Pandoc-X: Book

Examples and experiments with Pandoc

Author One Author Two

14 February 2026

Abstract

Your summary goes here.
And here is another paragraph of the summary.

Contents

1 Introduction

This PDF is an example of using Pandoc to build LaTex/PDF documents from Markdown sources.

It has all the features of book or article formats ready for publishing: title, author, abstract, table of contents (with links), numbered chapters and sections, bibliography (with references), appendix, code blocks with syntax highlighting and line numbers, as well as equations, figures, and tables with cross-references (and links).

At the same time, the source Markdown files are straightforward and readable with Markdown Preview showing properly rendered equations, figures, and tables. That makes source Markdown files to be proper documentation in itself while serving as the source for the PDF.

This example is based on pandoc-book-template.

It also borrows freely from the approach used by Nathan Lambert for his book "Reinforcement Learning from Human Feedback" in https://github.com/natolambert/rlhf-book/tree/main/book.

As a side note, the "Reinforcement Learning from Human Feedback" by Nathan Lambert https://rlhfbook.com/ is an excellent book written by a top expert in the field of RLHF. It is not just very informative but also a delight to read. Read this book to get a sense of where the present cutting-edge AI research is going.

1.1 Implementation Example

Implementing the reward modeling loss is quite simple. More of the implementation challenge is on setting up a separate data loader and inference pipeline. Given the correct dataloader with tokenized, chosen, and rejected prompts with completions, the loss is implemented as:

```
import torch.nn as nn

# inputs_chosen / inputs_rejected include the prompt tokens x and the respective
# completion tokens (y_c or y_r) that the reward model scores jointly.
rewards_chosen = model(**inputs_chosen)
rewards_rejected = model(**inputs_rejected)

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

As for the bigger picture, this is often within a causal language model that has an additional head added (and learned with the above loss) that transitions from the final hidden state to the score of the inputs. This model will have a structure as follows:

```
import torch
import torch.nn as nn
import torch.nn.functional as F


class BradleyTerryRewardModel(nn.Module):
    """
    Standard scalar reward model for Bradley-Terry preference learning.
```

```
    Usage (pairwise BT loss):
        rewards_chosen = model(**inputs_chosen)     # (batch,)
        rewards_rejected = model(**inputs_rejected)  # (batch,)
        loss = -F.logsigmoid(rewards_chosen - rewards_rejected).mean()
    """

    def __init__(self, base_lm):
        super().__init__()
        self.lm = base_lm  # e.g., AutoModelForCausalLM
        self.head = nn.Linear(self.lm.config.hidden_size, 1)

    def _sequence_rep(self, hidden, attention_mask):
        """
        Get a single vector per sequence to score.
        Default: last non-padding token (EOS token); if no mask, last token.
        hidden: (batch, seq_len, hidden_size)
        attention_mask: (batch, seq_len)
        """

        # Index of last non-pad token in each sequence
        # attention_mask is 1 for real tokens, 0 for padding
        lengths = attention_mask.sum(dim=1) - 1  # (batch,)
        batch_idx = torch.arange(hidden.size(0), device=hidden.device)
        return hidden[batch_idx, lengths]  # (batch, hidden_size)

    def forward(self, input_ids, attention_mask):
        """
        A forward pass designed to show inference structure of a standard reward model.
        To train one, this function will need to be modified to compute rewards from both
         chosen and rejected inputs, applying the loss above.
        """
        outputs = self.lm(
            input_ids=input_ids,
            attention_mask=attention_mask,
            output_hidden_states=True,
            return_dict=True,
        )
        # Final hidden states: (batch, seq_len, hidden_size)
        hidden = outputs.hidden_states[-1]

        # One scalar reward per sequence: (batch,)
        seq_repr = self._sequence_rep(hidden, attention_mask)
        rewards = self.head(seq_repr).squeeze(-1)

        return rewards
```

2 Training Overview

In this chapter we provide a cursory overview of RLHF training, before getting into the specifics later in the book. RLHF, while optimizing a simple loss function, involves training multiple, different AI models in sequence and then linking them together in a complex, online optimization.

Here, we introduce the core objective of RLHF, which is optimizing a proxy reward for human preferences with a distance-based regularizer (along with showing how it relates to classical RL problems). Then we showcase canonical recipes which use RLHF to create leading models to show how RLHF fits in with the rest of post-training methods. These example recipes will serve as references for later in the book, where we describe different optimization choices you have when doing RLHF, and we will point back to how different key models used different steps in training.

2.1 Problem Formulation

The optimization of reinforcement learning from human feedback (RLHF) builds on top of the standard RL setup. In RL, an agent takes actions a_t sampled from a policy $\pi(a_t \mid s_t)$ given the state of the environment s_t to maximize reward $r(s_t, a_t)$ [1]. Traditionally, the environment evolves according to transition (dynamics) $p(s_{t+1} \mid s_t, a_t)$ with an initial state distribution $\rho_0(s_0)$. Together, the policy and dynamics induce a trajectory distribution:

$$p_\pi(\tau) = \rho_0(s_0) \prod_{t=0}^{T-1} \pi(a_t \mid s_t)\, p(s_{t+1} \mid s_t, a_t). \tag{1}$$

Across a finite episode with horizon T, the goal of an RL agent is to solve the following optimization:

$$J(\pi) = \mathbb{E}_{\tau \sim p_\pi} \left[\sum_{t=0}^{T-1} \gamma^t r(s_t, a_t) \right], \tag{2}$$

For continuing tasks, one often takes $T \to \infty$ and relies on discounting ($\gamma < 1$) to keep the objective well-defined. γ is a discount factor from 0 to 1 that balances the desirability of near-term versus future rewards. Multiple methods for optimizing this expression are discussed in Chapter 6.

A standard illustration of the RL loop is shown in fig. 1 (compare this to the RLHF loop in fig. 4).

2.1.1 A Simple Example: The Thermostat

To build a basic intuition for what RL does, consider a thermostat trying to keep a room at a target temperature of 70°F. In RL, the agent starts with no knowledge of the task and must discover a good policy through trial and error. The thermostat example has the following components (see fig. 2 for how each maps to the trajectory distribution in eq. 1):

- **State (s_t)**: the current room temperature, e.g. 65°F.

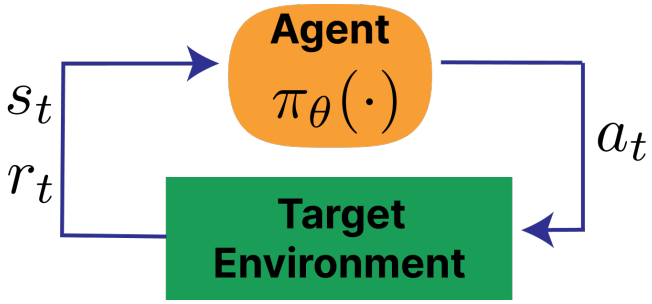

Figure 1: Standard RL loop

- **Action** (a_t): turn the heater on or off.
- **Reward** (r): $+1$ when the temperature is within $2°$ of the target, 0 otherwise.
- **Policy** (π): the rule that decides whether to turn the heater on or off given the current temperature. An example policy, which may not be optimal depending on the exact transition dynamics of the environment:

$$\pi(a_t = \text{on} \mid s_t) = \begin{cases} 1 & \text{if } s_t < 70°\text{F} \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

- **Transition**: the room warms when the heater is on and cools when it is off – this is the environment dynamics that the agent cannot control directly.

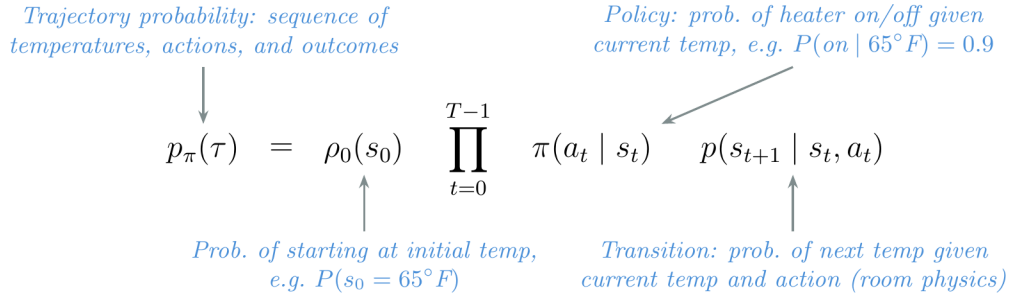

Figure 2: Each term in the trajectory distribution (eq. 1) mapped to the thermostat RL example.

Initially, the thermostat's policy is essentially random – it flips the heater on and off with no regard for the current temperature, and the room swings wildly. Over many episodes of trial and error, the agent discovers that turning the heater on when the room is cold and off when it is warm leads to more reward, and gradually converges on a sensible policy. This is the core RL loop: observe a state, choose an action, receive a reward, and update the policy to get more reward over time.

2.1.2 Example RL Task: CartPole

For a richer example with continuous dynamics, consider the classic *CartPole* (inverted pendulum) control task, which appears in many RL textbooks, courses, and even research

papers. Where the thermostat had a single state variable and a binary action, CartPole involves four continuous state variables and physics-based transitions – making it a standard benchmark for RL algorithms.

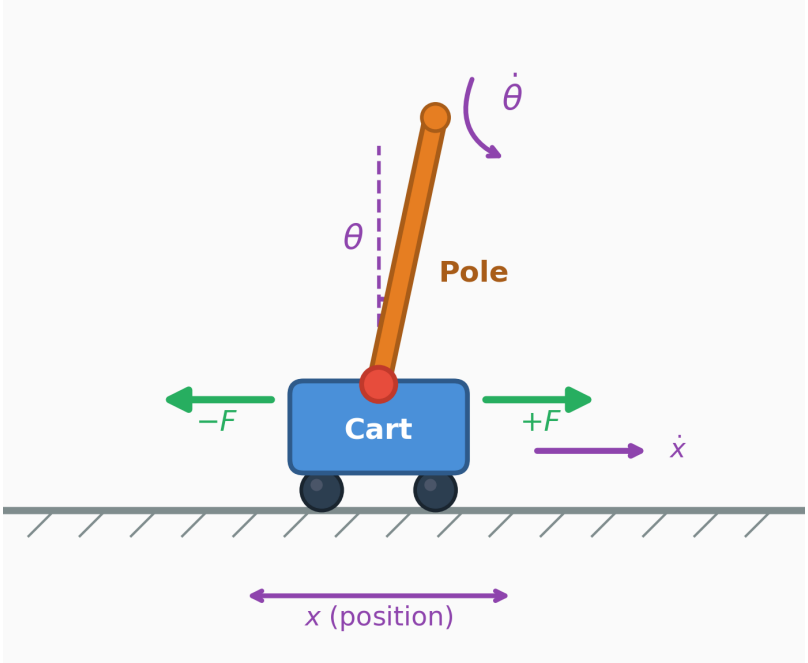

Figure 3: CartPole environment showing state variables (x, $\dot{x}$, θ, $\dot{\theta}$) and actions ($\pm F$).

- **State (s_t)**: the cart position/velocity and pole angle/angular velocity,

$$s_t = (x_t,\, \dot{x}_t,\, \theta_t,\, \dot{\theta}_t). \tag{4}$$

- **Action (a_t)**: apply a left/right horizontal force to the cart, e.g. $a_t \in \{-F, +F\}$.
- **Reward (r)**: a simple reward is $r_t = 1$ each step the pole remains balanced and the cart stays on the track (e.g. $|x_t| \leq 2.4$ and $|\theta_t| \leq 12°$), and the episode terminates when either bound is violated.
- **Dynamics / transition ($p(s_{t+1} \mid s_t, a_t)$)**: in many environments the dynamics are deterministic (so p is a point mass) and can be written as $s_{t+1} = f(s_t, a_t)$ via Euler integration with step size Δt. A standard simplified CartPole update uses constants cart mass m_c, pole mass m_p, pole half-length l, and gravity g:

$$\text{temp} = \frac{a_t + m_p l\, \dot{\theta}_t^2 \sin\theta_t}{m_c + m_p} \tag{5}$$

$$\ddot{\theta}_t = \frac{g \sin\theta_t - \cos\theta_t\, \text{temp}}{l\left(\frac{4}{3} - \frac{m_p \cos^2\theta_t}{m_c + m_p}\right)} \tag{6}$$

$$\ddot{x}_t = \text{temp} - \frac{m_p l\, \ddot{\theta}_t \cos\theta_t}{m_c + m_p} \tag{7}$$

$$x_{t+1} = x_t + \Delta t\, \dot{x}_t, \quad \dot{x}_{t+1} = \dot{x}_t + \Delta t\, \ddot{x}_t, \tag{8}$$

$$\theta_{t+1} = \theta_t + \Delta t\, \dot{\theta}_t, \quad \dot{\theta}_{t+1} = \dot{\theta}_t + \Delta t\, \ddot{\theta}_t. \tag{9}$$

This is a concrete instance of the general setup above: the policy chooses a_t, the transition function advances the state, and the reward is accumulated over the episode.

2.1.3 Manipulating the Standard RL Setup

The RL formulation for RLHF is seen as a less open-ended problem, where a few key pieces of RL are set to specific definitions in order to accommodate language models. There are multiple core changes from the standard RL setup to that of RLHF: Table tbl. 1 summarizes these differences between standard RL and the RLHF setup used for language models.

1. **Switching from a reward function to a reward model.** In RLHF, a learned model of human preferences, $r_\theta(s_t, a_t)$ (or any other classification model) is used instead of an environmental reward function. This gives the designer a substantial increase in the flexibility of the approach and control over the final results, but at the cost of implementation complexity. In standard RL, the reward is seen as a static piece of the environment that cannot be changed or manipulated by the person designing the learning agent.
2. **No state transitions exist.** In RLHF, the initial states for the domain are prompts sampled from a training dataset and the "action" is the completion to said prompt. During standard practices, this action does not impact the next state and is only scored by the reward model.
3. **Response level rewards.** Often referred to as a bandit problem, RLHF attribution of reward is done for an entire sequence of actions, composed of multiple generated tokens, rather than in a fine-grained manner.

Table 1: Key differences between standard RL and RLHF for language models.

Aspect	Standard RL	RLHF (language models)
Reward signal	Environment reward function $r(s_t, a_t)$	Learned reward / preference model $r_\theta(x, y)$ (prompt x, completion y)
State transition	Yes: dynamics $p(s_{t+1} \mid s_t, a_t)$	Typically no: prompts x sampled from a dataset; the completion does not define the next prompt
Action	Single environment action a_t	A completion y (a sequence of tokens) sampled from $\pi_\theta(\cdot \mid x)$
Reward granularity	Often per-step / fine-grained	Usually response-level (bandit-style) over the full completion
Horizon	Multi-step episode ($T > 1$)	Often single-step ($T = 1$), though multi-turn can be modeled as longer-horizon

Given the single-turn nature of the problem, the optimization can be re-written without the time horizon and discount factor (and with an explicit reward model):

$$J(\pi) = \mathbb{E}_{\tau\sim\pi}\left[r_\theta(s_t, a_t)\right]. \tag{10}$$

In many ways, the result is that while RLHF is heavily inspired by RL optimizers and problem formulations, the actual implementation is very distinct from traditional RL.

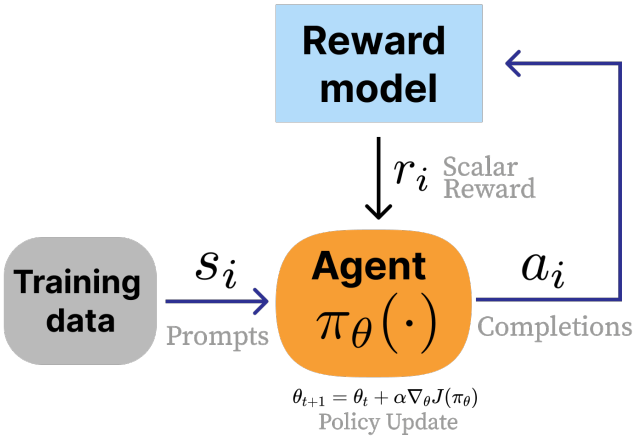

Figure 4: Standard RLHF loop

Bibliography

[1] R. S. Sutton, "Reinforcement learning: An introduction," *A Bradford Book*, 2018.

[2] A. Vaswani *et al.*, "Attention is all you need," in *Neural information processing systems*, 2017. Available: https://api.semanticscholar.org/CorpusID:13756489

[3] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," in *International conference on learning representations (ICLR)*, 2015. Available: https://arxiv.org/abs/1409.0473

A Definitions & Background

This chapter includes all the definitions, symbols, and operations frequently used in the RLHF process, with a quick overview of language models, which is the guiding application of this book.

A.1 Language Modeling Overview

The majority of modern language models are trained to learn the joint probability distribution of sequences of tokens (words, subwords, or characters) in an autoregressive manner. Autoregression simply means that each next prediction depends on the previous entities in the sequence. Given a sequence of tokens $x = (x_1, x_2, \ldots, x_T)$, the model factorizes the probability of the entire sequence into a product of conditional distributions:

$$P_\theta(x) = \prod_{t=1}^{T} P_\theta(x_t \mid x_1, \ldots, x_{t-1}). \tag{11}$$

In order to fit a model that accurately predicts this, the goal is often to maximize the likelihood of the training data as predicted by the current model. To do so, we can minimize a negative log-likelihood (NLL) loss:

$$\mathcal{L}_{\text{LM}}(\theta) = -\,\mathbb{E}_{x\sim\mathcal{D}}\left[\sum_{t=1}^{T} \log P_\theta\left(x_t \mid x_{<t}\right)\right]. \tag{12}$$

In practice, one uses a cross-entropy loss with respect to each next-token prediction, computed by comparing the true token in a sequence to what was predicted by the model.

Language models come in many architectures with different trade-offs in terms of knowledge, speed, and other performance characteristics. Modern LMs, including ChatGPT, Claude, Gemini, etc., most often use **decoder-only Transformers** [2]. The core innovation of the Transformer was heavily utilizing the **self-attention** [3] mechanism to allow the model to directly attend to concepts in context and learn complex mappings. Throughout this book, particularly when covering reward models in Chapter 5, we will discuss adding new heads or modifying a language modeling (LM) head of the transformer. The LM head is a final linear projection layer that maps from the model's internal embedding space to the tokenizer space (a.k.a. vocabulary). We'll see in this book that different "heads" of a language model can be applied to fine-tune the model to different purposes – in RLHF this is most often done when training a reward model, which is highlighted in Chapter 5.