CSE 6369 – Special Topics in Advance Artificial Intelligence

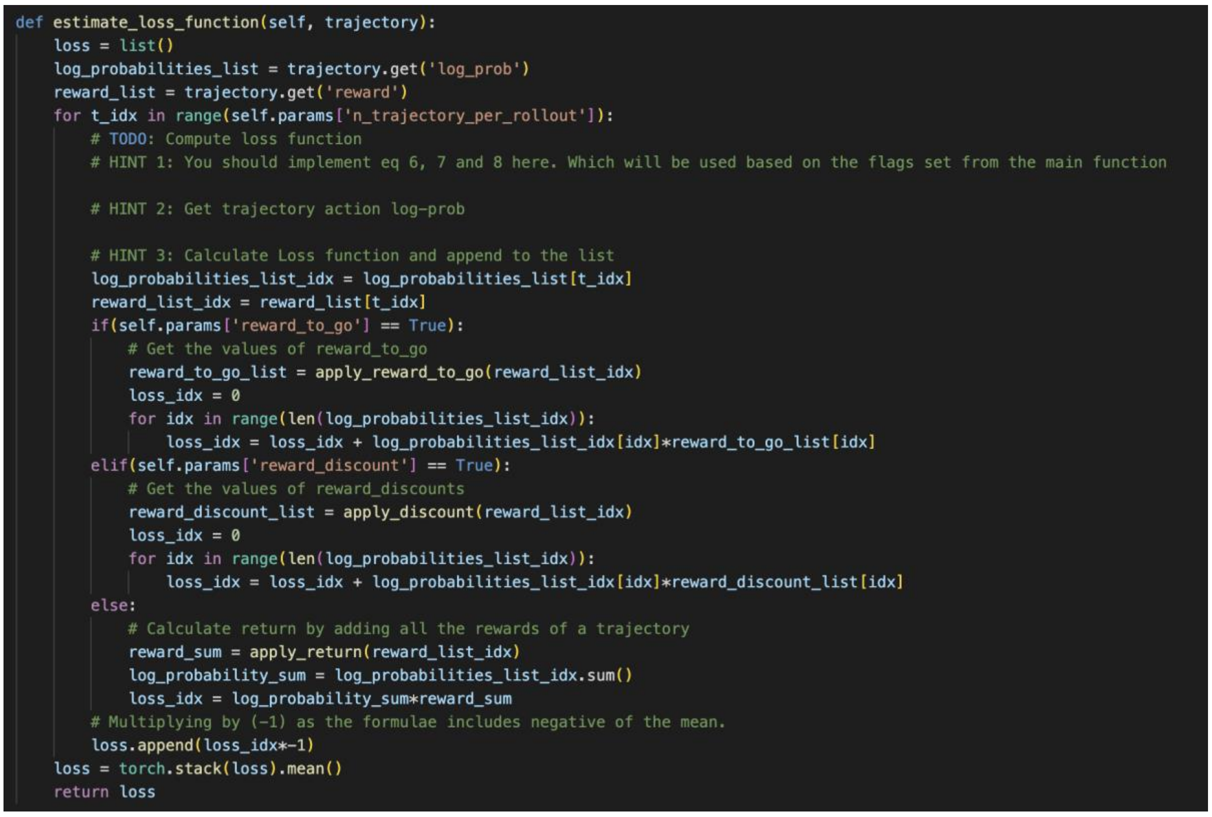
Pavan Tejaa Dharmisetty
UTA ID: 1002023651

Introduction:

This report briefly describes about the implementation of the *Policy Gradient (PG)* algorithm and its variations. The implemented PG algorithm is applied in two experiments *CartPole* and *LunarLander*, and the implemented variants are compared with each other in terms of learning curves and variance.

Implementation:

The loss function is derived as the negative mean of sum of the product of log-probabilities sum and reward sum.

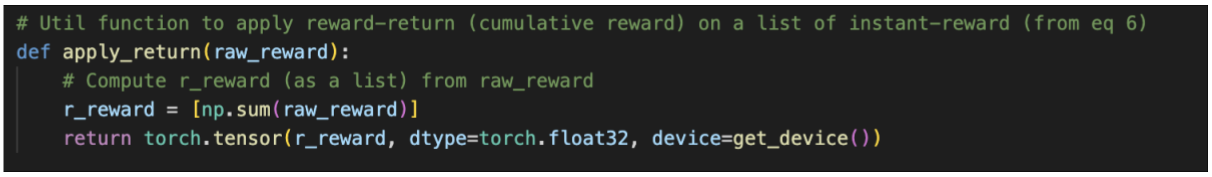


```
def estimate_loss_function(self, trajectory):
    loss = list()
    log_probabilities_list = trajectory.get('log_prob')
    reward_list = trajectory.get('reward')
    for t_idx in range(self.params['n_trajectory_per_rollout']):
        # TODO: Compute loss function
        # HINT 1: You should implement eq 6, 7 and 8 here. Which will be used based on the flags set from the main function

        # HINT 2: Get trajectory action log-prob

        # HINT 3: Calculate Loss function and append to the list
        log_probabilities_list_idx = log_probabilities_list[t_idx]
        reward_list_idx = reward_list[t_idx]
        if(self.params['reward_to_go'] == True):
            # Get the values of reward_to_go
            reward_to_go_list = apply_reward_to_go(reward_list_idx)
            loss_idx = 0
            for idx in range(len(log_probabilities_list_idx)):
                loss_idx = loss_idx + log_probabilities_list_idx[idx]*reward_to_go_list[idx]
        elif(self.params['reward_discount'] == True):
            # Get the values of reward_discounts
            reward_discount_list = apply_discount(reward_list_idx)
            loss_idx = 0
            for idx in range(len(log_probabilities_list_idx)):
                loss_idx = loss_idx + log_probabilities_list_idx[idx]*reward_discount_list[idx]
        else:
            # Calculate return by adding all the rewards of a trajectory
            reward_sum = apply_return(reward_list_idx)
            log_probability_sum = log_probabilities_list_idx.sum()
            loss_idx = log_probability_sum*reward_sum
        # Multiplying by (-1) as the formulae includes negative of the mean.
        loss.append(loss_idx*-1)
    loss = torch.stack(loss).mean()
    return loss
```

The reward is calculated in 3 different ways, as shown below, and one among three will be used in calculating the loss function based on the user-given flag.

- **Return**

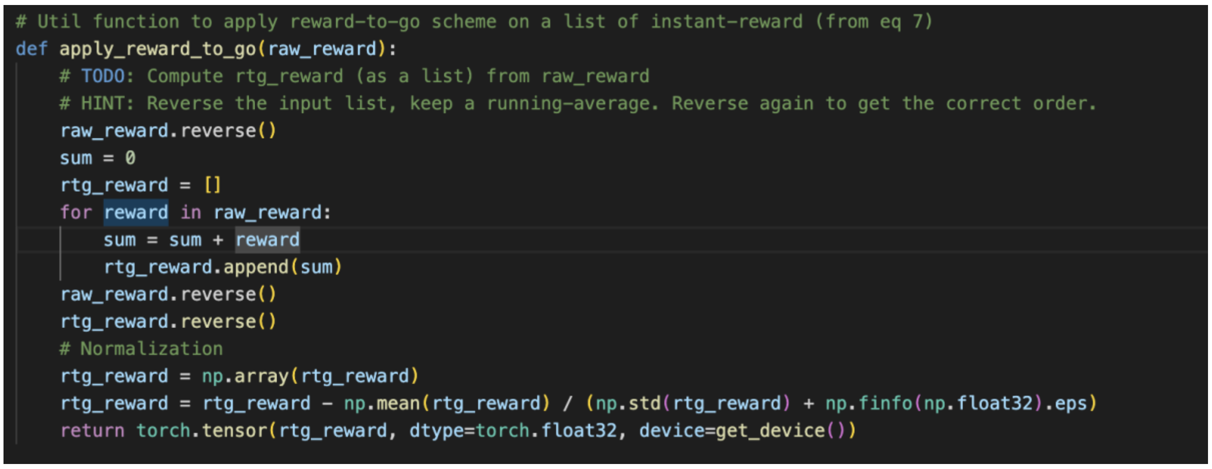
  This is equivalent to the sum of all the rewards obtained in a trajectory. The implementation is given below:



```python
# 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)]
    return torch.tensor(r_reward, dtype=torch.float32, device=get_device())
```

- **Reward-to-go:**

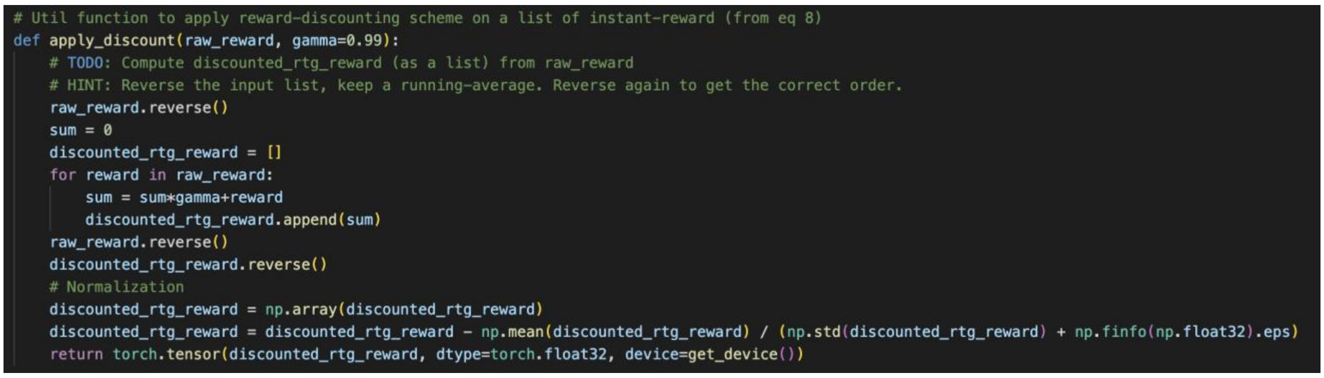
  Here, the reward-to-go of a trajectory for an action a_t is calculated as the sum of rewards from that time t to the end of the trajectory. Its implementation is given below:



```python
# 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.
    raw_reward.reverse()
    sum = 0
    rtg_reward = []
    for reward in raw_reward:
        sum = sum + reward
        rtg_reward.append(sum)
    raw_reward.reverse()
    rtg_reward.reverse()
    # 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())
```

- **Reward Discount:**

  In this method, a discounting factor γ will be multiplied to the reward adding more weight on near-future rewards compared to the far-future rewards. This helps sum-of-reward to converge better to a lesser-variance distribution. The implementation is given below:



```python
# 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.
    raw_reward.reverse()
    sum = 0
    discounted_rtg_reward = []
    for reward in raw_reward:
        sum = sum*gamma+reward
        discounted_rtg_reward.append(sum)
    raw_reward.reverse()
    discounted_rtg_reward.reverse()
    # 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())
```

Experiment 1 (CartPole):

In this experiment by gymnasium, a pole is attached by an un-actuated joint to a cart, which moves along a frictionless track. The pendulum is placed upright on the cart, and the goal is to balance the pole by applying forces in the left and right direction on the cart. The above-implemented policy is trained in this environment and the action determined by the policy is applied to the state in the environment and a reward is received. The agent updates the policy based on the rewards it received by the above-implemented methods.

The below three trials are executed with the agent on the CartPole-v1 environment,

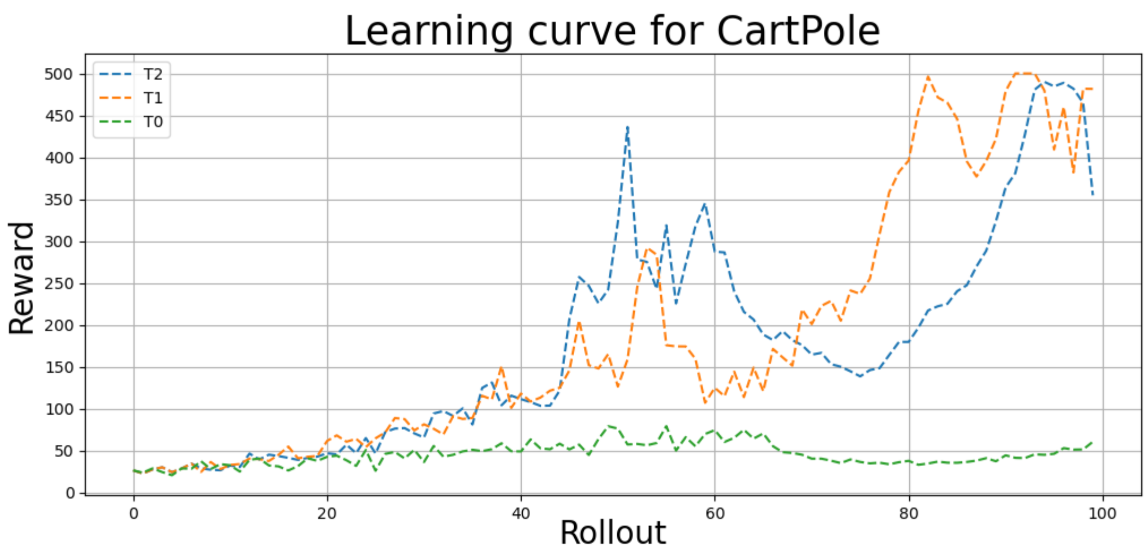
- **T0: python main hw1.py -e CartPole-v1 -nr 100 -ntr 10 -hdim 64 -lr 3e-3 -xn CartPole v1 t0**
- **T1: python main hw1.py -e CartPole-v1 -rtg -nr 100 -ntr 10 -hdim 64 -lr 3e-3 -xn CartPole v1 t1**
- **T2: python main hw1.py -e CartPole-v1 -rd -nr 100 -ntr 10 -hdim 64 -lr 3e-3 -xn CartPole v1 t2**

A graph is plotted to compare the learning curves from the three trials,

T0 – Total Return

T1 – Reward-to-go

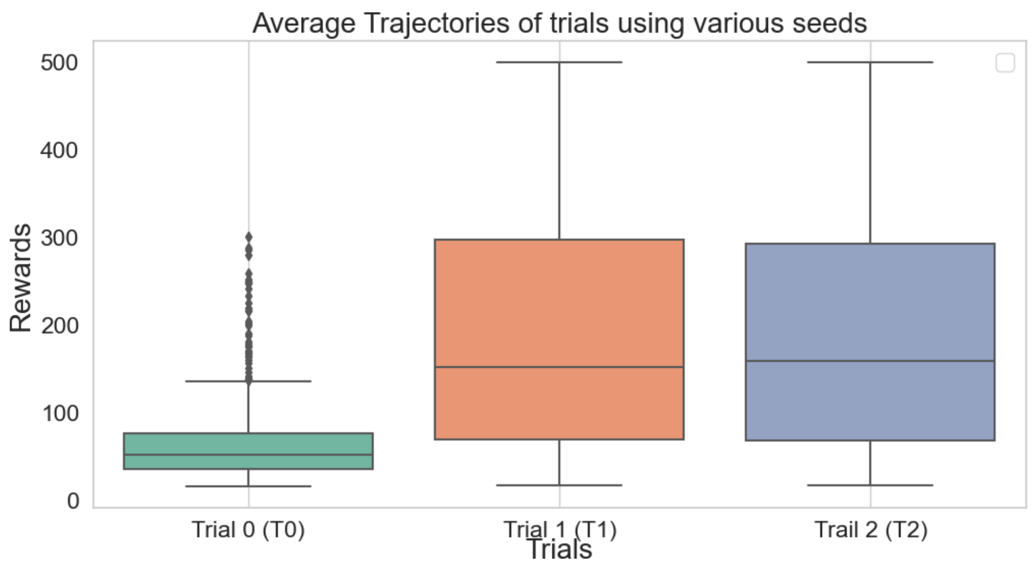
T2 – Reward Discount



From the above graph, it is clear that **T1 and T2** performs better compared to **T0**. But in this particular experiment ,the learning rate of both T1 and T2 are so close and mostly similar. But as we observe, the increase in reward achieved quick in **T2** compared to **T1**. So, according to this graph, it can be asserted that **discounted-reward-based is slightly better than reward-to-go.**

BONUS:

In order to compare the variance among three implementations T0 (Total Return), T1 (Reward-to-go), T2(Discounted Reward), I have ran each trial with 5 different seeds and combined all the average trajectory rewards of each trial (i.e. 100*5 = 500 avg trajectpry rewards per trial). The three lists of average trajectory rewards are compared to check the variance below,



As we can see from the above graph, **T0** has lower variance compared to the other two.

Experiment 2 (LunarLander):

In this experiment by the gymnasium, the environment is a classic rocket trajectory optimization problem. After every step a reward is granted. The total reward of an episode is the sum of the rewards for all the steps within that episode.

The above-implemented policy is trained in this environment and the action determined by the policy is applied to the state in the environment and a reward is received. The agent updates the policy based on the rewards it received by the above-implemented methods.

The below three trials are executed with the agent on the LunarLander-v2 environment,

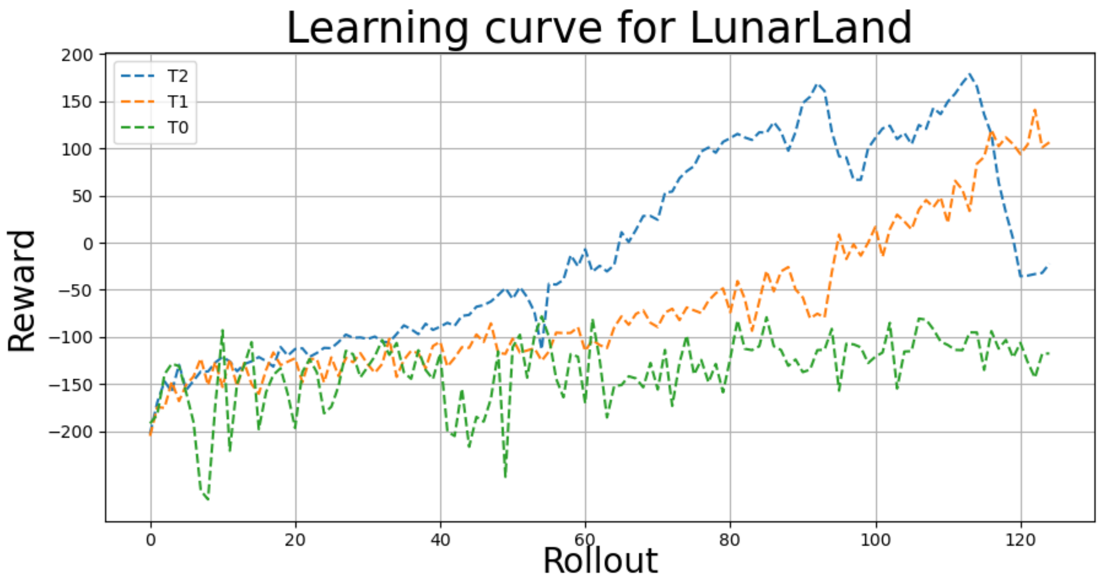
- **T0: python main hw1.py -e LunarLander-v2 -rd -nr 125 -ntr 5 -hdim 128 -lr 3e-3 -xn LunarLand v2 t0**
- **T1: python main hw1.py -e LunarLander-v2 -rd -nr 125 -ntr 20 -hdim 128 -lr 3e-3 -xn LunarLand v2 t1**
- **T3: python main hw1.py -e LunarLander-v2 -rd -nr 125 -ntr 60 -hdim 128 -lr 3e-3 -xn LunarLand v2 t2**

A graph is plotted to compare the learning curves from the three trials,

T0 – 5 trajectories per rollout

T1 – 20 trajectories per rollout

T2 – 60 trajectories per rollout



As we can see from the above graph, **the higher the number-of-trajectories-per-rollout is, higher the learning rate will be**. Since T2 has 60 trajectories per rollout, its learning is better compared to others.

Git Repository:

https://github.com/PavanTejaa/cse6369_assignment1.git

References:

https://gymnasium.farama.org/environments/classic_control/cart_pole/#cart-pole
https://gymnasium.farama.org/environments/box2d/lunar_lander/
https://pytorch.org/docs/stable/distributions.html
https://gymnasium.farama.org/api/env/#gymnasium.Env.step
https://pytorch.org/docs/stable/generated/torch.tensor.html