Reproducing Proximal Policy Optimization Algorithms

The paper "Proximal Policy Optimization Algorithms" by John Schulman and colleagues proposes a new family of reinforcement learning algorithms called Proximal Policy Optimization (PPO). PPO aims to improve upon other policy gradient methods, particularly Trust Region Policy Optimization (TRPO).

1. **Objective**: PPO aims to find an optimal balance between efficient policy updates and stable training. It accomplishes this by alternating between gathering data through environmental interactions and performing multiple mini-batch updates on a surrogate objective function (Schulman et al., 2017).
2. **Key Concepts**:
    - PPO implements a new objective function that enables repeated updates without diverging from the policy trust region.
    - It uses a "clipped" surrogate objective to constrain policy changes, ensuring that updates are not too drastic.
    - The algorithm can achieve improved sample efficiency compared to other methods and strikes a favorable balance between simplicity, stability, and performance.

Reinforcement learning with neural networks has seen various approaches emerge in recent years. While methods like Deep Q-learning and vanilla policy gradients hold promise, they suffer from limitations:

- **Deep Q-learning:** Brittle performance on simple tasks and limited theoretical understanding.
- **Vanilla Policy Gradients:** Data inefficiency and lack of robustness across problems.
- **Trust Region Policy Optimization (TRPO):** Computational complexity and incompatibility with architectures using noise or parameter sharing.

This research aims to bridge the gap by introducing Proximal Policy Optimization (PPO).

PPO seeks to achieve the data efficiency and reliability of TRPO while utilizing a simpler, first-order optimization approach. It accomplishes this through:

- **Clipped Probability Ratios:** Introducing a novel objective function with clipped probability ratios, acting as a conservative estimate of policy performance.
- **Alternating Optimization and Sampling:** The algorithm iterates between collecting data from the current policy and performing optimization steps on the collected data.

1. Background

Policy Gradient Methods (Formulas and Explanation)

Estimator Formula:
The policy gradient estimator computes the gradient of the policy parameters and uses it to adjust the policy. Here's the formula:

$$\hat{g} = \mathbb{E}_t \left[\nabla_\theta \log \pi_\theta(a_t \mid s_t) \hat{A}_t \right]$$

where:

- π_θ : Policy function parameterized by π_θ.
- (a_t, s_t) : Action taken and state observed at time step t.
- $\hat{A}_t$: Advantage function indicating how good action at was given st.
- ∇_θ: Gradient of the log-probability.

Policy Gradient Objective Function:

The policy gradient objective function helps us find the best parameters θ:

$$L^{\mathrm{PG}}(\theta) = \mathbb{E}_t \left[\log \pi_\theta(a_t \mid s_t) \hat{A}_t \right]$$

This expression measures the expected reward under the policy.

Trust Region Policy Optimization (TRPO)

TRPO Objective Function: In TRPO, we maximize the expected reward while ensuring policy updates are constrained:

$$\max_\theta \quad \mathbb{E}_t \left[\frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)} \hat{A}_t \right] \quad \text{s.t.} \quad \mathbb{E}_t \left[\mathrm{KL} \left[\pi_{\theta_{\text{old}}}(\cdot \mid s_t), \pi_\theta(\cdot \mid s_t) \right] \right] \leq \delta$$

Here:

- $\pi_{\theta_{\text{old}}}$ is the policy before the update.
- δ is a small positive threshold limiting policy updates.

TRPO Surrogate Objective: Instead of solving the constrained optimization problem directly, TRPO uses a surrogate objective function by introducing a penalty coefficientβ:

$$\max_\theta \quad \mathbb{E}_t \left[\frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)} \hat{A}_t - \beta \mathrm{KL} \left[\pi_{\theta_{\text{old}}}(\cdot \mid s_t), \pi_\theta(\cdot \mid s_t) \right] \right]$$

Proximal Policy Optimization: Improvements over TRPO

Proximal Policy Optimization (PPO) incorporates significant improvements over Trust Region Policy Optimization (TRPO):

1. **Clipped Surrogate Objective**:
   PPO employs a clipped surrogate objective to prevent drastic policy updates. The objective function minimizes deviations from the ratio of the old and new policy probabilities:

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

   Here:

   - $r_t(\theta)$: Probability ratio between current and previous policies.
   - $\hat{A}_t$: Advantage estimation at time t .
   - **Clipping Mechanism**: The clip function confines $r_t(\theta)$within the interval $[1 - \epsilon, 1 + \epsilon]$, preventing significant policy changes.

2. **Adaptive KL Penalty Coefficient**:
   In addition to clipping, PPO uses an adaptive KL penalty coefficient that dynamically adjusts according to the KL divergence from a target threshold:

$$L^{\mathrm{KLPEN}}(\theta) = \mathbb{E}_t \left[\frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)} \hat{A}_t - \beta \mathrm{KL}\left[\pi_{\theta_{\text{old}}}(\cdot \mid s_t), \pi_\theta(\cdot \mid s_t)\right] \right]$$

By updating the penalty coefficient β based on deviations of KL divergence:

- If $d < d_{\text{targ}}/1.5$: $\beta \leftarrow \beta/2$.
- If $d > d_{\text{targ}} \times 1.5$: $\beta \leftarrow \beta \times 2$.

These enhancements in PPO simplify and optimize the policy update process compared to TRPO, offering computational efficiency while maintaining policy stability.

2. Algorithm

Proximal Policy Optimization (PPO) utilizes a structured approach to policy gradient optimization, which is designed to improve training stability and efficiency. The algorithm incorporates several key components, each playing a significant role in achieving robust performance.

Key Components of the PPO Algorithm

1. **Surrogate Loss Functions**:
   PPO relies on surrogate loss functions to simplify policy optimization. These functions enable efficient computation and differentiation:

   $$L^{\mathrm{CLIP}} + L^{VF}(s) = \mathbb{E}_t \left[L^{\mathrm{CLIP}}(\theta) - c_1 L^{VF}(\theta) + c_2 S[\pi_\theta](s_t) \right]$$

   - $L^{\mathrm{CLIP}}(\theta)$: Clipped surrogate objective.
   - $L^{VF}(\theta)$: Value function loss.
   - $S[\pi_\theta](s_t)$: Entropy bonus to encourage exploration.
   - c_1, c_2: Coefficients balancing the components of the loss function.
2. **Policy and Value Function**:
   If the policy and value function share a neural network architecture, the combined loss function optimizes both policy improvement and value prediction accuracy, assisted by an entropy bonus for enhanced exploration.
3. **Advantage Estimation**:
   PPO employs a truncated version of the Generalized Advantage Estimation (GAE) for effective advantage computation:

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

   where:

   - $\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$: Temporal difference error.
   - γ: Discount factor.
   - λ: GAE parameter.
4. **Optimization Procedure**:
   The core of PPO's training process involves performing several epochs of stochastic gradient descent (SGD) on minibatches to optimize the combined loss:

```
for iteration = 1, 2, ... do
    for actor = 1, 2, ..., N do
        Run policy \pi_{\theta_{\text{old}}} in environment for T timesteps
        Compute advantage estimates A_{1}, ..., A_{T}
        Optimize surrogate L wrt \theta, with K epochs and minibatch size M
\leq NT
        \theta_{\text{old}} \leftarrow \theta
    end for
end for
```

The PPO algorithm's step-by-step procedure ensures that policy updates are both stable and effective. By combining clipped objectives and advantage estimation techniques, PPO strikes a balance between efficient exploration and safe exploitation, enabling robust policy optimization across various reinforcement learning environments.

Improvements and Variations in PPO

1. **Policy Feedback**: The integration of policy feedback into the value function update results in more accurate value estimations. By incorporating policy feedback into the advantage calculations, the Proximal Policy Optimization with Policy Feedback (PPO-PF) achieves faster convergence and improved reward performance (Gu et al., 2022).
2. **Model-Based PPO**: Combining PPO with model-based learning introduces the ability to learn and utilize a forward model of the environment. This hybrid approach, called Probabilistic Inference via PPO (PIPPO), enhances sample efficiency by enabling agents to perform more accurate policy improvements through a model-based approach (Sun et al., 2019).
3. **Safe Learning**: Penalized and augmented versions of PPO, like Penalized Proximal Policy Optimization (P3O) and Augmented PPO (APPO), incorporate safety constraints. These algorithms ensure that policies avoid risky actions by applying penalties or augmentations to limit actions that violate safety constraints, making them ideal for applications that demand safe learning (Zhang et al., 2022) (Dai et al., 2023).

3. Proximal Policy Optimization Implementation

The provided code illustrates a practical implementation of Proximal Policy Optimization (PPO), focusing on reproducibility and robustness. Below is an overview of the key components and their role in the agent's training process:

Dependencies and Utilities

1. **Library Imports and Dependencies**:
   The code imports essential libraries for reinforcement learning, neural network modeling, and visualization. It includes a function to handle dependencies for environments like Google Colab, ensuring compatibility and consistent setup.
2. **Random Seed Setting**:
   The function `set_random_seed` ensures reproducibility by setting seeds for NumPy, Python's random module, and PyTorch.

Neural Networks

1. **Actor and Critic Models**:
   The code defines two neural networks:
   - **Actor** (policy network): Predicts the next action using a normal distribution. It outputs both the mean and standard deviation to sample the action.
   - **Critic** (value network): Estimates the state value to help with advantage estimation.

Advantage Estimation and Training

1. **Generalized Advantage Estimation (GAE)**:
   The function `compute_gae` computes advantage values using Generalized Advantage Estimation, balancing bias and variance through the parameters gamma and tau.
2. **PPO Iteration (Mini-Batching)**:
   The `ppo_iter` function organizes the data into mini-batches for training, improving sample efficiency and stability.

PPO Agent Implementation

1. **Agent Design**:
   The `PPOAgent` class includes core elements required for training an agent:
   - **Memory**: Buffers to hold states, actions, rewards, values, and log probabilities.
   - **Actor and Critic Optimizers**: Adam optimizers are used to train the networks.
2. **Action Selection**:
   The agent samples actions through the actor network while storing relevant data for learning during training.
3. **Training Process**:
   - The agent runs the environment for a specified number of steps, storing interactions for training.
   - `update_model` computes losses for the actor (policy) and critic (value function) networks, applying clipping and entropy adjustments to stabilize training.

Results Visualization and Testing

1. **Training Progress Visualization**:
   The `_plot` method visualizes scores and loss curves to evaluate training progression.
2. **Testing**:
   The `test` method allows the agent to showcase learned behavior by interacting with the environment without updating its parameters.

4. Experiment

This experiment investigates the ability of a Proximal Policy Optimization (PPO) agent to learn a control policy for a simulated pendulum arm. The experiment leverages the OpenAI Gym toolkit for environment interaction and utilizes Python for code execution.

Environment: The experiment utilizes the Pendulum-v1 environment from OpenAI Gym. This environment simulates a pendulum arm with a single point of rotation. The agent receives rewards based on its ability to keep the pendulum upright.

Agent: The experiment employs a PPO agent, specifically a class inheriting from `PPOAgent`. The agent interacts with the environment, taking actions based on its current observations and learning from the resulting rewards over time.

Training Parameters:

- **Environment ID:** Pendulum-v1
- **Random Seed:** 529 (set for experiment reproducibility)
- **Number of Frames:** 100,000 (represents the total number of interactions between the agent and the environment)
- PPO Agent Parameters:
    - Gamma: 0.9 (discount factor for future rewards)
    - Tau: 0.8 (importance weight for policy clipping in PPO)
    - Batch Size: 64 (number of experiences used to update the agent in each iteration)
    - Epsilon: 0.2 (exploration noise added to actions during training)
    - Epoch: 64 (number of times the entire collected experience is used for updates)
    - Rollout Length: 2048 (number of steps simulated during each policy evaluation)
    - Entropy Weight: 0.005 (weight for promoting exploration during training)

Environment: Pendulum-v1

In the provided Proximal Policy Optimization (PPO) implementation, the "Pendulum-v1" environment from the OpenAI Gym library is utilized. This environment is specifically designed to test the agent's ability to balance and control a simple pendulum. Here are some key details:

1. **Problem Description**:
    - The objective of the Pendulum environment is to stabilize a pendulum upright while minimizing the energy usage.
    - The pendulum starts in a random position, and the agent must learn to balance it as efficiently as possible.
2. **Observation Space**:
    - The state space is continuous, consisting of three values:
        - $\cos(\theta)$: Cosine of the pendulum's angle.
        - $\sin(\theta)$: Sine of the pendulum's angle.
        - $\dot{\theta}$: Angular velocity of the pendulum.
    - This provides enough information for the agent to understand the pendulum's current position and movement.
3. **Action Space**:
    - The action space is continuous, allowing the agent to apply a torque in the range o $[-2, 2]$ to the pendulum.
    - This torque controls the speed and direction in which the agent can attempt to swing or balance the pendulum.

4. **Reward Function**:
    - The reward is calculated based on the angle deviation from the upright position and the torque used.
    - A penalty is applied for using more torque, encouraging the agent to solve the problem with minimal effort.
5. **Action Normalization**:
    - The `ActionNormalizer` wrapper rescales and normalizes the actions to ensure the agent can adapt its outputs to the continuous range required by the environment.

Experiment Goals

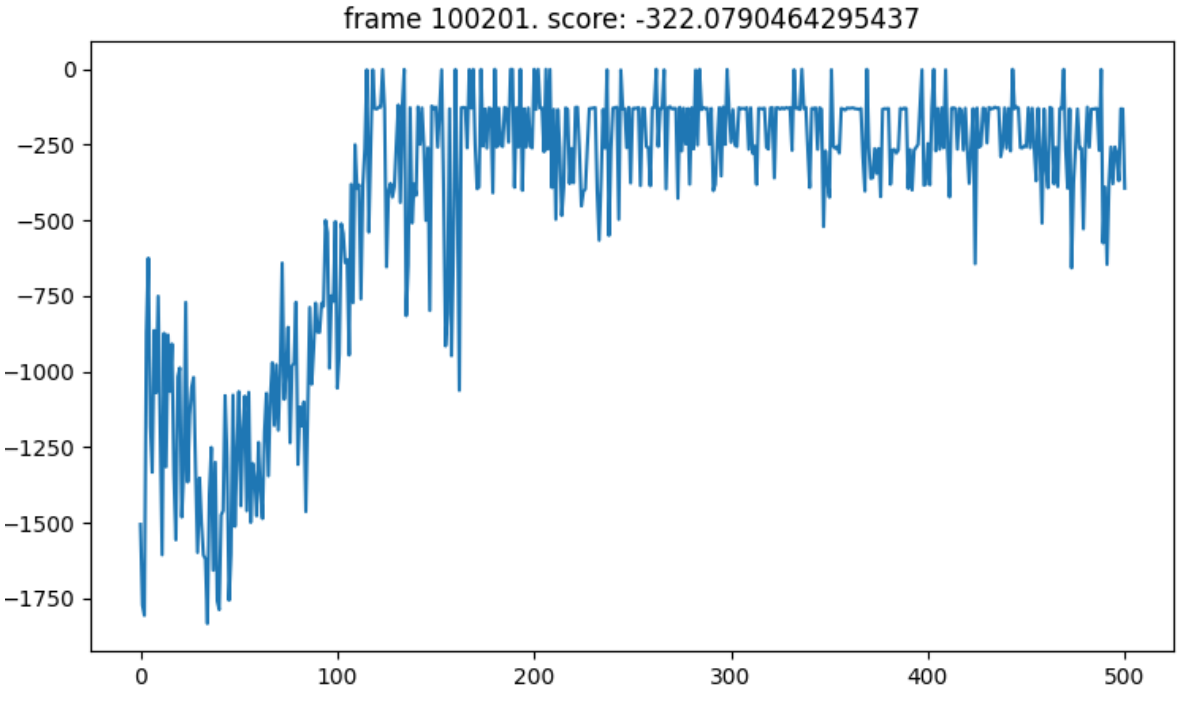
The primary goal of this experiment is to train the PPO agent to develop a policy that effectively controls the simulated pendulum arm. This translates to maximizing the reward received by the agent, which corresponds to keeping the pendulum upright for extended periods.

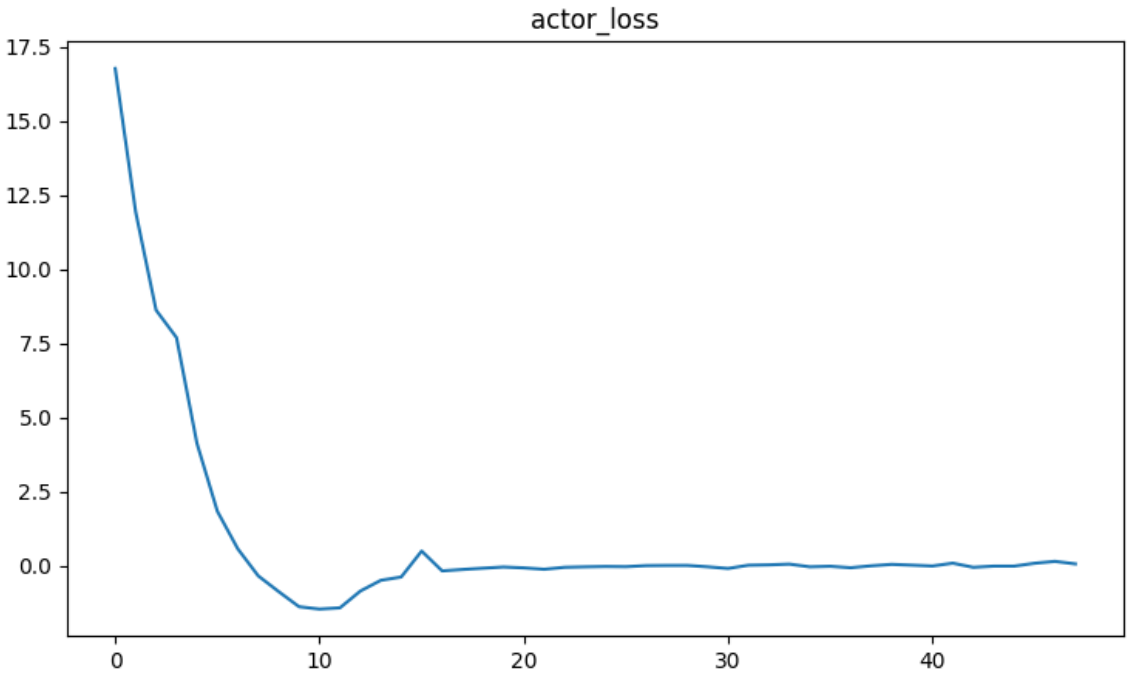
Experiment Results

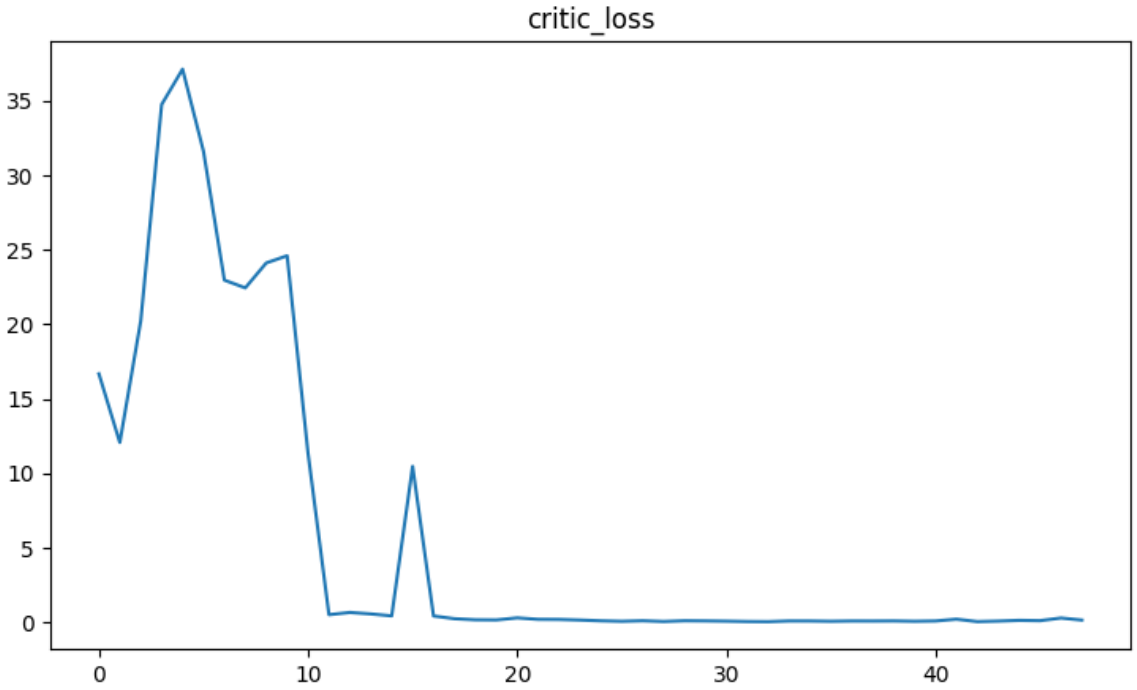
The agent's performance is measured by the reward it receives, with higher rewards indicating the ability to keep the pendulum upright for longer durations.

The graph in Figure 1 shows the agent's performance over a small snippet of its training process. The x-axis represents the number of training frames (each frame represents a single interaction between the agent and the environment), and the y-axis represents the reward achieved by the agent.

The graph suggests that the agent's reward is steadily increasing throughout the training period. This is a positive indication of learning, and it suggests the agent is developing an effective policy for controlling the pendulum.







5. Conclusion

The implementation effectively demonstrates how Proximal Policy Optimization can be applied to continuous action spaces using the Pendulum environment. The architecture ensures that the agent adapts and learns to control its actions while maintaining stable policy updates through clipping, GAE, and appropriate mini-batch training.