11 Policy Gradient Algorithms

The algorithms that popularized RLHF for language models were policy-gradient reinforcement learning algorithms. These algorithms, such as PPO, GRPO, and REINFORCE, use recently generated samples to update their model rather than storing scores in a replay buffer. In this section we will cover the fundamentals of the policy gradient algorithms and how they are used in the modern RLHF framework.

At a machine learning level, this section is the subject with the highest complexity in the RLHF process. Though, as with most modern AI models, the largest determining factor on its success is the data provided as inputs to the process.

The most popular algorithms used for RLHF has evolved over time. When RLHF came onto the scene with ChatGPT, it was largely known that they used a variant of PPO, and many initial efforts were built upon that. Over time, multiple research projects showed the promise of REINFORCE style algorithms [128] [112], touted for its simplicity over PPO without a reward model (saves memory and therefore the number of GPUs required) and with simpler value estimation (no GAE). More algorithms have emerged, including Group Relative Policy Optimization, which is particularly popular with reasoning tasks, but in general many of these algorithms can be tuned to fit a specific task. In this chapter, we cover the core policy gradient setup and the three algorithms mentioned above due to their central role in the establishment of a canonical RLHF literature.

For definitions of symbols, see the problem setup chapter.

11.1 Policy Gradient Algorithms

Reinforcement learning algorithms are designed to maximize the future, discounted reward across a trajectory of states, $s \in \mathcal{S}$, and actions, $a \in \mathcal{A}$ (for more notation, see Chapter 3, Definitions). The objective of the agent, often called the *return*, is the sum of discounted, future rewards (where $\gamma \in [0, 1)$ is a factor that prioritizes near term rewards) at a given time t:

$$G_t = R_{t+1} + \gamma R_{t+2} + \cdots = \sum_{k=o}^{\infty} \gamma^k R_{t+k+1}. \tag{29}$$

The return definition can also be estimated as:

$$G_t = \gamma G_{t+1} + R_{t+1}. \tag{30}$$

This return is the basis for learning a value function $V(s)$ that is the estimated future return given a current state:

$$V(s) = \mathbb{E}[G_t | S_t = s]. \tag{31}$$

All policy gradient algorithms solve an objective for such a value function induced from a specific policy, $\pi(s|a)$.

Where $d_\pi(s)$ is the stationary distribution of states induced by policy $\pi(s)$, the optimization is defined as:

$$J(\theta) \ = \ \sum_s d_\pi(s) V_\pi(s), \tag{32}$$

The core of policy gradient algorithms is computing the gradient with respect to the finite time expected return over the current policy. With this expected return, J, the gradient can be computed as follows, where α is the learning rate:

$$\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta) \tag{33}$$

The core implementation detail is how to compute said gradient. Schulman et al. 2015 provides an overview of the different ways that policy gradients can be computed [129]. The goal is to *estimate* the exact gradient $g := \nabla_\theta \mathbb{E}[\sum_{t=0}^{\infty} r_t]$, of which, there are many forms similar to:

$$g = \mathbb{E}\Big[\sum_{t=0}^{\infty} \Psi_t \nabla_\theta \log \pi_\theta(a_t|s_t) \Big], \tag{34}$$

Where Ψ_t can be the following (where the rewards can also often be discounted by γ):

1. $\sum_{t=0}^{\infty} r_t$: total reward of the trajectory.
2. $\sum_{t'=t}^{\infty} r_{t'}$: reward following action a_t, also described as the return, G.
3. $\sum_{t'=t}^{\infty} r_{t'} - b(s_t)$: baselined version of previous formula.
4. $Q^\pi(s_t, a_t)$: state-action value function.
5. $A^\pi(s_t, a_t)$: advantage function, which yields the lowest possible theoretical variance if it can be computed accurately.
6. $r_t + V^\pi(s_{t+1}) - V^\pi(s_t)$: TD residual.

The *baseline* is a value used to reduce variance of policy updates (more on this below).

For language models, some of these concepts do not make as much sense. For example, we know that for a deterministic policy the value function is defined as $V(s) = \max_a Q(s, a)$ or for a stochastic policy as $V(s) = \mathbb{E}_{a \sim \pi(a|s)}[Q(s, a)]$. If we define $s + a$ as the continuation a to the prompt s, then $Q(s, a) = V(s + a)$, which gives a different advantage trick:

$$A(s, a) = Q(s, a) - V(s) = V(s + a) - V(s) = r + \gamma V(s + a) - V(s) \tag{35}$$

Which is a combination of the reward, the value of the prompt, and the discounted value of the entire utterance.

11.1.1 Vanilla Policy Gradient

The vanilla policy gradient implementation optimizes the above expression for $J(\theta)$ by differentiating with respect to the policy parameters. A simple version, with respect to the overall return, is:

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_\tau \left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) R_t \right] \tag{36}$$

A common problem with vanilla policy gradient algorithms is the high variance in gradient updates, which can be mitigated in multiple ways. In order to alleviate this, various techniques are used to normalize the value estimation, called *baselines*. Baselines accomplish this in multiple ways, effectively normalizing by the value of the state relative to the downstream action (e.g. in the case of Advantage, which is the difference between the Q value and the value). The simplest baselines are averages over the batch of rewards or a moving average. Even these baselines can de-bias the gradients so $\mathbb{E}_{a\sim\pi(a|s)}[\nabla_\theta \log \pi_\theta(a|s)] = 0$, improving the learning signal substantially.

Many of the policy gradient algorithms discussed in this chapter build on the advantage formulation of policy gradient:

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_\tau \left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) A^{\pi_\theta}(s_t, a_t) \right] \tag{37}$$

A core piece of the policy gradient implementation involves taking the derivative of the probabilistic policies. This comes from:

$$\nabla_\theta \log \pi_\theta(a|s) = \frac{\nabla_\theta \pi_\theta(a|s)}{\pi_\theta(a|s)} \tag{38}$$

Which is derived from the chain rule:

$$\nabla_\theta \log x = \frac{1}{x} \nabla_\theta x \tag{39}$$

We will use this later on in the chapter.

11.1.2 REINFORCE

The algorithm REINFORCE is likely a backronym, but the components of the algorithms it represents are quite relevant for modern reinforcement learning algorithms. Defined in the seminal paper *Simple statistical gradient-following algorithms for connectionist reinforcement learning* [130]:

> The name is an acronym for "REward Increment = Nonnegative Factor X Offset Reinforcement X Characteristic Eligibility."

The three components of this are how to do the *reward increment*, a.k.a. the policy gradient step. It has three pieces to the update rule:

1. Nonnegative factor: This is the learning rate (step size) that must be a positive number, e.g. α below.
2. Offset Reinforcement: This is a baseline b or other normalizing factor of the reward to improve stability.

3. Characteristic Eligibility: This is how the learning becomes attributed per token. It can be a general value, e per parameter, but is often log probabilities of the policy in modern equations.

Thus, the form looks quite familiar:

$$\Delta_\theta = \alpha(r - b)e \tag{40}$$

With more modern notation and the generalized return G, the REINFORCE operator appears as:

$$\nabla_\theta J(\theta) \;=\; \mathbb{E}_{\tau\sim\pi_\theta}\Big[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t \mid s_t)\,(G_t - b)\Big], \tag{41}$$

Here, the value $G_t - b(s_t)$ is the *advantage* of the policy at the current state, so we can reformulate the policy gradient in a form that we continue later with the advantage, A:

$$\nabla_\theta J(\theta) \;=\; \mathbb{E}_{\tau\sim\pi_\theta}\Big[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t \mid s_t)\,A_t\Big], \tag{42}$$

REINFORCE is a specific implementation of vanilla policy gradient that uses a Monte Carlo estimator of the gradient.

REINFORCE can be run without value network – the value network is for the baseline in the policy gradient. PPO on the other hand needs the value network to accurately compute the advantage function.

11.1.2.1 REINFORCE Leave One Out (RLOO) The core implementation detail of REINFORCE Leave One Out versus standard REINFORCE is that it takes the average reward of the *other* samples in the batch to compute the baseline – rather than averaging over all rewards in the batch [131], [128], [132].

Crucially, this only works when generating multiple responses per prompt, which is common practice in multiple domains of finetuning language models with RL.

Specifically, for the REINFORCE Leave-One-Out (RLOO) baseline, given K sampled trajectories or actions $a_1, \ldots, a_K$, to a given prompt s we define the baseline explicitly as the following *per-prompt*:

$$b(s, a_k) = \frac{1}{K-1} \sum_{i=1, i\neq k}^{K} R(s, a_i), \tag{43}$$

resulting in the advantage:

$$A(s, a_k) = R(s, a_k) - b(s, a_k). \tag{44}$$

Equivalently, this can be expressed as:

$$A(s, a_k) = \frac{K}{K-1}\left(R(s, a_k) - \frac{1}{K}\sum_{i=1}^{K} R(s, a_i)\right). \tag{45}$$

This is a simple, low-variance advantage update that is very similar to GRPO, which will be discussed later, where REINFORCE is used with a different location of KL penalty and without step-size clipping. Still, the advantage from RLOO could be combined with the clipping of PPO, showing how similar many of these algorithms are.

RLOO and other algorithms that do not use a value network assign the advantage (or reward) of the sequence to every token for the loss computation. Algorithms that use a learned value network, such as PPO, assign a different value to every token individually, discounting from the final reward achieved at the EOS token. For example, with the KL divergence distance penalty, RLOO sums it over the completion while PPO and similar algorithms compute it on a per-token basis and subtract it from the reward (or the advantage, in the case of GRPO). These details and trade-offs are discussed later in the chapter.

11.1.3 Proximal Policy Optimization

Proximal Policy Optimization (PPO) [133] is one of the foundational algorithms to Deep RL's successes (such as OpenAI's DOTA 5 [134] and large amounts of research). The loss function is as follows per sample:

$$J(\theta) = \min\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_{old}}(a|s)}A, \text{clip}\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_{old}}(a|s)}, 1-\varepsilon, 1+\varepsilon\right)A\right). \tag{46}$$

For language models, the loss is computed per token, which intuitively can be grounded in how one would compute the probability of the entire sequence of autoregressive predictions – by a product of probabilities. From there, the common implementation is with *log-probabilities* that make the computation far more tractable.

$$J(\theta) = \frac{1}{|a|}\sum_{t=0}^{|a|}\min\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}A_t, \text{clip}\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}, 1-\varepsilon, 1+\varepsilon\right)A_t\right). \tag{47}$$

This is the per-token version of PPO, which also applies to other policy-gradient methods, but is explored further later in the implementation section of this chapter. Here, the term for averaging by the number of tokens in the action, $\frac{1}{|a|}$, comes from common implementation practices, but is not in a formal derivation of the loss (shown in [135]).

Here we will explain the difference cases this loss function triggers given various advantages and policy ratios. At an implementation level, the inner computations for PPO involve standard policy gradient and a clipped policy gradient.

To understand how different situations emerge, we can define the policy ratio as:

$$R(\theta) = \frac{\pi_\theta(a|s)}{\pi_{\theta_{old}}(a|s)} \tag{48}$$

The policy ratio is a centerpiece of PPO and related algorithms. It emerges from computing the gradient of a policy and controls the parameter updates in a very intuitive way. For any batch of data, the policy ratio starts at 1 for the first gradient step for that batch (common practice is to take 1-4 gradient steps per batch with policy gradient algorithms). Then, the policy ratio will be above one if that gradient step increased the likelihood of certain tokens with an associated positive advantage, or less than one for the other case.

The policy ratio and advantage together can occur in a few different configurations.

The first case is when the advantage is positive and the policy ratio exceeds $1+\varepsilon$ (meaning that the new policy is more likely to take said action), which is clipped, and the objective becomes:

$$J(\theta) = \min\left(R(\theta), (1+\varepsilon)\right) A = (1+\varepsilon)A \tag{49}$$

This will increase the probability ratio, making the action even more likely, but only up until the clipping parameter epsilon. The similar conditions are when the advantage is still positive, but the likelihood ratio shifts.

For positive advantage and ratio less than $1-\varepsilon$, a we get a partially substituted equation:

$$J(\theta) = \min\left(R(\theta), (1-\varepsilon)\right) A \tag{50}$$

That reduces to

$$J(\theta) = R(\theta)A \tag{51}$$

because of the less than assumption.

Similarly, if the probability ratio is not clipping, the objective also reduces to the $\min(R(\theta), R(\theta))$, yielding a standard policy gradient with an advantage estimator.

If the advantage is negative, this looks similar. A clipped objective will occur when $R(\theta) < (1-\varepsilon)$, appearing through:

$$J(\theta) = \min\left(R(\theta)A, (1-\varepsilon)A\right), \tag{52}$$

Which, because $A < 0$ we have $R(\theta)A > (1-\varepsilon)A$ and can flip the min to the max when pulling A from the equation, is equivalent to

$$J(\theta) = \max\left(R(\theta), (1-\varepsilon)\right) A. \tag{53}$$

Then the objective becomes:

$$J(\theta) = (1-\varepsilon)A \tag{54}$$

The other cases follow as above, inverted, and are left as an exercise to the reader.

All of these are designed to make the behaviors where advantage is positive more likely and keep the gradient step within the trust region. It is crucial to remember that PPO within the trust region is the same as standard forms of policy gradient.

11.1.4 Group Relative Policy Optimization

Group Relative Policy Optimization (GRPO) is introduced in DeepSeekMath [136], and used in other DeepSeek works, e.g. DeepSeek-V3 [137] and DeepSeek-R1 [138]. GRPO can be viewed as PPO-inspired algorithm with a very similar surrogate loss, but it avoids learning a value function with another copy of the original policy language model (or another checkpoint for initialization). This brings two posited benefits:

1. Avoiding the challenge of learning a value function from a LM backbone, where research hasn't established best practices.
2. Saves memory by not needing to keep another set of model weights in memory.

GRPO does this by simplifying the value estimation and assigning the same value to every token in the episode (i.e. in the completion to a prompt, each token gets assigned the same value rather than discounted rewards in a standard value function) by estimating the advantage or baseline. The estimate is done by collecting multiple completions (a_i) and rewards (r_i), i.e. a Monte Carlo estimate, from the same initial state / prompt (s).

To state this formally, the GRPO objective is very similar to the PPO objective above. For GRPO, the loss is accumulated over a group of responses $\{a_1, a_2, ..., a_G\}$ to a given question s:

$$J(\theta) = \frac{1}{G}\sum_{i=1}^{G}\left(\min\left(\frac{\pi_\theta(a_i|s)}{\pi_{\theta_{old}}(a_i|s)}A_i, \text{clip}\left(\frac{\pi_\theta(a_i|s)}{\pi_{\theta_{old}}(a_i|s)}, 1-\varepsilon, 1+\varepsilon\right)A_i\right) - \beta D_{KL}(\pi_\theta||\pi_{ref})\right). \tag{55}$$

As above, we can expand this into a per-token loss computation:

$$J(\theta) = \frac{1}{G}\sum_{i=1}^{G}\frac{1}{|a_i|}\sum_{t=1}^{|a_i|}\left(\min\left(\frac{\pi_\theta(a_{i,t}|s_{i,t})}{\pi_{\theta_{old}}(a_{i,t}|s_{i,t})}A_{i,t}, \text{clip}\left(\frac{\pi_\theta(a_{i,t}|s_{i,t})}{\pi_{\theta_{old}}(a_{i,t}|s_{i,t})}, 1-\varepsilon, 1+\varepsilon\right)A_{i,t}\right) - \beta D_{KL}(\pi_\theta(\cdot|s_{i,t})||\pi_{ref}(\cdot|s_{i,t})\right) \tag{56}$$

Note that relative to PPO, the standard implementation of GRPO includes the KL distance in the loss. With the advantage computation for the completion index i:

$$A_i = \frac{r_i - \text{mean}(r_1, r_2, \cdots, r_G)}{\text{std}(r_1, r_2, \cdots, r_G)}. \tag{57}$$

Intuitively, the GRPO update is comparing multiple answers to a single question within a batch. The model learns to become more like the answers marked as correct and less like the others. This is a very simple way to compute the advantage, which is the measure of how much better a specific action is than the average at a given state. Relative to PPO, REINFORCE, and broadly RLHF performed with a reward model rating (relative to output

reward), GRPO is often run with a far higher number of samples per prompt. Here, the current policy generates multiple responses to a given prompt, and the group-wise GRPO advantage estimate is given valuable context.

The advantage computation for GRPO has trade-offs in its biases. The normalization by standard deviation is rewarding questions in a batch that have a low variation in answer correctness. For questions with either nearly all correct or all incorrect answers, the standard deviation will be lower and the advantage will be higher. [135] proposes removing the standard deviation term given this bias, but this comes at the cost of down-weighing questions that were all incorrect with a few correct answers, which could be seen as valuable learning signal.

eq. 57 is the implementation of GRPO when working with outcome supervision (either a standard reward model or a single verifiable reward) and a different implementation is needed with process supervision. In this case, GRPO computes the advantage as the sum of the normalized rewards for the following reasoning steps.

Finally, GRPO's advantage estimation can also be applied without the PPO clipping to more vanilla versions of policy gradient (e.g. REINFORCE), but it is not the canonical form. As an example of how these algorithms are intertwined, we can show that the advantage estimation in a variant of GRPO, Dr. GRPO (GRPO Done Right) [135], is equivalent to the RLOO estimation up to a constant scaling factor (which normally does not matter due to implementation details to normalize the advantage). Dr. GRPO removes the standard deviation normalization term from eq. 57 – note that this also scales the advantage *up*, which is equivalent to increasing the GRPO learning rate on samples with a variance in answer scores. This addresses a bias towards questions with low reward variance – i.e. almost all the answers are right or wrong – but comes at a potential cost where problems where just one sample gets the answer right are important to learn from. The Dr. GRPO advantage for completion i within a group of size G is defined as:

$$\tilde{A}_i = r_i - \text{mean}(r_1, r_2, \cdots, r_G) = r_i - \frac{1}{G} \sum j = 1^G r_j \tag{58}$$

Here, in the same notation we can recall the RLOO advantage estimation as:

$$A_i^{\text{RLOO}} = r_i - \frac{1}{G-1} \sum_{j=1, i \neq j}^{G} r_j \tag{59}$$

Thus, if we multiply the Dr. GRPO advantage definition by $\frac{G}{G-1}$ we can see an scaled equivalence:

$$
\begin{aligned}
\frac{G}{G-1}\tilde{A}_i &= \frac{G}{G-1}\left(r_i - \frac{1}{G}\sum_{j=1}^{G} r_j\right) \\
&= \frac{G}{G-1}r_i - \frac{1}{G-1}\sum_{j=1}^{G} r_j \\
&= \frac{G}{G-1}r_i - \frac{1}{G-1}\sum_{j=1, j\neq i}^{G} r_j - \frac{1}{G-1}r_i \\
&= r_i\left(\frac{G}{G-1} - \frac{1}{G-1}\right) - \frac{1}{G-1}\sum_{j=1, j\neq i}^{G} r_j \\
&= r_i - \frac{1}{G-1}\sum_{j=1, j\neq i}^{G} r_j \\
&= A_i^{\text{RLOO}}
\end{aligned}
\tag{60}
$$

11.2 Implementation

Compared to the original Deep RL literature where many of these algorithms were developed, implementing RL for optimizing language models or other large AI models requires many small implementation details. In this section, we highlight some key factors that differentiate the implementations of popular algorithms.

There are many other small details that go into this training. For example, when doing RLHF with language models a crucial step is generating text that will then be rated by the reward model. Under normal circumstances, the model should generate a end-of-sequence (EOS) token indicating it finished generating, but a common practice is to put a hard cap on generation length to efficiently utilize infrastructure. A failure mode of RLHF is that the model is regularly truncated in its answers, driving the ratings from the reward model out of distribution and to unpredictable scores. The solution to this is to *only* run a reward model ranking on the `eos_token`, and to otherwise assign a penalty to the model for generating too long.

The popular open-source tools for RLHF have a large variance in implementation details across the algorithms (see table 10 in [139]). Some decisions not covered here include:

- **Value network initialization**: The internal learned value network used by PPO and other similar algorithms can be started from a different model of the same architecture or randomly selected weights. This can have a large impact on performance.
- **Reward normalization, reward whitening, and/or advantage whitening**: Where normalization bounds all the values from the RM (or environment) to be between 0 and 1, which can help with learning stability, whitening the rewards or the advantage estimates to uniform covariates can provide an even stronger boost to stability.
- **Different KL estimators**: With complex language models, precisely computing the KL divergence between models can be complex, so multiple approximations are used to substitute for an exact calculation [116].

- **KL controllers**: Original implementations of PPO and related algorithms had dynamic controllers that targeted specific KLs and changed the penalty based on recent measurements. Most modern RLHF implementations use static KL penalties, but this can also vary.

For more details on implementation details for RLHF, see [140]. For further information on the algorithms, see [141].

11.2.1 Policy Gradient Basics

A simple implementation of policy gradient, using advantages to estimate the gradient to prepare for advanced algorithms such as PPO and GRPO follows:

```
pg_loss = -advantages * ratio
```

Ratio here is the logratio of the new policy model probabilities relative to the reference model.

In order to understand this equation it is good to understand different cases that can fall within a batch of updates. Remember that we want the loss to *decrease* as the model gets better at the task.

Case 1: Positive advantage, so the action was better than the expected value of the state. We want to reinforce this. In this case, the model will make this more likely with the negative sign. To do so it'll increase the logratio. A positive logratio, or sum of log probabilities of the tokens, means that the model is more likely to generate those tokens.

Case 2: Negative advantage, so the action was worse than the expected value of the state. This follows very similarly. Here, the loss will be positive if the new model was more likely, so the model will try to make it so the policy parameters make this completion less likely.

Case 3: Zero advantage, so no update is needed. The loss is zero, don't change the policy model.

11.2.2 Loss Aggregation

The question when implementing any policy gradient algorithm with language models is: How do you sum over the KL distance and loss to design different types of value-attribution.

Most of the discussions in this section assume a token-level action, where the RL problem is formatted as a Markov Decision Process (MDP) rather than a bandit problem. In a bandit problem, all the tokens in an action will be given the same loss, which has been the default implementation for some algorithms such as Advantage-Leftover Lunch RL (A-LoL) [142]. The formulation between MDP and bandit is actually an implementation detail over how the loss is aggregated per-sample. A bandit approach takes a mean that assigns the same loss to every token, which also aligns with DPO and other direct alignment algorithms' standard implementations.

Consider an example where we have the following variables, with a batch size B and sequence length L.

```
advantages # [B, 1]
per_token_probability_ratios # [B, L]
```

We can approximate the loss as above with a batch multiplication of `pg_loss = -advantages * ratio`. Multiplying these together is broadcasting the advantage per each completion in the batch (as in the outcome reward setting, rather than a per-token value model setting) to be the same. They are then multiplied by the per token probability logratios.

In cases where a value network is used, it is easy to see that the different losses can behave very differently. When outcome rewards are used, the advantages are set to be the same per token, so the difference in per-token probability is crucial to policy gradient learning dynamics.

In the below implementations of GRPO and PPO, the loss is summed over the tokens in the completion:

```
sequence_loss = ((per_token_loss * completion_mask).sum(dim=1) / \
             completion_mask.sum(dim=1)).mean()
```

The operation above is very similar to a `masked_mean` operation. An alternative is to average over each token individually.

```
token_loss = ((per_token_loss * completion_mask).sum() / \
            completion_mask.sum())
```

Intuitively, it could seem that averaging over the sequence is best, as we are trying to reward the model for *outcomes* and the specific tokens are not as important. This can introduce subtle forms of bias. Consider two sequences of different lengths, assigned two different advantages `a_1` and `a_2`.

```
seq_1_advs = [a_1, a_1, a_1, a_1, a_1] # 5 tokens
seq_2_advs = [a_2, a_2, a_2, a_2, a_2, a_2, a_2, a_2, a_2, a_2] # 10
     tokens
```

Now, consider if the last token in each sequence is important to the advantage being positive, so it gets increased over the multiple gradient steps per batch. When you convert these to per-token losses, you could get something approximate to:

```
seq_1_losses = [1, 1, 1, 1, 10] # 5 tokens
seq_2_losses = [1, 1, 1, 1, 1, 1, 1, 1, 1, 10] # 10 tokens
```

If we average these over the sequences, we will get the following numbers:

```
seq_1_loss = 2.8
seq_2_loss = 1.9
```

If we average these together weighting sequences equally, we get a loss of 2.35. If, instead we apply the loss equally to each token, the loss would be computed by summing all the per token losses and normalizing by length, which in this case would be 2.27. If the sequences had bigger differences, the two loss values can have substantially different values.

For a more complete example on how loss aggregation changes the loss per-token and per-example, see the below script that computes the loss over a toy batch with two samples, one long and one short. The example uses three loss aggregation techniques: `masked_mean` corresponds to a per-sample length normalization, the loss proposed in DAPO [143] with token level normalization per batch, `masked_mean_token_level`, and `masked_sum_result` with a fixed length normalization from the max length from Dr. GRPO [135].

```
from typing import Optional
import torch

def masked_mean(values: torch.Tensor, mask: torch.Tensor, axis: Optional[
    int] = None) -> torch.Tensor:
    """Compute␣mean␣of␣tensor␣with␣a␣masked␣values."""
    if axis is not None:
        return (values * mask).sum(axis=axis) / mask.sum(axis=axis)
    else:
        return (values * mask).sum() / mask.sum()

def masked_sum(
        values: torch.Tensor,
        mask: torch.Tensor,
        axis: Optional[bool] = None,
        constant_normalizer: float = 1.0,
    ) -> torch.Tensor:
    """Compute␣sum␣of␣tensor␣with␣a␣masked␣values.␣Use␣a␣constant␣to␣
        normalize."""
    if axis is not None:
        return (values * mask).sum(axis=axis) / constant_normalizer
    else:
        return (values * mask).sum() / constant_normalizer

ratio = torch.tensor([
    [1., 1, 1, 1, 1, 1, 1,],
    [1, 1, 1, 1, 1, 1, 1,],
], requires_grad=True)


advs = torch.tensor([
    [2, 2, 2, 2, 2, 2, 2,],
    [2, 2, 2, 2, 2, 2, 2,],
])

masks = torch.tensor([
    # generation 1: 4 tokens
    [1, 1, 1, 1, 0, 0, 0,],
    # generation 2: 7 tokens
    [1, 1, 1, 1, 1, 1, 1,],
])

max_gen_len = 7

masked_mean_result = masked_mean(ratio * advs, masks, axis=1)
masked_mean_token_level = masked_mean(ratio, masks, axis=None)
masked_sum_result = masked_sum(ratio * advs, masks, axis=1,
    constant_normalizer=max_gen_len)

print("masked_mean", masked_mean_result)
print("masked_sum", masked_sum_result)
print("masked_mean_token_level", masked_mean_token_level)
```

```
# masked_mean tensor([2., 2.], grad_fn=<DivBackward0>)
# masked_sum tensor([1.1429, 2.0000], grad_fn=<DivBackward0>)
# masked_mean_token_level tensor(1., grad_fn=<DivBackward0>)

masked_mean_result.mean().backward()
print("ratio.grad", ratio.grad)
ratio.grad.zero_()
# ratio.grad tensor([[0.2500, 0.2500, 0.2500, 0.2500, 0.0000, 0.0000,
    0.0000],
# [0.1429, 0.1429, 0.1429, 0.1429, 0.1429, 0.1429, 0.1429]])

masked_sum_result.mean().backward()
print("ratio.grad", ratio.grad)
# ratio.grad tensor([[0.1429, 0.1429, 0.1429, 0.1429, 0.0000, 0.0000,
    0.0000],
# [0.1429, 0.1429, 0.1429, 0.1429, 0.1429, 0.1429, 0.1429]])

masked_mean_token_level.mean().backward()
print("ratio.grad", ratio.grad)
# ratio.grad tensor([[0.2338, 0.2338, 0.2338, 0.2338, 0.0000, 0.0000,
    0.0000],
# [0.2338, 0.2338, 0.2338, 0.2338, 0.2338, 0.2338, 0.2338]])
```

Here it can be seen for the default GRPO implementation, `masked_mean`, the short length has a bigger per-token gradient than the longer one, and the two implementations of Dr. GRPO and DAPO balance it out. Note that these results can vary substantially if gradient accumulation is used, where the gradients are summed across multiple mini batches before taking a backward step. In this case, the balance between shorter and longer sequences can flip.

Another way to aggregate loss is discussed in [135] that has its origins in pre language model RL research, where every per-token loss is normalized by the max sequence length set in the experiment. This would change how the losses compare across batches per tokens in the above example.

In practice, the setup that is best likely is the one that is suited to the individual, online learning setup. Often in RLHF methods the method with the best numerical stability and or the least variance in loss could be preferred.

11.2.3 Proximal Policy Optimization

There are many, many implementations of PPO available. The core *loss* computation is shown below. Crucial to stable performance is also the *value* computation, where multiple options exist (including multiple options for the *value model* loss).

Note that the reference policy (or old logprobs) here are from the time the generations were sampled and not necessarily the reference policy. The reference policy is only used for the KL distance constraint/penalty.

```
# B: Batch Size, L: Sequence Length, G: Num of Generations
# Apply KL penalty to rewards
rewards = rewards - self.beta * per_token_kl  # Shape: (B*G, L)
```

```
# Get value predictions
values = value_net(completions)  # Shape: (B*G, L)

# Compute simple advantages
advantages = rewards - values.detach()  # Shape: (B*G, L)

# Normalize advantages (optional but stable)
advantages = (advantages - advantages.mean()) / (advantages.std() + 1e-8)
advantages = advantages.unsqueeze(1)  # Shape: (B*G, 1)

# Compute probability ratio between new and old policies
ratio = torch.exp(new_per_token_logps - per_token_logps)  # Shape: (B*G,
    L)

# PPO clipping objective
eps = self.cliprange  # e.g. 0.2
pg_losses1 = -advantages * ratio  # Shape: (B*G, L)
pg_losses2 = -advantages * torch.clamp(ratio, 1.0 - eps, 1.0 + eps)  #
    Shape: (B*G, L)
pg_loss_max = torch.max(pg_losses1, pg_losses2)  # Shape: (B*G, L)

# Simple value function loss
vf_loss = 0.5 * ((rewards - values) ** 2)  # Shape: (B*G, L)

# Combine policy and value losses
per_token_loss = pg_loss_max + self.vf_coef * vf_loss  # Shape: (B*G, L)

# Apply completion mask and compute final loss
loss = ((per_token_loss * completion_mask).sum(dim=1) / completion_mask.
    sum(dim=1)).mean()
 # Scalar

# Compute metrics for logging
with torch.no_grad():
    # Compute clipping fraction
    clip_frac = ((pg_losses2 > pg_losses1).float() * completion_mask).sum
        () / completion_mask.sum()

    # Compute approximate KL
    approx_kl = 0.5 * ((new_per_token_logps - per_token_logps)**2).mean()

    # Compute value loss for logging
    value_loss = vf_loss.mean()
```

The core piece to understand with PPO is how the policy gradient loss is updated. Focus on these three lines:

```
pg_losses1 = -advantages * ratio  # Shape: (B*G, L)
pg_losses2 = -advantages * torch.clamp(ratio, 1.0 - eps, 1.0 + eps)  #
    Shape: (B*G, L)
pg_loss_max = torch.max(pg_losses1, pg_losses2)  # Shape: (B*G, L)
```

`pg_losses1` is the same as the vanilla advantage-based PR loss above, which is included

in PPO, but the loss (and gradient update) can be clipped. Though, PPO is controlling the update size to not be too big. Because losses can be negative, we must create a more conservative version of the vanilla policy gradient update rule.

We know that if we *do not* constrain the loss, the policy gradient algorithm will update the weights exactly to the new probability distribution. Hence, by clamping the logratio's, PPO is limiting the distance that the update can move the policy parameters.

Finally, the max of two is taken as mentioned above, in order to take the more conservative loss update.

For PPO, all of this happens *while* learning a value function, which opens more complexity, but this is the core logic for the parameter update.

11.2.3.1 PPO/GRPO simplification with 1 gradient step per sample (no clipping) PPO (and GRPO) implementations can be handled much more elegantly if the hyperparameter "number of gradient steps per sample" is equal to 1. Many normal values for this are from 2-4 or higher. In the main PPO or GRPO equations, see eq. 46, the "reference" policy is the previous parameters – those used to generate the completions or actions. Thus, if only one gradient step is taken, $\pi_\theta = \pi_{\theta_{old}}$, and the update rule reduces to the following (the notation $[]_\nabla$ indicates a stop gradient):

$$J(\theta) = \frac{1}{G}\sum_{i=1}^{G}\left(\frac{\pi_\theta(a_i|s)}{[\pi_\theta(a_i|s)]_\nabla}A_i - \beta D_{KL}(\pi_\theta||\pi_{ref})\right). \tag{61}$$

This leads to PPO or GRPO implementations where the second policy gradient and clipping logic can be omitted, making the optimizer far closer to standard policy gradient.

11.2.4 Group Relative Policy Optimization

The DeepSeekMath paper details some implementation details of GRPO that differ from PPO [136], especially if comparing to a standard application of PPO from Deep RL rather than language models. For example, the KL penalty within the RLHF optimization (recall the KL penalty is also used when training reasoning models on verifiable rewards without a reward model) is applied directly in the loss update rather to the reward function. Where the standard KL penalty application for RLHF is applied as $r = r_\theta + \beta D_{KL}$, the GRPO implementation is along the lines of:

$$L = L_{\text{policy gradient}} - \beta * D_{KL}$$

Though, there are multiple ways to implement this. Traditionally, the KL distance is computed with respect to each token in the completion to a prompt s. For reasoning training, multiple completions are sampled from one prompt, and there are multiple prompts in one batch, so the KL distance will have a shape of [B, L, N], where B is the batch size, L is the sequence length, and N is the number of completions per prompt.

Putting it together, using the first loss accumulation, the pseudocode can be written as below.

```
# B: Batch Size, L: Sequence Length, G: Number of Generations
# Compute grouped-wise rewards # Shape: (B,)
mean_grouped_rewards = rewards.view(-1, self.num_generations).mean(dim=1)
std_grouped_rewards = rewards.view(-1, self.num_generations).std(dim=1)


# Normalize the rewards to compute the advantages
mean_grouped_rewards = mean_grouped_rewards.repeat_interleave(self.
    num_generations, dim=0)
std_grouped_rewards = std_grouped_rewards.repeat_interleave(self.
    num_generations, dim=0)
# Shape: (B*G,)

# Compute advantages
advantages = (rewards - mean_grouped_rewards) / (std_grouped_rewards + 1e
    -4)
advantages = advantages.unsqueeze(1)
# Shape: (B*G, 1)

# Compute probability ratio between new and old policies
ratio = torch.exp(new_per_token_logps - per_token_logps)  # Shape: (B*G,
    L)

# PPO clipping objective
eps = self.cliprange  # e.g. 0.2
pg_losses1 = -advantages * ratio  # Shape: (B*G, L)
pg_losses2 = -advantages * torch.clamp(ratio, 1.0 - eps, 1.0 + eps)  #
    Shape: (B*G, L)
pg_loss_max = torch.max(pg_losses1, pg_losses2)  # Shape: (B*G, L)

# important to GRPO -- PPO applies this in reward traditionally
# Combine with KL penalty
per_token_loss = pg_loss_max + self.beta * per_token_kl  # Shape: (B*G, L
    )

# Apply completion mask and compute final loss
loss = ((per_token_loss * completion_mask).sum(dim=1) / completion_mask.
    sum(dim=1)).mean()
 # Scalar

# Compute core metric for logging (KL, reward, etc. also logged)
with torch.no_grad():
    # Compute clipping fraction
    clip_frac = ((pg_losses2 > pg_losses1).float() * completion_mask).sum
        () / completion_mask.sum()

    # Compute approximate KL
    approx_kl = 0.5 * ((new_per_token_logps - per_token_logps)**2).mean()
```

For more details on how to interpret this code, see the PPO section above.

11.2.4.1 RLOO vs. GRPO The advantage updates for RLOO follow very closely to GRPO, highlighting the conceptual similarity of the algorithm when taken separately from the PPO style clipping and KL penalty details. Specially, for RLOO, the advantage is computed relative to a baseline that is extremely similar to that of GRPO – the completion reward relative to the others for that same question. Concisely, the RLOO advantage estimate follows as (expanded from TRL's implementation):

```
# rloo_k --> number of completions per prompt
# rlhf_reward --> Initially a flat tensor of total rewards for all
    completions. Length B = N x k
rlhf_reward = rlhf_reward.reshape(rloo_k, -1) #
# Now, Shape: (k, N), each column j contains the k rewards for prompt j.

baseline = (rlhf_reward.sum(0) - rlhf_reward) / (rloo_k - 1)
# baseline --> Leave-one-out baseline rewards. Shape: (k, N)
#  baseline[i, j] is the avg reward of samples i' != i for prompt j.

advantages = rlhf_reward - baseline
# advantages --> Same Shape: (k, N)

advantages = advantages.flatten() # Same shape as original tensor
```

The rest of the implementation details for RLOO follow the other trade-offs of implementing policy-gradient.

11.3 Auxiliary Topics

In order to master the application of policy-gradient algorithms, there are countless other considerations. Here we consider some, but not all of these discussions.

11.3.1 Generalized Advantage Estimation (GAE)

Generalized Advantage Estimation (GAE) is an alternate method to compute the advantage for policy gradient algorithms [129] that better balances the bias-variance tradeoff. Traditional single-step advantage estimates often suffer from high variance, while using complete trajectories can introduce too much bias. GAE works by combining two ideas – multi-step prediction and weighted running average (or just one of these).

Advantage estimates can take many forms, but we can define a k step advantage estimator (similar to the TD residual at the beginning of the chapter) as follows:

$$\hat{A}_t^{(n)} = \begin{cases} r_t + \gamma V(s_{t+1}) - V(s_t), & n = 1 \\ r_t + \gamma r_{t+1} + \gamma^2 V(s_{t+2}) - V(s_t), & n = 2 \\ \vdots & \\ r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots - V(s_t), & n = \infty \end{cases} \tag{62}$$

Here a shorter k will have lower variance but higher bias as we are attributing more learning power to each trajectory – it can overfit. GAE attempts to generalize this formulation into a weighted multi-step average instead of a specific k. To start, we must define the temporal difference (TD) residual of predicted value.

$$\delta_t^V = r_t + \gamma V(s_{t+1}) - V(s_t) \tag{63}$$

To utilize this, we introduce another variable λ as the GAE mixing parameter. This folds into an exponential decay of future advantages we wish to estimate:

$$\begin{aligned}
&\hat{A}_t^{GAE(\gamma,\lambda)} = (1-\lambda)(\hat{A}_t^{(1)} + \lambda \hat{A}_t^{(2)} + \lambda^2 \hat{A}_t^{(3)} + \cdots) \\
&= (1-\lambda)(\delta_t^V + \lambda(\delta_t^V + \gamma\delta_{t+1}^V) + \lambda^2(\delta_t^V + \gamma\delta_{t+1}^V + \gamma^2\delta_{t+2}^V) + \cdots) \\
&= (1-\lambda)(\delta_t^V(1 + \lambda + \lambda^2 + \cdots) + \gamma\delta_{t+1}^V(\lambda + \lambda^2 + \cdots) + \cdots) \\
&= (1-\lambda)(\delta_t^V \tfrac{1}{1-\lambda} + \gamma\delta_{t+1}^V \tfrac{\lambda}{1-\lambda} + \cdots) \\
&= \textstyle\sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l}^V
\end{aligned} \tag{64}$$

Intuitively, this can be used to average of multi-step estimates of Advantage in an elegant fashion.

For further reading, see [144].

11.3.2 Double Regularization

Many popular policy gradient algorithms from Deep Reinforcement Learning originated due to the need to control the learning process of the agent. In RLHF, as discussed extensively in Chapter 8 on Regularization and in Chapter 4 on Problem Formulation, there is a built in regularization term via the distance penalty relative to the original policy one is finetuning. In this view, a large part of the difference between algorithms like PPO (which have internal step-size regularization) and REINFORCE (which is simpler, and PPO under certain hyperparameters reduces to) is far less meaningful for finetuning language models than training agents from scratch.

In PPO, the objective that handles capping the step-size of the update is known as the surrogate objective. To monitor how much the PPO regularization is impacting updates in RLHF, one can look at the clip fraction variable in many popular implementations, which is the percentage of samples in the batch where the gradients are clipped by this regularizer in PPO. These gradients are *reduced* to a maximum value.

11.3.3 Further Reading

As RLHF has cemented itself at the center of modern post-training, other policy-gradient RL algorithms and RL algorithms generally have been proposed to improve the training process, but they have not had a central role in governing best practices. Examples for further reading include:

- **Pairwise Proximal Policy Optimization (P3O)** [145] uses pairwise data directly in a PPO-style policy update without learning an intermediate reward model.
- Off-policy policy-gradient algorithms could enable further asynchronous training, such as **Contrastive Policy Gradient (CoPG)** [146] (a generalization of the direct alignment algorithm IPO and vanilla policy gradient), which was used by Cohere for their Command A model [147].
- Other implementations of REINFORCE algorithms have been designed for language models, such as **ReMax** [148], which implements a baseline normalization designed specifically to accommodate the sources of uncertainty from reward model inference.

- Some foundation models, such as Apple Intelligence Foundation Models [149] or Kimi k1.5 reasoning model [150], have used variants of **Mirror Descent Policy Optimization (MDPO)** [151]. Research is still developing further on the fundamentals here [152], but Mirror Descent is an optimization method rather than directly a policy gradient algorithm. What is important here is that it is substituted in very similarly to existing RL infrastructure.
- **Decoupled Clip and Dynamic sAmpling Policy Optimization (DAPO)** proposes 4 modifications to GRPO to better suit reasoning language models, where long traces are needed and new, underutilized tokens need to be increased in probability [143]. The changes are: 1, have two different clip hyperparameters, ϵ_{low} and ϵ_{high}, so clipping on the positive side of the logratio can take bigger steps for better exploration; 2, dynamic sampling, which removes all samples with reward = 0 or reward = 1 for all samples in the batch (no learning signal); 3, use the per token loss as discussed above in Implementation: GRPO; and 4, a soft penalty on samples that are too long to avoid trying to learn from truncated answers.
- **Value-based Augmented Proximal Policy Optimization (VAPO)** [153] combines optimizations from DAPO (including clip-higher, token level policy-gradient, and different length normalization) with insights from Value-Calibrated PPO [154] to pretrain the value function and length-adaptive GAE to show the promise of value base methods relative to GRPO.