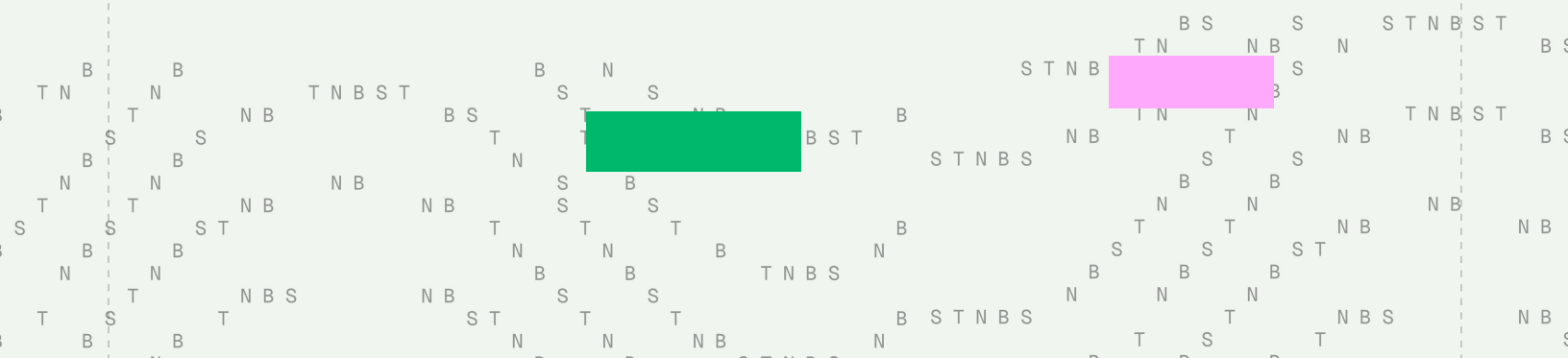
baseten

Why reinforcement learning matters

Breaking down RL vs. SFT in practice

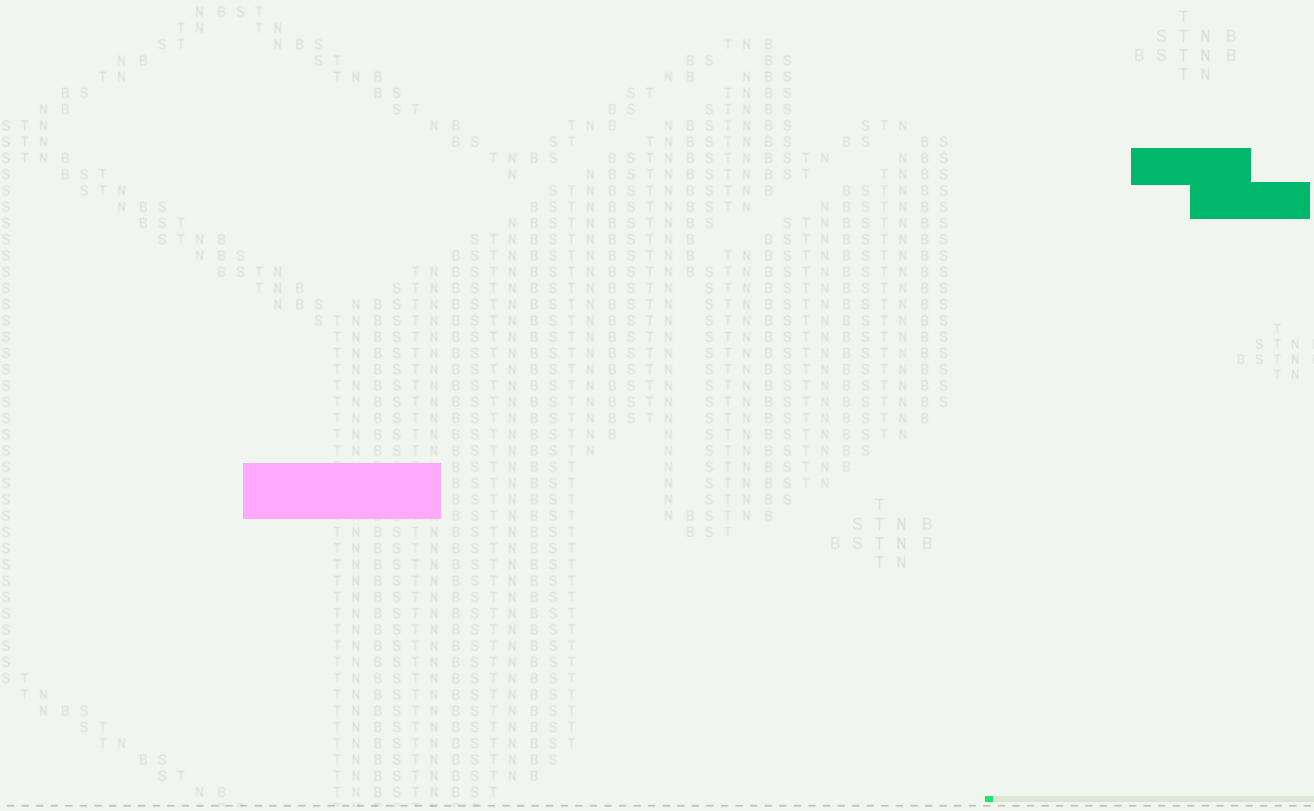


TLDR:

Reinforcement Learning (RL) lets you optimize for outcomes (does the code work? is the user satisfied?) instead of just pattern matching. This guide explains RL's terminology in simple terms, how their concepts map to familiar supervised fine-tuning (SFT) ideas, compares RLHF and GRPO, and helps you decide when to use SFT vs RL.

Most ML engineers are familiar with SFT, but RL can feel overwhelming especially with the recent explosion of techniques like RLHF (Reinforcement Learning from Human Feedback) and GRPO (Group Relative Policy Optimization). Understanding reinforcement learning for LLMs is critical because it now plays a key part in aligning models toward serving production AI applications with specific requirements.

In this guide, we will first review the foundational reinforcement learning terminology, show how SFT concepts map on to RL, and what the RL training loop looks like. We'll briefly explain the intuitions behind RLHF and GRPO, two techniques/algorithms that are most popular in RL today. Finally, we explain when it might be appropriate to use SFT vs. RL (or both!) depending on how you would like your application to behave.



Why should you care about RL?

Before we dive into the weeds, let's talk about why you should understand the distinctions between SFT vs RL.

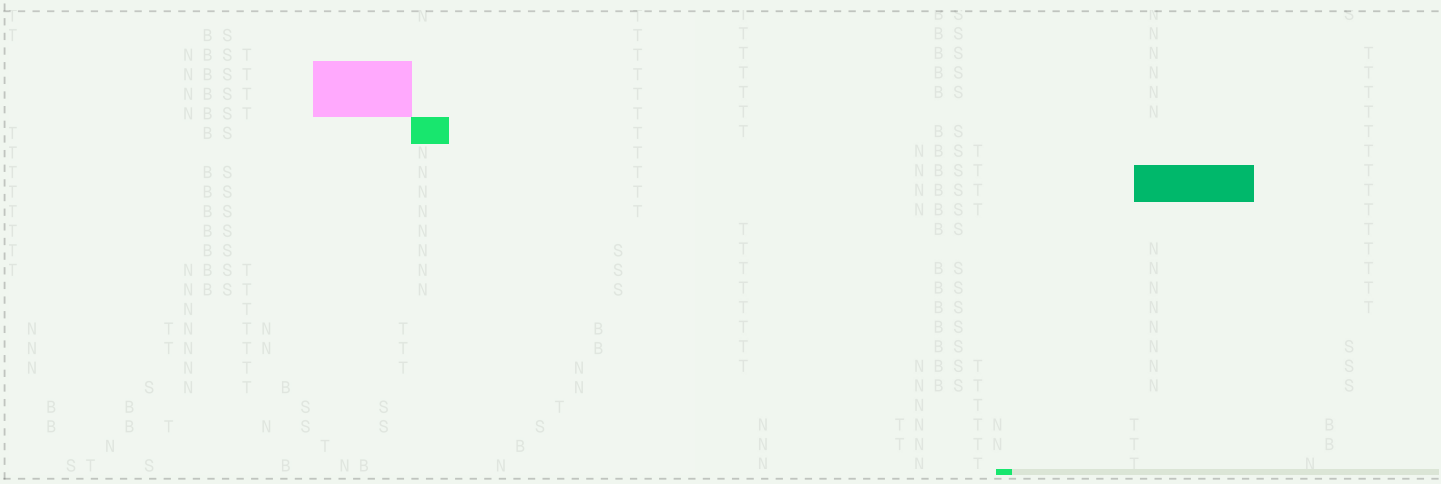
RL is how modern LLMs work. Major models like ChatGPT, Claude, and DeepSeek use RL (specifically RLHF and GRPO) after SFT. SFT gets you a model that can follow instructions, but RL is what makes it helpful, safe, and aligned with human preferences.

SFT has a ceiling. For tasks where "correct" is fuzzy or involves tradeoffs, like balancing helpfulness with safety, SFT struggles. When responses can be helpful in different ways or require nuanced judgment, you can't just show examples. Reinforcement learning lets the model learn through feedback on outcomes.

You can optimize for what you actually care about. SFT optimizes for "predict the next token correctly." RL lets you optimize for "does this code correct?" or "did the agent complete the task?" or "is this response both accurate and engaging?" You can optimize for outcomes rather than just pattern matching to maximize prediction accuracy.

The core mental model shift: SFT vs. RL

SFT is straightforward: you have data, you have labels, you minimize loss. RL is a similar loop, but it's a nested loop with a bunch of new concepts that interact in non-obvious ways.



The reinforcement learning vocabulary

Environment

Think of the environment as the world your agent lives in. It defines the rules of engagement—what actions are possible, what outcomes they lead to, and how rewards get distributed. Every environment starts with an initial state. For a chess board, it is all the pieces in their starting position. For an LLM doing code generation, that's the prompt. Each action your policy takes changes that state, and the environment responds with a new state and a reward. The environment also determines when an episode ends, such as checkmate in chess, or generating an end-of-sequence token for an LLM.

Policy: Your Agent's Decision-Making System

The policy, most commonly denoted as π, is a probability distribution over actions given the current state, essentially acting as your agent's decision-making function.

Here's where it gets interesting for ML practitioners: **your model parameterizes the policy.** The model (with its parameters θ) defines the policy π_θ(a|s)—the probability of taking action `a` given a state `s`. When you're training an LLM with RL, you're updating those parameters to make the policy choose better actions that lead to higher rewards.

Action: What Your Agent Actually Does

Actions are what the policy produces and the environment processes in response. In Chess, it's the different sets of moves that specific pieces could make.

For LLMs, an action is typically a single token. Not the full response—each token generation is its own action. The full response is a sequence of many actions, which we call a trajectory or rollout. Some RL setups do treat the full sequence as one action, but the token-by-token approach is more common and gives you finer-grained control.

Each action changes the environment's state. This is where RL gets its power: your model's outputs directly influence what happens next, creating a feedback loop that SFT doesn't have.

CONTINUED >

The reinforcement learning vocabulary

Reward: The Feedback Signal

Reward is how the environment tells your agent "good job" or "try again." Positive reward = good. Negative reward = bad. Rewards can be immediate (you get points for each coin in Mario) or delayed (you only get rewarded when the full code compiles). This creates the temporal credit assignment problem: if your LLM generates 50 tokens and only then gets feedback, which tokens deserve credit? This is one of RL's core challenges.

The goal is to maximize the return: the cumulative reward over time, often with a discount factor γ that makes near-term rewards more valuable than distant ones. For LLMs, this means the model learns to make early choices that set up better outcomes later.

Value Function (The Critic): The Confusing Part

Here's where things get weird if you're coming from SFT land. Many RL algorithms introduce a value function (also called a critic) that estimates expected future reward.
There are two main flavors:

- **V(s):** The state-value function estimates the expected future reward from state s
- **Q(s,a):** The action-value function estimates the expected future reward from taking action a in states

Good news: for algorithms like GRPO, you don't need this.
Bad news: if you're reading RL papers, you'll see this everywhere, and it can be genuinely confusing at first.

The value function helps stabilize training and guide the policy toward better decisions, especially when rewards are sparse. The critic typically needs to be trained alongside your policy, adding another layer of complexity. The value function essentially answers: 'If I'm in this state, how much total reward can I expect to get from here onward?' This helps the agent distinguish between actions that are locally beneficial versus those that are globally optimal, supporting better long-term planning. We will not be diving deep into this here, but recommend David Silver's RL course if you're interested in exploring more.

Mapping SFT Concepts to RL

Let's talk about how familiar SFT concepts translate into RL terms. This mental mapping helped me the most.

Dataset → Collection of initial states

In SFT, your dataset is everything. In RL, the dataset provides initial states that start your trajectories.

Think of it this way: your dataset of prompts doesn't become the environment—instead, each prompt initializes the environment's state. The environment is the system that processes actions and returns rewards (like a code executor with messages or a human preference model with scores).

This distinction matters because in RL, the model's outputs influence what happens next. The prompt "write a Python function to sort a list" is the initial state. The environment then watches as the model generates tokens (actions), potentially compiles the code, and returns reward signals based on whether it worked. This way we could capture more nuanced, dynamic interactions.

Model → Policy parameters

Your model parameterizes the policy. The model's weights define the policy π_θ: the probability distribution over actions given states.

The model takes in a state (the current context/prompt) and produces a probability distribution over possible next tokens. During training, you update θ to make the policy select actions that lead to higher cumulative reward.

This loop, state to model to action to new state, continues until you hit a terminal state (Mario dies, the conversation ends, the code finishes running).

CONTINUED >

Mapping SFT Concepts to RL

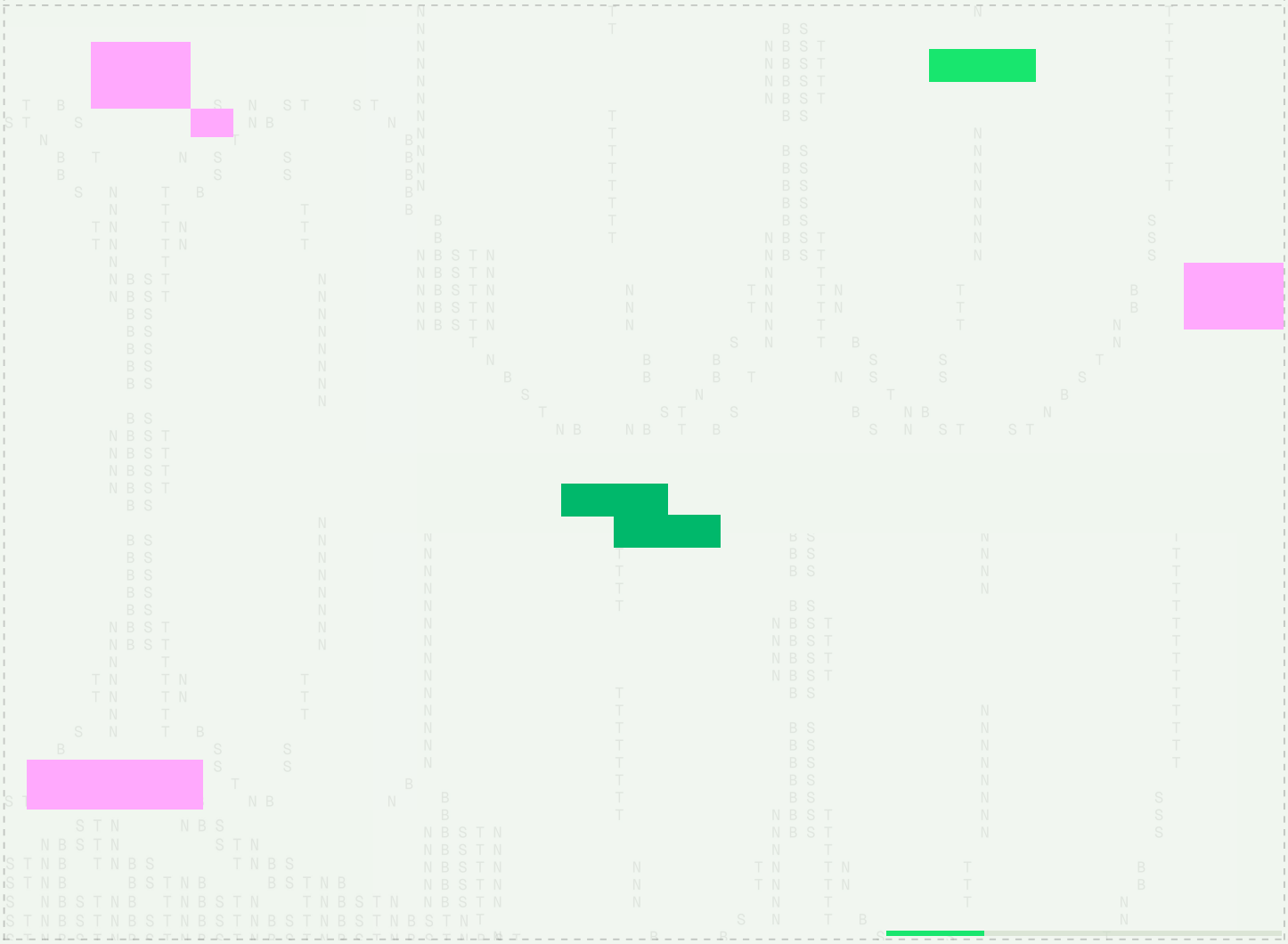
Loss → Expected cumulative reward (...sort of)

This is where the mapping is not as clean.

In SFT, you minimize a fixed loss function like cross-entropy. The loss is defined, you compute gradients, backpropagate, update the weights, and you're done.

In RL, you maximize **expected cumulative reward**.

So while reward and loss are related conceptually—reward defines the objective—the actual loss formulation depends on whether you're using Proximal Policy Optimization (PPO), Advantage Actor-Critic (A2C), GRPO, or whatever other alphabet soup algorithm you've chosen.



The training loop in RL: How it all fits together

Here's where theory meets practice. Let's walk through what actually happens during one iteration of the training loop.

Step 1: Sample From Your Dataset

You grab an example from your dataset or sample a prompt from your dataset in the case of LLMs. This provides the initial state for your environment.

Step 2: Generate N Rollouts

You typically generate N samples (often called rollouts, trajectories, or episodes—these terms are roughly interchangeable. Although a rollout could be a partial or complete episode).
Each sample is processed **independently** through a loop between the policy and environment, orchestrated by some controller or simulator.

Step 3: The State-Action-Reward Loop (How one rollout is generated)

Starting from the initial state:

1. Policy produces an action: The model looks at the current state and outputs a probability distribution over actions. You sample an action from this distribution (e.g., generate the next token).
2. Environment processes the action: The environment takes that action and produces:
   - A new state (the updated context with the new token appended)
   - Possibly a reward. In most LLM RL setups, rewards come at the end of the complete sequence (like when a reward model scores the full response), though some environments provide intermediate rewards.
3. Repeat: This cycle continues, building up a sequence of (state, action, reward) tuples until you hit a terminal state.

The full sequence you collect is your trajectory or rollout. For an LLM, this might be dozens or hundreds of tokens—each one its own action.

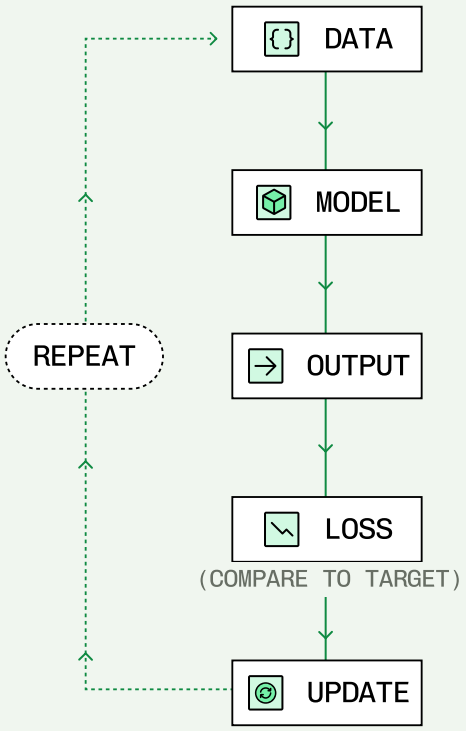
Step 4: Compute losses and update

Once you have your trajectories, you compute your specific RL losses based on the algorithm you're using. These losses push the policy toward actions that led to high cumulative rewards. You backpropagate, update your model parameters via (multiple) gradient descent, and start over from step 1.

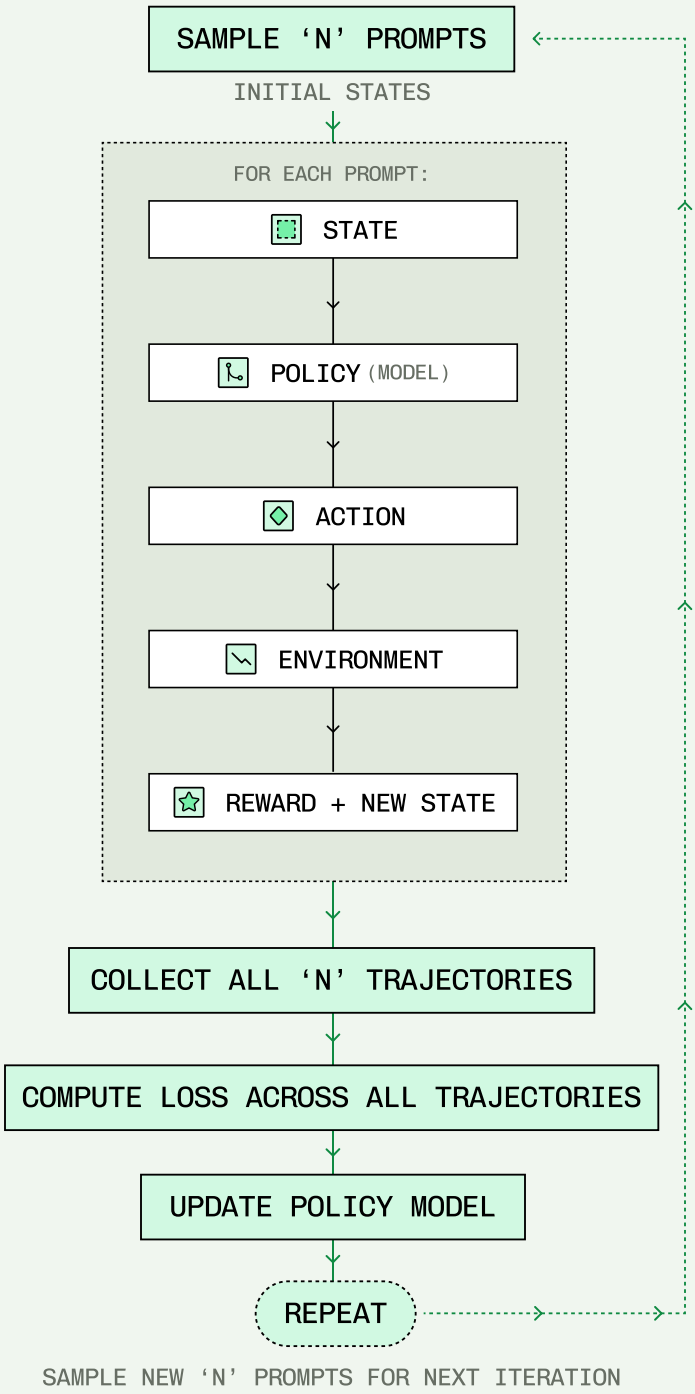
CONTINUED >

The training loop in RL: How it all fits together

SFT (Sequential)



RL (Feedback loop)



Intuitions behind two RL algorithms (RLHF with PPO vs. GRPO)

Now let's dive into two of the most popular and cornerstone RL algorithms, and understand when you should use them.

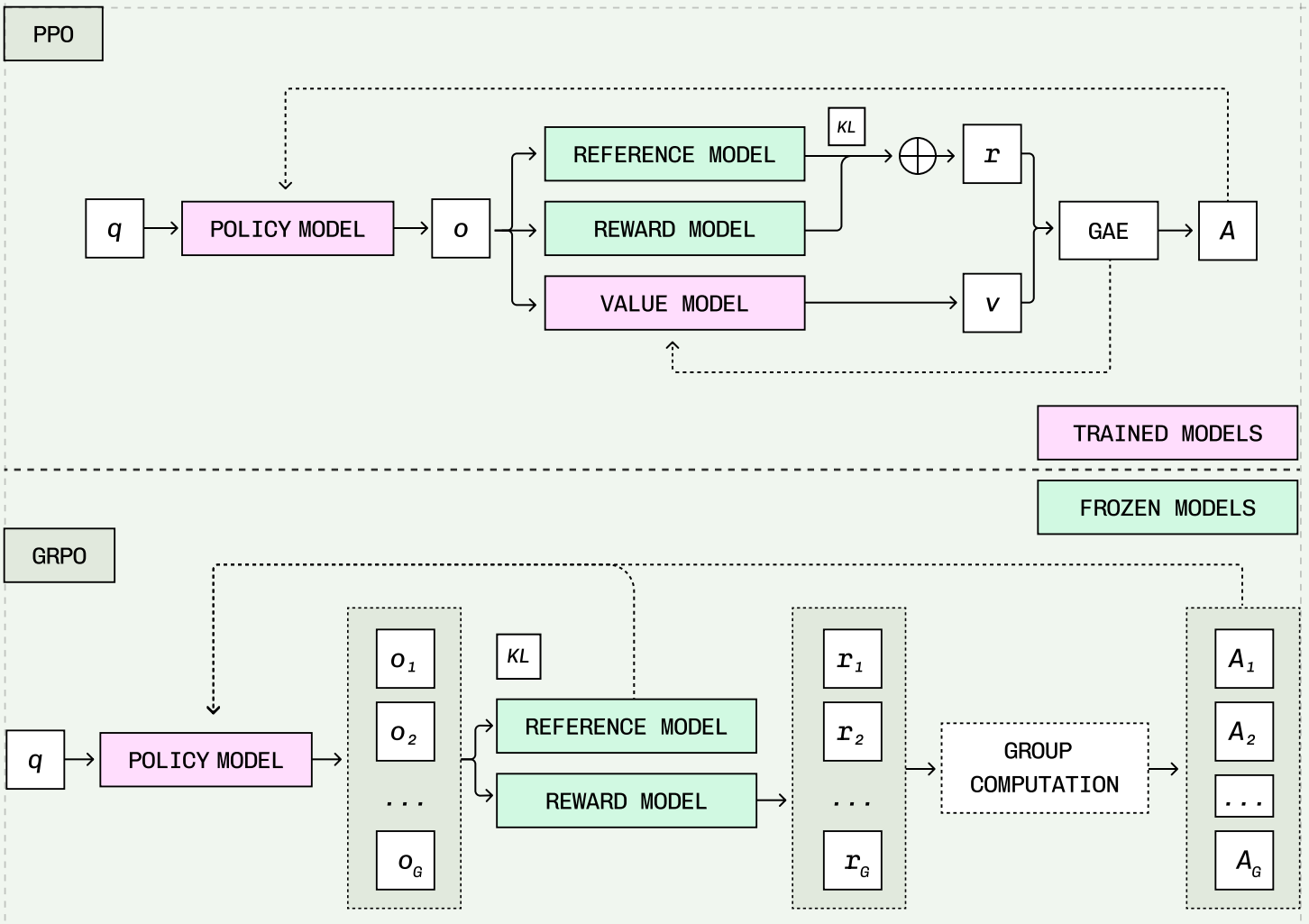
RLHF with PPO is the established approach behind models like ChatGPT and Claude. It works in three stages: first, supervised fine-tuning; second, training a separate reward model on human preference comparisons; and third, using PPO with that reward model to optimize the policy. PPO maintains a separate critic (value function) to estimate expected returns and stabilize training. This pipeline is powerful at capturing nuanced preferences, but requires maintaining multiple models simultaneously (policy, value function, reward model, reference policy), careful hyperparameter tuning, and significant compute resources.

GRPO simplifies this significantly and is used in state-of-the-art models like DeepSeek-R1. The key innovation: instead of training a separate value function, GRPO uses the average reward across multiple sampled responses as a baseline. For each prompt, generate multiple responses from the policy model, score them using your reward model, then update the policy to increase the probability of above-average responses relative to below-average ones. This eliminates an entire model from the training loop, making GRPO more stable, cheaper, and easier to implement.

When to choose each: Use **RLHF with PPO** when you have complex, multidimensional preferences and the compute budget for the full pipeline. If you'd like to save on setup cost and train time complexity, you can instead implement **Direct Preference Optimization DPO**, which cuts much of the complexity out of **RLHF with PPO** while taking advantage of the pairwise preference data. Choose **GRPO** when you want simplicity, have reliable evaluation signals (unit tests for code, task completion for agents), or need faster iteration. For most practitioners, starting with GRPO or DPO makes sense, reaching for full RLHF+PPO only if needed.

CONTINUED >

Intuitions behind two RL algorithms (RLHF with PPO vs. GRPO)



When to use RL vs. SFT

Stepping back to a higher level, when should one stick to traditional SFT or even consider using RL for training?

RL and SFT inherently teach the model different things. SFT teaches distributions, while RL teaches strategies. The most effective approach is often hybrid: use SFT to bootstrap basic competence and establish a reasonable baseline, then apply RL to optimize for the specific outcomes you care about.
Reinforcement learning is particularly powerful for tasks with verifiable outcomes, sparse reward signals, or when you want the model to discover strategies beyond what humans can demonstrate concretely.

The difference between SFT vs RL fundamentally comes down to whether you can effectively demonstrate the desired behavior through examples. An example that is appropriate for SFT is high-quality demonstrations (input and output pairs) ; the task has a well-defined definition of what correctness looks like. Examples include parsing documents or OCR (Optical Character Recognition). A version of SFT is detailed here by our in-house research team called iterative SFT, a way of learning dense reward signals.

SFT is data-efficient and stable, making it the natural starting point for most applications. To improve the quality of your AI application output, you should always think about gathering in-distribution, domain specific data, and defining output that you would deem high quality. Supervised fine-tuning with thousand data points could make a drastic difference in pattern matching to similar examples in the future.

However, SFT becomes inadequate when the notion of "correctness" is multidimensional or context-dependent. This is where RL shines: when you can define a reward signal but cannot easily provide demonstrations, when you need to optimize for measurable outcomes (code that is correct, agents that complete tasks, responses that satisfy human preferences). RL is also ideal when there are multiple valid solutions with different quality levels that require exploring the solution space, such as training an agent to use the set of custom tools you provide in your application effectively.

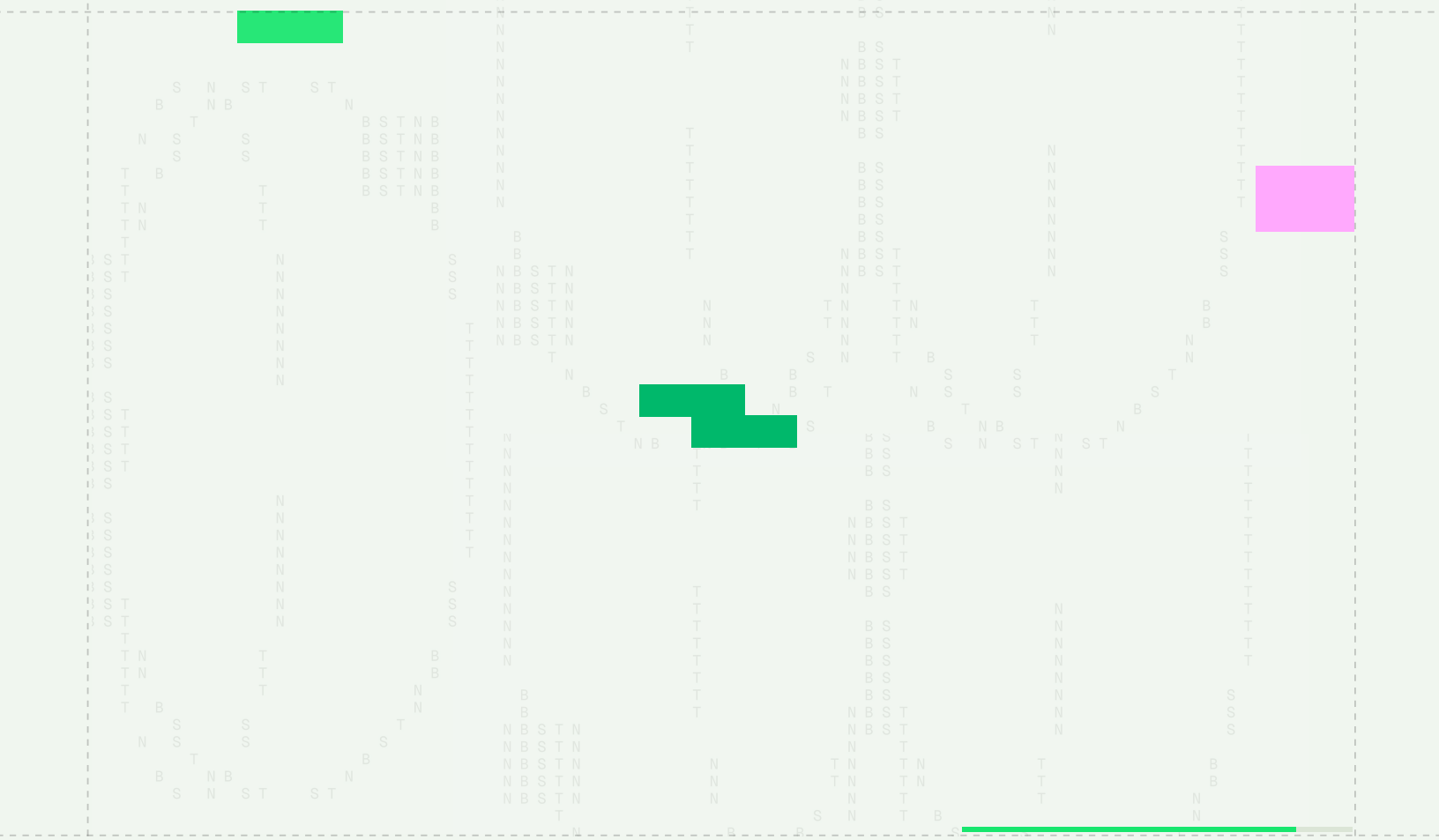
CONTINUED >

When to use RL vs. SFT

Overall, RL is particularly powerful for tasks with verifiable outcomes, sparse reward signals, or when you want the model to discover strategies beyond what humans can demonstrate.

To summarize, using both SFT and RL is the standard paradigm in modern LLM training. SFT teaches the model to follow instructions and generate coherent text, while RL (via RLHF or GRPO) aligns the model with human preferences depending on the nature of the consumers and optimizes for harder-to-demonstrate qualities like helpfulness, harmlessness, and honesty.

The key insight is that SFT optimizes for behavioral cloning (minimize divergence from demonstrations thus can almost be seen as memorizing examples), while RL optimizes for returns (maximize expected cumulative reward)—and depending on your application, one objective may be far more appropriate than the other. And sometimes, the combination will take it to an experience that brings customers delight.



Takeaway

RL is harder than SFT. There's no getting around that. But once you internalize the core concepts (environment, policy, action, reward, and value), everything starts to click.

The key insight: reinforcement learning turns static prediction into dynamic interaction. Your model doesn't just output tokens; it takes actions that change the world, receives feedback, and learns from the consequences. That's a fundamentally more powerful paradigm, albeit a more complicated one.

Now you've got the conceptual foundation to decide when RL makes sense for your use case. If you're ready to experiment with SFT vs RL on your own models, we built Baseten Training to make it painless: robust checkpointing, multi-node jobs, comprehensive metric tracking, and on-demand compute. No more Colab GPU drops. Get started here.