ME5418 - Machine Learning in Robotics
Group 16

Ji-Dog quadrupedal Robots Locomotion via Reinforcement Learning
Agent Report

Group members:

Bai Yue (ID: A0304233U)

Ji Shuchen (ID: A0305152U)

Zhang Binlin (ID: A0304090U)

November 8, 2024

1 Project Introduction

In this project, we intend to train a quadrupedal robot to perform locomotion using reinforcement learning. In previous reports, we finished an OpenAI Gym environment and a neural network for training. In this part, we choose Proximal Policy Optimization (PPO) to realize the reinforcement learning of the agent for its continuous action space support and stability. In *ji_dog_net.py*, we define *PPO* class to implement the PPO algorithm and also realize the interaction with the Isaac simulated environment we established previously.

2 Code Explanation

2.1 PPO Algorithm

In the PPO algorithm, the agent first accumulates experiences by interacting with the environment. It then calculates the advantage of taking certain actions over others in each state. Using this advantage, PPO updates the policy by maximizing the objective function, balancing between exploration and exploitation. In our *class PPO_Penalty*, functions are as follows.

- *select_action():* This function takes the current state, and uses *policy_old* to compute the action mean, log probability. And the critic calculates the value of the state. The state, action, action log probability, and state value are stored in the buffer for later training.
- *update():* This function computes and standardizes the rewards. The advantage function is also calculated. Then in each PPO update loop, the agent resets hidden states, evaluates new policy, calculates loss and conducts backpropagation.

2.1.1 Actor-Critic Framework

In the PPO algorithm, the Actor-Critic framework plays a crucial role in helping the agent learn more effectively, which is shown in Figure 1.

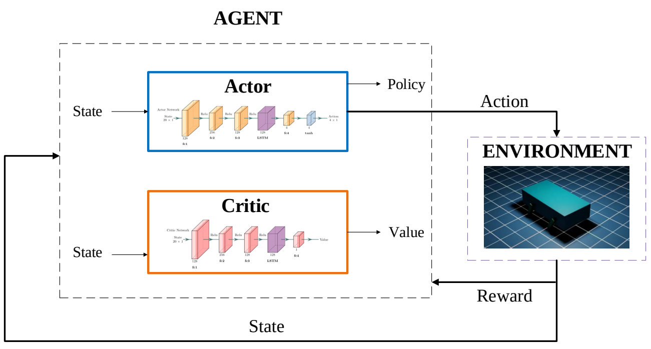

Figure 1: The Actor-Critic Framework in PPO

- *The Actor (Policy Network):* The actor is responsible for selecting actions based on the current state. It represents the policy which maps the state to probabilities of actions. The actor is trained to maximize cumulative rewards. In PPO, the actor is optimized using either a clipped surrogate objective or a KL-penalized objective function, which prevents violent changes to the policy and ensures stable updates.
- *The Critic (Value Network):* The critic evaluates the quality of the current policy by estimating the expected value of each state under this policy. The critic calculates the advantage function which shows how good a certain action is compared to others in this state and then makes better decisions based on the advantage function.

2.1.2 Two Version of PPO

We provide two versions of PPO for comparison: PPO-Clip and PPO-Penalty, which are realized in *ji_dog_net_v1.py* and *ji_dog_net_v2.py* respectively. The former tends to be more efficient while the latter provides better adaptability.
PPO-Clip:

- PPO-Clip doesn't apply any explicit constraint on policy updates.
- PPO-Clip limits the update magnitude through simply clipping. When the difference between the new policy and the old one exceeds a set threshold, it clips the objective function, effectively preventing the policy from deviating too far from its previous version.

PPO-Penalty:

- PPO-Penalty works similarly to Trust Region Policy Optimization(TRPO). But instead of a hard Kullback-Leibler(KL) constraint, it uses a penalty on the KL divergence between the new and old policies. By penalizing KL divergence in the objective function, PPO-Penalty enforces a ”soft” constraint, rather than an exact or ”hard” constraint, on policy changes.
- During training, PPO-Penalty adjusts the penalty coefficient dynamically based on the policy’s KL divergence. This adaptive tuning allows appropriate scaling of the penalty so that the policy update neither becomes too large nor too restrictive, helping to maintain stability in training.

2.2 Training Process

First, the simulated environment is set up and the agent is initialized. For each training episode, after resetting the parameters, the current state is obtained via *process_state* function. For each time step, the agent selects an action based on the current state. Then the agent takes the action, and returns and stores the next state, reward, a *done* flag and other necessary information. The episode will terminate in advance if *done* flag is 1.

2.2.1 PPO Update

For a set number of epochs (K epochs), the network updates the policy based on the experiences stored in the buffer.

- *Policy Evaluation:* The *evaluate* function calculates the log probabilities, state values, and entropy for the current policy.
- *KL Divergence Calculation:* The KL divergence between *policy_old* and the current policy is computed to measure how much the new policy differs from the old one.
- *Dynamic KL Penalty Adjustment:* If the average KL divergence exceeds the targeted KL by 1.5 times, the KL penalty coefficient will increase to penalize further divergence. If the average KL is less than the targeted KL by 1/1.5, the coefficient is decreased, allowing more flexibility in policy updates.
- *Loss Calculation:* The *compute_loss* function combines several components into the total loss: Policy Loss, Value Loss, KL Penalty, Entropy Bonus.

2.2.2 Visualization and Monitoring

To facilitate further analysis and reporting, TensorBoard-generated images were saved to the following folder: *Jidog 2.0/runs/TensorBoard_image*
The following key images were saved:

- **Average Loss.png**: Displays the average loss across episodes.
- **Policy Loss.png**: Shows the trend of the policy network’s loss.
- **Value Loss.png**: Shows the trend of the value network’s loss.

3 Existing Codes/Libraries

Existing Codes: We use our customized simulation environment *Ji_dog_env* built based on Isaac Sim to provide the PPO with necessary interface information such as states, rewards, and actions in reinforcement learning and also enable it to optimize the strategy with simulation feedback data. We also use our neural network *ActorCritic* established previously to train the agent.
PyTorch: An open-source deep-learning library for building and training neural networks.
TensorBoard: A visualization toolkit for monitoring metrics such as loss, reward, and model performance over time. In our code, *TensorBoard* is used to track training progress by logging key metrics, which helps analyze the agent’s learning dynamics and diagnose potential issues in real-time.

4 Reflections/Lessons Learned

4.1 Reflections

During the training process, we observed how subtle changes in parameters such as learning rate and exploration rate significantly impacted the stability and performance of the model. We also recognized the utility of visualization tools like TensorBoard. By continuously monitoring metrics such as average loss, policy loss, and value loss, we gained valuable insights into the model’s progress.

4.2 Lessons Learned

we noticed that the results did not meet our expectations, and the learned strategy occasionally showed instability. After analyzing the training process, we concluded that the reward function might not be effectively guiding the agent towards its goal. If the reward function is poorly designed, it may lead to suboptimal performance, as the agent might prioritize actions that bring higher immediate rewards but do not align with the overall objective. To address this, we plan to introduce more continuous reward and penalty parameters, involving additional processing and analysis of observation values.