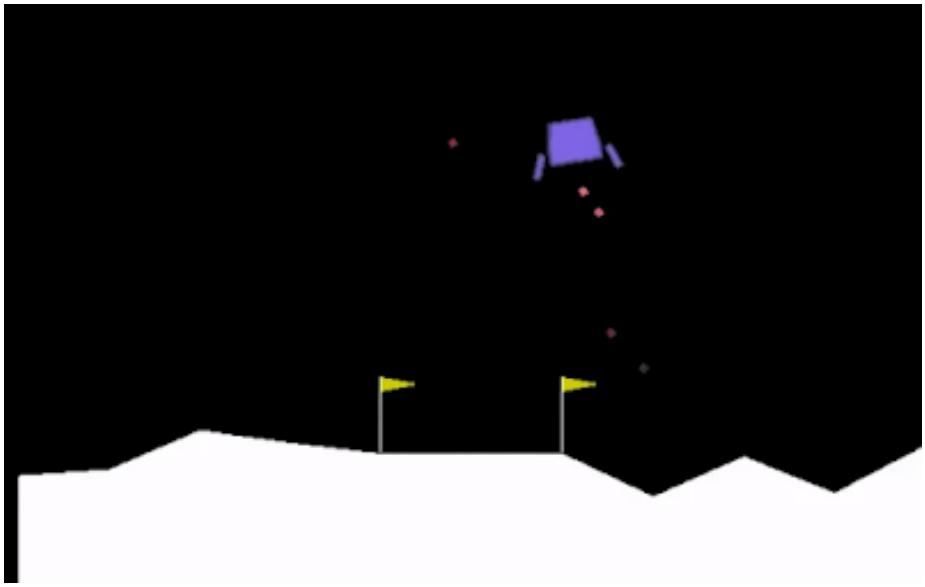
# Probabilistic Artificial Intelligence Project 4: Reinforcement Learning

## Task Description

In the lunar lander environment, a *lander* must be controlled to land between two flags on the ground. When controlling the lander, the goal is to successfully land upright on its two legs, between the flags and in a limited timeframe, whilst expending minimal fuel. Your task is to implement an algorithm that, by practicing on a simulator, learns a control policy for the lander.

The `.pdf` of the handout contains the following additional sections. Environment and Scoring Details will give a more thorough description of the Lunar Lander environment, along with the requirements for passing the baseline. Solution Details will guide you through a set of minimal modifications to `solution.py` for passing the task baseline. We also provide an appendix on policy gradients which will briefly review the lecture material and give you the additional information needed for completing the `TODO`s in the code for this task.

## 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 positive reward for lander is for landing on the legs: if you land so fast that the legs hit the ground followed by the main lander body, it is counted as only crashing. At each time-step, the agent gets an 8-dimensional observation (some floats, some binary), giving the lander's motion and position relative to the flags. Since the purpose of this task is the implementation of reinforcement learning algorithms, it should not be necessary to have a detailed understanding of the mechanics of the lunar lander environment beyond the state and action space sizes.

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`. Feel free to use `solution.py` for testing.

When you run `runner.sh` with docker running, your algorithm will first train. Then, the resulting policy will be evaluated on a modified lunar lander environment. This modified environment will be different every time the code is run and unknown to you. The modified task will always have the same difficulty as the standard lunar lander problem used in `solution.py` and the same size of state and action space. A solution that gets a good score on the default lunar lander environment using the kinds of techniques seen in the lecture should have no problem achieving a similar score on the modified version. When `runner.sh` has finished, a visualization of your final policy in an episode at evaluation time is saved to a `.mp4` file. `runner.sh` will also generate the `.byte` score file you need to upload to the leaderboard.

In a single run of `runner.sh`, you will be able to query up to one million transitions (single timepoints) in order to learn a policy for the modified lunar lander environment. The score is then based upon the average performance of the learned policy after the training episodes. The final score will be

$$\mathbb{E}\sum_{t=1}^{T} R_t$$

where $R_t$ is the reward achieved at time step $t$ on the modified version of the environment, when using your final policy. The expectation is taken over the stochasticity in the environment and in your policy. In other words, the final score is the expected cumulative reward of your final policy over an episode. Your goal is to maximize this final score.

To pass the baseline for this task you will need to achieve a final score of 100.

## Solution Details

We have provided skeleton code that will allow you to implement a simple variant of policy gradients with two additional features. The first is the use of "rewards to go". The second is the use of a generalized advantage estimate

for reducing the variance of policy gradient updates. We also provide you with the implementation of neural networks for computing the policy and the value function estimate. It is possible to pass the baseline by correctly implementing policy gradients with the two modifications and not changing the given network architectures. Concretely, you can do this by reading the skeleton code provided in solution.py, and adding code in the appropriate sections marked `TODO`:

- Implement the `step` function in the `MLPActorCritic` class. Given an observation, this function samples an action from the policy, computes its log-likelihood, and computes the value function estimate of the observation.
- Implement the computation of the TD-residual at each timestep. For more information on what this means see the next section. In a seperate `TODO` later in the code, you should also normalize the TD-residual values. This is an empirical trick which stabilises gradient updates by ensuring that in each batch of updates, the distribution of the TD-residuals is the same.
- Implement the computation of rewards-to-go. For more information on what this means see the next section.
- Use the optimizers and architectures already given to you in order to update the policy and value function neural networks at the end of each epoch.
- Implement `get_action` which is used by your agent to control the lander at evaluation time. It should take in a state and return an action.

We follow the pattern of the evaluation environment and terminate episodes after 300 timesteps. If an epoch has ended before the episode is complete, we use the value function estimate to estimate the remaining return of that episode.

Our recommendation is to read the section below on rewards-to-go and the use of a baseline; read the skeleton code carefully; and then implement the TODOS, testing after each one. You can pass the baseline without modifying anything outside of the TODOs. The skeleton code makes use of Pytorch, but since a lot of the structure is given to you, experience with Pytorch should not be necessary in order to implement what is required to pass the baseline. After passing the baseline, to score higher on the leaderboard you are encouraged to implement additional improvements or a completely different reinforcement learning algorithm (for example Q-learning).

## Appendix on Policy Gradients

Recall the policy gradients algorithm seen in class. Notation: superscript is episode number and subscript is timepoint in the episode. So $s^i_j$ is the state at the $t$th timepoint of the $i$th episode. We write a generic episode generated by the current policy and environment as $\tau$ and the $i$th episode in our dataset as $\tau^i$. $\tau^i = (s^0_0, a^0_0, r^0_0, s^0_1, a^0_1, r^0_1, \dots)$ 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 $\tau$ is sampled. Therefore the sources of noise are the stochasticity of the

policy and the environment.

**Algorithm 1:** The policy gradients algorithm

**Result:** Optimized policy $\pi_\theta$
**Input:** Randonly initialized parameters $\theta$, environment $E$, stepsize $\alpha$

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

$\nabla_\theta J(\pi_\theta)$ 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. More mathematical details can be found in **blog post 1**, **blog post 2** and **paper 1**. 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.

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 $\phi_t$ can take numerous possible values and still be equivalent to the policy gradient given above. Examples include $\phi_t = R_{t:}$ as above or $\phi_t(\tau) = R_{t:}(\tau) - b(s_t)$ where $b$ is any function that only depends on the current state. Here $b(s_t)$ is often refered to as a baseline. Another quantity we can use in place of $\phi_t(\tau)$, which we suggest you use in this task, is the Temporal Difference residual $r_t + V^\pi(s_{t+1}) - V^\pi(s_t)$. Intuitively, using the TD-residual instead of $R_{t:}(\tau)$ leads to estimators with lower variance because the use of a value function pools information across other episodes in the batch. When training the value function, the reward-to-go can be used as a target for the loss. Typically an extra hyperparameter $\lambda$ is used in the discouting of the TD residual, incorporated to reduce variance in the TD residual estimate. The intuition behind discounting reducing variance in value estimates is that it moreso ignores the (more noisy) reward outcomes occuring in the distant future, which will have a weaker relationship with the current state than near-term outcomes. If you follow the TODOs given in the skeleton, you can use $\gamma = 0.99$ and $\lambda = 0.97$ to get a solution that beats the baseline. This value function-based quantity can be estimated by averaging across many episodes, instead of estimated based upon the noisy outcomes of a single episode. Since the Generalized Advantage Estimation policy gradient is equivalent to the policy gradient used above, an unbiased estimate of this expectation will give an unbiased estimate of the policy gradient.