Reinforcement Learning from Human Feedback (RLHF): An Accessible Explanation

Seth J. Chandler, with help from AI

September 12, 2025

Introduction

Large language models (LLMs) such as ChatGPT began as systems trained on huge amounts of text. Their first job is simple: predict the next word in a sequence. From this humble objective, surprising fluency emerges. But fluency is not the same as helpfulness, safety, or honesty. That is where reinforcement learning from human feedback (RLHF) comes in.

This handout explains RLHF in a way that does not assume deep mathematical background. We will repeat core ideas often, use minimal notation, and emphasize intuition. Our goal is to understand why RLHF matters, how it works at a high level, and what makes it different from the supervised learning you may have already seen.

Roadmap

We'll start by understanding how base models work through pretraining (Section 1), then see why fluency alone is insufficient (Section 2), and finally explore how RLHF solves these problems through reward models and policy gradients (Sections 3-6). Throughout, we'll use the running example of teaching a model to answer "How do I bake bread?" helpfully.

1 Pretraining: Predict the Next Word

How the base model works

A large language model begins by learning to predict the next word from context.

- **Input:** a sequence of words (the context).
- **Output:** a set of raw scores (called *logits*) for every word in the vocabulary.
- The logits are passed through the *softmax* function, which converts them into probabilities.
- The model then chooses one of the words, usually by sampling according to those probabilities.

Softmax refresher

If the logits for two words are z_1 and z_2, their probabilities are:

$$p_1 = \frac{e^{z_1}}{e^{z_1} + e^{z_2}}, \qquad p_2 = \frac{e^{z_2}}{e^{z_1} + e^{z_2}}.$$

Softmax ensures that every probability is between 0 and 1 and that all probabilities sum to 1.

Supervised training with cross-entropy

During pretraining, the model knows the "true" next word from its dataset. Suppose the true next word is "red." We represent this as a *one-hot vector*—all zeros except a 1 at the position for "red."

The loss function is *cross-entropy*: it punishes the model if the probability assigned to the true word is too low.

The parameters of the network, denoted θ, are updated by backpropagation so that next time, the model makes "red" more likely in that context.

This process, repeated billions of times, produces a fluent model.

Running Example

Prompt: "How do I bake bread?"
After pretraining alone, the model might produce:

- A helpful recipe (good!)
- A story about someone's grandmother baking bread (fluent but unhelpful)
- Random facts about wheat cultivation (off-topic)
- "Bread is typically baked in an oven at..." then trail off (incomplete)

The model is fluent but doesn't know which type of response humans actually want.

2 The Problem: Fluency is Not Enough

A pretrained model can generate human-like text, but it does not know how to be *helpful*, *safe*, or *truthful*.

- It may eagerly provide dangerous instructions.
- It may invent confident but wrong answers.
- It may ramble when asked to summarize.

Why? Because the pretraining objective was only: "sound like text from the internet." It was never: "be useful to a human asking a question."

3 Reinforcement Learning from Human Feedback (RLHF)

RLHF introduces a new element: a *reward model*. This model is trained not to predict the next word, but to score entire answers according to human preference.[1]

How to build a reward model

1. Give the base model a prompt.
2. Generate two or more answers (A, B, ...).
3. Ask humans: which answer is better?
4. Train a smaller model so that it assigns a higher score to the preferred answer.

Example: Training the Reward Model

Prompt: "How do I bake bread?"
Response A: [Clear recipe with ingredients and steps] → Human rates: Better
Response B: [Story about bread in ancient Egypt] → Human rates: Worse
The reward model learns: Recipe-style responses score higher for how-to questions.

Over time, this reward model learns to generalize: given a new prompt and answer, it can estimate how much humans would like it.

Why we need a separate reward model

Why not have humans score every output? Because the language model generates millions of responses during training. The reward model acts as a learned approximation of human judgment, making training feasible. Think of it as training a "taste tester" that can evaluate outputs at scale, so we don't need a human to rate every single generation.

4 Reinforcement Learning Updates

Now we come to the heart of RLHF: using the reward model's scores to update the large model.

[1] An earlier method—and one that is still sometimes useful—is called supervised fine tuning (SFT). The idea is to collect good examples of answers and train the model to imitate them. SFT helps, but it has two serious problems:

1. **Coverage problem:** Humans cannot provide good answers for every possible input.
2. **One-answer problem:** Many prompts have multiple acceptable answers. SFT forces the model to imitate only one. For "How do I bake bread?", there are many valid recipes, but SFT would teach only one specific approach.

The challenge: Credit assignment

In supervised training, every token has a known "true" next word. In RLHF, we do not know the true next token. We only know whether the *whole answer* was good or bad.

Analogy: Imagine a student writes a 10-page essay and gets an A. Which sentences earned the grade? Which word choices were brilliant, and which were just okay? In supervised learning, we know exactly which word should come next (like having an answer key). In RLHF, we only know the whole essay was good. The model must figure out which word choices contributed to success. This is the *credit assignment problem*.

The solution: Policy gradients

Think of the model as a policy: it assigns probabilities to actions (tokens). The key update rule is:

$$\text{Loss}(\theta) = -R \cdot \sum_{t=1}^{T} \log p_\theta(a_t \mid s_t),$$

where:

- R is the scalar reward for the whole sequence,
- a_t is the token actually chosen at step t,
- s_t is the context at step t,
- $p_\theta(a_t \mid s_t)$ is the softmax probability the model assigned to a_t.

Alternative notation for clarity

Instead of writing the loss in the compact reinforcement learning style, which I find confusing, we can make the functional dependency more explicit:

$$L(\theta) = -R \cdot \sum_{t=1}^{T} \log \Big(\text{Softmax}_\theta(s_t)[a_t] \Big).$$

Here:

- s_t = the context at step t (the prompt plus all tokens so far),
- $\text{Softmax}_\theta(s_t)$ = the probability distribution over the vocabulary that the model produces at that step,
- $[a_t]$ = indexing into that distribution to select the probability assigned to the token the model actually chose.

So the expression says very directly:

> "Given the context s_t, look at the probability the model assigned (via softmax) to the token it actually generated, a_t. Take the log of that probability, sum across the sequence, and multiply by the reward."

[2]

Why the negative sign in both the classical and reformulated formulae? We want to *minimize* loss but *maximize* reward. When the reward is positive, the negative sign converts "maximize reward" into "minimize negative reward," which fits our optimization framework.

In words:

> Multiply the reward by the sum of the log-probabilities of the chosen tokens. Take the negative of that. That is the loss to minimize with backpropagation.

Key distinction: Unlike pretraining where we use cross-entropy loss against known correct tokens, RLHF uses policy gradient loss based on rewards for entire sequences.

Intuition

- If reward is positive: increase the probabilities of the tokens that were chosen.
- If reward is negative: decrease them.

Notice the difference from supervised learning:

- In supervised learning, the target is a one-hot vector ("the answer is 'red' ").
- In RLHF, the "target" is the reward applied to the actual sequence chosen ("that whole response was good").

From probability changes to model updates

So far, we have described how RLHF changes the *desired* probabilities: if the reward is positive, we want to increase $p(a_t|s_t)$ for the tokens that were chosen. But here is the crucial step: these probability changes are just *targets*. The model's neural network—with its millions or billions of parameters θ—must now be updated to actually produce these new probabilities.

This is where backpropagation re-enters the picture. The loss function $\text{Loss}(\theta) = -R \cdot \sum_{t=1}^{T} \log p_\theta(a_t \mid s_t)$ tells us how to adjust the desired probabilities, but we need backpropagation to translate this into updates for all the weights and biases throughout the neural

[2]For students who want to see the analogy to cross-entropy loss, we can also write:

$$L(\theta) = -R \cdot \sum_{t=1}^{T} \sum_{w \in V} \delta_{a_t}(w) \, \log p_\theta(w \mid s_t),$$

where V is the vocabulary and $\delta_{a_t}(w)$ is 1 if $w = a_t$ and 0 otherwise. This is the same mathematical structure as cross-entropy, except that in supervised learning the one-hot vector picks out the *true* token, while in reinforcement learning it picks out the *sampled* token.

network. The gradient $\nabla_\theta \text{Loss}(\theta)$ flows backward through the network, adjusting every parameter slightly so that next time the model sees a similar context, it will be more (or less) likely to generate the same tokens, depending on whether the reward was positive or negative.

Think of it this way: the reward tells us "make 'cat' more likely in this context," but that instruction must propagate through all the layers of the network—the attention mechanisms, the feed-forward layers, the embeddings—so the entire model learns to recognize patterns that should lead to producing 'cat.' Without this backpropagation step, we would just have wished-for probabilities with no way to actually achieve them.

5 A Tiny Example

Suppose the vocabulary has only two tokens: A and B.

- Current probabilities: $p(A) = 0.5$, $p(B) = 0.5$.
- The model samples A.
- The reward model assigns reward $R = +2$.

Update step:

$$\Delta z_A = \alpha \cdot R \cdot (1 - p(A)) = \alpha,$$
$$\Delta z_B = \alpha \cdot R \cdot (-p(B)) = -\alpha,$$

where α is the learning rate.

Interpretation:

- The logit for A goes up.
- The logit for B goes down.

Next time, $p(A)$ will be a bit larger than 0.5, and $p(B)$ a bit smaller.
If the reward had been negative, the adjustments would go in the opposite direction.

6 Key Phrase to Remember

> In RLHF, you do not reward the top choice. You reward what the model actually did.

This is the crucial idea that allows exploration and improvement.

Why exploration matters: If the model always chose the highest-probability token, it would never discover better alternatives. By rewarding what was actually sampled—even if it wasn't the top choice—the model can explore different approaches and learn from both successes and failures. Sometimes a lower-probability choice turns out to be brilliant, and RLHF can learn from that!

7 Why PPO (Briefly)

Proximal Policy Optimization (PPO) is a refinement that keeps RLHF stable.[3]

8 What RLHF Achieves

After pretraining, a model is fluent but not aligned.

After SFT, it can imitate a few good examples.

After RLHF, it can generalize: producing outputs that are consistently more helpful, safe, and aligned with human preferences.

This is what turned GPT-style models from autocomplete machines into useful assistants. This technique is why ChatGPT can refuse harmful requests, admit uncertainty, and stay focused on being helpful—behaviors that emerged from human preference data, not explicit programming.

9 Common Misconceptions

- **"The reward model generates text"** → No, it only scores text generated by the main model.
- **"RLHF replaces pretraining"** → No, it refines a pretrained model.
- **"The model is explicitly programmed to be safe"** → No, it learns safety from patterns in human preferences.
- **"RLHF guarantees perfect outputs"** → No, it makes outputs more aligned with human preferences on average.

Summary

1. **Pretraining:** predict the next word (cross-entropy with one-hot targets).
2. **Supervised fine-tuning:** imitate human-written answers (still cross-entropy).
3. **Reward model:** trained from human comparisons to score answers.
4. **RLHF:** update the big model by nudging probabilities of chosen tokens up or down based on the sequence reward.

And again, the golden phrase:

> Reinforcement learning rewards what the model actually did and what we ended up liking, not just what would have been the top choice.

[3] PPO prevents the model from changing too drastically in one update. Think of it as keeping the model on a leash—it can explore and improve, but can't wander too far from what already works. Without PPO, the model might drift too far from its pretrained state or collapse into trivial answers (like always saying "I don't know" to avoid being wrong).

Further Reading

For students interested in diving deeper:

- **Accessible:** "Illustrating Reinforcement Learning from Human Feedback" (Hugging Face Blog)
- **Technical:** "Training language models to follow instructions with human feedback" (Ouyang et al., 2022—the InstructGPT paper)
- **PPO Details:** "Proximal Policy Optimization Algorithms" (Schulman et al., 2017)