GRPO

Shreeyam Bangera and Ankur Dahiya

April 2025

1 Introduction

LLMs or Large Language Models are the talk of the hour, with everyone using ChatGPT, Copilot, Claude, Gemini and other famous LLMs for generating human like text, generating code, research and many other things. One may ask as to how they are able to get access and knowledge of all this worldly knowledge? This is done by a process called "**pretraining**" where they are fed huge corpus of text from the internet, various books, articles, forums and any other publicly available data source with the sole purpose of predicting the next probable word while generating.

But pretraining on its own is not enough for getting a model that gives optimized outputs aligning itself to the user's prompts. This is because pretraining is like giving the model a lot of general knowledge, i.e. not providing it with specific skills for a particular task. Fine-tuning solves this particular problem as it helps the model to learn various patterns by training it a bit more on the particular task related dataset and tuning the model's weights to the task allowing the model to apply the knowledge learnt(during pretraining) to the task given by the user.

But even after all this, the model might generate texts or information that might not align with human values, for example: if any student asks the way to deal with increasing competition in college exams, then instead of helping him/her to improve the studies, it may suggest to sabotage other students. This is the problem of **AI disalignment**. Text generated like this may be toxic, biased on various premises cause harm and even at times provide faulty and incorrect responses.

AI alignment can be done by various methods but the most famous and widely used method is that of Reinforcement Learning (RL). It usually uses a reward model for teaching the AI to align itself to human measures and optimize it correctly, giving it reward for each correct generation and a negative reward for each incorrect one. Other than using the reward model, Reinforcement Learning with Human Feedback (RLHF) has been an impactful technique for training modern language models such as ChatGPT (like when you are asked your preferred response by ChatGPT while is generates 2 responses). In RLHF, the model is fine-tuned based on scores/labels provided by humans via various

approaches like PPO, DPO and GRPO which would be expanded on in this blog.

2 Prerequisites Crash Course (LLMs and RL)

A) Large Language Models

Word2Vec learns word representations by maximizing the probability of context words given a target word using the Skip-Gram model:

$$\max \sum_{(w,c)\in D} \log P(c \mid w) \quad \text{where} \quad P(c \mid w) = \frac{e^{\vec{v}_c \cdot \vec{v}_w}}{\sum_{c' \in V} e^{\vec{v}_{c'} \cdot \vec{v}_w}}$$

LLMs model the probability of the next word in a sequence, given the previous ones. Mathematically:

$$P(w_1, w_2, ..., w_n) = \prod_{t=1}^{n} P(w_t \mid w_1, ..., w_{t-1})$$

The model's parameters are optimized by minimizing the negative log-likelihood loss over a large corpus using stochastic gradient descent (SGD) or its variants:

$$\mathcal{L} = -\sum_{t=1}^{n} \log P(w_t \mid w_{<t})$$

This is pretty much all one needs to know about LLMs to understand RLHF.

B) Reinforcement Learning Crash Course

State: The state s_t represents the environment's configuration at time t.

$$s_t \in \mathcal{S}$$

Agent: The agent observes the state and selects actions to maximize cumulative reward.
Reward: The reward r_t is the feedback received after taking action a_t in state s_t.

$$r_t = R(s_t, a_t)$$

Policy: The policy π defines a probability distribution over actions given a state.

$$\pi(a \mid s) = P(a_t = a \mid s_t = s)$$

3 An overview of the Classic Re-inforcement Learning from Human Feedback

The agent is the language model, the state is the query and the token output of the model, the reward is the ranking of the output to the queries given by the human, and the policy is the parameters of the model.

Lemma 1 (Stochastic Transitions): We model the next state as stochastic, i.e.,

$$s_{t+1} \sim P(s_{t+1} \mid s_t, a_t)$$

Trajectory Probability: The probability of a trajectory τ under policy π is given by:

$$P(\tau \mid \pi) = \rho_0(s_0) \prod_{t=0}^{T-1} P(s_{t+1} \mid s_t, a_t)\pi(a_t \mid s_t)$$

Lemma 2 (Discounted Rewards): We discount rewards since immediate rewards are preferred:

$$G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}, \quad \text{where } \gamma \in [0, 1)$$

Trajectories are basically a series of states and actions. The goal is to select a policy that maximizes the expected return:

$$\pi^* = \arg\max_{\pi} J(\pi)$$

The function $J(\pi)$ represents this expected return. It is calculated by averaging the total rewards $R(\tau)$ received over all possible trajectories τ, weighted by how likely each trajectory is under the policy π. In other words, the better the policy, the more likely it is to generate high-reward trajectories:

$$J(\pi) = \int_{\tau} P(\tau \mid \pi)R(\tau) = E_{\tau\sim\pi}[R(\tau)].$$

To maximize the expected return in LLMs where the policy is parameterized by θ, we use gradient ascent as follows:

$$\theta_{k+1} = \theta_k + \alpha \nabla_\theta J(\pi_\theta)\big|_{\theta_k}$$

Now the goal is to find an expression of 'J' and compute it. Of course it is computationally impossible to calculate the return over all possible trajectories. Therefore we approximate it as:

$$\hat{g} = \frac{1}{|\mathcal{D}|} \sum_{\tau\in\mathcal{D}} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)R(\tau)$$

The original gradient estimator has pretty high variance because it dumps the entire return $R(\tau)$ on every action taken during the episode, even if that action had little to do with the final reward. This ends up making learning noisy and unstable. Now, thanks to the Central Limit Theorem, we know that as we collect more data, our estimate should eventually converge to the true gradient—but high variance means we need a *lot* of data to get there.
To deal with this, we switch to using the *advantage function*, defined as $A^\pi(s_t, a_t) = Q^\pi(s_t, a_t) - V^\pi(s_t)$. It tells us how much better (or worse) an action is compared to what the agent would normally do in that state. So instead of crediting every action equally with the total return, we adjust for how good the action actually was. This gives us a new and improved gradient estimator:

$$\hat{g} = \frac{1}{|\mathcal{D}|} \sum_{\tau \in \mathcal{D}} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) A^\pi(s_t, a_t),$$

which is still unbiased but way less noisy, making training smoother and more efficient.

3.1 Advantage Function and Its Estimation

The *advantage function* quantifies the relative benefit of taking a particular action in a given state, compared to the average performance of the policy from that state. It is defined as:

$$A^\pi(s, a) = Q^\pi(s, a) - V^\pi(s)$$

- $Q^\pi(s, a)$ is the expected return when the agent starts in state s, takes action a, and then follows the policy π thereafter:

$$Q^\pi(s, a) = E_\pi \left[\sum_{t=0}^{\infty} \gamma^t r_t \,\middle|\, s_0 = s, a_0 = a \right]$$

- $V^\pi(s)$ is the expected return when the agent starts in state s and follows the policy π from the beginning, with the first action also sampled from π:

$$V^\pi(s) = E_{a \sim \pi(\cdot|s)} \left[Q^\pi(s, a) \right] = E_\pi \left[\sum_{t=0}^{\infty} \gamma^t r_t \,\middle|\, s_0 = s \right]$$

Intuitively, $A^\pi(s, a)$ measures how much better (or worse) an action a is than what the policy would typically do in state s.

3.1.1 Monte Carlo Estimation

Monte Carlo (MC) methods estimate returns by sampling entire trajectories (episodes) and using the observed total return from a state (or state-action pair) as an unbiased estimator of expected return.

Let G_t denote the total return starting from time t:

$$G_t = \sum_{l=0}^{T-t-1} \gamma^l r_{t+l}$$

Then, the MC estimate of the advantage is:

$$\hat{A}^\pi(s_t, a_t) = G_t - \hat{V}^\pi(s_t)$$

where $\hat{V}^\pi(s_t)$ is estimated as the average of G_t's over all times s_t is visited.

Intuition: This approach directly compares what actually happened (via the observed return) to what the policy would expect from that state on average.

3.1.2 Temporal-Difference (TD) Estimation

TD methods bootstrap from the value of the next state to estimate returns, which allows for online and incremental learning. The 1-step TD error is defined as:

$$\delta_t = r_t + \gamma \hat{V}^\pi(s_{t+1}) - \hat{V}^\pi(s_t)$$

This TD error serves as a low-variance, biased estimator of the advantage:

$$\hat{A}^\pi(s_t, a_t) \approx \delta_t$$

Intuition: Instead of waiting to see how the episode ends, TD uses the immediate reward and the estimated future return to approximate the advantage.

3.1.3 Generalized Advantage Estimation (GAE)

Generalized Advantage Estimation (GAE) provides a principled way to interpolate between the high-variance MC estimator and the high-bias TD estimator. It does so by taking an exponentially weighted sum of k-step TD errors.
Let δ_t be the 1-step TD error as before. Then, GAE is defined as:

$$\hat{A}_t^{\mathrm{GAE}(\gamma,\lambda)} = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l}$$

For finite trajectories, this is truncated at the episode end. Alternatively, it can be computed efficiently in reverse via the recursion:

$$\hat{A}_t = \delta_t + \gamma\lambda \hat{A}_{t+1}$$

Parameters:

- γ: discount factor, controlling horizon of future rewards.
- λ: GAE parameter, controlling the trade-off between bias and variance.

Interpretation:

- When $\lambda = 0$, GAE reduces to 1-step TD: fast and low variance but biased.
- When $\lambda = 1$, GAE becomes equivalent to the MC estimate: unbiased but high variance.
- Intermediate λ values allow tuning the bias-variance tradeoff.

3.1.4 Summary Table

Method	Estimator	Bias	Variance
Monte Carlo	$G_t - \hat{V}(s_t)$	Low	High
TD(0)	$r_t + \gamma \hat{V}(s_{t+1}) - \hat{V}(s_t)$	High	Low
GAE(λ)	$\sum_l (\gamma\lambda)^l \delta_{t+l}$	Tunable	Tunable

3.2 Failure Modes of Vanilla Policy Gradient (VPG)

The Vanilla Policy Gradient (VPG) method attempts to maximize the expected return by directly optimizing:

$$J(\theta) = E_{\pi_\theta}\left[\sum_{t=0}^{T} \gamma^t r_t\right]$$

Using the policy gradient theorem, the gradient is:

$$\nabla_\theta J(\theta) = E_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a_t|s_t) \cdot \hat{A}_t\right]$$

The VPG loss is defined as:

$$L^{\text{VPG}}(\theta) = E_t\left[\log \pi_\theta(a_t|s_t) \cdot \hat{A}_t\right]$$

Mathematical Issues:

1. **Unconstrained Update Magnitude:** The policy π_θ is updated without any mechanism to control how far it moves from the original policy $\pi_{\theta_{\text{old}}}$. A single large gradient step can lead to:

   $$\pi_\theta(a|s) \ll \pi_{\theta_{\text{old}}}(a|s) \quad \text{even if } \hat{A}(s,a) > 0$$

   This destroys the probability of good actions and leads to performance collapse.

2. **Distribution Mismatch:** The trajectories are sampled from $\pi_{\theta_{\text{old}}}$, but the gradient is applied to π_θ. The advantage estimates $\hat{A}_t$ are only valid under $\pi_{\theta_{\text{old}}}$, and large updates make them invalid for π_θ.

3. **High Variance and Instability:** Without any regularization or trust region, the updates are sensitive to noise in advantage estimates, leading to high variance and poor convergence.

3.3 Derivation of the PPO Objective

To address these issues, Proximal Policy Optimization (PPO) introduces a clipped surrogate objective that discourages large policy updates.

Step 1: Define the Probability Ratio Let $\pi_{\theta_{\text{old}}}$ be the current policy and π_θ the new policy. Define the probability ratio:

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$$

Step 2: Surrogate Objective We want to improve the policy by maximizing the expected advantage weighted by this ratio:

$$L^{\text{CPI}}(\theta) = E_t\left[r_t(\theta) \cdot \hat{A}_t\right]$$

This is the basis for Conservative Policy Iteration. However, this still allows for large updates if $r_t(\theta)$ becomes too large or too small.

Step 3: Clipped Objective PPO introduces a clipped surrogate loss:

$$L^{\text{CLIP}}(\theta) = E_t\left[\min\left(r_t(\theta) \cdot \hat{A}_t,\ \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon) \cdot \hat{A}_t\right)\right]$$

Interpretation:

- If $\hat{A}_t > 0$: the objective increases with r_t, but is capped at $1+\epsilon$.
- If $\hat{A}_t < 0$: the objective decreases with r_t, but is floored at $1-\epsilon$.
- This prevents the optimizer from moving π_θ too far from $\pi_{\theta_{\text{old}}}$.

Final PPO Objective: In practice, the complete PPO loss also includes a value function loss and an entropy bonus:

$$L^{\text{PPO}}(\theta) = E_t\left[L_t^{\text{CLIP}}(\theta) - c_1 \cdot (V_\theta(s_t) - V_t^{\text{target}})^2 + c_2 \cdot \mathcal{H}[\pi_\theta](s_t)\right]$$

Where:

- c_1 weights the value function MSE loss
- c_2 weights the entropy bonus
- $\mathcal{H}[\pi]$ encourages exploration by maximizing policy entropy

4 Direct preference optimization

5 Group Relative Policy Optimization