Policy Gradients (RL)

Note: You can view more implementation details and exact changes in this readme file:
https://github.com/LukaMeladze/CP2107/blob/main/hw2/README.md

1 Policy Gradients

Reinforcement Learning (RL) aims to find an optimal policy π_θ that maximizes the expected return $J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta(\tau)}[r(\tau)]$, where τ represents a trajectory and $r(\tau)$ is the total reward of that trajectory. Policy gradient methods directly optimize this objective by computing its gradient with respect to the policy parameters θ. The general form of the policy gradient is given by:

$$\nabla_\theta J(\theta) = E_{\tau\sim p_\theta(\tau)}[\nabla_\theta \log p_\theta(\tau) r(\tau)]$$

In practice, this expectation is approximated using a batch of N sampled trajectories:

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\left[\left(\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t})\right)\left(\sum_{t=1}^{T} r(s_{i,t},a_{i,t})\right)\right]$$

1.1 Variance Reduction Techniques (Implemented)

- **Reward-to-Go:** This technique exploits causality: the policy cannot affect rewards in the past. The sum of rewards is modified to only include rewards from the current timestep onwards, acting as a sample estimate of the Q-function, referred to as the "reward-to-go".

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=0}^{T-1}\nabla_\theta log\ \pi_\theta(a_{it}|s_{it})(\sum_{t'=t}^{T-1} r(s_{it'},a_{it'}))$$

- **Discounting:** A discount factor $\gamma \in [0,1]$ is applied to rewards, encouraging the agent to focus more on closer rewards. This can also reduce variance. The discount factor can be applied to the full trajectory return or the reward-to-go.
  - Full Trajectory Discounting (Case 1 in code):

$$r(\tau_i) = \sum_{t'=1}^{T}\gamma^{t'} r(s_{it'},a_{it'})$$

  - Reward-to-Go Discounting (Case 2 in code):

$$r(\tau_i) = \sum_{t'=t}^{T}\gamma^{t'-t} r(s_{it'},a_{it'})$$

2 Environment Background: CartPole-v0

CartPole-v0 is a classic control problem in OpenAI Gym. The goal is to balance a pole on a cart by moving the cart left or right. A reward of +1 is given for every timestep the pole remains upright. The episode terminates if the pole falls beyond a certain angle, or the cart moves too far from the center, or after 200 timesteps. The action space is discrete (left or right).

3 Implementation Details

My changes focus on completing the `TODO` sections in `run_hw2.py`, `pg_agent.py`, `utils.py`, and `policies.py`. You can view more details about the exact changes on this readme file:
https://github.com/LukaMeladze/CP2107/blob/main/hw2/README.md

4 Policy Gradients

- Create two graphs:
  - In the first graph, compare the learning curves (average return vs. number of environment steps) for the experiments prefixed with `cartpole`. (The small batch experiments.)
  - In the second graph, compare the learning curves for the experiments prefixed with `cartpole_lb`. (The large batch experiments.)

Small Batch Experiments (Average Return vs. Environment Steps)

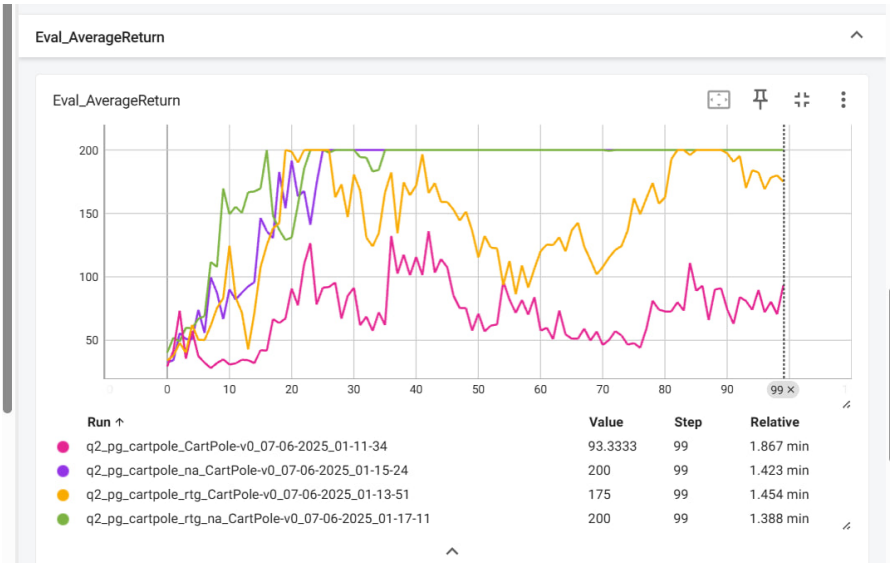

Figure 1: Graph 1: Small Batch Experiments

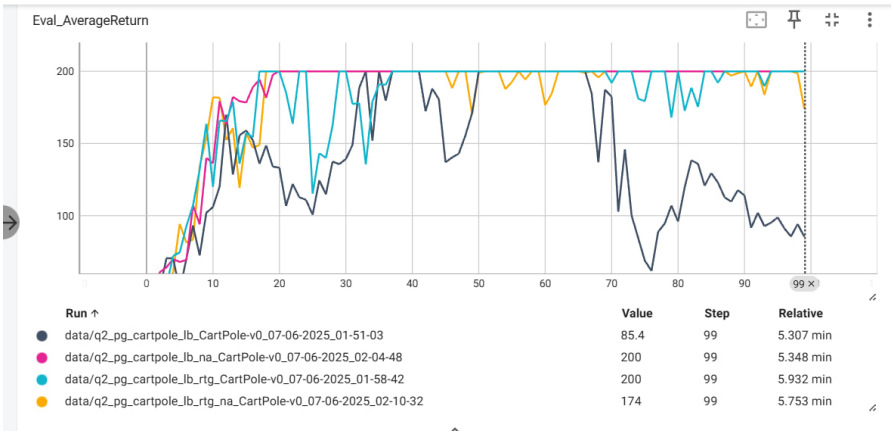

Figure 2: Graph 2: Large Batch Experiments, $\gamma = 0.95$

- Brief general summary of results:
  - The reward-to-go estimator generally shows better performance. This is because reward-to-go reduces variance by only considering future rewards, which is a more accurate estimate of the Q-function for the current state-action pair compared to the full trajectory return.

- Advantage normalisation generally also helps. It stabilizes training by ensuring that the advantages have a consistent scale, preventing large gradients that could destabilise the policy update.
- Larger batch sizes (4000) generally lead to more stable and consistent learning, as they provide a more accurate estimate of the policy gradient, but it really depends on parameters, so different seeds, discount factors, etc. need to be experimented with for more robust results. However, a large batch may also require more environment steps per iteration. For CartPole, both batch sizes with good variance reduction techniques converge to a score of 200.

Command Line Configurations:

Small Batch Experiments (batch_size=1000):

```
python cs285/scripts/run_hw2.py --env_name CartPole-v0 -n 100 -b 1000 --exp_name cartpole
python cs285/scripts/run_hw2.py --env_name CartPole-v0 -n 100 -b 1000 -rtg --exp_name cartpole_rtg
python cs285/scripts/run_hw2.py --env_name CartPole-v0 -n 100 -b 1000 -na --exp_name cartpole_na
python cs285/scripts/run_hw2.py --env_name CartPole-v0 -n 100 -b 1000 -rtg -na --exp_name cartpole_
```

Large Batch Experiments (batch_size=4000):

```
python cs285/scripts/run_hw2.py --env_name CartPole-v0 -n 100 -b 4000 --exp_name cartpole_lb
python cs285/scripts/run_hw2.py --env_name CartPole-v0 -n 100 -b 4000 -rtg --exp_name cartpole_lb_r
python cs285/scripts/run_hw2.py --env_name CartPole-v0 -n 100 -b 4000 -na --exp_name cartpole_lb_na
python cs285/scripts/run_hw2.py --env_name CartPole-v0 -n 100 -b 4000 -rtg -na --exp_name cartpole_
```