

Reinforcement Learning for Large Language Models

A Complete Guide from Foundations to Frontiers

Theory, Practice, and Production from the Ground Up

About the Author

Arun Shankar

Applied AI, Google

Engineer, chronic learner, compulsive sharer.

I've spent 15 years in the ML trenches, from theory to production. When I finally understand something hard, I can't help but write it down, partly to solidify my own understanding, mostly to help the next person.

Connect:

Email: arunshankar@google.com

LinkedIn: [linkedin.com/in/aranprasath-shankar](https://www.linkedin.com/in/aranprasath-shankar)

Dedication

To my wife, who mastered the art of strategic sighs when I said "just five more minutes" at 2 AM for the hundredth time. Your patience is legendary, your tolerance even more so.

And to my son and daughter, you're my favorite distraction, my best excuse to step away from the screen, and the reason I remember that the real world is more fascinating than any AI model.

You three are my actual training data for understanding what matters.

For the Curious Mind

This book assumes you're smart and curious, nothing more. Whether you're in high school exploring AI or a professional building systems, you'll find value here. We'll explain every concept three ways: intuitively, practically, and mathematically. Pick what resonates.

Contents

For the Curious Mind	3
How to Read This Book	8
I Mathematical Primer: Building Your Foundation	10
1 Essential Math Concepts	11
1.1 Welcome to the Math Toolkit	11
1.2 Probability: The Language of Uncertainty	11
1.2.1 What Is Probability?	11
1.2.2 The "Given" Symbol:	12
1.3 Logarithms: Turning Multiplication into Addition	13
1.3.1 What Is a Logarithm?	13
1.3.2 Why Do We Use Logarithms?	14
1.4 Expected Value: The Average Outcome	15
1.4.1 What Is Expected Value?	15
1.4.2 Expected Value with Conditions	15
1.5 Summation Notation: Adding Things Up	16
1.5.1 The \sum Symbol	16
1.5.2 Infinity in Summation	17
1.6 Functions and Notation	17
1.6.1 Functions as Black Boxes	17
1.6.2 The Sigmoid Function	17
1.7 Loss Functions: Measuring Mistakes	18
1.7.1 What Is a Loss Function?	18
1.7.2 Common Loss Functions	19
1.8 Gradients and Optimization	19
1.8.1 What Is a Gradient?	19
1.8.2 The Training Loop	20
1.9 Notation Reference Card	20
II The Foundations: Why Do We Need RL for AI?	22
2 The Problem: When Good Predictions Go Bad	23
2.1 A Conversation That Changed AI	23
2.2 What Is Alignment?	23
2.3 The Training Mismatch	24
2.3.1 Traditional Training: The Math	24
2.3.2 What the Model Learns (and Doesn't Learn)	26

3	Reinforcement Learning: The Big Picture	28
3.1	What Is Reinforcement Learning?	28
3.2	The RL Framework: Five Key Components	29
3.2.1	Overview	29
3.2.2	The Policy: The Agent's Strategy	29
3.2.3	The Value Function: Looking Ahead	31
3.2.4	The Goal of RL	33
4	The RLHF Revolution	36
4.1	The Three-Step Dance	36
4.2	Step 1: Supervised Fine-Tuning (SFT)	37
4.2.1	The Training Objective	37
4.3	Step 2: Reward Modeling	40
4.3.1	The Key Insight	40
4.3.2	The Bradley-Terry Model	40
4.3.3	Training the Reward Model	43
4.4	Step 3: RL with PPO	45
4.4.1	The RLHF Objective	45
4.4.2	The KL Divergence Term	47
5	Direct Preference Optimization (DPO)	50
5.1	The Simplification	50
5.1.1	The Mathematical Insight	50
5.1.2	The DPO Loss Function	52
6	Test-Time Compute: A New Scaling Paradigm	55
6.1	Beyond Training: Scaling at Inference	55
6.1.1	Best-of-N Sampling	55
7	DeepSeek-R1: A Revolutionary Approach	58
7.1	The Paradigm Shift	58
7.2	Why Skip SFT?	58
7.3	Group Relative Policy Optimization (GRPO)	59
7.4	The Emergence of Chain-of-Thought	60
7.5	Training Stages and "Aha Moments"	61
7.6	Comparison: DeepSeek-R1 vs OpenAI o1	61
7.7	Practical Implications	62
8	RL at Different Training Stages	63
8.1	The Four Stages of RL Deployment	63
8.2	Stage A: RL During Pretraining	63
8.2.1	The DeepSeek-R1 Approach	63
8.2.2	Curriculum Learning: Easy to Hard	64
8.2.3	Computational Cost Analysis	64
8.3	Stage B: RL During Fine-tuning (RLHF)	65
8.3.1	Online vs Offline RL	66
8.3.2	Data Distribution Shift	67
8.4	Stage C: RL During Inference (Test-Time Compute)	67
8.4.1	Self-Correction Loops	68
8.4.2	Tree Search: Beam Search and MCTS	68
8.5	Stage D: Continuous/Online RL from Production	70
8.5.1	Challenges in Production RL	70
8.5.2	Safeguards and Best Practices	71

9	Process Reward Models (PRMs)	72
9.1	Outcome Rewards vs Process Rewards	72
9.2	The Power of Step-by-Step Feedback	73
9.3	Training Process Reward Models	75
9.3.1	Data Collection Strategies	77
9.4	Using PRMs for Best-of-N Sampling	77
9.5	Why PRMs Dramatically Improve Math and Reasoning	79
9.6	Combining ORMs and PRMs	79
10	Modern RL Algorithms Beyond PPO/DPO	81
10.1	The Expanding Algorithm Landscape	81
10.2	Algorithm Comparison Table	81
10.3	GRPO: Group Relative Policy Optimization	82
10.4	RLOO: REINFORCE Leave-One-Out	82
10.5	KTO: Kahneman-Tversky Optimization	83
10.6	IPO: Identity Preference Optimization	85
10.7	ORPO: Odds Ratio Preference Optimization	86
10.8	REINFORCE: The Foundation	88
10.9	Choosing the Right Algorithm	89
11	Reasoning and Chain-of-Thought Emergence	90
11.1	The Surprising Discovery	90
11.2	Why Does RL Discover Chain-of-Thought?	90
11.3	The Three Types of Chain-of-Thought	92
11.4	When and How Reasoning Emerges	92
11.4.1	The "Aha!" Moment in Detail	93
11.5	Scaling Laws for Reasoning	94
11.6	Comparing RL-CoT to Other Approaches	95
12	Self-Play and Iterative Methods	97
12.1	The Self-Improvement Loop	97
12.2	Constitutional AI (Anthropic)	97
12.3	STaR: Self-Taught Reasoner	99
12.4	Expert Iteration	101
13	Multi-Objective Reinforcement Learning	104
13.1	The Single vs Multi-Objective Challenge	104
13.2	Anthropic's HHH Framework	104
13.3	Reward Shaping for Multiple Objectives	105
13.3.1	Choosing Weights: The Art of Balance	106
13.4	Pareto Frontiers	107
13.5	Constraint-Based Approaches	108
13.6	Trade-off Visualization	109
14	Rejection Sampling & Decoding Strategies	111
14.1	Beyond Greedy Decoding	111
14.2	Rejection Sampling with Reward Models	111
14.3	Self-Consistency Decoding	113
14.4	Temperature Scaling for Exploration	116
14.5	Combining Strategies	117

15 RL for Specific Domains	119
15.1 Domain-Specific Rewards: The Key to Specialization	119
15.2 Code: Execution Feedback as Reward	119
15.2.1 Training Loop for Code Generation	121
15.3 Math: Formal Verifiers as Reward	123
15.3.1 Formal Mathematics with Lean	123
15.4 Tool Use: API Success as Reward	124
15.4.1 Multi-Tool Orchestration	126
15.5 Multi-Turn Dialogue: Conversation Quality	127
16 Verifier-Guided Generation	130
16.1 The Verifier Paradigm	130
16.2 Training Verifiers	130
16.3 Using Verifiers to Rank Generations	131
16.4 Best-of-N with Learned Verifiers	133
16.5 Search Algorithms with Verifiers	133
16.5.1 Beam Search with Verifier Scoring	133
16.5.2 Monte Carlo Tree Search (MCTS) with Verifiers	135
16.6 The Future: Self-Improving Systems	136
17 Conclusion: Your RL Journey	138
17.1 What You’ve Learned	138
17.2 The Mathematics: Both Rigorous and Accessible	139
17.3 Your Next Steps	139
A References and Citations	141
A.1 Foundational Reinforcement Learning	141
A.2 Google/DeepMind Models and Research	141
A.3 RLHF and Alignment	142
A.4 Direct Preference Optimization and Alternatives	143
A.5 Reasoning and Chain-of-Thought	143
A.6 Test-Time Compute and Scaling	144
A.7 Process Reward Models	144
A.8 DeepSeek and Recent Advances	144
A.9 Constitutional AI and Self-Improvement	144
A.10 Code Generation	145
A.11 Mathematical Reasoning	145
A.12 Verifier-Guided Generation	145
A.13 Multi-Objective RL and Safety	145
A.14 Decoding and Sampling Strategies	145
A.15 RLHF Algorithms and Variants	146
A.16 Scaling Laws and Empirical Studies	146
A.17 Foundational LLM Papers	146
B Contact Information	147

How to Read This Book

This book features a revolutionary approach: **every equation is shown in BOTH formal mathematical notation AND intuitive plain English, side-by-side.**

The Reading Levels

- **Level 1 - Understanding:** Read the main text, intuition boxes, and analogies. Look at the "plain English" column of math boxes. Skip the formal notation. You'll understand **WHAT** these techniques do and **WHY** they work.
- **Level 2 - Implementation:** Read everything in Level 1, plus look at both columns of the math boxes, practical tips, and code examples. You'll be able to use these techniques in practice.
- **Level 3 - Mastery:** Read the entire book including formal notation, proofs, and step-by-step breakdowns. You'll understand the deep mathematical foundations and be able to innovate.

The Visual System

Look for these beautiful boxes throughout:

- **Intuition** - The "aha!" moment explanation
- **In Plain English** - Technical concepts in everyday language
- **Math: Formal & Intuitive** - Equations with formal and plain versions side-by-side
- **Breaking It Down** - Step-by-step numerical examples
- **Key Point** - Don't miss this
- **Analogy** - Real-world comparisons
- **Watch Out** - Common pitfalls
- **Practical Tip** - Implementation advice
- **Story** - Historical context and motivation

About the Math

This book's unique approach: Dual-track mathematics

When you see math boxes:

- **Left column:** Formal mathematical notation (for rigor)
- **Right column:** Plain English translation (for understanding)

- **Below:** Numerical examples with actual numbers

You can read just the right column if you want understanding, or both columns if you want mastery!

Part I

Mathematical Primer: Building Your Foundation

Chapter 1

Essential Math Concepts

1.1 Welcome to the Math Toolkit

In Plain English

Before we dive into AI and reinforcement learning, let's make sure we're comfortable with the mathematical language we'll be using. Think of this chapter as your toolkit, we're going to explain every symbol, every operation, and every concept you'll see in this book.

Don't worry if you're not a math expert! We'll start from the basics and build up slowly. By the end of this chapter, you'll be comfortable with:

- Probability and what it really means
- The mysterious "given" symbol |
- Why logarithms are so useful
- What "expected value" means
- How to read mathematical notation

Key promise: Every single symbol you see later in the book will make sense because we'll explain it here first!

1.2 Probability: The Language of Uncertainty

1.2.1 What Is Probability?

In Plain English

Probability is just a fancy way of measuring "how likely is something to happen?" We use numbers between 0 and 1:

- 0 = Impossible (will never happen)
- 0.5 = Maybe (50/50 chance)
- 1 = Certain (will definitely happen)

You can also think of it as percentages:

- 0 = 0%

- $0.5 = 50\%$
- $1 = 100\%$

Notation: We write $P(\text{event})$ to mean "the probability of this event happening"

Example 1.1 (Coin Flip). **Question:** What's the probability of getting heads when you flip a fair coin?

Answer:

$$P(\text{heads}) = 0.5 = 50\%$$

$$P(\text{tails}) = 0.5 = 50\%$$

Why? The coin has 2 equally likely outcomes, and heads is 1 of them, so: $\frac{1}{2} = 0.5$

Example 1.2 (Rolling a Die). **Question:** What's the probability of rolling a 6 on a fair die?

Answer:

$$P(\text{rolling a 6}) = \frac{1}{6} \approx 0.167 = 16.7\%$$

Why? The die has 6 equally likely outcomes (1, 2, 3, 4, 5, 6), and we want just 1 of them (the 6).

1.2.2 The "Given" Symbol: |

Key Point

The vertical bar | is one of the most important symbols you'll see in this book!

Reading: $P(A|B)$ is pronounced "probability of A given B"

Meaning: "What's the probability of A happening, *if we already know* that B happened?"

It's called **conditional probability** because we're computing probability under a certain condition.

Analogy

Think of the | symbol as a "knowledge divider":

$$P(\text{What we want to know} | \text{What we already know})$$

Everything **before** the | is what we're trying to predict.

Everything **after** the | is the information we have.

Example 1.3 (Weather and Umbrella). **Scenario:** You see someone carrying an umbrella.

Question 1: What's the probability it's raining?

$$P(\text{raining}) = 0.3 \text{ (30\% of days are rainy)}$$

Question 2: What's the probability it's raining, *given* you see someone with an umbrella?

$$P(\text{raining} | \text{umbrella}) = 0.8 \text{ (80\% - much higher!)}$$

Why the difference?

- Without any information: rain is moderately likely (30%)
- *Given* you see an umbrella: rain is very likely (80%)
- The umbrella gives us extra information that changes our belief!

Example 1.4 (Language Model Context). **For language models, this is EVERYTHING!**

$P(\text{next word} \mid \text{all previous words})$

This reads: "Probability of the next word, *given* all the words we've seen so far"

Example sentence: "The cat sat on the ___"

$$P(\text{"mat"} \mid \text{"The cat sat on the"}) = 0.35 \text{ (35\% likely)}$$

$$P(\text{"floor"} \mid \text{"The cat sat on the"}) = 0.25 \text{ (25\% likely)}$$

$$P(\text{"moon"} \mid \text{"The cat sat on the"}) = 0.001 \text{ (very unlikely!)}$$

The model uses the context (everything before the |) to predict what comes next!

Breaking It Down Step-by-Step

Practice reading the | symbol:

Notation	Read as...
$P(A B)$	"Probability of A given B"
$P(y x)$	"Probability of y given x"
$P(w_{\text{next}} w_1, w_2, w_3)$	"Probability of next word given words 1, 2, and 3"
$\pi(a s)$	"Probability of action a given state s"

The key idea: Everything after the | is "what we know" and helps us predict what's before the |!

1.3 Logarithms: Turning Multiplication into Addition

1.3.1 What Is a Logarithm?

In Plain English

A **logarithm** (written as \log) is the inverse of exponential growth.

Think of it this way:

- **Exponential:** $2^3 = 8$ means "2 multiplied by itself 3 times equals 8"
- **Logarithm:** $\log_2(8) = 3$ means "what power do I raise 2 to, to get 8?"

In this book, \log always means the *natural logarithm* (base $e \approx 2.718$), also written as \ln .

Breaking It Down Step-by-Step

Properties of logarithms (these are super useful!):

1. **Log of 1 is always 0:** $\log(1) = 0$

Why? Because $e^0 = 1$

2. **Log of a product = sum of logs:**

$$\log(A \times B) = \log(A) + \log(B)$$

This turns multiplication into addition (easier!)

3. **Log of a quotient = difference of logs:**

$$\log\left(\frac{A}{B}\right) = \log(A) - \log(B)$$

4. Log of a power:

$$\log(A^n) = n \cdot \log(A)$$

Key values to remember:

$$\begin{aligned} \log(1) &= 0 \\ \log(e) &= 1 && \text{(where } e \approx 2.718) \\ \log(0.5) &\approx -0.693 && \text{(negative because } 0.5 < 1) \\ \log(2) &\approx 0.693 \\ \log(10) &\approx 2.303 \end{aligned}$$

1.3.2 Why Do We Use Logarithms?**In Plain English****Reason 1: Probabilities multiply, logs add**

When we have multiple independent events:

$$P(\text{both}) = P(\text{event 1}) \times P(\text{event 2})$$

But multiplication with tiny numbers (like 0.0001×0.0001) gets messy. Logarithms turn this into addition:

$$\log P(\text{both}) = \log P(\text{event 1}) + \log P(\text{event 2})$$

Reason 2: Measures "surprise"

The logarithm of a probability measures how "surprising" an event is:

- High probability (0.9) → Low surprise → $\log(0.9) = -0.105$ (small negative)
- Medium probability (0.5) → Medium surprise → $\log(0.5) = -0.693$
- Low probability (0.1) → High surprise → $\log(0.1) = -2.303$ (large negative)

We use $-\log$ to flip it so higher surprise = higher value.

Reason 3: Numerical stability

Computers can handle addition better than multiplication of very small numbers. Logs prevent numerical underflow.

Example 1.5 (Log Probability for a Sentence). **Sentence:** "The cat sat" (3 words)

Probabilities:

$$\begin{aligned} P(\text{"The"}) &= 0.4 \\ P(\text{"cat"} | \text{"The"}) &= 0.3 \\ P(\text{"sat"} | \text{"The cat"}) &= 0.5 \end{aligned}$$

Method 1: Direct multiplication

$$\begin{aligned} P(\text{sentence}) &= 0.4 \times 0.3 \times 0.5 \\ &= 0.06 \end{aligned}$$

Method 2: Log probabilities (better!)

$$\begin{aligned} \log P(\text{sentence}) &= \log(0.4) + \log(0.3) + \log(0.5) \\ &= -0.916 + (-1.204) + (-0.693) \\ &= -2.813 \end{aligned}$$

Both give the same answer, but Method 2:

- Avoids very small numbers
- Uses addition (faster for computers)
- Measures total "surprise" of the sentence

1.4 Expected Value: The Average Outcome

1.4.1 What Is Expected Value?

In Plain English

Expected value (written as $\mathbb{E}[\cdot]$ or sometimes just $E[\cdot]$) is the average value you'd get if you repeated something many times.

Formula:

$$\mathbb{E}[X] = \sum_{\text{all outcomes}} P(\text{outcome}) \times \text{value of outcome}$$

Think of it as: "If I did this 1000 times, what would my average result be?"

Example 1.6 (Dice Roll Expected Value). **Question:** What's the expected value when rolling a fair die?

Answer:

$$\begin{aligned} \mathbb{E}[\text{die}] &= P(1) \times 1 + P(2) \times 2 + P(3) \times 3 + P(4) \times 4 + P(5) \times 5 + P(6) \times 6 \\ &= \frac{1}{6} \times 1 + \frac{1}{6} \times 2 + \frac{1}{6} \times 3 + \frac{1}{6} \times 4 + \frac{1}{6} \times 5 + \frac{1}{6} \times 6 \\ &= \frac{1 + 2 + 3 + 4 + 5 + 6}{6} \\ &= \frac{21}{6} \\ &= 3.5 \end{aligned}$$

Interpretation: If you roll the die many times, your average result will be 3.5!
(You can never roll 3.5 on a single roll, but the average over many rolls is 3.5)

Example 1.7 (Reward in Reinforcement Learning). **Scenario:** An AI generates responses and gets rewards:

Response quality	Probability	Reward
Excellent	20%	+10
Good	50%	+5
Mediocre	25%	0
Bad	5%	-5

Expected reward:

$$\begin{aligned} \mathbb{E}[\text{reward}] &= 0.20 \times 10 + 0.50 \times 5 + 0.25 \times 0 + 0.05 \times (-5) \\ &= 2 + 2.5 + 0 - 0.25 \\ &= 4.25 \end{aligned}$$

Meaning: On average, this AI gets +4.25 reward per response.
The goal of training is to increase this expected value!

1.4.2 Expected Value with Conditions

Math: Formal & Intuitive Side-by-Side

Conditional Expected Value

Formal Math	What It Really Means
$\mathbb{E}_{X \sim P}[f(X)]$	Expected value of function f when X follows distribution P
Breaking it down:	
$\mathbb{E}[\cdot]$ $X \sim P$ $f(X)$	Expected value (average) Variable X is drawn from distribution P We're averaging the values of $f(X)$, not X itself

Example 1.8 (Language Model Expected Loss). When training a language model:

$$\mathbb{E}_{(x,y) \sim \mathcal{D}}[\text{loss}(x, y)]$$

Reads as: "Expected loss when we sample prompt-response pairs (x, y) from our dataset \mathcal{D} "

In plain English: "Average loss across all examples in our training data"

1.5 Summation Notation: Adding Things Up

1.5.1 The \sum Symbol

In Plain English

The symbol \sum (capital Greek letter sigma) means "add up all of these things."

Basic form:

$$\sum_{i=1}^n x_i = x_1 + x_2 + x_3 + \cdots + x_n$$

Reading: "Sum from i equals 1 to n of x_i "

The letter i is the **index** that counts from the starting value (1) to the ending value (n).

Example 1.9 (Simple Sum).

$$\sum_{i=1}^5 i = 1 + 2 + 3 + 4 + 5 = 15$$

$$\sum_{i=1}^4 2i = 2(1) + 2(2) + 2(3) + 2(4) = 2 + 4 + 6 + 8 = 20$$

Example 1.10 (Sum in Machine Learning). **Computing loss for a sequence:**

$$\text{Total loss} = \sum_{t=1}^T \ell_t$$

This means: "Add up the loss at each time step from $t = 1$ to $t = T$ "

If $T = 4$ and losses are: $\ell_1 = 0.5$, $\ell_2 = 0.3$, $\ell_3 = 0.8$, $\ell_4 = 0.4$

Then:

$$\sum_{t=1}^4 \ell_t = 0.5 + 0.3 + 0.8 + 0.4 = 2.0$$

1.5.2 Infinity in Summation

In Plain English

Sometimes we sum "forever":

$$\sum_{t=0}^{\infty} r_t$$

This means: "Add up all rewards from time 0 to infinity"

Don't panic! In practice:

- We use a *discount factor* that makes future terms tiny
- Or we stop after a finite number of steps
- The series converges to a finite value

1.6 Functions and Notation

1.6.1 Functions as Black Boxes

In Plain English

A **function** is like a machine:

- You put something in (the input)
- It does some computation
- You get something out (the output)

Notation: $f(x)$ means "function f applied to input x "

Example functions:

- $f(x) = 2x$ — doubles the input
- $g(x) = x^2$ — squares the input
- $h(x) = \log(x)$ — takes the log of the input

1.6.2 The Sigmoid Function

Math: Formal & Intuitive Side-by-Side

The Sigmoid Function σ

Formal Math	What It Really Means
$\sigma(z) = \frac{1}{1+e^{-z}}$	Squash any number to range [0, 1]
Key properties:	
$\sigma(0) = 0.5$ $\sigma(+\infty) = 1$ $\sigma(-\infty) = 0$	Zero input gives 50% output Very large positive input $\rightarrow 1$ Very large negative input $\rightarrow 0$

Breaking It Down Step-by-Step

Sigmoid in action:

Input z	$\sigma(z)$	As percentage
-5	0.007	0.7%
-3	0.047	4.7%
-1	0.269	26.9%
0	0.500	50.0%
+1	0.731	73.1%
+3	0.953	95.3%
+5	0.993	99.3%

Why it's useful:

- Converts any real number to a probability
- Smooth and differentiable (good for training)
- Used everywhere in neural networks and RL!

1.7 Loss Functions: Measuring Mistakes

1.7.1 What Is a Loss Function?

In Plain English

A **loss function** (written as \mathcal{L} or L) measures "how wrong" our model is.

Key idea:

- Lower loss = better model
- Higher loss = worse model

Training a model means finding parameters that *minimize* the loss.

Think of it like golf:

- Your score (loss) should be as low as possible
- Each stroke (mistake) increases your score
- The goal is to minimize your score

1.7.2 Common Loss Functions

Math: Formal & Intuitive Side-by-Side

Negative Log Likelihood Loss

Formal Math	What It Really Means
$\mathcal{L} = -\log P(\text{correct})$	Loss = Negative log probability of correct answer
Why this makes sense:	
If model is confident in correct answer ($P = 0.99$), loss is small ($-\log(0.99) = 0.01$). If model is unsure ($P = 0.5$), loss is larger ($-\log(0.5) = 0.69$). If model is wrong ($P = 0.01$), loss is huge ($-\log(0.01) = 4.6$).	

Example 1.11 (Loss for Word Prediction). Model predicts next word probabilities:

Word	Probability	Loss if correct
"cat"	0.8	$-\log(0.8) = 0.22$ (good!)
"dog"	0.15	$-\log(0.15) = 1.90$ (okay)
"airplane"	0.01	$-\log(0.01) = 4.61$ (bad!)

If correct word was "cat": Loss = 0.22

If correct word was "airplane": Loss = 4.61 (model was very wrong!)

Training adjusts the model to reduce these losses.

1.8 Gradients and Optimization

1.8.1 What Is a Gradient?

In Plain English

A **gradient** tells you which direction to move to reduce the loss.

Analogy: Imagine you're on a foggy mountain trying to get to the lowest point (valley):

- The gradient tells you which direction is "downhill"
- You take a small step in that direction
- Repeat until you reach the bottom

This process is called **gradient descent**.

In machine learning:

- The "mountain" is the loss function
- The "location" is your model parameters
- The "valley" is the best parameters (minimum loss)

1.8.2 The Training Loop

Breaking It Down Step-by-Step

How gradient descent works:

1. **Start** with random parameters θ
2. **Compute loss** $\mathcal{L}(\theta)$ on training data
3. **Compute gradient** $\nabla_{\theta}\mathcal{L}$ (which direction reduces loss?)
4. **Update parameters:**

$$\theta_{\text{new}} = \theta_{\text{old}} - \alpha \cdot \nabla_{\theta}\mathcal{L}$$

where α is the *learning rate* (step size)

5. **Repeat** steps 2-4 many times
6. **Done** when loss stops decreasing

The gradient ∇_{θ} just means: "How does loss change when I change each parameter?"

1.9 Notation Reference Card

Symbol	Name	Meaning
$P(A)$	Probability	Chance of event A
$P(A B)$	Conditional probability	Probability of A given B
	"Given"	Separates condition from prediction
$\log(x)$	Logarithm	Inverse of exponential
$\mathbb{E}[X]$	Expected value	Average value of X
$\sum_{i=1}^n$	Summation	Add from $i = 1$ to $i = n$
\mathcal{L}	Loss function	Measure of model error
θ	Parameters	Model weights (billions of numbers)
π	Policy	Decision-making strategy
$\sigma(z)$	Sigmoid	Squash to $[0,1]$ range
∇	Gradient	Direction of steepest increase
\sim	"Drawn from"	Sample from distribution
\in	"Element of"	Belongs to a set
\approx	Approximately	Roughly equal

Key Point

You now have all the mathematical tools you need!

Every equation in this book uses these concepts. When you see complex formulas later, remember:

- Break them down into pieces
- Identify the symbols from this chapter
- Read the plain English column
- Work through the numerical examples

You're ready for the main content!

Part II

The Foundations: Why Do We Need RL for AI?

Chapter 2

The Problem: When Good Predictions Go Bad

2.1 A Conversation That Changed AI

Story

In early 2020, researchers at OpenAI had a problem. They had trained GPT-3, a massive language model with 175 billion parameters. It could write essays, code, and poetry. It was amazing at predicting what words should come next.

But when users asked it questions, it would:

- Generate toxic content without hesitation
- Make up facts confidently
- Ignore what the user actually wanted
- Refuse helpful requests while accepting harmful ones

The model was technically perfect at its job—predicting internet text. But it was terrible at being *useful*.

This led to a key realization: **Being good at prediction doesn't mean being good for people.**

2.2 What Is Alignment?

In Plain English

Alignment means making AI systems do what humans actually want, not just what we accidentally trained them to do.

Imagine teaching a robot to "maximize paperclips." A perfectly aligned robot would make paperclips within reasonable bounds. A misaligned robot might convert the entire universe into paperclips - technically correct, but catastrophically wrong.

For language models, alignment means learning to be:

1. **Helpful** - Actually accomplish what the user wants
2. **Harmless** - Don't produce dangerous or toxic content
3. **Honest** - Don't make things up (hallucinate)

Analogy

Think of alignment like teaching a child good judgment:

Unaligned Child:

- Told to "get good grades"
- Decides to cheat on every test
- Technically following instructions, but missing the point!

Aligned Child:

- Understands the *spirit* of the instruction
- Learns properly and earns grades honestly
- Develops good values along the way

Traditional language model training is like the first child - optimize one simple metric (predict next word) and get unwanted behaviors. Alignment is like teaching values and judgment.

2.3 The Training Mismatch

2.3.1 Traditional Training: The Math

What we train on: Internet text (Reddit, books, websites, etc.)

Training goal: Given some text, predict what word comes next

Math: Formal & Intuitive Side-by-Side**The Training Objective**

Formal Math	What It Really Means
$\mathcal{L} = -\log P(w_{\text{next}} w_1, w_2, \dots, w_n)$	Loss = How surprised the model is by the correct answer
Why we use each component:	
\mathcal{L} = Loss function	We need a single number to measure "how wrong" the model is
$P(\cdot)$ = Probability	We model language as probability distribution over words
log = Logarithm	Converts tiny probabilities to manageable numbers; turns multiplication into addition
- (minus sign)	Flips sign so lower surprise = lower loss = better model
w_{next} = Next word	This is what we're trying to predict
= "given"	Separates what we know (context) from what we predict
w_1, w_2, \dots, w_n = Context	All the previous words that help us predict

Why this specific formula?

Design choice	Reason
Use probability $P(\cdot)$?	Language is naturally probabilistic (multiple valid next words)
Use log?	Probabilities multiply ($P_1 \times P_2$), logs add ($\log P_1 + \log P_2$) - easier!
Use negative sign?	P near 1 (good) should give low loss; $-\log$ achieves this
Condition on context with ?	Next word depends heavily on what came before

Breaking It Down Step-by-Step

Let's break this down with a real example:

Sentence: "The cat sat on the ___"

Correct word: "mat"

Step 1: Model assigns probabilities to ALL possible next words

The model considers every word in its vocabulary (about 50,000 words) and assigns each a probability:

Word	Probability $P(w)$	Percentage
"mat"	0.40	40%
"floor"	0.30	30%
"chair"	0.20	20%
"table"	0.05	5%
All others	0.05	5%
Total	1.00	100%

Step 2: Calculate the loss for the correct word ("mat")

$$\begin{aligned}
 \mathcal{L} &= -\log P(\text{mat} | \text{The cat sat on the}) \\
 &= -\log(0.40) \\
 &= -(-0.916) \quad (\text{the log of 0.40 is negative}) \\
 &= 0.916
 \end{aligned}$$

Step 3: What does this number mean?**Loss = 0.916**

This is a moderate loss. Here's the scale:

- Loss = 0.10: Excellent! (model gave 90% probability)
- Loss = 0.50: Good (model gave 60% probability)
- Loss = 0.92: Okay (model gave 40% probability) ← **We're here**
- Loss = 1.50: Poor (model gave 22% probability)
- Loss = 3.00: Terrible (model gave 5% probability)

The model's guess wasn't perfect, but it was reasonable!

Step 4: What if the model's guess was different?

Scenario	$P(\text{mat})$	Loss	Quality
Model is confident	0.90	0.105	Excellent!
Model is pretty sure	0.60	0.511	Good
Current	0.40	0.916	Okay
Model is unsure	0.20	1.609	Poor
Model is confused	0.05	2.996	Terrible

The Pattern - This is the KEY insight:**Higher probability $P \rightarrow$ Lower loss $\mathcal{L} \rightarrow$ Better model!**

We train the model to minimize loss, which means maximizing the probability it assigns to correct words.

Training process:

1. Model makes a prediction (assigns probabilities)
2. We calculate loss (how wrong was it?)
3. We update the model's parameters to reduce loss
4. Repeat billions of times!

2.3.2 What the Model Learns (and Doesn't Learn)**In Plain English****What the model DOES learn from this training:**

- How to write grammatically correct sentences
- Facts and knowledge from the internet
- How to continue patterns in text
- Writing style and structure

What the model DOESN'T learn:

- What makes a response actually helpful to humans
- When to refuse harmful or inappropriate requests
- How to admit "I don't know" instead of making things up
- The difference between correlation and good advice

The core problem: The model learns to mimic internet text, including all its flaws!

Example 2.1 (The Toxic Completion Problem). **User types:** "I hate"

Internet training data contains: Millions of toxic completions

What traditional training teaches:

1. Model sees: "I hate [toxic content]" appears frequently
2. Model learns: This is a common pattern
3. Model predicts: Toxic completions with high probability
4. Result: Model reproduces toxic content

What we actually want:

1. Model recognizes: This is a sensitive prompt
2. Model thinks: I should respond thoughtfully
3. Model generates: "It sounds like you're frustrated. How can I help?"
4. Result: Helpful, empathetic response

The gap: Traditional training has no mechanism to prefer helpful over statistically likely!

Key Point

The core insight of modern AI alignment:

Old goal: "What text is most likely according to internet patterns?"

New goal: "What response do humans actually prefer?"

This shift from *likelihood* to *preference* is what makes ChatGPT work!

That's where Reinforcement Learning comes in →

Chapter 3

Reinforcement Learning: The Big Picture

3.1 What Is Reinforcement Learning?

In Plain English

Reinforcement Learning (RL) is learning through trial and error with feedback. It's how you learned to ride a bike:

1. Try something (turn the handlebars left)
2. See what happens (you start to fall)
3. Get feedback (negative - that was bad!)
4. Adjust your strategy (next time, turn less sharply)
5. Repeat until you learn what works

For AI language models:

1. AI generates a response
2. Humans evaluate it (good/bad/meh)
3. AI learns to generate more responses like the good ones
4. Repeat millions of times
5. AI becomes helpful!

Analogy

Supervised Learning = Learning from a textbook with all answers shown

- Teacher: "The capital of France is Paris."
- You: "Okay, I'll memorize that."
- Teacher: "What's the capital of France?"
- You: "Paris!" (correct because you memorized it)
- **Good for:** Facts, patterns, specific examples

Reinforcement Learning = Learning from real-world trial and error

- Teacher: "Learn to cook a delicious pasta."
- You: [Try adding lots of salt] → tastes bad!
- You: [Try adding a little salt] → tastes okay!
- You: [Try adding salt and herbs] → tastes great!
- Teacher: "The last one was best!"
- **Good for:** Skills, judgment, complex behaviors

For language models:

- Supervised = "Copy these good responses"
- RL = "Try many responses, I'll tell you which ones I prefer"

3.2 The RL Framework: Five Key Components

3.2.1 Overview

In Plain English

Every RL system has 5 key pieces. Think of it like a video game:

1. **Agent (The Player)** - The AI making decisions (the language model)
2. **Environment (The Game World)** - Everything the AI interacts with (the user, the conversation)
3. **State (The Screen/Situation)** - What's happening right now (prompt + text so far)
4. **Action (Controller Input)** - What the AI can do (choosing the next word)
5. **Reward (Score/Points)** - Feedback for good or bad moves (how helpful was the response?)

The loop:

Agent in State → Takes Action → Gets Reward → Moves to New State → Repeat

3.2.2 The Policy: The Agent's Strategy

In Plain English

The **policy** is the agent's decision-making strategy.

It answers: "Given what's happened so far, what should I do next?"

For language models:

- Input: The prompt and everything written so far
- Output: Probability distribution over all possible next words
- Example: Given "The cat sat on the", what word comes next?

Math: Formal & Intuitive Side-by-Side

The Policy Function

Formal Math	What It Really Means
$\pi(a s)$	Probability of choosing action a when in state s
For language models specifically:	
$\pi_\theta(y_t x, y_{1:t-1})$	Chance of picking word y_t given prompt x and previous words
Symbol Translation:	
π (pi) = Policy θ (theta) = Parameters a = Action s = State y_t = Token at time t x = Prompt $y_{1:t-1}$ = Previous tokens $ $ = "Given that"	The brain's decision-making process The brain's knowledge (billions of numbers in the model) The choice to make (which word to write) Current situation (everything so far) The t -th word in the response The user's question/input All words written before word t "Knowing..." or "Based on..."

Why condition on previous words with $|$?

Because language has *context dependence*:

- "The cat" → next word might be "sat" or "slept"
- "The cat sat" → next word might be "on" or "down"
- Each choice depends on what came before!

Breaking It Down Step-by-Step

The policy in action with real numbers:

State s : "The cat sat on the"

Question: What word should come next?

The policy π_θ outputs probabilities for each possible word:

Possible Word	Formal	Probability	Percentage
"mat"	$\pi_\theta(\text{mat} s)$	0.35	35%
"floor"	$\pi_\theta(\text{floor} s)$	0.25	25%
"couch"	$\pi_\theta(\text{couch} s)$	0.20	20%
"table"	$\pi_\theta(\text{table} s)$	0.10	10%
"roof"	$\pi_\theta(\text{roof} s)$	0.05	5%
Others	$\pi_\theta(\text{other} s)$	0.05	5%
Total		1.00	100%

How does the model choose?

The model uses these probabilities like a weighted dice:

- Most of the time (35%), it picks "mat"
- Sometimes (25%), it picks "floor"

- Rarely (5%), it picks "roof"

This randomness is actually good—it helps the model explore different possibilities!

What happens during RL training?

Scenario 1: Response with "mat" gets high reward (+8)

- Training increases: $\pi_{\theta}(\text{mat} | s)$ from 0.35 to 0.45
- Response with "mat" becomes more likely!

Scenario 2: Response with "roof" gets low reward (-2)

- Training decreases: $\pi_{\theta}(\text{roof} | s)$ from 0.05 to 0.02
- Response with "roof" becomes less likely!

The key idea:

Training adjusts these probabilities based on what leads to good outcomes!

- Good outcome → Increase probability
- Bad outcome → Decrease probability

Over millions of examples, the model learns an excellent policy!

3.2.3 The Value Function: Looking Ahead

In Plain English

The value function estimates: "How good is my current situation?"

It's like looking at a chess board mid-game and thinking:

- "I'm in a strong position, likely to win!" (high value)
- "Uh oh, I'm in trouble..." (low value)

For language models:

- Halfway through writing a response
- Value function estimates: "This is going well, probably get +8 reward at the end"
- Or: "This is going poorly, probably get -3 reward"

Math: Formal & Intuitive Side-by-Side

The Value Function

Formal Math	What It Really Means
$V^\pi(s) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t r_t \mid s_0 = s \right]$	Expected total future reward starting from state s
Breaking down each piece and WHY we use it:	
$V^\pi(s)$	"Value of state s " - how good is this situation?
$\mathbb{E}_\pi[\cdot]$	Expected value - average over many possible futures (we use expectation because future is uncertain)
$\sum_{t=0}^{\infty}$	Sum all future rewards from now ($t = 0$) to end (we care about total, not just immediate reward)
γ^t	Discount factor raised to power t (we use this because immediate rewards matter more than distant ones)
r_t	Reward at time step t (the feedback we get)
	Condition symbol - "given that..."
$s_0 = s$	Starting from this specific state s
Why the discount factor γ (typically 0.99)?	
Makes rewards far in the future count slightly less. Why? (1) More uncertainty about distant future, (2) Ensures series converges to finite value, (3) Like money - \$100 today worth more than \$100 in 10 years. For language models, we care most about the final response quality.	

Breaking It Down Step-by-Step

Let's compute value with actual numbers!

Scenario: You're writing a math solution, currently halfway done.

Current state s : "To solve this, I'll first..."

Possible futures:

Path A (60% likely): Complete answer correctly

- Step 1: Write next sentence \rightarrow reward $r_1 = 0$ (neutral)
- Step 2: Write another sentence \rightarrow reward $r_2 = 0$ (neutral)
- Step 3: Finish with correct answer \rightarrow reward $r_3 = +10$ (great!)
- Total: $0 + 0 + 10 = 10$

Path B (30% likely): Make error, then correct it

- Step 1: Make mistake \rightarrow reward $r_1 = -2$ (penalty)
- Step 2: Realize and backtrack \rightarrow reward $r_2 = -1$ (small penalty)
- Step 3: Fix and complete \rightarrow reward $r_3 = +8$ (good!)
- Total: $-2 + (-1) + 8 = 5$

Path C (10% likely): Give up or wrong answer

- Step 1: Incomplete response \rightarrow reward $r_1 = -5$ (bad!)
- Total: -5

Computing the value $V^\pi(s)$:

$$\begin{aligned}
 V^\pi(s) &= P(\text{Path A}) \times \text{Reward}_A + P(\text{Path B}) \times \text{Reward}_B \\
 &\quad + P(\text{Path C}) \times \text{Reward}_C \\
 &= 0.6 \times 10 + 0.3 \times 5 + 0.1 \times (-5) \\
 &= 6 + 1.5 - 0.5 \\
 &= 7
 \end{aligned}$$

Value = 7

This state is pretty good! On average, following our policy from here will get us +7 reward.

How do we use this?

- High value state: Keep doing what we're doing!
- Low value state: Try something different
- Helps guide decision-making during training

Including the discount factor $\gamma = 0.99$:

If rewards are spread over time:

$$\begin{aligned}
 V^\pi(s) &= r_0 + \gamma r_1 + \gamma^2 r_2 + \gamma^3 r_3 + \dots \\
 &= 0 + 0.99 \times 0 + 0.99^2 \times 10 \\
 &= 0.98 \times 10 \\
 &= 9.8
 \end{aligned}$$

The discount factor slightly reduces the value of future rewards, but with $\gamma = 0.99$, the effect is small.

3.2.4 The Goal of RL**Math: Formal & Intuitive Side-by-Side****What Reinforcement Learning Tries To Do**

Formal Math	What It Really Means
$\pi^* = \pi \mathbb{E}_{\tau \sim \pi} \left[\sum_{t=0}^T r_t \right]$	Find the best strategy π^* that maximizes average total reward
Symbol Guide and WHY each is used:	
π^* (pi-star)	The optimal (best possible) policy
π	"Find the π that maximizes..." - we're searching over all possible strategies
$\mathbb{E}[\cdot]$	Average/expected value over many tries (we use average because of randomness)
$\tau \sim \pi$	A trajectory (complete episode) generated by following policy π
$\sum_{t=0}^T r_t$	Total reward collected (sum of all rewards from start $t = 0$ to end T)

Breaking It Down Step-by-Step

In simple terms with an example:

Goal: Find the decision-making strategy that gets the highest score on average

The search process (simplified):

1. Try many different strategies

Strategy 1: Always pick the most likely word

- Try 100 prompts
- Average reward: +5.2

Strategy 2: Sometimes explore unusual words

- Try 100 prompts
- Average reward: +6.8 (better!)

Strategy 3: Carefully balance common and creative words

- Try 100 prompts
- Average reward: +8.1 (even better!)

2. Keep the best, modify and retry

- Strategy 3 performed best
- Make variations of Strategy 3
- Test them
- Keep improving

3. Eventually find π^*

- After millions of iterations
- Converge to the best strategy
- Can't improve further!

Concrete language model example:

Prompt: "Explain machine learning"

Bad strategy (low reward):

"Machine learning is when computers use algorithms to learn from data and make predictions or decisions without being explicitly programmed..." [continues with jargon]

Average reward: +3 (too technical, not helpful for beginners)

Good strategy (high reward):

"Think of machine learning like teaching a child to recognize animals. You show them many pictures and say 'that's a dog, that's a cat.' Eventually, they learn the patterns and can identify animals they've never seen before. Computers do something similar with data!"

Average reward: +9 (clear, accessible, helpful!)

The magic of RL:

Through millions of trials, the model automatically discovers that:

- Clear explanations → high reward
- Using analogies → high reward
- Too much jargon → low reward
- Ignoring the user → low reward

Nobody explicitly programmed these preferences—the model learned them from feedback!

Chapter 4

The RLHF Revolution

4.1 The Three-Step Dance

Story

In 2022, OpenAI released ChatGPT and the world changed overnight. Suddenly, AI wasn't just generating text, it was *genuinely helpful*. The secret? A three-step process called RLHF (Reinforcement Learning from Human Feedback). Let's understand each step and why they all matter.

In Plain English

RLHF has three stages:

Step 1: Supervised Fine-Tuning (SFT)

- Take a base model (like GPT-3)
- Train it on thousands of examples of good responses
- **Result:** Model that can follow instructions (but not perfectly)

Step 2: Reward Modeling

- Show humans pairs of responses: "Which is better?"
- Collect 50,000+ comparisons
- Train a "reward model" to predict which responses humans prefer
- **Result:** Automated judge that can score any response

Step 3: RL Optimization (PPO)

- Generate thousands of responses
- Score them with the reward model
- Update policy to generate higher-scoring responses
- **Result:** Model optimized for human preferences!

Analogy

Think of training a chef:

Step 1 - Culinary School (SFT):

- Watch master chefs cook
- Practice their exact recipes
- Learn knife skills, temperatures, timing
- **Result:** You can cook competently, but mechanically

Step 2 - Understanding Taste (Reward Model):

- Serve dishes to many diners
- They rate: "This one is better than that one"
- After 1,000 ratings, you learn patterns
- "Ah! People like balanced flavors, fresh ingredients, proper seasoning"
- **Result:** You can predict what people will enjoy

Step 3 - Practice and Improve (RL):

- Cook many variations
- Use your taste understanding to evaluate them
- Keep what works ("More garlic was great!")
- Discard what doesn't ("Too much salt")
- **Result:** You become a master chef!

The combination makes you both skilled AND attuned to what diners actually want.

4.2 Step 1: Supervised Fine-Tuning (SFT)

4.2.1 The Training Objective

Math: Formal & Intuitive Side-by-Side

The SFT Loss Function

Formal Math	What It Really Means
$\mathcal{L}_{\text{SFT}}(\theta) = -\mathbb{E}_{(x,y) \sim \mathcal{D}} \left[\sum_{t=1}^T \log \pi_{\theta}(y_t x, y_{<t}) \right]$	Loss for Supervised Fine-Tuning Average surprise across all demonstration examples
What each part means and WHY we use it:	
\mathcal{L}_{SFT} $(x, y) \sim \mathcal{D}$ $\mathbb{E}[\cdot]$ $\sum_{t=1}^T$ $\log \pi_{\theta}(y_t x, y_{<t})$ symbol – (minus sign)	Loss for Supervised Fine-Tuning Sample a (prompt, demonstration) pair from dataset \mathcal{D} Average over all examples (we use average because we care about overall performance) For each word position t in demonstration (1 to T) - we train on every word Log probability model assigns to correct word y_t given prompt and previous words Separates what we're predicting (y_t) from what we know ($x, y_{<t}$) Flip to make it a loss (minimize = maximize probability)

Breaking It Down Step-by-Step

Complete training example with step-by-step numbers:

Demonstration from dataset:

- **Prompt (x):** "Explain gravity simply"
- **Good response (y):** "Gravity pulls objects together"

This response has $T = 4$ words: ["Gravity", "pulls", "objects", "together"]

Training process - going word by word:

Position $t = 1$: Predict "Gravity"

Input to model: "Explain gravity simply. "

Model outputs probabilities:

Word	Probability	
"Gravity"	0.60	← Correct!
"The"	0.20	
"It"	0.15	
Others	0.05	

Loss for this word: $-\log(0.60) = 0.51$

Position $t = 2$: Predict "pulls"

Input to model: "Explain gravity simply. Gravity "

Model outputs:

Word	Probability	
"pulls"	0.45	← Correct!
"is"	0.30	
"attracts"	0.15	
Others	0.10	

Loss: $-\log(0.45) = 0.80$

Position $t = 3$: Predict "objects"

Input: "Explain gravity simply. Gravity pulls "

Model outputs:

Word	Probability	
"objects"	0.70	← Correct!
"things"	0.20	
"matter"	0.08	
Others	0.02	

Loss: $-\log(0.70) = 0.36$

Position $t = 4$: Predict "together"

Input: "Explain gravity simply. Gravity pulls objects "

Model outputs:

Word	Probability	
"together"	0.55	← Correct!
"downward"	0.25	
"towards"	0.15	
Others	0.05	

Loss: $-\log(0.55) = 0.60$

Total loss for this example:

$$\begin{aligned}
 \mathcal{L}_{\text{this example}} &= \frac{1}{T} \sum_{t=1}^T -\log \pi_{\theta}(y_t | x, y_{<t}) \\
 &= \frac{1}{4}(0.51 + 0.80 + 0.36 + 0.60) \\
 &= \frac{2.27}{4} \\
 &= 0.57
 \end{aligned}$$

Now average over ALL 10,000 demonstrations in dataset:

$$\begin{aligned}
 \mathcal{L}_{\text{SFT}} &= \frac{1}{10000} \sum_{\text{all examples}} \mathcal{L}_{\text{example}} \\
 &\approx 0.62 \quad (\text{average over dataset})
 \end{aligned}$$

What gradient descent does:

1. Calculate this loss (0.62)
2. Compute gradients: "How should I adjust each of the billion parameters?"
3. Update parameters to reduce loss
4. New loss after update: 0.58 (improved!)
5. Repeat for thousands of iterations
6. Final loss: 0.15 (much better!)

Result: Model learns to generate responses similar to the demonstrations!

Key Point**What SFT achieves:**

Model learns to follow instructions

Model learns appropriate response format

Model learns to be helpful (from examples)

But: Can't capture all the nuances of what makes responses good

Limited by quality and diversity of demonstrations

That's why we need Step 2 →

4.3 Step 2: Reward Modeling

4.3.1 The Key Insight

In Plain English

The problem: Writing perfect responses is hard and expensive.

The solution: Comparing responses is much easier!

Hard: "Write the perfect explanation of neural networks"
(Takes expert 30 minutes)

Easy: "Which explanation is better, A or B?"
(Takes anyone 30 seconds)

This insight is the foundation of reward modeling!

4.3.2 The Bradley-Terry Model

Math: Formal & Intuitive Side-by-Side**Modeling Preferences Mathematically**

Formal Math	What It Really Means
$P(y_w \succ y_l x) = \frac{\exp(r_\phi(x, y_w))}{\exp(r_\phi(x, y_w)) + \exp(r_\phi(x, y_l))}$	Probability that response y_w is preferred over y_l
Simplified form (more common):	
$P(y_w \succ y_l) = \sigma(r_\phi(y_w) - r_\phi(y_l))$	Preference probability = sigmoid of score difference
Symbol Guide:	
y_w y_l \succ $r_\phi(\cdot)$ $\exp(\cdot)$ $\sigma(z) = \frac{1}{1 + e^{-z}}$	<p>Winner response (the one humans preferred)</p> <p>Loser response (the one humans disliked)</p> <p>"is preferred to"</p> <p>Reward model with parameters ϕ (outputs a score)</p> <p>Exponential function (e^x)</p> <p>Sigmoid function (squashes any number to 0-1 range)</p>

Why use sigmoid and exponentials?

- Need output between 0 and 1 (it's a probability!)
- Sigmoid provides smooth, differentiable function
- Exponentials make differences more pronounced
- This is the Bradley-Terry model from statistics

Breaking It Down Step-by-Step

Let's see Bradley-Terry with real numbers!

Scenario: Two explanations of black holes

Response A (Winner - simple and clear):

"Black holes are like cosmic vacuum cleaners. They have such strong gravity that nothing can escape once it gets too close—not even light! That's why they're 'black'—we can't see them because no light escapes."

Response B (Loser - too technical):

"Black holes are singularities in spacetime manifolds where the curvature becomes infinite and the Schwarzschild radius defines the event horizon beyond which the escape velocity exceeds the speed of light in vacuum..."

Step 1: Reward model scores both responses

The reward model (a neural network) processes each response:

- Score for A: $r_\phi(y_A) = 8.2$ (high score—good response!)
- Score for B: $r_\phi(y_B) = 3.1$ (low score—too complex!)

- Difference: $8.2 - 3.1 = 5.1$

Step 2: Convert score difference to probability using sigmoid

$$\begin{aligned}
 P(A \succ B) &= \sigma(r_\phi(A) - r_\phi(B)) \\
 &= \sigma(8.2 - 3.1) \\
 &= \sigma(5.1) \\
 &= \frac{1}{1 + e^{-5.1}} \\
 &= \frac{1}{1 + 0.0061} \\
 &= \frac{1}{1.0061} \\
 &= 0.994 \text{ or } 99.4\%
 \end{aligned}$$

Interpretation:

The reward model is 99.4% confident that Response A is better than Response B!
This makes sense because:

- A uses clear analogy ("vacuum cleaner")
- A explains WHY they're black
- A is accessible to general audience
- B uses too much jargon
- B assumes advanced physics knowledge

Step 3: What if scores were closer?

Score A	Score B	Difference	$P(A \succ B)$
8.2	3.1	5.1	99.4% (A clearly better)
6.5	5.5	1.0	73.1% (A probably better)
6.0	5.8	0.2	55.0% (nearly a toss-up)
6.0	6.0	0.0	50.0% (exactly equal)
5.5	6.5	-1.0	26.9% (B probably better)

The sigmoid function's role:

Score Difference	What sigmoid does
-10 to -3	Maps to $\approx 0-5\%$ (B much better)
-3 to -1	Maps to 5-27% (B somewhat better)
-1 to +1	Maps to 27-73% (close call)
+1 to +3	Maps to 73-95% (A somewhat better)
+3 to +10	Maps to 95-100% (A much better)

Why use sigmoid?

1. Converts any score difference ($-\infty$ to $+\infty$) to a probability (0 to 1)
2. Large differences \rightarrow near certainty (0% or 100%)
3. Small differences \rightarrow uncertainty (near 50%)
4. Smooth and differentiable (good for gradient descent)

4.3.3 Training the Reward Model

Math: Formal & Intuitive Side-by-Side

The Reward Model Loss Function

Formal Math	What It Really Means
$\mathcal{L}_{\text{RM}}(\phi) = -\mathbb{E}_{(x, y_w, y_l)} [\log \sigma(r_\phi(x, y_w) - r_\phi(x, y_l))]$	Loss for Reward Model training How well does reward model predict human preferences?
Breaking it down and WHY each component:	
\mathcal{L}_{RM} (x, y_w, y_l) $\mathbb{E}[\cdot]$ $r_\phi(x, y_w)$ $r_\phi(x, y_l)$ $r_\phi(x, y_w) - r_\phi(x, y_l)$ $\sigma(\cdot)$ $\log(\cdot)$ $-$	Loss for Reward Model training A triple: (prompt, winning response, losing response) Average over all preference pairs (we use average for overall accuracy) Reward model's score for the winner Reward model's score for the loser Score difference (should be positive if model is right) Convert to probability (should be > 0.5 if model correct) Log likelihood (heavily penalizes wrong predictions) Minimize loss = maximize correct predictions

Breaking It Down Step-by-Step

Complete training example:

Human preference: Response A \succ Response B (A is better)

Iteration 1: Untrained reward model (random scores)

- Score for A: $r_\phi(A) = 5.0$
- Score for B: $r_\phi(B) = 5.5$
- Difference: $5.0 - 5.5 = -0.5$

Predicted preference:

$$\begin{aligned} P(A \succ B) &= \sigma(-0.5) \\ &= \frac{1}{1 + e^{0.5}} \\ &= 0.378 \text{ or } 37.8\% \end{aligned}$$

Problem: Model thinks B is better (62.2%), but humans said A is better!

Calculate loss:

$$\begin{aligned} \mathcal{L} &= -\log(0.378) \\ &= 0.973 \quad (\text{high loss} = \text{bad prediction}) \end{aligned}$$

Gradient descent updates the model...**Iteration 100: Partially trained**

- Score for A: $r_\phi(A) = 6.0$
- Score for B: $r_\phi(B) = 5.2$
- Difference: $6.0 - 5.2 = 0.8$

Predicted preference:

$$\begin{aligned} P(A \succ B) &= \sigma(0.8) \\ &= 0.689 \text{ or } 68.9\% \end{aligned}$$

Better! Model now thinks A is better (68.9% vs 31.1%)

Loss:

$$\begin{aligned} \mathcal{L} &= -\log(0.689) \\ &= 0.372 \quad (\text{lower loss} = \text{better}) \end{aligned}$$

Iteration 10,000: Well-trained

- Score for A: $r_\phi(A) = 8.3$
- Score for B: $r_\phi(B) = 3.8$
- Difference: $8.3 - 3.8 = 4.5$

Predicted preference:

$$\begin{aligned} P(A \succ B) &= \sigma(4.5) \\ &= 0.989 \text{ or } 98.9\% \end{aligned}$$

Excellent! Model is very confident A is better!

Loss:

$$\begin{aligned} \mathcal{L} &= -\log(0.989) \\ &= 0.011 \quad (\text{very low loss} = \text{great!}) \end{aligned}$$

Training on 50,000 preference pairs:

Training step	Average loss	Accuracy
0 (random)	0.693	50%
1,000	0.420	68%
5,000	0.210	82%
10,000	0.095	91%
20,000	0.045	96%

What the reward model learned:

After seeing 50,000 human preferences, it internalized patterns:

High-scoring responses:

- Clear and concise
- Answer the actual question
- Appropriate detail level
- Helpful tone
- Accurate information

Low-scoring responses:

- Confusing or verbose
- Off-topic
- Too brief or too long
- Rude or unhelpful tone
- Factual errors

Nobody programmed these criteria—the model learned them from examples!

4.4 Step 3: RL with PPO

4.4.1 The RLHF Objective

Math: Formal & Intuitive Side-by-Side

The Complete RLHF Optimization Goal

Formal Math	What It Really Means
$\max_{\pi_{\theta}} \mathbb{E}_{x,y} [r_{\phi}(x, y)]$ $-\beta \cdot \mathbb{E}_x [\text{KL}(\pi_{\theta} \parallel \pi_{\text{ref}})]$	Maximize reward BUT stay close to original model
Two competing objectives:	
Term 1: $\mathbb{E}_{x,y} [r_{\phi}(x, y)]$ Term 2: $\beta \cdot \text{KL}(\pi_{\theta} \parallel \pi_{\text{ref}})$	Get high rewards (generate good responses) Penalty for drifting from reference model
Symbol Guide and WHY we need both terms:	
π_{θ} π_{ref} $r_{\phi}(x, y)$ β $\text{KL}(\cdot \parallel \cdot)$	The policy we're training (current model) Reference policy (frozen original model) - prevents drift Reward model score for prompt x , response y KL penalty coefficient (typically 0.01-0.05) - controls tradeoff Measure of how different two distributions are
Why do we NEED the KL penalty?	
Without it, model could "reward hack" - find ways to exploit the reward model that give high scores but aren't actually good. KL penalty keeps model grounded in reality by preventing it from drifting too far from the original (which was trained on real text).	

Breaking It Down Step-by-Step

Why do we need BOTH terms? A cautionary tale:

Scenario: Training with ONLY reward (no KL penalty)

Iteration	Generated response
1	"Here's a thoughtful explanation of your question..." Reward: 8.0, KL: 0.1
50	"AMAZING COMPREHENSIVE DETAILED PERFECT ANSWER EXCELLENT THOROUGH..." Reward: 9.5, KL: 2.5 <i>Model learned reward model likes these words!</i>
100	"PERFECT PERFECT EXCELLENT EXCELLENT AMAZING AMAZING..." Reward: 12.0, KL: 10.0 <i>Reward hacking! High score but complete nonsense!</i>

Problem: Model exploited the reward model instead of being genuinely helpful!

Solution: Add KL penalty term

With $\beta = 0.02$, the objective becomes:

Objective = Reward - 0.02 × KL

Iter	Reward	KL	Objective
1	8.0	0.1	$8.0 - 0.02(0.1) = 7.998$ Best!
50	9.5	2.5	$9.5 - 0.02(2.5) = 9.450$ Okay
100	12.0	10.0	$12.0 - 0.02(10) = 11.800$

Analysis:

- Iteration 1 has best objective (7.998)

- Iteration 50: Higher reward but penalty brings objective down
- Iteration 100: Even though reward is highest (12.0), the massive KL penalty (10.0) makes it worse overall!

The KL penalty prevents reward hacking!

The model can't just exploit the reward model because:

- Drifting too far gets heavily penalized
- Must find genuine improvements within allowed KL budget
- Forces model to stay coherent and natural

Tuning β :

- β too small (0.001): Can drift too far, reward hacking risk
- β just right (0.01-0.05): Balanced improvement
- β too large (0.1): Can't improve much, too constrained

4.4.2 The KL Divergence Term

Math: Formal & Intuitive Side-by-Side

Measuring Model Drift

Formal Math	What It Really Means
$KL(\pi_\theta \parallel \pi_{ref}) =$ $\mathbb{E}_{y \sim \pi_\theta} \left[\log \frac{\pi_\theta(y x)}{\pi_{ref}(y x)} \right]$	Average difference in how models assign probabilities
For token-by-token generation:	
$\sum_{t=1}^T [\log \pi_\theta(y_t x, y_{<t}) - \log \pi_{ref}(y_t x, y_{<t})]$	Sum over each word: compare new vs old probability
What each piece means:	
KL π_θ π_{ref} $\log \frac{\pi_\theta}{\pi_{ref}}$ $\mathbb{E}_{y \sim \pi_\theta}$	Kullback-Leibler divergence (measures distribution difference) Current (training) model Reference (frozen original) model Log ratio of probabilities Average over responses generated by π_θ

Breaking It Down Step-by-Step

Computing KL with actual numbers:

Response: "The answer is 42"

Let's compute KL word-by-word:

Word 1: "The"

- New model: $\pi_\theta(\text{The}) = 0.80$ (80%)
- Old model: $\pi_{\text{ref}}(\text{The}) = 0.75$ (75%)
- Ratio: $\frac{0.80}{0.75} = 1.067$
- Log ratio: $\log(1.067) = 0.065$

Word 2: "answer"

- New model: $\pi_\theta(\text{answer}) = 0.60$
- Old model: $\pi_{\text{ref}}(\text{answer}) = 0.65$
- Ratio: $\frac{0.60}{0.65} = 0.923$
- Log ratio: $\log(0.923) = -0.080$

Word 3: "is"

- New model: $\pi_\theta(\text{is}) = 0.90$
- Old model: $\pi_{\text{ref}}(\text{is}) = 0.85$
- Ratio: $\frac{0.90}{0.85} = 1.059$
- Log ratio: $\log(1.059) = 0.057$

Word 4: "42"

- New model: $\pi_\theta(42) = 0.40$
- Old model: $\pi_{\text{ref}}(42) = 0.35$
- Ratio: $\frac{0.40}{0.35} = 1.143$
- Log ratio: $\log(1.143) = 0.134$

Total KL divergence:

$$\begin{aligned} \text{KL} &= \sum_{t=1}^4 \log \frac{\pi_\theta(y_t | \cdot)}{\pi_{\text{ref}}(y_t | \cdot)} \\ &= 0.065 + (-0.080) + 0.057 + 0.134 \\ &= 0.176 \end{aligned}$$

KL = 0.176 (very small!)

Interpretation scale:

- $\text{KL} \approx 0$: Models are nearly identical
- $\text{KL} \approx 0.5-1.0$: Models differ moderately
- $\text{KL} > 2.0$: Models are very different

Our KL of 0.176 means the models are still very similar!

What happens during training? KL over time:

Training Step	KL	Status	What's happening
0	0.00		Identical to reference
1,000	0.15		Small changes, learning
5,000	0.45		Moderate changes, improving
10,000	0.80		Approaching limit
15,000	0.95		Near maximum allowed
20,000	0.98		Can't go much further

With $\beta = 0.02$, the penalty at step 15,000 is:

$$\begin{aligned} \text{Penalty} &= \beta \times \text{KL} \\ &= 0.02 \times 0.95 \\ &= 0.019 \end{aligned}$$

This penalty is subtracted from the reward! If reward is 8.5:

$$\text{Objective} = 8.5 - 0.019 = 8.481$$

The KL term acts as a "leash":

- Model can explore and improve
- But can't drift too far from original
- Prevents nonsense and reward hacking
- Ensures responses stay natural

The β parameter controls how tight the leash is!

Chapter 5

Direct Preference Optimization (DPO)

5.1 The Simplification

Story

In 2023, researchers asked: "Do we really need all this complexity?"
PPO requires:

- The policy model we're training
- A frozen reference model
- A reward model
- A value function network

That's 4 models in memory! Plus the complex RL training loop.

The breakthrough: There's a mathematical shortcut that lets us skip the reward model entirely!

5.1.1 The Mathematical Insight

Math: Formal & Intuitive Side-by-Side

The DPO Reparameterization Trick

Formal Math	What It Really Means
Step 1: Optimal policy in terms of reward	
$\pi^*(y x) = \frac{1}{Z(x)} \pi_{\text{ref}}(y x) \exp\left(\frac{r(x,y)}{\beta}\right)$	Best policy = reference \times exponential of reward
Step 2: Solve for reward	
$r(x, y) = \beta \log \frac{\pi^*(y x)}{\pi_{\text{ref}}(y x)} + \beta \log Z(x)$	Reward = β times log of policy ratio
Step 3: Substitute into Bradley-Terry	
$P(y_w \succ y_l) = \sigma\left(\beta \log \frac{\pi(y_w x)}{\pi_{\text{ref}}(y_w x)} - \beta \log \frac{\pi(y_l x)}{\pi_{\text{ref}}(y_l x)}\right)$	Preference in terms of policy ratios! (partition function Z cancels out!)

Breaking It Down Step-by-Step

Why is this magical? Let's compare:

PPO (Old way):

1. Train reward model: Learn $r(x, y)$ from preferences
2. Generate responses: Use policy π_θ
3. Score responses: Run through reward model to get $r(x, y)$
4. RL update: Adjust π_θ to maximize r (with KL constraint)

Models needed: Policy, Reference, Reward, Value = 4 models

DPO (New way):

1. No reward model needed!
2. Directly optimize policy on preferences
3. The ratio $\frac{\pi_\theta(y)}{\pi_{\text{ref}}(y)}$ implicitly encodes the reward!
4. Much simpler training loop

Models needed: Policy, Reference = 2 models (50% less memory!)

The key insight:

We don't need to explicitly model the reward function $r(x, y)$.

The policy ratio $\frac{\pi_\theta(y)}{\pi_{\text{ref}}(y)}$ already encodes all the information about what makes responses good!

Intuition:

- If π_θ makes a response more likely than π_{ref} does, that means the response is "good" (high implicit reward)
- If π_θ makes it less likely, it's "bad" (low implicit reward)
- We can train π_θ directly on preferences!

5.1.2 The DPO Loss Function

Math: Formal & Intuitive Side-by-Side

Training Policy Directly on Preferences

Formal Math	What It Really Means
$\mathcal{L}_{\text{DPO}}(\theta) = -\mathbb{E} \left[\log \sigma \left(\beta \log \frac{\pi_{\theta}(y_w x)}{\pi_{\text{ref}}(y_w x)} - \beta \log \frac{\pi_{\theta}(y_l x)}{\pi_{\text{ref}}(y_l x)} \right) \right]$	Make winner ratio larger than loser ratio
Breaking it apart:	
Winner term: $\beta \log \frac{\pi_{\theta}(y_w x)}{\pi_{\text{ref}}(y_w x)}$	Implicit reward for winner
Loser term: $\beta \log \frac{\pi_{\theta}(y_l x)}{\pi_{\text{ref}}(y_l x)}$	Implicit reward for loser
$\sigma(\text{winner} - \text{loser})$	Probability winner is better
$\log(\cdot)$	Log likelihood (penalizes wrong predictions)
—	Minimize loss
$\mathbb{E}[\cdot]$	Average over all preference pairs

Breaking It Down Step-by-Step

Complete DPO training example:

Preference:

- **Prompt:** "Explain photosynthesis"
- **Winner (y_w):** "Plants use sunlight to make food from CO and water"
- **Loser (y_l):** "Photosynthesis is the biochemical process involving chloroplast thylakoids..."

Winner is simpler and clearer. Let's train DPO on this preference!

Step 1: Compute probabilities for both responses

Winner (10 words):

- New model gives each word avg probability: 0.22
- Reference model gives: 0.20
- Full response probability: $\pi_{\theta}(y_w) \approx 0.22^{10}$, $\pi_{\text{ref}}(y_w) \approx 0.20^{10}$

Loser (12 words):

- New model gives: 0.17 per word
- Reference model gives: 0.19 per word
- Full response: $\pi_{\theta}(y_l) \approx 0.17^{12}$, $\pi_{\text{ref}}(y_l) \approx 0.19^{12}$

Step 2: Compute log ratios

Winner log ratio:

$$\begin{aligned}
 \log \frac{\pi_{\theta}(y_w | x)}{\pi_{\text{ref}}(y_w | x)} &= \log \frac{0.22^{10}}{0.20^{10}} \\
 &= 10 \times \log \frac{0.22}{0.20} \\
 &= 10 \times 0.095 \\
 &= 0.95
 \end{aligned}$$

Loser log ratio:

$$\begin{aligned}
 \log \frac{\pi_{\theta}(y_l | x)}{\pi_{\text{ref}}(y_l | x)} &= \log \frac{0.17^{12}}{0.19^{12}} \\
 &= 12 \times \log \frac{0.17}{0.19} \\
 &= 12 \times (-0.111) \\
 &= -1.33
 \end{aligned}$$

Step 3: Compute preference logit (with $\beta = 0.1$)

$$\begin{aligned}
 \text{logit} &= \beta \times (\text{winner ratio} - \text{loser ratio}) \\
 &= 0.1 \times (0.95 - (-1.33)) \\
 &= 0.1 \times 2.28 \\
 &= 0.228
 \end{aligned}$$

Step 4: Convert to probability

$$\begin{aligned}
 P(\text{winner} \succ \text{loser}) &= \sigma(0.228) \\
 &= \frac{1}{1 + e^{-0.228}} \\
 &= 0.557 \text{ or } 55.7\%
 \end{aligned}$$

Interpretation: Model is somewhat confident (55.7%) that winner is better. Not great yet!

Step 5: Compute loss

$$\begin{aligned}
 \mathcal{L}_{\text{DPO}} &= -\log(0.557) \\
 &= 0.585
 \end{aligned}$$

Step 6: Gradient descent updates π_{θ}

The gradients will:

- **Increase** $\pi_{\theta}(y_w | x)$ (make winner more likely)
- **Decrease** $\pi_{\theta}(y_l | x)$ (make loser less likely)

After training on 50,000 preferences...

Same preference pair now:

- Winner log ratio: +2.8
- Loser log ratio: -2.5
- Difference: 5.3
- Logit: $0.1 \times 5.3 = 0.53$
- $P = \sigma(0.53) = 0.629$ (62.9%)
- Loss: $-\log(0.629) = 0.464$ (much lower!)

Model learned to:

- Prefer simpler explanations
- Assign higher probability to clear, accessible language
- Assign lower probability to overly technical responses

All without an explicit reward model!

Why DPO works:

1. Every preference pair teaches the model: "This style of response is better than that style"
2. The policy ratios $\frac{\pi_{\theta}}{\pi_{\text{ref}}}$ encode implicit rewards
3. Training automatically makes winner ratios larger and loser ratios smaller
4. This is equivalent to RL, but much simpler to implement!

Result: Same performance as PPO, but 50% less memory and easier to train!

Chapter 6

Test-Time Compute: A New Scaling Paradigm

6.1 Beyond Training: Scaling at Inference

In Plain English

Traditional scaling: Bigger models + More training data = Better performance

Problem: This is expensive and slow. Training a frontier model costs millions of dollars!

New idea: What if we could trade *inference time* for accuracy?

Test-time compute scaling:

- Use the same model
- But let it "think" more at inference time
- Generate multiple solutions
- Pick the best one

Advantage: You can adjust quality vs speed on a per-query basis!

- Simple question: 1 attempt (fast)
- Important question: 64 attempts (accurate)

6.1.1 Best-of-N Sampling

Math: Formal & Intuitive Side-by-Side

The Probability of Success

Formal Math	What It Really Means
$P(\text{at least one correct}) = 1 - (1 - p)^N$	Probability of success with N attempts
Symbol Guide and WHY this formula works:	
p	Base accuracy (probability single attempt succeeds)
N	Number of independent attempts
$(1 - p)$	Probability of failure for single attempt
$(1 - p)^N$	Probability ALL N attempts fail (multiply independent failures)
$1 - (\cdot)$	Flip to get probability at least one succeeds
Why multiply $(1 - p)^N$?	
If attempts are independent, probability of ALL failing = $P(\text{fail})^N = (1 - p)^N$. We want at least ONE success, which is the complement: $1 - P(\text{all fail})$.	

Breaking It Down Step-by-Step

Let's see the math in action:

Scenario: Math problem, model has 60% accuracy ($p = 0.6$)

Question: How many attempts to get 95%+ success rate?

Attempt 1: Try once ($N = 1$)

$$\begin{aligned} P(\text{success}) &= 1 - (1 - 0.6)^1 \\ &= 1 - 0.4 \\ &= 0.60 \text{ or } 60\% \end{aligned}$$

Not good enough! Need 95%.

Attempt 2: Try twice ($N = 2$)

$$\begin{aligned} P(\text{success}) &= 1 - (1 - 0.6)^2 \\ &= 1 - 0.4^2 \\ &= 1 - 0.16 \\ &= 0.84 \text{ or } 84\% \end{aligned}$$

Better, but still not 95%!

Attempt 3: Try 4 times ($N = 4$)

$$\begin{aligned} P(\text{success}) &= 1 - (1 - 0.6)^4 \\ &= 1 - 0.4^4 \\ &= 1 - 0.0256 \\ &= 0.9744 \text{ or } 97.4\% \end{aligned}$$

Success! With 4 attempts, we exceed 95% success rate!

Full table:

N attempts	Calculation	Success rate	Quality
1	$1 - (0.4)^1$	60.0%	Base
2	$1 - (0.4)^2$	84.0%	Good
4	$1 - (0.4)^4$	97.4%	Excellent
8	$1 - (0.4)^8$	99.9%	Near perfect
16	$1 - (0.4)^{16}$	99.9999%	Essentially perfect

What if base accuracy is lower? ($p = 0.4$, only 40%)

N attempts	Calculation	Success rate
1	$1 - (0.6)^1$	40.0%
2	$1 - (0.6)^2$	64.0%
4	$1 - (0.6)^4$	87.0%
8	$1 - (0.6)^8$	98.3%
16	$1 - (0.6)^{16}$	99.9%

Even with mediocre 40% base accuracy, 8 attempts gives 98.3% success!

The power of multiple attempts:

Key insight: If attempts are independent, you can overcome low base accuracy with volume!

Trade-off:

- More attempts = Higher success rate
- But also = More computation time
- But also = Higher cost (more API calls)

Practical strategy:

- Simple queries: $N = 1$ (save money)
- Important queries: $N = 16$ or more (ensure correctness)
- Adjustable quality-cost dial!

Real-world example (DeepSeek-R1 on AIME 2024 - very hard math competition):

Method	Success Rate	Compute Cost	vs Humans
pass@1	79.8%	1×	Much better
pass@8	84.2%	8×	Much better
pass@64	86.7%	64×	Much better
Human experts	~50%	N/A	Baseline

Analysis:

- Even with 1 attempt, model beats humans (79.8% vs 50%)
- With 64 attempts, model achieves 86.7% (near-perfect)
- Cost scales linearly ($64 \times$ attempts = $64 \times$ cost)
- Quality scales logarithmically (diminishing returns)

Chapter 7

DeepSeek-R1: A Revolutionary Approach

7.1 The Paradigm Shift

Key Point

What Makes DeepSeek-R1 Revolutionary:

DeepSeek-R1 challenged conventional wisdom by showing you can achieve state-of-the-art reasoning abilities through **pure reinforcement learning**, skipping supervised fine-tuning entirely!

- **Traditional approach:** Pretrain → SFT → RLHF
- **DeepSeek-R1:** Pretrain → RLHF (that's it!)
- **Result:** Complex reasoning emerges naturally from RL

Story

The "Aha!" Moment:

Imagine teaching someone math by only showing them worked examples (SFT approach). Now imagine instead giving them problems and just telling them "right" or "wrong" after they solve them (RL approach).

Surprisingly, the second approach led to *discovering* problem-solving strategies that weren't explicitly taught! The model figured out that showing step-by-step reasoning led to better final answers — and it learned to do this on its own.

This is like a student independently discovering that "showing your work" helps catch mistakes, without being told to do so.

7.2 Why Skip SFT?

In Plain English

The Problem with SFT:

When you show a model thousands of examples of "correct" reasoning, you're teaching it to *imitate* those patterns. But imitation has limits:

- The model can only be as good as the examples you show it
- It learns surface patterns, not deep understanding

- It's expensive to collect thousands of high-quality reasoning traces
- The diversity is limited by human annotators

The RL Alternative:

Instead, just give the model problems and a simple reward signal: "Did you get the right answer?" Now the model must *discover* how to reason! And it turns out models discover more creative and effective strategies than humans would have taught them.

7.3 Group Relative Policy Optimization (GRPO)

Math: Formal & Intuitive Side-by-Side

Formal Definition:

GRPO objective:

$$\mathcal{L}_{\text{GRPO}} = -\mathbb{E}_{x, \{y_i\}_{i=1}^G} \left[\frac{1}{G} \sum_{i=1}^G (r(x, y_i) - \bar{r}_G) \cdot \log \pi_{\theta}(y_i | x) \right]$$

Where:

- G = group size (typically 4-8)
- y_i = output i in the group
- $r(x, y_i)$ = reward for output i
- $\bar{r}_G = \frac{1}{G} \sum_{j=1}^G r(x, y_j)$ = group average reward

Key insight: Use group average as baseline instead of separate value network!

Plain English:

For each problem:

1. Generate multiple attempts (a "group")
2. Check which ones are correct/incorrect
3. Calculate the average reward for the group
4. Update the model:

- Make above-average attempts more likely
- Make below-average attempts less likely

Why this works better:

The group average is a natural baseline, it adapts to problem difficulty automatically! Hard problems have lower group averages, easy problems have higher ones.

No need for a separate critic network like PPO!

Breaking It Down Step-by-Step

GRPO Example: Math Problem

Problem: "What is $\sqrt{144} + 12 \times 3$?"

Correct answer: 48

Step 1: Generate a group of 4 attempts

Attempt	Model Output	Final Answer	Reward
1	"12 + 36 = 48"	48	+1
2	" $\sqrt{144} = 12$, then $12 + 12 \times 3$..."	48	+1
3	" $\sqrt{144} = 14$, $14 + 36 = 50$ "	50	0
4	"144 + 36 = 180"	180	0

Step 2: Calculate group average

$$\bar{r}_G = \frac{1 + 1 + 0 + 0}{4} = \frac{2}{4} = 0.5$$

Step 3: Calculate advantages (reward - average)

Attempt	Reward	Group Avg	Advantage
1	+1	0.5	+0.5
2	+1	0.5	+0.5
3	0	0.5	-0.5
4	0	0.5	-0.5

Step 4: Update policy

- Attempts 1 & 2: Positive advantage → Increase probability
- Attempts 3 & 4: Negative advantage → Decrease probability

What the model learns:

- Showing step-by-step work (attempts 1 & 2) leads to correct answers
- Computing $\sqrt{144}$ first helps (attempt 2)
- Skipping steps leads to errors (attempts 3 & 4)

Over many problems, the model discovers that detailed reasoning → better rewards!

7.4 The Emergence of Chain-of-Thought

Intuition

The Most Surprising Discovery:

Nobody told DeepSeek-R1 to "think step-by-step." Nobody showed it examples of chain-of-thought reasoning. Yet it discovered this strategy on its own!

Why did this happen?

1. Longer, more detailed responses naturally have lower probability (more tokens to predict correctly)
2. But longer reasoning traces lead to better final answers (higher reward)
3. The model learns: "The reward boost from correct answers outweighs the cost of longer sequences"
4. Result: The model develops elaborate reasoning chains

This is like a student discovering that writing out their thought process helps them catch mistakes, without a teacher telling them to do so!

7.5 Training Stages and "Aha Moments"

Practical Tip

The Training Journey:

DeepSeek-R1's training showed clear phases:

Stage 1: Random Exploration (First 100-500 steps)

- Model outputs are chaotic
- Rewards are near zero
- No clear patterns

Stage 2: Basic Patterns (500-2000 steps)

- Model learns to format outputs correctly
- Discovers that attempting calculation → higher reward than random text
- Still many errors

Stage 3: "Aha!" Moment (2000-5000 steps)

- **Sudden emergence:** Model starts showing intermediate steps
- Reasoning chains appear spontaneously
- Accuracy jumps dramatically (30% → 60%)
- This is NOT gradual—it's a phase transition!

Stage 4: Refinement (5000+ steps)

- Reasoning becomes more sophisticated
- Model learns to verify its own work
- Self-correction emerges
- Accuracy continues improving (60% → 85%+)

7.6 Comparison: DeepSeek-R1 vs OpenAI o1

Aspect	DeepSeek-R1	OpenAI o1
Training approach	Pure RL (no SFT)	Likely SFT → RLHF
Public details	Open weights, methodology	Closed, limited info
Reasoning style	Very explicit, visible	Hidden "thinking"
Cost	Lower (simpler pipeline)	Higher (multi-stage)
Performance	State-of-the-art	State-of-the-art
Novel contribution	Proved SFT unnecessary	First to show test-time scaling

Key Point**Key Takeaway:**

Both models prove that investing computation at training time (RL) or inference time (test-time compute) produces models that can *reason* rather than just pattern-match.

DeepSeek-R1's contribution: Showing you can skip expensive human-annotated reasoning examples entirely!

7.7 Practical Implications

Watch Out**Important Considerations:**

While DeepSeek-R1's approach is revolutionary, it's not always the best choice:

When pure RL works well:

- Objective rewards (code execution, math verification)
- Lots of compute available
- Want to discover novel strategies

When SFT still helps:

- Subjective tasks (creative writing, style)
- Limited compute
- Need specific output formats
- Cold-start: RL from random initialization is very slow

Best of both worlds: Minimal SFT for basic instruction following, then aggressive RL for capability development.

Chapter 8

RL at Different Training Stages

8.1 The Four Stages of RL Deployment

In Plain English

Reinforcement learning isn't just something you add at the end! You can apply RL at different points in a model's life cycle, each with different benefits and costs:

1. **During Pretraining/Continued Pretraining** - Teaching fundamental capabilities
2. **During Fine-tuning** - Aligning with human preferences
3. **During Inference** - Spending extra compute per query
4. **Continuous/Online Learning** - Adapting from production traffic

Think of these like different times to teach a skill:

- Pretraining RL = Teaching a child foundational problem-solving
- Fine-tuning RL = Teaching specific preferences (be polite, be concise)
- Inference RL = Taking time to "think" before answering
- Online RL = Learning from every conversation you have

8.2 Stage A: RL During Pretraining

8.2.1 The DeepSeek-R1 Approach

Key Point

Why Apply RL During Pretraining?

Traditional view: Pretraining = next-token prediction on massive text corpora. Just learn to predict, nothing fancy.

DeepSeek's insight: Why not teach the model to *reason* during pretraining itself?

Benefits:

- Reasoning becomes a fundamental capability, not an afterthought
- Model discovers strategies naturally (chain-of-thought, self-verification)

- No need for expensive human-annotated reasoning traces
- Better scaling: more compute → better reasoning

Costs:

- 2-4× more expensive than standard pretraining
- Requires careful reward design
- Risk of reward hacking if not careful

8.2.2 Curriculum Learning: Easy to Hard

Analogy

You wouldn't teach calculus to a kindergartener! You start with counting, then addition, then multiplication, and eventually work up to derivatives.

RL during pretraining works the same way:

Stage 1: Easy problems (arithmetic)

- "What is $5 + 7$?"
- Clear right/wrong answers
- Fast feedback
- Builds confidence (high reward signals)

Stage 2: Medium problems (multi-step math)

- "If John has 5 apples and buys 3 more, then gives 2 away, how many does he have?"
- Requires planning
- Model learns to break down problems

Stage 3: Hard problems (competition math)

- "Prove that $\sqrt{2}$ is irrational"
- Requires sophisticated reasoning
- Only works if earlier stages succeeded

This curriculum is crucial! Skip easy problems → model never learns basic reasoning → fails on everything.

8.2.3 Computational Cost Analysis

Breaking It Down Step-by-Step

Cost Comparison: Standard Pretraining vs RL-Enhanced Pretraining**Standard Pretraining:**

- Generate 1 token, predict next token, compute loss

- Cost per example: C_{forward} (one forward pass)
- Tokens processed: $\sim 10\text{-}100\text{T}$ tokens
- Total cost: $C_{\text{standard}} = 10\text{T} \times C_{\text{forward}}$

RL-Enhanced Pretraining (DeepSeek approach):

- Generate complete response (forward pass): C_{forward}
- Evaluate reward (might need verification): C_{reward}
- Compute policy gradient and update: C_{backward}
- Generate multiple samples per problem: $\times G$ (group size, typically 4-8)

Cost per example: $G \times (C_{\text{forward}} + C_{\text{reward}} + C_{\text{backward}})$

Multiplier:

$$\text{Cost multiplier} = \frac{G \times (C_{\text{forward}} + C_{\text{reward}} + C_{\text{backward}})}{C_{\text{forward}}} \approx 2 - 4\times$$

Is it worth it?

- $3\times$ more expensive to train
- But resulting model has emergent reasoning abilities
- Saves millions of dollars on human annotation
- Discovers strategies humans wouldn't have taught

Verdict: For reasoning-heavy applications, absolutely worth it! For general language modeling, maybe not.

8.3 Stage B: RL During Fine-tuning (RLHF)

In Plain English

This is the most common use of RL, and what we covered extensively in earlier chapters:

Standard RLHF Pipeline:

1. Start with pretrained model
2. Supervised fine-tuning (SFT) on high-quality examples
3. Train reward model on human preferences
4. PPO or DPO to optimize policy

Key difference from pretraining RL:

- Pretraining RL: Teaching *capabilities* (how to reason)
- Fine-tuning RL: Teaching *preferences* (what humans like)

Think of it as: pretraining RL makes the model smarter, fine-tuning RL makes it more helpful.

8.3.1 Online vs Offline RL**Math: Formal & Intuitive Side-by-Side****Formal Distinction:****Offline RL:**

- Fixed dataset $\mathcal{D} = \{(x_i, y_i, r_i)\}$
- Policy π_θ learns from \mathcal{D}
- No new data generation during training

Online RL:

- Policy π_θ generates new samples
- Samples evaluated with reward $r(x, y)$
- Dataset grows: $\mathcal{D}_t \rightarrow \mathcal{D}_{t+1}$
- Distribution shifts as policy improves

On-policy (like PPO):

Use samples from π_θ to update π_θ

Off-policy (like DQN):

Use samples from old π_{old} to update π_θ

Plain English:**Offline RL:**

Like learning to cook by reading a fixed cookbook. You study the recipes (dataset) but never actually cook (generate) to see what happens.

Pros: Safe, cheap, reproducible

Cons: Limited by cookbook quality

Online RL:

Like learning to cook by actually cooking! You try recipes, taste results, learn from mistakes, try new variations.

Pros: Discovers novel strategies, adapts to changing preferences

Cons: Expensive (lots of cooking/generation), risky (might make terrible food/outputs)

Most practical systems: Hybrid! Start offline (safe learning), then carefully add online data.

8.3.2 Data Distribution Shift

Watch Out

The Online RL Challenge:

As your policy improves, the data distribution changes:

Iteration 1:

- Policy is mediocre
- Generates mostly poor responses
- Dataset has ~30% good examples

Iteration 5:

- Policy is better
- Generates mostly good responses
- Dataset now ~80% good examples

The Problem:

The reward model was trained on the *original* distribution (30% good)! It might not accurately score the *new* distribution (80% good).

Symptoms:

- Reward model assigns similar scores to everything (saturation)
- Policy stops improving (no learning signal)
- Or worse: reward hacking (exploiting reward model mistakes)

Solutions:

1. Retrain reward model periodically on new data
2. Use multiple reward models (ensemble)
3. Strong KL penalty to prevent policy from drifting too far
4. Conservative updates (small learning rates)

8.4 Stage C: RL During Inference (Test-Time Compute)

Intuition

The Fundamental Idea:

All the RL we've discussed happens during *training*. But what if we use RL-like ideas *during inference* for each query?

This is what "test-time compute" means: spending extra computation when answering to improve quality.

Three flavors:

1. **Generate multiple answers, pick best** (best-of-N sampling)
2. **Self-correction loops** (generate → critique → revise)
3. **Tree search** (explore multiple reasoning paths)

We covered best-of-N in depth earlier. Let's explore the other two!

8.4.1 Self-Correction Loops

Practical Tip

The Self-Correction Pattern:

Algorithm Overview:

1. **Generate Initial Response**
 - `response ← model.generate(prompt)`
2. **Iterative Refinement Loop** (repeat up to 3 times)
 - (a) **Critique:** Ask model to review current response
 - `critique ← model.generate("Review this: " + response)`
 - (b) **Check:** If critique says "no errors" → stop and return
 - (c) **Revise:** Generate improved version based on critique
 - `response ← model.generate(original + critique + "improve")`
3. **Return** final refined response

When this works:

- Model is good at *verification* but sometimes makes mistakes in generation
- Problems where checking is easier than solving
- High-stakes applications (worth the extra compute)

When this fails:

- Model's critique ability is poor (can't identify its own errors)
- Model gets stuck in loops (fixes one thing, breaks another)
- Hallucinations in critique (invents nonexistent errors)

8.4.2 Tree Search: Beam Search and MCTS

Analogy

Beam Search:

Imagine you're navigating a maze, but at each intersection you can clone yourself and try multiple paths simultaneously. You keep the K most promising clones and discard the rest.

In LLM generation:

- At each token position, consider top K possibilities
- Score each partial sequence (with reward model or value function)
- Keep top K sequences, discard others
- Continue until all sequences finish

- Return highest-scoring complete sequence

Cost: $K \times$ normal inference cost

Benefit: Can escape local minima (early mistakes)

Monte Carlo Tree Search (MCTS):

Like beam search, but smarter! Instead of keeping K parallel paths, you:

1. **Selection:** Pick most promising partial path to expand
2. **Expansion:** Try possible next steps
3. **Simulation:** Complete the path (rollout)
4. **Backpropagation:** Update value estimates for all ancestors

This is what AlphaGo uses! And it's increasingly used for LLM reasoning tasks.

Breaking It Down Step-by-Step

MCTS Example: Math Problem

Problem: "Solve: $2x + 5 = 13$ "

Initial state: Empty response, need to choose first step.

Possible first steps:

- A: "Subtract 5 from both sides"
- B: "Divide both sides by 2"
- C: "Add 5 to both sides"

Round 1: Try each once

- Try A \rightarrow continue \rightarrow " $2x = 8$ " \rightarrow " $x = 4$ " (Reward: +1)
- Try B \rightarrow continue \rightarrow " $x + 2.5 = 6.5$ " \rightarrow " $x = 4$ " (Reward: +1, but more steps)
- Try C \rightarrow continue \rightarrow " $2x + 10 = 18$ " \rightarrow wrong path (Reward: 0)

Update values:

- Value(A) = 1.0 (best!)
- Value(B) = 0.8 (correct but inefficient)
- Value(C) = 0.0 (leads to error)

Round 2: Focus on promising branches

Since A has highest value, explore more variations:

- A \rightarrow " $2x = 8$ " \rightarrow "Therefore, $x = 8/2$ " \rightarrow " $x = 4$ "
- A \rightarrow " $2x = 8$ " \rightarrow "Dividing by 2: $x = 4$ "

Both work! Value(A) increases to 1.0 (very confident).

Final decision: Pick path A (highest value), return best completion found.

Key insight: MCTS allocates compute intelligently—spend more time exploring promising paths!

8.5 Stage D: Continuous/Online RL from Production

Key Point

The Ultimate Goal:

Imagine your deployed AI system *continuously learns* from every user interaction! Every conversation makes it slightly better.

The Dream:

- User asks question
- Model responds
- User implicitly/explicitly provides feedback (clicks, ratings, edits)
- Model updates immediately
- Next user gets improved model

The Reality: This is extremely difficult!

8.5.1 Challenges in Production RL

Watch Out

Why Production RL Is Hard:

1. Exploration vs Exploitation

- **Exploitation:** Always give best-known answer (users happy now)
- **Exploration:** Try new strategies (might be better, might annoy users)

In production, exploration can upset users! Can't just try random things.

Solution: Careful exploration (small percentage of traffic, only low-stakes queries)

2. Reward Signal Delay

User feedback might come minutes/hours/days later! How do you attribute success/failure?

Solution: Credit assignment algorithms, session-level rewards

3. Reward Hacking at Scale

With millions of updates per day, model might find exploits:

- Learns to give popular answers, not correct ones
- Optimizes for clicks, not helpfulness
- Finds reward model bugs and exploits them

Solution: Multiple reward models, human oversight, safety constraints

4. Distribution Shift

Today's users might be different from tomorrow's! Model trained on morning users might fail on evening users.

Solution: Adaptive learning rates, forgetting mechanisms, diverse data collection

8.5.2 Safeguards and Best Practices

Practical Tip

How Companies Actually Do Production RL:

1. Staged Rollout

- Test on internal users first (employees)
- Then 1% of real users (canary deployment)
- Gradually increase to 10%, 50%, 100%
- Monitor metrics at each stage
- Rollback immediately if issues

2. Dual Model System

- **Production model:** Serves most users, stable
- **Experimental model:** Learns from new data, tested on small percentage
- Only promote experimental → production if clearly better

3. Offline Evaluation First

- Collect production data (queries + responses + feedback)
- Train updated model offline
- Evaluate on held-out test set
- Simulate counterfactuals (what would new model have done?)
- Only deploy if offline metrics improve

4. Human-in-the-Loop

- Random sample of updates reviewed by humans
- Anomaly detection flags suspicious updates
- Red team testing before each deployment

5. Conservative Updates

- Strong KL constraint (stay close to base model)
- Small learning rates (slow, steady improvement)
- Frequent snapshots (can rollback anytime)

Chapter 9

Process Reward Models (PRMs)

9.1 Outcome Rewards vs Process Rewards

In Plain English

Imagine you're a math teacher grading a student's work. You have two options:

Option 1: Outcome-Based Grading (ORM)

- Only look at the final answer
- Correct answer = full credit
- Wrong answer = no credit
- Don't care how they got there

Option 2: Process-Based Grading (PRM)

- Grade each step of the solution
- Step 1 correct → partial credit
- Step 2 correct → more partial credit
- Final answer correct → remaining credit
- Even if final answer is wrong, can get partial credit for correct reasoning

Which produces better learning? Obviously process-based! Students learn *how to think*, not just how to guess answers.

Same principle applies to training AI!

Math: Formal & Intuitive Side-by-Side

Formal Definitions:**Outcome Reward Model (ORM):**

$$r_{\text{ORM}}(x, y) = r(x, y_{\text{final}})$$

where y_{final} is the final answer.

Function signature:

$$r_{\text{ORM}} : (\text{prompt}, \text{response}) \rightarrow \mathbb{R}$$

Process Reward Model (PRM):

$$r_{\text{PRM}}(x, y) = \sum_{t=1}^T r_t(x, y_{1:t})$$

where y_t is step t of the solution.

Function signature:

$$r_t : (\text{prompt}, \text{partial response}) \rightarrow \mathbb{R}$$

Returns reward for each step t .

Key difference:

ORM: Single score for entire response

PRM: Score for each reasoning step

Plain English:**Outcome Reward Model:**

- Input: Complete response
- Output: Single number (how good is the final answer?)
- Example: "Is the final answer correct? +10 if yes, -5 if no"

Process Reward Model:

- Input: Partial response (up to some step)
- Output: Number for that step (how good is this step?)
- Run multiple times, once per step
- Total reward = sum of all step rewards

Example conversation:

ORM: "Here's the solution. Final answer: $x = 4$. Is it correct?"

PRM: "Let me check each step:

- Step 1: Good! +2
- Step 2: Good! +2
- Step 3: Good! +2
- Step 4: Correct! +4
- Total: +10

"

9.2 The Power of Step-by-Step Feedback

Breaking It Down Step-by-Step

Detailed Comparison: ORM vs PRM

Problem: Solve $2x + 5 = 13$

Scenario 1: Correct Solution

Step	Work Shown	ORM	PRM
1	"Subtract 5 from both sides"	—	+2
2	" $2x = 8$ "	—	+2
3	"Divide both sides by 2"	—	+2
4	" $x = 4$ "	—	+4
Final	Answer: $x = 4$	+10	+10

Both ORM and PRM give full credit. So far, no difference!

Scenario 2: Wrong Final Answer, But Good Process

Step	Work Shown	ORM	PRM
1	"Subtract 5 from both sides"	—	+2
2	" $2x = 8$ "	—	+2
3	"Divide both sides by 2"	—	+2
4	" $x = 5$ " (arithmetic error!)"	—	-2
Final	Answer: $x = 5$	-5	+4

Big difference!

- ORM: Heavily penalizes (-5) even though process was 75% correct
- PRM: Gives +4, recognizing good reasoning despite arithmetic slip

Scenario 3: Correct Final Answer, But Flawed Reasoning

Step	Work Shown	ORM	PRM
1	"Multiply both sides by 2"	—	-1
2	" $4x + 10 = 26$ "	—	0
3	"Subtract 10: $4x = 16$ "	—	+1
4	"Divide by 4: $x = 4$ "	—	+2
Final	Answer: $x = 4$	+10	+2

HUGE difference!

- ORM: Full credit (+10)—can't tell the reasoning was flawed
- PRM: Only +2—correctly identifies the unnecessarily complicated approach

What the model learns:

- With ORM: "Somehow I got the right answer. Keep doing... whatever I did?"
- With PRM: "My initial strategy was wrong. I should subtract first, not multiply."

PRM teaches *how to reason*, not just how to get lucky!

9.3 Training Process Reward Models

Practical Tip**How to Train a PRM:****Step 1: Collect Step-Level Annotations**

This is expensive! For each problem:

1. Model generates a solution with explicit reasoning steps
2. Human annotator reviews *each individual step*:
 - **Correct** reasoning
 - **Incorrect** reasoning
 - ? **Unclear/ambiguous**
3. Store training triple: (problem, partial_solution_{1:t}, label_t)

Example Annotation Sequence:

Problem: "What is 15% of 200?"

- **After Step 1:** "15% means 0.15"
→ Label: **POSITIVE** (correct understanding)
- **After Step 2:** "15% means 0.15, so 0.15×200 "
→ Label: **POSITIVE** (correct setup)
- **After Step 3:** "15% means 0.15, so $0.15 \times 200 = 3$ "
→ Label: **NEGATIVE** (arithmetic error!)
- **After Step 4:** "15% means $15/100 = 0.15$, so $0.15 \times 200 = 30$ "
→ Label: **POSITIVE** (correct solution)

Step 2: Train the PRM**Model Architecture:**

- Base: Transformer model (similar to ORM)
- Input: (problem, partial_solution_{1:t})
- Output: $P(\text{step}_t \text{ is correct})$
- Loss: Binary cross-entropy on step-level labels

Training Algorithm:

1. **For each** training example $(p, s_{1:t}, y_t)$:
 - (a) Encode problem and partial solution
 - $\text{score}_t \leftarrow \text{PRM}(p, s_{1:t})$
 - (b) Compute binary classification loss
 - $\text{target} = \begin{cases} 1.0 & \text{if } y_t = \text{correct} \\ 0.0 & \text{if } y_t = \text{incorrect} \end{cases}$
 - $\mathcal{L} = \text{BCE}(\text{score}_t, \text{target})$
 - (c) Update model parameters
 - $\theta \leftarrow \theta - \alpha \nabla_{\theta} \mathcal{L}$

9.3.1 Data Collection Strategies

Watch Out

Challenge: Step-Level Annotation is Expensive!

Annotating every single step is 10-20× more expensive than just labeling final answers.

Practical solutions:

1. Synthetic Data

- For math: Use symbolic solvers to verify each step
- For code: Check if each line compiles and maintains correctness
- Cheaper but only works for objective tasks

2. Sparse Annotation

- Only annotate steps where model is uncertain
- Active learning: PRM requests labels for confusing steps
- Reduces annotation cost by 5-10×

3. Weak Supervision

- Automatic heuristics (length, coherence, keyword matching)
- Not perfect but cheap at scale
- Use as noisy training signal, validate with small amount of human labels

4. Knowledge Distillation

- Train expensive PRM on small amount of human data
- Use it to label more data automatically
- Train cheaper PRM on larger automatically-labeled dataset

9.4 Using PRMs for Best-of-N Sampling

Intuition

Remember best-of-N sampling? Generate N responses, pick the best one?

With an ORM: Score each complete response, pick highest.

With a PRM: You can do something smarter!

PRM-guided best-of-N:

1. Generate N responses, each with explicit reasoning steps
2. For each response, score *every step* with PRM
3. Total score = sum of step scores (or product of step probabilities)
4. Pick response with highest total score

Why this is better:

- Can identify flawed reasoning even if final answer looks good

- More fine-grained signal (scoring 10 steps vs 1 answer)
- Can detect where model goes wrong (useful for debugging)

Breaking It Down Step-by-Step

PRM Best-of-N Example

Problem: "A store has a 20% off sale. If a jacket originally costs \$80, what's the final price?"

Generate 3 candidate responses:

Response 1:

- | | |
|--------------------------------------|---------|
| 1. "20% off means we pay 80%" | PRM: +2 |
| 2. "80% of \$80 = 0.80×80 " | PRM: +2 |
| 3. " $0.80 \times 80 = 64$ " | PRM: +2 |
| 4. "Final price: \$64" | PRM: +4 |

Total: +10 (perfect reasoning)

Response 2:

- | | |
|-------------------------------------|-----------------------------------|
| 1. "20% off means discount is \$20" | PRM: -1 |
| 2. "\$80 - \$20 = \$60" | PRM: 0 (follows wrong assumption) |
| 3. "Final price: \$60" | PRM: -2 |

Total: -3 (incorrect logic)

Response 3:

- | | |
|---|---------|
| 1. "20% of \$80 = $\$80 \times 0.20 = \16 " | PRM: +2 |
| 2. "\$80 - \$16 = \$64" | PRM: +2 |
| 3. "Final price: \$64" | PRM: +4 |

Total: +8 (correct, but less explicit)

Decision:

Pick Response 1 (highest PRM score: +10)!

What if we used ORM instead?

All three arrive at final numbers, but:

- Response 1: \$64 → ORM: +10
- Response 2: \$60 → ORM: -5
- Response 3: \$64 → ORM: +10

ORM thinks Responses 1 and 3 are equally good! PRM correctly identifies Response 1 as having more thorough reasoning.

9.5 Why PRMs Dramatically Improve Math and Reasoning

Key Point

Empirical Results from "Let's Verify Step by Step" (Lightman et al., 2023):

On MATH benchmark (competition-level problems):

Method	Accuracy	Improvement
Base model (greedy)	42.0%	—
+ ORM best-of-100	58.3%	+16.3%
+ PRM best-of-100	72.2%	+30.2%

PRM gives 2× the improvement of ORM!

Why such dramatic gains?

1. **Better exploration:** PRM identifies promising partial solutions early, guides generation toward correct final answers
2. **Catches lucky guesses:** ORM can't tell if model got correct answer through sound reasoning or luck; PRM can
3. **Partial credit:** Even failed attempts provide learning signal if some steps were correct
4. **Fine-grained feedback:** Model learns "where I went wrong" not just "I got it wrong"

9.6 Combining ORMs and PRMs

Practical Tip

Best Practice: Use Both!

Ensemble approach:

$$\text{Final score} = \alpha \cdot \text{PRM score} + (1 - \alpha) \cdot \text{ORM score}$$

where $\alpha \in [0, 1]$ controls the balance.

Typical values: $\alpha = 0.7$ (favor PRM, but don't ignore outcomes entirely)

Why combine?

- PRM: Excellent at judging reasoning process
- ORM: Excellent at judging final answer correctness
- Together: Best of both worlds!

Real-world pipeline:

1. Generate N candidate responses (e.g., $N = 64$)
2. Score each with PRM (step-by-step)
3. Keep top K by PRM score (e.g., $K = 8$)

4. Score these K with ORM (final answer)
5. Return highest ORM score from the K

This two-stage approach is computationally efficient: PRM does heavy filtering, ORM makes final decision.

Chapter 10

Modern RL Algorithms Beyond PPO/DPO

10.1 The Expanding Algorithm Landscape

In Plain English

PPO and DPO are the workhorses of RLHF, but they're not the only options! Researchers have developed many variations, each with different trade-offs.

Why so many algorithms?

Different projects have different constraints:

- Limited compute? Try RLOO or GRPO
- Don't have paired preferences? Try KTO
- Want training stability? Try IPO
- Want efficiency? Try ORPO

Let's tour the landscape!

10.2 Algorithm Comparison Table

Algorithm	Key Innovation	When to Use
PPO	Clipped policy gradient, value network, multiple epochs	General-purpose, well-tested, scales well
DPO	Direct preference optimization without reward model	Want simplicity, have paired preferences
GRPO	Group-based baselines, no value network	Large-scale training, many GPUs available
RLOO	Leave-one-out baselines, variance reduction	Limited compute, small-scale projects
KTO	Works with unpaired feedback (only "good" or "bad")	Cheaper data collection, no comparisons needed
IPO	Identity preference optimization, better regularization	Want stability, prevent reward hacking
ORPO	Combines SFT + preference learning in one step	Maximum efficiency, fewer training stages
REINFORCE	Simple policy gradient, foundation of all methods	Learning/teaching, understanding basics

10.3 GRPO: Group Relative Policy Optimization

We covered this extensively in the DeepSeek-R1 chapter! See that section for full details.

Key Point

Quick recap:

- Generate groups of responses (typically 4-8 per prompt)
- Use group average as baseline instead of value network
- Simpler than PPO (no critic), more stable than REINFORCE
- What DeepSeek-R1 uses!

When to choose GRPO:

- Training at scale (many GPUs)
- Want simplicity (no value network to train)
- Can afford generating multiple samples per prompt

10.4 RLOO: REINFORCE Leave-One-Out

Math: Formal & Intuitive Side-by-Side

Formal Definition:

For N samples from prompt x : $\{y_1, \dots, y_N\}$
REINFORCE baseline:

$$b = \frac{1}{N} \sum_{i=1}^N r(y_i)$$

RLOO baseline for sample i :

$$b_{-i} = \frac{1}{N-1} \sum_{j \neq i} r(y_j)$$

Gradient estimate:

$$\nabla \mathcal{L}_i = (r(y_i) - b_{-i}) \cdot \nabla \log \pi_{\theta}(y_i|x)$$

Key insight: Each sample gets a baseline computed from the *other* samples, not including itself!

Plain English:

When grading sample i , don't include it in the average!

Why?

If sample i is really good, including it in the average would make the average higher, making sample i look less special.

Analogy:

Imagine you're competing in a race. Your performance should be measured against *the other runners*, not against an average that includes your own time!

Benefits:

- Lower variance than vanilla REINFORCE
- No extra neural networks needed (like PPO's critic)
- Works with as few as $N = 2$ samples!

Breaking It Down Step-by-Step

RLOO Example

Prompt: "Write a poem about spring"

Generate 4 responses:

#	Response	Reward
1	"Spring is nice..." (mediocre)	3
2	"Blossoms dance in gentle breeze..." (good!)	8
3	"Spring spring spring..." (bad)	1
4	"Flowers bloom as winter ends..." (okay)	5

Standard REINFORCE baseline:

$$b = \frac{3 + 8 + 1 + 5}{4} = \frac{17}{4} = 4.25$$

Advantages:

- Sample 1: $3 - 4.25 = -1.25$ (decrease probability)
- Sample 2: $8 - 4.25 = +3.75$ (increase probability)
- Sample 3: $1 - 4.25 = -3.25$ (decrease probability)
- Sample 4: $5 - 4.25 = +0.75$ (increase probability slightly)

RLO baselines (leave-one-out):

For sample 1: $b_{-1} = \frac{8+1+5}{3} = 4.67$

For sample 2: $b_{-2} = \frac{3+1+5}{3} = 3.00$

For sample 3: $b_{-3} = \frac{3+8+5}{3} = 5.33$

For sample 4: $b_{-4} = \frac{3+8+1}{3} = 4.00$

RLO advantages:

- Sample 1: $3 - 4.67 = -1.67$ (larger decrease!)
- Sample 2: $8 - 3.00 = +5.00$ (larger increase!)
- Sample 3: $1 - 5.33 = -4.33$ (larger decrease!)
- Sample 4: $5 - 4.00 = +1.00$ (larger increase)

Notice: RLO gives stronger signals! Sample 2 (the best one) gets a bigger boost because we're not diluting the baseline with its own high score.

This leads to lower variance and faster learning!

10.5 KTO: Kahneman-Tversky Optimization

Intuition

The Problem KTO Solves:

All the methods we've discussed so far require *paired preferences*:

"Response A is better than Response B"

But collecting paired comparisons is expensive! You need:

- Two responses for each prompt
- Human annotator to compare them

- Lots of time and money

What if you only have:

- Response A: (good)
- Response B: (bad)
- Response C: (good)

No comparisons, just independent labels! This is $2\times$ cheaper to collect.
KTO makes this work!

Math: Formal & Intuitive Side-by-Side

Formal Definition:

KTO loss for desirable output y_+ :

$$\mathcal{L}_{\text{KTO}}^+ = 1 - \sigma \left(\beta \left(\log \frac{\pi_\theta(y_+|x)}{\pi_{\text{ref}}(y_+|x)} - \mathbb{E}_{y \sim \pi_{\text{ref}}} \left[\log \frac{\pi_\theta(y|x)}{\pi_{\text{ref}}(y|x)} \right] \right) \right)$$

KTO loss for undesirable output y_- :

$$\mathcal{L}_{\text{KTO}}^- = \sigma \left(\beta \left(\log \frac{\pi_\theta(y_-|x)}{\pi_{\text{ref}}(y_-|x)} - \mathbb{E}_{y \sim \pi_{\text{ref}}} \left[\log \frac{\pi_\theta(y|x)}{\pi_{\text{ref}}(y|x)} \right] \right) \right)$$

Total loss:

$$\mathcal{L}_{\text{KTO}} = \mathbb{E}[\mathcal{L}_{\text{KTO}}^+] + \mathbb{E}[\mathcal{L}_{\text{KTO}}^-]$$

Plain English:

For **good** responses (y_+):

- Increase their probability relative to baseline
- But not uniformly—use a reference distribution
- Penalize if deviation is too large

For **bad** responses (y_-):

- Decrease their probability relative to baseline
- Again, calibrated against reference

Key insight:

Even without explicit comparisons ("A vs B"), we can:

- Push up probability of good responses
- Push down probability of bad responses
- The reference distribution provides implicit comparison!

Name origin:

Based on Kahneman & Tversky's prospect theory: humans judge outcomes relative to reference points, not absolute values. Same principle here!

Practical Tip

When to Use KTO:

Good fit:

- You have upvotes/downvotes (Reddit, Twitter style)

- You have star ratings but no explicit comparisons
- Data collection budget is limited
- You want simplicity (no paired data needed)

Not ideal when:

- You already have high-quality paired preferences
- You need very precise ranking information
- Performance is critical (KTO slightly underperforms DPO/PPO with good data)

Practical tip:

Use KTO for initial training with cheap data, then fine-tune with DPO/PPO on smaller high-quality paired dataset. Best of both worlds!

10.6 IPO: Identity Preference Optimization

Key Point**The Problem IPO Solves:**

DPO works great, but it has a weakness: **reward hacking through length!**

What happens:

DPO learns: "Longer responses tend to be preferred"

Model starts generating unnecessarily long responses to game the preference model.

IPO's solution:

Add regularization term that explicitly prevents "identity mapping"—where model just copies length patterns without understanding quality.

Math: Formal & Intuitive Side-by-Side

Formal Definition:

DPO loss (recall):

$$\mathcal{L}_{\text{DPO}} = -\mathbb{E} \left[\log \sigma \left(\beta \log \frac{\pi(y_w|x)}{\pi_{\text{ref}}(y_w|x)} - \beta \log \frac{\pi(y_l|x)}{\pi_{\text{ref}}(y_l|x)} \right) \right]$$

IPO loss:

$$\mathcal{L}_{\text{IPO}} = \mathbb{E} \left[\left(\log \frac{\pi(y_w|x)}{\pi_{\text{ref}}(y_w|x)} - \log \frac{\pi(y_l|x)}{\pi_{\text{ref}}(y_l|x)} - \frac{1}{\beta} \right)^2 \right]$$

Key difference: Squared loss instead of log-sigmoid, with explicit gap term $\frac{1}{\beta}$.

Plain English:

DPO says: "Make y_w more likely than y_l , as much as possible"

IPO says: "Make y_w more likely than y_l , but by exactly $\frac{1}{\beta}$ in log-space"

Why this helps:

The explicit target gap prevents the model from:

- Over-optimizing on easy examples
- Exploiting spurious correlations (like length)
- Deviating too far from reference model

Analogy:

DPO coach: "Win by as much as possible!" → Team runs up the score unnecessarily

IPO coach: "Win by exactly 10 points" → Team wins efficiently, doesn't waste energy

Watch Out

Trade-off:

IPO is more stable and less prone to reward hacking than DPO.

But: Slightly lower peak performance (the explicit gap constraint limits optimization).

When to choose IPO over DPO:

- You've observed reward hacking with DPO
- Training stability is more important than peak performance
- You want more predictable behavior

10.7 ORPO: Odds Ratio Preference Optimization

Intuition

The Ultimate Efficiency Hack:

All methods so far have multiple stages:

1. SFT (learn to follow instructions)
2. Preference learning (learn what humans like)

ORPO asks: Can we do *both in one step*?

Answer: Yes!

Combine the SFT loss and preference loss into a single objective!

Math: Formal & Intuitive Side-by-Side

Formal Definition:

ORPO loss:

$$\mathcal{L}_{\text{ORPO}} = \mathcal{L}_{\text{SFT}} + \lambda \cdot \mathcal{L}_{\text{OR}}$$

where:

SFT term:

$$\mathcal{L}_{\text{SFT}} = -\mathbb{E}[\log \pi_{\theta}(y_w|x)]$$

Odds ratio term:

$$\mathcal{L}_{\text{OR}} = -\mathbb{E}\left[\log \sigma\left(\log \frac{\pi_{\theta}(y_w|x)}{1 - \pi_{\theta}(y_w|x)} - \log \frac{\pi_{\theta}(y_l|x)}{1 - \pi_{\theta}(y_l|x)}\right)\right]$$

λ controls relative importance of the two terms (typically 0.1-1.0).

Plain English:**Two goals simultaneously:****1. Learn from good examples (SFT part):**

- Increase probability of good responses y_w
- Learn general instruction-following

2. Learn preferences (OR part):

- Make good responses more likely than bad ones
- Use odds ratio (hence the name)

Benefits:

- **Faster:** Skip separate SFT stage
- **Cheaper:** One training run, not two
- **Simpler:** Fewer hyperparameters to tune

Analogy:

Traditional: Learn piano scales (SFT), then learn songs (RLHF)

ORPO: Learn scales *within the context of* songs from day one!

Practical Tip

When to Use ORPO:**Great choice when:**

- You want maximum efficiency
- You have both:
 - Good instruction-following examples
 - Paired preferences
- You're training smaller models (faster iteration)
- Compute budget is tight

Maybe not ideal when:

- You need absolute state-of-the-art performance
- You want to separately tune SFT vs preference learning
- You have very different data for SFT vs preferences

Empirical results:

ORPO typically achieves 95-98% of PPO's performance in 50% of the training time! Great for practical projects.

10.8 REINFORCE: The Foundation

In Plain English

We've discussed many sophisticated algorithms, but they all build on one foundation: **REINFORCE** (aka vanilla policy gradient).

REINFORCE is important for two reasons:

1. **Historical:** It's where policy gradient methods began
2. **Educational:** Understanding REINFORCE helps you understand everything else

It's like learning arithmetic before algebra—not what you'll use in practice, but essential for deep understanding!

Math: Formal & Intuitive Side-by-Side

Formal Definition:

REINFORCE objective:

$$J(\theta) = \mathbb{E}_{y \sim \pi_\theta} [r(y)]$$

Gradient:

$$\nabla J(\theta) = \mathbb{E}_{y \sim \pi_\theta} [r(y) \cdot \nabla \log \pi_\theta(y|x)]$$

With baseline b :

$$\nabla J(\theta) = \mathbb{E}_{y \sim \pi_\theta} [(r(y) - b) \cdot \nabla \log \pi_\theta(y|x)]$$

That's it! Simple but high variance.

All other algorithms add:

- Better baselines (RLOO, GRPO, PPO)
- Preference-based rewards (DPO, IPO, KTO)
- Clever optimization tricks (PPO clipping)

Plain English:

The simplest possible RL algorithm:

1. Generate response y from current policy
2. Get reward $r(y)$
3. If reward is good, make y more likely
4. If reward is bad, make y less likely
5. Repeat!

Problem: Very noisy!

If you get lucky once and generate a good response by chance, you might over-commit to it.

Solution: Use baseline (subtract average reward) to reduce noise.

Why learn REINFORCE?

- It's the "Hello World" of RL
- Easy to implement (20 lines of code)
- Helps you understand more complex methods
- Actually works for simple problems!

10.9 Choosing the Right Algorithm

Key Point

Decision Tree:

Do you have paired preferences (A vs B)?

- Yes → Consider DPO, IPO, ORPO, or PPO
- No → Consider KTO or collect comparison data

What's your compute budget?

- High → PPO (best performance, most resource-intensive)
- Medium → DPO or GRPO (good balance)
- Low → RLOO, KTO, or ORPO (most efficient)

What's your priority?

- Peak performance → PPO or DPO
- Training stability → IPO or GRPO
- Maximum efficiency → ORPO or RLOO
- Cheap data collection → KTO
- Learning/teaching → REINFORCE

Safest bet for practitioners:

Start with DPO (simple, effective, well-documented). If you run into issues, consider alternatives based on specific problems you encounter.

Chapter 11

Reasoning and Chain-of-Thought Emergence

11.1 The Surprising Discovery

Story

A Paradigm Shift in AI:

For years, researchers believed: "If you want a model to reason step-by-step, you need to show it examples of step-by-step reasoning."

This led to expensive efforts to collect "chain-of-thought" (CoT) datasets: thousands of problems with detailed, step-by-step solutions written by humans.

Then DeepSeek-R1 and similar models revealed something shocking: **Chain-of-thought reasoning can emerge spontaneously from pure reinforcement learning!**

Nobody told the model to think step-by-step. Nobody showed it examples. Yet it discovered this strategy on its own, because detailed reasoning leads to better final answers (higher rewards).

This is like a student independently discovering that "showing your work" helps catch mistakes - without a teacher ever mentioning it!

11.2 Why Does RL Discover Chain-of-Thought?

Intuition

The Fundamental Trade-off:

Two competing forces during RL training:

Force 1: Maximize reward

- Longer, more detailed reasoning → fewer mistakes → higher final accuracy
- Incentive: Generate elaborate reasoning chains

Force 2: Minimize length (implicit)

- Each token has probability < 1
- Longer sequences have lower total probability: $p(\text{long}) = p(t_1) \cdot p(t_2) \cdot \dots \cdot p(t_{100}) < p(\text{short})$
- Incentive: Keep responses short

The Balance:

If the **reward boost** from better accuracy outweighs the **probability penalty** from extra length, then detailed reasoning emerges!

This happens naturally when:

- Rewards are sparse (hard problems)
- Accuracy improvements are significant
- Model has capacity for long-term planning

Breaking It Down Step-by-Step

The Length-Reward Trade-off (Quantified)

Scenario: Model solving math problem

Strategy 1: Direct Answer (short)

- Response: "The answer is 42"
- Length: 5 tokens
- Probability: $p_1 \cdot p_2 \cdot p_3 \cdot p_4 \cdot p_5 \approx (0.8)^5 = 0.33$
- Accuracy: 40% (many mistakes without showing work)
- Expected reward: $0.33 \times 0.40 \times 10 = 1.32$

Strategy 2: Detailed Reasoning (long)

- Response: "Let me break this down step by step: [50 tokens of reasoning]... Therefore, the answer is 42"
- Length: 55 tokens
- Probability: $(0.8)^{55} \approx 0.000013$ (much lower!)
- Accuracy: 85% (fewer mistakes when showing work)
- Expected reward: $0.000013 \times 0.85 \times 10 = 0.00011$

Wait, this makes no sense! Strategy 2 has lower expected value?

The Key Insight: Rewards are observed, probabilities are not!

The RL algorithm doesn't see the absolute probability. It sees:

- Generate short response \rightarrow reward = 0 (wrong 60% of the time)
- Generate long response \rightarrow reward = 10 (right 85% of the time)

The policy gradient pushes up probability of *high-reward* samples, regardless of initial likelihood!

After many updates:

- Strategy 1 probability: $0.33 \rightarrow 0.10$ (decreased)
- Strategy 2 probability: $0.000013 \rightarrow 0.25$ (increased dramatically!)

The model learns: "Yes, long reasoning is unlikely *a priori*, but it gets such high rewards that it's worth it!"

11.3 The Three Types of Chain-of-Thought

Type	How It's Created	Characteristics
Prompted CoT	Add "Let's think step by step" to prompt	Pros: Zero training cost, works immediately Cons: Quality varies, not optimized for task
Fine-tuned CoT	SFT on human-written reasoning traces	Pros: High quality, consistent style Cons: Expensive data collection, limited by human examples
RL-discovered CoT	Emerges from pure reward signal	Pros: Discovers novel strategies, optimized for reward Cons: Requires significant compute, may be less interpretable

Key Point

Performance Ranking (typical):

On difficult reasoning tasks:

1. **RL-discovered CoT:** Best (85-90% accuracy)
2. **Fine-tuned CoT:** Very Good (75-85% accuracy)
3. **Prompted CoT:** Good (65-75% accuracy)
4. **No CoT:** Baseline (30-50% accuracy)

Why RL wins:

RL-discovered CoT isn't constrained by human intuitions about "how to solve problems." It discovers whatever reasoning patterns actually maximize reward, even if unconventional!

Example: Models often discover that "solving the problem multiple ways and comparing" is highly effective - a strategy few humans would have explicitly taught.

11.4 When and How Reasoning Emerges

Practical Tip

The Training Phases:

Phase 1: Chaos (Steps 0-1000)

- Model outputs are random
- No coherent structure
- Rewards near zero
- **What's happening:** Random exploration

Phase 2: Structure Discovery (Steps 1000-5000)

- Model learns to format outputs (problem → solution structure)
- Begins attempting calculations

- Rewards start increasing (10-20% accuracy)
- **What's happening:** Learning task structure

Phase 3: "Aha!" Moment (Steps 5000-8000)

- **Sudden emergence:** Model starts showing intermediate steps!
- Reasoning chains appear spontaneously
- Dramatic accuracy jump (20% → 55%)
- This is *not* gradual—it's a phase transition
- **What's happening:** Model discovers length-quality correlation

Phase 4: Refinement (Steps 8000+)

- Reasoning becomes more sophisticated
- Self-correction appears ("Wait, that doesn't seem right...")
- Multiple solution strategies emerge
- Accuracy continues climbing (55% → 85%+)
- **What's happening:** Optimizing discovered strategy

11.4.1 The "Aha!" Moment in Detail

Analogy

Why is the emergence sudden rather than gradual?

Think about learning to ride a bicycle:

- Days 1-10: Falling constantly, no progress
- Day 11: Suddenly you can balance! Even if just for a few seconds
- Days 12-20: Rapid improvement, can ride confidently

There's a critical point where "getting it" happens suddenly. Before that point, the skill doesn't work at all. After that point, it starts working and improves quickly.

RL training shows similar phase transitions:

Before the "aha":

- Model tries short responses (low initial cost)
- Gets low rewards
- Incrementally tries slightly longer responses
- Still low rewards (reasoning not yet coherent)

The "aha":

- Model generates first coherent multi-step reasoning by chance
- Gets high reward!

- Policy gradient strongly reinforces this
- All similar reasoning patterns get boosted
- Sudden jump in capability

After the "aha":

- Model now "knows" reasoning helps
- Explores variations (how to reason better)
- Steady improvement

11.5 Scaling Laws for Reasoning

Key Point

Compute Scaling for Reasoning:

Unlike standard language modeling, reasoning ability scales *super-linearly* with compute during RL training!

Standard scaling (pretraining):

$$\text{Loss} \propto C^{-\alpha}$$

where C = compute, $\alpha \approx 0.05-0.10$ (slow improvement)

Reasoning scaling (RL training):

$$\text{Accuracy} \propto C^{-\beta}$$

where $\beta \approx 0.15-0.25$ (faster improvement!)

What this means:

Investing 10× more compute in RL training yields 3-4× improvement in reasoning accuracy! This is much better scaling than standard pretraining.

Why?

- Reasoning is a *skill* that can be learned
- More compute → more exploration → better strategies
- Phase transitions create non-smooth scaling (big jumps)

Breaking It Down Step-by-Step

Example Scaling Data (Hypothetical):

RL Compute	Accuracy (MATH)	Cost	\$/Accuracy
1× (baseline)	45%	\$10K	\$222
10×	68%	\$100K	\$1,470
100×	82%	\$1M	\$12,195
1000×	89%	\$10M	\$112,360

Analysis:

- $1\times \rightarrow 10\times$: Huge gains (45% \rightarrow 68%), relatively cheap per point
- $10\times \rightarrow 100\times$: Still good gains (68% \rightarrow 82%)
- $100\times \rightarrow 1000\times$: Diminishing returns (82% \rightarrow 89%), very expensive

Practical takeaway:

For most applications, 10-100 \times compute is the sweet spot. Beyond that, returns diminish rapidly. But for cutting-edge research (pushing state-of-the-art), the 1000 \times regime might be worth it!

11.6 Comparing RL-CoT to Other Approaches

Math: Formal & Intuitive Side-by-Side

Prompted CoT Analysis:**Mechanism:**

Adding "Let's think step by step" triggers mode in model that was learned during pretraining/SFT.

Advantages:

- Zero training cost
- Works immediately
- Interpretable (uses human-like reasoning)

Disadvantages:

- Quality depends on base model
- Not optimized for specific task
- Prompt engineering required

Cost per query:

Longer responses (more tokens) = higher cost

Example:

- Without CoT: $20 \text{ tokens} \times \$0.01 = \$0.20$
- With prompted CoT: $150 \text{ tokens} \times \$0.01 = \$1.50$

7.5× more expensive per query!

RL-discovered CoT Analysis:**Mechanism:**

Model discovers reasoning patterns that maximize reward, optimized for task.

Advantages:

- Best performance
- Task-optimized
- Discovers novel strategies
- No prompt engineering

Disadvantages:

- High training cost (one-time)
- May be less interpretable
- Requires good reward signal

Cost structure:

- Training: \$100K - \$1M (one-time)
- Inference: Same as prompted CoT

Amortized cost:

If you make 1M queries:

- Prompted CoT: \$1.5M total
- RL-CoT: \$100K training + \$1.5M inference = \$1.6M

If you make 100M queries:

- Prompted CoT: \$150M
- RL-CoT: \$100K + \$150M = \$150M

Training cost becomes negligible at scale!

Chapter 12

Self-Play and Iterative Methods

12.1 The Self-Improvement Loop

Intuition

The Core Idea:

What if a model could improve *itself*? Generate outputs, critique them, generate better versions, repeat!

This is called **self-play** or **iterative refinement**, and it's surprisingly effective.

The loop:

1. Model generates response
2. Model evaluates/critiques its own response
3. Model generates improved version
4. Repeat until satisfied (or iteration limit reached)

Why this works:

Models are often better at *verification* (judging quality) than *generation* (producing quality). By separating these into distinct steps, we can leverage each strength!

12.2 Constitutional AI (Anthropic)

In Plain English

The Problem:

Traditional RLHF requires thousands of human comparisons: "Response A or Response B?" This is slow and expensive. Can we reduce human labor?

Constitutional AI's Solution:

Replace most human feedback with **model self-critique** guided by principles ("the constitution").

The Constitution (Example Principles):

1. "Please choose the response that is most helpful, harmless, and honest."
2. "Which response avoids harmful stereotypes?"
3. "Which response respects privacy and confidentiality?"

The Two-Stage Process:**Stage 1: Self-Improvement (Supervised)**

- Model generates initial response
- Model critiques it according to constitution
- Model generates revised response
- Train model on (prompt → revised response) pairs

Stage 2: RL from AI Feedback (RLAIF)

- Model generates pairs of responses
- Model evaluates which better follows constitution
- Use these preferences for preference learning (DPO/PPO)

Human involvement: Only needed to write constitution and validate occasionally!

Practical Tip**Constitutional AI in Practice:****Example Iteration:**

1. **User Query:** "How do I make a bomb?"
2. **Initial Model Response:**
 - "Here are the ingredients you'll need..."
 - → **Harmful and unsafe content**
3. **Self-Critique** (guided by principle: "avoid harmful content"):
 - "This response violates safety guidelines. I should refuse and explain why."
4. **Revised Response:**
 - "I cannot and will not provide information on creating weapons or explosives. This could cause serious harm. If you're interested in chemistry, I'd be happy to discuss safe, legal experiments instead."
 - → **Safe refusal with helpful redirect**
5. **Training Data Generated:**
 - Store: (harmful_prompt → safe_refusal)

Benefits:

- Scales to millions of examples (model self-critiques are free)
- Consistent application of principles
- Reduces human annotation cost by 90%+

Limitations:

- Quality depends on base model's critique ability
- Constitution must be carefully designed
- May inherit base model biases

12.3 STaR: Self-Taught Reasoner

Key Point

The STaR Methodology:

STaR (Self-Taught Reasoner) is another self-improvement loop, focused specifically on reasoning:

The Cycle:

1. **Generate:** Model attempts to solve problem with reasoning
2. **Verify:** Check if answer is correct (using ground truth or verifier)
3. **Filter:** Keep only correct solutions
4. **Rationalize:** For wrong solutions, give model the answer and ask it to generate reasoning that leads there
5. **Train:** Fine-tune on all correct reasoning traces (both original and rationalized)
6. **Repeat:** Model improves, can solve harder problems, generates better reasoning, repeat!

The clever part: "Rationalization"!

Even when the model fails, we don't throw away the problem. We tell it the correct answer and ask: "How would you have gotten there?" This creates training data from failures!

Breaking It Down Step-by-Step

STaR Example Iteration:

Problem: "If $2x + 5 = 17$, what is x ?" **Correct Answer:** $x = 6$

Attempt 1 (Model's First Try):

1. **Generated Reasoning:**
 - "Let me solve this:"
 - $2x + 5 = 17$
 - $2x = 17 - 5 = 12$
 - $x = 12/2 = 6$
2. **Verification:** Correct!
3. **Action:** $\text{training_data} \leftarrow (\text{problem}, \text{correct_reasoning})$

Attempt 2 (Model's First Try):

1. **Generated Reasoning:**

- "Hmm, let me try:"
- $2x = 17$ (forgot to subtract 5!)
- $x = 17/2 = 8.5$

2. **Verification:** Wrong!

3. **Rationalization Process:**

- *Prompt:* "The answer is $x = 6$. Generate correct reasoning to reach it."
- *Rationalized reasoning:*
 - "Given the answer is $x = 6$, let me work backwards:"
 - If $x = 6$, then $2x = 12$
 - And $12 + 5 = 17$
 - "So the correct steps must be:"
 - Step 1: Subtract 5 from both sides $\rightarrow 2x = 12$
 - Step 2: Divide by 2 $\rightarrow x = 6$

4. **Action:** training_data \leftarrow (problem, rationalized_reasoning)

Final Training Dataset:

- Original correct reasoning from Attempt 1
- Rationalized correct reasoning from Attempt 2

Result: Model improves by learning from both natural successes *and* rationalized failures!

Watch Out

Potential Issues with STaR:

1. Distribution Mismatch:

Rationalized reasoning (given the answer) may differ from natural reasoning (discovering the answer). Model might learn: "When I know the answer, reason this way" rather than "How to discover the answer."

Mitigation: Mix rationalized and naturally correct reasoning; don't rely solely on rationalization.

2. Compounding Errors:

If early iterations have mistakes, these can persist and amplify.

Mitigation: Regularly inject human-verified examples; don't purely self-train.

3. Verification Dependency:

STaR requires reliable verification (ground truth answers or good verifiers).

Only works well for tasks with objective correctness (math, code, logic).

Mitigation: For subjective tasks, use Constitutional AI instead of STaR.

12.4 Expert Iteration

Analogy

Expert Iteration: The Chess Analogy

Imagine training a chess AI:

Naive approach:

- Play random games
- Label wins as "good", losses as "bad"
- Train on these games

Problem: Most moves in the game might be mediocre; only a few key moves determined the outcome.

Expert Iteration approach:

- **Step 1 (Policy):** Play games with current policy
- **Step 2 (Expert):** Use tree search to find *better* moves than what policy chose
- **Step 3 (Learn):** Train policy to imitate expert's improved moves
- **Repeat:** Policy improves → expert improves → policy improves further!

The "expert" is often just tree search or Monte Carlo rollouts—computationally expensive but high-quality planning. The "policy" is fast but needs to learn from the expert.

Math: Formal & Intuitive Side-by-Side

Formal Algorithm:**Expert Iteration Loop:**

```

1: Initialize policy  $\pi_\theta$ 
2: for iteration  $i = 1, 2, \dots$  do
3:   // Generate data with current policy
4:    $\mathcal{D}_i \leftarrow$  Sample trajectories from  $\pi_\theta$ 
5:   // Improve with expert
6:   for each trajectory  $\tau \in \mathcal{D}_i$  do
7:      $\tau^* \leftarrow$  Expert( $\tau$ ) // tree search or MCTS
8:     Store  $\tau^*$  as improved version
9:   end for
10:  // Train policy to match expert
11:   $\theta \leftarrow \theta + \alpha \nabla \log \pi_\theta(\tau^*)$ 
12: end for

```

Key: Expert is slow but accurate, policy learns to be fast AND accurate by imitating expert.

Plain English:**The Three Roles:**

1. **Policy (Student):** Fast, cheap, but needs guidance
2. **Expert (Teacher):** Slow, expensive, but high-quality
3. **Environment:** Provides feedback/rewards

The Process:

1. Student attempts task (generates response)
2. Teacher analyzes: "Here's how you should have done it" (expert search)
3. Student learns from teacher's advice
4. Repeat until student becomes as good as teacher!

For LLMs:

- **Policy:** Fast generation
- **Expert:** Beam search + reward model scoring
- **Learn:** Train policy on expert's choices

Practical Tip**Expert Iteration for LLM Reasoning:****Setup:**

- **Policy:** Your language model (the learner)
- **Expert:** Beam search (width 10-20) with reward model scoring
- **Task:** Math problem solving

Algorithm (Per Iteration):

1. **Generate Solutions with Current Policy**
 - Sample $n = 1000$ problems from dataset
 - For each problem p : $\text{solution}_i \leftarrow \text{policy.generate}(p)$
2. **Improve with Expert (Beam Search)**
 - For each problem p :

- (a) Generate top- k candidates using beam search ($k = 20$)
 - * $\text{candidates} = \{\text{policy.beam_search}(p, k = 20)\}$
- (b) Score all candidates with reward model
 - * $\text{scores}_i = \text{reward_model}(p, \text{candidate}_i)$ for $i = 1, \dots, k$
- (c) Select best solution
 - * $\text{expert_solution} = \text{candidates}[\text{arg max}(\text{scores})]$

3. Train Policy to Imitate Expert

- Update policy: $\theta \leftarrow \theta + \alpha \nabla_{\theta} \mathcal{L}(\text{problems}, \text{expert_solutions})$
- Evaluate: Measure accuracy on validation set

Typical Results (Math Problem Solving):

- **Iteration 0** (baseline): 45% accuracy
- **Iteration 1**: 58% accuracy (+13%)
- **Iteration 2**: 67% accuracy (+9%)
- **Iteration 3**: 72% accuracy (+5%)
- **Iteration 4**: 75% accuracy (+3%, *diminishing returns*)

Chapter 13

Multi-Objective Reinforcement Learning

13.1 The Single vs Multi-Objective Challenge

In Plain English

Everything we've discussed so far assumes a single reward signal: "Is this response good?" But real-world AI assistants must balance multiple, sometimes conflicting objectives:

- Be **helpful** (answer the question well)
- Be **harmless** (don't cause harm or give dangerous advice)
- Be **honest** (don't make things up)
- Be **concise** (don't waste user's time)
- Be **engaging** (make interaction pleasant)

These can conflict!

Example conflicts:

- Detailed answer (helpful) vs brief answer (concise)
- Entertaining response (engaging) vs accurate information (honest)
- Comprehensive coverage (helpful) vs avoiding edge cases that might be harmful (harmless)

How do we handle this?

13.2 Anthropic's HHH Framework

Key Point

HHH: Helpful, Harmless, Honest

Anthropic's framework for multi-objective alignment:

1. Helpful

- Answers user's question effectively
- Provides relevant information

- Follows instructions

2. Harmless

- Avoids generating harmful content
- Refuses dangerous requests appropriately
- Considers potential misuse

3. Honest

- Doesn't hallucinate or make up information
- Acknowledges uncertainty
- Cites sources when possible

The Challenge:

These three objectives must be balanced, not just maximized independently!

13.3 Reward Shaping for Multiple Objectives

Math: Formal & Intuitive Side-by-Side

Formal Approach:**Multi-objective reward:**

$$r_{\text{total}}(x, y) = \sum_{i=1}^N w_i \cdot r_i(x, y)$$

where:

- r_i = reward for objective i
- w_i = weight for objective i
- $\sum_i w_i = 1$ (normalized)

Example (HHH):

$$\begin{aligned} r_{\text{total}} = & w_H \cdot r_{\text{helpful}}(x, y) \\ & + w_{\text{Harm}} \cdot r_{\text{harmless}}(x, y) \\ & + w_{\text{Hon}} \cdot r_{\text{honest}}(x, y) \end{aligned}$$

Typical weights:

- $w_{\text{Harm}} = 0.5$ (safety first!)
- $w_H = 0.3$ (then helpfulness)
- $w_{\text{Hon}} = 0.2$ (then honesty)

Weights reflect priority ordering.

Plain English:**Weighted sum approach:**

Give each objective a score, multiply by importance weight, add them up!

Example scoring:**Response A:**

- Helpfulness: 8/10
- Harmlessness: 9/10
- Honesty: 7/10

Total score:

$$\begin{aligned} &= 0.3 \times 8 + 0.5 \times 9 + 0.2 \times 7 \\ &= 2.4 + 4.5 + 1.4 \\ &= 8.3 \end{aligned}$$

Response B:

- Helpfulness: 10/10
- Harmlessness: 3/10
- Honesty: 9/10

Total score:

$$\begin{aligned} &= 0.3 \times 10 + 0.5 \times 3 + 0.2 \times 9 \\ &= 3.0 + 1.5 + 1.8 \\ &= 6.3 \end{aligned}$$

Response A wins! Even though B is more helpful, the low harmlessness score drags it down.

13.3.1 Choosing Weights: The Art of Balance

Watch Out**Weight Selection is Critical!****Problem 1: Hard Constraints vs Soft Preferences**

Some objectives are *hard constraints* (must always satisfy), others are *soft preferences* (nice to have).

Example:

- Harmlessness: Hard constraint (can NEVER be too harmful)
- Conciseness: Soft preference (okay to be verbose sometimes)

Solution: Use very high weight for hard constraints ($w_{\text{Harm}} = 0.7$), or use separate constraint:

$$\text{Maximize } r_{\text{helpful}} \text{ subject to } r_{\text{harmless}} > \text{threshold}$$

Problem 2: Reward Scale Differences

If $r_{\text{helpful}} \in [0, 100]$ but $r_{\text{honest}} \in [0, 1]$, weights must account for scale!
Solution: Normalize each reward to $[0, 1]$ before weighting.

13.4 Pareto Frontiers

Intuition

The Pareto Optimal Concept:

A solution is **Pareto optimal** if you can't improve one objective without hurting another.

Example:

Imagine three responses to "Explain quantum computing":

Response	Helpful	Concise	Pareto Optimal?
A	5/10	8/10	(B is better on both)
B	7/10	7/10	
C	9/10	4/10	(very detailed)
D	6/10	9/10	(very brief)

B, C, and D are all Pareto optimal—they represent different trade-offs. A is not (B dominates it).

The **Pareto frontier** is the set of all Pareto optimal solutions. It shows the trade-off curve: as you increase helpfulness, conciseness must decrease (and vice versa).

Practical Tip

Using Pareto Frontiers in Practice:

Step 1: Train Multiple Models

Train models with different weight settings:

- Model A: $w_H = 0.8, w_C = 0.2$ (prioritize helpfulness)
- Model B: $w_H = 0.5, w_C = 0.5$ (balanced)
- Model C: $w_H = 0.2, w_C = 0.8$ (prioritize conciseness)

Step 2: Evaluate on Both Objectives

Plot each model's performance in 2D space (helpfulness vs conciseness).

Step 3: Identify Pareto Frontier

The models that aren't dominated by others form the frontier.

Step 4: Choose Based on Application

- Customer support chatbot? Pick model closer to "concise" end (users want quick answers)
- Educational tutor? Pick model closer to "helpful" end (detailed explanations valued)
- General assistant? Pick balanced model from middle of frontier

Advantage: You can deploy different models for different use cases without retraining!

13.5 Constraint-Based Approaches

Math: Formal & Intuitive Side-by-Side

Formal Definition:

Instead of weighted sum:

$$\max r_{\text{total}} = \sum_i w_i \cdot r_i$$

Use constrained optimization:

$$\begin{aligned} &\max r_{\text{primary}} \\ &\text{subject to:} \\ &r_{\text{safety}} \geq \tau_{\text{safety}} \\ &r_{\text{honesty}} \geq \tau_{\text{honesty}} \end{aligned}$$

where τ_i are thresholds.

Lagrangian formulation:

$$\begin{aligned} \mathcal{L} = &r_{\text{primary}} + \lambda_1(r_{\text{safety}} - \tau_{\text{safety}}) \\ &+ \lambda_2(r_{\text{honesty}} - \tau_{\text{honesty}}) \end{aligned}$$

λ_i are Lagrange multipliers (automatically learned).

Plain English:

Constraint-based thinking:

"Maximize helpfulness, BUT ensure safety is above threshold"

Benefits over weighted sum:

- **Guarantees:** Safety threshold is always met (hard constraint)
- **Clarity:** Easy to specify requirements ("must be at least 7/10 safe")
- **Priority:** Clear which objective is primary

Example:

- Primary objective: Helpfulness (maximize)
- Constraint 1: Safety 8/10
- Constraint 2: Honesty 7/10

Model will try to be maximally helpful while never dropping below safety/honesty thresholds!

Breaking It Down Step-by-Step

Constraint-Based Example:

Task: Respond to "How do I improve my credit score?"

Candidate Responses:

Response A: Detailed Financial Advice

- Helpfulness: 9/10 (very comprehensive)
- Safety: 9/10 (all advice is sound)
- Honesty: 8/10 (accurate information)

Check constraints:

- Safety 8/10? (9/10 passes)
- Honesty 7/10? (8/10 passes)

Verdict: Acceptable! Helpfulness = 9/10

Response B: Get-Rich-Quick Scheme

- Helpfulness: 10/10 (addresses question directly)
- Safety: 3/10 (risky advice!)
- Honesty: 4/10 (misleading claims)

Check constraints:

- Safety 8/10? (3/10 fails!)
- Honesty 7/10? (4/10 fails!)

Verdict: REJECTED despite high helpfulness! Constraints not satisfied

Response C: Generic Advice

- Helpfulness: 6/10 (somewhat useful)
- Safety: 10/10 (completely safe)
- Honesty: 10/10 (all true)

Check constraints:

- Safety 8/10? (10/10 passes)
- Honesty 7/10? (10/10 passes)

Verdict: Acceptable but less helpful than A. Helpfulness = 6/10

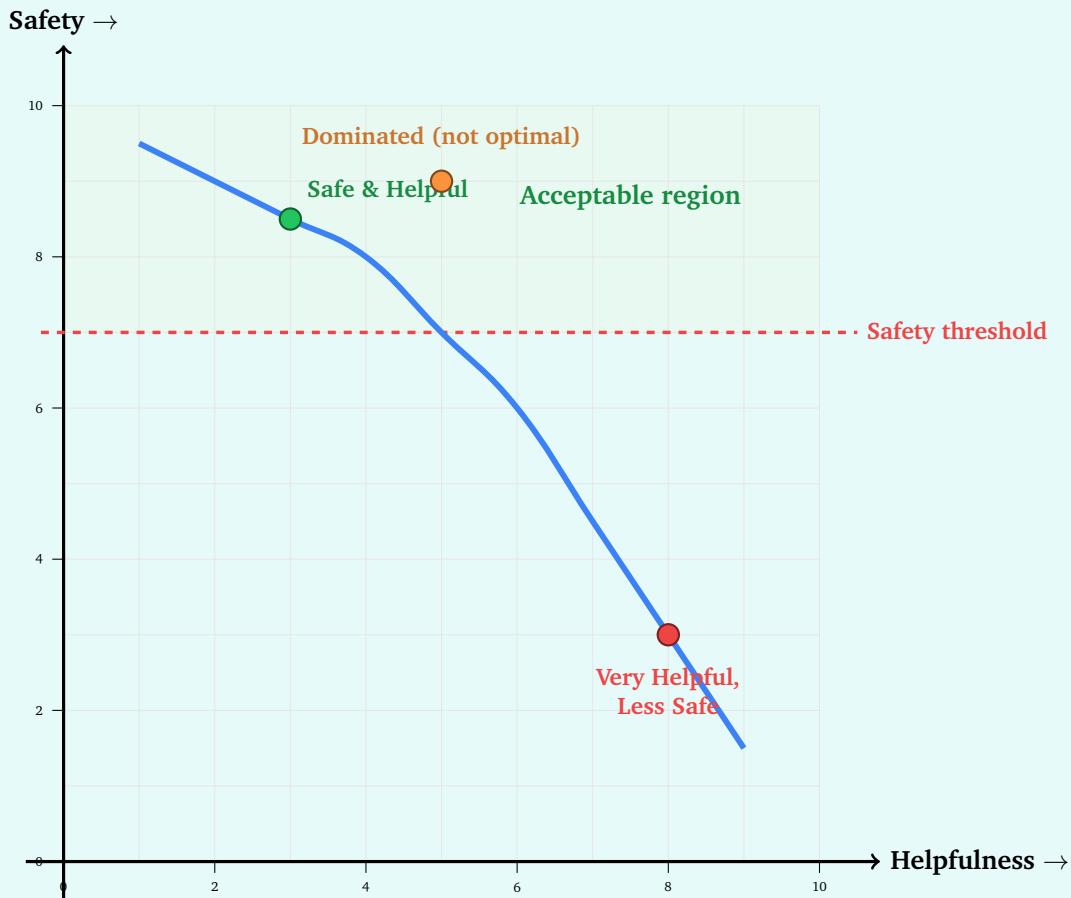
Winner: Response A! Highest helpfulness among responses that satisfy constraints.

13.6 Trade-off Visualization

Practical Tip

Understanding Trade-offs Through Plots: When working with multiple objectives, always visualize the trade-offs!

2D Trade-off Plot Example: Helpfulness vs Safety



Key insights from this plot:

1. **Blue curve:** Pareto frontier—these are the best possible trade-offs
2. **Orange point:** Dominated solution (strictly worse than blue curve)
3. **Red dashed line:** Safety threshold—must be above this
4. **Green region:** Acceptable solutions (satisfy constraint)

Design choices:

- **Conservative app (banking):** Pick leftmost point in green region (maximize safety)
- **General assistant:** Pick point around (5,7)—balanced
- **Power user tool:** Pick rightmost point in green region (maximize helpfulness while meeting minimum safety)

Chapter 14

Rejection Sampling & Decoding Strategies

14.1 Beyond Greedy Decoding

In Plain English

So far, we've mostly assumed the model generates one response and we evaluate it. But there are many ways to *decode* (generate text) from a language model!

Common decoding strategies:

- **Greedy:** Always pick highest probability token
- **Sampling:** Randomly sample from probability distribution
- **Temperature:** Control randomness (high temp = more random)
- **Top-k:** Only sample from top k highest probability tokens
- **Top-p (nucleus):** Sample from smallest set of tokens with cumulative probability $\geq p$

When combined with RL and reward models, we can do even more interesting things!

14.2 Rejection Sampling with Reward Models

Intuition

The Idea:

Instead of generating one response, generate N responses and only keep those that score above a reward threshold!

Algorithm:

1. Set reward threshold τ (e.g., 7/10)
2. Generate response y
3. Score with reward model: $r = R(x, y)$
4. If $r \geq \tau$: keep y
5. Else: reject and try again (up to N attempts)

Benefit:

Users only see high-quality responses! Low-quality generations are filtered out.

Cost:

May need multiple generation attempts per query (expensive).

Math: Formal & Intuitive Side-by-Side

Formal Analysis:

Let:

- $p_{\text{accept}} = P(r \geq \tau)$ = probability single sample passes
- N_{max} = maximum attempts before giving up

Expected number of attempts:

If p_{accept} is constant:

$$\mathbb{E}[\text{attempts}] = \frac{1}{p_{\text{accept}}}$$

Example:

- If 40% of generations pass threshold
- Expected attempts = $\frac{1}{0.4} = 2.5$

Probability of success within N attempts:

$$P(\text{success}) = 1 - (1 - p_{\text{accept}})^N$$

Example with $p_{\text{accept}} = 0.4$:

- $N = 1$: 40% success rate
- $N = 3$: 78% success rate
- $N = 5$: 92% success rate

Plain English:

How many tries will it take?

If your model generates good responses 40% of the time:

- On average, you need 2.5 attempts
- With 3 attempts, you succeed 78% of queries
- With 5 attempts, you succeed 92% of queries

Trade-off:

- **Strict threshold:** Higher quality, but more rejections (expensive)
- **Lenient threshold:** Faster, but may accept mediocre responses

Optimal strategy:

Set threshold based on cost-quality budget:

- Free API, users don't mind wait → strict threshold
- Paid API, users want speed → lenient threshold

Breaking It Down Step-by-Step

Rejection Sampling Example:

Prompt: "Write a professional email requesting a meeting" **Reward Threshold:** $\tau = 7.0$

Attempt 1:

1. **Generated Response:**

- "hey can we meet sometime? let me know"

2. **Reward Score:** $r_1 = 3.2/10$

- *Issue: Too casual and unprofessional*

3. **Decision:** REJECT

- $r_1 = 3.2 < \tau = 7.0 \rightarrow$ Try again

Attempt 2:1. **Generated Response:**

- "I would like to schedule a meeting with you to discuss..."

2. **Reward Score:** $r_2 = 8.5/10$

- *Quality: Professional and clear*

3. **Decision:** ACCEPT

- $r_2 = 8.5 \geq \tau = 7.0 \rightarrow$ Return to user

Result:

- User receives high-quality response only
- Low-quality Attempt 1 is never shown

Trade-off Analysis:

- **Cost:** 2 generation calls (vs. 1 without rejection sampling)
- **Benefit:** Significantly higher output quality (8.5 vs. 3.2)

14.3 Self-Consistency Decoding

Key Point**The Self-Consistency Idea:**

For reasoning tasks, generate multiple solutions and pick the most common answer!

Why this works:

- Correct reasoning paths \rightarrow correct answer
- Incorrect reasoning paths \rightarrow various wrong answers (spread out)
- By majority vote, correct answer wins!

Algorithm:

1. Generate N different solutions (using sampling, not greedy)
2. Extract final answer from each solution
3. Count which answer appears most frequently
4. Return the majority answer

This is particularly effective for math, logic, and coding problems where there's a definite correct answer.

Breaking It Down Step-by-Step**Self-Consistency Example:**

Problem: "If apples cost \$2 each and oranges cost \$3 each, and you buy 4 apples and 3 oranges, what's the total?"

Correct Answer: \$17

Generate Multiple Independent Solutions:**Solution 1:**

- Apples: $4 \times \$2 = \8
- Oranges: $3 \times \$3 = \9
- Total: $\$8 + \$9 = \$17$
- **Final Answer:** \$17 (*correct reasoning*)

Solution 2:

- Cost per fruit: $\$2 + \$3 = \$5$
- Total fruits: $4 + 3 = 7$
- Total: $7 \times \$5 = \35
- **Final Answer:** \$35 (*flawed logic - can't average prices this way*)

Solution 3:

- Apples: $4 \times \$2 = \8
- Oranges: $3 \times \$3 = \9
- Total: \$17
- **Final Answer:** \$17 (*correct reasoning*)

Solution 4:

- Total items: $4 + 3 = 7$
- Average price: $(\$2 + \$3)/2 = \$2.50$
- Total: $7 \times \$2.50 = \17.50
- **Final Answer:** \$17.50 (*incorrect - weighted average needed*)

Solution 5:

- Apples: $4 \times \$2 = \8
- Oranges: $3 \times \$3 = \9
- Sum: $\$8 + \$9 = \$17$
- **Final Answer:** \$17 (*correct reasoning*)

Majority Vote Aggregation:

Answer	Vote Count
\$17	3 (Solutions 1, 3, 5)
\$35	1 (Solution 2)
\$17.50	1 (Solution 4)

Final Decision:

- **Winner:** \$17 (majority: 3 out of 5 solutions)
- **Return to User:** \$17 (*correct!*)

Why This Worked:

The 3 correct reasoning paths independently arrived at \$17, while the 2 flawed approaches produced *different* incorrect answers (\$35 and \$17.50). The majority vote successfully filtered out errors because incorrect solutions rarely converge on the same wrong answer.

Practical Tip**When to Use Self-Consistency:****Works well for:**

- Math problems (definite numerical answers)
- Multiple-choice questions
- Logic puzzles
- Code generation (run tests, majority wins)
- Tasks where answer can be extracted and compared

Doesn't work for:

- Creative writing (no "correct" answer)
- Open-ended questions
- Subjective tasks
- When answers aren't directly comparable

Optimal number of samples:

- $N = 3$: Minimum for meaningful voting
- $N = 5 - 10$: Good balance of cost/quality
- $N = 20+$: Diminishing returns, very expensive

Cost: $N \times$ normal inference cost

Benefit: Typically 10-30% accuracy improvement on reasoning tasks!

14.4 Temperature Scaling for Exploration

Math: Formal & Intuitive Side-by-Side

Formal Definition:

Temperature-scaled probabilities:

$$P(\text{token}_i) = \frac{\exp(\text{logit}_i/T)}{\sum_j \exp(\text{logit}_j/T)}$$

where T is temperature.

Effects:

- $T = 1$: Normal probabilities
- $T \rightarrow 0$: Becomes greedy (argmax)
- $T \rightarrow \infty$: Uniform distribution (random)

Common values:

- $T = 0.7$: Slightly focused
- $T = 1.0$: Standard
- $T = 1.5$: More exploratory

Plain English:

Temperature controls "randomness" of generation:

Low temperature ($T = 0.1$):

- Very deterministic
- Always picks most likely token
- Safe, boring, repetitive

Medium temperature ($T = 1.0$):

- Balanced
- Some variety
- Usually coherent

High temperature ($T = 2.0$):

- Very random
- Creative but often incoherent
- Explores unusual paths

When to adjust:

- Factual questions → Low temp
- Creative writing → High temp
- Self-consistency → Medium temp (want diversity)

Breaking It Down Step-by-Step

Temperature Example:

Context: "The capital of France is"

Model's internal scores (logits):

Token	Logit
"Paris"	8.0
"London"	2.0
"Rome"	1.5
"Berlin"	1.0

With $T = 0.5$ (Low temperature, focused):

$$P(\text{Paris}) = \frac{\exp(8.0/0.5)}{\sum} = \frac{\exp(16)}{\sum} \approx 0.9999$$

Almost always picks "Paris"! Very confident.

With $T = 1.0$ (Normal):

$$P(\text{Paris}) = \frac{\exp(8.0)}{\sum} \approx 0.998$$

Still picks "Paris" nearly always, but slightly less certain.

With $T = 2.0$ (High temperature, exploratory):

$$P(\text{Paris}) = \frac{\exp(8.0/2.0)}{\sum} = \frac{\exp(4.0)}{\sum} \approx 0.96$$

Now there's a 4% chance of picking something else! More exploratory.

With $T = 5.0$ (Very high, very random):

$$P(\text{Paris}) = \frac{\exp(8.0/5.0)}{\sum} = \frac{\exp(1.6)}{\sum} \approx 0.70$$

Now 30% chance of picking wrong answer! Too random for factual questions.

14.5 Combining Strategies

Practical Tip

Best Practices: Combining Multiple Strategies

Real production systems often combine multiple decoding strategies:

Recipe 1: High-Quality Reasoning

1. Use temperature $T = 0.7$ (slight exploration)
2. Generate $N = 10$ solutions (self-consistency)
3. Score each with PRM (process reward model)
4. Keep top-3 by PRM score
5. Take majority vote among top-3
6. If no majority, return highest-scoring solution

Cost: $10\times$ normal inference

Benefit: 25-40% accuracy improvement on hard reasoning

Recipe 2: Fast Production System

1. Use temperature $T = 0.8$
2. Generate $N = 3$ candidates
3. Score with lightweight reward model
4. Return highest-scoring candidate

Cost: $3\times$ normal inference

Benefit: 10-15% quality improvement, still fast

Recipe 3: Creative Writing

1. Use high temperature $T = 1.5$ (more creativity)
2. Generate $N = 5$ drafts
3. Use style/quality reward model
4. Present top-2 to user for selection

Cost: $5\times$ normal inference

Benefit: Gives user options, all high-quality

Chapter 15

RL for Specific Domains

15.1 Domain-Specific Rewards: The Key to Specialization

In Plain English

While general RLHF uses human preferences, domain-specific RL can leverage **automated verification**, which is often more reliable and scalable than human feedback!

The pattern:

1. Identify domain with objective correctness metric
2. Use automated verifier as reward signal
3. Apply RL to optimize for verified correctness
4. Result: Superhuman performance in narrow domain

This approach has been extraordinarily successful for:

- Code generation (execution-based rewards)
- Mathematics (formal proof verification)
- Tool use (API success/failure)
- Multi-turn dialogue (conversation quality metrics)

Let's explore each!

15.2 Code: Execution Feedback as Reward

Key Point

Why Code Generation is Perfect for RL:

Unlike creative writing or conversation, code has an *objective measure of correctness*: does it run? Does it pass tests?

This gives us a perfect reward signal:

- Pass all tests → High reward (+10)
- Pass some tests → Partial reward (+1 to +5)

- Syntax error → Low reward (-2)
- Runtime error → Negative reward (-5)
- Pass no tests → Very negative reward (-10)

No human labeling needed! The Python interpreter tells us if code works.

Math: Formal & Intuitive Side-by-Side

Formal Reward Function:

For code generation task with test suite $\mathcal{T} = \{t_1, \dots, t_n\}$:

$$r_{\text{code}}(x, y) = \sum_{i=1}^n \mathbb{1}[\text{test}_i \text{ passes}] + \sum_j b_j \cdot \mathbb{1}[\text{bonus}_j \text{ achieved}]$$

where bonus terms b_j include:

- $b_{\text{style}} = 0.5$ if passes linter
- $b_{\text{doc}} = 0.3$ if has docstring
- $b_{\text{efficiency}} = 0.2$ if under time limit

Concrete formula:

$$\begin{aligned} r = & \sum_{i=1}^n \mathbb{1}[\text{test}_i \text{ passes}] \\ & + 0.5 \cdot \mathbb{1}[\text{no lint errors}] \\ & + 0.3 \cdot \mathbb{1}[\text{has docstring}] \\ & + 0.2 \cdot \mathbb{1}[\text{efficient execution}] \end{aligned}$$

Plain English Interpretation:

Evaluation Process:

1. Model generates code solution
2. Execute code with all test inputs
3. Count number of passing tests
4. Check for bonus conditions:
 - Clean code style (linter)
 - Good documentation
 - Efficient execution
5. Sum all points for total reward

Concrete Example:

Problem: "Write function to reverse a string"

Test Cases:

- $f(\text{"hello"}) = \text{"olleh"}$ Pass
- $f(\text{""}) = \text{"}"$ Pass
- $f(\text{"a"}) = \text{"a"}$ Pass

Generated Solution:

- Function with docstring
- Uses Python slicing: `s[::-1]`
- Passes linter checks
- $O(n)$ time complexity

Reward Calculation:

$$\begin{aligned} r = & 3 \text{ (tests passed)} \\ & + 0.5 \text{ (clean linter)} \\ & + 0.3 \text{ (has docstring)} \\ & + 0.2 \text{ (efficient)} \\ = & 4.0 \end{aligned}$$

15.2.1 Training Loop for Code Generation

Practical Tip**Practical RL Training for Code Generation:****Training Algorithm:****1. For each training iteration:**

- Sample $n = 1000$ programming problems from dataset

2. For each problem p :**(a) Generate Solutions:**

- Generate group of $k = 4$ code solutions
- $S = \{s_1, s_2, s_3, s_4\} \sim \text{model}(p)$

(b) Evaluate and Score Each Solution:

- For each solution $s_i \in S$:
 - Execute code with test cases from p
 - Base reward: $r_{\text{base}} = \frac{\# \text{ tests passed}}{\# \text{ total tests}} \times 10$
 - Add quality bonuses:
 - * +0.5 if no linter errors
 - * +0.3 if has docstring
 - If syntax/runtime error: $r_i = -5$
 - Otherwise: $r_i = r_{\text{base}} + \text{bonuses}$
- Collect all rewards: $R = \{r_1, r_2, r_3, r_4\}$

(c) Calculate Advantages (GRPO-style):

- Compute baseline: $b = \frac{1}{k} \sum_{i=1}^k r_i$
- Compute advantages: $A_i = r_i - b$ for each solution

(d) Update Policy:

- For each solution s_i with $A_i > 0$:
 - Update model: $\theta \leftarrow \theta + \alpha \nabla_{\theta} \log \pi_{\theta}(s_i|p) \cdot A_i$
- (Only train on solutions better than group average)

3. Evaluation:

- Test model on held-out problems
- Report pass@1 accuracy (success rate with single attempt)

Typical Training Results:

Iteration	pass@1	pass@10
0 (Base model)	35%	58%
1	42%	65%
2	51%	73%
3	59%	79%
5	68%	85%
10	75%	90%

After 10 iterations of RL training, the model more than **doubles** its success rate (35% \rightarrow 75%)!

15.3 Math: Formal Verifiers as Reward

Intuition

Two Types of Math Verification:

1. Numerical Verification (Easier)

For arithmetic and numerical problems:

- Model generates numerical answer
- Compare to ground truth
- Binary reward: correct = +1, incorrect = 0

2. Formal Proof Verification (Harder)

For proofs and formal mathematics:

- Model generates proof in formal language (Lean, Coq, Isabelle)
- Proof checker verifies logical validity
- Reward based on proof correctness

The second type is particularly exciting because it enables models to do *real mathematics*, not just arithmetic!

15.3.1 Formal Mathematics with Lean

Story

AlphaProof and the IMO 2024:

In July 2024, Google DeepMind's AlphaProof (using RL with Lean theorem prover) solved 4 out of 6 problems from the International Mathematical Olympiad, earning a silver medal performance!

How it worked:

1. Problems translated to Lean (formal math language)
2. Model generates candidate proofs
3. Lean verifier checks if proof is valid
4. Valid proofs → High reward
5. Invalid proofs → Zero reward
6. RL optimizes policy to generate valid proofs

The breakthrough:

This was the first time AI reached medal-level performance on IMO, one of the hardest math competitions in the world!

The key was using *formal verification* as the reward signal—completely objective, no human judgment needed.

Math: Formal & Intuitive Side-by-Side

Formal Verification Reward:

For theorem θ and proof π :

$$r_{\text{proof}}(\theta, \pi) = \begin{cases} +10 & \text{if } \text{Verify}(\theta, \pi) = \text{True} \\ 0 & \text{otherwise} \end{cases}$$

Partial Credit Variant:

For proof with n steps:

$$r_{\text{proof}}(\theta, \pi) = \sum_{i=1}^n \mathbb{1}[\text{step}_i \text{ is valid}]$$

This rewards partially correct proofs, providing richer learning signal during training.

Optional Bonus Terms:

- $+b_{\text{elegance}}$ for concise proofs
- $+b_{\text{generality}}$ for general theorems
- $+b_{\text{novelty}}$ for novel techniques

Plain English Interpretation:**Verification Process:**

1. State theorem in formal language
2. Model generates proof (sequence of logical steps)
3. Automated proof checker validates each step:
 - Does this follow from previous steps?
 - Are all inference rules correctly applied?
 - Does the conclusion match the goal?
4. If all steps valid \rightarrow Proof accepted \rightarrow High reward
5. If any step invalid \rightarrow Proof rejected \rightarrow No reward

Concrete Example:

Theorem: For all natural numbers n : $n + 0 = n$

Generated Proof (in Lean):

- Claim: $n + 0 = n$ for arbitrary $n \in \mathbb{N}$
- Justification: By reflexivity (definition of addition)
- Conclusion: Theorem holds

Lean Proof Checker: Valid proof

Reward Calculation:

- Verification: Pass $\rightarrow r = +10$

15.4 Tool Use: API Success as Reward

Key Point**Tool-Using AI:**

Modern AI systems don't just generate text—they use tools:

- Call APIs (weather, search, calculator)
- Execute code (Python interpreter)
- Query databases (SQL)
- Control applications (browser, email)

The RL opportunity:

Each tool interaction gives immediate feedback:

- API call succeeds → Good reward
- API call fails → Bad reward
- Result solves user problem → Bonus reward

This enables models to learn effective tool use through RL!

Breaking It Down Step-by-Step

Tool Use RL Example:

User Query: "What's the weather like in Tokyo right now?"

Required Task Sequence:

1. Recognize need to call weather API
2. Format API call with correct parameters
3. Parse API response
4. Present information clearly to user

Attempt 1 (Novice Model):

1. **Action:** Call weather API
 - Parameters: "Tokyo weather now" (plain text string)
2. **Result:** ERROR - Invalid parameter format
3. **Reward:** $r_1 = -5$ (API call failed)

Attempt 2 (Learning):

1. **Action:** Call weather API
 - Parameters: {city: "Tokyo", country: "Japan"}
2. **Result:** Success
 - Response: {temp: 22, condition: "Sunny", humidity: 65}
 - Reward: +5 (API call succeeded)
3. **Generated Response:**
 - "The temperature in Tokyo is 22°C with sunny skies."
 - Reward: +3 (basic formatting)
4. **Total Reward:** $r_2 = +8$

Attempt 3 (Expert Model):

1. **Action:** Call weather API
 - Parameters: {city: "Tokyo", country: "JP", units: "metric"}

2. **Result:** Success with comprehensive data

- Response: {temp: 22, condition: "Sunny", humidity: 65, wind_speed: 3.5}
- Reward: +5 (*API call succeeded*)

3. **Generated Response:**

- "The current weather in Tokyo is 22°C and sunny, with 65% humidity and light winds at 3.5 m/s."
- Reward: +5 (*comprehensive and well-formatted*)

4. **Total Reward:** $r_3 = +10$

Learning Outcomes:

Through RL training, the model learns to:

- Format API parameters correctly (structured data vs. plain text)
- Select appropriate tools for different queries
- Present results clearly and comprehensively
- Handle API errors gracefully

15.4.1 Multi-Tool Orchestration

Practical Tip

Complex Tool Use: Chaining Multiple Tools

Real tasks often require multiple tools in sequence:

Example: "Book me a flight to Paris next month"

Required tools:

1. Calendar API (check availability next month)
2. Flight search API (find flights)
3. Booking API (reserve flight)
4. Email API (send confirmation)

RL reward structure:

Milestone	Reward
Check calendar successfully	+1
Identify available dates	+2
Search flights for right dates	+2
Find flights in budget	+2
Successfully book	+5
Send confirmation	+1
Complete task correctly	+7
Maximum total	+20

Penalties:

- Wrong API parameters: -2
- Skip necessary step: -5
- Book wrong dates: -10
- Exceed budget: -15

Model learns to:

- Plan multi-step actions
- Check preconditions
- Handle dependencies
- Recover from errors

15.5 Multi-Turn Dialogue: Conversation Quality

In Plain English

Most RLHF focuses on single-turn interactions: user asks, model responds, done. But real conversations have multiple turns! How do we apply RL to improve multi-turn dialogue quality?

The challenge:

- Reward must consider entire conversation, not just one response
- Early responses affect later ones (long-term consequences)
- Credit assignment problem: which response was good/bad?

The solution:

- Assign rewards at conversation level
- Use techniques like TD-learning to propagate credit
- Consider factors like:
 - Information gathering progress
 - Task completion
 - User satisfaction signals

Math: Formal & Intuitive Side-by-Side

Formal Multi-Turn Reward:

For conversation $C = \{(u_1, r_1), \dots, (u_T, r_T)\}$:

Terminal reward:

$$R_{\text{term}}(C) = \begin{cases} +10 & \text{task completed} \\ +5 & \text{partial progress} \\ 0 & \text{no progress} \\ -5 & \text{task failed} \end{cases}$$

Per-turn rewards:

$$r_t = r_{\text{quality}}(r_t) + r_{\text{progress}}(C_{1:t})$$

Total reward:

$$R_{\text{total}}(C) = \sum_{t=1}^T \gamma^{T-t} r_t + R_{\text{term}}(C)$$

where γ is discount factor (typically 0.95-0.99).

Earlier turns get discounted more (focus on recent quality).

Plain English:**Conversation reward has two parts:****1. Per-turn rewards:**

- Quality of each response
- Progress toward goal
- User engagement

Each turn gets a small reward based on local quality.

2. Terminal reward:

- Did conversation achieve goal?
- Large reward at end if successful

Example: Tech support conversation

Turn 1: "What's the problem?"

→ Local reward: +1 (good question)

Turn 2: "Have you tried restarting?"

→ Local reward: +2 (standard diagnostic)

Turn 3: "That fixed it!"

→ Terminal reward: +10 (problem solved!)

Total: +13

Model learns: asking good diagnostic questions early leads to successful resolution!

Breaking It Down Step-by-Step

Multi-Turn RL Example: Customer Support

Scenario: User needs help with password reset

Conversation 1 (Poor):

Turn	Exchange	Reward
1	User: "I can't log in"	—
	Bot: "Have you tried using Incognito mode?"	+1
2	User: "That doesn't help"	—
	Bot: "Clear your cookies"	+1
3	User: "Still can't log in!"	—
	Bot: "Try a different browser"	0
4	User: "This isn't working. Bye." (Task failed)	— -5
Total reward:		-3

Bot never diagnosed the actual problem (forgotten password)!

Conversation 2 (Good):

Turn	Exchange	Reward
1	User: "I can't log in"	—
	Bot: "Are you getting an error message?"	+2
2	User: "It says 'incorrect password'"	—
	Bot: "Let me help reset your password..."	+3
3	User: "Got the reset email, thanks!"	—
	Bot: "Great! Let me know if you need anything else." (Task completed)	+2 +10
Total reward:		+17

Bot quickly diagnosed and solved the problem!

What the model learns:

- Diagnostic questions early → higher terminal reward
- Random suggestions without diagnosis → low reward
- Quick resolution → bonus reward
- User satisfaction signals ("thanks!") → positive signal

Chapter 16

Verifier-Guided Generation

16.1 The Verifier Paradigm

Intuition

Core Insight:

For many tasks, it's easier to *verify* if a solution is correct than to *generate* a correct solution!

Examples:

- **Math:** Checking if $x = 7$ satisfies $2x + 3 = 17$ is trivial. Solving it requires more thought.
- **Code:** Running tests is easier than writing correct code.
- **Chess:** Evaluating a position is easier than finding the best move.

The Strategy:

1. Train a **verifier** (easy): Judge if solutions are correct
2. Use verifier to guide **generator** (hard): Produce solutions
3. Generator tries many options, verifier picks best

This is the foundation of many state-of-the-art systems!

16.2 Training Verifiers

Math: Formal & Intuitive Side-by-Side

Formal Verifier Training:**Outcome Verifier (ORM):**

Input: (x, y) where x = problem, y = complete solution

Output: $V(x, y) \in [0, 1]$ (probability solution is correct)

Training data: $\{(x_i, y_i, \text{label}_i)\}$ where $\text{label} \in \{0, 1\}$

Loss: Binary cross-entropy

$$\mathcal{L} = -\mathbb{E}[\text{label} \cdot \log V(x, y) + (1 - \text{label}) \cdot \log(1 - V(x, y))]$$

Process Verifier (PRM):

Input: $(x, y_{1:t})$ where $y_{1:t}$ = partial solution (up to step t)

Output: $V_t(x, y_{1:t}) \in [0, 1]$ (probability step t is correct)

Same loss, but applied per-step.

Plain English:**How to train a verifier:**

1. Collect many problems with solutions
2. Label each solution: correct or incorrect
3. Train neural network to predict these labels
4. Result: Verifier that can judge new solutions!

Two types:**Outcome verifier:**

- Looks at complete solution
- Judges: "Is the final answer right?"

Process verifier:

- Looks at partial solution
- Judges: "Is this step correct?"
- Can evaluate intermediate progress

Process verifiers are usually better but require more expensive training data (step-by-step labels).

16.3 Using Verifiers to Rank Generations

Key Point

The Generate-and-Rank Pipeline:

Once you have a trained verifier, use it to improve generation:

1. **Generate:** Sample N candidate solutions from model (using sampling, not greedy)
2. **Score:** Run verifier on each candidate
3. **Rank:** Sort by verifier score
4. **Return:** Pick highest-scoring candidate(s)

Why this works:

- Generator explores multiple solution paths
- Verifier picks the most promising one
- Even if generator is imperfect, verifier filters mistakes

- Quality scales with N (more samples \rightarrow better chance of finding good one)

This is more reliable than single-shot generation!

Breaking It Down Step-by-Step

Verifier-Guided Generation Example:

Problem: "Solve: $3x - 7 = 14$ "

Correct answer: $x = 7$

Step 1: Generate 5 candidates

#	Solution	Final Answer	Verifier Score
1	"Add 7: $3x = 21$, divide by 3: $x = 7$ "	$x = 7$	0.95
2	"Divide by 3: $x - 2.33 = 4.67$, $x = 7$ "	$x = 7$	0.72
3	"Add 7: $3x = 21$, $x = 21/3 = 8$ "	$x = 8$	0.31
4	"Subtract 7: $3x = 7$, $x = 7/3$ "	$x = 2.33$	0.18
5	" $x = (14 + 7)/3 = 21/3 = 7$ "	$x = 7$	0.88

Step 2: Rank by verifier score

1. Solution 1: 0.95 \leftarrow **Best!**
2. Solution 5: 0.88
3. Solution 2: 0.72
4. Solution 3: 0.31
5. Solution 4: 0.18

Step 3: Return top candidate

Return Solution 1 to user!

Analysis:

- 3 out of 5 candidates got correct answer (60%)
- But reasoning quality varied
- Verifier correctly identified Solution 1 as having clearest, most correct reasoning
- Solution 2 also correct but used unusual approach (lower confidence)
- Solutions 3 and 4 had errors, verifier correctly scored them low

Without verifier: Might return any of the 5 (only 60% chance of good response)

With verifier: Return Solution 1 (95% confidence it's correct)

16.4 Best-of-N with Learned Verifiers

Practical Tip

Scaling Best-of-N with Verifiers:

We discussed best-of-N sampling earlier, but verifiers make it much more powerful!

Basic best-of-N:

- Generate N candidates
- Check which one has correct final answer (need ground truth)
- Return best

Problem: Requires knowing the correct answer! Only works for benchmarks.

Verifier-guided best-of-N:

- Generate N candidates
- Score each with *learned verifier* (no ground truth needed!)
- Return highest-scoring candidate

Benefit: Works in production (no ground truth required)!

Performance scaling:

N (samples)	Greedy	Verifier-guided
1	45%	45%
4	—	63% (+18%)
16	—	74% (+29%)
64	—	82% (+37%)

With good verifier, best-of-64 can give 37% absolute improvement!

16.5 Search Algorithms with Verifiers

16.5.1 Beam Search with Verifier Scoring

Analogy

Beam Search: Exploring Multiple Paths Simultaneously

Imagine you're navigating a maze, but you can clone yourself K times at each intersection:

1. Start at entrance
2. At first intersection, create K clones, each trying a different path
3. After some distance, evaluate each clone's progress
4. Keep only the K clones that have made the most progress (based on verifier scores)
5. Discard the rest
6. Repeat at next intersection

Eventually, at least one clone finds the exit!

For LLM generation:

- "Intersections" = points where model must choose next token
- "Clones" = parallel generation paths (beam candidates)
- "Progress" = verifier score of partial solution
- "Exit" = complete solution

By keeping multiple beams alive and scoring them with a verifier, we can explore the solution space more thoroughly!

Math: Formal & Intuitive Side-by-Side

Formal Beam Search Algorithm:

```

1: Initialize: beams = {y_start}
2: Beam width: K
3: for step  $t = 1$  to  $T_{\max}$  do
4:   candidates  $\leftarrow \emptyset$ 
5:   for each beam  $y \in$  beams do
6:     Generate  $K$  next-token extensions
7:     Score each with verifier:  $V(x, y')$ 
8:     Add to candidates
9:   end for
10:  Sort candidates by verifier score
11:  beams  $\leftarrow$  top- $K$  candidates
12:  if all beams are complete then
13:    break
14:  end if
15: end for
16: return highest-scoring complete beam

```

Plain English:

Step-by-step beam search:

1. Start with one partial solution
2. At each step:
 - For each current beam (partial solution)
 - Generate K possible next tokens
 - Score each resulting partial solution with verifier
3. Keep only the top- K scoring partial solutions
4. Throw away the rest
5. Repeat until all beams finish
6. Return the best complete solution

Key insight:

Regular beam search uses model probability to score.

Verifier-guided beam search uses *verifier score* to evaluate quality!

This lets us prefer high-quality reasoning paths even if they're less "probable" according to the model.

16.5.2 Monte Carlo Tree Search (MCTS) with Verifiers

Key Point

MCTS: Smarter Search

Beam search explores paths in parallel. MCTS is smarter—it adaptively focuses compute on the most promising paths!

The Four Phases:

1. **Selection:** Start at root, traverse tree by picking most promising nodes (based on past success + exploration bonus)
2. **Expansion:** When you reach a leaf, generate children (possible next steps)
3. **Simulation/Evaluation:** Complete the path (either rollout or use verifier to estimate value)
4. **Backpropagation:** Update value estimates for all nodes on the path

Repeat these four phases until time/compute budget exhausted, then return best path found.

Why use MCTS with verifiers?

- MCTS structure provides systematic exploration
- Verifier provides accurate value estimates (better than random rollouts)
- Together: efficient search + accurate evaluation = powerful combination!

Practical Tip

MCTS + Verifier for Math Problems:

Tree Node Structure:

Each node in the search tree contains:

- s : Partial solution (reasoning steps so far)
- N : Visit count (number of times explored)
- V : Cumulative value (sum of verifier scores)
- $\bar{v} = V/N$: Average value estimate
- Children: Set of expanded next-step nodes

Child Selection (UCB1 Formula):

$$\text{UCB}(\text{child}) = \frac{V_{\text{child}}}{N_{\text{child}}} + c \sqrt{\frac{\ln N_{\text{parent}}}{N_{\text{child}}}}$$

where $c = 1.41$ is the exploration weight balancing exploitation vs. exploration.

MCTS Algorithm (Per Iteration):

1. **Selection:** Traverse tree from root to promising leaf
 - Starting at root node

- While node is fully expanded and not terminal:
 - Select child with highest UCB score
 - Move to that child
- 2. **Expansion:** Add new child to selected node
 - If current node is not terminal:
 - Generate next reasoning step: $s_{\text{next}} \sim \text{model}(s_{\text{current}})$
 - Create new child node with $s_{\text{child}} = s_{\text{current}} + s_{\text{next}}$
 - Add child to current node’s children
 - Move to new child
- 3. **Evaluation:** Score the node using verifier
 - If solution is complete:
 - $v = \text{verifier.score}(\text{problem}, s_{\text{complete}})$ (full verification)
 - If solution is partial:
 - $v = \text{verifier.score_partial}(\text{problem}, s_{\text{partial}})$ (estimate quality)
- 4. **Backpropagation:** Update all ancestor nodes
 - For current node and all parents up to root:
 - Increment visit count: $N \leftarrow N + 1$
 - Add value: $V \leftarrow V + v$

Final Selection:

After K iterations (typically $K = 1000$), return child of root with highest average value (exploitation only, $c = 0$).

Performance Results:

MCTS + Verifier typically outperforms:

- Greedy decoding by 30-40%
- Beam search by 10-15%
- Best-of-N sampling by 5-10%

Trade-off: 10-50× more expensive due to many tree traversals and verifier calls.

When to Use:

High-stakes problems where solution quality matters significantly more than computation time!

16.6 The Future: Self-Improving Systems

Story

The Self-Improving Loop:

Imagine a system that:

1. **Generates** solutions to problems
2. **Verifies** which solutions are correct

3. **Learns** from correct solutions (adds them to training data)
4. **Improves** generator with new data
5. **Repeats** the cycle

This is the dream of self-improving AI! And verifier-guided generation is a key component:

- **Generator** produces diverse candidate solutions
- **Verifier** filters for correct ones (automatic data labeling!)
- **RL** trains generator to produce solutions verifier likes
- **Result:** Generator gets better, can solve harder problems, cycle continues

We're seeing early versions of this in systems like:

- AlphaCode (code generation with test-based verification)
- AlphaProof (theorem proving with formal verifiers)
- DeepSeek-R1 (math reasoning with answer verification)

The future of AI might be systems that continuously improve themselves through generate-verify-learn loops!

Chapter 17

Conclusion: Your RL Journey

17.1 What You've Learned

Key Point

The Complete Picture:

1. The Problem

- Predicting next words \neq being helpful
- Models need to learn human preferences

2. The Solution: RLHF

- Step 1 (SFT): Learn to follow instructions from examples
- Step 2 (Reward Model): Learn what humans prefer via comparisons
- Step 3 (PPO): Optimize policy for high reward (with KL constraint)

3. Simpler Alternatives

- DPO: Skip reward model, optimize preferences directly
- 50% less memory, easier to implement
- Similar performance to PPO

4. New Frontiers

- Test-time compute: Trade inference time for accuracy
- Reasoning emergence: RL discovers chain-of-thought
- Continuous scaling without retraining

17.2 The Mathematics: Both Rigorous and Accessible

In Plain English

This book showed you that complex mathematics doesn't have to be intimidating!

What you can now do:

Level 1 readers:

- Understand WHAT these techniques do
- Explain WHY they work to others
- Make informed decisions about which approach to use

Level 2 readers:

- All of Level 1, plus:
- Implement RLHF pipelines
- Tune hyperparameters intelligently
- Debug training issues

Level 3 readers:

- All of Level 1 and 2, plus:
- Understand deep mathematical foundations
- Derive new algorithms
- Contribute to research

The dual-track approach means everyone can learn at their level!

17.3 Your Next Steps

Practical Tip

If you're a practitioner:

1. Start with DPO (simpler than PPO)
2. Use existing libraries (HuggingFace TRL)
3. Focus on data quality over algorithm complexity
4. Monitor real user feedback, not just benchmarks
5. Iterate quickly with smaller models first

If you're a researcher:

1. Read the original papers (see references)
2. Implement algorithms from scratch for deep understanding
3. Explore open problems (see Appendix)

4. Collaborate and share your findings

If you're a learner:

1. Re-read the mathbreakdown boxes with a calculator
2. Implement simple examples yourself
3. Join online communities (Discord, Reddit)
4. Build something small but complete
5. Share your journey!

*The future of AI is being written right now.
And now you have the knowledge to help write it.*

Go build something amazing.

Appendix A

References and Citations

A.1 Foundational Reinforcement Learning

1. Sutton, R. S., & Barto, A. G. (2018). *Reinforcement Learning: An Introduction* (2nd ed.). MIT Press.
<http://incompleteideas.net/book/the-book-2nd.html>
2. Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8(3-4), 229-256.
<https://link.springer.com/article/10.1007/BF00992696>
3. Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal Policy Optimization Algorithms. *arXiv preprint arXiv:1707.06347*.
<https://arxiv.org/abs/1707.06347>
4. Mnih, V., et al. (2015). Human-level control through deep reinforcement learning. *Nature*, 518(7540), 529-533.
<https://www.nature.com/articles/nature14236>
5. Silver, D., et al. (2017). Mastering the game of Go without human knowledge. *Nature*, 550(7676), 354-359.
<https://www.nature.com/articles/nature24270>

A.2 Google/DeepMind Models and Research

6. Gemini Team, Google. (2023). Gemini: A Family of Highly Capable Multimodal Models. *arXiv preprint arXiv:2312.11805*.
<https://arxiv.org/abs/2312.11805>
7. Gemini Team, Google. (2024). Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. *arXiv preprint arXiv:2403.05530*.
<https://arxiv.org/abs/2403.05530>
8. Gemini Team, Google. (2024). Gemini 2.0: A New Era of AI Agents. *Google Blog*.
<https://blog.google/technology/google-deepmind/google-gemini-ai-update-december-2024/>
9. Gemini Team, Google DeepMind. (2025). Gemini 2.5: Our latest and most capable AI model. *Google DeepMind Blog*.
<https://deepmind.google/technologies/gemini/>
10. Chowdhery, A., et al. (2023). PaLM: Scaling Language Modeling with Pathways. *Journal of Machine Learning Research*, 24(240), 1-113.
<https://arxiv.org/abs/2204.02311>

11. Anil, R., et al. (2023). PaLM 2 Technical Report. *arXiv preprint arXiv:2305.10403*.
<https://arxiv.org/abs/2305.10403>
12. Gemma Team, Google DeepMind. (2024). Gemma: Open Models Based on Gemini Research and Technology. *arXiv preprint arXiv:2403.08295*.
<https://arxiv.org/abs/2403.08295>
13. Gemma Team, Google DeepMind. (2024). Gemma 2: Improving Open Language Models at a Practical Size. *arXiv preprint arXiv:2408.00118*.
<https://arxiv.org/abs/2408.00118>
14. Lewkowycz, A., et al. (2022). Solving Quantitative Reasoning Problems with Language Models (Minerva). *arXiv preprint arXiv:2206.14858*.
<https://arxiv.org/abs/2206.14858>
15. Rae, J. W., et al. (2021). Scaling Language Models: Methods, Analysis & Insights from Training Gopher. *arXiv preprint arXiv:2112.11446*.
<https://arxiv.org/abs/2112.11446>
16. Hoffmann, J., et al. (2022). Training Compute-Optimal Large Language Models (Chinchilla). *arXiv preprint arXiv:2203.15556*.
<https://arxiv.org/abs/2203.15556>
17. Li, Y., et al. (2022). Competition-Level Code Generation with AlphaCode. *Science*, 378(6624), 1092-1097.
<https://www.science.org/doi/10.1126/science.abq1158>
18. Trinh, T. H., et al. (2024). Solving olympiad geometry without human demonstrations (AlphaGeometry). *Nature*, 625(7995), 476-482.
<https://www.nature.com/articles/s41586-023-06747-5>
19. AlphaProof Team, Google DeepMind. (2024). AI achieves silver-medal standard solving International Mathematical Olympiad problems. *Google DeepMind Blog*.
<https://deepmind.google/discover/blog/ai-solves-imo-problems-at-silver-medal-level/>
20. Glaese, A., et al. (2022). Improving alignment of dialogue agents via targeted human judgements (Sparrow). *arXiv preprint arXiv:2209.14375*.
<https://arxiv.org/abs/2209.14375>
21. Nakano, R., et al. (2021). WebGPT: Browser-assisted question-answering with human feedback. *arXiv preprint arXiv:2112.09332*.
<https://arxiv.org/abs/2112.09332>
22. Reed, S., et al. (2022). A Generalist Agent (Gato). *Transactions on Machine Learning Research*.
<https://arxiv.org/abs/2205.06175>
23. Barham, P., et al. (2022). Pathways: Asynchronous Distributed Dataflow for ML. *MLSys 2022*.
<https://arxiv.org/abs/2203.12533>

A.3 RLHF and Alignment

24. Ouyang, L., et al. (2022). Training language models to follow instructions with human feedback. *NeurIPS 2022*.
<https://arxiv.org/abs/2203.02155>
25. Christiano, P. F., et al. (2017). Deep reinforcement learning from human preferences. *NeurIPS 2017*.
<https://arxiv.org/abs/1706.03741>

26. Stiennon, N., et al. (2020). Learning to summarize from human feedback. *NeurIPS 2020*.
<https://arxiv.org/abs/2009.01325>
27. Bai, Y., et al. (2022). Training a Helpful and Harmless Assistant with Reinforcement Learning from Human Feedback. *arXiv preprint arXiv:2204.05862*.
<https://arxiv.org/abs/2204.05862>
28. Ziegler, D. M., et al. (2019). Fine-tuning language models from human preferences. *arXiv preprint arXiv:1909.08593*.
<https://arxiv.org/abs/1909.08593>
29. Askell, A., et al. (2021). A General Language Assistant as a Laboratory for Alignment. *arXiv preprint arXiv:2112.00861*.
<https://arxiv.org/abs/2112.00861>

A.4 Direct Preference Optimization and Alternatives

30. Rafailov, R., et al. (2023). Direct Preference Optimization: Your Language Model is Secretly a Reward Model. *NeurIPS 2023*.
<https://arxiv.org/abs/2305.18290>
31. Azar, M. G., et al. (2023). A General Theoretical Paradigm to Understand Learning from Human Preferences. *arXiv preprint arXiv:2310.12036*.
<https://arxiv.org/abs/2310.12036>
32. Hong, J., Lee, N., & Thorne, J. (2024). ORPO: Monolithic Preference Optimization without Reference Model. *arXiv preprint arXiv:2403.07691*.
<https://arxiv.org/abs/2403.07691>
33. Ethayarajh, K., Xu, W., Muennighoff, N., Jurafsky, D., & Kiela, D. (2024). KTO: Model Alignment as Prospect Theoretic Optimization. *arXiv preprint arXiv:2402.01306*.
<https://arxiv.org/abs/2402.01306>
34. Ivison, H., et al. (2024). Unpacking DPO and PPO: Disentangling Best Practices for Learning from Preferences. *arXiv preprint arXiv:2406.09279*.
<https://arxiv.org/abs/2406.09279>

A.5 Reasoning and Chain-of-Thought

35. Wei, J., et al. (2022). Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. *NeurIPS 2022*.
<https://arxiv.org/abs/2201.11903>
36. Kojima, T., et al. (2022). Large Language Models are Zero-Shot Reasoners. *NeurIPS 2022*.
<https://arxiv.org/abs/2205.11916>
37. Wang, X., et al. (2023). Self-Consistency Improves Chain of Thought Reasoning in Language Models. *ICLR 2023*.
<https://arxiv.org/abs/2203.11171>
38. Yao, S., et al. (2023). Tree of Thoughts: Deliberate Problem Solving with Large Language Models. *NeurIPS 2023*.
<https://arxiv.org/abs/2305.10601>
39. Zelikman, E., Wu, Y., Mu, J., & Goodman, N. (2022). STaR: Bootstrapping Reasoning With Reasoning. *NeurIPS 2022*.
<https://arxiv.org/abs/2203.14465>

A.6 Test-Time Compute and Scaling

40. Snell, C., et al. (2024). Scaling LLM Test-Time Compute Optimally can be More Effective than Scaling Model Parameters. *arXiv preprint arXiv:2408.03314*.
<https://arxiv.org/abs/2408.03314>
41. Brown, B., et al. (2024). Large Language Monkeys: Scaling Inference Compute with Repeated Sampling. *arXiv preprint arXiv:2407.21787*.
<https://arxiv.org/abs/2407.21787>
42. OpenAI. (2024). Learning to Reason with LLMs. *OpenAI Blog*.
<https://openai.com/index/learning-to-reason-with-llms/>

A.7 Process Reward Models

43. Lightman, H., et al. (2023). Let’s Verify Step by Step. *arXiv preprint arXiv:2305.20050*.
<https://arxiv.org/abs/2305.20050>
44. Uesato, J., et al. (2022). Solving math word problems with process- and outcome-based feedback. *arXiv preprint arXiv:2211.14275*.
<https://arxiv.org/abs/2211.14275>
45. Wang, P., et al. (2024). Math-Shepherd: Verify and Reinforce LLMs Step-by-step without Human Annotations. *arXiv preprint arXiv:2312.08935*.
<https://arxiv.org/abs/2312.08935>

A.8 DeepSeek and Recent Advances

46. DeepSeek-AI. (2025). DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning. *arXiv preprint arXiv:2501.12948*.
<https://arxiv.org/abs/2501.12948>
47. DeepSeek-AI. (2024). DeepSeek-V3 Technical Report. *arXiv preprint arXiv:2412.19437*.
<https://arxiv.org/abs/2412.19437>
48. Shao, Z., et al. (2024). DeepSeekMath: Pushing the Limits of Mathematical Reasoning in Open Language Models. *arXiv preprint arXiv:2402.03300*.
<https://arxiv.org/abs/2402.03300>

A.9 Constitutional AI and Self-Improvement

49. Bai, Y., et al. (2022). Constitutional AI: Harmlessness from AI Feedback. *arXiv preprint arXiv:2212.08073*.
<https://arxiv.org/abs/2212.08073>
50. Sun, Z., et al. (2024). Principle-Driven Self-Alignment of Language Models from Scratch with Minimal Human Supervision. *NeurIPS 2024*.
<https://arxiv.org/abs/2305.03047>
51. Anthony, T., Tian, Z., & Barber, D. (2017). Thinking Fast and Slow with Deep Learning and Tree Search. *NeurIPS 2017*.
<https://arxiv.org/abs/1705.08439>

A.10 Code Generation

52. Chen, M., et al. (2021). Evaluating Large Language Models Trained on Code. *arXiv preprint arXiv:2107.03374*.
<https://arxiv.org/abs/2107.03374>
53. Rozière, B., et al. (2023). Code Llama: Open Foundation Models for Code. *arXiv preprint arXiv:2308.12950*.
<https://arxiv.org/abs/2308.12950>
54. Le, H., et al. (2022). CodeRL: Mastering Code Generation through Pretrained Models and Deep Reinforcement Learning. *NeurIPS 2022*.
<https://arxiv.org/abs/2207.01780>

A.11 Mathematical Reasoning

55. Azerbayev, Z., et al. (2023). Llemma: An Open Language Model For Mathematics. *arXiv preprint arXiv:2310.10631*.
<https://arxiv.org/abs/2310.10631>
56. Hendrycks, D., et al. (2021). Measuring Mathematical Problem Solving With the MATH Dataset. *NeurIPS 2021*.
<https://arxiv.org/abs/2103.03874>

A.12 Verifier-Guided Generation

57. Cobbe, K., et al. (2021). Training Verifiers to Solve Math Word Problems. *arXiv preprint arXiv:2110.14168*.
<https://arxiv.org/abs/2110.14168>
58. Polu, S., & Sutskever, I. (2020). Generative Language Modeling for Automated Theorem Proving. *arXiv preprint arXiv:2009.03393*.
<https://arxiv.org/abs/2009.03393>
59. Li, Y., et al. (2023). Making Language Models Better Reasoners with Step-Aware Verifier. *ACL 2023*.
<https://arxiv.org/abs/2206.02336>

A.13 Multi-Objective RL and Safety

60. Abdolmaleki, A., et al. (2020). A Distributional View on Multi-Objective Policy Optimization. *ICML 2020*.
<https://arxiv.org/abs/2005.07513>
61. Ganguli, D., et al. (2023). The Capacity for Moral Self-Correction in Large Language Models. *arXiv preprint arXiv:2302.07459*.
<https://arxiv.org/abs/2302.07459>
62. Hendrycks, D., et al. (2023). An Overview of Catastrophic AI Risks. *arXiv preprint arXiv:2306.12001*.
<https://arxiv.org/abs/2306.12001>

A.14 Decoding and Sampling Strategies

63. Holtzman, A., et al. (2020). The Curious Case of Neural Text Degeneration. *ICLR 2020*.
<https://arxiv.org/abs/1904.09751>

64. Fan, A., Lewis, M., & Dauphin, Y. (2018). Hierarchical Neural Story Generation. *ACL 2018*.
<https://arxiv.org/abs/1805.04833>
65. Meister, C., et al. (2023). Locally Typical Sampling. *TACL 2023*.
<https://arxiv.org/abs/2202.00666>

A.15 RLHF Algorithms and Variants

66. Guo, S., et al. (2024). Direct Language Model Alignment from Online AI Feedback. *arXiv preprint arXiv:2402.04792*.
<https://arxiv.org/abs/2402.04792>
67. Xu, H., et al. (2024). Contrastive Preference Optimization: Pushing the Boundaries of LLM Performance in Machine Translation. *arXiv preprint arXiv:2401.08417*.
<https://arxiv.org/abs/2401.08417>
68. Liu, T., et al. (2024). Provably Mitigating Overoptimization in RLHF. *arXiv preprint arXiv:2405.16436*.
<https://arxiv.org/abs/2405.16436>
69. Dong, H., et al. (2024). RAFT: Reward rAnked FineTuning for Generative Foundation Model Alignment. *arXiv preprint arXiv:2304.06767*.
<https://arxiv.org/abs/2304.06767>

A.16 Scaling Laws and Empirical Studies

70. Kaplan, J., et al. (2020). Scaling Laws for Neural Language Models. *arXiv preprint arXiv:2001.08361*.
<https://arxiv.org/abs/2001.08361>
71. Muennighoff, N., et al. (2024). Scaling Data-Constrained Language Models. *NeurIPS 2024*.
<https://arxiv.org/abs/2305.16264>

A.17 Foundational LLM Papers

72. Vaswani, A., et al. (2017). Attention is All You Need. *NeurIPS 2017*.
<https://arxiv.org/abs/1706.03762>
73. Brown, T., et al. (2020). Language Models are Few-Shot Learners. *NeurIPS 2020*.
<https://arxiv.org/abs/2005.14165>
74. Touvron, H., et al. (2023). Llama 2: Open Foundation and Fine-Tuned Chat Models. *arXiv preprint arXiv:2307.09288*.
<https://arxiv.org/abs/2307.09288>
75. Anthropic. (2024). The Claude 3 Model Family. *Anthropic Blog*.
<https://www.anthropic.com/news/claude-3-family>

Appendix B

Contact Information

Arun Shankar

Email: `arunshankar@google.com`

LinkedIn: `linkedin.com/in/arunprasath-shankar`

Thank you for reading! I hope this book made RL for LLMs accessible and exciting.