Introduction Welcome to our comprehensive exploration of Reinforcement Learning from Human Feedback (RLHF)

ロビン（Robin Daily Life） • 19 Apr 2024

Introduction

Welcome to our comprehensive exploration of Reinforcement Learning from Human Feedback (RLHF). In this blog, we will thoroughly explore various aspects of Reinforcement Learning, including but not limited to vanilla policy gradient, methods to reduce variance, generalized advantage estimation, PPO, and RLHF. Our aim is to clearly present these concepts, ensuring a deep understanding of each component and its role in the algorithm's success.

We begin with an introduction to reinforcement learning.

What is Reinforcement Learning?

Reinforcement Learning is a general description of a learning problem where the aim is to maximize a long-term objective. The system consists of an agent which interacts with the environment via its actions at discrete time steps and receives a reward. This transitions the agent into a new state. An agent-environment feedback loop is depicted by the figure below.

Observation of State: The agent observes the current state `$latex s_t$` Action Taken: Based on the policy `$latex \pi$`, the agent takes an action `$latex a_t$` Environment Response: The environment reacts to the agent's action, leading to a new state Continued Actions: The agent takes further actions based on the new observed state, continuing the interaction cycle. Policy Update: After a complete trajectory, the agent updates its policy according to the cumulative rewards using the policy gradient update rule (derived further below).

The agent's continuous cycle of observation, action, and response, as detailed above, forms the basis of how it learns and adapts within its environment. The agent must formally work through a theoretical framework known as a Markov Decision Process which consists of a decision (what action to take?) to be made at each state. This gives rise to a sequence of states, actions and rewards known as a trajectory:

The agent's continuous cycle of observation, action, and response, as detailed above, forms the basis of how it learns and adapts within its environment. The agent must formally work through a theoretical framework known as a Markov Decision Process which consists of a decision (what action to take?) to be made at each state. This gives rise to a sequence of states, actions and rewards known as a trajectory:

$latex \tau = {s_0, a_0, r_0, s_1, a_1, r_1 \ldots }$

and the objective is to maximize this set of rewards. More formally, we look at the Markov Decision Process framework.

Markov Decision Process

A Markov Decision Process (MDP) provides a mathematical framework for modeling decision-making situations where outcomes are partly random and partly under the control of a decision maker. MDPs are particularly useful for studying optimization problems solved via dynamic programming and reinforcement learning.

An MDP is defined by the following components:

Set of States $latex S$: All possible states the environment can be in. Set of Actions $latex A$: Actions available to the agent. Dynamics/Transition Model $latex P(s_{t+1} \mid s_t, a_t)$: Probability of moving to state $latex s_{t+1}$ from state $latex s_t$ after taking action $latex a_t$. Reward Function $latex R(\tau)$: Reward associated with each trajectory. Start State $latex s_0$: Initial state where the agent starts. Discount Factor $latex \gamma$: Factor between 0 and 1 used to discount future rewards. Horizon $latex T$: Number of steps in the decision process, which can be finite or infinite.

The most crucial aspect for an MDP is the so-called Markov Property:

"Future is Independent of the past given the present"

Stated more formally, we have that

$latex P(s_{t+1} = s' \mid s_0, a_0, s_1, a_1, \ldots , s_t, a_t) = P(s_{t+1} \mid s_t, a_t)$ In particular, this means that the probabilitity of $latex s_{t+1}$ being a certain state $latex s'$ given all previous states and actions depends only on state $latex s_t$ and action $latex a_t$. This allows us to neatly factor the probability of a trajectory $latex \tau$ over a horizon $latex T$ as such:

$latex P(\tau) = \prod *{t=0}^{T-1} P(s*{t+1} \mid s_t, a_t) $

The Reward Hypothesis

A large amount of theory behind RL lies under the assumption of Reward Hypothesis which in summary states that all goals and purposes of an agent can be explained by a single scalar called the reward. More formally, the reward hypothesis is given below

“All of what we mean by goals and purposes can be well thought of as the maximization of the expected value of the cumulative sum of a received scalar signal (called reward).”

As an RL practitioner and researcher, one’s job is to find the right set of rewards for a given problem known as reward shaping.

Our primary goal in reinforcement learning is thus to maximize the expected reward. This means we design a control policy `$latex \pi_\theta$` (often taking the form of a neural network and LLM later on), parameterized by `$latex \theta$`, that chooses actions in each state to increase the total reward the agent collects over time. Formally, we aim to solve:

$latex `\max_\theta \mathbb{E} \left[\sum_{t=0}^{T-1} r(s_t,a_t) \mid \pi_\theta \right]` $

Here, the policy is controlled by parameters `$latex \theta$`, and our task is to adjust these parameters so that the expected cumulative reward, summed over a horizon `$latex T$`, is as high as possible. Essentially, we seek the best strategy or policy that maximizes the sum of rewards an agent receives while interacting with its environment, considering both immediate and potential future rewards to ensure decisions contribute positively towards long-term goals.

Likelihood Ratio Policy Gradient

In reinforcement learning, computing policy gradients is often approached through the likelihood ratio policy gradient method. This involves some important concepts and notations we need to understand:

The reward for a given trajectory `$latex \tau$` is calculated as the sum of rewards for each state-action pair within the trajectory:

`$latex R(\tau) = \sum_{t=0}^{T-1} r(s_t, a_t)$`

The utility function `$latex J(\theta)$` represents the expected sum of rewards obtained by following a policy parameterized by `$latex \theta$`. Each setting of `$latex \theta$` in the neural network corresponds to different probabilities of observing certain trajectories, thus leading to different expected rewards. The utility function is expressed as:

$latex J(`\theta`) = `\mathbb{E}` [r(s_t, a_t); `\pi _ \theta`] = `\sum _{ \tau }` P(`\tau` ; `\theta`) R(`\tau`) $

Our goal is to optimize the policy by finding the best parameters `$latex \theta$` that maximize `$J(\theta)$`:

`$latex \max_{\theta}J(\theta) = \max_{\theta} \sum_{\tau} P(\tau; \theta) R(\tau)$`

Adjusting `$latex \theta$` changes the likelihood of trajectories, guiding us towards those that are more likely to yield higher rewards and away from less rewarding ones. This optimization seeks the parameter settings that preferentially select high-reward trajectories.

To optimize our objective function `$latex J(\theta)$`, we employ a gradient-based estimation approach. The core idea is to adjust the parameter `$latex \theta$` to maximize `$latex J(\theta)$`, which involves computing the gradient of `$latex J(\theta)$` with respect to `$latex \theta$`.

We begin by taking the gradient of `$latex J(\theta)$` with respect to `$latex \theta$`, distributing the gradient operation over the summation of all trajectories. We then cleverly manipulate the terms to facilitate computation:

`$latex \displaystyle \begin{array}{lcl} \nabla_\theta J(\theta) & = & \nabla_\theta \sum_{\tau} P(\tau; \theta) R(\tau) \\ & = & \sum_{\tau} \nabla_\theta P(\tau; \theta) R(\tau) \\ & = & \sum_{\tau} \frac{P(\tau; \theta)}{P(\tau; \theta)} \nabla_\theta P(\tau; \theta) R(\tau) \\ & = & \sum_{\tau} P(\tau; \theta) \frac{\nabla_\theta P(\tau; \theta)}{P(\tau; \theta)} R(\tau) \\ & = & \sum_{\tau} P(\tau; \theta) \nabla_\theta \log P(\tau; \theta) R(\tau) \end{array}$`

The transformation in the fourth step to the final form in the fifth step utilizes the log trick, that is, replacing `$latex \frac{\nabla_\theta P(\tau; \theta)}{P(\tau; \theta)}$` with `$latex \nabla_\theta \log P(\tau; \theta)$`.

Finally, we are thus able to approximate $latex \nabla_\theta J(\theta)$ with the following empirical estimate for $latex m$ sample trajectories under policy $latex \pi_\theta$:

$latex \nabla_\theta J(\theta) \approx \widehat{g} = \frac{1}{m} \sum_{i=1}^{m} \nabla_\theta \log P(\tau^{(i)}; \theta)R(\tau^{(i)})$

Of course, there's still a considerable amount of work required, but this outlines the basic framework. What's particularly compelling is that this method is applicable regardless of the specific form of the reward function. Crucially, it's not necessary for the reward function to be differentiable. The derivatives required are solely with respect to the distribution of trajectories generated by the neural network. For instance, this approach is effective even with a binary reward function, such as one that assigns a value of 1 when a goal is achieved and 0 otherwise.

Temporal Decomposition

So far, we have discussed entire paths, but often our rewards are more localized. For a specific trajectory $latex \tau^{(i)}$ with $latex i \in [m]$, we decompose $latex P (\tau ^{(i)}; \theta)$ into a dynamics model $latex P(s_{t+1}^{(i)} \mid s_t^{(i)}, a_t^{(i)})$ and policy $latex \pi_\theta (a_t^{(i)} \mid s_t^{(i)})$ as follows:

$latex \displaystyle \begin{array}{lcl} \nabla_\theta \log P (\tau ^{(i)}; \theta) & = & \nabla_\theta \log \left[\prod_{t=0}^{T-1} P(s_{t+1}^{(i)} \mid s_t^{(i)}, a_t^{(i)}) \cdot \pi_\theta (a_t^{(i)} \mid s_t^{(i)}) \right] \\ & = & \nabla_\theta \left[\sum_{t=0}^{T-1} \log P(s_{t+1}^{(i)} \mid s_t^{(i)}, a_t^{(i)}) + \sum_{t=0} ^{T-1} \log \pi_\theta (a_t^{(i)} \mid s_t^{(i)}) \right] \\ & = & \nabla_\theta \sum_{t=0}^{T-1} \log \pi_\theta (a_t^{(i)} \mid s_t^{(i)}) \\ & = & \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta (a_t^{(i)} \mid s_t^{(i)}) \end{array}$

Since the dynamics model does not depend on $latex \theta$, we have drop $latex \theta$ early in the calculation. Thus, to increase the probability of a trajectory, the optimization step increases the log probabilities of actions along that trajectory, and to decrease the probability of a trajectory, the optimization decreases the log probabilities of actions along that trajectory.

Thus, the approximated policy gradient, also known as the REINFORCE algorithm, can be written as: $latex `\widehat{g}` = `\frac{1}{m}` `\sum`_{i=1}^{m} `\left[ \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta (a_t^{(i)} \mid s_t^{(i)}) R(\tau^{(i)}) \right]` $

where the estimation is unbiased: $latex `\mathbb{E}`[`\widehat{g}`] = `\nabla`_`\theta` J(`\theta`) $ as our estimation is simply the mean over a sample of `$m$` trajectories. However, the estimation is very noisy and requires many samples to be precise. Following, we explore various methods to decrease the variance of the estimation.

Variance Reduction

Recall that the reward function `$latex R(\tau)$` can be written as the sum of all individual rewards earned at each time step of the trajectory. $latex R(`\tau`) = `\sum`_{t=0}^{T-1} r(s_t,a_t) $

Thus, the approximated vanilla policy gradient can be rewritten as `$latex \displaystyle \begin{array}{lcl} \widehat{g} & = & \frac{1}{m} \sum_{i=1}^{m} \left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta (a_t^{(i)} \mid s_t^{(i)}) R(\tau^{(i)}) \right] \\ &=& \frac{1}{m} \sum_{i=1}^{m} \left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta (a_t^{(i)} \mid s_t^{(i)}) \sum_{t=0}^{T-1} r(s_t,a_t) \right] \end{array}$`

The equation has many terms - There are m trajectories, and each trajectory has `$latex T$` time steps. Each of the `$latex T$` log probabilities, `$latex \pi_{\theta}(a_t^{(i)} \mid s_t^{(i)})$`, per trajectory are multiplied by the sum of `$latex T$` reward terms. As a result, one major issue with the algorithm is that it has high variance due to the high number of terms and the nature of the equation, which is an approximation of the expected policy gradient.

Changing the number of sampled trajectories does not address the issue sufficiently. As fewer trajectories are sampled, the variance increases since the sampled trajectories are unable to represent the overall trajectory distribution. As more trajectories are sampled, the variance does decrease since the sampled trajectories can better represent the overall trajectory distribution. However, sampling more trajectories also entails a larger number of terms computed for the policy gradient, making training inefficient. Thus, increasing sample trajectory size is possible, but difficult in practice.

There are practical two methods to reduce the variance: rewards to go and baseline subtraction.

Rewards To Go

The “rewards to go” trick removes some terms from the policy gradient approximation by taking advantage of the fact that the trajectory is a Markov Decision Process. Intuitively, the probability of taking an action `$latex a_t$` for a certain state `$latex s_t$` for timestep `$latex t$` does not influence past rewards `$latex r(s_i, a_i)$`, for `$latex i <t$`. Thus:

`$latex \displaystyle \begin{array}{lcl} \nabla_\theta J(\theta) & = & \mathbb{E}{\tau \sim \pi} \left[\sum{t=0}^{T-1} \left(\nabla_\theta \log \pi_\theta (a_t^{(i)} \mid s_t^{(i)}) \sum_{t=0}^{T-1} r(s_t,a_t)\right) \right] \\ & \approx & \frac{1}{m} \sum_{i=1}^{m} \left[\sum_{t=0}^{T-1} \left(\nabla_\theta \log \pi_\theta (a_t^{(i)} \mid s_t^{(i)}) \sum_{t'=0}^{T-1} r(s_{t'},a_{t'})\right) \right] \\ &=& \frac{1}{m} \sum_{i=1}^{m} \left[\left(\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta (a_t^{(i)} \mid s_t^{(i)}) \right) \sum_{t'=0}^{T-1} r(s_{t'},a_{t'}) \right] \\ & \approx & \frac{1}{m} \sum_{i=1}^{m} \left[\left(\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta (a_t^{(i)} \mid s_t^{(i)}) \right)\sum_{t'=t}^{T-1} r(s_{t'},a_{t'}) \right] \end{array}$`

Effectively, we remove the reward terms `$latex r(s_{t'},a_{t'})$` for all `$latex t' < t$`, since the probability of taking an action at time `$latex t$` should not be able to influence the rewards of past time steps.

Proof: $latex `\displaystyle` R(`\tau`) = `\sum` *{t’=0}^{T-1} r*{t’}(s_{t’},a_{t’}) = `\sum` *{t’=0}^{t-1}r*{t’}(s_{t’},a_{t’}) + `\sum` *{t’=t}^{T-1} r*{t’}(s_{t’},a_{t’}) $ Thus, the sum of expectation equals the expectation of the sum, so

`$latex \displaystyle \begin{array}{lcl} \nabla_\theta J(\theta) &=& \mathbb{E}{\tau \sim \pi} \left[\sum{t=0}^{T-1} \left(\nabla_\theta \log \pi_\theta (a_t^{(i)} \mid s_t^{(i)}) \sum_{t'=0}^{T-1} r(s_{t'},a_{t'})\right) \right] \\ &=& \mathbb{E}{\tau \sim \pi} \left[\sum{t=0}^{T-1} \left(\nabla_\theta \log \pi_\theta (a_t^{(i)} \mid s_t^{(i)}) \sum_{t'=0}^{t-1} r(s_{t'},a_{t'})\right) \right] + \mathbb{E}{\tau \sim \pi} \left[\sum{t=0}^{T-1}`

`\left(\nabla_\theta \log \pi_\theta (a_t^{(i)} \mid s_t^{(i)}) \sum_{t'=t}^{T-1} r(s_{t'},a_{t'})\right) \right] \end{array}$`

Now let's focus our attention on the first term on the RHS. Since the trajectory is a Markov Decision Process, the action taken at a particular state is only dependent on that state. Thus the probability of taking an action at a particular state is independent of the probability of taking an action at any another state. Likewise, the rewards from past time steps before `$latex t$` are independent from the action taken at time step `$latex t$`. Given these independent terms, we can expand we can expand the expectation:

$latex `\displaystyle \mathbb{E}_{\tau \sim \pi} \left[ \sum{t=0}^{T-1} \left(\nabla_\theta \log \pi_\theta (a_t^{(i)} \mid s_t^{(i)}) \sum_{t'=0}^{t-1} r(s_{t'},a_{t'})\right) \right] = \sum`{t=0}^{T-1} `\mathbb{E}_{\tau \sim \pi} \left[ \nabla\theta \log \pi_\theta (a_t^{(i)} \mid s_t^{(i)}) \right] * \mathbb{E}`{`\tau \sim \pi`} `\left[\sum{t'=0}^{T-1} r(s_{t'}, a_{t'})] \right]`$

We can simplify the argument of the first summation term on the RHS. `$latex \displaystyle \begin{array}{lcl} \mathbb{E}_{\tau \sim \pi} \left[\nabla\theta \log \pi_\theta (a_t^{(i)} \mid s_t^{(i)}) \right] &=& \int \pi_\theta(a_t \mid s_t) \nabla_\theta \log \pi_\theta(a_t \mid s_t) d\pi \\ &=& \int \pi_\theta(a_t \mid s_t) \frac{\nabla_\theta \pi_\theta(a_t \mid s_t)}{\pi_\theta (a_t \mid s_t)} d\pi \\ &=& \nabla_\theta \int \pi_\theta(a_t \mid s_t) dt \\ &=& \nabla_\theta * 1 \\ &=& 0 \end{array}$`

Thus, $latex `\displaystyle \mathbb{E}_{\tau \sim \pi} \left[ \sum{t=0}^{T-1} \left(\nabla_\theta \log \pi_\theta (a_t^{(i)} \mid s_t^{(i)}) \sum_{t'=0}^{t-1} r(s_{t'},a_{t'})\right) \right] = 0 * \mathbb{E}`{`\tau \sim \pi`} `\left[\sum{t'=0}^{T-1} r(s_{t'}, a_{t'}) \right] = 0`$

Since the expectation of the product of the log probabilities of a trajectory and the rewards of past action-state pairs is `$latex 0$`, we can simplify the objective function `$latex \displaystyle \begin{array}{lcl} \nabla_\theta J(\theta) &=& \mathbb{E}_{\tau \sim \pi} \left[ \sum{t=0}^{T-1} \left(\nabla_\theta \log \pi_\theta`

(a_t^{(i)} \mid s_t^{(i)}) \sum_{t'=0}^{T-1} r(s_{t'},a_{t'})\right) \right] \\ &=& \mathbb{E}_{\tau \sim \pi} \left[\sum{t=0}^{T-1} \left(\nabla_\theta \log \pi_\theta (a_t^{(i)} \mid s_t^{(i)}) \sum_{t'=0}^{t-1} r(s_{t'},a_{t'})\right) \right] + \mathbb{E}_{\tau \sim \pi} \left[\sum{t=0}^{T-1} \left(\nabla_\theta \log \pi_\theta (a_t^{(i)} \mid s_t^{(i)}) \sum_{t'=t}^{T-1} r(s_{t'},a_{t'})\right) \right] \\ &=& 0 + \mathbb{E}_{\tau \sim \pi} \left[\sum{t=0}^{T-1} \left(\nabla_\theta \log \pi_\theta (a_t^{(i)} \mid s_t^{(i)}) \sum_{t'=t}^{T-1} r(s_{t'},a_{t'})\right) \right] \\ &=& \mathbb{E}_{\tau \sim \pi} \left[\sum{t=0}^{T-1} \left(\nabla_\theta \log \pi_\theta (a_t^{(i)} \mid s_t^{(i)}) \sum_{t'=t}^{T-1} r(s_{t'},a_{t'})\right) \right] \end{array}$

Thus, using rewards to go does not change the overall bias but does reduce the variance

Baseline Subtraction

A standard approach to reduce variance is to subtract a “baseline” $latex b$ from the total reward to estimate the policy gradient:

$latex \displaystyle \nabla \theta J(\theta) \approx \widehat{g} = \frac{1}{m} \sum{i=1}^{m} \nabla_ \theta \log P(\tau ^{(i)}; \theta)R(\tau ^{(i)} - b) $

In particular, the resulting gradient estimates become less sensitive to variations in rewards that are not directly influenced by actions taken by the policy. This helps to reduce variance of gradient estimates, which in turn leads to more stable and efficient learning.

As proven below, the estimation of $latex \nabla_\theta J(\theta)$ remains unbiased.

Proof: $latex \displaystyle \begin{array}{lcl} \mathbb{E} [\nabla_\theta \log P(\tau; \theta)b] &=& \sum_{\tau} P(\tau; \theta) \nabla_\theta \log P(\tau; \theta)b \\ &=& \sum_{\tau} P(\tau; \theta) \frac{\nabla_\theta P(\tau; \theta)}{P(\tau; \theta)} \\ &=& \sum_{\tau} \nabla_\theta P(\tau; \theta)b \\ &=& \nabla_\theta \left(\sum_{\tau} P(\tau)b \right) \\ &=& b * \nabla_\theta \left(\sum_{\tau} P(\tau) \right) \\ &=& b * 0 \end{array}$

Our current estimate with the baseline is thus:

$latex \displaystyle \begin{array}{lcl} \widehat{g} &=& \frac{1}{m} \sum_{i=1}^{m} \nabla_\theta \log P(\tau^{(i)}; \theta)R(\tau^{(i)} - b) \\ &=& \frac{1}{m} \sum_{i=1}^{m} \left(\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta (a_t^{(i)} \mid s_t^{(i)}) \right) \left(\sum_{t=0}^{T-1} r(s_t^{(i)}, a_t^{(i)}) - b \right) \\ &=& \frac{1}{m} \sum_{i=1}^{m} \left(\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta (a_t^{(i)} \mid s_t^{(i)}) \left[\left(\sum_{k=0}^{t-1} r(s_t^{(i)}, a_t^{(i)})\right) + \left(\sum_{k=t}^{T-1} r(s_t^{(i)}, a_t^{(i)}) - b \right) \right] \right) \\ &=& \frac{1}{m} \sum_{i=1}^{m} \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta (a_t^{(i)} \mid s_t^{(i)})\left(\sum_{k=t}^{T-1} r(s_t^{(i)}, a_t^{(i)}) - b(s_t^{(i)}) \right) \end{array}$

where the last step utilizes our “rewards to go trick”.

Choices for the Baseline

Now, what are good choices we can use for the baseline? Loosely, we would like $latex b$ to represent some form of “average” of our reward. Let’s explore several common approaches used for this purpose.

A simple choice is a constant baseline, which involves averaging the rewards across all trajectories: $latex b = \mathbb{E}[R(\tau)] \approx \frac{1}{m} \sum_{i=1}^m R(\tau^{(i)})$

For a more refined approach, we can consider a minimal variance baseline. Unlike a simple average, this method involves a weighted average where each trajectory’s weight is proportional to the square of the gradient log probability. This technique assigns more weight to trajectories with higher gradients, leading to a lower variance estimation: $latex b = \frac{\sum_i (\nabla_\theta \log P (\tau^{(i)}; \theta))^2 R(\tau^{(i)}) }{\sum_i (\nabla_\theta \log P (\tau^{(i)}; \theta))^2}$

Although this method can be effective, it is rarely used in contemporary applications due to its complexity.

Time-dependent baselines are another popular choice, especially useful in scenarios with finite horizon rollouts. They account for the diminishing rewards as time progresses within a rollout:

$latex \displaystyle b_t = \frac{1}{m} \sum_{i=1}^m \sum_{k=t}^{T-1} r (s_k^{(i)}, a_k^{(i)})$

For an even more tailored approach, state-dependent baselines use the expected return from a specific state and time – the value function $latex V^\pi(s_t)$. This method calculates the expected cumulative reward from a state at time t until the end of the horizon: $latex b(s_t) = \mathbb{E}[r_t + r_{t+1} + \cdots + r_{T-1}] = V^\pi(s_t)$

Value Function Estimation

Using the value function, our estimation of $latex \nabla_\theta J(\theta)$ is thus: $latex \displaystyle \widehat{g} = \frac{1}{m} \sum*{i=1}^{m} \sum*{t=0}^{T-1} \nabla *\theta \log \pi* \theta (a_t^{(i)} \mid s_t^{(i)}) \left(\sum _{k=t}^{T-1} r(s_t^{(i)}, a_t^{(i)}) - V^{\pi (s_k}{(i)}) \right) $

Now, how exactly do we estimate $latex V^\pi(s_t)$? A popular choice is to initialize a neural network with parameters $latex \phi_0$ and estimate $latex {V_\phi}_0^\pi$ under policy $latex \pi$ as follows:

Begin by initializing $latex {V_\phi}_0^\pi$ Collect trajectories $latex \tau_1, \ldots , \tau_m$ Regress against empirical return:

$latex \displaystyle \phi*{i+1} \gets {\arg\min} \phi \frac{1}{m} \sum{i=1}^m \sum*{t=0}^{T-1} \left({V_ \phi }^{\pi (s_t}{(i)}) - \left(\sum _{k=t}^{T-1} r(s_k^{(i)}, a_k^{(i)}) \right) \right)^2 $

Using a separate network to estimate the value function will be explored in the Actor-Critic Algorithm.

Generalized Advantage Estimation (GAE) Advantage

At this point, our policy gradient method utilizes a single sample estimate $latex \sum_{k=t}^{T-1} r(s_t^{(i)}, a_t^{(i)})$ of the so-called Q function for a single roll out:

$latex Q^\pi(s, a) = \mathbb{E}\left[\sum_{l=0}^\infty r_l \mid s_0 = s, a_0 = a\right]$

Following, we define the advantage function as follows: $latex A^\pi(s_t, a_t) := Q^\pi(s_t, a_t) - V^\pi(s_t) $

The idea is that the Advantage function calculates how better taking that action at a state is compared to the average value of the state. It's subtracting the mean value `$latex V(s)$` of the state from the state action pair `$latex Q(s, a)$` where `$V(s) = \mathbb{E} \left[Q(s,a)\right]$`. In other words, this function calculates the extra reward we get if we take this action at that state compared to the mean reward we get at that state.

The extra reward is what's beyond the expected value of that state.

If `$latex A(s,a) > 0$`: our gradient is pushed in that direction. If `$latex A(s,a) < 0$`: our action does worse than the average value of that state, so our gradient is pushed in the opposite direction.

Hence, our estimation is thus:

$latex `\displaystyle` `\widehat{g}` = `\frac{1}{m}` `\sum`*{i=1}^{m}* `\sum`{t=0}^{T-1} `\nabla` `\theta` `\log` `\pi` `\theta` (a_t^{(i)} `\mid` s_t^{(i)}) A^ `\pi` (s_t, a_t) $

GAE

A clear question arises: How do we go about estimating `$latex A^\pi(s_t, a_t)$`? With Generalized Advantage Estimation (GAE), we introduce a parameter `$latex \gamma$` that allows us to reduce variance by downweighting rewards corresponding to delayed effects. Of course, this comes at the cost of introducing bias. With the discount, `$latex Q^\pi$` and `$latex V^\pi$` are represented as follows:

$latex Q^ `\pi` (s, a) = `\mathbb{E}` `\left[ \sum_{l=0}^\infty \gamma^l r_l \mid s_0 = s, a_0 = a\right]` `\quad` V^ `\pi` (s) = `\mathbb{E}` `\left[ \sum_{l=0}^\infty \gamma^l r_l \mid s_0 = s\right]` $

Now, in estimating `$latex A^\pi(s_t, a_t)$`, we are presented with the following bias-variance tradeoff:

`$latex \displaystyle \begin{array}{lcl} \hat{A}^\pi(s_t, a_t) &=& [r(s_t, a_t) + \gamma V^\pi(s_{t+1})] - V^\pi(s_t) \\ \hat{A}^\pi(s_t, a_t) &=& [r(s_t, a_t) + \gamma r(s_{t+1}, a_{t+1}) + \gamma^2 V^\pi(s_{t+2})] - V^\pi(s_t) \\ \hat{A}^\pi(s_t, a_t) &=& [r(s_t, a_t) + \gamma r(s_{t+1}, a_{t+1}) + \gamma^2 r(s_{t+2}, a_{t+2}) + \gamma V^\pi(s_{t+3})] - V^\pi(s_t) \end{array}$`

In particular, if we go further in the roll out, that is, utilize more `$latex r(s, a)$` terms we sample, the higher our variance as we are summing up more terms. Conversely, the sooner we stop, the higher our bias since `$latex V^\pi(s)$` is an estimate. One approach is picking a specific value for this estimation - for example, choosing the third `$latex \hat{A}^\pi(s_t, a_t)$` estimate. Instead, GAE proceeds as follows: We define the Temporal Difference (TD) error `$latex \delta_t$` as such:

`$latex \delta_t^V = r(s_t, a_t) + \gamma V(s_{t+1}) - V(s_t) $`

Then, instead of empirically testing different step sizes to determine an optimal one, we utilize the exponentially-weighted average of these k-step estimators:

`$latex \displaystyle \begin{array}{lcl} \hat{A}t^{(1)} &:=& \delta_t^V \\ \hat{A}_t^{(2)} &:=& \delta_t^V + \gamma \delta{t+1}^V \\ \hat{A}t^{(3)} &:=& \delta_t^V + \gamma \delta{t+1}^V + \gamma^2 \delta_{t+2}^V \\ & \vdots & \\ \hat{A}t^{(k)} &:=& \sum{l=0}^{k-1} \gamma^l \delta_{t+l}^V \end{array}$`

Applying the exponentially-weighted average, we get the final form of GAE:

`$latex \displaystyle \begin{array}{lcl} \hat{A}t^{GAE} &:=& (1-\lambda) \left(\hat{A}_t^{(1)} + \lambda \hat{A}_t^{(2)} + \lambda^2 \hat{A}_t^{(3)} + \cdots \right) \\ &:=& (1 - \lambda)(\delta_t^V + \lambda(\delta_t^V + \gamma\delta{t+1}^V) + \lambda^2 (\delta_t^V + \gamma \delta_{t+1}^V + \gamma^2 \delta_{t+1}^V) + \cdots) \\ &:=& (1 - \lambda) (\delta_t^V (1 + \lambda + \lambda^2 + \cdots) + \gamma \delta_{t+1}^V (\lambda + \lambda^2 + \lambda^3 \cdots) + \gamma^2 \delta_{t+2}^V (\lambda^2 + \lambda^3 + \lambda^4 \cdots) \\ &:=& (1 - \lambda) \left(\delta_t^V \left(\frac{1}{1-\lambda} \right) + \gamma\delta_{t+1}^V \left(\frac{1}{1-\lambda} \right) + \gamma^2\delta_{t+2}^V \left(\frac{1}{1-\lambda} \right) + \cdots \right) \\ &=& \sum_{l=0}^\infty(\gamma\lambda)^l \delta_{t+l}^V \end{array}$`

Note here we introduced another parameter `$latex \lambda$`. When `$latex \lambda = 0$`, GAE is essentially the same as TD(`$latex 0$`) but applied in the context of policy optimization. It has high bias as it heavily relies on the estimated value function. When `$latex \lambda = 1$`, this is

the case of vanilla policy gradient with a baseline that has high variance due to the sum of terms. In practice we set `$latex 0 < \lambda < 1$` to control the compromise between bias and variance just like the lambda parameter in TD lambda.

Vanilla Policy Gradient

The “vanilla” policy gradient is thus given as follows

Pseudocode from OpenAI https://spinningup.openai.com/en/latest/algorithms/vpg.html

Actor-Critic Algorithms Actor-Critic Algorithm

The Actor-Critic algorithm is a combination of policy-based and value-based methods for reinforcement learning intended to reduce the variance of Vanilla Policy Gradient. It has two networks: Actor and Critic. The actor decided which action should be taken and critic inform the actor how good was the action and how it should adjust. The learning of the actor is based on policy gradient approach. In comparison, critics evaluate the action produced by the actor by computing the value function.

The Actor-Critic learns two function approximations: a policy that controls how our agent acts `$latex \pi_\theta(s,a)$`, and a value function `$latex Q_w (s,a)$` to assist the value update by measuring how good the action is. Let’s see how the actor and critic are optimized

At each timestep, `$latex t$`, we get the current state `$latex s_t$` from the environment and pass it as input through our Actor and Critic. Our Policy takes the state `$latex s_t$` and outputs an action `$latex a_t$`. The Critic takes that action also as input and, using `$latex s_t$` and `$latex a_t$`, computes the value of taking that action at that state: the Q-value. The action `$latex a_t$` performed in the environment outputs a new state `$latex s_{t+1}$` and reward `$latex r_{t+1}$`. The actor then updates its policy parameters using gradient ascent using the parameter gradient `$latex \Delta \theta_t = \alpha \nabla_\theta (\log \pi_\theta (s_t, a_t)) Q_w(s_t,a_t)$` After the actor updates its parameters, it determines state `$latex a_{t+1}$` from its current state `$latex s_{t+1}$`. The critic then is able to updates its values using the temporal difference (TD) error between timestep `$latex t+1$` and `$latex t$`. `$latex \Delta w = \beta (r(s_t, a_t) + \gamma Q_w(s_{t+1}, a_{t+1}) - Q_w(s_t, a_t)) $` Advantage Actor Critic (A2C) Algorithm

We can stabilize policy learning further by using the Advantage function `$latex A$` as the Critic instead of the `$latex Q$` value function.

`$latex A_w(s, a) = Q(s,a) - V(s)$`

The problem with implementing this advantage function is that it requires two value functions — `$latex Q(s,a)$` and `$latex V(s)$`. Fortunately, we can use the temporal difference error described previously as a good estimator of the advantage function,.

`$latex \displaystyle \begin{array}{lcl} \hat{A_w}(s, a) &=& Q_w(s,a) - V_w(s) \\ &=& r(s,a) + \gamma V_w(s_{t+1}) - V_w(s_t) \\ &=& r(s_t, a_t) + \gamma V_w(s_{t+1}) - V_w(s_t) \end{array}$`

The gradient of the policy is thus expressed as `$latex \displaystyle \begin{array}{lcl} \nabla_\theta J(\theta) &\approx \frac{1}{m} \sum_{i=1}^{m} \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta (a_t^{(i)} \mid s_t^{(i)}) A_w(s_t,a_t) \\ &=& \frac{1}{m} \sum_{i=1}^{m} \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta (a_t^{(i)} \mid s_t^{(i)})\left(r(s_t, a_t) + \gamma V_w(s_{t+1}) - V_w(s_t) \right) \end{array}$`

Using gradient ascent, we can update the policy parameters with `$latex \theta = \theta + \alpha \nabla J(\theta)$`.

The critic loss function is `$latex L_{VF} = \frac{1}{m} \sum_{i=1}^{m}\frac{1}{2} ||V_{\theta} - (\sum_{t=0}^{T-1} \gamma^t r_t \mid s_0 = s)||_2^2$`. Thus, using gradient descent, we update the `$latex w$` parameters of the critic with

`$latex \displaystyle w = w - \beta \nabla_w \left( \frac{1}{m} \sum_{i=1}^{m}\frac{1}{2} ||V_{\theta} - (\sum_{t=0}^{T-1} \gamma^t r_t \mid s_0 = s)||_2^2\right)$`

Importance Sampling

Motivation One issue with Vanilla Policy Gradient is that the approximated expectation `$\hat{g}$` forces us to sample trajectories every time we update the parameters. This can be very inefficient and difficult in practice because when training a neural network, we update the parameters many times by taking small steps.

Importance sampling is an approximation method that allows us to evaluate an expectation over a distribution X using samples taken from another distribution Y. It is typically used when the distribution of interest is difficult to sample from. Using a simple transformation formula, we can get

$latex `\displaystyle`

```
\begin{array}{lcl}
\mathbb{E}_{x \sim p(x)}[f(x)] &=& \int p(x) f(x) dx \
\
&=& \int \frac{q(x)}{q(x)} p(x) f(x) dx \\
&=& \int q(x) \frac{p(x)}{q(x)} f(x) dx \\
&=& \mathbb{E}_{x \sim q(x)} \left[ \frac{p(x)}{q(x)}
f(x)]\right]
\end{array}
```

$

where `$latex x$` is sampled from distribution `$latex q(x)$`, and `$latex q(x)$` should not be `$latex 0$`. In this way, the expectation can be estimated by sampling from another distribution `$latex q(x)$`. And `$latex p(x)/q(x)$` is called the sampling ratio or sampling weight, which acts as a correction weight to offset the probability of sampling from a different distribution.

In the context of reinforcement learning, this allows us to sample trajectories not from the policy `$latex \pi_{\theta}$` itself, but from a separate policy `$latex \pi_{\theta'}$`.

`$latex \displaystyle \begin{array}{lcl} \nabla_\theta J(\theta) &\approx& \frac{1}{N} \sum_{i=1}^m \left(\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta (a_{i,t} \mid s_{i,t}) \right) A^\pi (s,a) \\ \nabla_{\theta} J(\theta, \theta') &\approx & \frac{1}{N} \sum_{i=1}^m \left(\sum_{t=0}^{T-1} \nabla_{\theta} \frac{\log \pi_{\theta} (a_{i,t} \mid s_{i,t})}{\log \pi_{\theta'} (a_{i,t} \mid s_{i,t})} \right) A^\pi (s,a) \end{array}$`

While first equation for the objective function uses the policy we are trying to optimize, `$latex \pi_\theta$`, as the policy responsible for sampling trajectories, the second equation uses importance sampling. First a separate policy `$latex \pi_{\theta'}$` samples a batch of trajectories and stores it

in memory. Then, the rewards for all state-action pairs and advantages for the trajectories are calculated for `$latex \pi_\theta'$`. Then, a mini-batch of trajectories are taken and passed through `$latex \pi_{\theta}$`, the policy we are trying to optimize, to obtain the log probabilities `$latex \pi_\theta(a_{i,t} \mid s_{i,t})$`, and mini-batch stochastic gradient ascent is used to optimize `$latex \pi_\theta$`.

The process of taking mini-batches of trajectories sampled from `$latex \pi_{\theta'}$` allows us to only need to calculate the rewards and advantages once, which speeds up the calculation of the policy gradient. After multiple training epochs, when `$latex \pi_\theta$` begins to diverge significantly from the old policy, the accuracy decreases. Therefore, we set `$latex \pi_\theta' = \pi_{\theta}$`, and sample a new batch of trajectories from `$latex \pi_\theta'$`.

In practice, we can just reuse the old samples to recalculate the total rewards. Thus, `$latex \pi_\theta' = \pi_{\theta_{old}}$`. $latex `\displaystyle` `\nabla`*{* `\theta` *}* *J(* `\theta`, `\theta`{old}) `\approx` `\frac{1}{N}` `\sum`*{i=1}^m* `\left(` `\sum`{t=0}^{T-1} `\nabla`_{ `\theta`} `\frac{\log \pi_{\theta} (a_{i,t} \mid s_{i,t})}{\log \pi_{\theta_{old}} (a_{i,t} \mid s_{i,t})}` `\right`) A^ `\pi` (s,a) $

Importance Sampling is used in the Proximal Policy Optimization, a state-of-the-art reinforcement learning algorithm we will discuss now.

Proximal Policy Optimization

Motivation: Using normal policy gradient keeps new and old policies close in parameter space. However, seemingly small difference in the parameter space can lead to large differences in performance. This can be lead to instabilities, as a wrong gradient step can drop performance and it can take many more time steps to recover from this drop in performance. Thus, how can we take the biggest possible improvement step on a policy using the data we currently have, without stepping so far that we accidentally cause performance collapse?

Modified version from RL — Proximal Policy Optimization (PPO) Explained by Jonathan Hui: https://jonathan-hui.medium.com/rl-proximal-policy-optimization-ppo-explained-77f014ec3f12e_caption

Proximal Policy Optimization is an Actor-Critic algorithm that solves this problem. They are one of the most popular reinforcement learning algorithms for their state of the art performance, and is used extensively in domains like robotics. There are two primary variants of PPO: PPO-Penalty and PPO-Clip.

PPO-Penalty: PPO-Penalty approximately solves a KL-Divergence constrained update TRPO, but penalizes the KL-divergence in the objective function instead of making it a hard constraint, and automatically adjusts the penalty coefficient over the course of training so that it's scaled appropriately. This is similar to TRPO, which will be explored briefly later.

PPO-Clip: PPO-Clip doesn't have a KL-divergence term in the objective and doesn't have a constraint at all. Instead relies on specialized clipping in the objective function to remove incentives for the new policy to get far from the old policy.

The intuition behind this is that smaller policy updates during training are more likely to converge to an optimal solution since larger steps may overshoot the optimal parameter values.

In this section, we will focus on PPO-Clip, which is the variant explored in class.

PPO Loss Function

PPO updates its policy using:

$latex `\displaystyle \theta`*{k+1}* = `\text{arg max}` `\theta` `\space` `\mathbb{E}`*{s,a* `\sim` `\pi`{ `\theta`_k}} `\left[ L(s,a, \theta_k,\theta)\right]` $

by typically taking steps using mini-batch SGD to maximize the objective. The Loss function is:
`$latex \displaystyle L(s,a,\theta_{k},\theta) = \min\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}\hat{A}^{\pi_{\theta_k}}(s,a), \text{clip}\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}, \space 1-\epsilon, \space 1+\epsilon\right)\hat{A}^{\pi_{\theta_k}}(s,a) \right)$`

in which `$latex \epsilon$` is a (small) hyperparameter which roughly says how far away the new policy, `$latex \pi_{\theta}$`, is allowed to go from the old, `$latex \pi_{\theta_k}$`. `$\hat{A}$` is the generalized advantage term, but the advantage term `$latex A$` can also be used.

This is a pretty complex expression, and it's hard to tell at first glance what it's doing, or how it helps keep the new policy close to the old policy. To figure out what intuition to take away from this, let's look at a single state-action pair `$latex (s,a)$`, and think of cases.

Advantage is positive: Suppose the advantage for that state-action pair is positive, in which case its contribution to the objective reduces to

$latex `\displaystyle` L(s,a, `\theta`*k*, `\theta)` = `\min \left(` `\frac{\pi_{\theta}(a|s)}{\pi_{\theta_k}(a|s)}`*, (1 +* `\epsilon)` `\right)` `\hat{A}`^{ `\pi` { `\theta` _k}}(s,a). $

Because the advantage is positive, the objective will increase if the action becomes more likely—that is, if `$latex \pi_{\theta}(a|s)$` increases. But the min in this term puts a limit to how much the objective can increase. Once `$latex \pi_{\theta}(a|s) > (1+\epsilon) \pi_{\theta_k}(a|s)$`, the min kicks in and this term hits a ceiling of `$latex (1+\epsilon) A^{\pi_{\theta_k}}(s,a)$`. Thus, the new policy does not benefit by going far away from the old policy.

Advantage is negative: Suppose the advantage for that state-action pair is negative, in which case its contribution to the objective reduces to

$latex `\displaystyle` L(s,a, `\theta`*k*, `\theta)` = `\max \left(` `\frac{\pi_{\theta}(a|s)}{\pi_{\theta_k}(a|s)}`*, (1 -* `\epsilon)` `\hat{A}` `\right)` *A^{* `\pi` { `\theta` _k}}(s,a). $

Because the advantage is negative, the objective will increase if the action becomes less likely—that is, if `$latex \pi_{\theta}(a|s)$` decreases. But the max in this term puts a limit to how much the objective can increase. Once `$latex \pi_{\theta}(a|s) < (1-\epsilon) \pi_{\theta_k}(a|s)$`, the max kicks in and this term hits a ceiling of `$latex (1-\epsilon) \hat{A}^{\pi_{\theta_k}}(s,a)$`. Thus, again: the new policy does not benefit by going far away from the old policy.

What we have seen so far is that clipping serves as a regularizer by removing incentives for the policy to change dramatically, and the hyperparameter `$\epsilon$` corresponds to how far away the new policy can go from the old while still profiting the objective.

Below is pseudocode for the PPO-clip algorithm from OpenAI. Note that OpenAI's implementation has `$latex T+1$` timesteps for a trajectory `$latex \tau$`, while in this blog we defined a trajectory as having `$latex T$` timesteps. However, they are computationally equivalent if we adjust our

$latex T$ to be $latex 1$ greater than the pseudocode $latex T$. Also note that OpenAI uses the advantage itself, and not the GAE, but the functionality remains the same.

Pseudocode from OpenAI https://spinningup.openai.com/en/latest/algorithms/ppo.html

PPO Actor-Critic Objective Function

The PPO Actor-Critic Loss combines the PPO-clip loss, value loss function, and an entropy loss term: $latex \displaystyle \begin{array}{lcl} L_{policy}(s,a,\theta_k,\theta) &=& \min\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s,a), \space g(\epsilon, \space A^{\pi{\theta_k}}(s,a)) \right) \\ L_{VF} &=& \frac{1}{2} || V_{\theta} - (\sum_{t=0}^H \gamma^t r_t \mid s_0 = s)||2^2 \\ \ L{entropy} &=& -\sum_{x} p(x) \log p(x) \\ L_{PPO} &=& L_{policy} + c_1 L_{VF} + c_2 L_{ENTROPY} \end{array}$

where $latex c_1, c_2$ are hyperparameters to scale $latex L_{VF}, L_{entropy}$. The entropy term ensures sufficient exploitation by penalizing the model for making deterministic choices since we want the model to explore options during training. We want to maximize $latex L_{policy}$, minimized $latex L_{VF}$, and maximize $latex L_{entropy}$.

Trust Region Policy Optimization (TRPO)

Motivation: TRPO is an alternative to PPO that also addressed the problem of avoiding taking gradient steps that are too large. It uses a constraint expressed in terms of KL-Divergence, a measure of (something like, but not exactly) distance between probability distributions. Thus, instead of limiting the gradient in the parameter space, TRPO limits the gradient in the policy space.

Let $latex \pi_{\theta}$ denote a policy with parameters $latex \theta$. The theoretical TRPO update is:

$latex \displaystyle \theta*{k+1} = \arg \max{\theta} { \mathcal L}(\theta*k, *\theta) \quad \quad \text{s.t.} \space \bar*{D}{KL}(\theta || \theta_k) \leq \delta $

where $latex {\mathcal L}(\theta_k, \theta)$ is the surrogate advantage, a measure of how policy $$\pi_{\theta}$$ performs relative to the old policy $latex \pi_{\theta_k}$ using data from the old policy:

$latex \displaystyle {\mathcal L}(\theta_k, \theta) = \mathbb{E}{s,a \sim \pi{\theta_k}} \left[\frac{\pi_{\theta}(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s,a) \right] $

and $latex \bar{D}{KL}(\theta || \theta_k)$ is an average KL-divergence between policies across states visited by the old policy:

$latex \bar{D}{KL}(\theta || \theta_k) = \mathbb{E}{s \sim \pi{\theta_k}} \left[D_{KL}\left(\pi_{\theta}(\cdot|s) || \pi_{\theta_k} (\cdot|s) \right) \right] $

The theoretical TRPO update isn't the easiest to work with, so TRPO makes some approximations to get an answer quickly. We Taylor expand the objective and constraint to leading order around $latex \theta_k$:

$latex \displaystyle \begin{array}{lcl} {\mathcal L}(\theta_k, \theta) &\approx & g^T (\theta - \theta_k) \\ \bar{D}_{KL}(\theta || \theta_k) &\approx & \frac{1}{2}(\theta - \theta_k)^T H (\theta - \theta_k) \end{array}$

resulting in an approximate optimization problem, $latex \displaystyle \theta*{k+1} = *arg* *max*{\theta} g^T (\theta - \theta_k), \quad \quad \text{s.t.} \frac{1}{2} (\theta - \theta_k)^T H (\theta - \theta_k) \leq \delta. $

where $latex H$ is the Hessian matrix. This approximate problem can be analytically solved by the methods of Lagrangian duality, yielding the solution:

$latex \theta_{k+1} = \theta_k + \sqrt{\frac{2 \delta}{g^T H^{-1} g}} H^{-1} g$

If we were to stop here, and just use this final result, the algorithm would be exactly calculating the Natural Policy Gradient. A problem is that, due to the approximation errors introduced by the Taylor expansion, this may not satisfy the KL constraint, or actually improve the surrogate advantage. TRPO adds a modification to this update rule: a backtracking line search,

$latex \theta_{k+1} = \theta_k + \alpha^j \sqrt{\frac{2 \delta}{g^T H^{-1} g}} H^{-1} g$

where $latex \alpha \in (0,1)$ is the backtracking coefficient, and $latex j$ is the smallest nonnegative integer such that $latex \pi_{\theta_{k+1}}$ satisfies the KL constraint and produces a positive surrogate advantage.

Lastly: computing and storing the matrix inverse, `$latex H^{-1}$`, is painfully expensive when dealing with neural network policies with thousands or millions of parameters. TRPO sidesteps the issue by using the conjugate gradient algorithm to solve `$latex Hx = g for x = H^{-1} g$`, requiring only a function which can compute the matrix-vector product `$latex Hx$` instead of computing and storing the whole matrix `$latex H$` directly. This is not too hard to do: we set up a symbolic operation to calculate

`$latex Hx = \nabla_{\theta} \left( \left(\nabla_{\theta} \bar{D}_{KL}(\theta || \theta_k)\right)^T x \right)$`

which gives us the correct output without computing the whole matrix.

Compared to PPO, TRPO is a more complicated algorithm that empirically does not seem to yield any significantly better results than PPO. Due to the complexity of TRPO, PPO is a more popular algorithm.

Generic Template for Policy Gradient Reinforcement Learning

As you may have noticed, many reinforcement learning algorithms follow the same structure of calculating the log probabilities of state-action pairs multiplied by some weighting term. As discussed in class, we provide a generic template for policy gradient reinforcement learning.

For m batches, $latex `\displaystyle \nabla \theta \hat{J(\theta)} = \frac{1}{m} \sum` {i=1}^m `\left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta (a_t^i \mid s_t^i) * \Psi_t^i \right]` $ There are several reinforcement learning algorithms that simply involve substituting `$latex \Psi_t^i$` with the corresponding algorithm.

REINFORCE: This algorithm does not use the “rewards to go” trick, nor does it use baseline subtraction. `$latex \Psi_t^i = \sum_{t'=0}^{T-1} r(s_{t'}^i, a_{t'}^i)$` Reward to Go: This algorithm simply uses the “rewards to go” trick. `$latex \Psi_t^i = \sum_{t'=t}^{T-1} r(s_{t'}^i, a_{t'}^i)$` Baseline Corrected Rewards to Go: We combine baseline correction with rewards to go such that `$latex \Psi_t^i = \sum_{t'=0}^{T-1} r(s_{t'}^i, a_{t'}^i) - V_w^\pi(s_t^i)$` `$latex J(w) = \frac{1}{2} \mathbb{E} ||\Psi_t||_2^2$` Temporal-Difference Policy Gradient: From Baseline Corrected Rewards to Go, we reparameterize the summation of the rewards as the sum of the reward at the current time step `$latex t$` and the estimated value function at the future time step `$latex t+1$`. `$latex \Psi_t^i = r(s_t^i, a_t^i) +`

`V_w^\pi(s_{t+1}^i) - V_w^\pi (s_t^i)$` `$latex J(w) = \frac{1}{2} \mathbb{E} || r(s_t^i, a_t^i) + V_w^\pi(s_{t+1}^i) - V_w^\pi (s_t^i) ||_2^2$` Actor-Critic: The actor-critic algorithm is discussed above in its own section `$latex \Psi_t^i = r(s_t^i, a_t^i) + \max_{a_{t+1}^i} V_w^{\pi}(s_{t+1}^i, a_{t+1}^i) - \sum_{a^i} \pi_\theta (a^i \mid s^i) Q(s^i,a^i) $` Reinforcement Learning from Human Feedback (RLHF)

We now turn our attention to from the various reinforcement learning algorithms to a specialized application: Reinforcement Learning from Human Feedback (RLHF):

Making language models bigger does not inherently make them better at following a user's intent. For example, large language models (LLMs) can generate outputs that are untruthful, toxic, or simply not helpful to the user. In other words, these models are not aligned with their users. To solve this challenge, we can finetune LLMs using our previously developed PPO algorithm. Our setup is as follows:

Agent: the language model itself. State: the prompt (input tokens). Action: which token is selected as the next token Reward model: the language model should be rewarded for generating "good responses" and should not receive any reward for generating "bad responses". Policy: In case of LLMs, the policy is the LLM itself! By design, LLMs model the probability of the next token given the context, i.e., the action space given the current state of the agent `$latex a_t \sim \pi(\cdot \mid s_t)$`. Trajectories

In RLHF, we utilize the following trajectories:

In particular, we take the starting state `$latex s_0$` to be some sequence of words such as "Where is Shanghai?". When this is fed to an LLM, we obtain the probability of the next token given this context, thus our first action `$latex a_t \sim \pi(\cdot \mid s_t)$` is the next word the LLM predicts (note that LLMs do not use full words as tokens, but we shall proceed with this simplified view for clarity).

Following, we take `$latex s_t$` to be the concatenation of `$latex s_{t-1}$` with the word resulting from the last action `$latex a_{t-1}$`. Hence, observe that this design indeed satisfies the Markov Property, where:

`$latex P(s_{t+1} = s' \mid s_0, a_0, s_1, a_1, \ldots , s_t, a_t) = P(s_{t+1} \mid s_t, a_t)$`

which allows us to rewrite:

`$latex P(\tau) = \prod_{t=0}^{T-1} P(s_{t+1} \mid s_t, a_t)$`

Hence, sampling trajectories equates to using an LLM to generate language, token by token, starting from some initial `$latex s_0$` context.

Reward Model

Modeling the reward is the key innovation in RLHF. In particular, we need some way to output large rewards for “good responses” and small/no rewards for “bad responses”, where “good” and “bad” aligns our notions of how we’d like our LLM to behave. While it’s complicated for us to rank a response on its own, we are great at comparing which answer among `$latex 2$` is better. Consider the following small dataset of sentences:

When presented with `$latex 2$` potential choices, it’s much simpler to see which response is closer to the desired behavior of the LLM. Hence, we utilize these ranked preferences to train a model to assign a score to a given answer as follows:

As shown above, we concatenate the question and answer to be fed to a separate LLM (reward model) and use a linear layer with one output feature after the last hidden state. This output feature denotes the associated reward for each question+answer pair. Now, in training this LLM, we utilize the following loss function:

$latex Loss = - `\log \sigma` (r(x, y_w) - r(x, y_l)) $ where `$latex x$` denotes the question, `$latex y_w$` the “good” response and `$latex y_l$` the “bad” response. Let’s delve into the intuition behind this loss: Utilizing a sigmoid `$latex \sigma$` maps our reward differences `$latex r(x, y_w) - r(x, y_l)$` to `$(0, 1)$`. In particular:

Model outputs `$latex r(x, y_w) > r(x, y_l )$`: When the model correctly assigns a higher reward to the “good” response, output values ranging in `$latex (0.5, 1)$` are fed to `$latex -\log(\cdot)$` resulting in a small loss.

Model outputs `$latex r(x, y_w) < r(x, y_l)$`: When the model incorrectly assings a lower reward to the “bad” response, output values ranging in `$latex (0, 0.5)$` are fed to `$latex -\log(\cdot)$` resulting in a higher loss, especially for values near `$latex 0$`.

Value Function

Finally, in estimating the value function $latex V^\pi(s_t)$, we add an additional Linear layer on top of our Language Model (the policy $latex \pi_\theta$) that estimates the value of the state at a particulate time step:

In training the model, we use the same formulations discussed previously, that is,

Begin by initializing $latex {V_\phi}_0^\pi$ Collect trajectories $latex \tau_1, \ldots , \tau_m$ Regress against empirical return:

$latex \displaystyle \phi*{i+1} \gets {\arg\min} \phi \frac{1}{m} \sum*{i=1}^m \sum{t=0}^{T-1} \left({V_\phi} \pi (s_t{(i)}) - \left(\sum_{k=t}^{T-1} r(s_k^{(i)}, a_k^{(i)}) \right) \right)^2 $

Conclusion & Limitations

In summary, we have explored the mathematical formulations and derivations underlying Proximal Policy Optimization (PPO) and demonstrated its application in Reinforcement Learning from Human Feedback (RLHF), particularly within the context of large language models (LLMs). We conclude by presenting some limitations:

RL is difficult to train: Compared to directly fine-tuning LLMs, fine-tuning using RL presents serious difficulties. We direct readers to Direct Preference Optimization (DPO), which eliminates the need for sampling from the LM during fine-tuning or performing significant hyperparameter tuning. Cost of Human Preference Data: Gathering firsthand human input is expensive, creating a scalability bottleneck for RLHF. Methods like reinforcement learning from AI feedback (RLAIF) have been proposed to reduce reliance on costly human feedback. Fallibility and Malice in Human Evaluators: Human feedback is not always reliable; it can be influenced by contrarian views or even malicious intent. Research suggests the necessity for methods that assess the credibility of human input and safeguard against adversarial behaviors. Overfitting and Bias in RLHF: If human feedback is sourced from a limited demographic, the resulting model may exhibit biases or perform poorly when exposed to diverse groups or topics outside the evaluators' familiarity.