Homework 1: Policy Gradient

Krishna Khadka

1001751624

Github Link: https://github.com/krishnakhadka200416/CSE6369

(Please let me know for access. I've kept it private for now.)

Experiment I (CartPole)

1. Snapshots of code that show implementation of eq 7, 8, 6.

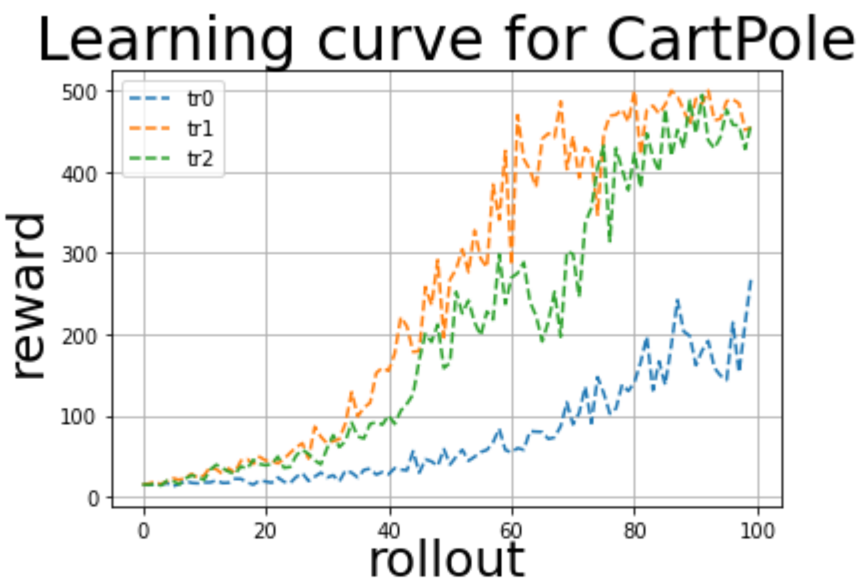
```
# Util function to apply reward-to-go scheme on a list of instant-reward (from eq 7)
def apply_reward_to_go(raw_reward):
    # TODO: Compute rtg_reward (as a list) from raw_reward
    # HINT: Reverse the input list, keep a running-average. Reverse again to get the correct order.
    s = 0
    rtg_reward = []
    for i in reversed(raw_reward):
        s += i
        rtg_reward.append(s)
    rtg_reward = np.array(rtg_reward)
    rtg_reward = np.flip(rtg_reward)

    # Normalization
    rtg_reward = np.array(rtg_reward)
    rtg_reward = rtg_reward - np.mean(rtg_reward) / (np.std(rtg_reward) + np.finfo(np.float32).eps)
    return torch.tensor(rtg_reward, dtype=torch.float32, device=get_device())


# Util function to apply reward-discounting scheme on a list of instant-reward (from eq 8)
def apply_discount(raw_reward, gamma=0.99):
    # TODO: Compute discounted_rtg_reward (as a list) from raw_reward
    # HINT: Reverse the input list, keep a running-average. Reverse again to get the correct order.
    discounted_rtg_reward = []
    s = 0
    for i in reversed(raw_reward):
        s = i + gamma * s
        discounted_rtg_reward.append(s)
    discounted_rtg_reward = np.array(discounted_rtg_reward)
    discounted_rtg_reward = np.flip(discounted_rtg_reward)


    # Normalization
    discounted_rtg_reward = np.array(discounted_rtg_reward)
    discounted_rtg_reward = discounted_rtg_reward - np.mean(discounted_rtg_reward) / (np.std(discounted_rtg_reward) +
np.finfo(np.float32).eps)
    return torch.tensor(discounted_rtg_reward, dtype=torch.float32, device=get_device())


# Util function to apply reward-return (cumulative reward) on a list of instant-reward (from eq 6)
def apply_return(raw_reward):
    # Compute r_reward (as a list) from raw_reward
    r_reward = [np.sum(raw_reward) for _ in raw_reward]
    return torch.tensor(r_reward, dtype=torch.float32, device=get_device())
```

Fig:- Implementation in utils.py

```python
    def estimate_loss_function(self, trajectory):
        loss = list()
        for t_idx in range(self.params['n_trajectory_per_rollout']):
            # TODO: Compute loss function
            # HINT 1: You should implement eq 6 (Vanilla Policy Gradient), 7(Reward to Go) and 8(Reward Discounting) here.
Which will be used based on the flags set from the main function

            rewards = trajectory["reward"][t_idx]
            # HINT 2: Get trajectory action log-prob
            log_prob = trajectory["log_prob"][t_idx]


            if self.params['reward_to_go']:
                computed_reward = apply_reward_to_go(rewards)


            elif self.params['reward_discount']:
                computed_reward = apply_discount(rewards)


            else:
                computed_reward = apply_return(rewards)
```

Fig:- Using eq 6, 7, 8 in learning_algorithms.py

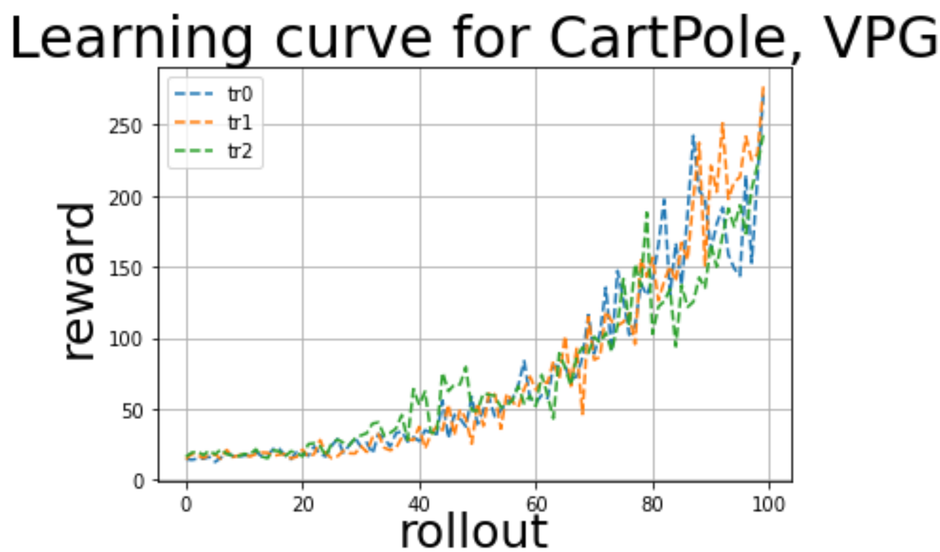
2. Graph that compares the learning curve of three trials.
    a. T0 -- VPG
    b. T1 – Reward to Go
    c. T2 -- Reward Discounting



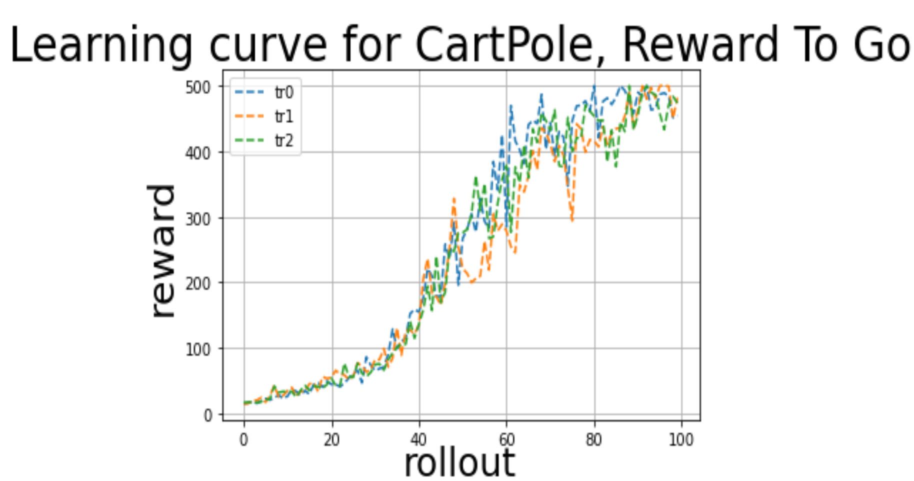
3.
    a. The reward-to-go method (implemented in Tr1) performed the best among the three policy gradient methods tested. This suggests that it's a powerful approach for problems with a clear sequence of actions. The discounted reward method may be more suitable for problems with delayed or sparse rewards, and return-based can serve as a baseline but may not perform as well in more complex environments.

b. Bonus

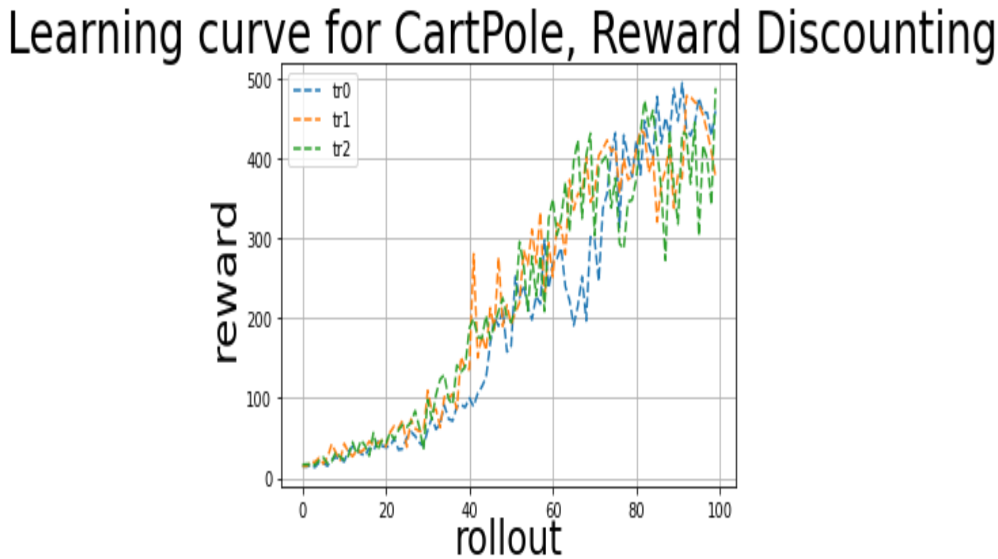
1. Random seed of 30, 40, and 6369 on return-based



2. Random seed of 30, 40, and 6369 on Reward to Go



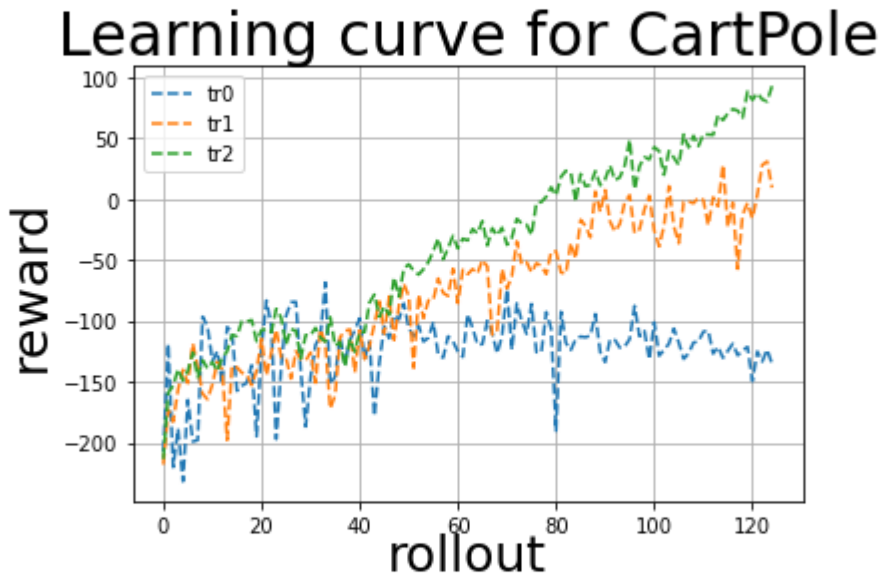
3. Random seed of 30, 40, 6369 on Reward Discounting



From the above charts, we can conclude that reward-to-go has higher variance, whereas return-based has the lowest variance.

Experiment II (LunarLander)

1. Graph comparison of the three trials.



2. a. Based on the chart, it appears that there is a positive correlation between the number of trajectories per rollout and the reward obtained. Specifically, the results show that Tr2 (which used 60 trajectories per rollout) achieved the highest reward, followed by Tr1 (with 20 trajectories per rollout) and Tr0 (with 5 trajectories per rollout). This suggests that increasing the number of trajectories can lead to better performance in policy gradient methods.