Reinforcement Learning from Human Feedback Basics

Nathan Lambert

13 August 2024

Abstract

Reinforcement learning from human feedback (RLHF) has become an important technical and storytelling tool to the deploy of the lastest machine learning systems. In this book, we hope to give a gentle introduction to the core methods for people with some level of quantitative background. The book starts with the origins of RLHF – both in recent literature and in a convergence of disparate fields of science in economics, philosophy, and optimal control. We then set the stage with definitions, problem formulation, data collection, and other common math used in the literature. We detail the detail the popular algorithms and future frontiers of RLHF.

Contents

1 Constitutional AI

2 Direct Alignment Algorithms

3 Evaluation

4 Instruction Tuning

5 Introduction

This is the first paragraph of the introduction chapter. This is a test of citing [1].

5.1 First: Images

This is the first subsection. Please, admire the gloriousnes of this seagull:

Figure 1: A cool seagull.

A bigger seagull:

Figure 2: A cool big seagull.

5.2 Second: Tables

This is the second subsection.

Please, check First: Images subsection.

Please, check this subsection.

Table 1: This is an example table.

Index	Name
0	AAA
1	BBB
...	...

5.3 Third: Equations

Formula example: $\mu = \sum_{i=0}^{N} \frac{x_i}{N}$

Now, full size:

$$\mu = \sum_{i=0}^{N} \frac{x_i}{N}$$

And a code sample:

```
def hello_world
  puts "hello world!"
end

hello_world
```

Check these unicode characters: áéß¢ð€đŋ

5.4 Fourth: Cross references

These cross references are disabled by default. To enable them, check the *Cross references* section on the README.md file.

Here's a list of cross references:

- Check fig. 3.
- Check tbl. 2.
- Check eq. 1.

A cool seagull

Figure 3: A cool seagull

$$y = mx + b \tag{1}$$

Table 2: This is an example table.

Index	Name
0	AAA
1	BBB
…	…

6 Over Optimization

7 Policy Gradient Algorithms

The algorithms that popularized RLHF for language models were policy-gradient reinforcement learning algoritms. These algorithms, such as PPO and

Reinforce, use recently generated samples to update their model rather than storing scores in a replay buffer. In this section we will cover the fundamentals of the policy gradient algorithms and how they are used in the modern RLHF framework.

For definitions of symbols, see the problem setup chapter.

7.1 Policy Gradient Algorithms

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

$$\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$$

7.1.1 Vanilla Policy Gradient

The vanilla policy gradient implementation optimizes the following expectation:

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

7.1.2 Reinforce

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

7.1.3 Proximal Policy Optimization

7.2 Computing Policy Gradients with a Language Model

7.3 Implementation Tricks

- Only score a response with a reward model with the `eos_token` is generated, otherwise the response is truncated.

TODO. Cite: https://spinningup.openai.com/en/latest/spinningup/rl_intro3.html#

https://lilianweng.github.io/posts/2018-04-08-policy-gradient/

8 Preference Data

9 Regularization

Throughout the RLHF optimization, many regularization steps are used to prevent over-optimization

9.1 KL Distances

9.1.1 Reference Policy

9.1.2 Reference Dataset

9.2 Likelihood Penalty

- https://arxiv.org/abs/2404.19733 on DPO loss

9.3 Reward Bonuses

- Nemotron

9.4 Margin Losses

- Llama 2
- Rebel
- Reward Preference Optimization (Nemotron)

10 Rejection Sampling

Rejection Sampling (RS) is a popular and simple baseline for performing preference fine-tuning. Rejection sampling operates by curating new candidate instructions, filtering them based on a trained reward model, and then fine-tuning the original model only on the top completions.

The name originates from computational statistics [2], where one wishes to sample from a complex distribution, but does not have a direct method to do so. To alleviate this, one samples from a simpler to model distribution and uses a heuristic to check if the sample is permissible. With language models, the target distribution is high-quality answers to instructions, the filter is a reward model, and the sampling distribution is the current model.

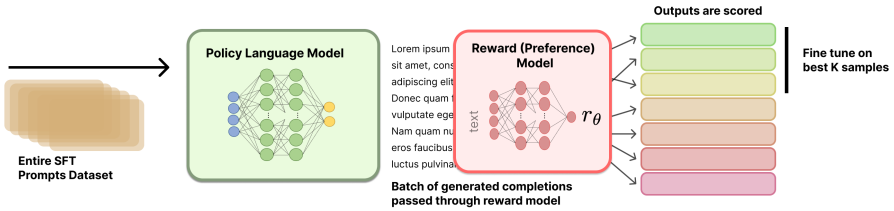
10.1 Related works

Many prominent RLHF and preference fine-tuning papers have used rejection sampling as a baseling, but a canonical implementation and documentation does not exist

WebGPT [3], Anthropic's Helpful and Harmless agent[4], OpenAI's popular paper on process reward models [5], Llama 2 Chat models [6], and other seminal works all use this baseline.

10.2 Training Process

A visual overview of the rejection sampling process is included below.



10.2.1 Generating Completions

Let's define a set of M prompts as a vector:

$$X = [x_1, x_2, ..., x_M]$$

These prompts can come from many sources, but most popularly they come from the instruction training set.

For each prompt x_i, we generate N completions. We can represent this as a matrix:

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

where $y_{i,j}$ represents the j-th completion for the i-th prompt. Now, we pass all of these prompt-completion pairs through a reward model, to get a matrix of rewards. We'll represent the rewards as a matrix R:

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

Each reward $r_{i,j}$ is computed by passing the completion $y_{i,j}$ and its corresponding prompt x_i through a reward model $\mathcal{R}$:

$$r_{i,j} = \mathcal{R}(y_{i,j}|x_i)$$

10.2.2 Selecting Top-N Completions

There are multiple methods to select the top completions to train on.

To formalize the process of selecting the best completions based on our reward matrix, we can define a selection function S that operates on the reward matrix R.

10.2.2.1 Top Per Prompt The first potential selection function takes the max per prompt.

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

This function S returns a vector of indices, where each index corresponds to the column with the maximum reward for each row in R. We can then use these indices to select our chosen completions:

$$Y_{chosen} = [y_{1,S(R)_1}, y_{2,S(R)_2}, ..., y_{M,S(R)_M}]$$

10.2.2.2 Top Overall Prompts Alternatively, we can select the top K prompt-completion pairs from the entire set. First, let's flatten our reward matrix R into a single vector:

$$R_{flat} = [r_{1,1}, r_{1,2}, ..., r_{1,N}, r_{2,1}, r_{2,2}, ..., r_{2,N}, ..., r_{M,1}, r_{M,2}, ..., r_{M,N}]$$

This R_{flat} vector has length $M \times N$, where M is the number of prompts and N is the number of completions per prompt.

Now, we can define a selection function S_K that selects the indices of the K highest values in R_{flat}:

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

where argsort returns the indices that would sort the array in ascending order, and we take the last K indices to get the K highest values.

To get our selected completions, we need to map these flattened indices back to our original completion matrix Y. We simply index the R_{flat} vector to get our completions.

10.2.2.3 Selection Example Consider the case where we have the following situation, with 5 prompts and 4 completions. We will show two ways of selecting the completions based on reward.

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

First, **per prompt**. Intuitively, we can highlight the reward matrix as follows:

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

Using the argmax method, we select the best completion for each prompt:

$$S(R) = [\arg\max_j r_{i,j} \text{ for } i \in [1, 4]]$$

$$S(R) = [1, 2, 1, 3, 4]$$

This means we would select:

- For prompt 1: completion 1 (reward 0.7)
- For prompt 2: completion 2 (reward 0.8)
- For prompt 3: completion 1 (reward 0.9)
- For prompt 4: completion 3 (reward 0.8)
- For prompt 5: completion 4 (reward 0.6)

Now, **best overall**. Let's highlight the top 5 overall completion pairs.

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

First, we flatten the reward matrix:

$$R_{flat} = [0.7, 0.3, 0.5, 0.2, 0.4, 0.8, 0.6, 0.5, 0.9, 0.3, 0.4, 0.7, 0.2, 0.5, 0.8, 0.6, 0.5, 0.4, 0.3, 0.6]$$

Now, we select the indices of the 5 highest values:

$$S_5(R_{flat}) = [8, 5, 14, 0, 19]$$

Mapping these back to our original matrix:

- Index 8 → prompt 3, completion 1 (reward 0.9)
- Index 5 → prompt 2, completion 2 (reward 0.8)
- Index 14 → prompt 4, completion 3 (reward 0.8)
- Index 0 → prompt 1, completion 1 (reward 0.7)
- Index 19 → prompt 3, completion 4 (reward 0.7)

10.2.2.4 Implementation Example Here is a code snippet showing how the selection methods could be implemented.

```
import numpy as np

x = np.random.randint(10, size=10)
print(f"{x=}")
sorted_indices = np.argsort(x)
x_sorted = x[sorted_indices]
print(f"{x_sorted=}")

# first way to recover the original array
i_rev = np.zeros(10, dtype=int)
i_rev[sorted_indices] = np.arange(10)
np.allclose(x, x_sorted[i_rev])

# second way to recover the original array
np.allclose(x, x_sorted[np.argsort(sorted_indices)])
```

10.2.3 Fine-tuning

With the selected completions, you then perform standard instruction fine-tuning on the current rendition of the model. More details can be found in the chapter on instruction tuning.

10.2.4 Details

Implementation details for rejection sampling are relatively sparse. The core hyperparameters for performing this training are very intuitive:

- **Sampling parameters**: Rejection sampling is directly dependent on the completions received from the model. Common settings for RS include tempetures above zero, e.g. between 0.7 and 1.0, with other modifications to paramters such as top-p or top-k sampling.
- **Completions per prompt**: Successful implementations of rejection sampling have included 10 to 30 or more completions for each prompt. Using too few completions will make training biased and or noisy.
- **Instruction tuning details**: No clear training details for the instruction tuning during RS have been released. It is likely that they use slightly different settings than the initial instruction tuning phase of the model.

- **Hetergenous model generations**: Some implementations of rejection sampling include generations from multiple models rather than just the current model that is going to be trained. Best practices on how to do this are not established.
- **Reward model training**: The reward model used will heavily impact the final result. For more resources on reward model training, see the relevant chapter.

10.2.4.1 Implementation Tricks

- When doing batch reward model inference, you can sort the tokenized completions by length so that the batches are of similar lengths. This eliminates the need to run inference on as many padding tokens and will improve throughput in exchange for minor implementation complexity.

10.3 Related: Best-of-N Sampling

Best-of-N (BoN) sampling is often included as a baseline relative to RLHF methods. It is important to remember that BoN *does not* modify the underlying model, but is a sampling technique. For this matter, comparisons for BoN sampling to online training methods, such as PPO, is still valid in some contexts. For example, you can still measure the KL distance when running BoN sampling relative to any other policy.

Here, we will show that when using simple BoN sampling over one prompt, both selection criteria shown above are equivalent.

Let R be a reward vector for our single prompt with N completions:

$$R = [r_1, r_2, ..., r_N]$$

Where r_j represents the reward for the j-th completion.

Using the argmax method, we select the best completion for the prompt:

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

Using the Top-K method is normally done with Top-1, reducing to the same method.

11 Key Related Works

In this chapter we detail the key papers and projects that got the RLHF field to where it is today. This is not intended to be a comprehensive review on RLHF and the related fields, but rather a starting point and retelling of how we got to today.

11.1 Early RL on Preferences

Christriano et al etc

11.2 RLHP on Language Models

Learning to summarize, first work on language models (zieglar et al)

11.3 Pre Modern Models

InstructGPT, WebgGPT, Sparrow, Etc

11.4 ChatGPT

12 Reward Modeling

13 Problem Setup

This chapter includes all the definitions, symbols, and operatings frequently used in the RLHF process.

13.1 ML Definitions

- **Kullback-Leibler (KL) divergence ($D_{KL}(P||Q)$)**, also known as KL divergence, is a measure of the difference between two probability distributions. For discrete probability distributions P and Q defined on the same probability space $\mathcal{X}$, the KL distance from Q to P is defined as:

$$D_{KL}(P||Q) = \sum_{x \in \mathcal{X}} P(x) \log \left(\frac{P(x)}{Q(x)} \right)$$

13.2 NLP Definitions

- **Prompt (x)**: The input text given to a language model to generate a response or completion.
- **Completion (y)**: The output text generated by a language model in response to a prompt. Often the completion is denoted as $y|x$.
- **Chosen Completion (y_c)**: The completion that is selected or preferred over other alternatives, often denoted as y_{chosen}.
- **Preference Relation ($\succ$)**: A symbol indicating that one completion is preferred over another, e.g., $y_{chosen} \succ y_{rejected}$.
- **Policy (π)**: A probability distribution over possible completions, parameterized by θ: $\pi_\theta(y|x)$.

13.3 RL Definitions

- **Reward (r)**: A scalar value indicating the desirability of an action or state, typically denoted as r.
- **Action (a)**: A decision or move made by an agent in an environment, often represented as $a \in A$, where A is the set of possible actions.
- **State (s)**: The current configuration or situation of the environment, usually denoted as $s \in S$, where S is the state space.
- **Trajectory (τ)**: A trajectory is a sequence of states, actions, and rewards experienced by an agent: $\tau = (s_0, a_0, r_0, s_1, a_1, r_1, ..., s_T, a_T, r_T)$.
- **Trajectory Distribution ($(\tau|\pi)$)**: The probability of a trajectory under policy π is $P(\tau|\pi) = p(s_0) \prod_{t=0}^{T} \pi(a_t|s_t) p(s_{t+1}|s_t, a_t)$, where $p(s_0)$ is the initial state distribution and $p(s_{t+1}|s_t, a_t)$ is the transition probability.
- **Policy (π)**: In RL, a policy is a strategy or rule that the agent follows to decide which action to take in a given state: $\pi(a|s)$.
- **Value Function (V)**: A function that estimates the expected cumulative reward from a given state: $V(s) = \mathbb{E}[\sum_{t=0}^{\infty} \gamma^t r_t | s_0 = s]$.
- **Q-Function (Q)**: A function that estimates the expected cumulative reward from taking a specific action in a given state: $Q(s, a) = \mathbb{E}[\sum_{t=0}^{\infty} \gamma^t r_t | s_0 = s, a_0 = a]$.
- **Advantage Function (A)**: The advantage function $A(s, a)$ quantifies the relative benefit of taking action a in state s compared to the average action. It's defined as $A(s, a) = Q(s, a) - V(s)$. Advantage functions (and value functions) can depend on a specific policy, $A^{\pi}(s, a)$.
- **Expectation of Reward Optimization**: The primary goal in RL, which involves maximizing the expected cumulative reward:

  $\max_{\theta} \mathbb{E}_{s \sim \rho_{\pi}, a \sim \pi_{\theta}}[\sum_{t=0}^{\infty} \gamma^t r_t]$

  where ρ_{π} is the state distribution under policy π, and γ is the discount factor.
- **Finite Horizon Reward ($J(\pi_{\theta})$)**: The expected finite-horizon undiscounted return of the policy π_{θ}, parameterized by θ is defined as: $J(\pi_{\theta}) = E_{\tau\ \pi_{\theta}}[\sum_{t}^{T} = 0 r_t]$ where $\tau\ \pi_{\theta}$ denotes trajectories sampled by following policy π_{θ} and T is the finite horizon.

14 Synthetic Data

Bibliography

[1] N. Lambert, T. K. Gilbert, and T. Zick, "Entangled preferences: The history and risks of reinforcement learning and human feedback," *arXiv preprint arXiv:2310.13595*, 2023.

[2] W. R. Gilks and P. Wild, "Adaptive rejection sampling for gibbs sampling," *Journal of the Royal Statistical Society: Series C (Applied Statistics)*, vol. 41, no. 2, pp. 337–348, 1992.

[3] R. Nakano *et al.*, "Webgpt: Browser-assisted question-answering with human feedback," *arXiv preprint arXiv:2112.09332*, 2021.

[4] Y. Bai *et al.*, "Training a helpful and harmless assistant with reinforcement learning from human feedback," *arXiv preprint arXiv:2204.05862*, 2022.

[5] H. Lightman *et al.*, "Let's verify step by step," *arXiv preprint arXiv:2305.20050*, 2023.

[6] H. Touvron *et al.*, "Llama 2: Open foundation and fine-tuned chat models," *arXiv preprint arXiv:2307.09288*, 2023.