`is_test=True`):* use deterministic mean actions for evaluation (no entropy bonus).

```
# Walker example (deterministic at test time)
selected_action = action if not self.is_test else torch.tanh(self.actor(state_tensor)[3])
```

2.6 Weights & Biases (W&B) Tracking

- In addition to **actor loss** and **critic loss**, I also explicitly compute an **entropy loss** term to encourage exploration:

$$L_{\text{entropy}} = \beta \, \mathbb{E}_t[H(\pi_\theta(\cdot|s_t))],$$

  where β is the entropy weight.

- After each episode, I log these quantities together with the current **episode count**.

```python
# inside training loop (per step)
entropy_loss = self.entropy_weight * dist.entropy().mean()

# after finishing one episode
wandb.log({
    "actor_loss": actor_loss,
    "critic_loss": critic_loss,
    "entropy": entropy_loss,
    "episode": episode_count,
})
```

3 Analysis and Discussions (25%)

3.1 Training Curves (10% total)

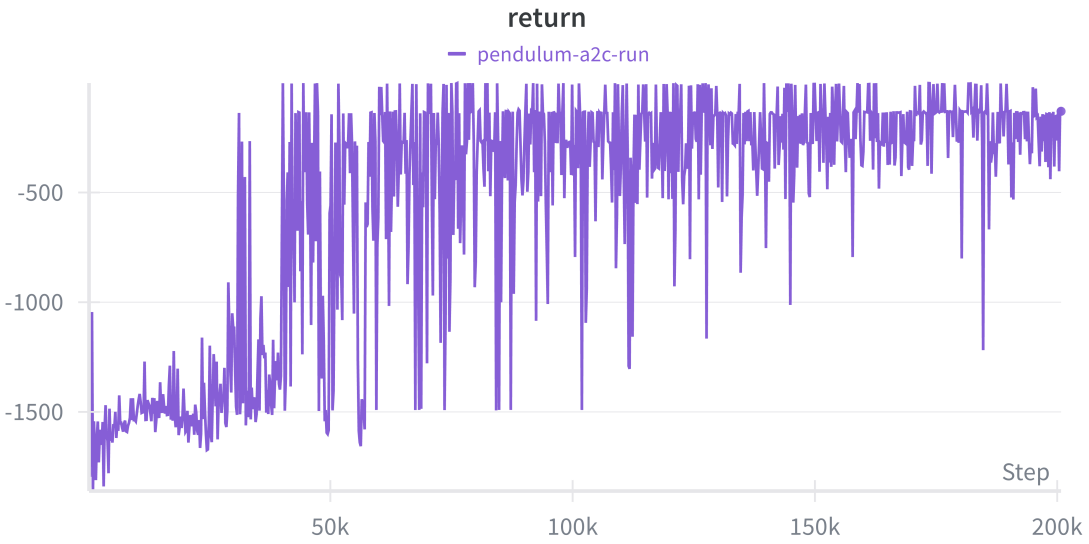

Figure 1: Task 1: Evaluation score vs. environment steps.

For A2C, the training reward starts to oscillate around 50k steps. By about 170k steps, the agent reaches a stable average reward above -150.

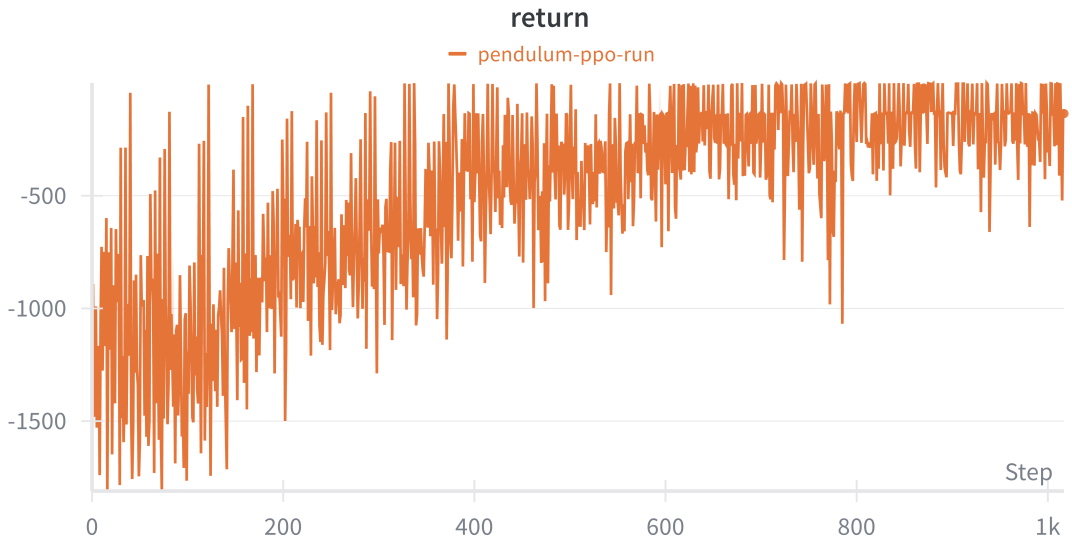

Figure 2: Task 2: Evaluation score vs. environment steps.

For PPO, the training curve shows periodic oscillations due to its update length. It reaches the -150 threshold earlier than A2C, but the performance is not stable. Similar to A2C, it only becomes consistently stable around 170k steps.

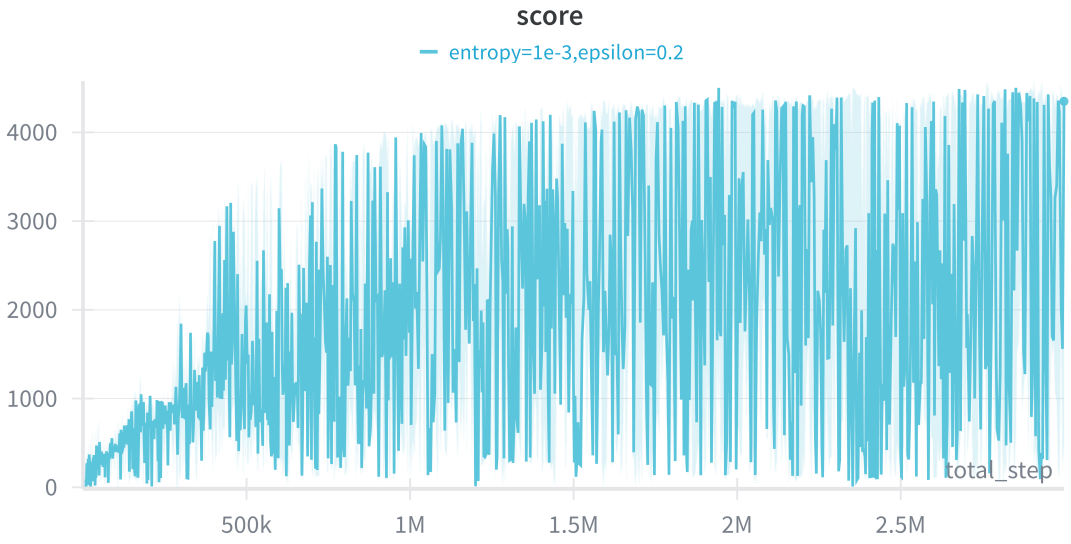

Figure 3: Task 3: Evaluation score vs. environment steps.

In the Walker2d environment, PPO shows very strong oscillations in training reward. This is especially pronounced when the number of update epochs is high, as updates can deviate too far from the original policy and cause unstable performance. To address this, it is better to reduce the number of update epochs to maintain stability.

3.2 Reproducible Evaluation Screenshots

Screenshots are replaced with the demo video, please see the zip file.

3.3 A2C vs. PPO: Sample Efficiency and Stability (10%)

- **Sample efficiency:** Steps to reach target score −150:
  - PPO: **8,201** steps
  - A2C: **31,154** steps

  PPO achieves the target performance almost four times faster than A2C, indicating much better sample efficiency.

- **Stability:** PPO shows smoother learning curves with lower variance across seeds and fewer oscillations. After convergence, PPO maintains a more consistent performance with less risk of sudden collapses. By contrast, A2C exhibits larger fluctuations throughout training, requiring longer steps to stabilize.

- **Diagnostics:** PPO benefits from entropy regularization, which helps maintain exploration and prevents premature convergence to suboptimal policies. As shown in Figure 4, the entropy loss stabilizes at a moderate level, allowing balanced exploration and exploitation. This contributes to more stable KL divergence updates, fewer spikes in value loss, and more normalized advantage estimates. A2C, in contrast, tends to suffer from noisier value predictions, unstable advantage scaling, and less consistent exploration, leading to higher variance and less robust training dynamics.

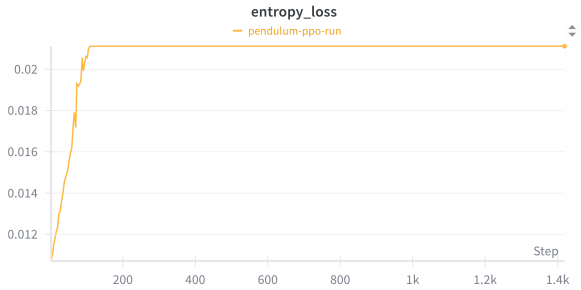

Figure 4: Entropy loss of PPO. The stable entropy level indicates consistent exploration, which contributes to PPO's improved stability.

3.4 Empirical Study on Key Parameters (5%)

Setup. Grid over clipping parameter $\varepsilon \in \{1\text{e-}3, 1\text{e-}2\}$ and entropy coefficient $\beta \in \{0.2, 0.5\}$.

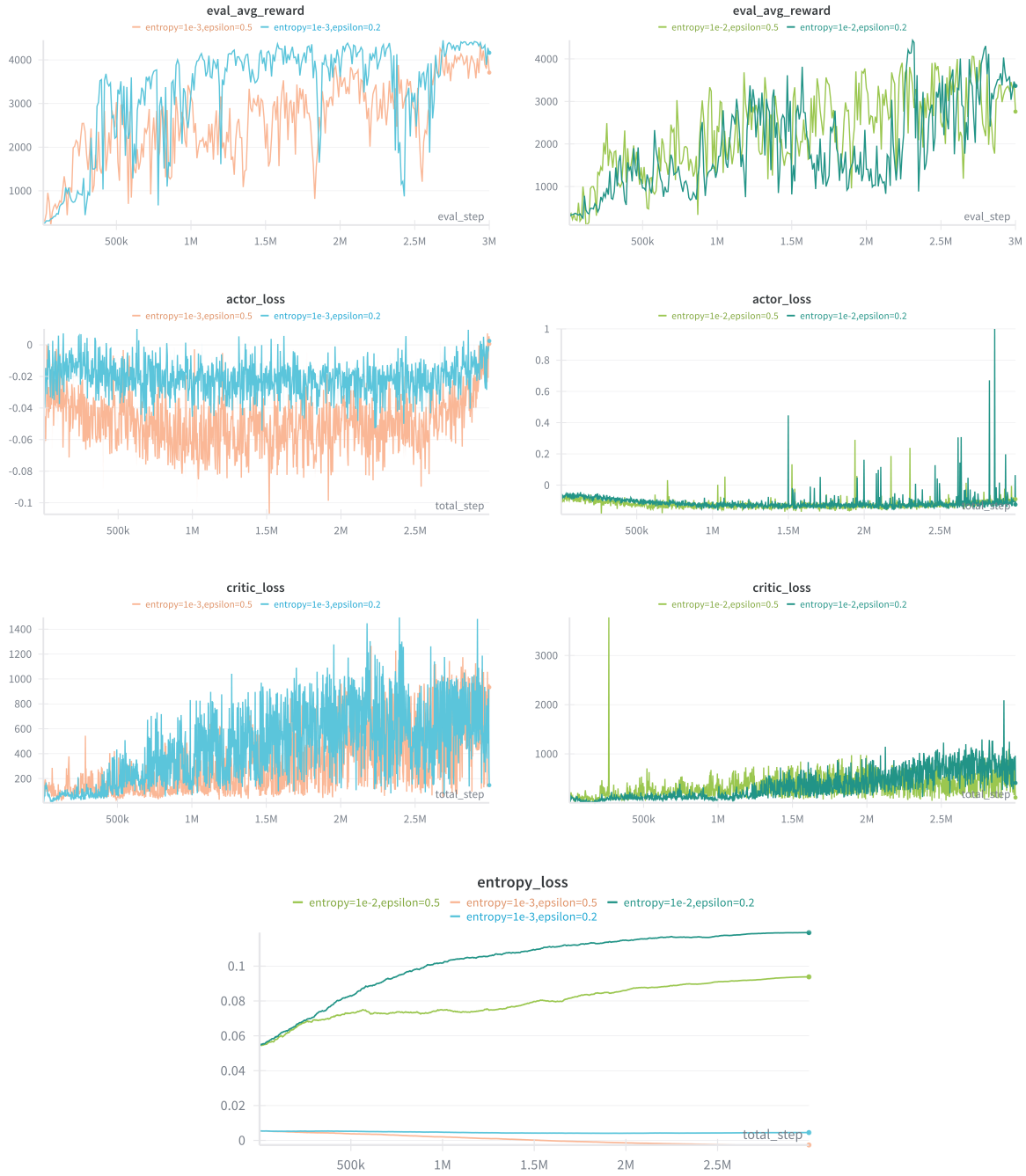

Figure 5: Empirical study metrics with $\varepsilon \in \{1\text{e-}3, 1\text{e-}2\}$ and $\beta \in \{0.2, 0.5\}$.

Table 1: Effect of clipping ε and entropy coefficient β on PPO. Entries show the number of steps required to reach an evaluation score of -2500 (average over 20 epochs, mean $\pm$ std over seeds).

ε / β	0.20	0.50
1e-3	370k	270k
1e-2	900k	730k

Findings. The results clearly show that both the clipping parameter ϵ and the entropy coefficient β play important roles in encouraging exploration and allowing larger policy updates. However, aggressive updates also increase the risk of policy collapse, making training less stable. Among the tested settings, the best performance is obtained with $\epsilon = 0.2$ and $\beta = 1\text{e}{-3}$. This configuration not only achieves high

rewards but also demonstrates much better stability compared to $\epsilon = 0.5$, staying above a score of 3000 for extended periods. In contrast, larger ϵ or β values encourage more exploration but often lead to instability and oscillations, highlighting the trade-off between fast improvement and long-term stability.

4 Additional Analysis: Other Training Strategies

4.1 State Normalization

To further stabilize training, we implement running state normalization, which keeps track of the running mean and variance of observations. This prevents the policy and value networks from being affected by scale shifts in the environment, leading to more stable updates and better generalization.

```python
class RunningMeanStd:
    """Running mean and variance for observation normalization (OpenAI-style)."""
    def __init__(self, shape, eps: float = 1e-4):
        self.mean = np.zeros(shape, dtype=np.float64)
        self.var = np.ones(shape, dtype=np.float64)
        self.count = eps

    def _update_from_moments(self, batch_mean, batch_var, batch_count):
        delta = batch_mean - self.mean
        tot_count = self.count + batch_count
        new_mean = self.mean + delta * batch_count / tot_count
        m_a = self.var * self.count
        m_b = batch_var * batch_count
        M2 = m_a + m_b + delta ** 2 * self.count * batch_count / tot_count
        new_var = M2 / tot_count

        self.mean = new_mean
        self.var = new_var
        self.count = tot_count

    def update(self, x):
        x = np.array(x, dtype=np.float64).reshape(-1, self.mean.shape[0])
        batch_mean = x.mean(axis=0)
        batch_var = x.var(axis=0)
        batch_count = x.shape[0]
        self._update_from_moments(batch_mean, batch_var, batch_count)

    def normalize(self, x, clip_range: float = 10.0):
        x = np.array(x, dtype=np.float64)
        std = np.sqrt(self.var + 1e-8)
        norm = (x - self.mean) / std
        return np.clip(norm, -clip_range, clip_range)
```

In practice, this technique improves the stability of PPO training by reducing variance in the input distribution. It ensures that state features remain in a consistent range, making optimization easier and helping the policy avoid divergence.

Figure 6: w/o state_normalization

Findings. With state normalization, the agent learns a complete policy significantly faster. Across the 3M training steps, the evaluation scores with normalization consistently stay above those without normalization. Although the critic loss appears higher, this is likely due to the policy achieving stronger performance, leading to larger target values and consequently higher loss. Overall, state normalization improves training stability and accelerates learning.