Learning Agent C and R

Group 13 Du Xuezeng, Xue Jinhan, Yang Chengxu

Code Explanation

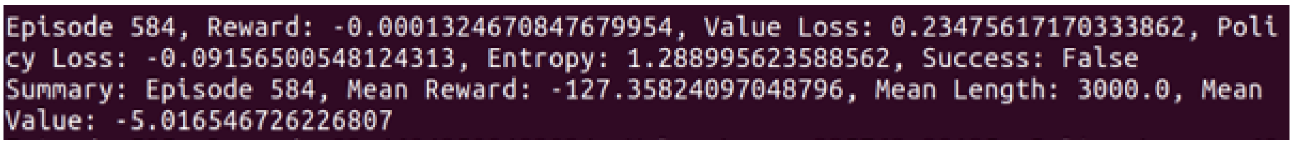
The main script for this project, `train_ppo.py`, is responsible for training the Actor-Critic Network (ACNet) using Proximal Policy Optimization (PPO) to control a cargo lander in a custom simulation environment. The code is organized into several key components, each contributing to different aspects of the training and interaction process.

The core logic of the training process is encapsulated within the `Worker` class in `train_ppo.py`. This class manages the interaction between the cargo lander and the environment, collects experience during each episode, and performs the necessary computations to update the model parameters. The primary goal is to optimize the neural network's policy and value functions using the PPO algorithm.

The `train()` method is one of the most important functions in the script. It computes the policy and value losses, followed by applying gradient updates to improve the model's decision-making ability. The PPO algorithm is used here to ensure stable updates through policy ratio clipping, which prevents the network from making overly drastic changes during optimization. This allows for consistent improvement while maintaining exploration of the action space.

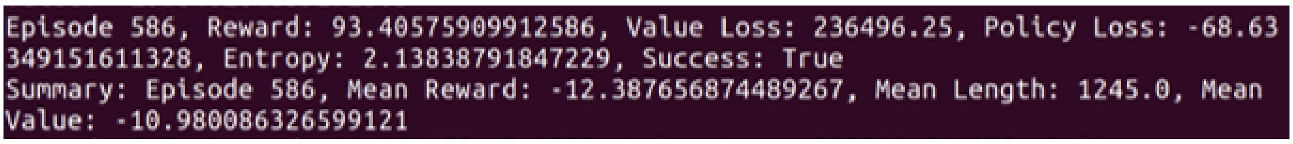
The `work()` method in the `Worker` class runs multiple episodes, wherein the lander interacts with the environment, takes actions based on its current policy, and learns from the rewards it receives. The reward for each episode is progressively getting higher, indicating that the learning process is moving in the right direction. However, the policy convergence is still relatively slow, which implies that more parameter tuning is needed to enhance the learning speed. Each episode's data is accumulated and used to update the neural network after enough experiences are gathered. This is essential to train the model effectively, as it balances the exploration of new actions with the exploitation of learned strategies.

Overall, `train_ppo.py` orchestrates the entire training pipeline by creating an instance of the `Worker` class, initializing the environment, and repeatedly running episodes to improve the policy and value functions through PPO. The combination of the environment, the Actor-Critic neural network, and the training process forms a cohesive system aimed at solving the complex task of lunar landing.



```
Episode 584, Reward: -0.0001324670847679954, Value Loss: 0.23475617170333862, Poli
cy Loss: -0.09156500548124313, Entropy: 1.288995623588562, Success: False
Summary: Episode 584, Mean Reward: -127.35824097048796, Mean Length: 3000.0, Mean
Value: -5.016546726226807
```

Figure 1: Before training



```
Episode 586, Reward: 93.40575909912586, Value Loss: 236496.25, Policy Loss: -68.63
349151611328, Entropy: 2.13838791847229, Success: True
Summary: Episode 586, Mean Reward: -12.387656874489267, Mean Length: 1245.0, Mean
Value: -10.980086326599121
```

Figure 2: Mean reward during training

Existing Codes/Libraries

We used several existing libraries and tools to implement this project:

- **TensorFlow 1.x**: The neural network was implemented using TensorFlow 1.x, as it provided compatibility with existing codebases and was more convenient to use for this specific project.
- **PyBullet**: The cargo lander environment was built using PyBullet, which allows for realistic physics simulations, including lunar gravity. PyBullet provided a versatile platform for creating custom environments and simulating the interactions of the cargo lander with its surroundings.
- **NumPy**: NumPy was used for numerical computations, including state vector manipulation, reward calculations, and other mathematical operations.

Using these existing libraries significantly sped up the development process, as they provided reliable tools for both machine learning and physics simulation. The use of TensorFlow and PyBullet allowed us to focus on designing the reinforcement learning model and tuning it for optimal performance.

Reflections/Lessons Learned

During the development of this project, we encountered several challenges and made modifications to the original plan to achieve a stable and efficient learning model. One of the main challenges was maintaining compatibility between different versions of TensorFlow. We initially considered using TensorFlow 2.x, but due to compatibility issues with some existing modules, we decided to use TensorFlow 1.x.

Another key lesson was the importance of balancing exploration and exploitation. The inclusion of an entropy term in the policy loss helped the model explore more diverse actions, which was crucial for avoiding local optima and achieving a stable landing. Regular tuning of hyperparameters, such as the learning rate and gradient clipping thresholds, was also essential to ensure training convergence. However, the learning process is still quite slow, and the parameters are still under tuning to find the optimal setup.

We also plan to incorporate CUDA for GPU acceleration in future iterations. Currently, the training speed is limited due to CPU processing, and using CUDA could significantly speed up the training process, allowing us to experiment more rapidly with different configurations.

Additionally, we realized the value of simplifying the environment during the initial phases of development. For instance, we started with a simpler version of the environment that did not include complex terrain or additional disturbances. This allowed us to verify the correctness of our implementation before gradually increasing the complexity.

Moving forward, one area we would like to improve is the efficiency of the simulation. Currently, the time taken to simulate each episode is considerable, primarily due to the high number of simulation steps needed for the lander to reach the surface safely. Reducing the simulation time while maintaining the accuracy of the lander's behavior could significantly enhance the model's practical applicability.

In conclusion, this project provided valuable insights into the challenges of applying reinforcement learning to control a complex dynamic system, such as a cargo lander. We learned the importance of careful tuning, modular code design, and leveraging existing tools to efficiently tackle complex tasks. Although there are still areas for improvement, such as speeding up learning and incorporating CUDA for faster training, we believe the progress made thus far demonstrates the potential of reinforcement learning for similar control problems.