Yuge (Jimmy) Shi
Senior Research Scientist, Google DeepMind
Follow

A vision researcher's guide to some RL stuff: PPO & GRPO

20 minute read

Published: January 31, 2025

> *First up, some rambles as usual.*

It has been a while since I last wrote a blog post. Life has been hectic since I started work, and the machine learning world is also not what it was since I graduated in early 2023. Your average parents having LLM apps installed on their phones is already yesterday's news – I took two weeks off work to spend Lunar New Year in China, which only serves to give me **plenty** of time to scroll on twitter and witness DeepSeek's (quite well-deserved) hype peak on Lunar New Year's eve while getting completely overwhelmed.

So this feels like a good time to read, learn, do some basic maths, and write some stuff down again.

What this blog post covers, and who is it for

This is a deep dive into Proximal Policy Optimization (PPO), which is one of the most popular algorithm used in RLHF for LLMs, as well as Group Relative Policy Optimization (GRPO) proposed by the DeepSeek folks, and there's also a quick summary of the tricks I find impressive in the DeepSeek R1 tech report in the end.

This is all done by someone who's mostly worked on vision and doesn't know much about RL. If that's you too, I hope you will find this helpful.

LLM pre-training and post-training

The training of an LLM can be separated into a pre-training and post-training phase:

1. **Pre-training:** the classic "throw data at the thing" stage where the model is trained to do next token prediction using large scale web data;

2. **Post-training:** This is where we try to improve the model's reasoning capability. Typically there are two stages to post-training, namely
    - ***Stage 1: SFT (Supervised Finetuning)***: as the name implies, we use supervised learning first by fine-tuning the LLM on a small amount of high quality *expert reasoning data*; think instruction-following, question-answering and/or chain-of-thoughts. The hope is, by the end of this training stage, the model has learned how to mimic expert demonstrations. This is obviously the ideal way to learn if we had unlimited amount of high quality, expert data, but since we don't –
    - ***Stage 2: RLHF (Reinforcement Learning from Human Feedback)***: Not enough human expert reasoning data? This is where ✨ RL ✨ gets to shine! RLHF uses human feedback to train a reward model, which then guides the LLM's learning via RL. This aligns the model with nuanced human preferences, which…I think we all agree is important 📎.

DeepSeek's ultra efficient post-training

Notably, one of the most surprising thing about the DeepSeek R1 tech report is that their R1-zero model completely skips the SFT part and applies RL directly to the base model (DeepSeek V3). There are a few benefits to this:

- **Computational efficiency**: skipping one stage of post-training brings computational efficiency;
- **Open-ended learning**: Allows the model to "self-evolve" reasoning capabilities through exploration;
- **Alignment**: Avoiding biases introduced by human-curated SFT data.

Caveat*: while it seems like a "duh" moment to see someone saving compute by skipping a whole stage of post-training, I suspect you won't be able to pull it off without a very good base model.*

But they didn't stop there! DeepSeek also make the RLHF part more efficient by introducing GRPO to replace PPO, which eliminates the need for a separate critic model (typically as large as the policy model), reducing memory and compute overhead by ~50%. To see why and how they did this, and for our own intellectual indulgence, let's now have a look at exactly how RLHF is done and where these algorithms comes in.

RLHF

Let's break down the workflow of RLHF into steps:

- **Step 1**: For each prompt, sample multiple responses from the model;
- **Step 2**: Humans rank these outputs by quality;
- **Step 3**: Train a **reward model** to predict human preferences / ranking, given any model responses;
- **Step 4**: Use **RL (e.g. PPO, GRPO)** to fine-tune the model to maximise the reward model's scores.

As we can see the process here is relatively simple, with two learnable components, i.e. the **reward model** and **"the RL"**. Now let's dive into each component with more details.

Reward Model

The reward model is truly on the front-line of automating jobs: realistically, we can't have humans rank all the outputs of the model. A cost-saving approach is to then have annotators rate a small portion of the LLM outputs, then **train a model to predict these annotators' preferences** — and that is where the reward model comes in. With that said, now let's look at some maths:

Let's denote our learnable reward model as R_ϕ. Given a prompt p, the LLM generate N responses $r_1, r_2, \ldots r_N$. Then given that a response r_i is preferrable to r_j according to the human rater, the reward model is trained to minimise the following objective:

$$\mathcal{L}(\phi) = -\log \sigma(R_\phi(p, r_i) - R_\phi(p, r_j)), \tag{1}$$

where σ denotes the sigmoid function.

> ***Side note**: The objective is derived from the **Bradley-Terry model**, which defines the probability that a rater prefers r_i over r_j as:*
> $$P(r_i \succ r_j) = \frac{\exp\left(R_\phi(p,r_i)\right)}{\exp\left(R_\phi(p,r_i)\right)+\exp\left(R_\phi(p,r_j)\right)}.$$
> *Taking the negative log-likelihood of this probability gives the loss $\mathcal{L}(\phi)$ above. The sigmoid σ emerges naturally from rearranging the Bradley-Terry ratio.*

Note that the reward for a partial response is always 0; only for complete responses from the LLM would the reward model return a non-zero scalar score. This important fact will become relevant later.

"The RL part": PPO

> *This part is only for the readers who are curious about PPO, and you don't really need to understand this if your goal of opening this blog post is to understand GRPO. All I can say is though it brought me great joy to finally understand how PPO works, and then great sense of vindication when I realised how much simpler GRPO is compared to PPO. So if you're ready for an emotional rollercoaster – let's dive in.*

First, a high level overview. PPO stands for proximal policy optimization, and it requires the following components:

- **Policy (π_θ)**: the LLM that has been pre-trained / SFT'ed;
- **Reward model (R_ϕ)**: a trained and frozen network that provides scalar reward given **complete response** to a prompt;
- **Critic (V_γ)**: also known as value function, which is a learnable network that takes in **partial response** to a prompt and predicts the scalar reward.

Congratulations – by calling the LLM a "policy" you are now an RL person 🎉! The purpose of each component becomes a little clearer once we get to know the workflow, which contains five stages:

1. **Generate responses:** LLM produces multiple responses for a given prompt;
2. **Score responses:** The reward model assigns reward for each response;
3. **Compute advantages:** Use GAE to compute advantages (more on this later, it's used for training the LLM);
4. **Optimise policy:** Update the LLM by optimising the total objective;
5. **Update critic:** train the value function to be better at predicting the rewards given partial responses.

Now let's take a look at some of these stages/components in more details, and then see how they all come together.

Terminologies: states and actions

Some more RL terminologies before we move on. In the discussion of this section we are going to use the term **state**, denote as s_t, and **action**, denote as a_t. Note that here the subscript t is used to denote the state and action at a **token level**; in contrast, previously when we defined our prompt p and responses r_i, the subscript i is used to denote the response at an **instance level**.

To make this a little clearer, let’s say we give our LLM a prompt p. The LLM then starts generating a response r_i of length T one token at a time:

- $t = 0$: our state is just the prompt, i.e. $s_0 = p$, and the first action a_0 is just the first word token generated by the LLM;
- $t = 1$: the state becomes $s_1 = p, a_0$, as the LLM is generating the next action a_1 while conditioned on the state; …
- $t = T - 1$: the state is $s_{T-1} = p, a_{0:T-2}$, and the LLM generates the final action a_{T-1}.

Connecting this to the previous notations again, all the actions stringing together makes one response, i.e. $r_i = a_0, a_1, \dots a_{T-1}$.

General Advantage Estimation (GAE)

Our policy is updated to optimise **advantage** – intuitively, it defines how much better a **specific action** a_t (i.e. word) is compared to an **average action** the policy will take in state s_t (i. e. prompt + generated words so far). Formally:

$$A_t = Q(s_t, a_t) - V(s_t) \tag{2}$$

Where $Q(s_t, a_t)$ is the expected cumulative reward of taking a specific action a_t in state s_t, and $V(s_t)$ is the expected cumulative reward of average action the policy takes in state s_t.

There are two main ways of estimating this advantage, each with their trade-offs, namely, 1) **Monte-Carlo (MC)**: Use the reward of the full trajectory (i.e. full responses). This approach has high variance due to the sparse reward – it is expensive to take enough samples from the LLM to optimise using MC, but it does have low bias as we can accurately model the reward; 2) **Temporal difference (TD)**: Use one-step trajectory reward (i.e. measure how good is the word that’s just been generated given the prompt). By doing so we can compute reward on a token level, which significantly reduces the variance, but at the same time the bias goes up as we can’t as accurately anticipate the final reward from a partially generated response.

This is where GAE comes in – it is proposed to **balance the bias and variance through a multi-step TD**. However, recall that previously we mentioned that the reward model will return 0 if the response was incomplete: how will we compute TD without knowing how the reward would change before and after generating a word? We therefore introduce a model that does just that, which we call "the critic".

The critic (value function) 🕵️

The critic is trained to **anticipate the final reward given only a partial state**, so that we can compute the TD. Training the critic V_γ is fairly straightforward:

Given a partial state s_t, we want to predict the reward model's output given the full state $s_T = p, r$. The objective for the critic can be written as

$$L(\gamma) = \mathbb{E}_t \left[(V_\gamma(s_t) - \mathrm{sg}(R_\phi(s_T)))^2 \right], \tag{3}$$

where sg denotes the stop gradient operation. As we can see, the critic is trained with a simple L2 loss to the reward model's score.

You might notice that while the reward model R_ϕ is trained before PPO and frozen, the critic is trained alongside the LLM, even though its job is also just to predict the reward. This is because the value function must estimate the reward for partial response given the **current policy**; as a result, it must be updated alongside the LLM, to avoid its predictions to become outdated and misaligned. And this, is what they call, actor-critic in RL (mic-drop).

Back to GAE

With the critic V_γ, we now have a way to anticipate the reward from a partial state. Now let's get on with GAE, which as mentioned computes a multi-step TD objective:

$$A_K^{\mathrm{GAE}} = \delta_0 + \lambda\delta_1 + \lambda^2\delta_2 ... + (\lambda)^{K-1}\delta_{K-1} = \sum_{t=0}^{K-1} (\lambda)^t \delta_t, \tag{4}$$

where K denotes the number of TD steps and $K < T$ (because obviously you can't compute TD beyond the length of the trajectory). δ_t denotes the TD error at step t, and is computed as:

$$\delta_t = V_\gamma(s_{t+1}) - V_\gamma(s_t) \tag{5}$$

To put simply, the TD error computes the difference between expected total reward of one time step, and A_K^{GAE} estimates advantage by computing the aggregated single-step TD errors over K steps. The λ in the GAE equation controls the trade-off between the variance and the bias: when $\lambda = 0$, GAE reduces to single-step TD; and when $\lambda = 1$, GAE becomes MC.

In RLHF, we want to maximise this advantage term, thereby maximising the reward for every token the LLM generates.

> ***Side note****: ok, I cut some corners for simplicity here. Originally there is also a discount factor* η *in GAE:* $A_K^{\text{GAE}} = \sum_{t=0}^{K-1} (\lambda\eta)^t \delta_t,$ *which is also used in the TD error* δ_t*, and there is also an extra reward term* $\delta_t = R_\phi(s_t) + \eta V_\gamma(s_{t+1}) - V_\gamma(s_t).$ *But since we almost always have* $\eta = 1$*, and* $R_\phi(s_t) = 0$ *for* $t < T$ *which is always the case, I took a shortcut to simplify and omit those terms.*

Putting it together – PPO objective

There are a few components to the PPO objective, namely 1) the clipped surrogate objective, 2) the entropy bonus, 3) the KL penalty.

1. The clipped surrogate objective

This is where we maximise A_K^{GAE}, so that each token the LLM predicted maximises the reward (or, by definition of advantage earlier, each token the LLM predicts should be much better than its average prediction). The clipped surrogate objective constrains policy updates with a probability ratio $c_t(\pi_\theta)$:

$$L^{\text{clip}}(\theta) = \mathbb{E}_t\left[\min(c_t(\pi_\theta)A_t^{GAE}, \text{clip}(c_t(\pi_\theta), 1-\epsilon, 1+\epsilon)A_t^{GAE})\right], \tag{6}$$

where ϵ controls the clipping range, $c_t(\pi_\theta)$ the probability ratio of predicting a specific token a_t at given cumulative state s_t, before and after the update:

$$c_t(\pi_\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}. \tag{7}$$

Concrete example:

- Let's say the LLM assigns the word `unlimited` with the following probabilities:

  - ***Before update***: 0.1,
  - ***After update***: 0.3. Then the probability ratio $c_t = 0.3/0.1 = 3$;
- If we take $\epsilon = 0.2$, c_t gets clipped to 1.2;
- The final clipped surrogate loss is $L^{\text{clip}}(\pi_\theta) = 1.2A_K^{\text{GAE}}$.

You can think of clipping as a way to prevent overconfidence – without clipping, a large A_K^{GAE} could cause the policy to overcommit to an action.

2. KL divergence penalty

Additionally, we have the KL divergence penalty which prevents the current policy θ from deviating too far from the original model that we are finetuning from θ_{orig}:

$$\text{KL}(\theta) = \mathbb{E}_{s_t}\left[\mathbb{D}_{\text{KL}}(\pi_{\theta\text{orig}}(\cdot|s_t)||\pi_\theta(\cdot|s_t))\right] \tag{8}$$

The KL is simply estimated by taking the average over sequence and batch.

Pseudocode:

```
# Compute KL divergence between original and current policy/model
logits_orig = original_model(states)  # Original model's logits
logits_current = current_model(states)  # Current model's logits

probs_orig = F.softmax(logits_orig, dim=-1)
log_probs_orig = F.log_softmax(logits_orig, dim=-1)
log_probs_current = F.log_softmax(logits_current, dim=-1)

kl_div = (probs_orig * (log_probs_orig - log_probs_current)).sum(dim=-1)
kl_penalty = kl_div.mean()  # Average over sequence and batch
```

3. Entropy bonus

The entropy bonus encourages exploration of LLM's generation by penalising low entropy:

$$H(\theta) = -\mathbb{E}_{a_t}[\log \pi_\theta(a_t|s_t)]. \tag{9}$$

Pseudocode:

```
# Compute entropy of current policy
probs_current = F.softmax(logits_current, dim=-1)
log_probs_current = F.log_softmax(logits_current, dim=-1)

entropy = -(probs_current * log_probs_current).sum(dim=-1)
entropy_bonus = entropy.mean()  # Average over sequence and batch
```

Finally, the PPO objective

Given the three terms above, in addition to the value function MSE loss (recall it is optimised along with the LLM), the PPO objective is defined as follows:

$$\mathcal{L}_{\text{PPO}}(\theta, \gamma) = \underbrace{\mathcal{L}_{\text{clip}}(\theta)}_{\text{aaximise reward}} + \underbrace{w_1 H(\theta)}_{\text{Maximise entropy}} - \underbrace{w_2 \text{KL}(\theta)}_{\text{Penalise KL divergence}} - \underbrace{w_3 \mathcal{L}(\gamma)}_{\text{Critic L2}} \quad (10)$$

A summary of the different terms in this objective is as follows:

Term	Purpose
$\mathcal{L}_{\text{clip}}(\theta)$	Maximize rewards for high-advantage actions (clipped to avoid instability).
$H(\theta)$	Maximize entropy to encourage exploration.
$\text{KL}(\theta)$	Penalize deviations from the reference policy (stability).
$\mathcal{L}(\gamma)$	Minimize error in value predictions (critic L2 loss).

"The RL part": GRPO

It's super easy to understand GRPO now that we have a good understanding of PPO, and the key difference lies in how the two algorithms estimate advantage A: instead of estimating advantage through the critic like in PPO, GRPO does so by taking multiple samples from the LLM using the same prompt.

Workflow:

1. For each prompt p, sample a group of N responses $\mathcal{G} = r_1, r_2, \ldots r_N$ from the LLM policy π_θ;
2. Compute rewards $R_\phi(r_1), R_\phi(r_2), \ldots R_\phi(r_N)$ for each response using the reward model R_ϕ;

3. Calculate group-normalised advantage for each response:
$$A_i = \frac{R_\phi(r_i) - \text{mean}(\mathcal{G})}{\text{std}(\mathcal{G})}, \tag{11}$$
where $\text{mean}(\mathcal{G})$ and $\text{std}(\mathcal{G})$ denotes the within-group mean and standard deviation, respectively.

A lot simpler, right? In GRPO, advantage is approximated as the normalised reward of each response within its group of responses. This removes the need of a critic network calculating per-step rewards, not to mention the mathematical simplicity and elegance. It does somewhat beg the question – why didn't we do this sooner?

> *I don't have a good answer to this question due to a lack of hands-on experience: I'm guessing this is tied to hardware capabilities, as the modern GPUs/TPUs we have access to these days make it possible to sample in a much faster and more efficient manner. Again I'm not an expert, so insights on this are very welcomed!*

> ***Update**: some insights from @him_sahni on this, who "did RL in his past life": **the reason "why no one has tried GRPO before" is – we have**. In REINFORCE, you update the policy by subtracting a baseline (typically the average reward from several trajectories) to reduce variability. In fact, theory shows that the ideal baseline is the total expected future reward from a state, often called the "value". Using a value function as the baseline is known as the actor-critic approach, and PPO is a stable version of that. Now, in traditional REINFORCE, the baseline can be any function of the current state, and traditionally is just the reward for the trajectories in a single batch; in GRPO, this baseline is computed over 1000 samples generated for each prompt, which is 🌈 novel 🌈.*

The GRPO objective

Similar to PPO, GRPO still make use of a **clipped surrogate loss** as well as the **KL penalty**. The entropy bonus term is not used here, as the group-based sampling already encourages exploration. The clipped surrogate loss is identical to the one used in PPO, but for completeness sake here it is:

$$\mathcal{L}_{\text{clip}}(\theta) =$$
$$\frac{1}{N}\sum_{i=1}^{N}\left(\min\left(\frac{\pi_\theta(r_i|p)}{\pi_{\theta_{\text{old}}}(r_i|p)}A_i,\ \text{clip}\left(\frac{\pi_\theta(r_i|p)}{\pi_{\theta_{\text{old}}}(r_i|p)}, 1-\epsilon, 1+\epsilon\right)A_i\right)\right),$$

then with the KL penalty term, the final GRPO objective can be written as:

$$\mathcal{L}_{\text{GRPO}}(\theta) = \underbrace{\mathcal{L}_{\text{clip}}(\theta)}_{\text{Maximise reward}} - \underbrace{w_1 \mathbb{D}_{\text{KL}}(\pi_\theta || \pi_{\text{orig}})}_{\text{Penalise KL divergence}} \tag{12}$$

More thoughts on R1: Brutal Simplicity

Finally, a few words on R1.

Overhyped or not, one thing that really stands out about the R1 from reading the paper is that it embraces a **stripped-down, no-nonsense approach** to LLM training, prioritising brutal simplicity over sophistication. GRPO is just the tip of the iceberg. Here are some more examples on of its brutal simplicity:

1. Rule-Based, Deterministic Rewards

- **What**: Abandon neural Process Reward Models (PRMs) or Outcome Reward Models (ORMs). Use **binary checks**, including:
  - **Answer Correctness**: Final answer matches ground truth (e.g., math solutions, code compilation).
  - **Formatting**: Force answers into `<think>...</think><answer>...</answer>` templates.
  - **Language Consistency**: Penalise mixed-language outputs (e.g., English reasoning for Chinese queries).
- **Why**: Deterministic rules sidestep **reward hacking** (e.g., models tricking neural reward models with plausible-but-wrong steps) and eliminate reward model training costs.

2. Cold-Start Data: Minimal Human Touch

- **What**: Instead of curating massive SFT datasets, collect **a few thousand high-quality CoT examples** via:
  - Prompting the base model with few-shot examples.
  - Light human post-processing (e.g., adding markdown formatting).
- **Why**: Avoids costly SFT stages while bootstrapping RL with "good enough" starting points.

3. Rejection Sampling: Filter Hard, Train Harder

- **What**: After RL training, generate **600k reasoning trajectories**, then **throw away all incorrect responses**. Only keep the "winners" (correct answers) for supervised fine-tuning (SFT). No fancy reranking, no preference pairs. Just survival-of-the-fittest filtering.
- **Why**: It works, why not!

4. Distillation: Copy-Paste Reasoning

- **What**: To train smaller models, directly fine-tune them on **800k responses** generated by DeepSeek-R1. No RL, no iterative alignment—just mimicry.
- **Why**: Smaller models inherit reasoning patterns discovered by the larger model's brute-force RL, bypassing costly RL for small-scale deployments.

DeepSeek-R1's design reflects a broader trend in AI: **scale and simplicity often outperform clever engineering**. By ruthlessly cutting corners — replacing learned components with rules, leveraging massive parallel sampling, and anchoring to pre-trained baselines — R1 achieves SOTA results with fewer failure modes. It's not elegant, but it's *effective*.

Who would've thought the best way to incentivise good thinking is to 🌈 **stop overthinking it** 🌈.

Tags: Large Language Models Machine Learning RLHF

Previous Next

LEAVE A COMMENT