# Lecture 2: IFT, Reward Models, & Rejection Sampling

RLHFBOOK.COM

Nathan Lambert

Course on RLHF and post-training. Chapters 4, 5 & 9

# Lecture 2: IFT, Reward Models, & Rejection Sampling

### Overview

1. Introduction
2. Key Related Works
3. Training Overview

### Core Training Pipeline

4. **Instruction Tuning**
5. **Reward Models**
6. Reinforcement Learning
7. Reasoning
8. Direct Alignment
9. **Rejection Sampling**

### Data & Preferences

10. What are Preferences
11. Preference Data
12. Synthetic Data & CAI

### Practical Considerations

13. Tool Use
14. Over-optimization
15. Regularization
16. Evaluation
17. Product & Character

### Appendices

- A. Definitions
- B. Style & Information
- C. Practical Issues

### Course Home

- rlhfbook.com
- GitHub repo

# Why these three chapters together?

These chapters are the simpler tools for getting started in RLHF & post-training. The core lectures will be on RL, but we need to build foundations first.

# Why these three chapters together?

These chapters are the simpler tools for getting started in RLHF & post-training. The core lectures will be on RL, but we need to build foundations first.

Plus, these chapters form a natural pipeline — **the simplest complete path from a pretrained model to a preference-tuned one (with a reward model)**. A natural step before full RLHF:

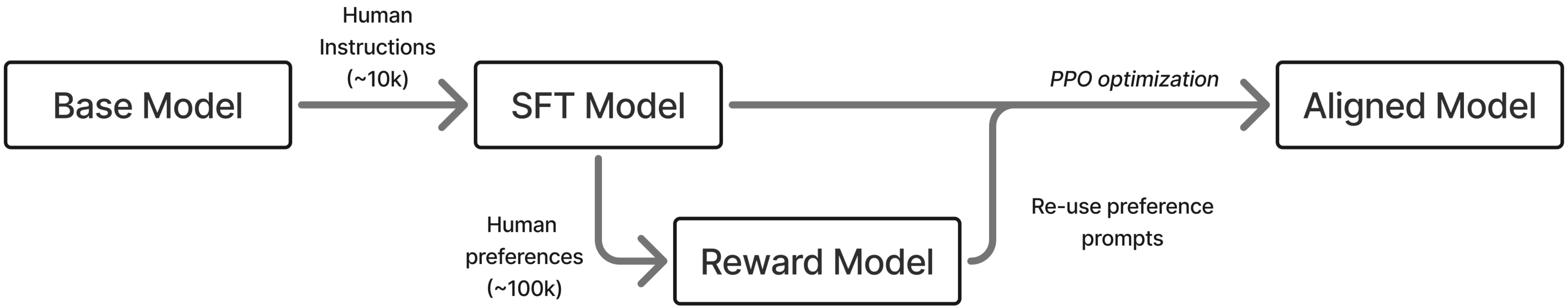
1. **Instruction tuning** teaches the model to follow instructions (the format)
2. **Reward models** learn to score quality from human preferences (the signal)
3. **Rejection sampling** uses those scores to curate better training data (the optimization, which will be replaced with RL)

# Part 1: Instruction Tuning

# Where does instruction tuning fit?

Instruction fine-tuning (IFT), also called supervised fine-tuning (SFT), is the first post-training step.

- Takes a pretrained model that predicts next tokens
- Teaches it to **follow instructions** in a conversational format
- The foundation for all downstream preference tuning (and other methods, e.g. RL-focused recipes for math/code)



The RLHF pipeline — instruction tuning is step 1.

# Two lines of work converged

Instruction tuning emerged from two parallel research threads:

1. **Unified text-to-text frameworks**: T5 (Raffel et al., 2020) showed that framing every NLP task as “text in, text out” worked surprisingly well
2. **Scaling + instruction following**: FLAN (Wei et al., 2022), T0 (Sanh et al., 2022), and Natural Instructions (Mishra et al., 2022) showed that training on diverse tasks with explicit instructions improved zero-shot generalization

# Two lines of work converged

Instruction tuning emerged from two parallel research threads:

1. **Unified text-to-text frameworks**: T5 (Raffel et al., 2020) showed that framing every NLP task as “text in, text out” worked surprisingly well
2. **Scaling + instruction following**: FLAN (Wei et al., 2022), T0 (Sanh et al., 2022), and Natural Instructions (Mishra et al., 2022) showed that training on diverse tasks with explicit instructions improved zero-shot generalization

**SFT took off after ChatGPT**: Alpaca (Taori et al., 2023), OpenAssistant (K{\"o}pf et al., 2023), Tulu (Wang et al., 2023), and many others helped turn instruction tuning into a broadly reproducible open recipe

# Chat templates: the structure of instruction tuning

The model needs a structured format to manage **who is speaking** and **what to generate**. Early chat templates defined three roles:

- **System**: background instructions (persona, constraints)
- **User**: the human’s messages
- **Assistant**: the model’s responses

```
<|im_start|>system
You are a friendly chatbot who always responds in the style of a pirate<|im_end|>
<|im_start|>user
How many helicopters can a human eat in one sitting?<|im_end|>
<|im_start|>assistant
```

The model generates until it produces an end-of-text token (in this case, it is `<|im_end|>`).

# A conversation gets rendered into template text

## Conversation object:

How most post-training data is stored (model-agnostic):

```
messages = [
    {
        "role": "system",
        "content": "You are a friendly chatbo
    },
    {
        "role": "user",
        "content": "How many helicopters can
    },
]
```

## Rendered chat template text:

What the data looks like to a model (tokenizer-specific):

```
<|im_start|>system
You are a friendly chatbot who always responds in
<|im_start|>user
How many helicopters can a human eat in one sitti
<|im_start|>assistant
```

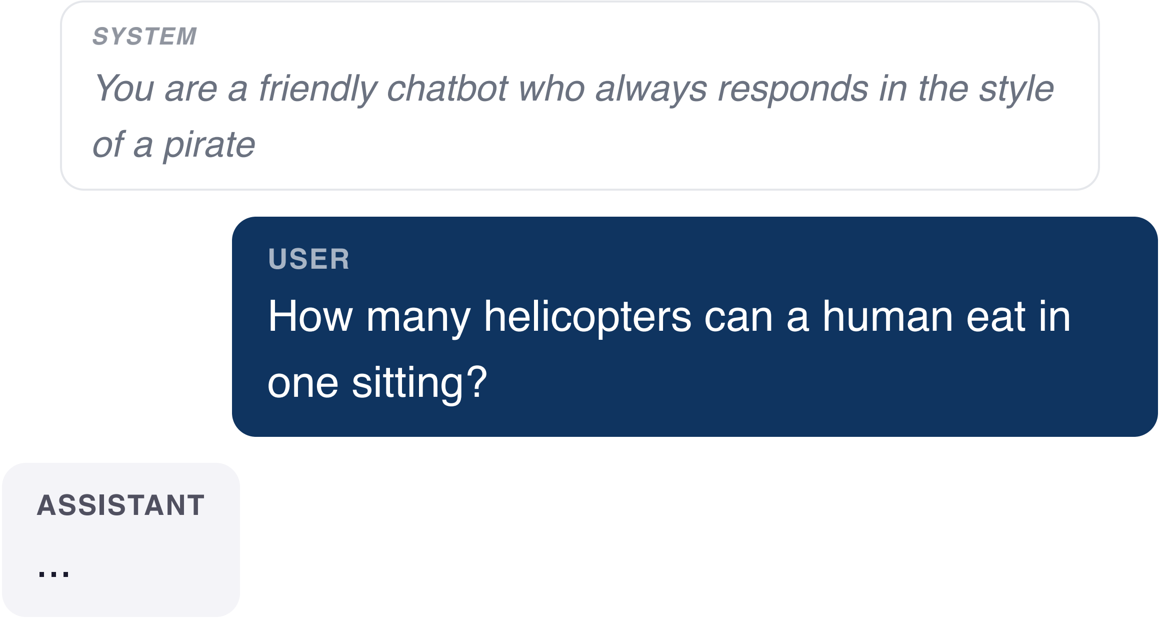
`tokenizer.apply_chat_template(messages)` performs this conversion before tokenization.

# The same example, visualized

What the model sees:

```
<|im_start|>system
You are a friendly chatbot who always
responds in the style of a pirate<|im_end|>
<|im_start|>user
How many helicopters can a human eat
in one sitting?<|im_end|>
<|im_start|>assistant
```

What the user sees:



SYSTEM

*You are a friendly chatbot who always responds in the style of a pirate*

USER

How many helicopters can a human eat in one sitting?

ASSISTANT

...

# Chat templates under the hood: Jinja code

Chat templates are implemented as **Jinja code snippets** stored in the tokenizer config. This is the raw code that converts a list of Python dicts into the token sequence the model sees:

```
{{ bos_token }}
{% for message in messages %}
    {{ '<|im_start|>' + message['role'] + '\n'
       + message['content'] | trim + '<|im_end|>\n' }}
{% endfor %}
{% if add_generation_prompt %}
    {{ '<|im_start|>assistant\n' }}
{% endif %}
```

The full template also enforces role alternation (`user`/`assistant`/`user`/…) and handles the optional `system` message. Applied in code via `tokenizer.apply_chat_template(messages)`. For example, see `Olmo-3-7B-Instruct`’s.

# The pain of Jinja chat templates

oof.

# The pain of Jinja chat templates

There are many ways that working with chat templates is difficult.

- Not easily human-interpretable
- Minor issues, or mismatch to the tokenizer can break training
- Increasing in complexity with reasoning models and tool-use

# Chat templates vary across models

Different model families use different special tokens, but the structure is the same.

**Zephyr (Tunstall et al., 2024):**

```
<|system|>
You are a friendly chatbot...</s>
<|user|>
How many helicopters can a human eat in one sitting?</s>
<|assistant|>
```

**Tulu (Lambert et al., 2024):**

```
<|user|>
How are you doing?
<|assistant|>
I'm just a computer program, so I don't have feelings, but I'm
functioning as expected. How can I assist you today?<|endoftext|>
```

These are applied via Jinja templates stored in the tokenizer config (`apply_chat_template`).

# Multi-turn conversations

Conversations extend naturally by alternating roles:

```
<|im_start|>system
You are a friendly chatbot who always responds in the style of a pirate<|im_end|>
<|im_start|>user
How many helicopters can a human eat in one sitting?<|im_end|>
<|im_start|>assistant
Oh just 6.<|im_end|>
<|im_start|>user
Are you sure about that?<|im_end|>
<|im_start|>assistant
```

The entire history is packed into one token sequence — the model sees all prior turns as context when generating.

# An alternative: OpenAI's Harmony format

OpenAI released **Harmony** alongside gpt-oss, replacing Jinja with a Rust-based renderer that separates output into **channels**:

- `analysis` — internal reasoning / chain-of-thought (hidden from user)
- `commentary` — tool calls go here
- `final` — user-facing response

```
<|start|>assistant<|channel|>analysis<|message|>I need to check the weather...<|end|>
<|start|>assistant<|channel|>commentary to=functions.get_weather
<|constrain|>json<|message|>{"location":"SF"}<|call|>
<|start|>assistant<|channel|>final<|message|>It's 65°F and sunny in SF.<|return|>
```

Why? Jinja can't cleanly handle tool calls (`tojson` escaping, ambiguous boundaries). Harmony moves the complexity into a dedicated library (`openai-harmony` on PyPI) instead of a template string.

# Harmony Python example

```python
from openai_harmony import (Role, Message, Conversation,
    SystemContent, load_harmony_encoding, HarmonyEncodingName)

# Build messages
system = Message.from_role_and_content(Role.SYSTEM, SystemContent.new())
user = Message.from_role_and_content(Role.USER, "What is 2 + 2?")

# Assemble a conversation
convo = Conversation.from_messages([system, user])

# Render to tokens using the OSS encoding
enc = load_harmony_encoding(HarmonyEncodingName.HARMONY_GPT_OSS)
tokens = enc.render_conversation_for_completion(convo, Role.ASSISTANT)

# Decode back to text
print(enc.decode_utf8(tokens))
```

The same `render_conversation_for_completion` / `render_conversation_for_training` split replaces `add_generation_prompt` from Jinja templates.

# What does the model actually learn?

**Prompt masking**: during IFT, the model only learns to predict **assistant responses**, not user messages.

- System and user tokens (usually) are masked from the loss
- Only assistant completion tokens contribute to gradient updates
- The model learns *how to respond*, not *how to ask*

For multi-turn conversations, two strategies:

1. **Final-turn only**: mask everything except the last assistant response
2. **All assistant turns**: mask only user/system tokens, train on every assistant response

Ouyang et al., 2022

# Best practices for instruction tuning

**Data quality matters more than quantity:**

- High-quality completions are critical — prompts are masked anyway, so the model learns from responses
- ~1M prompts is sufficient for excellent RLHF-ready models
  - Further scaling helps, but with diminishing returns
  - SFT prompt quantity has *decreased* a bit with reasoning models. Open research problem
- Best prompts match the downstream task distribution

**Training details differ from pretraining:**

- **Batch size**: much smaller (e.g. 256 vs. 1024–2048 sequences in pretraining)
- **Learning rate**: 1-2 orders of magnitude lower (e.g. $1 \times 10^{-5}$ vs. $3 \times 10^{-4}$), prevent overfitting in narrower data distribution
- **Loss function**: same cross-entropy, but only on unmasked (assistant) tokens

Lambert et al., 2024; Zhou et al., 2023

# Data scaling for IFT

The amount of instruction data needed has evolved rapidly:

- **Early post-ChatGPT**: ~10K high-quality, human-generated samples could be state-of-the-art (Ouyang et al., 2022) (Rajani et al., 2023)
- **Current practice**: large-scale synthetic datasets work best, but quality filtering is essential

Scaling the prompts quickly enabled more performance. Now, reasoning models are using more compute and tokens to train via more tokens per prompt in SFT (and longer context lengths).

# Ingredients for post-training: prompts

Successful post-training starts with **meaningful evaluations** for targeted skills and **prompts of representative queries** for those skills.

All post-training stages require prompts in distribution of tasks. Example prompt budgets from (Lambert et al., 2024):

- **Supervised fine-tuning**: ~1 million prompts
- **Preference fine-tuning**: ~1 million (partial overlap with SFT can be useful)
- **Reinforcement fine-tuning**: ~10-100 thousand (data less available)

Large variance on these numbers is possible — but the key point is that **prompts are the starting material** for every stage. Recent work has of course scaled up various RL training stages :D!

# Building SFT data: synthetic completions

**Synthetic data** has become the dominant approach for building SFT datasets – following (Wang et al., 2023):

1. Start with $N$ high-quality (often human-written) prompts
2. Ask a strong LM to create modified versions of these instructions
3. Generate completions with another (or same) strong LM
4. Result: easily 10x more training data

**Quality of responses** is the simpler part — strong models (e.g. GPT-4o, Llama 3.1 405B) generate good completions to most instructions.

**Human data** is still needed for out-of-distribution or novel tasks. At the time of recording, this is “knowledge work” tasks like healthcare/law.

# The SFT design process

Two repeated and parallelizable tracks:

**Data mixing:**

- Take existing datasets, combine with current mix, observe performance
- Substantial effort in trying to **remove** data and maintain performance
- Start fully with mixing before curation

**Data curation:**

- Identify evaluations where the model is behind
- Create new targeted data for those skills
- [Optionally] filter responses based on quality or correctness

These two tracks iterate: mix what you have, evaluate, curate what's missing, mix again.

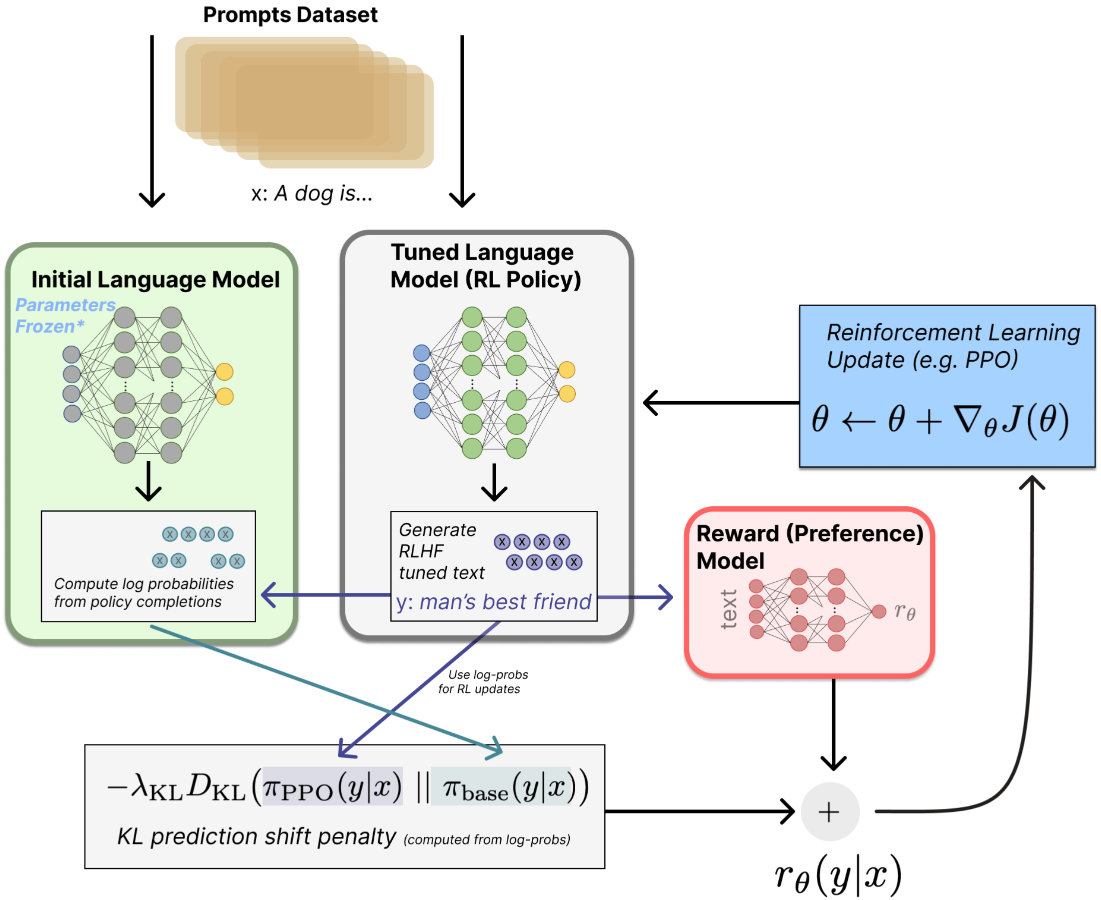
# Part 2: Reward Models

# The role of reward models in RLHF

In RLHF, the reward model plays the role of the **environment** — it returns a reward signal that tells the policy how well it did.

The key difference from standard RL: in RLHF, we get to **control and learn** this reward function from human preferences, rather than having it fixed by the environment.

A reward model compresses complex, subjective human judgments into a single scalar score.


Prompts Dataset
x: *A dog is...*
Initial Language Model
*Parameters Frozen**
*Compute log probabilities from policy completions*
Tuned Language Model (RL Policy)
*Generate RLHF tuned text*
*y: man's best friend*
*Reinforcement Learning Update (e.g. PPO)*
$\theta \leftarrow \theta + \nabla_\theta J(\theta)$
Reward (Preference) Model
text
$r_\theta$
*Use log-probs for RL updates*
$-\lambda_{\text{KL}} D_{\text{KL}}(\pi_{\text{PPO}}(y|x) \,||\, \pi_{\text{base}}(y|x))$
*KL prediction shift penalty* *(computed from log-probs)*
+
$r_\theta(y|x)$

The reward model in RLHF plays the role of the environment's reward signal.

Christiano et al., 2017

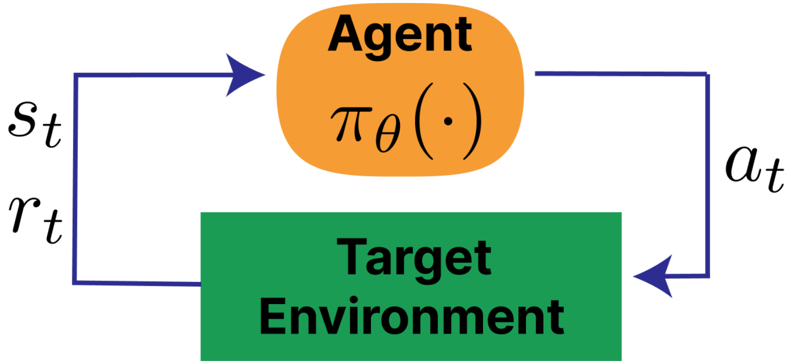
# Recall: Classical Reinforcement Learning

A reinforcement learning problem is often written as a **Markov Decision Process (MDP)**:

- state space $\mathcal{S}$, action space $\mathcal{A}$
- transition dynamics $P(s_{t+1} \mid s_t, a_t)$
- reward function $r(s_t, a_t)$ and discount $\gamma$
- optimize cumulative return over a trajectory

$$\text{MDP } (\mathcal{S}, \mathcal{A}, P, r, \gamma)$$

$$J(\pi) = \mathbb{E}_{\tau \sim \pi}\left[\sum_{t=0}^{T} \gamma^t r(s_t, a_t)\right]$$



Sutton & Barto, 2018

# Recall: Classical RL vs. RLHF

### Classical RL

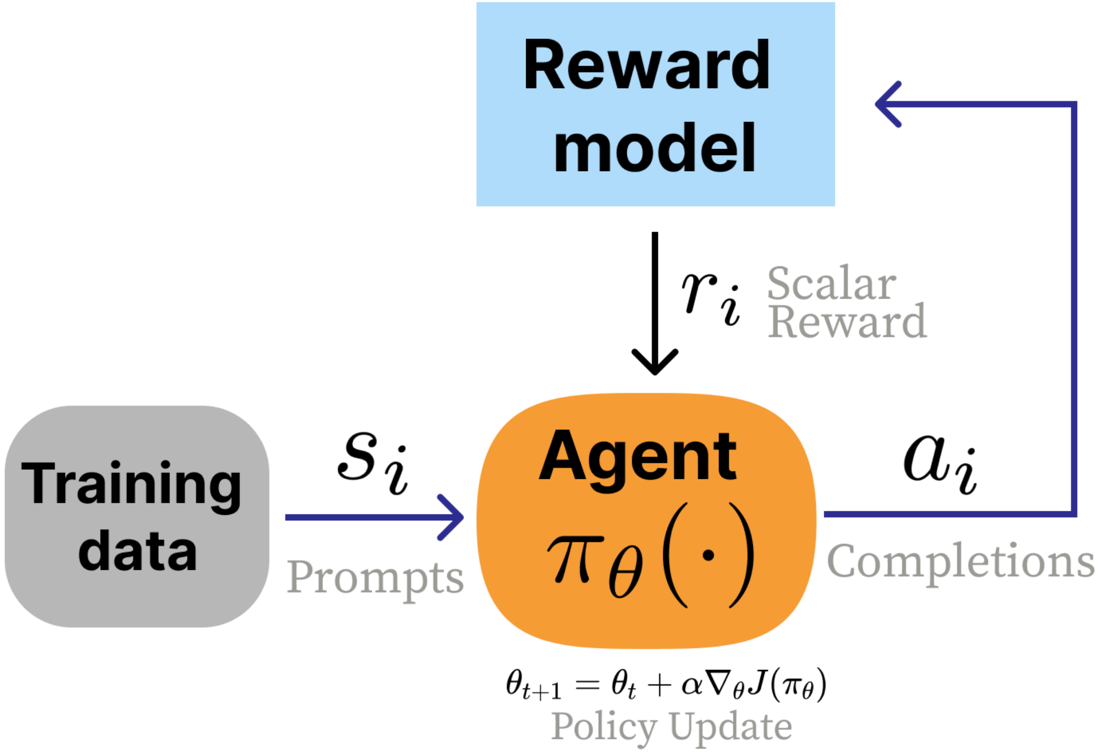
- Agent takes actions $a_t$ in an environment with states $s_t$
- Reward is a known function $r(s_t, a_t)$ from the environment per step
- Optimize cumulative return over a trajectory (total steps $T$)

$$J(\pi) = \mathbb{E}_{\tau\sim\pi}\left[\sum_{t=0}^{T} \gamma^t r(s_t, a_t)\right]$$

### RLHF

- No environment — prompts sampled from a dataset
- Reward is **learned** from human preferences (a proxy)
- **Response-level** reward (bandit-style, not per-token)
- Regularized with **KL penalty** to stay close to the base model

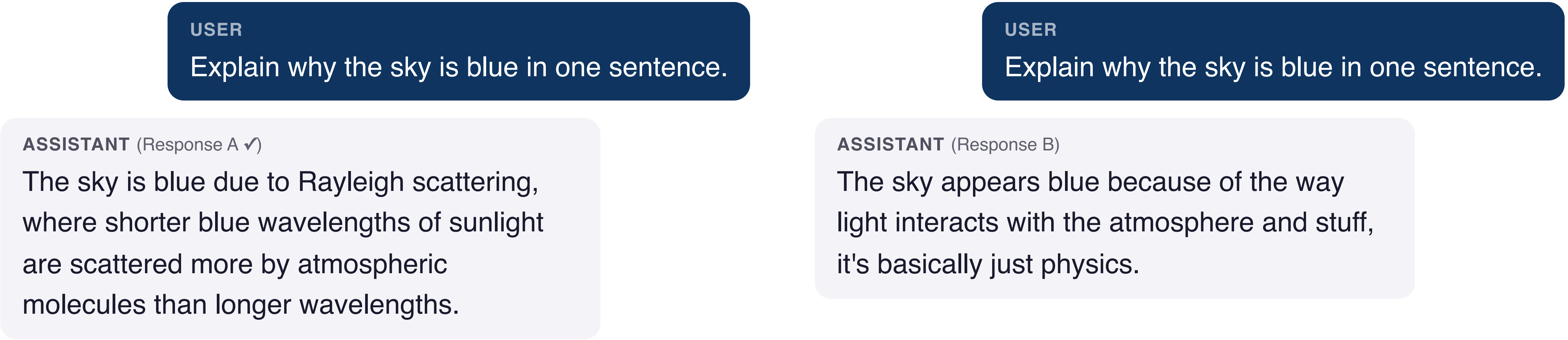
$$J(\pi) = \mathbb{E}\left[r_\theta(x, y)\right] - \beta\, D_{\mathrm{KL}}(\pi \| \pi_{\mathrm{ref}})$$



Christiano et al., 2017; Ouyang et al., 2022

# Recall: What RLHF comparison data looks like

A human (or AI) annotator sees two responses to the same prompt and picks the better one — this preference pair becomes training data for the reward model.



**USER**

Explain why the sky is blue in one sentence.

**ASSISTANT** (Response A ✓)

The sky is blue due to Rayleigh scattering, where shorter blue wavelengths of sunlight are scattered more by atmospheric molecules than longer wavelengths.

**USER**

Explain why the sky is blue in one sentence.

**ASSISTANT** (Response B)

The sky appears blue because of the way light interacts with the atmosphere and stuff, it's basically just physics.

# The Bradley-Terry model of preferences

A **probability model** is a mathematical form that we assume matches how real judgments work — then we fit its parameters to data. The canonical reward model uses the **Bradley-Terry model** (1952).

Bradley & Terry, 1952

# The Bradley-Terry model of preferences

A **probability model** is a mathematical form that we assume matches how real judgments work — then we fit its parameters to data. The canonical reward model uses the **Bradley-Terry model** (1952). Given two items $i$ and $j$, the probability that a judge prefers $i$ over $j$:

$$P(i > j) = \frac{p_i}{p_i + p_j}$$

Each item has a latent strength $p_i > 0$. Reparametrizing with $p_i = e^{r_i}$ lets us work with unbounded scores $r_i \in \mathbb{R}$ — which is what a neural network naturally outputs:

$$P(i > j) = \frac{e^{r_i}}{e^{r_i} + e^{r_j}} = \sigma(r_i - r_j)$$

Only **score differences** matter — adding the same constant to all scores leaves preferences unchanged.

This is not the only possible model — but it's what worked in early RLHF and has stuck. It's a useful approximation, not a law of nature.

Bradley & Terry, 1952

# From Bradley-Terry to a reward model

Given a prompt $x$, a chosen completion $y_c$, and a rejected completion $y_r$, we score both with a reward model $r_\theta$:

$$P(y_c > y_r \mid x) = \frac{\exp(r_\theta(y_c \mid x))}{\exp(r_\theta(y_c \mid x)) + \exp(r_\theta(y_r \mid x))}$$

We want to find $\theta$ that maximizes this probability — i.e. the model should assign a higher score to the completion humans preferred.

# Deriving the reward model loss

Start from maximizing the preference probability:

$$\theta^* = \arg\max_\theta \frac{\exp(r_\theta(y_c \mid x))}{\exp(r_\theta(y_c \mid x)) + \exp(r_\theta(y_r \mid x))}$$

Ouyang et al., 2022

# Deriving the reward model loss

Start from maximizing the preference probability:

$$\theta^* = \arg\max_\theta \frac{\exp(r_\theta(y_c \mid x))}{\exp(r_\theta(y_c \mid x)) + \exp(r_\theta(y_r \mid x))}$$

Divide numerator and denominator by $\exp(r_\theta(y_c \mid x))$:

$$= \arg\max_\theta \frac{\exp(r_\theta(y_c \mid x)) \ / \ \exp(r_\theta(y_c \mid x))}{(\exp(r_\theta(y_c \mid x)) + \exp(r_\theta(y_r \mid x))) \ / \ \exp(r_\theta(y_c \mid x))}$$

Ouyang et al., 2022

# Deriving the reward model loss

Start from maximizing the preference probability:

$$\theta^* = \arg\max_\theta \frac{\exp(r_\theta(y_c \mid x))}{\exp(r_\theta(y_c \mid x)) + \exp(r_\theta(y_r \mid x))}$$

Divide numerator and denominator by $\exp(r_\theta(y_c \mid x))$:

$$= \arg\max_\theta \frac{\exp(r_\theta(y_c \mid x)) \;/\; \exp(r_\theta(y_c \mid x))}{(\exp(r_\theta(y_c \mid x)) + \exp(r_\theta(y_r \mid x))) \;/\; \exp(r_\theta(y_c \mid x))}$$

The numerator becomes $1$, and in the denominator $\frac{\exp(r_\theta(y_r|x))}{\exp(r_\theta(y_c|x))} = \exp(r_\theta(y_r \mid x) - r_\theta(y_c \mid x))$:

$$= \arg\max_\theta \frac{1}{1 + \exp(r_\theta(y_r \mid x) - r_\theta(y_c \mid x))}$$

Ouyang et al., 2022

# Deriving the reward model loss

Start from maximizing the preference probability:

$$\theta^* = \arg\max_\theta \frac{\exp(r_\theta(y_c \mid x))}{\exp(r_\theta(y_c \mid x)) + \exp(r_\theta(y_r \mid x))}$$

Divide numerator and denominator by $\exp(r_\theta(y_c \mid x))$:

$$= \arg\max_\theta \frac{1}{1 + \exp(r_\theta(y_r \mid x) - r_\theta(y_c \mid x))}$$

Rewrite as $\frac{1}{1+\exp(-(r_\theta(y_c|x)-r_\theta(y_r|x)))}$. Recognize the sigmoid $\sigma(z) = \frac{1}{1+e^{-z}}$:

$$= \arg\max_\theta \ \sigma(r_\theta(y_c \mid x) - r_\theta(y_r \mid x))$$

Ouyang et al., 2022

# Deriving the reward model loss

$$\theta^* = \arg\max_\theta \ \sigma(r_\theta(y_c \mid x) - r_\theta(y_r \mid x))$$

$\log$ is monotonic, so $\arg\max \sigma(\cdot) = \arg\max \log \sigma(\cdot)$:

$$= \arg\max_\theta \ \log \sigma(r_\theta(y_c \mid x) - r_\theta(y_r \mid x))$$

Flip the sign to turn maximization into minimization (a loss):

$$\mathcal{L}(\theta) = -\log \sigma \left(r_\theta(y_c \mid x) - r_\theta(y_r \mid x)\right)$$

Ouyang et al., 2022

# Two equivalent forms of the loss

The first form, as in InstructGPT (Ouyang et al., 2022):

$$\mathcal{L}(\theta) = -\log\left(\sigma\left(r_\theta(y_c \mid x) - r_\theta(y_r \mid x)\right)\right)$$

The second form, as in Anthropic's work (Askell et al., 2021):

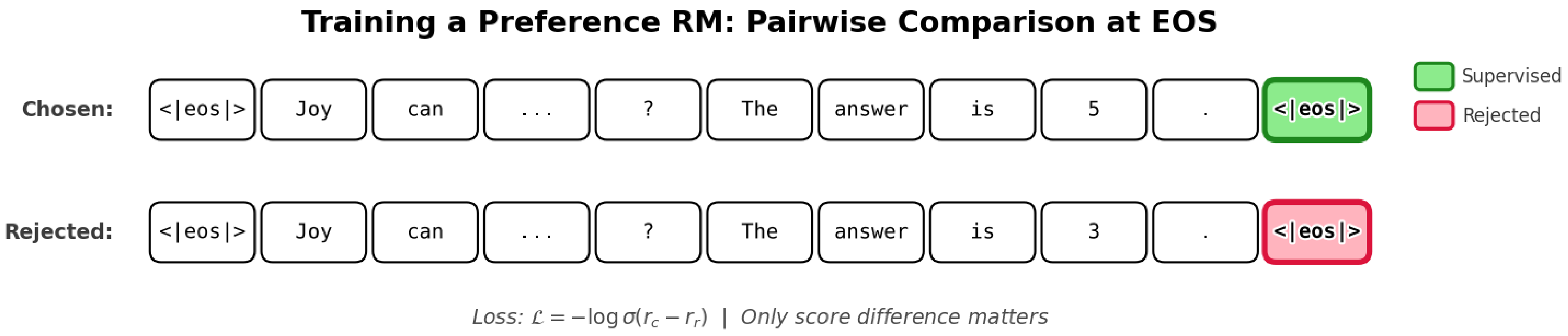
$$\mathcal{L}(\theta) = \log\left(1 + e^{r_\theta(y_r|x) - r_\theta(y_c|x)}\right)$$

These are equivalent: using $\sigma(\Delta) = \frac{1}{1+e^{-\Delta}}$ gives $-\log\sigma(\Delta) = \log(1 + e^{-\Delta})$.

Both are just **binary cross-entropy** — the reward model is learning to classify which completion was preferred.

# Reward model training visualized

The model computes a scalar score at the EOS token for each completion. The contrastive loss depends only on the **score difference** between chosen and rejected.



Training a Preference RM: Pairwise Comparison at EOS

Chosen: <|eos|> Joy can ... ? The answer is 5 . <|eos|>

Rejected: <|eos|> Joy can ... ? The answer is 3 . <|eos|>

🟩 Supervised
🟥 Rejected

*Loss:* $\mathcal{L} = -\log \sigma(r_c - r_r)$ | *Only score difference matters*

Training a preference reward model requires pairs of chosen and rejected completions.

Ouyang et al., 2022

# Reward model architecture

The most common implementation: append a **linear head** to a language model that outputs a single scalar.

```python
class BradleyTerryRewardModel(nn.Module):
    def __init__(self, base_lm):
        super().__init__()
        self.lm = base_lm
        self.head = nn.Linear(self.lm.config.hidden_size, 1)

    def forward(self, input_ids, attention_mask):
        hidden = self.lm(
            input_ids=input_ids, attention_mask=attention_mask,
            output_hidden_states=True, return_dict=True,
        ).hidden_states[-1]
        lengths = attention_mask.sum(dim=1) - 1
        batch_idx = torch.arange(hidden.size(0), device=hidden.device)
        seq_repr = hidden[batch_idx, lengths]
        return self.head(seq_repr).squeeze(-1)
```

# The loss is simple

Given the model above, the training loss is just three lines:

```
rewards_chosen = model(**inputs_chosen)
rewards_rejected = model(**inputs_rejected)

loss = -nn.functional.logsigmoid(rewards_chosen - rewards_rejected).mean()
```

**Practical note**: reward models are typically trained for only **1 epoch** to avoid overfitting to the preference data.

# Reward Model Variants

# Preference margin loss

When annotators provide **Likert scale ratings** (e.g. 1-5), the magnitude of the preference can inform training. Llama 2 proposed a margin term $m(y_c, y_r)$:

$$\mathcal{L}(\theta) = -\log\left(\sigma\left(r_\theta(y_c \mid x) - r_\theta(y_r \mid x) - m(y_c, y_r)\right)\right)$$

For example, if chosen scores 5 and rejected scores 2, then $m = 3$.

This encourages the model to produce **larger score gaps** for strongly preferred pairs.

**Note**: Llama 3 removed the margin term — the team observed diminishing improvements at scale.

Touvron et al., 2023

# K-wise loss and balancing comparisons

**InstructGPT** balances multiple comparisons per prompt to prevent overfitting (and included all prompts in the same batch at training time to make this work):

$$\mathcal{L}(\theta) = -\frac{1}{\binom{K}{2}} \mathbb{E}_{(x,y_c,y_r)\sim D} \log \left( \sigma \left( r_\theta(y_c \mid x) - r_\theta(y_r \mid x) \right) \right)$$

**K-wise loss** (Plackett-Luce model, used in Starling 7B/34B) handles full rankings over $K$ completions:

$$P(\sigma^i | s^i, a_0^i, \ldots, a_{K-1}^i) = \prod_{k=0}^{K-1} \frac{\exp(r_\theta(s^i, a_{\sigma^i(k)}^i))}{\sum_{j=k}^{K-1} \exp(r_\theta(s^i, a_{\sigma^i(j)}^i))}$$

When $K = 2$, this reduces to Bradley-Terry.

Ouyang et al., 2022; Zhu et al., 2023; Zhu et al., 2024

# Outcome reward models (ORMs)

For **reasoning tasks**, we often have verifiable correctness. ORMs train a per-token head using the completion-level correctness label repeated across completion tokens:

$$\mathcal{L}_{\text{CE}}(\theta) = -\mathbb{E}_{(s,r)\sim\mathcal{D}}[r \log p_\theta(s) + (1-r)\log(1-p_\theta(s))]$$

where $r \in \{0, 1\}$ is the completion-level correctness label, broadcast across completion tokens during training.

**Key differences from Bradley-Terry**: no pairwise comparisons needed — just correct/incorrect labels per response. And the model outputs a **per-token probability of correctness**, not a single score at the EOS token.

Cobbe et al., 2021

# BT reward model vs. ORM: forward pass

**Bradley-Terry RM** — score at the **last token**:

```
hidden = lm(**inputs,
    output_hidden_states=True
).hidden_states[-1]

# Extract EOS token representation
lengths = attention_mask.sum(dim=1) - 1
batch_idx = torch.arange(hidden.size(0))
seq_repr = hidden[batch_idx, lengths]

# Single scalar per completion
score = head(seq_repr).squeeze(-1)
#       shape: (batch,)
```

**ORM** — score at **every token**:

```
hidden = lm(**inputs,
    output_hidden_states=True
).hidden_states[-1]

# Score every token position
logits = head(hidden).squeeze(-1)
#        shape: (batch, seq_len)
```

Both start from the same base LM hidden states — the difference is **where** the head is applied.

# BT reward model vs. ORM in code

**Bradley-Terry RM loss** — contrastive, needs pairs:

```
score_chosen = model(**inputs_chosen)
score_rejected = model(**inputs_rejected)

loss = -F.logsigmoid(
    score_chosen - score_rejected
).mean()
```

One scalar per completion. Loss depends on the **difference** between chosen and rejected.

**ORM loss** — per-token, needs only labels:

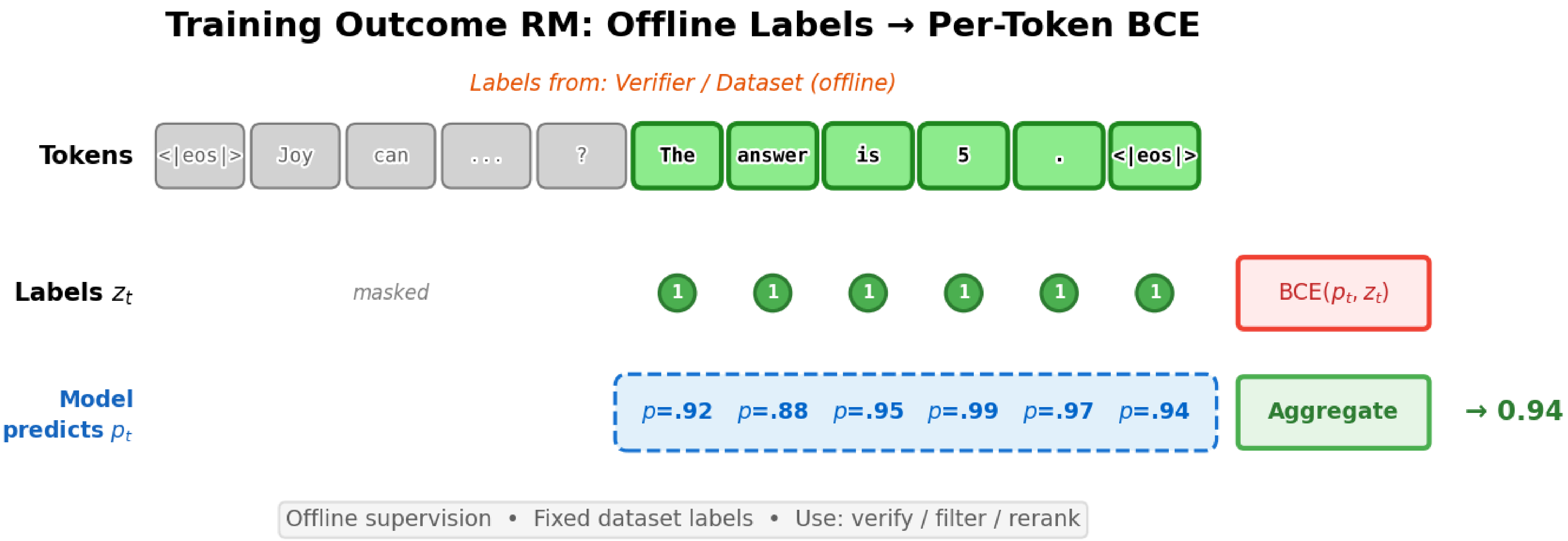
```
logits = model(**inputs)
# shape: (batch, seq_len)

mask = labels != -100
loss = F.binary_cross_entropy_with_logits(
    logits[mask], labels[mask].float()
)
```

One score per token. Every completion token is trained against the same outcome label ($r = 0$ or $1$).

# ORM training visualized

The outcome label ($r = 0$ or $1$) is **broadcast** to every completion token. Prompt tokens are masked. The model learns per-token correctness predictions from this repeated signal. In a larger batch of training, there is a diverse signal of supervision to learn from.



Training Outcome RM: Offline Labels → Per-Token BCE

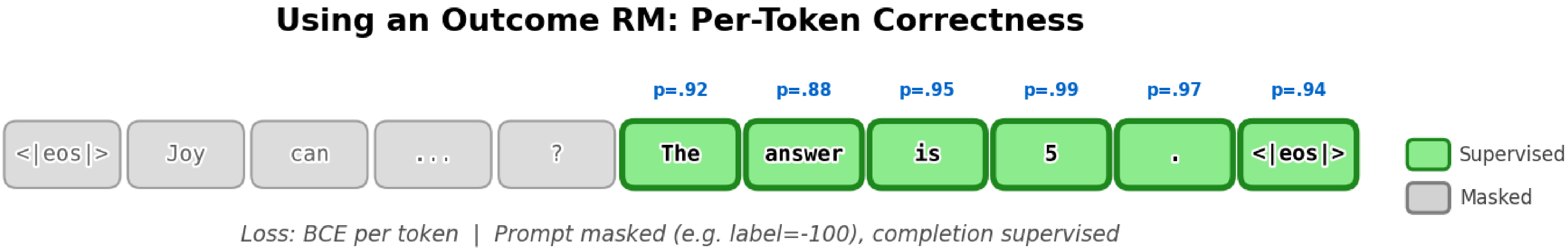
*Labels from: Verifier / Dataset (offline)*

Offline supervision • Fixed dataset labels • Use: verify / filter / rerank

Training an ORM: the completion-level correctness label is applied to every completion token via binary cross-entropy.

# ORM inference

At inference time, the ORM outputs a probability of correctness **at every token**. These token-level scores can then be aggregated into a response-level verifier score for filtering or reranking.


Using an Outcome RM: Per-Token Correctness
<|eos|>
Joy
can
...
?
p=.92
The
p=.88
answer
p=.95
is
p=.99
5
p=.97
.
p=.94
<|eos|>
Supervised
Masked
Loss: BCE per token | Prompt masked (e.g. label=-100), completion supervised

At inference time, an ORM outputs per-token correctness probabilities over the completion.

# Common confusion: “ORM” ≠ BT on correct vs. incorrect

Some papers use “outcome reward model” to mean a **Bradley-Terry model trained on correct vs. incorrect completions**. This conflates two different architectures:

| | **BT RM (on correct/incorrect)** | **ORM (Cobbe et al.)** |
|---|---|---|
| **Min. Training Input** | Two completions (pair) | One completion + label |
| **Output** | Single scalar at EOS | Per-token probability |
| **Loss** | $-\log \sigma(r_c - r_{ic})$ | Per-token binary cross-entropy |
| **Head** | Score at last token | Score at every token |

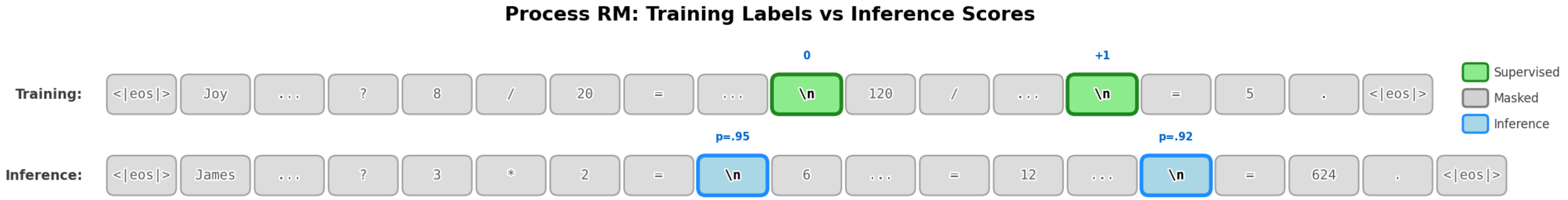
A BT model trained on (correct, incorrect) pairs is still a **preference RM** — it just uses correctness as the preference signal. A true ORM has a fundamentally different input-output structure.

# Process reward models (PRMs)

PRMs score intermediate **reasoning steps**, not just final outcomes:

$$\mathcal{L}_{\mathrm{PRM}}(\theta) = -\mathbb{E}_{(x,s)\sim\mathcal{D}}\left[\sum_{i=1}^{K} \log p_\theta(z_{s_i} \mid x, s_i)\right]$$

Labels are applied only at **step boundaries** (e.g. double newlines): incorrect (-1), neutral (0), correct (+1). All other tokens are masked during training.



*Training: 3-class labels at step boundaries | Inference: predicted step scores*

PRMs provide supervision only at step boundaries. Each step receives a 3-class label.

Lightman et al., 2024

# PRM in code

**3-class head** (not binary like ORM):

```
# head outputs 3 classes per token:
#   0 = incorrect, 1 = neutral, 2 = correct
head = nn.Linear(hidden_size, 3)

hidden = lm(**inputs,
    output_hidden_states=True
).hidden_states[-1]

logits = head(hidden)
#        shape: (batch, seq_len, 3)
```

**Loss at step boundaries only**:

```
# Labels: class index at step-end tokens,
#         -100 everywhere else
mask = labels != -100
loss = F.cross_entropy(
    logits[mask], labels[mask]
)
```

vs. ORM: binary labels at *every* completion token.

vs. PRM: 3-class labels at *step boundaries* only.

# Comparing reward model types

| Model | Predicts | Trained on | Output |
|---|---|---|---|
| **Preference RM** | Quality at EOS token | Pairwise (chosen vs. rejected) | Single scalar |
| **ORM** | Outcome signal over completion tokens | Binary outcome labels | Per-token probability |
| **PRM** | Per-step correctness | Step-level annotations | Score at step boundaries |
| **Value Function** | Expected remaining return | On-policy rollouts | Per-token expected return |

**Key distinction — ORM vs. Value Function:**

- **ORMs** predict offline-labeled correctness with a per-token head: $p(\text{correct}_t)$
- **Value functions** predict expected *remaining* return: $V(s_t) = \mathbb{E}[\sum_{k\geq t} \gamma^{k-t} r_k \mid s_t]$

Same architecture, different semantics and supervision pipeline. More on value functions in the policy gradient lecture(s).

# In summary

- **RM:** “How good is this whole answer?” — scalar value
- **ORM:** “Is this response on track to be correct?” — outcome signal trained across completion tokens
- **PRM:** “Are the reasoning steps sound?” — per-step scores
- **Value Function:** “How much reward remains from here?” — baseline for RL advantages

# Generative reward modeling: LLM-as-a-judge

An alternative to training a reward model: **prompt an LLM** to judge quality (prompt from (Zheng et al., 2023)).

```
[System]
Please act as an impartial judge and evaluate the quality of the
responses provided by two AI assistants to the user question below.
...
After providing your explanation, output your final verdict by strictly
following this format: "[[A]]" if assistant A is better, "[[B]]" if
assistant B is better, and "[[C]]" for a tie.
```

Spawned benchmarks: MT-Bench, AlpacaEval, Arena-Hard, WildBench.

**Current status**: generative RMs tend to **underperform** trained reward models on RM evaluations, but are cheaper to set up and used widely in evaluation and training pipelines. An evaluation/utility mismatch exists.

# Reward model evaluation

I very happily helped kickstart this area with the first one (Lambert et al., 2025)! The RM evaluation landscape has expanded rapidly:

- **General chat:** RewardBench, RewardBench2, RMB, RM-Bench
- **Specialized:** M-RewardBench (multilingual), RAG-RewardBench, RewardMATH
- **Process RMs:** PRM Bench, ProcessBench
- **Agentic:** Agent-RewardBench
- **Multimodal:** MJ-Bench, Multimodal RewardBench, VL RewardBench

The bulk of early RM research focused on establishing benchmarks and identifying behavior modes. Training innovations (aspect-conditioned models, high-quality human datasets, scaling experiments) are still an active area.

# Part 3: Rejection Sampling

# The simplest thing you can do with a reward model

**Rejection sampling** is a popular baseline for preference fine-tuning. The idea:

1. Generate many candidate completions
2. Score them with a reward model
3. Keep only the best ones
4. Fine-tune on those

No policy gradients, no online RL — just **filtered supervised learning**.

The name comes from computational statistics: sample from a simple distribution, use a heuristic to accept/reject samples, approximating a complex target distribution.

Gilks & Wild, 1992

# Rejection sampling overview

The four stages: 0. **Select prompts and reward model** (reuse IFT prompts or curate new ones)

1. **Generate** $N$ completions per prompt from the current model
2. **Score** all completions with the reward model
3. **Fine-tune** on the top completions (same SFT loss as instruction tuning)

Used in WebGPT (Nakano et al., 2021), Anthropic's Helpful and Harmless (Bai et al., 2022), Llama 2 Chat (Touvron et al., 2023), and many other seminal works.

This is still just **offline data curation**: generate first, then train on the filtered outputs.

# The math: prompts, completions, rewards

Given $M$ prompts and $N$ completions each:

$$X = [x_1, x_2, \ldots, x_M]$$

$$Y = \begin{bmatrix} y_{1,1} & y_{1,2} & \cdots & y_{1,N} \\ y_{2,1} & y_{2,2} & \cdots & y_{2,N} \\ \vdots & \vdots & \ddots & \vdots \\ y_{M,1} & y_{M,2} & \cdots & y_{M,N} \end{bmatrix} \quad R = \begin{bmatrix} r_{1,1} & r_{1,2} & \cdots & r_{1,N} \\ r_{2,1} & r_{2,2} & \cdots & r_{2,N} \\ \vdots & \vdots & \ddots & \vdots \\ r_{M,1} & r_{M,2} & \cdots & r_{M,N} \end{bmatrix}$$

Each reward: $r_{i,j} = \mathcal{R}(y_{i,j} \mid x_i)$

# Selection method 1: top per prompt

Select the highest-scoring completion for each prompt independently:

$$S(R) = \left[\arg\max_j r_{1,j},\ \arg\max_j r_{2,j},\ \ldots,\ \arg\max_j r_{M,j}\right]$$

**Example** (5 prompts, 4 completions):

$$R = \begin{bmatrix} \mathbf{0.7} & 0.3 & 0.5 & 0.2 \\ 0.4 & \mathbf{0.8} & 0.6 & 0.5 \\ \mathbf{0.9} & 0.3 & 0.4 & 0.7 \\ 0.2 & 0.5 & \mathbf{0.8} & 0.6 \\ 0.5 & 0.4 & 0.3 & \mathbf{0.6} \end{bmatrix} \quad S(R) = [1, 2, 1, 3, 4]$$

Result: one completion per prompt, every prompt represented.

# Selection method 2: top overall pairs

Flatten the reward matrix and select the top $K$ pairs globally:

$$S_K(R_{\text{flat}}) = \text{argsort}(R_{\text{flat}})[-K:]$$

Here, $R_{\text{flat}}$ is the length-$MN$ vector made by flattening the reward matrix. Each selected flat index is then mapped back to its original `(prompt, completion)` pair.

**Same example, top 5 overall:**

$$R = \begin{bmatrix} \mathbf{0.7} & 0.3 & 0.5 & 0.2 \\ 0.4 & \mathbf{0.8} & 0.6 & 0.5 \\ \mathbf{0.9} & 0.3 & 0.4 & \mathbf{0.7} \\ 0.2 & 0.5 & \mathbf{0.8} & 0.6 \\ 0.5 & 0.4 & 0.3 & 0.6 \end{bmatrix}$$

Now prompt 3 gets **two** completions (0.9 and 0.7), while prompt 5 gets **none**. This optimizes for absolute quality but can bias toward easy prompts.

# Implementation details

The core hyperparameters are intuitive:

- **Temperature**: 0.7-1.0 (need diversity in completions)
- **Completions per prompt**: 10-30+ (too few = noisy selection)
- **Fine-tuning**: standard SFT on selected completions (same loss, but details like learning rate may differ from initial IFT)

**Practical tip**: sort completions by length before batch RM inference to reduce padding token computation.

**Open questions**: how to sequence RS in a multi-stage pipeline, whether to use generations from multiple models, optimal prompt selection. There hasn't been a fully open reproduction of this, which is kind of confusing as to why. My hunch is that training an reward model has some subtle tricks.

# Best-of-N sampling: rejection sampling without fine-tuning

**Best-of-N (BoN)** follows the same generate-and-score procedure, but skips the fine-tuning step:

$$S(R) = \arg\max_{j \in [1,N]} r_j$$

- Used at **inference time** to pick the best completion
- Does **not** modify the model — it's a sampling technique
- Often used as a baseline comparison for online RL methods like PPO

BoN is the simplest possible reward-guided method: generate more, pick the best. Can also be done with verification / LLM-as-a-judge instead of traditional reward models.

Liu et al., 2024

# The full pipeline so far

Putting it all together — a simple path from pretrained model to preference-tuned model:

1. **Instruction fine-tune** the pretrained model → learns the chat format
2. **Collect preference data** → human annotators compare pairs of responses (more on data after the training lectures)
3. **Train a reward model** on preference data → learns to score quality
4. **Rejection sampling** → generate many completions, keep the best, fine-tune

This is a strong baseline.

Next lectures will cover more powerful optimization methods: PPO, which optimizes a policy against a learned reward signal, and DPO, which optimizes directly from preference pairs rather than filtering training data.

# Next up: RL! 🍒

### Overview

1. Introduction
2. Key Related Works
3. Training Overview

### Core Training Pipeline

4. **Instruction Tuning**
5. **Reward Models**
6. Reinforcement Learning
7. Reasoning
8. Direct Alignment
9. **Rejection Sampling**

### Data & Preferences

10. What are Preferences
11. Preference Data
12. Synthetic Data & CAI

### Practical Considerations

13. Tool Use
14. Over-optimization
15. Regularization
16. Evaluation
17. Product & Character

### Appendices

- A. Definitions
- B. Style & Information
- C. Practical Issues

### Course Home

- rlhfbook.com
- GitHub repo

# Thank you

Questions / discussion

Contact: nathan@natolambert.com

Newsletter: interconnects.ai

**rlhfbook.com**

BUILT WITH
**natolambert/colloquium**

# References (1/4)

Askell, A., Bai, Y., Chen, A., Drain, D., Ganguli, D., et al.. “*A general language assistant as a laboratory for alignment*.” *arXiv preprint arXiv:2112.00861*, 2021.

Bai, Y., Jones, A., Ndousse, K., Askell, A., Chen, A., et al.. “*Training a helpful and harmless assistant with reinforcement learning from human feedback*.” *arXiv preprint arXiv:2204.05862*, 2022.

Bradley, R., and Terry, M.. “*Rank Analysis of Incomplete Block Designs: I. The Method of Paired Comparisons*.” *Biometrika*, 1952.

Christiano, P., Leike, J., Brown, T., Martic, M., Legg, S., et al.. “*Deep Reinforcement Learning from Human Preferences*.” *Advances in Neural Information Processing Systems*, 2017.

Cobbe, K., Kosaraju, V., Bavarian, M., Chen, M., Jun, H., et al.. “*Training Verifiers to Solve Math Word Problems*.” *arXiv preprint arXiv:2110.14168*, 2021.

Gilks, W., and Wild, P.. “*Adaptive rejection sampling for Gibbs sampling*.” *Journal of the Royal Statistical Society: Series C (Applied Statistics)*, 1992.

K{\"o}pf, A., Kilcher, Y., R{\"u}tte, D., Anagnostidis, S., Tam, Z., et al.. “*OpenAssistant Conversations -- Democratizing Large Language Model Alignment*.” *Advances in Neural Information Processing Systems, Datasets and Benchmarks Track*, 2023. [link]

# References (2/4)

Lambert, N., Morrison, J., Pyatkin, V., Huang, S., Ivison, H., et al.. “*T\""ulu 3: Pushing Frontiers in Open Language Model Post-Training.*” *arXiv preprint arXiv:2411.15124*, 2024.

Lambert, N., Pyatkin, V., Morrison, J., Miranda, L., Lin, B., et al.. “*RewardBench: Evaluating reward models for language modeling.*” *Conference of the North American Chapter of the Association for Computational Linguistics*, 2025.

Lightman, H., Kosaraju, V., Burda, Y., Edwards, H., Baker, B., et al.. “*Let's verify step by step.*” *International Conference on Learning Representations*, 2024.

Liu, T., Zhao, Y., Joshi, R., Khalman, M., Saleh, M., et al.. “*Statistical Rejection Sampling Improves Preference Optimization.*” *International Conference on Learning Representations*, 2024.

Mishra, S., Khashabi, D., Baral, C., and Hajishirzi, H.. “*Cross-Task Generalization via Natural Language Crowdsourcing Instructions.*” *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics*, 2022.

Nakano, R., Hilton, J., Balaji, S., Wu, J., Ouyang, L., et al.. “*WebGPT: Browser-assisted question-answering with human feedback.*” *arXiv preprint arXiv:2112.09332*, 2021.

Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C., et al.. “*Training language models to follow instructions with human feedback.*” *Advances in Neural Information Processing Systems*, 2022.

Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., et al.. “*Exploring the limits of transfer learning with a unified text-to-text transformer.*” *Journal of Machine Learning Research*, 2020.

# References (3/4)

Rajani, N., Tunstall, L., Beeching, E., Lambert, N., Rush, A., et al.. “*No Robots*.” *Hugging Face repository*, 2023.

Sanh, V., Webson, A., Raffel, C., Bach, S., Sutawika, L., et al.. “*Multitask Prompted Training Enables Zero-Shot Task Generalization*.” *International Conference on Learning Representations*, 2022.

Sutton, R., and Barto, A.. “*Reinforcement Learning: An Introduction*.” *MIT Press*, 2018. [link]

Taori, R., Gulrajani, I., Zhang, T., Dubois, Y., Li, X., et al.. “*Stanford Alpaca: An Instruction-following LLaMA Model*.” *GitHub repository*, 2023.

Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., et al.. “*Llama 2: Open foundation and fine-tuned chat models*.” *arXiv preprint arXiv:2307.09288*, 2023.

Tunstall, L., Beeching, E., Lambert, N., Rajani, N., Rasul, K., et al.. “*Zephyr: Direct Distillation of LM Alignment*.” *First Conference on Language Modeling*, 2024.

Wang, Y., Kordi, Y., Mishra, S., Liu, A., Smith, N., et al.. “*Self-instruct: Aligning language models with self-generated instructions*.” *Annual Meeting of the Association for Computational Linguistics (ACL)*, 2023.

Wang, Y., Ivison, H., Dasigi, P., Hessel, J., Khot, T., et al.. “*How Far Can Camels Go? Exploring the State of Instruction Tuning on Open Resources*.” *Advances in Neural Information Processing Systems*, 2023. [link]

# References (4/4)

Wei, J., Bosma, M., Zhao, V., Guu, K., Yu, A., et al.. "*Finetuned Language Models are Zero-Shot Learners*." *International Conference on Learning Representations*, 2022.

Zheng, L., Chiang, W., Sheng, Y., Zhuang, S., Wu, Z., et al.. "*Judging LLM-as-a-judge with MT-Bench and Chatbot Arena*." *Advances in Neural Information Processing Systems*, 2023.

Zhou, C., Liu, P., Xu, P., Iyer, S., Sun, J., et al.. "*LIMA: Less Is More for Alignment*." *Advances in Neural Information Processing Systems*, 2023.

Zhu, B., Jordan, M., and Jiao, J.. "*Principled reinforcement learning with human feedback from pairwise or k-wise comparisons*." *International Conference on Machine Learning*, 2023.

Zhu, B., Frick, E., Wu, T., Zhu, H., Ganesan, K., et al.. "*Starling-7B: Improving helpfulness and harmlessness with RLAIF*." *First Conference on Language Modeling*, 2024.