HW2: Implement PPO in HalfCheetah Environment

Shui Jie, 2401112104

April 9, 2025

Abstract

This report details the implementation of a Proximal Policy Optimization (PPO) algorithm for continuous control reinforcement learning tasks. The implementation is divided into five main components, each handling different aspects of the learning process. Together, these components form a complete and modular system capable of learning complex continuous control tasks in the HalfCheetah environment.

1 Introduction

Proximal Policy Optimization (PPO) is a policy gradient method that has shown impressive results on a variety of reinforcement learning tasks. This implementation focuses on applying PPO to the HalfCheetah continuous control environment from OpenAI Gymnasium. The goal is to teach a simulated cheetah-like robot to run forward as fast as possible.

2 Implementation Components

My PPO implementation is structured into five main components, each with distinct responsibilities in the reinforcement learning process.

2.1 Data Management Module

The data management module provides utilities for saving, loading, and managing numerical data throughout the training process.

- **File Path Management:** Ensures directories exist before saving files, with fallback to current directory
- **Array Persistence:** Converts between NumPy arrays and JSON format for storage
- **Append Operations:** Supports adding new values to existing data files
- **Error Handling:** Robust handling of file operations and data corruption

This module is essential for maintaining training history, saving metrics, and enabling analysis of the learning process. It supports the persistence of rewards, losses, and other numerical data between program runs.

2.2 Neural Network Architecture

The neural network module implements the core policy and value networks required for the PPO algorithm.

- **ValueNetwork (Critic):** Estimates state values for advantage calculation
  - Three-layer architecture ($256 \to 256 \to 1$) with ReLU activations
  - MSE loss for value prediction
- **PolicyNetwork (Actor):** Generates actions from a learned probability distribution
  - Three-layer architecture ($256 \to 256 \to$ action_dim) with ReLU activations
  - Learns both the mean and standard deviation of a normal distribution
  - Implements clipped surrogate objective for PPO updates

Both networks include utilities for saving and loading weights, enabling continuity across training sessions and deployment.

2.3 Trajectory Processing

The trajectory processor handles the collection and preparation of experience data for training updates.

- **Parallel Data Collection:** Simultaneously collects trajectories from multiple environment instances
- **Advantage Estimation:** Implements Generalized Advantage Estimation (GAE) for stable policy updates
- **Data Formatting:** Organizes collected trajectories into tensor format for efficient processing
- **Batch Sampling:** Provides random sampling functionality for mini-batch training

This component bridges the environment interaction and neural network training, preparing the raw experience data into the format required for effective learning.

2.4 Training Loop

The training module orchestrates the overall learning process, integrating all other components.

At each time step:

1. **Data Collection:** Gathers trajectories using current policy in parallel environments
2. **Network Updates:** Performs 10 epochs of policy and value network updates
3. **Evaluation:** Periodically measures performance of the current policy
4. **Checkpointing:** Saves model weights at regular intervals for continuity
5. **Metric Logging:** Records training progress for later analysis

The training loop implements the iterative improvement process central to reinforcement learning, gradually refining the policy to maximize cumulative rewards.

2.5 Visualization

The visualization module provides tools for monitoring and analyzing the training process.

- **Metric Plotting:** Visualizes training metrics such as policy loss, value loss, and rewards
- **Progress Tracking:** Shows performance trends over time
- **Policy Simulation:** Renders learned policies in the environment for visual inspection
- **Result Saving:** Supports saving plots for documentation and reporting

This component is crucial for understanding the learning dynamics, diagnosing issues, and demonstrating the effectiveness of the trained policy.

3 Integration and Workflow

These five components work together in a cohesive reinforcement learning system:

1. The **Training Loop** coordinates the overall process
2. The **Trajectory Processor** collects experience from environment interactions
3. The **Neural Networks** learn from the processed trajectory data
4. The **Data Management** module persists metrics and model weights
5. The **Visualization** tools monitor progress and analyze results

This modular design allows for flexibility in adapting the implementation to different environments and requirements while maintaining a clear separation of concerns.

4 How to run the code

To train model and check effectiveness, only 2 files needs to run

- **python training.py** Runs the training, automatically generate a data folder which contains the weights of both networks and the training data (policy loss, value loss, reward)
- **python visual.py** generates 3 plots based on the data, loads the policy network and simulates it in Gym environment.

5 Mathematical Formulation

The core of the PPO algorithm is the clipped surrogate objective:

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t \left[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t) \right] \tag{1}$$

where $r_t(\theta)$ is the probability ratio:

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \tag{2}$$

The advantage estimator $\hat{A}_t$ is calculated using Generalized Advantage Estimation (GAE):

$$\hat{A}_t = \delta_t + (\gamma\lambda)\delta_{t+1} + ... + (\gamma\lambda)^{T-t+1}\delta_{T-1} \tag{3}$$

where $\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$ is the temporal difference error.

```python
def get_loss(self, trajectory_tensor, epsilon=0.2):
    """
    Calculate PPO clipped surrogate objective for policy optimization.
    """
    # Extract tensors from trajectory data
    states = trajectory_tensor['states']
    actions = trajectory_tensor['actions']
    old_log_probs = trajectory_tensor['old_log_probs']
    advantages = trajectory_tensor['advantages']

    # Calculate new action probabilities
    new_log_probs = self.get_action_probability(states, actions)

    # Calculate probability ratio
    ratio = torch.exp(new_log_probs - old_log_probs)

    # Clipped surrogate objective
    surrogate1 = ratio * advantages
    surrogate2 = torch.clamp(ratio, 1-epsilon, 1+epsilon) * advantages

    # Take minimum (pessimistic bound)
    policy_loss = -torch.min(surrogate1, surrogate2).mean()

    return policy_loss
```

Listing 1: PPO Loss Calculation

```python
    def calculate_gae_tensors(self, states, rewards, next_states, dones, value_network, gamma=0.99, lambda_=0.95):
        # Get shapes for reshaping
        batch_size = states.size(1)
        time_steps = states.size(0)
        state_dim = states.size(2)

        # Flatten the batches for efficient value network evaluation
        flat_states = states.reshape(-1, state_dim)
        flat_next_states = next_states.reshape(-1, state_dim)

        # Get value estimates (single batch processing)
        with torch.no_grad():
            flat_values = value_network.forward(flat_states)
            flat_next_values = value_network.forward(flat_next_states)

        # Reshape values back to [time_steps, batch_size]
        values = flat_values.reshape(time_steps, batch_size)
        next_values = flat_next_values.reshape(time_steps, batch_size)

        # Ensure rewards and dones have the right shape
        if rewards.dim() == 3:  # If rewards has shape [time_steps, batch_size, 1]
            rewards = rewards.squeeze(-1)
        if dones.dim() == 3:  # If dones has shape [time_steps, batch_size, 1]
            dones = dones.squeeze(-1)

        # Calculate td-error (      =  r   +  V (s_{t+1}) - V( s  ))
        deltas = rewards + gamma * next_values * (~dones).float() - values

        # Initialize advantages tensor
        advantages = torch.zeros_like(deltas)

        # Calculate GAE by working backwards
        last_gae = torch.zeros(batch_size, device=states.device)

        for t in reversed(range(time_steps)):
            # For done episodes, reset the last_gae
            mask = (~dones[t]).float()
            last_gae = deltas[t] + gamma * lambda_ * mask * last_gae
            advantages[t] = last_gae

        # Calculate returns (G = A + V)
        returns = advantages + values

        # Normalize advantages
        advantages = (advantages - advantages.mean()) / (advantages.std() + 1e-8)

        return advantages, returns
```

Listing 2: GAE advantage calculation

6 Performance Results

This section could include charts and graphs showing the performance of the PPO algorithm on the HalfCheetah environment.

6.1 Experiment design

I have experimented with 2 different trajectory collection setups which affected the way the PPO performs.In the first experiment, I collected 10 trajectories that extend to full 1000 time steps. For the second experiment, I collected 100 trajectories of 100 time steps and trained the networks first.

Unfortunately, due to a bug in my code, I was unable to collect the reward function for the second experiment. Due to a limited time, I am unable to conduct another training session and thus I will only display the reward for the first experiment.

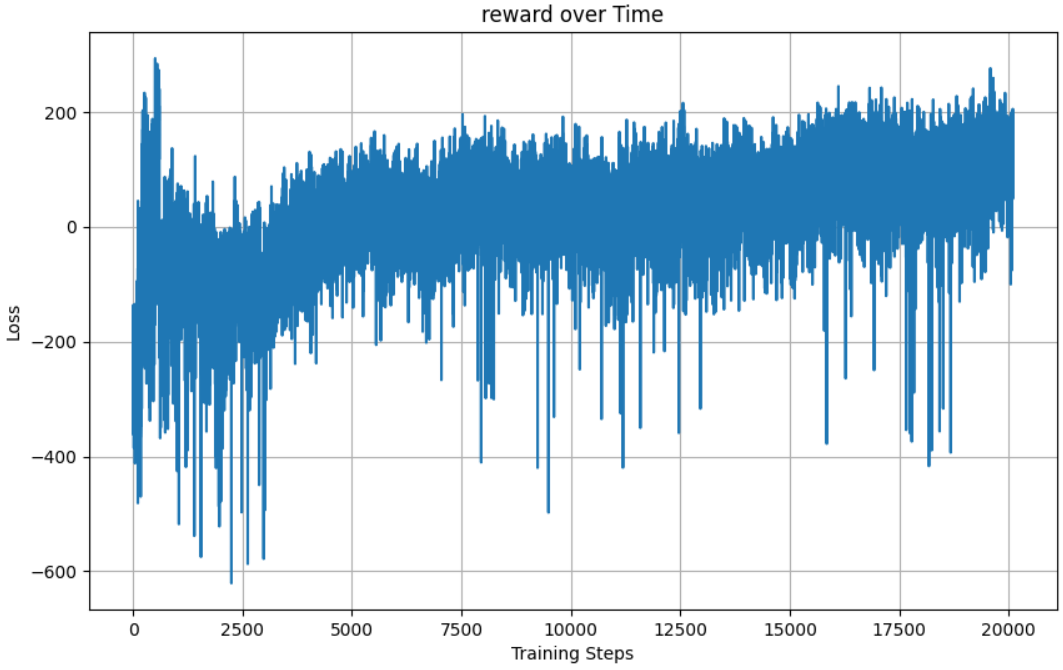


Figure 1: reward overtime shows consistent improvement from -600 to 200. Consistent positive reward collected after 15000 time-steps indicating the agent is able to effectively learn the task needed.

6.1.1 Experiment 1: small number of long trajectories

I trained a new policy network and value network for 20000 epoch using fixed trajectory collection. For each epoch, I collect 10 trajectories of 1000 time steps for training. The data collected for policy loss, value loss and reward overtime collected is displayed in Appendix section A.

1. **Policy loss** stabilizes within a band between approximately -0.4 and 0.5, with occasional spikes. The constant oscillation within a range indicates small adjustments rather than major strategy shifts, which is consistent with PPO's trust region constraint working as intended.

2. **Value loss** shows very high variance over time. This indicates that the policy network is extensively exploring the novel action and state space, which leads to inherent uncertainty in predicting future returns from current states.

3. **Actual simulation** We observe the HalfCheetah to almost immediately started flipping upside down the moment simulation starts. The agent then moved forward awkwardly from an upside down position.

At time step 20000, the agent has learn a suboptimal policy where it can move forward while being upside-down, generating a reward between 50 to 100.

Based on the data, I observed that both policy network and value network are consistent in their loss pattern, indicating that they are effectively learning from the environment and are still far from convergence. And thus I believe that given enough time, the agent can learn to correct the current suboptimal policy and move upright.

6.1.2 Experiment 2: 100 trajectory, 100 time steps

I wanted to experiment a different approach. Instead of small number of longer trajectories(10 trajectories, 1000 time steps), I collect a larger number of trajectories, with shorter time steps(100 trajectories, 100 time steps).

1. **Policy loss** is consistent with the first experiement, stabilizes within a band between approximately -0.4 and 0.5, with occasional spikes.

2. **Value loss** is very high initially, up to 250 as compared to less than 100 in the first experiment. However the value loss quickly decreased, and eventually maintained at a similar value as the first experiment. This could be due to the lower variance in the state spaces as the shorter trajectories also means a smaller distribution in state spaces.

3. **Reward** The agent was able to maintain upright for longer period of time before flipping over, generating a high reward between 100 to 200.

   Due to an error in the setup, I was unable to gather the correct reward data overtime for the second experiment. Further more I was unable to conduct another experiment due to time constraint. Therefore I am unable to conclude if the second experiment is more effective despite the higher reward.

7 Conclusion

This PPO implementation represents a complete and modular reinforcement learning system capable of learning complex continuous control tasks. The clear separation of concerns across the five components makes the code maintainable and extensible, while the efficient implementation allows for effective learning within reasonable computational constraints.

Experiment 1

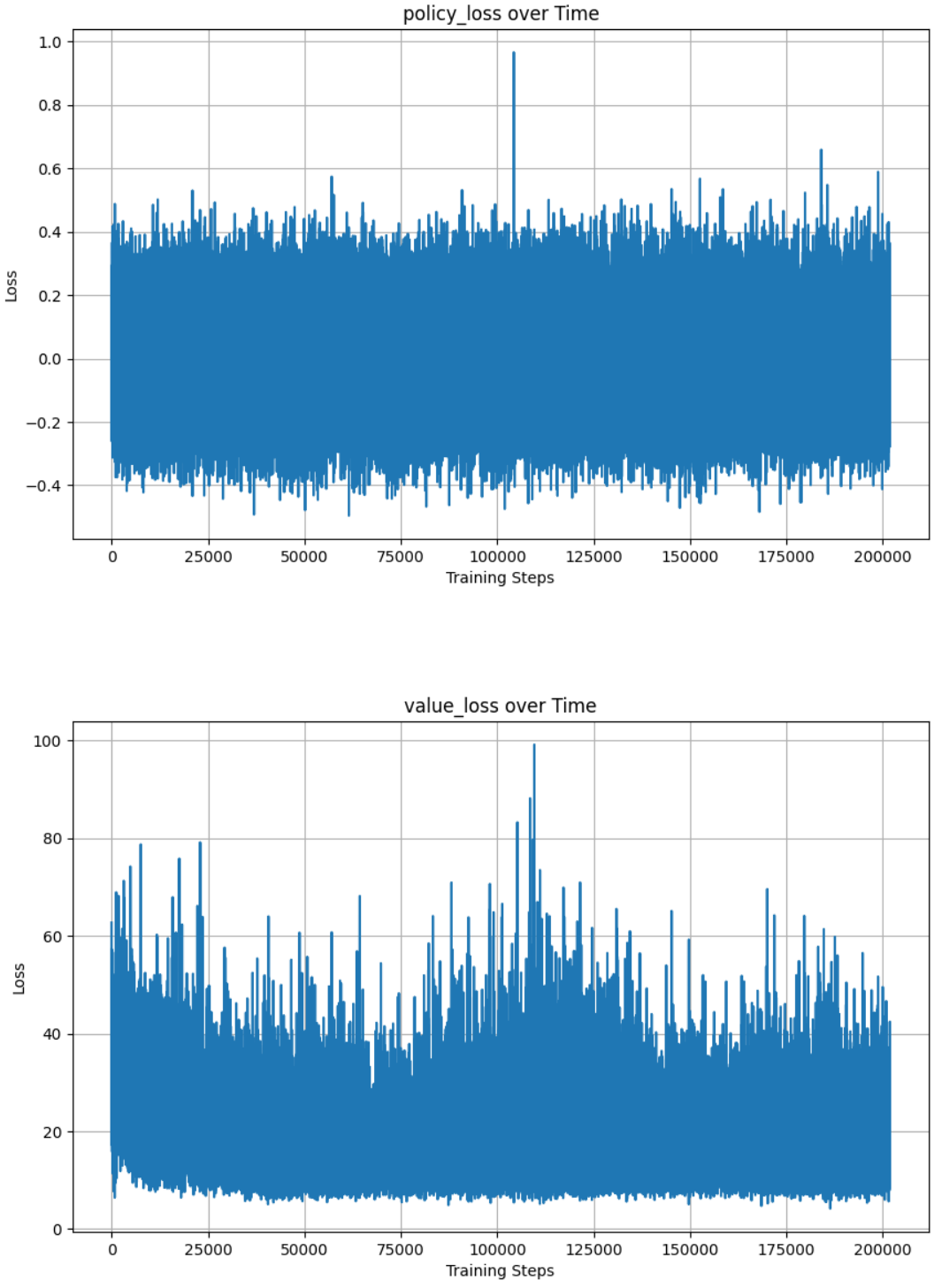
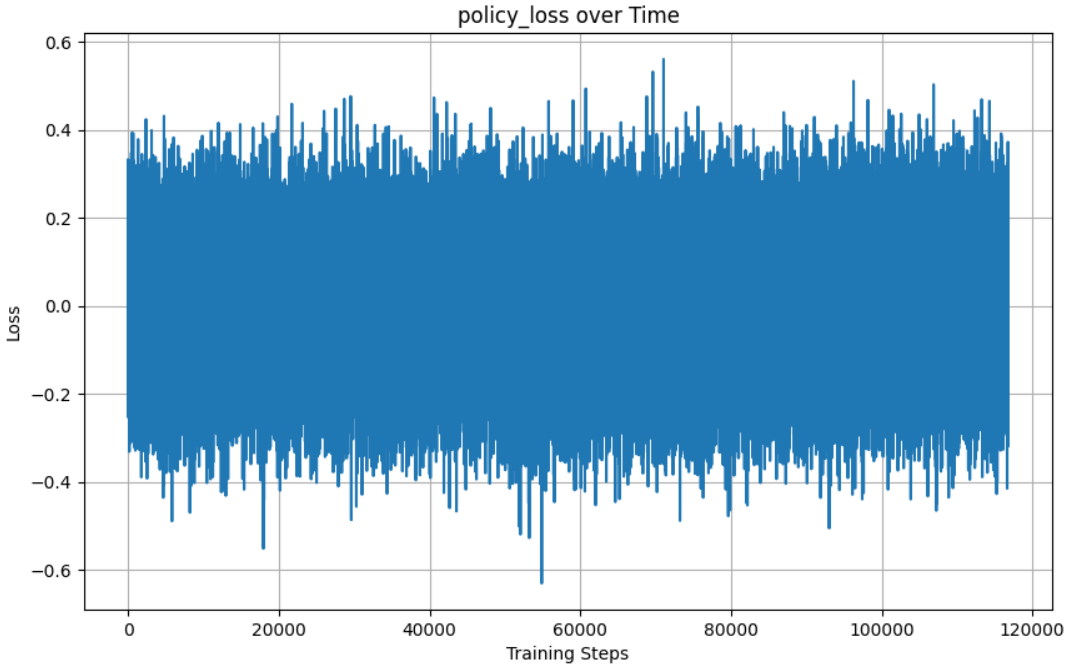


Figure 2: Training metrics for PPO implementation on HalfCheetah environment over 200,000 training steps using 10 trajectories with 1000 timesteps

Experiment 2



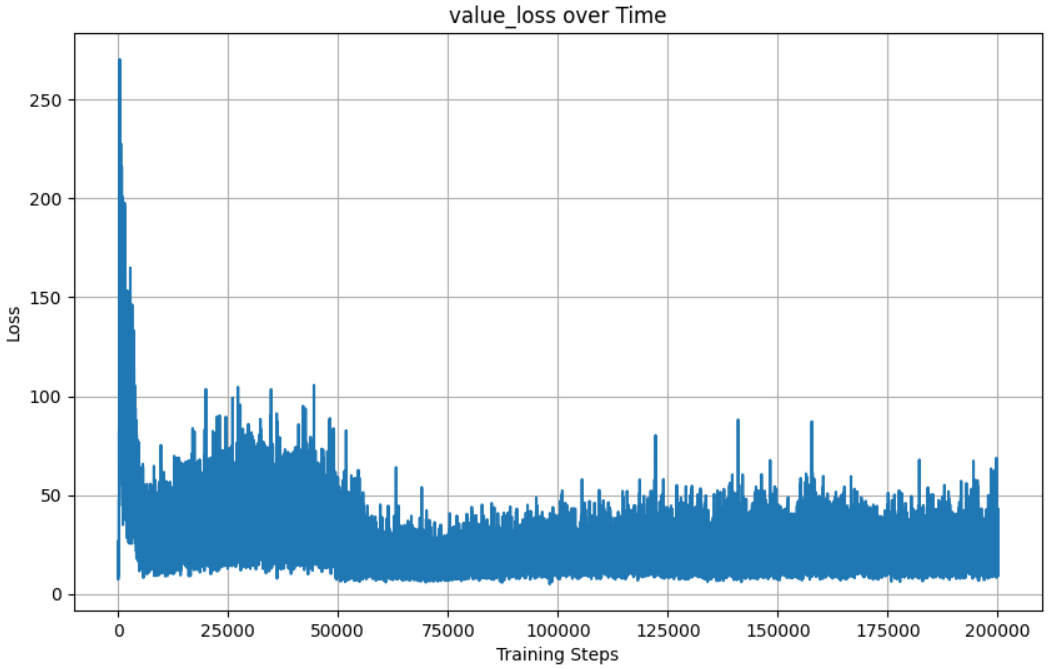
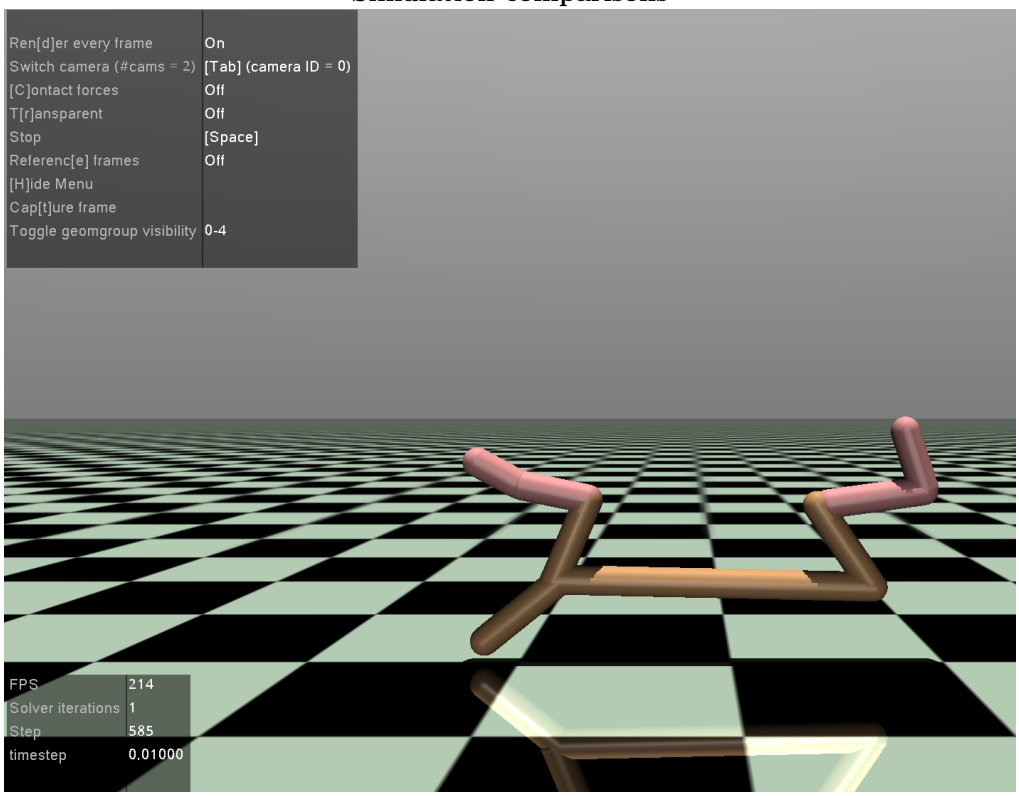
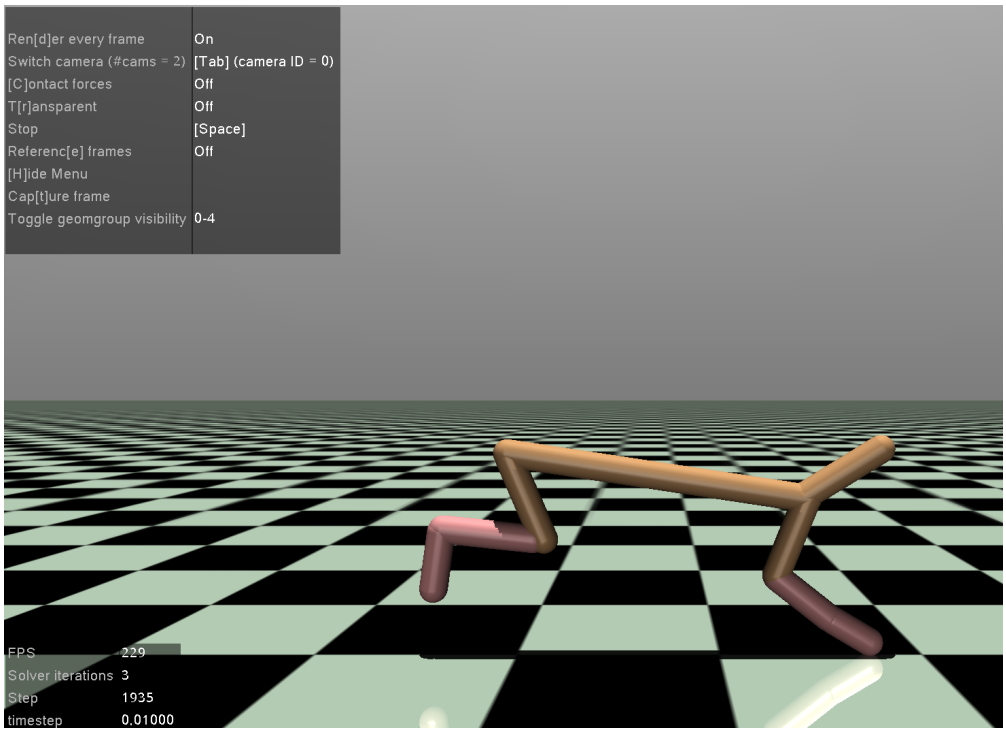


Figure 3: Training metrics for PPO implementation on HalfCheetah environment over 200,000 training steps using 100 trajectories with 100 timesteps

Simulation comparisons



(a) agent trained with small number of long trajectories almost immediately flipped over and starting moving upside down



(b) agent trained trained with large number of short trajectories maintained upright and ran for much longer, though eventually still flipped.

Figure 4