

# OpenRLHF: An Easy-to-use, Scalable and High-performance RLHF Framework

Jian Hu<sup>1</sup>, Xibin Wu<sup>2</sup>, Weixun Wang<sup>3</sup>, Xianyu<sup>1</sup>, Dehao Zhang<sup>1</sup>, Yu Cao<sup>4</sup>  
 OpenLLMAI Team<sup>1</sup>, ByteDance Inc.<sup>2</sup>, Netease Fuxi AI Lab<sup>3</sup>, Alibaba Group<sup>4</sup>  
 janhu9527@gmail.com, xianyuai@openllmai.top

## Abstract

As large language models (LLMs) continue to grow by scaling laws, reinforcement learning from human feedback (RLHF) has gained significant attention due to its outstanding performance. However, unlike pretraining or fine-tuning a single model, scaling reinforcement learning from human feedback (RLHF) for training large language models poses coordination challenges across four models. We present OpenRLHF, an open-source framework enabling efficient RLHF scaling. Unlike existing RLHF frameworks that co-locate four models on the same GPUs, OpenRLHF re-designs scheduling for the models beyond 70B parameters using Ray, vLLM, and DeepSpeed, leveraging improved resource utilization and diverse training approaches. Integrating seamlessly with Hugging Face, OpenRLHF provides an out-of-the-box solution with optimized algorithms and launch scripts, which ensures user-friendliness. OpenRLHF implements RLHF, DPO, rejection sampling, and other alignment techniques. Empowering state-of-the-art LLM development, OpenRLHF’s code is available at <https://github.com/OpenLLMAI/OpenRLHF>.

## 1 Introduction

Although large language models (LLM) have shown remarkable performance improvements following scaling laws, a critical challenge that intensifies with is aligning these models with human values and intentions. Reinforcement Learning from Human Feedback (RLHF) [18] has emerged as a powerful technique to address this longstanding challenge. However, as models grow larger, vanilla RLHF typically requires maintaining multiple models and a more complex learning pipeline, leading to increased demands for memory and computational resources. To illustrate, Proximal Policy Optimization (PPO) [22, 18], a commonly used algorithm in RLHF, requires maintaining four models during training. As a result, as language models increase in size beyond 70 billion parameters, the computational resources and scheduling complexity required for training and coordinating these multiple models grow significantly, posing new demands and challenges for current framework designs.

Existing open-source RLHF frameworks such as Transformer Reinforcement Learning (TRL), ColossalChat (CAIChat), and DeepSpeed-Chat (DSChat) rely on parallelization approaches like Zero Redundancy Optimizer (ZeRO) to co-locate the four models involved in RLHF training on the same GPU [14, 28, 20]. However, as models continue to grow past 70 billion parameters, this scheduling approach becomes increasingly inefficient with limited GPU memory. To address the limitations of co-location, some frameworks like TRL compromise on memory usage by merging the actor and critic models or employing techniques like Low-Rank Adaptation (LoRA) [10]. However, these can reduce model performance, and the merged actor-critic architecture is incompatible with the recommended practice of initializing the critic model’s weights using those of the reward model [18]. An alternative solution for large models is to leverage tensor parallelism and pipeline parallelism techniques from NVIDIA Megatron [24]. However, Megatron is not compatible with the popular

Hugging Face library [27], and adapting new models requires extensive source code modifications, hindering usability.

To enable easy RLHF training at scale, OpenRLHF redesigns model scheduling using Ray [17], vLLM [13], and DeepSpeed [21], enabling training of models beyond 70 billion parameters. OpenRLHF seamlessly integrates with Hugging Face Transformers [27] and supports popular technologies such as Mixture of Experts (MoE) [12], Jamba [16] and QLoRA [4]. Furthermore, OpenRLHF implements multiple alignment algorithms, including Direct Preference Optimization (DPO) [19], Kahneman-Tversky optimization (KTO) [9], conditional SFT [11], and rejection sampling [26], providing an accessible full-scale RLHF training framework. Table 1 compares popular RLHF frameworks.

Features		OpenRLHF	DSChat	CAIChat	TRL
Full Fine-tuning Model Size	70B PPO	✓	Limited	✗	✗
	7B PPO with 4 RTX4090	✓	✗	✗	✗
	34B DPO with 8 A100-80G	✓	✗	✗	✗
Ease of Use	Compatible with HuggingFace	✓	✓	✓	✓
Training Techniques	QLoRA	✓	✗	✗	✓
	MoE in PPO	✓	✗	✗	✗
	Jamba in DPO	✓	✗	✗	✓
	Unmerged Actor-Critic	✓	✓	✓	✗
	Inference Engine in PPO	✓	✓	✗	✗
Alignment Algorithms	PPO Implementation Tricks	✓	✗	✗	✓
	Multiple Reward Models	✓	✗	✗	✗
	DPO	✓	✗	✓	✓
	KTO	✓	✗	✗	✓
	Rejection Sampling	✓	✗	✗	✓
	Conditional SFT	✓	✗	✗	✗

Table 1: RLHF frameworks comparison. OpenRLHF supports multi-reward models using Ray, and accelerates popular HuggingFace models using vLLM. The compatibility with the Hugging Face library ensures the framework’s user-friendliness. **Limited:** DSChat’s HybridEngine only supports a limited range of model architectures, such as <https://github.com/microsoft/DeepSpeed/issues/4954>. In contrast, OpenRLHF supports all mainstream architectures including MoE using DeepSpeed and vLLM, see the docs [https://docs.vllm.ai/en/latest/models/supported\\_models.html](https://docs.vllm.ai/en/latest/models/supported_models.html).

## 2 Background

### 2.1 Reinforcement Learning from Human Feedback

The classic training of large language models [18] based on a pre-trained Generative Pre-trained Transformer (GPT) involves three steps: Supervised Fine-tuning (SFT), Reward Model (RM) training, and PPO training:

- Supervised Fine-tuning: The developers fine-tuned a GPT model on human demonstrations from their labelers using a supervised learning loss, as shown in Eq.1.

$$\text{loss}(\phi) = - \sum_i \log p_\phi(x_i | inst, x_{<i}), \quad (1)$$

where  $x_i$  is the  $i_{th}$  token in the sequence,  $inst$  is the human instructions and prompts, and  $\phi$  is the parameter of the model.

- Reward Model training: Starting from the SFT model with the final unembedding layer removed, the developers trained a model to take in a prompt and response and output a scalar reward. Specifically, the loss function for the reward model is given by Eq.2,

$$\text{loss}(\theta) = -\mathbb{E}_{(x, y_w, y_l) \sim D} [\log(\sigma(r_\theta(x, y_w) - r_\theta(x, y_l)))], \quad (2)$$

where  $r_\theta(x, y)$  is the scalar output of the reward model with parameters  $\theta$  for prompt  $x$  and response  $y$ ,  $y_w$  is the preferred response out of the pair of  $y_w$  and  $y_l$ , and  $D$  is the dataset of human comparisons.

- PPO training: The developers fine-tuned the language model on their bandit environment using Proximal Policy Optimization (PPO). In this environment, a random customer prompt is presented, and a response is expected. The environment then produces a reward determined by the reward model and ends the episode, given the prompt-response pair. Additionally, a per-token Kullback-Leibler (KL) divergence penalty from the SFT model is added at each token to mitigate over-optimization of the reward model. Initializing the value function from the RM weights provides a stable starting point for Reinforcement Learning (RL) fine-tuning. The loss function of PPO is shown in Eq.3.

$$loss(\phi) = -E_{(x,y) \sim D_{\pi^{\phi^{RL}}}} [r\theta(x, y) - \beta \log (\pi^{\phi^{RL}}(y | x) / \pi^{SFT}(y | x))] \quad (3)$$

where  $\pi^{RL}$  is the learned reinforcement learning policy,  $\pi^{SFT}$  is the supervised fine-tuned model, and  $\beta$  is the KL reward coefficient that controls the strength of the KL penalty.

## 2.2 Ray

Ray [17] is a distributed execution framework that provides powerful scheduling and scaling capabilities for parallel and distributed computing workloads. It employs a built-in distributed scheduler to efficiently assign tasks across available resources in a cluster, enabling seamless scaling from a single machine to large-scale deployments with thousands of nodes. Ray’s scheduling mechanism intelligently manages task parallelism, distributing computations as smaller tasks that can be executed concurrently across multiple cores and machines. Ray’s scalable architecture and adept scheduling make it well-suited for accelerating a wide range of data-intensive workloads spanning machine learning, scientific computing, and high-performance data processing pipelines. It provides the compute layer for parallel processing so that users don’t need to be a distributed systems expert.

## 2.3 vLLM

vLLM [13] is a fast and easy-to-use library for LLM inference and serving. It delivers state-of-the-art serving throughput through efficient management of attention key and value memory with PagedAttention, continuous batching of incoming requests, and fast model execution with CUDA graph. vLLM’s flexibility and ease of use are evident in its seamless integration with popular Hugging Face models, high-throughput serving with various decoding algorithms, tensor parallelism support for distributed inference, and streaming outputs. It supports experimental features like prefix caching and multi-LoRA support. vLLM seamlessly supports the most popular open-source models on HuggingFace, including Transformer-like LLMs (e.g., Llama), Mixture-of-Expert LLMs (e.g., Mixtral [12]).

## 2.4 DeepSpeed

DeepSpeed [21] is an optimization library designed to enhance the efficiency of large-scale deep-learning models. Its Zero Redundancy Optimizer (ZeRO) [20] significantly reduces memory consumption by partitioning model states, gradients, and optimizer states across data-parallel processes, enabling the training of models with trillions of parameters. Additionally, DeepSpeed’s offloading features seamlessly transfer data between CPU and GPU memory, further optimizing resource utilization and enabling efficient training of extensive models on hardware with limited GPU memory. DeepSpeed also seamlessly supports the most popular open-source models on HuggingFace.

# 3 Design of OpenRLHF

## 3.1 Scheduling Optimization

Scaling RLHF training to larger models requires efficiently allocating at least four component models (actor, critic, reward, reference) across multiple GPUs due to the memory limit of each accelerator (e.g., under 80 GB for NVIDIA A100). OpenRLHF innovates on model scheduling by leveraging the Ray [17] for model placement and fine-grained orchestration. Meanwhile, the Ray-based scheduler manages inference-optimized libraries such as vLLM [13] and training-optimized libraries such as deepspeed. OpenRLHF distributes the four models across multiple GPUs instead of co-locating them on the same GPU, as illustrated in Figure 1. This design naturally supports multi-reward models

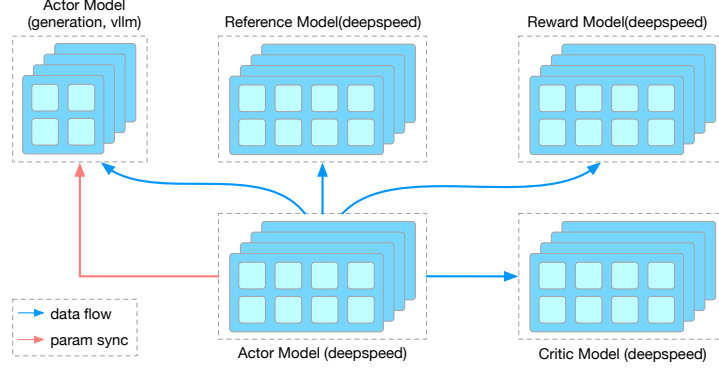


Figure 1: Ray Architecture of OpenRLHF. The four models in RLHF are distributed across different GPUs by Ray, which can also be freely merged or offloaded to save GPUs. The vLLM is used to accelerate actor generation. OpenRLHF synchronizes the weights of the ZeRO engine to the vLLM engine using the NVIDIA Collective Communications Library (NCCL).

[26] (as shown in Figure 2) during the RLHF training process for different algorithm implementation choices. Algorithm engineers could quickly build up various alignment strategies such as usefulness and harmfulness separation without concerning the detail of the underlying data flow.

Our scheduler design allows flexible model merging or offloading strategies using Ray and DeepSpeed. For example, the actor-reference or critic-reward models can be merged to save GPU resources. Besides the benefits of highly customizable algorithm implementation, the scheduler improves the overall training performance by optimally orchestrating GPUs. More details will be discussed in the next section, but scheduler optimization is the cornerstone for further efficiency improvements.

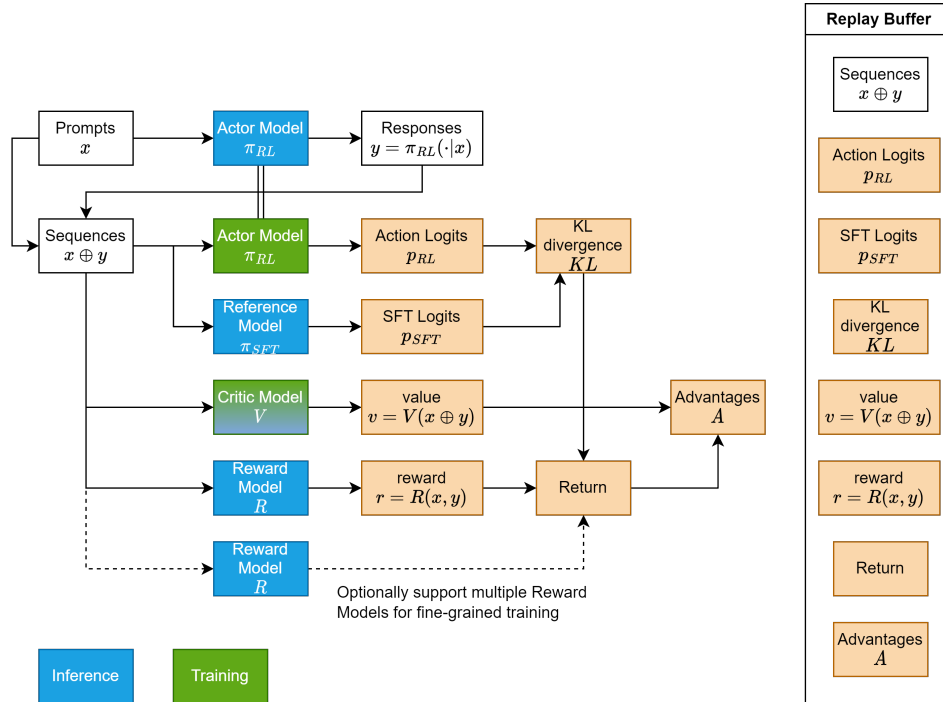


Figure 2: Flow diagram of RLHF in generation stage: the design of OpenRLHF supports flexible placement of multiple models with various algorithm implementations.

### 3.2 Performance Optimization

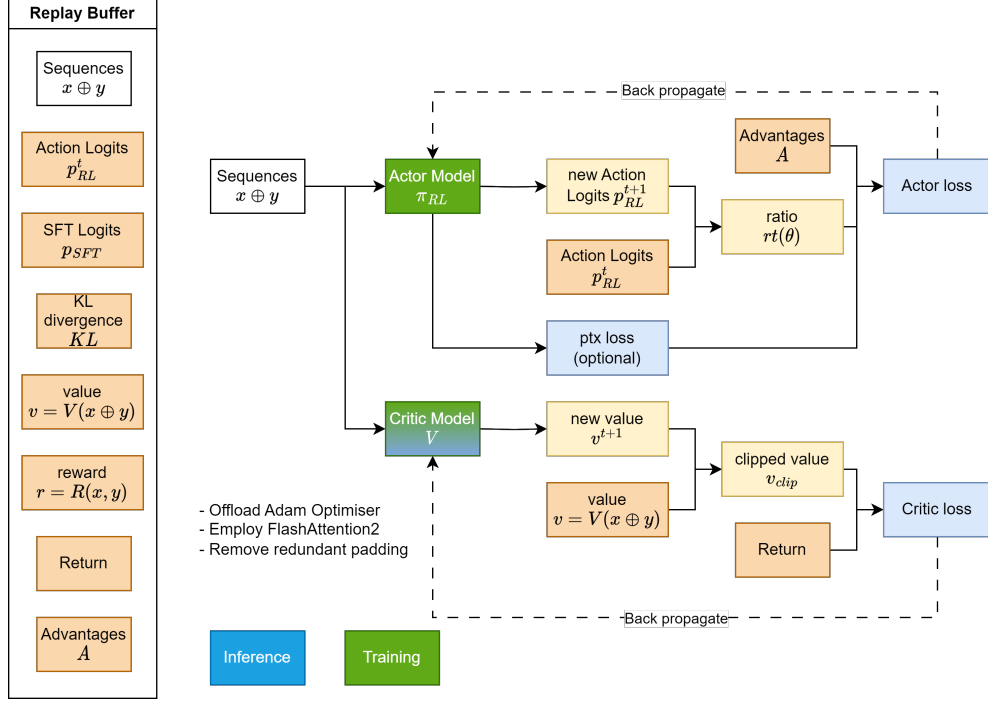


Figure 3: Flow diagram of RLHF in learning stage: two learnable models are scheduled by OpenRLHF to maximize the overall training throughput

The performance of RLHF algorithms depends on both training and inference efficiency. From the profiling result using LLaMA2 7B and NVIDIA A100 (as shown in Figure 4a), the primary bottleneck is at the PPO sample generation stage which takes up 80 % of overall training time. This is because, in the generation stage, the autoregressive decoding complexity is at  $O(n^2)$  and memory-bound. Figure 4b shows that the larger inference batch size can significantly improve the generation throughput. RLHF Frameworks such as DeepSpeedChat and TRL that share GPUs across all models lead to insufficient available memory during the generation stage, preventing an increase in batch size and exacerbating the issue of low memory access efficiency. OpenRLHF distributes the four models across multiple GPUs using Ray, effectively alleviating the problem.

To further accelerate sample generation and support larger LLMs such as 70B models that can't located in a single GPU, OpenRLHF leverages vLLM's tensor parallelism and other advanced techniques (e.g., continuous batching and paged attention [13]) for the generation, as shown in Figure 1. In the RLHF learning stage, OpenRLHF also employs the following techniques as additional improvements, see Figure 3:

- Offloading Adam optimizer states to the CPU frees up GPU memory, allowing for larger batch sizes during generation (without vLLM) and training. This approach enhances computational efficiency and reduces ZeRO communication costs. Pinned memory and gradient accumulation are applied to mitigate GPU-CPU communication overhead during gradient aggregation.
- Employing Flash Attention 2 [3] accelerates Transformer model training.
- Remove redundant padding from training samples using PyTorch tensor slicing.

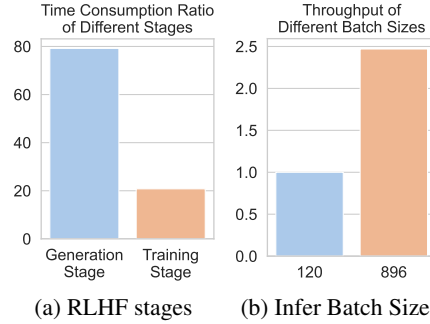


Figure 4: Performance Profiling using LLaMA2 7B and NVIDIA A100.

The remaining three models shown in Figure 2 use ZeRO stage 3 (sharding the model, gradients, and optimizer). OpenRLHF synchronizes the weights between the ZeRO and vLLM engines using NVIDIA NCCL and the vLLM weight loader, ensuring fast and simple integration. We compare the performance of OpenRLHF with our carefully tuned DSChat in Section 4.1.

### 3.3 PPO Implementation Tricks

Reinforcement learning (RL) algorithms like PPO can be prone to instability when training large language models (LLMs). We have verified the implementation details to our best effort and the general inference and learning procedure has been shown in Figure 2 and 3 for reference. Furthermore, OpenRLHF applies several tricks to stabilize training in the PPO implementation [7] including

- Predict reward only on the end-of-text token of the sequence.
- Use token-level reinforcement learning for language models.
- Use Kullback–Leibler (KL) divergence loss term in PPO.
- Use pre-trained loss term in PPO, tuned based on a relative scale of the policy loss.
- Apply reward normalization for training stability.
- Apply distributed advantage normalization with global statistics.
- Use the Linear Warmup Cosine Annealing learning rate scheduler.

### 3.4 Ease of Use

For user-friendliness, OpenRLHF provides one-click trainable scripts for supported algorithms, fully compatible with the Hugging Face library for specifying model and dataset names or paths. Below is the configuration for RLHF training of the 70B model on 16 A100s:

```

1 ray job submit -- python3 examples/train_ppo_ray.py \
2   --ref_num_gpus_per_node 4 \           # Number of GPUs for Ref model
3   --reward_num_gpus_per_node 4 \        # Number of GPUs for RM
4   --critic_num_gpus_per_node 4 \        # Number of GPUs for Critic
5   --actor_num_gpus_per_node 4 \         # Number of GPUs for Actor
6   --vllm_num_engines 4 \                # Number of vLLM engines
7   --vllm_tensor_parallel_size 2 \       # vLLM Tensor Parallel Size
8   --colocate_actor_ref \                # Colocate Actor and Ref
9   --colocate_critic_reward \            # Colocate Critic and RM
10  --ref_reward_offload \                 # Offload Ref and RM
11  --pretrain {HF Model name or path after SFT} \
12  --reward_pretrain {HF Reward model name or path} \
13  --zero_stage 3 \                       # DeepSpeed ZeRO stage
14  --bf16 \                               # Enable BF16
15  --init_kl_coef 0.01 \                  # KL penalty coefficient
16  --prompt_data {HF Prompt dataset name or path} \
17  --input_key {Prompt dataset input key}
18  --normalize_reward \                   # Enable Reward Normalization
19  --adam_offload \                       # Offload Adam Optimizer
20  --flash_attn \                         # Enable Flash Attention
21  --save_path {Model output path}

```

Listing 1: PPO startup method based on deepspeed (with Ray)

We provide usage scripts for all supported algorithms in the Appendix A.

## 4 Experiments

### 4.1 Performance Benchmark

In Tables 3 and 4, we present the configurations used for the RLHF performance experiments. We train the LLaMA2 models using NVIDIA A800 GPUs with NVLINK for data transfer. We optimized DSChat’s performance to the greatest extent possible by employing techniques such as enabling

Size	NVIDIA A800 GPUs	Optimized DSChat	OpenRLHF	Speedup
7B	16	855.09	471.11	1.82x
13B	32	1528.93	608.93	2.5x
34B	32	3634.98	1526.4	2.4x
70B	32	10407.0	4488.53	2.3x

Table 2: The average time (seconds) it took to train 1024 prompts with 1 PPO epoch using the Optimized DSChat and OpenRLHF.

Size	Total GPUs	Actor	Critic	Ref	RM	vLLM Engine
7B	16	4	4	2	2	DP=4, TP=1, MBS=16
13B	32	8	8	4	4	DP=8, TP=1, MBS=8
34B	32	8	8	4	4	DP=4, TP=2, MBS=8
70B	32	4	4	4	4	DP=4, TP=4, MBS=4

Table 3: OpenRLHF configurations. The numbers in the table indicate the number of GPUs assigned to each model. **We enabled Adam offload** for all models. DP: data parallel size, TP: tensor prallesim size, MBS: micro batch size (per GPU)

Adam offload, along with reward model (RM) and reference model (Ref) offload to increase the micro-batch size during the inference stage and avoid out-of-memory issues. We even fixed some bugs in DSChat to enable the Hybrid Engine (HE) for LLaMA2 <sup>1</sup>.

Table 5 and 6 illustrate the detailed time consumption of each stage of PPO in OpenRLHF and Optimized DSChat, respectively. Despite not fully optimizing the PPO performance of OpenRLHF, such as consolidating the actor and reference models nodes, as well as the critic and reward models nodes to reduce GPU resources, the experimental results demonstrate that OpenRLHF still exhibits considerable performance advantages in Table 2.

The performance advantages of OpenRLHF over DSChat primarily stem from vLLM and Ray. On the one hand, the generation acceleration of vLLM is substantially better than that of Hybrid Engine. On the other hand, Ray distributes the model across different nodes, which may seem to reduce GPU utilization. However, it avoids excessive model splitting and model weights offloading, thereby saving GPU memory and reducing communication overhead. This allows for an increase in the micro-batch size per GPU and the size of matrix multiplication in tensor parallelism, improving overall performance.

## 4.2 Training Stability and Convergence

We then evaluated the training convergence of the OpenRLHF framework based on the LLaMA2 7B model, where the supervised fine-tuning (SFT) stage utilized a 50k dataset from OpenOrca <sup>2</sup>, and the preference dataset for training the reward model was a mixture of around 200k samples from the Anthropic HH <sup>3</sup>, LMSys Arena <sup>4</sup>, and Open Assistant <sup>5</sup> datasets. The Proximal Policy Optimization (PPO) training used 80k prompts randomly sampled from these previous datasets. Direct Policy Optimization (DPO) training used the same dataset as the reward model training.

To stabilize the PPO training, we use a low learning rate  $5e^{-7}$  for the actor model and a higher learning rate  $9e^{-6}$  for the critic model. The PPO epoch is set to 1, the rollout batch size is 1024, the clip range is 0.2, and the mini-batch size is 128. For DPO, we set the learning rate to  $5e^{-7}$ , batch size to 128, and  $\beta$  to 0.1.

Thanks to the implementation tricks and previous hyper-parameters for PPO, Figure 5 shows the PPO training curves, where the reward and return value rise steadily, and the Kullback–Leibler (KL)

<sup>1</sup>such as <https://github.com/microsoft/DeepSpeedExamples/issues/864>

<sup>2</sup><https://huggingface.co/datasets/Open-Orca/OpenOrca>

<sup>3</sup><https://github.com/anthropics/hh-rlhf>

<sup>4</sup>[https://huggingface.co/datasets/lmsys/chatbot\\_arena\\_conversations](https://huggingface.co/datasets/lmsys/chatbot_arena_conversations)

<sup>5</sup><https://huggingface.co/datasets/OpenAssistant/oasst1>

Size	Total GPUs	Optimizer	Reward and Reference models	Hybrid Engine
7B	16	Adam Offload, Pinned Memory	ZeRO-3, Param Offload, Pinned Memory	DP=16, TP=1, MBS=8
13B	32			DP=32, TP=1, MBS=4
34B	32			DP=4, TP=8, MBS=4
70B	32			DP=4, TP=8, MBS=2

Table 4: DSChat configurations. DP: data parallel size, TP: tensor prallesim size, MBS: micro batch size (per GPU).

Size	GPUs	Generation	vLLM Weights Sync	Get Logits, Reward	Training	Total
7B	16	262.96	4.32	32.7	171.13	471.11
13B	32	372.14	10.03	29.58	197.18	608.93
34B	32	720.50	35.47	326.00	444.43	1526.40
70B	32	2252.79	111.65	323.38	1800.71	4488.53

Table 5: The time consumption (seconds) of each stage of PPO to train 1024 prompts with 1 PPO epoch in OpenRLHF.

divergence and loss values remain stable. Table 7 presents the evaluation results on AlpacaEval [15]. The winning rates indicate that the PPO model and DPO model outperform the SFT model, and the PPO model outperforms the DPO model. This may be because DPO is more sensitive to out-of-distribution samples.

## 5 Conclusion

We introduce OpenRLHF, an open-source framework enabling full-scale RLHF training of models with over 70B parameters, by distributing models across GPUs via Ray and optimizing efficiency leveraging vLLM. OpenRLHF also implements multiple alignment algorithms. Seamless integration with HuggingFace provides out-of-the-box usability.



Size	GPUs	Generation	HE Weights Sync	Get Logits, Reward	Training	Total
7B	16	590.157	65.573	73.68	125.68	855.09
13B	32	1146.614	156.356	87.28	138.68	1528.93
34B	32	1736.024	434.006	443.12	1021.83	3634.98
70B	32	3472.68	1157.56	2013.44	3763.32	10407.00

Table 6: The time consumption (seconds) of each stage of PPO to train 1024 prompts with 1 PPO epoch in Optimized DSChat.

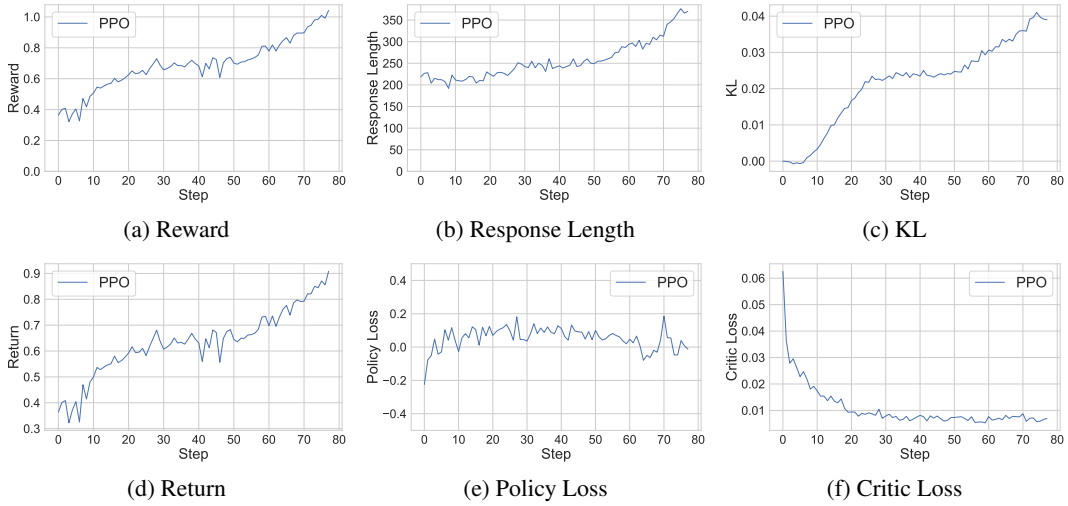


Figure 5: PPO training curves.

Algorithm	Win Rate (%)	Standard Error	Average Length
PPO vs SFT	63.52	3.83	2165
DPO vs SFT	60.06	3.83	1934

Table 7: AlpacaEval results using GPT-4. We evaluated the win rates of PPO and DPO against the SFT model using the 160 prompts from the MT Bench and Vicuna Bench [29].

## References

- [1] Mohammad Gheshlaghi Azar, Mark Rowland, Bilal Piot, Daniel Guo, Daniele Calandriello, Michal Valko, and Rémi Munos. A general theoretical paradigm to understand learning from human preferences. *arXiv preprint arXiv:2310.12036*, 2023.
- [2] Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemyslaw Debiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, et al. Dota 2 with large scale deep reinforcement learning. *arXiv preprint arXiv:1912.06680*, 2019.
- [3] Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning. *arXiv preprint arXiv:2307.08691*, 2023.
- [4] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. Qlora: Efficient finetuning of quantized llms. *Advances in Neural Information Processing Systems*, 36, 2024.
- [5] Hanze Dong, Wei Xiong, Deepanshu Goyal, Yihan Zhang, Winnie Chow, Rui Pan, Shizhe Diao, Jipeng Zhang, Kashun Shum, and Tong Zhang. Raft: Reward ranked finetuning for generative foundation model alignment. *arXiv preprint arXiv:2304.06767*, 2023.
- [6] Yi Dong, Zhilin Wang, Makes Narsimhan Sreedhar, Xianchao Wu, and Oleksii Kuchaiev. Steerlm: Attribute conditioned sft as an (user-steerable) alternative to rlhf. *arXiv preprint arXiv:2310.05344*, 2023.
- [7] Logan Engstrom, Andrew Ilyas, Shibani Santurkar, Dimitris Tsipras, Firdaus Janoos, Larry Rudolph, and Aleksander Madry. Implementation matters in deep policy gradients: A case study on ppo and trpo. *arXiv preprint arXiv:2005.12729*, 2020.
- [8] Mitchell Eric. A note on dpo with noisy preferences and relationship to ipo. <https://ericmitchell1.ai/cdpo.pdf>, 2023. Accessed: November 25, 2023.
- [9] Kawin Ethayarajh, Winnie Xu, Niklas Muennighoff, Dan Jurafsky, and Douwe Kiela. Kto: Model alignment as prospect theoretic optimization. *arXiv preprint arXiv:2402.01306*, 2024.
- [10] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, 2021.
- [11] Jian Hu, Li Tao, June Yang, and Chandler Zhou. Aligning language models with offline reinforcement learning from human feedback. *arXiv preprint arXiv:2308.12050*, 2023.
- [12] Albert Q Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, et al. Mixtral of experts. *arXiv preprint arXiv:2401.04088*, 2024.
- [13] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.
- [14] Shenggui Li, Hongxin Liu, Zhengda Bian, Jiarui Fang, Haichen Huang, Yuliang Liu, Boxiang Wang, and Yang You. Colossal-ai: A unified deep learning system for large-scale parallel training. In *Proceedings of the 52nd International Conference on Parallel Processing*, pages 766–775, 2023.
- [15] Xuechen Li, Tianyi Zhang, Yann Dubois, Rohan Taori, Ishaan Gulrajani, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. AlpacaEval: An automatic evaluator of instruction-following models. [https://github.com/tatsu-lab/alpaca\\_eval](https://github.com/tatsu-lab/alpaca_eval), 2023.
- [16] Opher Lieber, Barak Lenz, Hofit Bata, Gal Cohen, Jhonathan Osin, Itay Dalmedigos, Erez Safahi, Shaked Meirom, Yonatan Belinkov, Shai Shalev-Shwartz, et al. Jamba: A hybrid transformer-mamba language model. *arXiv preprint arXiv:2403.19887*, 2024.
- [17] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. Ray: A distributed framework for emerging {AI} applications. In *13th USENIX symposium on operating systems design and implementation (OSDI 18)*, pages 561–577, 2018.
- [18] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to

- follow instructions with human feedback. *Advances in neural information processing systems*, 35:27730–27744, 2022.
- [19] Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. Direct preference optimization: Your language model is secretly a reward model. *Advances in Neural Information Processing Systems*, 36, 2024.
  - [20] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16. IEEE, 2020.
  - [21] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 3505–3506, 2020.
  - [22] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.
  - [23] Thomas Scialom, Paul-Alexis Dray, Sylvain Lamprier, Benjamin Piwowarski, and Jacopo Staiano. Discriminative adversarial search for abstractive summarization. In Hal Daumé III and Aarti Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 8555–8564. PMLR, 13–18 Jul 2020. URL <https://proceedings.mlr.press/v119/scialom20a.html>.
  - [24] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.
  - [25] Nisan Stiennon, Long Ouyang, Jeffrey Wu, Daniel Ziegler, Ryan Lowe, Chelsea Voss, Alec Radford, Dario Amodei, and Paul F Christiano. Learning to summarize with human feedback. *Advances in Neural Information Processing Systems*, 33:3008–3021, 2020.
  - [26] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.
  - [27] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online, October 2020. Association for Computational Linguistics. URL <https://www.aclweb.org/anthology/2020.emnlp-demos.6>.
  - [28] Zhewei Yao, Reza Yazdani Aminabadi, Olatunji Ruwase, Samyam Rajbhandari, Xiaoxia Wu, Ammar Ahmad Awan, Jeff Rasley, Minjia Zhang, Conglong Li, Connor Holmes, et al. Deepspeed-chat: Easy, fast and affordable rlhf training of chatgpt-like models at all scales. *arXiv preprint arXiv:2308.01320*, 2023.
  - [29] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Judging llm-as-a-judge with mt-bench and chatbot arena, 2023.

## A Appendix: Supported Algorithms And Usage

### A.1 Preliminary Preparation

#### A.1.1 Supervised FineTuning

Supervised Finetuning (SFT) is a method for guiding pre-trained models to conform to human instructions, primarily achieved through a specific series of data forms using supervised learning. This process allows the pre-trained language model, after fine-tuning, to accept specific input formats and produce the required outputs. Typically, Supervised Finetuning serves as the preceding step for alignment. To better support the RLHF pipeline, the OpenRLHF framework can support Supervised Finetuning for various pre-trained models, making it relatively straightforward to expand new models within OpenRLHF.

```
1 deepspeed train_sft.py \  
2   --pretrain {Pre-trained model name or path} \  
3   --dataset {SFT dataset name} \  
4   --input_key {SFT dataset input key} \  
5   --output_key {SFT dataset response key} \  
6   --zero_stage 2 \  
7   --bf16 \  
8   --flash_attn \  
9   --learning_rate 5e-6 \  
10  --save_path {Model output path}
```

Listing 2: SFT startup method based on deepspeed

#### A.1.2 Reward Model Training

Typically, reinforcement learning algorithms require a definitive reward signal to guide optimization. However, in many natural language domains, such a clear reward signal is often absent. [25] has collected human preference data and trained a reward model through comparative methods to obtain the necessary reward signals for the reinforcement learning process. Similar to the training stage of Supervised FineTuning (SFT), to better support the RLHF pipeline, the OpenRLHF framework supports the corresponding reward model training approach.

```
1 deepspeed train_rm.py \  
2   --pretrain {Model name or path after SFT} \  
3   --bf16 \  
4   --zero_stage 3 \  
5   --learning_rate 9e-6 \  
6   --dataset {Pairwise preference dataset name} \  
7   --chosen_key {Pairwise dataset chosen key} \  
8   --rejected_key {Pairwise dataset rejected key} \  
9   --flash_attn \  
10  --save_path {Model output path}
```

Listing 3: RM startup method based on deepspeed

### A.2 Proximal Policy Optimization

Proximal Policy Optimization (PPO) is a reinforcement learning algorithm widely used in fields such as gaming and robotic control[22, 2]. In the domain of large language model fine-tuning, [18] has improved the stability of model fine-tuning by incorporating an approximate KL divergence with the base model as part of the reward and leveraging PPO’s clipped surrogate objective function to optimize model outputs while constraining the magnitude of output variations. This approach avoids over-optimizing language models for reward scores, effectively enhancing the stability of the model fine-tuning process.

### A.3 Direct Preference Optimization

To address issues such as overoptimization during the reward model training process and to prevent the misjudgment of newly generated responses by the reward model, Direct Preference Optimization (DPO) has redesigned the alignment formula into a simple loss function [19], which allows direct optimization of model outputs on preference datasets. However, DPO suffers from overfitting on preference datasets. Identity Preference Optimization (IPO) introduces a regularization term based on the DPO loss to reduce the risk of overfitting [1]. [8] further discusses the differences between IPO and DPO and introduces conservative DPO (cDPO), achieving an effect similar to introducing a regularization term in IPO by smoothing the sample labels. OpenRLHF has implemented the above method to ensure the stability of the training process.

```
1 deepspeed train_dpo.py \  
2   --train_batch_size 128 \  
3   --micro_train_batch_size 1 \  
4   --pretrain {Model name or path after SFT} \  
5   --bf16 \  
6   --zero_stage 3 \  
7   --beta 0.1 \  
8   --dataset {Pairwise preference dataset name} \  
9   --chosen_key {Pairwise dataset chosen key} \  
10  --rejected_key {Pairwise dataset rejected key} \  
11  --flash_attn \  
12  --ref_offload \  
13  --save_path {Model output path}
```

Listing 4: DPO startup method based on deepspeed

### A.4 Kahneman-Tversky Optimization

Kahneman-Tversky Optimization (KTO) is a specific implementation of Human-Aware Loss Functions (HALOs). Unlike methods such as PPO and DPO, KTO individually labels each response as good or bad, defining the loss function based on this, thus requiring only a binary signal indicating whether the output for a given input is desirable. This approach avoids the challenges of collecting pairwise preference data. The implementation in OpenRLHF is consistent with the official open-source implementation of KTO [9].

```
1 deepspeed train_kto.py \  
2   --pretrain {Model name or path after SFT} \  
3   --bf16 \  
4   --zero_stage 3 \  
5   --beta 0.1 \  
6   --learning_rate 5e-7 \  
7   --dataset {(Non-Pairwise or Pairwise) Preference dataset name} \  
8   --prompt_key {Non-Pairwise dataset prompt key} \  
9   --output_key {Non-Pairwise dataset output key} \  
10  --label_key {Non-Pairwise dataset label key} \  
11  --flash_attn \  
12  --vanilla_loss \  
13  --save_path {Model output path}
```

Listing 5: KTO startup method based on deepspeed

### A.5 Rejection Sampling Finetuning

The approach for implementing Rejection Sampling (RS) Finetuning in OpenRLHF is similar to that in [26] and RAFT [5], where sampling from the model’s output and selecting the best candidates based on existing rewards is done. For each response, the sample with the highest reward score is considered the new gold standard. Similar to [23], after selecting the best response, OpenRLHF treats it as a positive sample and fine-tunes the current model, enabling the model’s output to achieve greater reward values.

```

1 python batch_inference.py
2   --eval_task generate_vllm \
3   --pretrain {Model name or path after SFT} \
4   --dataset {Prompt dataset name} \
5   --intpu_key {Prompt dataset input key}
6   --temperature 0.9
7   --tp_size 8
8   --best_of_n 16 \
9   --output_path {GENERATED_SAMPLES}

```

Listing 6: RS GENERATE step startup method based on vLLM

```

1 deepspeed batch_inference.py
2   --eval_task rm \
3   --pretrain {Reward model name or path} \
4   --bf16 \
5   --dataset {GENERATED_SAMPLES} \
6   --zero_stage 0 \
7   --post_processor rs \
8   --output_path {RS_FINETUNING_SAMPLES}

```

Listing 7: RS Score step startup method based on deepspeed

## A.6 Conditional Supervised Finetuning

By introducing the method of model fine-tuning using conditionals, we collectively refer to it as Conditional Supervised Finetuning[11, 6]. Taking SteerLM as an example[6], by using a dataset with multidimensional scoring, we refine the training objectives, and after fine-tuning, by controlling input conditions, the model outputs customized effects. OpenRLHF naturally supports the training needs of Conditional Supervised Finetuning by providing the corresponding Batch infer and naive Supervised Finetuning.