

POLICY FILTRATION IN RLHF TO FINE-TUNE LLM FOR CODE GENERATION

Anonymous authors

Paper under double-blind review

ABSTRACT

While direct policy optimization methods exist, pioneering LLMs are fine-tuned with reinforcement learning from human feedback (RLHF) to generate better responses under the supervision of a reward model learned from preference data. One major challenge of RLHF is the inaccuracy of the intermediate reward model, especially in code generation tasks that requires complex reasoning for the reward model to score a response. We find that the reliability of the reward model varies across responses assigned with different rewards. This motivates us to filter the samples whose rewards may be unreliable to improve the signal-to-noise ratio during policy learning, resulting in Policy Filtration for Proximal Policy Optimization (PF-PPO). To choose a proper policy filtering strategy, we use the coefficient of determination (R^2) between the rewards and actual scores on filtered samples as the metrics to help us find promising strategies since it measures how well the rewards filtered by PF-PPO indicate real performance. We provide extensive experiments to validate the effectiveness of PF-PPO in code generation tasks. We find that some variants of PF-PPO are highly effective and achieve the state-of-the-art performance of 7-billion-parameter models on HumanEval (+7.9%) and MBPP (+0.7%). Moreover, we create the LeetCode Contest benchmark and demonstrate the advantage of PF-PPO (+10.0%) on this more challenging benchmark.

1 INTRODUCTION

Reinforcement Learning from Human Feedback (RLHF) is a key technique to align large language models (LLMs) with human values and preferences (Christiano et al., 2017; Ziegler et al., 2019; Ouyang et al., 2022). RLHF has been proven to be an essential process for LLMs to produce more helpful, harmless, and honest responses (Bai et al., 2022). Despite various non-RL algorithms such as DPO (Rafailov et al., 2024) are proposed, state-of-the-art applications such as ChatGPT/GPT-4 (OpenAI, 2023), Claude (Anthropic, 2023), and Gemini (Team et al., 2023) adopt the RL algorithm (e.g., PPO) for policy optimization. The key challenge of RLHF is the inaccuracy of the intermediate reward model. While there are researchers investigate how to learn reliable reward models (see e.g., Wang et al., 2024), we focus on how to learn better policy under the guidance of such inaccurate reward models.

We observe that, though the reward model gives inaccurate rewards in general, it can be more reliable in specific regions (e.g., when it gives high rewards) than the others. The observation is based on the simple experiment: We use a policy model fine-tuned for code generation to generate a set of responses for prompts in the HumanEval dataset. Later, we score these responses using a reward model trained with the common recipe (see Ouyang et al., 2022, and also Section 2) and compare them with the actual scores. We find that, across different sets of samples, the reward model is more reliable when it gives high or low rewards than when it gives moderate rewards. ([This property also holds on other datasets and see Appendix A for more experiment results and futher discussion.](#)) Considering that RLHF updates the policy solely based on the reward signal, this observation motivates us to filter out the samples with possibly unreliable rewards aiming to improve RLHF by increasing the signal-to-noise ratio on training samples.

Based on this motivation, we propose a simple modification to the standard PPO-based RLHF algorithm (Ouyang et al., 2022), Policy Filtration for PPO (PF-PPO), that learns a filtered version of the policy using PPO. Specifically, we generate N samples for each prompt, score these samples

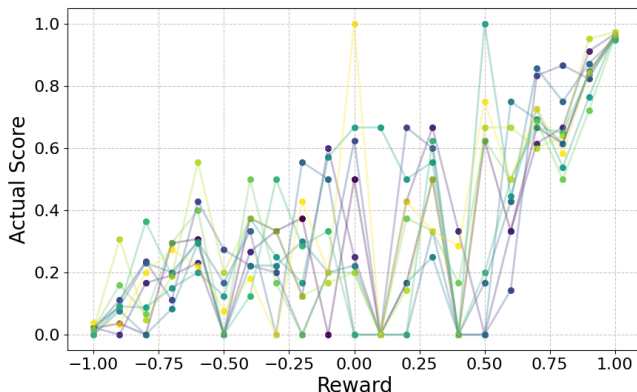


Figure 1: The reward model can be *inaccurate*, i.e., the actual score of the response does not align well with the reward given by the reward model. However, the reward model in specific regions (e.g., when it gives rewards higher than 0.8) is more *reliable*, i.e., the responses with similar rewards result in consistent performance. We use a fine-tuned policy to generate 10 responses for each of the 164 prompts in the HumanEval dataset and use a reward model trained with the common recipe to generate their rewards. We group the responses with similar rewards and calculate the average of their actual scores (i.e., the average correctness), indicating each group by one point. To evaluate the reliability of the reward model, we repeat the process ten times corresponding to the ten lines.

using the reward model, and use a filtered subset of these samples for subsequent policy training. We design filtering strategies to improve the reliability of the reward model on the filtered samples by maximizing the coefficient of determination (R^2) between the rewards and actual scores on these filtered samples. We show that the reward model can evaluate more accurately on these filtered samples, thus providing better training signal and improving the performance of the policy. Our method is also connected with reject sampling that filters out responses with low rewards during inference to yield a better response. Reject sampling is a simple but surprisingly strong inference-time strategy, whereas we adopt similar filtration in an RL algorithm.

Empirically, we show that PF-PPO can greatly improve the performance of LLMs on code generation tasks, which is challenging since complex logic behind these tasks makes the reward model inaccurate in general. We conduct extensive ablation studies to validate the design of our algorithm. Moreover, we illustrate the effectiveness of our algorithm by fine-tuning LLMs that achieves new sota on HumanEval and LeetCode Contest benchmarks across 7-billion-parameter LLMs. To evaluate whether PF-PPO can be effective on more challenging coding tasks, we create the LeetCode Contest benchmark that includes competition-level coding tasks for human experts. We find that the policy filtration technique can result in even more significant improvement on this challenging benchmark.

2 RELATED WORK

Limitation of reward model. The outcome of RLHF highly relies on the quality of the reward model. Unfortunately, the reward model can hardly provide accurate scores due to 1) the mis-specified reward modeling to represent human preferences (Lambert et al., 2023; Pitis, 2023); 2) the presence of incorrect and ambiguous preferences in the dataset (Ouyang et al., 2022; Bai et al., 2022), and 3) the poor generalization ability of the reward model (McKinney et al., 2023). The inaccuracy of reward model is attributed as one major cause of *reward hacking* and *hallucination* in LLMs (Kalai & Vempala, 2024). While there are previous papers try to improve the accuracy of the reward model itself (Wang et al., 2024; Coste et al., 2023; Zhang et al., 2024), the objective of our paper is to design a better RLHF algorithm in the face of inaccurate reward models.

Reject sampling. Reject sampling (or best-of- N sampling) is a popular and effective inference-time strategy to enhance the response of an LLM by generating N responses and select the best one according to a reward model (Nakano et al., 2021; Cobbe et al., 2021). This trick can yield good

responses while keeping a tight KL constraint to the original policy. Inspired by its effectiveness in inference, researchers also try to involve this trick in policy optimization. For example, RAFT (Dong et al., 2023), BOND (Sessa et al., 2024) and vBoN (Amini et al., 2024) learn a policy that distills the best-of- N policy using supervised fine-tuning losses. In a broader sense, the rank information of the N samples can also be leveraged. For example, RRHF (Yuan et al., 2023) and PRO (Song et al., 2024) train the policy using the combination of a ranking loss and a SFT loss (w.r.t. the best response) based on N responses for each prompt. However, these algorithms do not adopt an elaborate RL algorithm, while state-of-the-art language models adopt RL algorithms in alignment, benefiting from the generalization power of the reward model especially in reasoning tasks (Iverson et al., 2024). Unlike these algorithms, we adopt the idea of reject sampling in the sampling phase of an RL algorithm instead of using supervised learning losses.

RLHF algorithms in the face of inaccurate reward models. One key challenge in RLHF is the inaccuracy of reward model, which can lead to reward over-optimization (Gao et al., 2023; Skalse et al., 2022; Chaudhari et al., 2024). Optimization with a policy constraint (e.g., a KL divergence between the target policy and the reference policy) is a remedy frequently used in not only RL-based algorithms (Ouyang et al., 2022; Wu et al., 2023; Zhu et al., 2023) but also direct policy optimization algorithms (Rafailov et al., 2024; Zhao et al., 2023; Liu et al., 2023). Going beyond policy constraint, Moskovitz et al. (2023) only maximize rewards up to a threshold to avoid excessive deviation from a pre-trained policy. In this paper, we not only rely on the policy constraint to optimize in the face of inaccurate rewards but also try to avoid using samples with unreliable rewards.

3 PRELIMINARY

Notations. We use $[a, b]$ to denote the set $\{a, a + 1, \dots, b\}$ and use $[b]$ as the shorthand for $[1, b]$. We use \oplus to denote the concatenation on tokens, and use $x_{a:b}$ as the shorthand for the concatenation $(x_a \oplus x_{a+1} \oplus \dots \oplus x_b)$. We use c_i and y_i to indicate the i -th token in the context c (including task instruction, prompt, inputs, etc.) and the response y respectively.

MDP formulation. We adopt a Markov decision process (MDP) formulation for RLHF. Specifically, language generation is formulated as an MDP $M = (\mathcal{S}, \mathcal{A}, P, R)$ with states $s \in \mathcal{S}$, actions $a \in \mathcal{A}$, transition probabilities $P \in \Delta(\mathcal{S})^{\mathcal{S} \times \mathcal{A}}$, and the next-state-based reward function $R : \mathcal{S} \rightarrow [0, 1]$. Given a context c with T_c tokens, on each step $t \in [T_c + 1, T]^1$, the language model $\pi_\theta(a_t | s_t)$ selects a token $a_t = y_{t-T_c}$ based on the state $s_t := (c_{1:T_c} \oplus y_{1:t-T_c-1})$. Then, the language model enters the next state $s_{t+1} := (c_{1:T_c} \oplus y_{1:t-T_c})$ until the language model completes the response $y_{1:T-T_c}$. For simplicity, we will also use contextual-bandit-style notations, e.g., we denote the language generation process as $y \sim \pi_\theta(\cdot | c)$.

RLHF. Reinforcement learning with human feedback (RLHF) is an important process to address *objective mismatch* between the next-token-prediction objective in pre-training and our expectation of LLMs to follow the instructions and assist humans to complete various tasks. We briefly review the pipeline of RLHF.

- **Supervised fine-tuning.** In the supervised fine-tuning (SFT) phase, a pre-trained LLM is fine-tuned with a high-quality supervised dataset collected for specific downstream tasks. Typically, the LLM is fine-tuned with a maximum likelihood loss, and we denote the output of this phase as π^{SFT} . While subsequent RLHF procedure is necessary for training high-quality LLMs, this phase alone can also yield an LLM that reasonably follows human instructions (see e.g., Longpre et al., 2023).
- **Reward model learning.** In the reward model learning phase, we learn a reward model $r_\phi(y|c) \in [-1, 1]$ parameterized by ϕ that scores the response y to the context c based on collected preference data $\mathcal{D}_{\text{HF}} := \{(c, y^w, y^l)\}$ specifying that y^w is a preferred response to c than y^l . The reward model is initialized by π^{SFT} with an additional output layer. A preference model links the reward model with the preference data, and Bradley-Terry model (Bradley & Terry, 1952) is a common choice:

$$\mathbb{P}(y^w \succ y^l | c) = \sigma(R_\phi(y^w | c) - R_\phi(y^l | c)), \quad (1)$$

¹We fix the index of the terminal state to be the maximum length T . To adapt responses of different lengths, we left pad the context c .

where σ is the sigmoid function. The learning objective of reward model is to maximize the log-probability on preference data:

$$\max_{\phi} \mathbb{E}_{(c, y_w, y_l) \sim \mathcal{D}_{\text{HF}}} [\log \mathbb{P}(y_w \succ y_l | c)]. \quad (2)$$

- **RL fine-tuning.** In this stage, we fine-tune the language model π_{θ} to maximize the rewards given by the reward model with a policy constraint. The optimization problem is formulated as

$$\max_{\theta} \mathbb{E}_c \mathbb{E}_{y \sim \pi_{\theta}(\cdot|c)} [r_{\phi}(y|c) - \beta D_{\text{KL}}(\pi_{\theta}(\cdot|c) || \pi^{\text{SFT}}(\cdot|c))]. \quad (3)$$

The second term prevents the learned policy deviating too much from the SFT model, and this is a popular technique to alleviate reward over-optimization (Jaques et al., 2019; Stiennon et al., 2020).

PPO. Proximal policy optimization (PPO) (Schulman et al., 2017) is an RL algorithm that uses a clipped version of the policy gradient for more conservative and stable learning. It becomes a standard algorithm for RL fine-tuning in RLHF that optimizes the modified (cumulative) reward

$$r_{\phi}(y|c) - \sum_{t=T_c+1}^T \beta \left(\log \pi_{\theta}(y_t | c \oplus y_{1:t-1}) - \log \pi^{\text{SFT}}(y_t | c \oplus y_{1:t-1}) \right) \quad (4)$$

where the reward model gives sparse rewards and the policy constraint yields dense rewards. PPO is an on-policy algorithm where the policy gradient is estimated based on the samples collected by the current policy π_{θ} .

Algorithm 1 Proximal policy optimization (PPO)

for iteration = 1, 2, \dots **do**

 Fill the buffer \mathcal{B} with samples collected by the current language model π_{θ}

 Update π_{θ} using PPO w.r.t. the cumulative reward defined in Equation equation 4 based on \mathcal{B}

end for

4 METHODS

Our method is motivated by the observation that the reward model is more reliable for the responses assigned with high/low rewards (cf. Figure 1). Consequently, we conjecture that, if we wrap the policy with proper filtration during policy optimization of RLHF, the reward model can avoid yielding unreliable rewards and thus give better signal to guide policy learning.

Policy filtration. Given an unfiltered policy model $\pi_{\theta}(y|c)$ that generates responses y to the context c , we denote the corresponding filtered policy as $\mu_{\theta}(y|c)$. We consider a family of policy filtration, from which we can sample responses to the context c as follows: We first sample N responses from $\pi_{\theta}(\cdot|c)$ and rank them by the reward model R_{ϕ} , obtaining y_1, \dots, y_N with $R_{\phi}(y_1|c) \geq \dots \geq R_{\phi}(y_N|c)$. Then, given a weight vector $\mathbf{w} = (w_1, \dots, w_N)$ satisfying $\sum_{i \in [N]} w_i = 1$, we sample a one-hot vector $\mathbf{z} = (z_1, \dots, z_N)$ from the categorical distribution parameterized by \mathbf{w} such that $\mathbb{P}[z_i = 1] = w_i$. At last, the filtered policy $\mu_{\theta}(\cdot|c)$ yields the response selected by \mathbf{z} following $y = \sum_{i \in [N]} z_i y_i$.

We can define several filtered policies under this family. Specifically, we obtain the best-of- N (BoN), best-random (BR), and best-worst (BW) filtered policy by setting the weight vector to $\mathbf{w}^{\text{BoN}} = (1, 0, \dots, 0)$, $\mathbf{w}^{\text{BR}} = \left(\frac{1}{2}, \frac{1}{2(N-1)}, \dots, \frac{1}{2(N-1)} \right)$, and $\mathbf{w}^{\text{BW}} = \left(\frac{1}{2}, 0, \dots, 0, \frac{1}{2} \right)$ respectively.

Training objective. Since our target is to learn a good filtered policy μ_{θ} , we consider the follow objective:

$$\max_{\theta} \mathbb{E}_c \mathbb{E}_{y \sim \mu_{\theta}(\cdot|c)} [r_{\phi}(y|c) - \beta D_{\text{KL}}(\mu_{\theta}(\cdot|x) || \pi^{\text{SFT}}(\cdot|x))]. \quad (5)$$

In practice, use the samples collected by the filtered policy π_{θ} as if they were collected by μ_{θ} in the original PPO algorithm. This leads to Policy Filtration Proximal Policy Optimization (PF-PPO) listed in Algorithm 2, which is an algorithm that only modifies the sampling process of PPO.

Algorithm 2 Policy Filtration Proximal policy Optimization (PF-PPO)

```

for iteration = 1, 2, ... do
  Fill the buffer  $\mathcal{B}$  with samples collected by the current language model  $\mu_\theta$ 
  Update  $\pi_\theta$  using PPO w.r.t. the cumulative reward defined in Equation equation 4 based on  $\mathcal{B}$ 
end for

```

	No filter	BoN filter	BR filter	BW filter
SFT policy	0.886	0.454	0.922	0.952
Middle RLHF policy	0.907	0.389	0.935	0.956
Final RLHF policy	0.876	0.431	0.916	0.946

Table 1: The coefficient of determination (R^2) of unfiltered policy π_θ and different filtered policies μ_θ between the rewards given by the reward model and the actual scores. This metrics correlates well with the final performance (see Section 5) and helps us to determine the weight vector (or the policy filtering strategy) in our algorithm PF-PPO.

Weight choice. By defining different weight vectors \mathbf{w} , we can obtain different policy filtering strategies for PF-PPO. Our objective is to choose a weight vector \mathbf{w} such that the accuracy of the reward model on the responses generated by the filtered policies can be maximized. To measure this accuracy, we calculate the coefficient of determination (aka R-squared or R^2) (Draper, 1998) between the rewards and the actual scores of the responses generated by the policy. R^2 measures how well the actual scores can be predicted by the rewards with a linear model. Specifically, given a set of responses $\{(c_i, y_i)\}$ sampled from the filtered policy $y_i \sim \mu_\theta(\cdot|c_i)$, we can collect the corresponding reward $R_i := R_\phi(y_i|c_i)$ and the actual score s_i . Then, we fit a linear model f to predict the actual score based on the reward and denote the predicted score as $\hat{s}_i = f(R_i)$. The R-squared is calculated as $1 - \frac{\sum_i (s_i - \hat{s}_i)^2}{\sum_i (s_i - \bar{s})^2}$ where \bar{s} is the average of actual scores. Since PF-PPO optimizes the policy based on the rewards on these responses, how well these rewards indicate the actual performance is closely related to the final performance of our algorithm. We find R^2 well correlates with the final performance and can imply the level of reward over-optimization of the subsequent RLHF algorithm, therefore serving as a useful metrics to determine the weight vector used in PF-PPO.

To select a weight vector, we first checkpoint three policies π_θ collected from different stages of a standard RLHF process and collect responses using filtered policies μ_θ in combination with different policy filtering strategies. Then, we group the responses with similar rewards, record the average actual score and reward for each group, and calculate the R^2 by treating each group as a sample point. We exam how different policy filtering strategies can improve the reliability of the rewards on the responses generated by the corresponding filtered policies.

We present the results in Table 1. We observe that best-random (BR) and best-worst (BW) can improve the reliability of the given reward model on sampled responses compared with unfiltered policy. The BoN strategy does not improve the R^2 , which indicates that learning a BoN filtered policy may not result in good performance in RL, although learning for a best-of- N policy using supervised learning presents good performance (Sessa et al., 2024).

5 EXPERIMENTS

5.1 BENCHMARKS

To demonstrate the effectiveness of our method, we conduct experiments on the code generation task, which is a typical reasoning task where the quality of the responses from code LLMs can be precisely measured. Specifically, we compare different algorithms on two widely used benchmarks and a new challenging benchmark:

HumanEval benchmark and MBPP benchmark. HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021) are two popular benchmarks for evaluating code LLMs. HumanEval consists of 164 hand-written Python problems, each of which is validated using test cases to assess the accuracy of the code generated by a code LLM in a zero-shot setting. MBPP includes 378 test

270 problems, each of which includes the problem description, the standard code solution, and test cases
271 to help us evaluate the model’s ability to generate code. Both benchmarks play crucial roles These
272 two benchmarks are widely used to evaluate the performance of large language models on code
273 generation tasks.

274 **LeetCode contest benchmark.** To further evaluate the capability of the model on more challeng-
275 ing coding problems, we construct the LeetCode Contest benchmark. This benchmark includes
276 competition-level problems designed for human, and therefore is more challenging since it requires
277 human-level problem understanding and code generation skills. In this benchmark, we collect 160
278 problems from LeetCode weekly contests from July 2022 to January 2024. For each problem, we
279 include 100 test cases to ensure the generated code is assessed thoroughly.

281 5.2 DATASETS AND PRE-PROCESSING

282
283 For our experiments on the HumanEval and MBPP benchmarks, we select data from the 75k
284 Magicoder-OSS-instruct dataset (Wei et al., 2023b) and the 55k evol-codealpaca-v1 dataset (Luo
285 et al., 2023) to construct the SFT dataset, the reward model dataset, and the PPO query dataset.
286 Specifically, we use all the 130k training samples from Magicoder-OSS-instruct and evol-codealpaca-
287 v1 as the SFT dataset. To train a reward model, we curate 7k prompts from these 130k samples and
288 generate five responses using the SFT model for each prompt. Following the methodology in Pal
289 et al. (2024), we select two responses with the maximum edit distance to create response pairs for
290 each prompt. We use these 7k prompts with generated response pairs as the reward model dataset.
291 For policy optimization, we curate 3k prompts from the 130k samples as the PPO query dataset.

292 For the LeetCode benchmark, we construct LeetCode training datasets comprising 1,000 problems
293 collected from the LeetCode website. For SFT, we use self-generated correct answers to create the
294 SFT dataset following the methodology in Setlur et al. (2024). For reward modeling, we generate five
295 responses using the SFT model for each of the 400 curated prompts and selected two responses with
296 the maximum edit distance to form the response pairs for each prompt. We use these prompts and
297 response pairs to train the reward model. Finally, we used the full 1,000 prompts as our PPO query
298 dataset to train the code LLM.

299 5.3 IMPLEMENTATION DETAILS

300
301 We use deepseek-6.7B (Guo et al., 2024) as our base model. In the SFT phase, we train on the SFT
302 dataset for 5 epochs with the learning rate 1×10^{-5} , resulting in the SFT policy. In the reward model
303 training phase, we follow Ouyang et al. (2022) and train on our reward model dataset for 1 epoch
304 with the learning rate 1×10^{-5} . In the PPO phase, we adopt the training tricks from the blog (Shen
305 et al., 2024). Specifically, we adopt reward normalization and advantage normalization for stable
306 training. In addition, we set the learning rate for the policy network as 5×10^{-7} and learning rate for
307 the value network as 9×10^{-6} . In the PPO algorithm, we collect responses for the context in the PPO
308 query dataset and iterate through this dataset for 5 iterations (enough for convergence) and select the
309 best checkpoints on evaluation set as the outcome policy. For each collected context-response pair,
310 we use it to accumulate loss and gradient for 3 times on average. We use full parameter fine-tuning in
311 all the phases. We provide the source code for all experiments in the supplementary.

312 5.4 BASELINES

313
314 We compare different variants of PF-PPO with not only reinforcement learning algorithms but also
315 supervised fine-tuning methods and direct policy optimization methods. We use greedy decoding
316 during inference and pass@1 (Chen et al., 2021) as the performance metrics. For fair comparison
317 between different baselines, we re-implement these baselines with the same code base and the same
318 datasets. We also use the same reward model and the same SFT policy if applicable.

319 **Supervised fine-tuning.** Starting from deepseek-6.7B, we first fine-tune this policy on the SFT
320 dataset. Other algorithms learn based on this SFT policy. RAFT (Dong et al., 2023) and BOND (Sessa
321 et al., 2024) train the policy to fit the best-of- N (BoN) responses or the BoN policy via different
322 supervised learning losses. RAFT maximizes the log-probability of the BoN response, whereas
323 BOND minimizes a combination of the forward and backward KL divergence w.r.t. the BoN policy.
We set the coefficient to combine these two loss terms as $\beta_{\text{BOND}} = 1.0$. BOND is an iterative

Family	Method	HumanEval	MBPP	LeetCode
Supervised Fine-Tuning	SFT	74.2	70.8	15.2
	RAFT (Dong et al., 2023)	76.9	71.3	17.8
	BOND (Sessa et al., 2024)	80.8	75.2	30.0
Direct Policy Optimization	DPO (Rafailov et al., 2024)	78.4	73.7	23.0
	IPO (Azar et al., 2024)	78.2	72.9	23.2
	KTO (Ethayarajh et al., 2024)	77.9	72.5	22.4
	Iterative-DPO (Pang et al., 2024)	78.1	74.8	23.8
Reinforcement Learning	PPO-S (Hu et al., 2024)	78.1	73.8	25.2
	PPO-M (cf. Shao et al., 2024)	80.2	75.0	29.8
	PF-PPO (BoN)	75.8	71.7	16.8
	PF-PPO (BR)	82.9	<u>75.9</u>	33.0
	PF-PPO (BW)	<u>82.4</u>	76.2	<u>30.4</u>
SOTA (7B models)	Magocoder (Wei et al., 2023b)	76.8	75.7	

Table 2: **The performance of different algorithms on three benchmarks.** We compare pass@1 of PF-PPO (our algorithm) against baseline methods. For each benchmark, we select the best score across 5 epochs for each method. The highest and the second highest scores on each benchmark are highlighted in **bold** and underline respectively. All experiments are based on the same code base for fair comparison, except for the scores reported by Magocoder which is the best 7B model so far.

algorithm to fit the BoN policy based on the policy of the last iteration, and we train the policy for 4 iterations.

Direct policy optimization. To implement direct policy optimization methods, we use our reward model dataset as the preference dataset required in these methods. We implement DPO (Rafailov et al., 2024), IPO (Azar et al., 2024), KTO (Ethayarajh et al., 2024), and iterative DPO (Pang et al., 2024). For iterative DPO, we train the DPO model for three iterations. For each iteration, we construct the preference dataset as follows: The prompts are sampled from the reward model dataset and responses are generated by the trained DPO model from the previous iteration (if exists) or the previous SFT phase.

Reinforcement Learning. For standard RLHF, we use the implementation from OpenRLHF (Hu et al., 2024), which incorporates several advanced PPO training techniques and has demonstrates strong performance on various benchmarks. We denote this baseline as PPO-S. For our method PF-PPO, we implement three variants (BoN, BR, and BW) as introduced in the previous section. Since PF-PPO collects multiple responses given a prompt/context, we introduce a baseline called PPO-M (PPO with multiple responses) that uses all the N responses for training without filtering. Comparing with PPO-M can help us distinguish the effect of collecting multiple responses and that of filtering collected responses. The effective difference between PPO-S and PPO-M is that the buffer \mathcal{B} in PPO-M contains more samples with the same context c but with different responses y which may provide detailed token-level instruction by comparing the responses corresponding to the same context. PPO-M can also be regarded as integrating GRPO (Shao et al., 2024) into PPO, which has been adopted by Deepseek-V2 (Zhu et al., 2024) and Qwen2 (Yang et al., 2024). We also refer the readers to Section 5.7 for the analysis on the computational efficiency of PPO-S, PPO-M, and PF-PPO.

5.5 EXPERIMENT RESULTS ON THREE BENCHMARKS

We present the pass@1 results of different methods on the three benchmarks in Table 2. The experiment results show that PF-PPO (BR) and PF-PPO (BW) obtain the highest scores on these benchmarks, indicating the effectiveness of our method. Furthermore, we have the following observations:

- IPO and KTO (improved versions of DPO) do not outperform DPO when trained on properly selected datasets. This indicates that appropriate dataset construction can address the weaknesses of DPO found in previous papers, enabling DPO to achieve a performance comparable to its improved versions.

- 378 • PPO-based algorithms outperform SFT-based and DPO-based algorithms in general, demon-
379 strating that PPO is superior to these algorithms on reasoning tasks. We speculate that the
380 good performance of PPO may stem from the generalization ability of the reward model
381 and the value network used in PPO, which can be used to transform trajectory-level reward
382 modeling to token-wise advantages and thus provides more fine-grained guidance. More-
383 over, the gap between PPO-based algorithms and the others becomes larger on the more
384 challenging LeetCode benchmark, which further highlights the advantage of RL on complex
385 reasoning tasks
- 386 • BOND achieves the highest score among the baseline methods. It demonstrates that iterative
387 best-of- N (BoN) distillation is an effective alignment approach. We speculate that BOND
388 also benefits from its ability to reduce learning on samples with unreliable rewards by
389 selecting the best candidate from a set of N samples.
- 390 • Motivated by the good performance of BOND, we implement PF-PPO (BoN) as a natural
391 attempt to apply BoN to an RL-based algorithm. However, PF-PPO (BoN) results in poor
392 performance. This indicates that compared with SFT methods that only need good samples,
393 bad samples for the contrastive learning purposes are also important for RL-based methods.
394 This explains the reason why PF-PPO (BR) and PF-PPO (BW) outperform PF-PPO (BoN).
- 395 • PF-PPO (BR) and PF-PPO (BW) outperform the others with a larger gap challenging
396 LeetCode tasks. We find that the accuracy of the reward model decreases on this benchmark
397 since it is more difficult for the reward model to distinguish whether one response is better
398 than another, especially when both responses contain errors. This decreases the reliability
399 of the reward model in the moderate reward region (cf. Figure 1). Consequently, PF-PPO
400 (BR) and PF-PPO (BW) can improve the performance in these complex reasoning tasks by
401 avoiding learning on unreliable rewards.

402 5.6 CHOOSING FROM DIFFERENT POLICY FILTERING STRATEGIES

403 PF-PPO modifies the sampling procedure of standard PPO by sampling N responses and randomly
404 filtering responses based on their ranks. In this part, we consider other alternatives to filter by
405 threshold or down-weight the responses with unreliable rewards in the sampling procedure.

- 406 • **Filtering based on reward thresholds.** Given a reward model, we can filter the responses
407 based on their rewards using specified threshold. This results in three strategies, *PPO-top*
408 that only keeps the top samples whose rewards exceeding a certain threshold, *PPO-top-*
409 *random* that keeps also keeps random samples with 50% probability, and *PPO-top-bottom*
410 that keeps top samples and bottom samples whose rewards are below another specified
411 threshold. These strategies can be regarded as the threshold version of PF-PPO (BoN),
412 PF-PPO (BR) and PF-PPO (BW) respectively. The thresholds are tuned coarsely to achieve
413 good results on a separate validation set.
- 414 • **Filtering based on reward reweighting.** Compared with the above strategies that use
415 thresholds, we consider a softer version that adjusts the sample weights based on their
416 rewards, aiming at down-weight the samples with moderate and possibly unreliable rewards.
417 Specifically, we increase the sample weight of the responses with rewards in the reliable
418 region and decrease the sample weight otherwise. To achieve this goal, given a reward
419 model R_ϕ that returns rewards in the range $[-1, 1]$, we assign the weight for the sample
420 (c, y) proportional to $|R_\phi(y|c)|^k$ and collect samples with these weights from the buffer \mathcal{B} to
421 train the policy network and the value network. We denote these strategies as *PPO-pow-k*.

422 A question then arises: how to choose a policy filtering strategy from these strategies? To answer this
423 question, we propose to calculate the R^2 between the rewards and the actual scores on the samples
424 collected by different strategies, and then choose a strategy with good results on this metrics. We can
425 use the SFT policy as the unfiltered policy and calculate R^2 as described in Section 4. Since the SFT
426 policy is obtained prior to the PPO training phase, this metric can be used to predict the results of
427 different filtering strategies before actually conduct costly PPO training.

428 We compare theses strategies on HumanEval and present the performance of different policy filtering
429 strategies and their corresponding R^2 in Table 3. We make the following observations: First, the R^2
430 of different strategies positively correlate with their performance in general, indicating R^2 can serve
431

Policy filtering strategies	pass@1 on HumanEval	pass@1 on MBPP	R^2 based on SFT policy
PPO	78.1	73.8	0.782
PPO-M	80.8	75.0	0.886
PF-PPO (BoN)	75.8	71.7	0.454
PF-PPO (BR)	82.9	75.9	0.841
PF-PPO (BW)	82.4	76.2	0.952
PPO-top	80.5	71.2	0.621
PPO-top-random	81.9	75.3	0.889
PPO-top-bottom	81.7	75.4	0.927
PPO-pow-1	81.0	74.2	0.926
PPO-pow-2	81.3	75.4	0.939
PPO-pow-3	81.9	76.5	0.946

Table 3: The comparison on the pass@1 results of different policy filtering strategies on HumanEval and their corresponding R^2 based on the SFT policy. The background are colored based on their values with blue and red indicating the minimum and the maximum respectively.

as a tool to predict the performance of different policy filtering strategies. Second, different policy filtering strategies (except for BoN versions) improve the performance of the base PPO algorithms. This indicates that filtering samples with unreliable rewards can increase the signal-to-noise ratio of the reward model feedback and thus improve the performance. Third, PF-PPO strategies (which are rank-based) outperforms other strategies (which are threshold-based or reweighting-based). This may due to the fact that rank-based strategies are more robust to the reward distribution of the given reward model.

Discussion. The performance of different policy filtering strategies may vary across different tasks, different reward models, and different base models. Therefore, although we find that PF-PPO (BR) and PF-PPO (BW) are the best strategies in our setting, other policy filtering strategies may be a better choice in other settings. Therefore, a more practical procedure should be first calculate the R^2 using the given reward model and the corresponding SFT policy on the specific task and select candidate policy filtering strategies. Note that R^2 is not a perfect tool to select policy filtering strategies and we leave seeking for better predictive metrics as a future research direction.

5.7 FURTHER ANALYSIS

The training process of PPO-S, PPO-M, and PF-PPO. To provide a comprehensive view of the three algorithms, we show the training process.

We first present the training curves of PPO-S, PPO-M, and PF-PPO in Figure 2 (left). The training reward are evaluated on the samples collected by the filtered policy μ_θ and the evaluation rewards are calculated on the unfiltered policy π_θ . We observe that both the training reward and evaluation reward of PPO-M and PF-PPO surpass those of PPO-S. This indicates that sampling multiple responses from a context enhances the performance of the RLHF method, consistent with the findings in Shao et al. (2024). Moreover, in terms of optimizing reward for the given reward model, FP-PPO achieves a higher or equal reward compared with PPO-S and PPO-M, which indicates that the approximation made in the FP-PPO (i.e., optimizing π_θ as if it were μ_θ) does not induce negative effect on its capability to optimize the reward.

We also show the pass@1 results of different algorithms in Figure 2 (right). We observe that, while PF-PPO achieves a similar reward to that of PPO-M, the pass@1 result of PF-PPO exceeds that of PPO-M significantly. This results from the fact that PF-PPO optimizes on the reliable region of the reward model and thus alleviate the reward over-optimization issue.

Computational efficiency of PPO-S, PPO-M, and PF-PPO. PPO-S, PPO-M, and PF-PPO all collect different number of responses per query and train using different number of samples. For clarity, we list the computational complexity of these algorithms in Table 4. Note that, for all algorithms, we select the best checkpoint on the evaluation set and report the performance of this checkpoint. Combining the results in Table 4 and Figure 2, we can draw the following conclusions: First, the total computational complexity of PPO-S and PPO-M is almost the same, and the only difference is

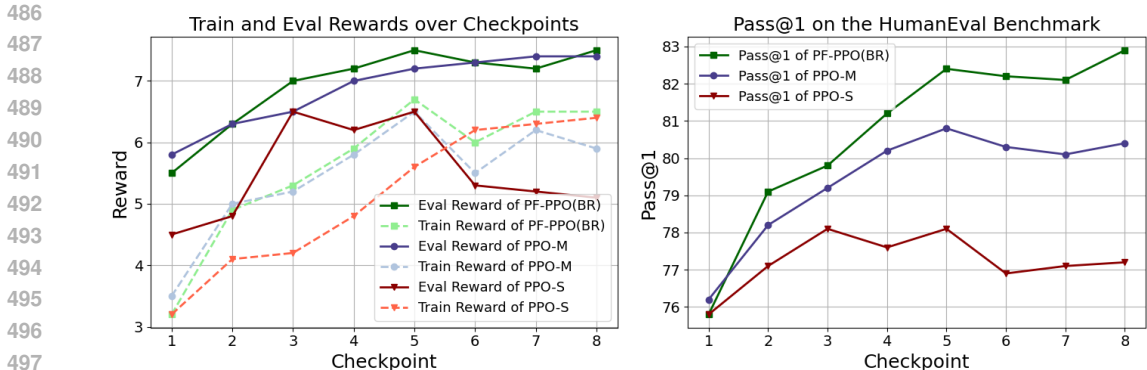


Figure 2: Left: The training and evaluation reward of PPO-S, PPO-M, and FP-PPO on HumanEval. The training reward and the evaluation reward are evaluated on the samples generated by the filtered policy μ_θ and the unfiltered policy π_θ respectively. Right: The pass@1 of PPO-S, PPO-M, and PF-PPO on the HumanEval benchmark.

	PPO-S	PPO-M	PF-PPO (BR / BW)
Queries sampled per iteration	$5n$	n	n
Responses sampled per query	1	5	5
#Query-response pairs per iteration	$5n$	$5n$	$5n$
Reward model forward pass per iteration	$5n$	$5n$	$5n$
Critic forward&backward pass per iteration	$5nm$	$5nm$	$2nm$
Policy forward&backward pass per iteration	$5nm$	$5nm$	$2nm$
HumanEval	100%	+2.69%	+6.15% / +5.51%
MBPP	100%	+1.63%	+2.85% / +3.25%
LeetCode	100%	+18.25%	+30.95% / +20.63%

Table 4: Comparison of computational complexity and the performance of PPO-S, PPO-M, and PF-PPO. We use n to denote the number of queries in the PPO query dataset, and use m to denote the number of PPO epochs (i.e., each query-response pair is used to accumulate loss and gradient for m times on average). PPO-M and PF-PPO collect $N = 5$ responses per query, and PF-PPO select 2 out of the $N = 5$ responses (on average) for network update. We also show the performance improvement of PPO-M and PF-PPO based on PPO-S.

that PPO-M is more likely to learn from different responses with the same query in the same batch or adjacent batches, which improves the performance. Second, the computational complexity of PF-PPO is less than that of PPO-S and PPO-M, while PF-PPO outperforms them. This indicates the effectiveness of our method.

6 CONCLUSION

In this paper, we propose a new reinforcement learning with human feedback (RLHF) method, **Policy Filtration for Proximal Policy Optimization (PF-PPO)**, aimed at mitigating the adverse effects of reward noise. When training the reward model using the Bradley-Terry approach, the reward signal is generally more reliable in the high or low reward regions but less reliable in the moderate reward regions. Motivated by this observation, we adopt a rank-based method to selectively use sample from these reliable regions more in PPO to improve the quality of the signal provided by the reward model. We conduct comprehensive experiments on code generation tasks, demonstrating that PF-PPO outperforms existing baselines. Additionally, we analyze PF-PPO, standard PPO, and PPO with multiple responses in details and show that filtering samples with unreliable rewards can improve the performance of the outcome policy.

REFERENCES

- 540
541
542 Afra Amini, Tim Vieira, and Ryan Cotterell. Variational best-of-n alignment. *arXiv preprint*
543 *arXiv:2407.06057*, 2024.
- 544
545 AI Anthropic. Introducing claude, 2023. URL [https://www.anthropic.com/news/](https://www.anthropic.com/news/introducing-claude)
546 [introducing-claude](https://www.anthropic.com/news/introducing-claude).
- 547
548 Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan,
549 Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language
550 models. *arXiv preprint arXiv:2108.07732*, 2021.
- 551
552 Mohammad Gheshlaghi Azar, Zhaohan Daniel Guo, Bilal Piot, Remi Munos, Mark Rowland, Michal
553 Valko, and Daniele Calandriello. A general theoretical paradigm to understand learning from
554 human preferences. In *International Conference on Artificial Intelligence and Statistics*, pp.
4447–4455. PMLR, 2024.
- 555
556 Yuntao Bai, Andy Jones, Kamal Ndousse, Amanda Askell, Anna Chen, Nova DasSarma, Dawn Drain,
557 Stanislav Fort, Deep Ganguli, Tom Henighan, et al. Training a helpful and harmless assistant with
558 reinforcement learning from human feedback. *arXiv preprint arXiv:2204.05862*, 2022.
- 559
560 Ralph Allan Bradley and Milton E Terry. Rank analysis of incomplete block designs: I. the method
561 of paired comparisons. *Biometrika*, 39(3/4):324–345, 1952.
- 562
563 Shreyas Chaudhari, Pranjal Aggarwal, Vishvak Murahari, Tanmay Rajpurohit, Ashwin Kalyan,
564 Karthik Narasimhan, Ameet Deshpande, and Bruno Castro da Silva. Rlhf deciphered: A critical
565 analysis of reinforcement learning from human feedback for llms. *arXiv preprint arXiv:2404.08555*,
566 2024.
- 567
568 Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared
569 Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large
570 language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.
- 571
572 Paul F Christiano, Jan Leike, Tom Brown, Miljan Martic, Shane Legg, and Dario Amodei. Deep
573 reinforcement learning from human preferences. *Advances in neural information processing*
574 *systems*, 30, 2017.
- 575
576 Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser,
577 Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. Training verifiers to solve
578 math word problems. *arXiv preprint arXiv:2110.14168*, 2021.
- 579
580 Thomas Coste, Usman Anwar, Robert Kirk, and David Krueger. Reward model ensembles help
581 mitigate overoptimization. *arXiv preprint arXiv:2310.02743*, 2023.
- 582
583 N Draper. *Applied regression analysis*. McGraw-Hill. Inc, 1998.
- 584
585 Kawin Ethayarajh, Winnie Xu, Niklas Muennighoff, Dan Jurafsky, and Douwe Kiela. Kto: Model
586 alignment as prospect theoretic optimization. *arXiv preprint arXiv:2402.01306*, 2024.
- 587
588 Leo Gao, John Schulman, and Jacob Hilton. Scaling laws for reward model overoptimization. In
589 *International Conference on Machine Learning*, pp. 10835–10866. PMLR, 2023.
- 590
591 Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi,
592 Yu Wu, YK Li, et al. Deepseek-coder: When the large language model meets programming—the
593 rise of code intelligence. *arXiv preprint arXiv:2401.14196*, 2024.
- Jian Hu, Xibin Wu, Weixun Wang, Dehao Zhang, Yu Cao, et al. Openrlhf: An easy-to-use, scalable
and high-performance rlhf framework. *arXiv preprint arXiv:2405.11143*, 2024.

- 594 Hamish Ivison, Yizhong Wang, Jiacheng Liu, Zeqiu Wu, Valentina Pyatkin, Nathan Lambert, Noah A
595 Smith, Yejin Choi, and Hannaneh Hajishirzi. Unpacking dpo and ppo: Disentangling best practices
596 for learning from preference feedback. *arXiv preprint arXiv:2406.09279*, 2024.
597
- 598 Natasha Jaques, Asma Ghandeharioun, Judy Hanwen Shen, Craig Ferguson, Agata Lapedriza, Noah
599 Jones, Shixiang Gu, and Rosalind Picard. Way off-policy batch deep reinforcement learning of
600 implicit human preferences in dialog. *arXiv preprint arXiv:1907.00456*, 2019.
- 601 Adam Tauman Kalai and Santosh S Vempala. Calibrated language models must hallucinate. In
602 *Proceedings of the 56th Annual ACM Symposium on Theory of Computing*, pp. 160–171, 2024.
603
- 604 Nathan Lambert, Thomas Krendl Gilbert, and Tom Zick. The history and risks of reinforcement
605 learning and human feedback. *arXiv e-prints*, pp. arXiv–2310, 2023.
- 606 Tianqi Liu, Yao Zhao, Rishabh Joshi, Misha Khalman, Mohammad Saleh, Peter J Liu, and Jialu Liu.
607 Statistical rejection sampling improves preference optimization. *arXiv preprint arXiv:2309.06657*,
608 2023.
- 609 Shayne Longpre, Le Hou, Tu Vu, Albert Webson, Hyung Won Chung, Yi Tay, Denny Zhou, Quoc V
610 Le, Barret Zoph, Jason Wei, et al. The flan collection: Designing data and methods for effective
611 instruction tuning. In *International Conference on Machine Learning*, pp. 22631–22648. PMLR,
612 2023.
- 613 Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing
614 Ma, Qingwei Lin, and Daxin Jiang. Wizardcoder: Empowering code large language models with
615 evol-instruct. *arXiv preprint arXiv:2306.08568*, 2023.
616
- 617 Lev McKinney, Yawen Duan, David Krueger, and Adam Gleave. On the fragility of learned reward
618 functions. *arXiv preprint arXiv:2301.03652*, 2023.
- 619 Ted Moskovitz, Aaditya K Singh, DJ Strouse, Tuomas Sandholm, Ruslan Salakhutdinov, Anca D
620 Dragan, and Stephen McAleer. Confronting reward model overoptimization with constrained rlhf.
621 *arXiv preprint arXiv:2310.04373*, 2023.
622
- 623 Reiichiro Nakano, Jacob Hilton, Suchir Balaji, Jeff Wu, Long Ouyang, Christina Kim, Christopher
624 Hesse, Shantanu Jain, Vineet Kosaraju, William Saunders, et al. Webgpt: Browser-assisted
625 question-answering with human feedback. *arXiv preprint arXiv:2112.09332*, 2021.
- 626 OpenAI. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.
627
- 628 Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong
629 Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow
630 instructions with human feedback. *Advances in Neural Information Processing Systems*, 35:
631 27730–27744, 2022.
- 632 Arka Pal, Deep Karkhanis, Samuel Dooley, Manley Roberts, Siddhartha Naidu, and Colin White.
633 Smaug: Fixing failure modes of preference optimisation with dpo-positive. *arXiv preprint*
634 *arXiv:2402.13228*, 2024.
- 635 Richard Yuanzhe Pang, Weizhe Yuan, Kyunghyun Cho, He He, Sainbayar Sukhbaatar, and Jason
636 Weston. Iterative reasoning preference optimization. *arXiv preprint arXiv:2404.19733*, 2024.
637
- 638 Silviu Pitis. Failure modes of learning reward models for llms and other sequence models. In *ICML*
639 *2023 Workshop The Many Facets of Preference-Based Learning*, 2023.
- 640 Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea
641 Finn. Direct preference optimization: Your language model is secretly a reward model. *Advances*
642 *in Neural Information Processing Systems*, 36, 2024.
- 643 John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy
644 optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.
645
- 646 Pier Giuseppe Sessa, Robert Dadashi, Léonard Hussenot, Johan Ferret, Nino Vieillard, Alexandre
647 Ramé, Bobak Shariari, Sarah Perrin, Abe Friesen, Geoffrey Cideron, et al. Bond: Aligning llms
with best-of-n distillation. *arXiv preprint arXiv:2407.14622*, 2024.

- 648 Amrith Setlur, Saurabh Garg, Xinyang Geng, Naman Garg, Virginia Smith, and Aviral Kumar. RL on
649 incorrect synthetic data scales the efficiency of llm math reasoning by eight-fold. *arXiv preprint*
650 *arXiv:2406.14532*, 2024.
- 651 Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Mingchuan Zhang, YK Li, Yu Wu,
652 and Daya Guo. Deepseekmath: Pushing the limits of mathematical reasoning in open language
653 models. *arXiv preprint arXiv:2402.03300*, 2024.
- 654 Wei Shen, Jian Hu, Pengyu Zhao, Xiaonan He, and Lichang Chen. Advanced tricks for training
655 large language models with proximal policy optimization. <https://difficult-link-dd7.notion.site/eb7b2d1891f44b3a84e7396d19d39e6f>, 2024. Notion Blog.
- 656 Joar Skalse, Nikolaus Howe, Dmitrii Krasheninnikov, and David Krueger. Defining and characterizing
657 reward gaming. *Advances in Neural Information Processing Systems*, 35:9460–9471, 2022.
- 658 Feifan Song, Bowen Yu, Minghao Li, Haiyang Yu, Fei Huang, Yongbin Li, and Houfeng Wang.
659 Preference ranking optimization for human alignment. In *Proceedings of the AAAI Conference on*
660 *Artificial Intelligence*, volume 38, pp. 18990–18998, 2024.
- 661 Nisan Stiennon, Long Ouyang, Jeffrey Wu, Daniel Ziegler, Ryan Lowe, Chelsea Voss, Alec Radford,
662 Dario Amodei, and Paul F Christiano. Learning to summarize with human feedback. *Advances in*
663 *Neural Information Processing Systems*, 33:3008–3021, 2020.
- 664 Gemini Team, Rohan Anil, Sebastian Borgeaud, Yonghui Wu, Jean-Baptiste Alayrac, Jiahui Yu, Radu
665 Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, et al. Gemini: a family of highly capable
666 multimodal models. *arXiv preprint arXiv:2312.11805*, 2023.
- 667 Qwen Team. Introducing qwen1.5, February 2024. URL <https://qwenlm.github.io/blog/qwen1.5/>.
- 668 Binghai Wang, Rui Zheng, Lu Chen, Yan Liu, Shihan Dou, Caishuang Huang, Wei Shen, Senjie Jin,
669 Enyu Zhou, Chenyu Shi, et al. Secrets of rlhf in large language models part ii: Reward modeling.
670 *arXiv preprint arXiv:2401.06080*, 2024.
- 671 Tianwen Wei, Jian Luan, Wei Liu, Shuang Dong, and Bin Wang. Cmath: Can your language model
672 pass chinese elementary school math test? *arXiv preprint arXiv:2306.16636*, 2023a.
- 673 Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. Magicoder: Source code is
674 all you need. *arXiv preprint arXiv:2312.02120*, 2023b.
- 675 Tianhao Wu, Banghua Zhu, Ruoyu Zhang, Zhaojin Wen, Kannan Ramchandran, and Jiantao Jiao.
676 Pairwise proximal policy optimization: Harnessing relative feedback for llm alignment. *arXiv*
677 *preprint arXiv:2310.00212*, 2023.
- 678 An Yang, Baosong Yang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Zhou, Chengpeng Li,
679 Chengyuan Li, Dayiheng Liu, Fei Huang, et al. Qwen2 technical report. *arXiv preprint*
680 *arXiv:2407.10671*, 2024.
- 681 Zheng Yuan, Hongyi Yuan, Chuanqi Tan, Wei Wang, Songfang Huang, and Fei Huang. Rrhf:
682 Rank responses to align language models with human feedback without tears. *arXiv preprint*
683 *arXiv:2304.05302*, 2023.
- 684 Shun Zhang, Zhenfang Chen, Sunli Chen, Yikang Shen, Zhiqing Sun, and Chuang Gan. Improving
685 reinforcement learning from human feedback with efficient reward model ensemble. *arXiv preprint*
686 *arXiv:2401.16635*, 2024.
- 687 Wei Zhao, Mingyue Shang, Yang Liu, Liang Wang, and Jingming Liu. Ape210k: A large-scale and
688 template-rich dataset of math word problems. *arXiv preprint arXiv:2009.11506*, 2020.
- 689 Yao Zhao, Rishabh Joshi, Tianqi Liu, Misha Khalman, Mohammad Saleh, and Peter J Liu. Slic-hf:
690 Sequence likelihood calibration with human feedback. *arXiv preprint arXiv:2305.10425*, 2023.
- 691 Jing Zhou, Chenglin Jiang, Wei Shen, Xiao Zhou, and Xiaonan He. Leveraging web-crawled data for
692 high-quality fine-tuning. *arXiv preprint arXiv:2408.08003*, 2024.

702 Banghua Zhu, Hiteshi Sharma, Felipe Vieira Frujeri, Shi Dong, Chenguang Zhu, Michael I Jordan,
703 and Jiantao Jiao. Fine-tuning language models with advantage-induced policy alignment. *arXiv*
704 *preprint arXiv:2306.02231*, 2023.
705
706 Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y Wu, Yukun Li,
707 Huazuo Gao, Shirong Ma, et al. Deepseek-coder-v2: Breaking the barrier of closed-source models
708 in code intelligence. *arXiv preprint arXiv:2406.11931*, 2024.
709
710 Daniel M Ziegler, Nisan Stiennon, Jeffrey Wu, Tom B Brown, Alec Radford, Dario Amodei, Paul
711 Christiano, and Geoffrey Irving. Fine-tuning language models from human preferences. *arXiv*
712 *preprint arXiv:1909.08593*, 2019.
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755

A REWARD MODEL

The design of our algorithm is motivated by the observation that the reward model is less reliable when it yields moderate rewards. To provide more evidence that this property is universal across a broader range of benchmarks, we provide analyze the reward function on the MBPP and LeetCode benchmarks. We repeat the process in Figure 1 on MBPP and LeetCode and plot the figures in Figure 3. Note that we train different reward functions based on the datasets from these two benchmarks. We observe that the property holds on these two additional benchmarks, indicating this property may extend to broader fields.

Intuitively, this property should be universal to a broader range of tasks. For code generation tasks, it is quite common that some samples (e.g., the response matches the known correct answer or the response contains an obvious error) are easier to evaluate than others (e.g., the response tries to solve the problem by a novel approach). Therefore, those samples that are hard to evaluate by human should also be hard instances for the reward model.

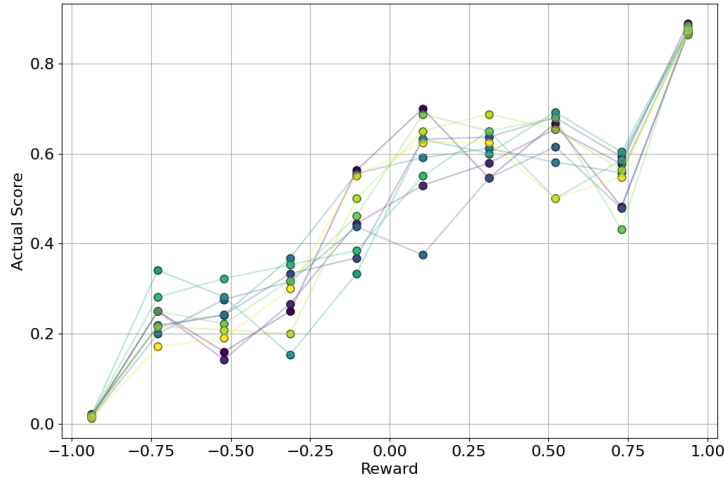
B EXPERIMENT RESULTS ON MATH REASONING TASKS

To evaluate the effectiveness of PF-PPO in other domains, we applied PF-PPO to solve math problems. We use Qwen1.5-7B (Team, 2024) as the SFT model and Ape210K (Zhao et al., 2020) and CMATH (Wei et al., 2023a) as the evaluation benchmarks. Other experimental settings are the same as Zhou et al. (2024). We use three types of reward models: the original reward model (ORM) that is trained on preference datasets using a Bradley–Terry model (Bradley & Terry, 1952), an oracle model (Oracle) that extracts the final answer from the response and compares it with the ground truth, and a combined reward model (CRM) that integrates the above two models, similar to the approach used in Qwen-Math (Yang et al., 2024). We compare PF-PPO to the standard PPO (PPO-S) using these reward models. We select the policy filtration strategy according to the procedure described in our main text, and choose the BR variant of PF-PPO.

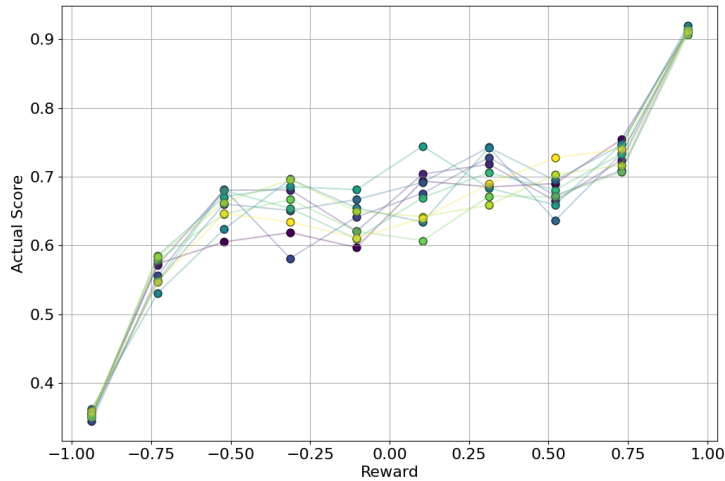
	Ape210K	CMATH
PPO-S + ORM	84.1	92.3
PF-PPO + ORM	86.2	95.1
PPO-S + Oracle	82.1	90.8
PF-PPO + Oracle	83.8	91.2
PPO-S + CRM	83.9	93.1
PF-PPO + CRM	84.3	94.2

Table 5: Comparison between PF-PPO and PPO-S on two math benchmarks (Ape210K and CMATH) using three different reward functions (the original reward model, the oracle model, and the combined reward model). Better results for each reward model is highlighted in **bold**.

We can observe that PF-PPO consistently outperforms the PPO algorithm on these two benchmarks across different reward models. In addition, the experiment results indicate that even if we can have access to the ground truth, using the oracle as the reward function does not perform as well as using a reward model (either the original reward model or the combined model). This finding is consistent with experiment results in Qwen-Math (Yang et al., 2024) and Deepseek-Math (Shao et al., 2024).



(a) The actual scores vs. the reward values for the reward model evaluated on MBPP



(b) The actual scores vs. the reward values for the reward model evaluated on LeetCode

Figure 3: We provide additional evidence that the reward model is less reliable when it yields moderate rewards than when it yields high or low rewards. We conduct the same statistics as in Figure 1 but on different benchmarks. Specifically, the reward models for the MBPP and LeetCode benchmarks are trained separately using the corresponding datasets for these two benchmarks. The MBPP and LeetCode benchmarks contains 378 and 1570 prompts respectively and we collect 10 responses for each prompt using a fine-tuned policy. We group the responses with similar rewards and calculate the average of their actual scores (i.e., the average correctness), indicating each group by one point. To evaluate the reliability of the reward model, we repeat the process ten times resulting in ten lines.

864 C QUALITATIVE RESULTS

865
866 In this section, we provide qualitative results on 1) how responses with high/middle/low rewards look
867 like and why responses with middle rewards are unreliable; and 2) the qualitative difference between
868 the code generated by the PF-PPO policy and the standard PPO (PPO-S) policy.
869

870 C.1 ANALYSIS ON THE THE RESPONSES ASSOCIATED WITH DIFFERENT REWARDS

871
872 We present a prompt along with several responses, including a correct response but assigned with
873 a low reward, an incorrect response but assigned with a high reward, an incorrect response with a
874 low reward, and a correct response with a high reward. The prompt describes a coding problem that
875 requires to convert fractions to decimals.

876 We have the following findings:

- 877
- 878 • For the correct response but assigned with a low reward, the generated code is less clear and
879 harder to read. For example, the code may mix several steps into one line.
- 880 • For the incorrect response but assigned with a high reward, the response incorrectly mixes
881 two correct approaches. This mistake can hardly be identified by the reward model (and
882 even GPT-4).
- 883 • For the incorrect response assigned with a low reward, the response contains an obvious
884 mistake which is easily detected by the reward model.

885
886 We also provide detailed analysis into the solutions to this problem. The given prompt is a coding
887 task to convert fraction to decimal.

```
888 1 Given two integers representing the numerator and denominator of a
889   fraction, return the fraction in string format.
890 2 If the fractional part is repeating, enclose the repeating part in
891   parentheses.
892 3 If multiple answers are possible, return any of them.
893 4 It is guaranteed that the length of the answer string is less than 104
894   for all the given inputs.
895 5
896 6 Please complete the following python code precisely:
897 7 ```python
898 8 class Solution:
899 9     def fractionToDecimal(self, numerator: int, denominator: int) -> str:
900 10 ```
```

901
902 This is a correct response with a high reward. This solution is thorough with clear variable names and
903 properly structured steps. It is easier to read due to breaking down steps explicitly such as calculating
904 the integer part and handling the remainder.

```
904 1
905 2
906 3 To solve the problem of converting a fraction to its decimal
907   representation, we can follow these steps:
908 4
909 5 1. **Determine the sign of the resultant decimal**: If both the numerator
910   and the denominator are negative, the result should be positive.
911   Otherwise, the result will have a leading minus sign if and only if
912   either the numerator or the denominator is negative.
913 6
914 7 2. **Convert the numerator and denominator to positive values**: Since
915   the division process involves integer division and the decimal part
916   requires modulus operation, both numerator and denominator should be
917   made positive for easier computation.
918 8
919 9 3. **Calculate the integer part**: The integer part is the result of
920   integer division of the numerator by the denominator.
921 10
```

```

918 11 4. **Manage the remainder**: As we perform integer division, the
919 12 remainder will be used to form the fractional part. This remainder
920 13 will be treated as the new numerator for the next step.
921 14
922 15 5. **Use a dictionary to manage repetition**: To detect recurring
923 16 decimals, we need to keep track of all remainders we have seen and
924 17 their respective positions in the decimal expansion. If a remainder
925 18 repeats, it indicates a recurring decimal.
926 19
927 20 6. **Build the result string**: Start with the integer part, then append
928 21 a decimal point. Then, use a loop to compute each digit of the
929 22 fractional part. Use the dictionary to detect and mark any recurring
930 23 decimals.
931 24
932 25 7. **Return the result**: Finally, return the constructed string for the
933 26 result.
934 27
935 28 Below is the complete Python code for the `fractionToDecimal` method:
936 29
937 30 ```python
938 31 class Solution:
939 32     def fractionToDecimal(self, numerator: int, denominator: int) -> str:
940 33         if numerator == 0:
941 34             return "0"
942 35
943 36         # Determine the sign of the result
944 37         result_sign = "-" if (numerator < 0) ^ (denominator < 0) else ""
945 38
946 39         # Make both numerator and denominator positive
947 40         numerator = abs(numerator)
948 41         denominator = abs(denominator)
949 42
950 43         # Calculate the integer part of the result
951 44         integer_part = numerator // denominator
952 45         remainder = numerator % denominator
953 46
954 47         # If there's no remainder, return the integer part
955 48         if remainder == 0:
956 49             return result_sign + str(integer_part)
957 50
958 51         # Prepare to build the fractional part
959 52         result = []
960 53         result.append(".") # Start with the decimal point
961 54         remainder_map = {} # To keep track of remainders and their
962 55         positions
963 56
964 57         # Process each place value in the fractional part
965 58         position = 0
966 59         while remainder not in remainder_map:
967 60             remainder_map[remainder] = position
968 61             remainder *= 10
969 62             result_digit = remainder // denominator
970 63             result.append(str(result_digit))
971 64             remainder %= denominator
972 65             if remainder == 0:
973 66                 break
974 67             position += 1
975 68
976 69         # If there's a repeat, insert parentheses
977 70         if remainder in remainder_map:
978 71             repeat_start = remainder_map[remainder]
979 72             result.insert(repeat_start + 1, "(")
980 73             result.append(")")
981 74
982 75         return result_sign + str(integer_part) + ".".join(result)

```

```

972 66 '''
973 67
974 68 This code correctly handles edge cases like zero denominator, negative
975 numbers, repeating decimals, and ensures that the result is less than
976 10^4 characters long.

```

This is a correct response but assigned with a low reward. Compared with the previous response with high reward, this response mixes multiple operations in one line, making it harder to understand (e.g., Line 34).

```

982 1 To solve the problem of converting a fraction to its decimal
983 representation, we can use the following approach:
984 2
985 3 1. Handle the sign of the result.
986 4 2. Calculate the integral part of the result, which is the part before
987 the decimal point.
988 5 3. Handle the case where the fractional part is repeating.
989 6 4. Calculate the fractional part and handle cycles.
990 7
991 8 Here's the completed Python code:
992 9
993 10 ```python
994 11 class Solution:
995 12     def fractionToDecimal(self, numerator: int, denominator: int) -> str:
996 13         # Handle sign of result
997 14         sign = '-' if numerator * denominator < 0 else ''
998 15         numerator, denominator = abs(numerator), abs(denominator)
999 16
1000 17         # Calculate integral part
1001 18         integral_part = numerator // denominator
1002 19
1003 20         # Calculate fractional part
1004 21         remainder = numerator % denominator
1005 22         if remainder == 0:
1006 23             return sign + str(integral_part)
1007 24
1008 25         # Use a dictionary to track remainders and their positions
1009 26         # in the fractional part to detect cycles
1010 27         seen_remainders = {}
1011 28         result = []
1012 29         index = 0
1013 30
1014 31         while remainder not in seen_remainders:
1015 32             seen_remainders[remainder] = index
1016 33             result.append(str(remainder * 10 // denominator))
1017 34             remainder = (remainder * 10) % denominator
1018 35             if remainder == 0:
1019 36                 break
1020 37             index += 1
1021 38
1022 39         # If there's a cycle, insert the parentheses
1023 40         if remainder in seen_remainders:
1024 41             index = seen_remainders[remainder]
1025 42             result.insert(index, '(')
1026 43             result.append(')')
1027 44
1028 45         return sign + str(integral_part) + '.' + ''.join(result)
1029 46
1030 47 '''
1031 48 ### Explanation:
1032 49
1033 50 - We first handle the signs of the numerator and denominator.
1034 51 - We then calculate the integral part of the result by integer division.

```

```

1026 52 - If the remainder is zero, the fraction is an integer and we return it
1027 as a string.
1028 53 - We use a dictionary 'seen_remainders' to keep track of the positions
1029 where each remainder appears, allowing us to detect cycles.
1030 54 - If a cycle is detected, we insert parentheses at the start of the
1031 repeating part.
1032 55 - Finally, we assemble the result as a string and return it.

```

This is an incorrect response but assigned with a high reward. In Line 32, the decimal point is added to the result list but is not later counted when getting wrapped by the parentheses, leading to the wrong format. This is a mixture of two correct approaches, one that adds the decimal points to result but sets an offset for this (cf. Line 44 and Line 62 in the first response) and one that outputs the decimal point separately (cf. Line 45 in the second response).

```

1039 1 To solve the problem of converting a fraction to a decimal string, we
1040 need to handle the following cases:
1041 2
1042 3 1. Integers: If the numerator and denominator are both integers, the
1043 decimal representation is straightforward: divide the numerator by
1044 the denominator and convert the result to a string.
1045 4
1046 5 2. Decimal Representation: If the division produces a remainder, it
1047 indicates a repeating decimal pattern. Use a dictionary to keep track
1048 of the remainders and their corresponding positions in the resultant
1049 decimal string. Whenever a remainder repeats, enclose the repeating
1050 part in parentheses.
1051 6
1052 7 3. Handling Negative Numbers: Handle both positive and negative
1053 numbers appropriately.
1054 8
1055 9 Here is the complete code for the solution:
1056 10
1057 11 ```python
1058 12 class Solution:
1059 13     def fractionToDecimal(self, numerator: int, denominator: int) -> str:
1060 14         # Handle sign of the result
1061 15         sign = '-' if numerator * denominator < 0 else ''
1062 16         numerator, denominator = abs(numerator), abs(denominator)
1063 17
1064 18         # Calculate integral part
1065 19         integral_part = numerator // denominator
1066 20         numerator %= denominator
1067 21
1068 22         # Handle zero remainder
1069 23         if numerator == 0:
1070 24             return sign + str(integral_part)
1071 25
1072 26         # Prepare list to store the decimal digits
1073 27         result = []
1074 28         # Dictionary to keep track of remainders and their positions
1075 29         remainder_map = {}
1076 30
1077 31         # Initialize the decimal part
1078 32         result.append('.')
1079 33         position = 0
1080 34
1081 35         # Loop until the remainder repeats or becomes zero
1082 36         while numerator > 0:
1083 37             # If the remainder has already appeared before, it's a
1084 38             repeating part
1085 39             if numerator in remainder_map:
1086 40                 result.insert(remainder_map[numerator], '(')
1087 41                 result.append(')')
1088 41                 break

```

```

1080 42
1081 43         # Record the position of the current remainder
1082 44         remainder_map[numerator] = position
1083 45         numerator *= 10
1084 46         result.append(str(numerator // denominator))
1085 47         numerator %= denominator
1086 48         position += 1
1087 49
1088 50     return sign + str(integral_part) + ''.join(result)
1089 51 '''
1090 52
1091 53 ### Explanation:
1092 54 - We use a dictionary 'remainder_map' to track the position of each
1093 55 remainder seen. This helps us identify the repeating part of the
1094 56 decimal if it exists.
1095 57 - The loop continues until the remainder becomes zero or repeats itself.
1096 58 - If the remainder repeats, we insert the opening '(' immediately before
1097 59 the repeated part, and close it with ')'.
1098 60 - The result is assembled from the integral part, the decimal point, and
1099 61 the decimal digits or parenthesized pattern.

```

This is an incorrect response with a low reward. In Line 59-61, the program contains an obvious error that it cannot handle the scenario where there exists a decimal part but does not contain any repeating part.

```

1102 1
1103 2
1104 3 To solve the problem of converting a fraction to its decimal
1105 4 representation, we need to consider the following steps:
1106 5
1107 6 1. Handling Zero Denominator: If the denominator is zero, it's
1108 7 impossible to divide, so we should return "NaN" (Not a Number) or
1109 8 handle it according to the problem constraints.
1110 9
1111 10 2. Handling Negative Numbers: If either the numerator or denominator
1112 11 is negative, the sign of the result should be negative.
1113 12
1114 13 3. Calculate the Integral Part: The integral part is the part before
1115 14 the decimal point, which can be calculated by integer division.
1116 15
1117 16 4. Calculate the Fractional Part:
1118 17 - Use a dictionary to keep track of the positions where the same
1119 18 remainder started repeating.
1120 19 - Divide the remainder by the denominator repeatedly, building the
1121 20 fractional part digit by digit.
1122 21 - If the remainder repeats, it indicates a recurring decimal, and we
1123 22 should insert parentheses to mark it.
1124 23
1125 24 Here's the complete code implementation:
1126 25
1127 26 ```python
1128 27 class Solution:
1129 28     def fractionToDecimal(self, numerator: int, denominator: int) -> str:
1130 29         # Handle zero denominator
1131 30         if denominator == 0:
1132 31             return "NaN"
1133 32
1134 33         # Handle signs
1135 34         result_sign = '-' if (numerator < 0) ^ (denominator < 0) else ''
1136 35         numerator, denominator = abs(numerator), abs(denominator)
1137 36
1138 37         # Calculate the integral part
1139 38         integral_part = numerator // denominator
1140 39         numerator %= denominator
1141 40

```

```

1134 32
1135 33     # Initialize the fractional part list and a dictionary to track
1136     remainders
1137 34     fractional_part = []
1138 35     remainder_index = {}
1139 36
1140 37     # Continue while there is a remainder
1141 38     position = 0
1142 39     while numerator > 0:
1143 40         # If the remainder has already occurred before, it's a
1144 41         repeating decimal
1145 42         if numerator in remainder_index:
1146 43             break
1147 44
1148 45         # Record this remainder's position
1149 46         remainder_index[numerator] = position
1150 47
1151 48         # Calculate the next digit
1152 49         numerator *= 10
1153 50         fractional_part.append(str(numerator // denominator))
1154 51         numerator %= denominator
1155 52         position += 1
1156 53
1157 54     # Build the result
1158 55     if not fractional_part:
1159 56         # No repeating part, return integral part
1160 57         return result_sign + str(integral_part)
1161 58     else:
1162 59         # Repeating part detected
1163 60         repeating_start = remainder_index[numerator]
1164 61         return result_sign + str(integral_part) + '.' + ''.join(
1165 62             fractional_part[:repeating_start]) + \
1166 63             '(' + ''.join(fractional_part[repeating_start:]) + ')'
1167 64
1168 65     """
1169 66     """
1170 67
1171 68     """
1172 69     """
1173 70
1174 71     """
1175 72     """
1176 73
1177 74
1178 75
1179 76
1180 77
1181 78
1182 79
1183 80
1184 81
1185 82
1186 83
1187 84
1188 85
1189 86
1190 87
1191 88
1192 89
1193 90
1194 91
1195 92
1196 93
1197 94
1198 95
1199 96
1200 97
1201 98
1202 99
1203 100
1204 101
1205 102
1206 103
1207 104
1208 105
1209 106
1210 107
1211 108
1212 109
1213 110
1214 111
1215 112
1216 113
1217 114
1218 115
1219 116
1220 117
1221 118
1222 119
1223 120
1224 121
1225 122
1226 123
1227 124
1228 125
1229 126
1230 127
1231 128
1232 129
1233 130
1234 131
1235 132
1236 133
1237 134
1238 135
1239 136
1240 137
1241 138
1242 139
1243 140
1244 141
1245 142
1246 143
1247 144
1248 145
1249 146
1250 147
1251 148
1252 149
1253 150
1254 151
1255 152
1256 153
1257 154
1258 155
1259 156
1260 157
1261 158
1262 159
1263 160
1264 161
1265 162
1266 163
1267 164
1268 165
1269 166
1270 167
1271 168
1272 169
1273 170
1274 171
1275 172
1276 173
1277 174
1278 175
1279 176
1280 177
1281 178
1282 179
1283 180
1284 181
1285 182
1286 183
1287 184
1288 185
1289 186
1290 187
1291 188
1292 189
1293 190
1294 191
1295 192
1296 193
1297 194
1298 195
1299 196
1300 197
1301 198
1302 199
1303 200
1304 201
1305 202
1306 203
1307 204
1308 205
1309 206
1310 207
1311 208
1312 209
1313 210
1314 211
1315 212
1316 213
1317 214
1318 215
1319 216
1320 217
1321 218
1322 219
1323 220
1324 221
1325 222
1326 223
1327 224
1328 225
1329 226
1330 227
1331 228
1332 229
1333 230
1334 231
1335 232
1336 233
1337 234
1338 235
1339 236
1340 237
1341 238
1342 239
1343 240
1344 241
1345 242
1346 243
1347 244
1348 245
1349 246
1350 247
1351 248
1352 249
1353 250
1354 251
1355 252
1356 253
1357 254
1358 255
1359 256
1360 257
1361 258
1362 259
1363 260
1364 261
1365 262
1366 263
1367 264
1368 265
1369 266
1370 267
1371 268
1372 269
1373 270
1374 271
1375 272
1376 273
1377 274
1378 275
1379 276
1380 277
1381 278
1382 279
1383 280
1384 281
1385 282
1386 283
1387 284
1388 285
1389 286
1390 287
1391 288
1392 289
1393 290
1394 291
1395 292
1396 293
1397 294
1398 295
1399 296
1400 297
1401 298
1402 299
1403 300
1404 301
1405 302
1406 303
1407 304
1408 305
1409 306
1410 307
1411 308
1412 309
1413 310
1414 311
1415 312
1416 313
1417 314
1418 315
1419 316
1420 317
1421 318
1422 319
1423 320
1424 321
1425 322
1426 323
1427 324
1428 325
1429 326
1430 327
1431 328
1432 329
1433 330
1434 331
1435 332
1436 333
1437 334
1438 335
1439 336
1440 337
1441 338
1442 339
1443 340
1444 341
1445 342
1446 343
1447 344
1448 345
1449 346
1450 347
1451 348
1452 349
1453 350
1454 351
1455 352
1456 353
1457 354
1458 355
1459 356
1460 357
1461 358
1462 359
1463 360
1464 361
1465 362
1466 363
1467 364
1468 365
1469 366
1470 367
1471 368
1472 369
1473 370
1474 371
1475 372
1476 373
1477 374
1478 375
1479 376
1480 377
1481 378
1482 379
1483 380
1484 381
1485 382
1486 383
1487 384
1488 385
1489 386
1490 387
1491 388
1492 389
1493 390
1494 391
1495 392
1496 393
1497 394
1498 395
1499 396
1500 397
1501 398
1502 399
1503 400
1504 401
1505 402
1506 403
1507 404
1508 405
1509 406
1510 407
1511 408
1512 409
1513 410
1514 411
1515 412
1516 413
1517 414
1518 415
1519 416
1520 417
1521 418
1522 419
1523 420
1524 421
1525 422
1526 423
1527 424
1528 425
1529 426
1530 427
1531 428
1532 429
1533 430
1534 431
1535 432
1536 433
1537 434
1538 435
1539 436
1540 437
1541 438
1542 439
1543 440
1544 441
1545 442
1546 443
1547 444
1548 445
1549 446
1550 447
1551 448
1552 449
1553 450
1554 451
1555 452
1556 453
1557 454
1558 455
1559 456
1560 457
1561 458
1562 459
1563 460
1564 461
1565 462
1566 463
1567 464
1568 465
1569 466
1570 467
1571 468
1572 469
1573 470
1574 471
1575 472
1576 473
1577 474
1578 475
1579 476
1580 477
1581 478
1582 479
1583 480
1584 481
1585 482
1586 483
1587 484
1588 485
1589 486
1590 487
1591 488
1592 489
1593 490
1594 491
1595 492
1596 493
1597 494
1598 495
1599 496
1600 497
1601 498
1602 499
1603 500
1604 501
1605 502
1606 503
1607 504
1608 505
1609 506
1610 507
1611 508
1612 509
1613 510
1614 511
1615 512
1616 513
1617 514
1618 515
1619 516
1620 517
1621 518
1622 519
1623 520
1624 521
1625 522
1626 523
1627 524
1628 525
1629 526
1630 527
1631 528
1632 529
1633 530
1634 531
1635 532
1636 533
1637 534
1638 535
1639 536
1640 537
1641 538
1642 539
1643 540
1644 541
1645 542
1646 543
1647 544
1648 545
1649 546
1650 547
1651 548
1652 549
1653 550
1654 551
1655 552
1656 553
1657 554
1658 555
1659 556
1660 557
1661 558
1662 559
1663 560
1664 561
1665 562
1666 563
1667 564
1668 565
1669 566
1670 567
1671 568
1672 569
1673 570
1674 571
1675 572
1676 573
1677 574
1678 575
1679 576
1680 577
1681 578
1682 579
1683 580
1684 581
1685 582
1686 583
1687 584
1688 585
1689 586
1690 587
1691 588
1692 589
1693 590
1694 591
1695 592
1696 593
1697 594
1698 595
1699 596
1700 597
1701 598
1702 599
1703 600
1704 601
1705 602
1706 603
1707 604
1708 605
1709 606
1710 607
1711 608
1712 609
1713 610
1714 611
1715 612
1716 613
1717 614
1718 615
1719 616
1720 617
1721 618
1722 619
1723 620
1724 621
1725 622
1726 623
1727 624
1728 625
1729 626
1730 627
1731 628
1732 629
1733 630
1734 631
1735 632
1736 633
1737 634
1738 635
1739 636
1740 637
1741 638
1742 639
1743 640
1744 641
1745 642
1746 643
1747 644
1748 645
1749 646
1750 647
1751 648
1752 649
1753 650
1754 651
1755 652
1756 653
1757 654
1758 655
1759 656
1760 657
1761 658
1762 659
1763 660
1764 661
1765 662
1766 663
1767 664
1768 665
1769 666
1770 667
1771 668
1772 669
1773 670
1774 671
1775 672
1776 673
1777 674
1778 675
1779 676
1780 677
1781 678
1782 679
1783 680
1784 681
1785 682
1786 683
1787 684
1788 685
1789 686
1790 687
1791 688
1792 689
1793 690
1794 691
1795 692
1796 693
1797 694
1798 695
1799 696
1800 697
1801 698
1802 699
1803 700
1804 701
1805 702
1806 703
1807 704
1808 705
1809 706
1810 707
1811 708
1812 709
1813 710
1814 711
1815 712
1816 713
1817 714
1818 715
1819 716
1820 717
1821 718
1822 719
1823 720
1824 721
1825 722
1826 723
1827 724
1828 725
1829 726
1830 727
1831 728
1832 729
1833 730
1834 731
1835 732
1836 733
1837 734
1838 735
1839 736
1840 737
1841 738
1842 739
1843 740
1844 741
1845 742
1846 743
1847 744
1848 745
1849 746
1850 747
1851 748
1852 749
1853 750
1854 751
1855 752
1856 753
1857 754
1858 755
1859 756
1860 757
1861 758
1862 759
1863 760
1864 761
1865 762
1866 763
1867 764
1868 765
1869 766
1870 767
1871 768
1872 769
1873 770
1874 771
1875 772
1876 773
1877 774
1878 775
1879 776
1880 777
1881 778
1882 779
1883 780
1884 781
1885 782
1886 783
1887 784
1888 785
1889 786
1890 787
1891 788
1892 789
1893 790
1894 791
1895 792
1896 793
1897 794
1898 795
1899 796
1900 797
1901 798
1902 799
1903 800
1904 801
1905 802
1906 803
1907 804
1908 805
1909 806
1910 807
1911 808
1912 809
1913 810
1914 811
1915 812
1916 813
1917 814
1918 815
1919 816
1920 817
1921 818
1922 819
1923 820
1924 821
1925 822
1926 823
1927 824
1928 825
1929 826
1930 827
1931 828
1932 829
1933 830
1934 831
1935 832
1936 833
1937 834
1938 835
1939 836
1940 837
1941 838
1942 839
1943 840
1944 841
1945 842
1946 843
1947 844
1948 845
1949 846
1950 847
1951 848
1952 849
1953 850
1954 851
1955 852
1956 853
1957 854
1958 855
1959 856
1960 857
1961 858
1962 859
1963 860
1964 861
1965 862
1966 863
1967 864
1968 865
1969 866
1970 867
1971 868
1972 869
1973 870
1974 871
1975 872
1976 873
1977 874
1978 875
1979 876
1980 877
1981 878
1982 879
1983 880
1984 881
1985 882
1986 883
1987 884
1988 885
1989 886
1990 887
1991 888
1992 889
1993 890
1994 891
1995 892
1996 893
1997 894
1998 895
1999 896
2000 897
2001 898
2002 899
2003 900
2004 901
2005 902
2006 903
2007 904
2008 905
2009 906
2010 907
2011 908
2012 909
2013 910
2014 911
2015 912
2016 913
2017 914
2018 915
2019 916
2020 917
2021 918
2022 919
2023 920
2024 921
2025 922
2026 923
2027 924
2028 925
2029 926
2030 927
2031 928
2032 929
2033 930
2034 931
2035 932
2036 933
2037 934
2038 935
2039 936
2040 937
2041 938
2042 939
2043 940
2044 941
2045 942
2046 943
2047 944
2048 945
2049 946
2050 947
2051 948
2052 949
2053 950
2054 951
2055 952
2056 953
2057 954
2058 955
2059 956
2060 957
2061 958
2062 959
2063 960
2064 961
2065 962
2066 963
2067 964
2068 965
2069 966
2070 967
2071 968
2072 969
2073 970
2074 971
2075 972
2076 973
2077 974
2078 975
2079 976
2080 977
2081 978
2082 979
2083 980
2084 981
2085 982
2086 983
2087 984
2088 985
2089 986
2090 987
2091 988
2092 989
2093 990
2094 991
2095 992
2096 993
2097 994
2098 995
2099 996
2100 997
2101 998
2102 999
2103 1000

```

C.2 ANALYSIS ON THE RESPONSES GENERATED BY PF-PPO COMPARED WITH STANDARD PPO (PPO-S)

We compare the answers from PF-PPO and PPO-S respectively for the same prompts, and conclude that the answer from the PF-PPO policy is more inclined to follow a standard approach and the response is more concise (or shorter), making it easier to understand and implement.

Specifically, we present their responses for the prompt that request the agent to write a code to *find all safe nodes in a directed graph*. The two responses given by PF-PPO and PPO-S are both correct. However, the difference is that PF-PPO adopts the deep first search (DFS) method while PPO-S adopts the topological sorting approach. The logic is simpler for DFS and the implementation is

1188 easier, making it easier to understand. Both approaches have roughly equivalent time and space costs,
 1189 but the DFS method is slightly more space efficient.

1190 These findings suggest an advantage in terms of readability and implementation simplicity when
 1191 using the PF-PPO policy, particularly for those familiar with recursive techniques.

1193 Prompt (find all safe nodes in a directed graph):

1194 There is a directed graph of n nodes with each node labeled from 0 to $n -$
 1195 1. The graph is represented by a 0-indexed 2D integer array `graph`
 1196 where `graph[i]` is an integer array of nodes adjacent to node i ,
 1197 meaning there is an edge from node i to each node in `graph[i]`.
 1198 A node is a terminal node if there are no outgoing edges. A node is a
 1199 safe node if every possible path starting from that node leads to a
 1200 terminal node (or another safe node).
 1201 Return an array containing all the safe nodes of the graph. The answer
 1202 should be sorted in ascending order.

1203 Please complete the following python code precisely:

```
1204 ```python
1205 class Solution:
1206     def eventualSafeNodes(self, graph: List[List[int]]) -> List[int]:
1207     ...
```

1208 The response from PF-PPO:

1210 To solve the problem of finding all safe nodes in a directed graph, we
 1211 can use a topological sorting approach. The idea is to explore the
 1212 graph using depth-first search (DFS) and maintain a status for each
 1213 node indicating whether it is safe or not. Here's the step-by-step
 1214 breakdown of the solution:

- 1217 1. ****Initialize Status****: We will use a status array to track the safety
 1218 of each node. 0 means the node is not visited yet, 1 means it is a
 1219 safe node, and -1 means it is currently in the recursion stack (i.e.,
 1220 being processed in the current DFS path).
- 1221 2. ****DFS Function****:
 - 1222 - For each node, if it's safe or not is already determined (i.e.,
 - 1223 status is not 0), return the status.
 - 1224 - Set the status to -1 to indicate that we are currently processing it
 - 1225 .
 - 1226 - For each neighbor of the current node, call DFS. If any neighbor
 - 1227 returns false, it indicates that this path has a cycle and the
 - 1228 node is not safe.
 - 1229 - If no cycle is found for all neighbors, mark the node as safe and
 - 1230 return true.
- 1231 3. ****Iterate over all nodes****: Starting from each node that is not
 1232 visited, perform a DFS.
- 1233 4. ****Collect Results****: All nodes that return true from DFS are safe
 1234 nodes. Collect these node indices and return them in ascending order.

1235 Here's the complete Python code for the solution:

```
1236 ```python
1237 from typing import List
1238
1239 class Solution:
1240     def eventualSafeNodes(self, graph: List[List[int]]) -> List[int]:
1241         n = len(graph)
1242         # status: 0 for unvisited, 1 for safe, -1 for processing
```

```

1242     status = [0] * n
1243
1244     def dfs(node):
1245         # If the node is already processed (i.e., marked as safe or
1246         # unsafe), return its status
1247         if status[node] != 0:
1248             return status[node] == 1
1249
1250         # Mark the node as being processed
1251         status[node] = -1
1252
1253         # Visit all neighbors
1254         for neighbor in graph[node]:
1255             # If any neighbor is unsafe or in a cycle, the node is
1256             # unsafe
1257             if not dfs(neighbor):
1258                 return False
1259
1260         # All neighbors are safe, mark the node as safe and return
1261         # true
1262         status[node] = 1
1263         return True
1264
1265     # Try to DFS from each node
1266     safe_nodes = []
1267     for i in range(n):
1268         if dfs(i):
1269             safe_nodes.append(i)
1270
1271     return safe_nodes
1272
1273 """
1274
1275 ### Explanation:
1276 - **Topological Sorting**: We are using DFS to check if a path exists
1277   from a node to a terminal node. If a cycle is detected, the node is
1278   not safe.
1279 - **DFS Function**: The `dfs` function determines if a node is safe by
1280   checking all of its neighbors recursively. If any neighbor is unsafe
1281   or in a cycle, the node is unsafe.
1282 - **Status Tracking**: The status array is used to track if a node is
1283   safe or not, and helps avoid redundant computations.
1284 - **Result Collection**: All nodes that return `True` from the `dfs`
1285   function are safe nodes, and are added to the result list.
1286
1287 This approach efficiently identifies safe nodes by using a topological
1288 sorting approach and a depth-first search approach to detect cycles.
1289 """

```

The response from PPO-S:

```

1286 To solve this problem, we can use a \red{topological sort approach}
1287 combined with a depth-first search (DFS) approach to determine the
1288 terminal and safe nodes in the graph. The topological sort ensures
1289 that we process nodes in the order of their out-degree, which is
1290 suitable for directed acyclic graphs (DAGs), where no cycles can form
1291 .
1292 The idea is to traverse the graph using DFS, keeping track of the current
1293 path during traversal to detect cycles. Nodes that do not lead to a
1294 cycle are considered terminal, and nodes that are part of a cycle or
1295 are part of a path that leads to a cycle are considered unsafe.

```

Here's the Python implementation of the solution:

```

1296
1297 ```python
1298 from typing import List
1299 from collections import defaultdict, deque
1300
1301 class Solution:
1302     def eventualSafeNodes(self, graph: List[List[int]]) -> List[int]:
1303         # Initialize a dictionary to store the out-degree of each node
1304         out_degree = defaultdict(int)
1305         # Initialize a dictionary to store the dependencies (incoming
1306         # edges) of each node
1307         dependencies = defaultdict(list)
1308
1309         n = len(graph)
1310         # Build the dependencies and out-degree dictionaries
1311         for u in range(n):
1312             out_degree[u] = len(graph[u])
1313             for v in graph[u]:
1314                 dependencies[v].append(u)
1315
1316         # Initialize a queue for BFS and a set to keep track of safe
1317         # nodes
1318         queue = deque()
1319         safe_nodes = set()
1320
1321         # Add all nodes with no outgoing edges (terminals) to the queue
1322         for u in range(n):
1323             if out_degree[u] == 0:
1324                 queue.append(u)
1325
1326         # BFS to find safe nodes
1327         while queue:
1328             u = queue.popleft()
1329             safe_nodes.add(u)
1330             # For each node v that depends on u, decrement its out-degree
1331             # If v's out-degree becomes zero, it becomes a terminal
1332             for v in dependencies[u]:
1333                 out_degree[v] -= 1
1334                 if out_degree[v] == 0:
1335                     queue.append(v)
1336
1337         # Sort and return safe nodes
1338         return sorted(safe_nodes)
1339
1340 ```
1341
1342 ### Explanation:
1343 1. Dependencies: We build a 'dependencies' dictionary, where each key
1344 is a node and the values are a list of nodes that depend on the key.
1345 This is used to track back edges.
1346 2. Out-degree: We also keep track of the out-degree of each node.
1347 This is used to determine which nodes are terminal (i.e., have no
1348 outgoing edges).
1349 3. Topological Sort (BFS): We start by adding all nodes with no
1350 outgoing edges (terminals) to the queue. We then process each node
1351 from this queue. For each node, we mark it as safe, reduce the out-
1352 degree of its dependencies, and if any of those dependencies become
1353 terminals (i.e., have no outgoing edges), they are added to the queue
1354 . This ensures we process all nodes that are part of cycles and
1355 prevent future revisits to nodes involved in such cycles.
1356 4. Result: The safe nodes are collected and sorted and returned as
1357 the final result.
1358
1359 This approach ensures that we correctly identify terminal nodes and use
1360 them to establish safe nodes, as well as handle cycles in the graph.

```