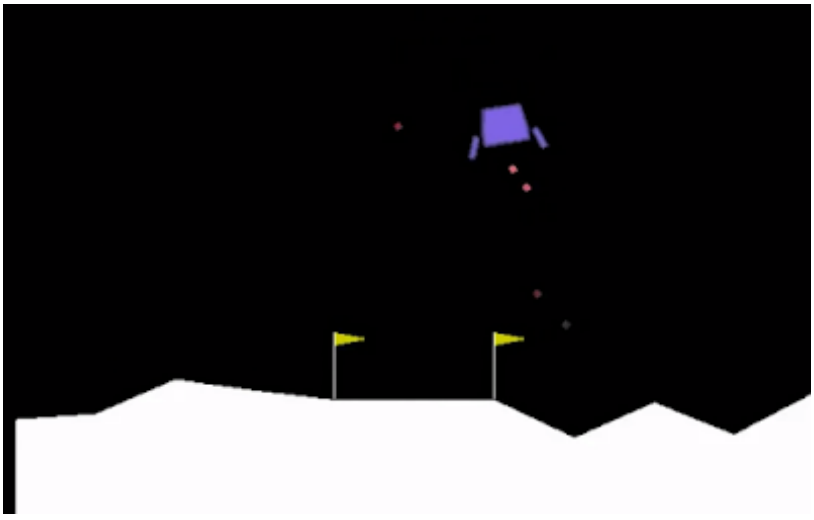
Probabilistic Artificial Intelligence Task 4: Reinforcement Learning

1 Task Description

Your task is to control a lunar lander in a smooth descent to the ground. It must land in-between two flags, quickly, using minimal fuel, and without damaging the lander. Implement a reinforcement learning algorithm that, by practicing on a simulator, learns a control policy for the lander.

This '.pdf' gives task-specific information. For logistics on submission and setup, see the description on the task webpage. In this document, we will first explain the task environment and scoring. Second, we will outline the minimal changes ('TODO's) needed to pass the baseline, before giving some ideas for extensions. Finally, we will provide some supplementary material on policy gradients to help you get started on the necessary 'TODO's.

2 Environment and Scoring Details



At each discrete time step, the controller can maneuver the lander by taking one of four actions: doing nothing, firing the left engine, firing the right engine, or firing the main engine. The goal is to accumulate as much reward as possible in 300 timesteps. In evaluation, after 300 timesteps the episode ends. Positive reward is obtained for landing between the flags and landing on the lander's legs. Negative reward is obtained for firing the main engine or side engines (more negative for the main engine) or for crashing the lander. Note that the lander only obtains positive reward for landing on its legs: if you land so fast that the legs hit the ground followed by the main lander body, it is counted as only crashing. Since the focus of this task is the implementation of reinforcement learning algorithms, it is not necessary to have a detailed understanding of the mechanics of the lunar lander environment beyond the observation and action space sizes (given in the train function of 'solution.py').

When you run 'solution.py', your algorithm will have access to the standard lunar lander environment that is commonly used in evaluating RL algorithms. To run 'solution.py' you will need to install the packages listed in 'requirements.txt'. You are encouraged to run 'solution.py' for testing.

In a single run of 'runner.sh', you will be able to query up to 150000 transitions (single timepoints) in order to learn a policy for the modified lunar lander environment. Each individual episode can contain up to 300 transitions. The score is then based upon the average performance of the learned policy over 100 episodes after the training episodes. The final score will be

$$\frac{1}{100}\sum_{i=1}^{100}\sum_{t=1}^{T} R_t^i.$$

where R_t^i is the reward achieved at time step t, episode i on the modified version of the environment, when using your final policy. In other words, the final score is an estimate of the expected cumulative reward of your final policy over an episode. Your goal is to maximize this final score.

The maximum possible score is around 200.

3 Solution Details

We give you skeleton code that provides most of the infrastructure for policy gradient algorithms. The 'TODO's in the code will guide you through implementing vanilla policy gradients with Generalized Advantage Estimation. It is possible to pass the baseline by correctly implementing policy gradients with the suggested modifications and not changing the hyperparameters we provide.

Below we list all the 'TODO's for this task. You only have to make changes in 'solution.py'.

- Implement the function 'mlp', which returns a torch neural network module, i.e., a multilayer perceptron, according to specified layer sizes, activation function for the hidden layer, and activation function for the output layer.
- Implement the '_distribution', '_log_prob_from_distribution', and 'forward' functions of 'Actor' class. The '_distribution' function takes as input state observations and returns the policy distributions for the states. The '_log_prob_from_distribution' function takes as input a policy distribution 'pi' and an action, and returns the log probability of the action under the policy distribution 'pi'. The 'forward function takes a state observation and action, and returns the policy distribution 'pi' for the state as well as the log probability of the action under the distribution 'pi'.
- We provide you with a data buffer class 'VPGBuffer'. Implement the 'store' method which is called to store a new transition to the buffer, and the 'end_traj' method which calculates and stores in the buffer the cost-to-go and TD residuals at the of an episode.
- The 'Agent' class consists of an actor and a critic. For this class, please implement the 'step' method which returns for a given state, an action sampled from the policy/actor, the value function, and the log probability of the action under the given policy output distribution. Please also implement the 'get_action' method which samples and returns an action from the policy/actor.
- Finally, please implement the policy and critic updates in the 'train' function.

All the 'TODO's are also marked in the skeleton code. Our recommendation is to read the section below on rewards-to-go and the original paper introducing Generalized Advantage Estimation; read the skeleton code carefully; and then implement the TODOS. You can pass the baseline without modifying anything outside of the TODOs. The skeleton code makes use of Pytorch. After passing the baseline, to score higher on the leaderboard and learn more about reinforcement learning you are encouraged to implement additional improvements. For example

- Try different neural network architectures and hyperparameter choices.
- Other variants of policy gradients such as Proximal Policy Optimization.
- Other types of reinforcement learning algorithms such as Q-learning or model-based approaches.

4 Policy Gradients

Recall the policy gradients algorithm seen in class. Notation: superscript is episode number and subscript is timepoint in the episode. So s_t^i is the state at the tth timepoint of the ith episode. We write a generic episode generated by the current policy and environment as τ and the ith episode in our dataset as τ^i. $\tau^i = (s_0^i, a_0^i, r_0^i, s_1^i, a_1^i, r_1^i, \dots)$ and for a generic episode $\tau = (s_0, a_0, r_0, \dots)$.

Expectations are always being taken over the distribution from which a single episode τ is sampled. Therefore the sources of noise are the stochasticity of the policy and the environment.

Result: Optimized policy π_θ
Input: Randomly initialized parameters θ, environment E, stepsize α

while *training* **do**
  generate data $\mathcal{D} = [\tau^0, \tau^1 \dots]$ by interacting with E using π_θ for T timesteps
  $\forall i$ compute $G(\tau^i) = \sum_{t=0} \gamma^t r_t^i$, the discounted cumulative rewards of the ith episode
  $\theta \leftarrow \theta + \alpha \nabla_\theta J(\pi(\theta))$
end

Algorithm 1: The policy gradients algorithm

$\nabla_\theta J(\pi_\theta)$ is the policy gradient and is given by $\nabla_\theta \mathbb{E}[G(\tau)] = \mathbb{E}[\sum_t G(\tau) \nabla_\theta \log \pi_\theta(a_t \mid s_t)]$ (the proof of this equality is not given here). Since the policy gradient is just an expectation over episodes we can estimate it unbiasedly on a finite dataset by computing $\frac{1}{\mathcal{D}} \sum_{i,t} G(\tau^i) \nabla_\theta \log \pi_\theta(a_t^i \mid s_t^i)$.

The above estimate of the policy gradient has high variance for small sample sizes, and obtaining episodes can be time-intensive. Therefore we seek still unbiased but lower variance estimates of the policy gradient. We give two ways of doing this below: rewards-to-go and the use of a baseline via the advantage function. More mathematical details can be found in blog post 1, blog post 2 and Generalized Advantage Estimation. Here we just present a summary and the intuition of the two ideas.

Rewards-to-go is a modification based upon the observation that the policy gradient naively increases the probability of action a_t at state s_t depending upon the total reward accumulated in the episode. However, this includes rewards accumulated before the action was taken, which are not a consequence of the action and hence are just adding noise to our policy gradient. It can be shown that the policy gradient can also be written by only considering rewards obtained after the action was taken

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}[\sum_t R_{t:}(\tau) \nabla_\theta \log \pi_\theta(a_t \mid s_t)]$$

where $R_{t:}(\tau)$ is the discounted sum of rewards obtained in an episode after and including timepoint t. Our new estimate will have lower variance since we have removed some reward terms that were just contributing noise and no signal (unrelated to whether we chose this action or not).

Whilst the basic policy gradient method optimizes the policy directly, one can reduce the variance of gradient updates by maintaining an estimate of the value function. In Generalized Advantage Estimation it's shown that one can actually write the policy gradient as

$$\mathbb{E}[\sum_t \phi_t(\tau) \nabla_\theta \log \pi_\theta(a_t \mid s_t)]$$

where ϕ_t can take numerous possible values and still be equivalent to the policy gradient given above. Examples include $\phi_t = R_{t:}$ as above. Generalized Advantage Estimation gives several other alternatives for ϕ_t. Most of these make some use of a value function, which reduces variance by pooling information across trajectories. Please see the paper for more details on the suggested use of ϕ for this task, the advantage function, and how to estimate it. For estimating the advantage function, note the use of two different discount factors in the paper. Implementing the 'TODO's and using $\gamma = 0.99$ and $\lambda = 0.97$ should beat the baseline for passing the task.