
Thinking LLMs: General Instruction Following with Thought Generation

Tianhao Wu^{1,2} Janice Lan¹ Weizhe Yuan^{1,3} Jiantao Jiao² Jason Weston^{1,3} Sainbayar Sukhbaatar¹

Abstract

LLMs are typically trained to answer user questions or follow instructions similarly to how human experts respond. However, in the standard alignment framework they lack the basic ability of explicit thinking before answering. Thinking is important for complex questions that require reasoning and planning – but can be applied to *any* task. We propose a training method for equipping existing LLMs with such thinking abilities for general instruction following without use of additional human data. We achieve this by an iterative search and optimization procedure that explores the space of possible thought generations, allowing the model to learn how to think without direct supervision. For each instruction, the thought candidates are scored using a judge model to evaluate their responses only, and then optimized via preference optimization. We show that this procedure leads to superior performance on AlpacaEval and Arena-Hard, and shows gains from thinking on non-reasoning categories such as marketing, health and general knowledge, in addition to more traditional reasoning & problem-solving tasks.

1. Introduction

Large Language Models (LLMs) are based on the Transformer architecture (Vaswani et al., 2017) that predicts the next token at each step. Each token takes the same amount of compute, so when LLMs are prompted with a user instruction, they have a fixed compute budget to generate the first response token regardless of the instruction’s complexity. One way to increase the compute budget for harder instructions is to allow LLMs to *think* internally before outputting an response. This is similar to humans who will take more

time and think before answering complex questions.

One approach is to generate thoughts as text, which takes advantage of the natural language capabilities of LLMs. LLMs are pre-trained on text containing human-written thoughts, which are hence encoded into the model. Chain-of-Thought (CoT) (Wei et al., 2022) is a widely used prompting technique that elicits such behavior by asking the model to write down its reasoning steps. However, the usage of CoT has been mostly limited to math and reasoning tasks. Meta-analysis by Sprague et al. (2024) found CoT methods to be unhelpful on tasks that do not involve math and logic.

In this paper, we focus on general instruction following instead of focusing on math or logic tasks. We argue that “thinking” should have broad utility. For example, in a creative writing task, internal thoughts can be used to plan overall structure and characters. In other tasks, internal thoughts can be used for understanding the user instruction better. Of course, it is likely that less thinking is required for simpler tasks, and more thinking for more complex ones. In general, we hypothesize that such *Thinking LLMs* will have an advantage on all sufficiently complex tasks. The emergence of models like OpenAI-O1 (OpenAI) and DeepSeek-R1 (Guo et al., 2025) also support our motivation.

However, it is challenging to train a model to think due to the lack of supervised training data. Although pre-training data does contain valuable information, coverage can be limited in certain domains as internal thoughts are often omitted in human writing. Existing post-training datasets typically consist of human responses, or preferences over responses, with no information on thought processes. The same is true for existing reward models. Combined with the difficulty and cost considerations of collecting human thought data, these factors impose a barrier in training Thinking LLMs.

The goal of this paper is to investigate the possibility of converting existing LLMs into Thinking LLMs that work across a wide variety of tasks, without any additional data. To this end, we introduce Thought Preference Optimization (TPO) that further trains an instruction-tuned LLM to make it capable of having internal thoughts. Our method is simple and reuses many parts of existing training pipelines. The LLM is first instructed to produce an output sequence that can be divided into thought and response parts. The thought part is considered internal, and not part of the response shown

¹Meta FAIR ²University of California, Berkeley
³New York University. Correspondence to: Tianhao Wu <thw@berkeley.edu>, Sainbayar Sukhbaatar <sainbar@meta.com>.

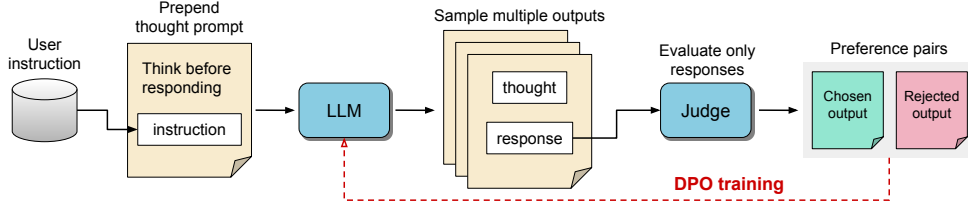


Figure 1. **Thought Preference Optimization:** We start by prompting the LLM to generate thoughts before its response. After sampling different outputs, we feed the response parts to the judge model which determines the best and worst ones. Then we use the corresponding full outputs as chosen and rejected pairs for DPO optimization. We perform multiple iterations of this training.

to the user. We optimize this thought and response output through iterative Reinforcement Learning from AI Feedback (RLAIF) training. We rely on a standard judge model that is trained to evaluate responses only, and implicitly judge the quality of the thoughts via the induced responses. This has the advantage of not requiring human curated thoughts or a special judge model capable of evaluating thoughts. Through preference optimization the thoughts are then optimized to improve the resulting responses. This contrasts with recent works such as Snell et al. (2024); Kumar et al. (2024b) which use additional supervision signals to guide models in self-refinement or self-correction. Instead of directly guiding the internal thought process, we allow the model to independently *learn to think*.

Given the multitude of evidence showing the effectiveness of CoT on logic-based tasks like math or coding, we focus our experiments on general instruction following instead. We train on diverse user instructions and evaluate our models on AlpacaEval and Arena-Hard, benchmarks that test general instruction following. We obtain a strong win rate of 52.5% and 37.3% respectively on them, outperforming the direct LLM counterpart without explicit thinking. We also conduct more fine-grained evaluations to determine which types of instructions benefit from thought. Surprisingly, we observe that thinking not only benefits topics like reasoning and problem solving, but also leads to better performance on categories that are not typically considered in the reasoning domain, such as general knowledge, marketing, and health. This opens up a new opportunity to develop Thinking LLMs aimed at general instruction following rather than specializing in more narrow technical fields.

2. Thought Preference Optimization

We now describe our Thought Preference Optimization (TPO) method for teaching LLMs to think before responding, as depicted in Figure 1. We start with a typical instruction-tuned LLM that outputs a response directly after the user instruction. We assume that there is no provided labeled thought data that we can finetune on, which makes training much more challenging. Instead, as a starting point to bootstrap our training process, for a given training user

Generic Thought Prompt

Respond to the following user query in a comprehensive and detailed way. You can write down your thought process before responding. Write your thoughts after “Here is my thought process:” and write your response after “Here is my response:”.

User query: {user instruction}

Specific Thought Prompt

Respond to the following user query in a comprehensive and detailed way. But first write down your internal thoughts. This must include your draft response and its evaluation. After this, write your final response after “<R>”.

User query: {user instruction}

Figure 2. **Thought Prompting.** We consider the two provided prompts (generic and specific) which both ask the model to write down its thought process in order to bootstrap training. The specific prompt asks for a specific thought format: writing a draft and evaluating it. Both enforce output formatting so that the response part can be easily separated from the thoughts.

instruction, we prompt the model to generate its thought process followed by the response. Sampling multiple such outputs, we then use preference optimization to improve the quality of thoughts (and paired responses) based solely on the quality of the responses.

2.1. Generating Thoughts from Thinking LLMs

Ideally, thought generation should be simple and compatible with existing LLM infrastructures. Hence, we keep the model architecture the same, as an autoregressive Transformer, although our method is potentially compatible with any model that outputs a sequence of tokens. At inference time, the core process is that the output consists of two parts: a thought part followed by a response part, both of which are in natural language. After generation, instead of directly sending that entire token sequence to the user, we preprocess it by splitting it into the two parts, and only sending the response part.

At the beginning of our training, we achieve this by prompt-

ing the model to write its thought process. We consider two possible *thought prompts*, shown in Figure 2. In order to separate thought from answer, we need the model to follow a strict format. The thought prompts contain fixed keywords the model should use so that we can use simple string matching to locate where the response part begins.

Thought Prompt Types Although the training process will change and optimize the type of thoughts, the initial thoughts are still important as they act as a starting point. The first thought prompt given in Figure 2 (top) is more generic and leaves it up to the model what the thoughts will contain. We also experiment with a more *specific* thought prompt, given in Figure 2 (bottom), that specifies that the thought should contain a draft response and its evaluation. Such specific prompts give us more control over the content of the thoughts, but also requires expert knowledge about what type of thoughts are helpful in LLMs.

Making Thoughts Internal As we mentioned, the thought part will be hidden from the end user, and only the response part will be provided to them. This differentiates our outputs from CoT prompting, where the reasoning steps typically become part of the overall response, sometimes without there being a clear distinction. While the latter might be useful in certain cases like solving math problems, in general the user expects to receive a response without excessive intermediate reasoning steps. Hiding the thought part allows it to take many forms that are usually not interesting to the user: making mistakes, drafting responses and evaluating them, trying to understand the question better, etc. Of course, we can also give an option to reveal the thought part to the user for the purpose of interpretability and for analysing the underlying thought process behind the response.

In theory, the thoughts can take any form that is comprised of generated tokens, and do not even have to be in natural language. Their primary goal is to allow the model to perform extra computation to improve the quality of the response (Pfau et al., 2024). However, thoughts in natural language have several benefits such as taking advantage of human-written LLM pre-training data, and allowing humans to inspect and interpret the behaviour of the model. Hence, we use this setting because current LLMs can generate thoughts well in natural language.

2.2. Optimizing Thoughts via Preference Optimization

While our initial thought prompting generates thoughts via the instruction tuned model, they are not optimized to be actually useful in making the response better. We find they typically underperform *thoughtless* direct responses, which instruction-tuned LLMs have been heavily optimized for. Therefore, we need to train our model so it makes better

use of thought generation. We employ the Reinforcement Learning from AI Feedback (RLAIF) paradigm (Bai et al., 2022; Zhu et al., 2024) where we generate from the model and rank its responses using a reward model that acts as a judge. In particular, we use iterative Direct Preference Optimization (DPO) (Rafailov et al., 2024; Xu et al., 2023) for its simplicity and efficacy.

Unlike conventional RLAIF, we will not feed the whole model output to the judge. Instead, the judge can only see the response part of the outputs, so the thought part cannot influence its judgement. We chose this approach for several reasons. First, there is a lack of a judge model that is capable of evaluating internal thoughts. Building such a judge is inherently challenging because it is hard to collect human thoughts. In any case, even if such data was collected, it is not clear if human-written thoughts will be equally useful for LLMs. Secondly, the ultimate goal is to provide better responses to the user. Thus, it might be better to optimize the final objective instead of relying on an auxiliary objective that might not align well.

Our training starts with a seed model \mathcal{M}_0 that is instruction-tuned to directly respond to the user instruction. We also need a dataset of user instructions $\{x_i\}$ to begin training the model. At each training iteration t , we feed instructions to the current model \mathcal{M}_t along with our thought prompt p as described in Section 2.1:

$$\mathcal{M}_t(p + x_i) \rightarrow \{z_i^k, y_i^k\}.$$

Here “+” means the prompts are concatenated as input context to the LLM. For each input, we sample $k \leq K$ outputs, each containing thought z_i^k and response y_i^k parts.

Building Preference Pairs After extracting the response parts y_i^k , we feed them to the judge model \mathcal{J} for scoring. For pointwise judge models that take a single response and output a scalar score, the process is simple:

$$\mathcal{J}(x_i, y_i^k) \rightarrow s_i^k \in \mathbb{R}.$$

We also consider judge models that take a pair of responses and output the winner. In this case we apply the judge model to all possible pairs $\{y_i^m, y_i^n\}$ from the set of responses. This includes swapping positions in order to reduce the position-bias of the judge. Once we have all pairwise winners, we convert those to individual pointwise scores s_i^k using ELO scoring as performed in Wu et al. (2024). See Appendix B for more details.

Next, we select the highest and lowest scoring responses as “chosen” and “rejected” samples to construct a preference pair. Note that the preference pairs contain both thought and response parts.

$$\begin{aligned} \text{Pair} = \{p + x_i \rightarrow z_i^c + y_i^c; p + x_i \rightarrow z_i^r + y_i^r\} \\ \text{where } c = \operatorname{argmax}_k s_i^k \quad r = \operatorname{argmin}_k s_i^k. \end{aligned}$$

Table 1. Benchmark win rates (%) for AlpacaEval (length-controlled (LC)) and Arena-Hard. We compare our method Thought Preference Optimization (TPO) to the direct response baseline, Llama-3-8B-Instruct, and Llama-3-8B-Instruct using Thought Prompting. The latter, which does not perform well, is used as initialization for the first iteration of TPO training. TPO optimizes thought generation during iterative training, which then outperforms the baselines. We also include several well-known LLMs as reference which are typically larger than our TPO model.

Method	AlpacaEval (LC)	Arena-Hard
<i>Llama-3-8B-Instruct-based</i>		
Llama-3-8B-Instruct	24.9	20.6
Llama-3-8B-Instruct + Thought prompt	17.3	14.1
Direct response baseline	48.4	33.0
TPO	52.5	37.3
<i>Larger models</i>		
GPT-4 (06/13)	30.2	37.9
Llama-3-70b-instruct	34.4	46.6
Mistral Large (24/02)	32.7	37.7
Qwen2 72B Instruct	38.1	36.1

Using this process, the model can learn which thought led to a better response.

Iterative Training Once we have built preference pairs, we use them with the DPO loss to train the current model \mathcal{M}_t . This gives us a new model \mathcal{M}_{t+1} that will be used for the next training iteration. Note that we do not use data derived from previous iterations for training the current iteration, under the assumption that they are lower quality. In addition to DPO, we also experiment with the IRPO loss (Pang et al., 2024) that combines DPO with the NLL loss.

Length-Control It is known that some judge models tend to favor longer responses (Dubois et al., 2024; Yuan et al., 2024a). This length-bias causes the response length to grow with each training iteration, resulting in an overly verbose model. To mitigate this, we implement a length-control (LC) mechanism. Let us define a normalization function $N(l_i^k) = (l_i^k - \text{mean}_k(l_i^k)) / \text{std}_k(l_i^k)$. We recompute the scores by penalizing longer responses

$$s_i^k \leftarrow N(s_i^k) - \rho N(l_i^k).$$

The hyper-parameter ρ controls the strength of the length-control mechanism. Note we normalize both the score and the length to align them into a similar scale.

3. Experiments

3.1. Setup

We use Llama-3-8B-Instruct (Dubey et al., 2024) as a seed model in our training. As a judge model, we consider two choices of model: Self-Taught Evaluator (STE) (Wang et al., 2024b) and ArmoRM (Wang et al., 2024a). STE is a LLM-as-a-Judge model based on Llama-3-70B-Instruct. Given

two responses, it outputs its preference in natural language after generating a CoT. ArmoRM is a 8B reward model that directly outputs a scalar score to a single response.

For initial experiments, we use the synthetic instructions from Yuan et al. (2024b) for training. These instructions are generated from Llama-2-70B-Chat using 8-shot prompting consisting of random samples from the Open Assistant dataset (Köpf et al., 2024). For later experiments, we switched to UltraFeedback (Cui et al., 2023), which contains actual human instructions. Each training iteration uses 5000 instructions that were not part of the previous iterations.

We generate $K = 8$ responses per prompt using temperature 0.8 and top-p of 0.95. We train for 10 epochs in each iteration and select the best checkpoint using a validation set of 1500 prompts randomly sampled from UltraFeedback. We perform up to 4 iterations. We usually set the length-control parameter $\rho \in [0, 0.5]$, with 0 equivalent to no length-control. Unless otherwise specified, we use the specific thought prompt trained using the ArmoRM judge on UltraFeedback instructions as the default setup.

As a baseline, we train the same seed model that outputs responses directly without any thinking (note, this can still perform CoT as a part of the response due to its initial instruction training). We train this baseline in the exactly same way, using the same judge, data and loss. This allows us to directly measure the effect of the thoughts on response quality.

For evaluation, we use two public benchmarks that test general instruction following capability: AlpacaEval 2 (Dubois et al., 2024) and Arena-Hard (Li et al., 2024). Both benchmarks perform auto-evaluation using gpt-4-1106-preview as a judge. In addition, we also create our own benchmark to perform fine-grained analysis.

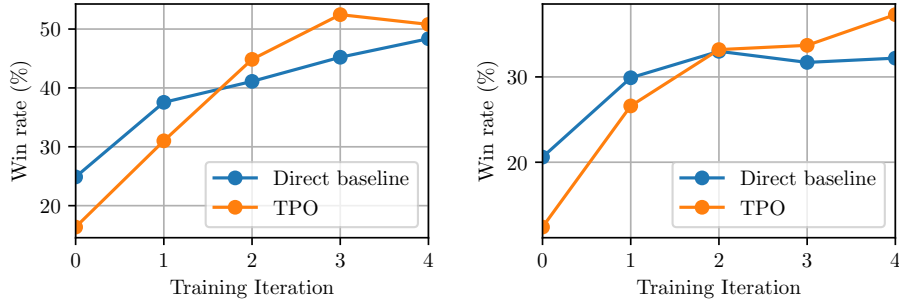


Figure 3. Comparing Thought Preference Optimization to the direct baseline in each iterations on AlpacaEval (left) and Arena-Hard (right).

3.2. AlpacaEval Results

The highest win rate our model TPO achieves is 52.5%, which is +4.1% better than the direct baseline, as shown in Table 1. It is also a +27.6% increase over the seed model and puts our method in 3rd position on the leaderboard¹, just after GPT-4 Omni and GPT-4 Turbo. This is an impressive result given the small size (8B) of our model.

In Figure 3, we plot the win rate for different iterations of training. We can see that before training (iteration 0) the direct baseline performs much better. This is expected as the seed model is instruction tuned to directly output a response. Simply prompting the model to write down its thought processes actually hurts performance. This agrees with the findings by Sprague et al. (2024) who showed CoT prompting only helps math and logic related tasks.

However, after several iterations of training TPO catches up with the baseline, even outperforming it in iteration 3 and 4. This is a promising indication that the model is adapting to think in a way that uses those thoughts to improve its responses. A more detailed breakdown of results is given in Appendix Table 5. The best performing setup uses UltraFeedback instructions for training with an ArmoRM judge for both TPO and direct models. For TPO, both generic and specific thought prompts produced similar outcomes, but the latter performs slightly better (both Table 1 and Figure 3 use this setup).

3.3. Arena-Hard Results

Results on the Arena-Hard benchmark are shown in Table 1 (right) and Figure 3 (right). They follow a similar trend to the results from AlpacaEval. Thinking performance is poor with the initial seed model at the start of training, but with more training iterations it matches the direct baseline and starts to outperform it. TPO reaches a win rate of 37.3%, which is +4.3% better than the baseline. This makes

¹As of Jan. 30th 2025. https://tatsu-lab.github.io/alpaca_eval/

our model the best model on the leaderboard with a such small size². It performs similarly to much larger models like GPT-4 (06/13) or Mistral Large (24/02). Detailed results comparing different experimental setups are shown in Appendix Table 6.

3.4. Fine-grained Evaluation

While the above benchmarks evaluate overall performance, they lack granularity to inform which types of instructions benefit from thinking. To obtain a more fine-grained evaluation, we build our own evaluation using UltraFeedback. We take instructions not used in training, and assign them individually to one of 20 categories until each category has 200 samples.

To measure the performance on this dataset, we compare responses generated from our TPO model against responses from the direct baseline model. We use the evaluation prompt from Arena-Hard and GPT4 (gpt-4-1106) as a judge. The ordering of responses is randomized to reduce position-bias. Figure 4 shows the win rates of TPO on all 20 categories. Surprisingly, we observe that non-reasoning categories obtain large gains through thinking. This includes language and translation, marketing and health. We also see improvement in reasoning categories like research and analysis, math and calculations. See Appendix A for more details and additional results.

3.5. Analysis and Ablations

In this section, we present analysis and several ablation results to further understand TPO.

Qualitative analysis of thoughts In Figure 6, we present an example of a non-reasoning task to illustrate the broader utility of thinking. While writing a poem is not typically

²Excluding some closed models like GPT-4 where the size is not disclosed. Compared against Aug. 31st 2024 version. <https://github.com/lm-sys/arena-hard-auto>

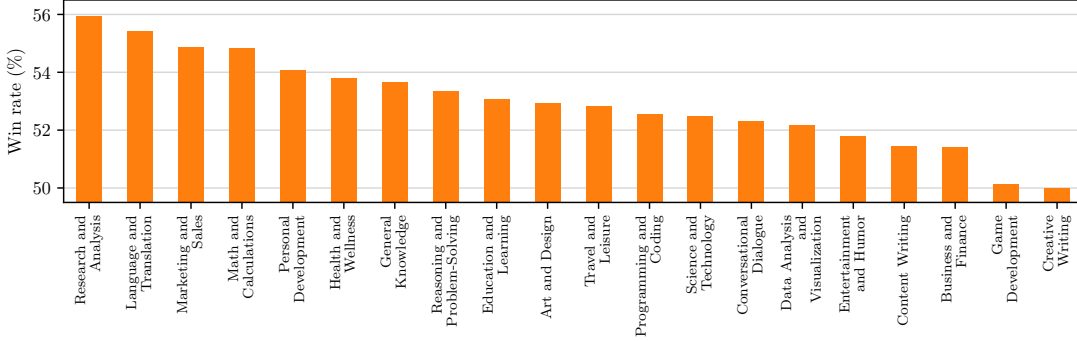


Figure 4. **Fine-Grained evaluation on unseen instructions from UltraFeedback, broken down by category.** We measure the win rate of TPO against the direct baseline as judged by GPT4.

viewed as a reasoning task, it can benefit from better planning and understanding of the instruction. This is also a good example of when it makes sense to hide the thought process. Figure 5 shows the thinking of a TPO model trained with the specific thought prompt on a factoid question (which dog breed is smallest). We see that the model thinks first about the question, then evaluates its draft response. We provide more thought examples in Appendix C.

Thought Prompt Types Table 2 compares different thought prompt types in terms of different metrics across iterations. As we have observed previously, the seed model performs poorly when it is asked to generate thoughts compared to directly generating a response, which is the result of the model being well optimized to do the latter. However, after one iteration of training, we see the gap is shrinking. Between the two thought prompt types we try, there is not much difference in terms of win rate, but the thought and answer lengths vary greatly. Also, the specific thought prompt leads to longer responses and have a higher average ArmoRM score (see Appendix Figure 21).

Thought lengths As shown in Table 2, the specific thought prompt has the longest thought generations because it asks the model to generate a draft response and evaluate it within the thought. Although we did not directly supervise the thought process, we found the model learns to shorten and condense the thought throughout the training. Table 2 shows the length changes after 1 training iteration, but more detailed statistics across training are given in Appendix Table 4. We find that after 4 iterations of training, the thought part is shrunk by 61% and 30% for the generic and specific thought prompts respectively. We also measure thought lengths by category (Appendix Figure 14) and find that categories such as research, art and writing have longer thoughts, while conversation, language, math and general knowledge are on the opposite end.

Table 2. Comparing the generic and specific (Figure 2) thought prompts. We measure the length (in characters) of the thought and final answer on the UltraFeedback train prompts, as well as AlpacaEval (LC) win rates before and after one training iteration.

Type	Seed Model / Iter 1		
	Thought Len	Response Len	Win Rate
Direct	-	2878 / 2787	24.9% / 37.5%
Generic	799 / 606	1717 / 1638	17.3% / 32.7%
Specific	2193 / 1613	1300 / 1431	16.4% / 31.0%

Length-Control We find that length-control (LC) is a must during training, especially with the STE judge. When no LC is performed, the average response length grows rapidly (Wu et al., 2024). For instance, after just 1 iteration of training of the direct baseline, the average response length grows around 15%. If we utilize our LC technique when building preference pairs, we can maintain the same length or even decrease the length if necessary. In our experiments, we tune the LC coefficient ρ by measuring the average length of chosen and rejected samples, and choose the smallest value such that the former is not longer than the latter. This simple method was very effective, and both TPO and the direct baseline model did not grow their response length much during training, as demonstrated in Table 2.

Training Instruction set and Judges We experiment with two training instruction sets: the synthetic instruction set generated by few-shot prompting Llama-2-70B, and UltraFeedback - a more curated instruction set that contains human instructions. As shown in Appendix Table 5 and Table 6, we do not observe a large difference between these two datasets in terms of final performance on the benchmarks. We also experimented with two different judges for training, ArmoRM and the STE judge. We find that when using either judge TPO outperforms the direct baseline on AlpacaEval as shown in Table 5. However, STE required

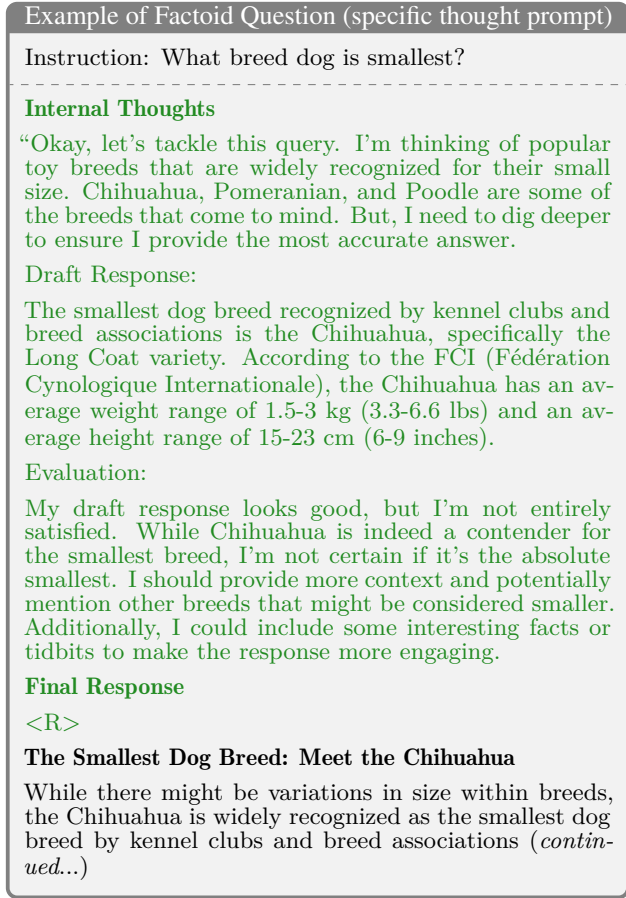


Figure 5. Example of TPO answering a simple factoid question. This model is trained with the specific thought prompt, so it writes a draft and evaluates it in the thought part (in green).

more length-control to be applied during training. On both benchmarks, we obtained better results with the ArmoRM judge.

Parsing errors For the initial seed model we use prompting to maintain the desired output format, where thoughts and answer are separated, see Figure 2. However, we find during training that without additional safeguards, the rate of parse errors keep fluctuating, and is sometimes hard to control. In order to teach the model to follow the constraint, we add responses with parse errors as rejected examples in the preference data creation process. Typically, the ratio of parse errors as rejected is no more than 10% in each iteration, but we observe for the specific prompt template, it is much harder to control. We found that sampling again if a parse error occurred helped alleviate the issue during evaluations.

DPO vs IRPO loss In addition to training with the DPO loss, we also experimented the IRPO loss that has shown promising results in math tasks (Pang et al., 2024). IRPO

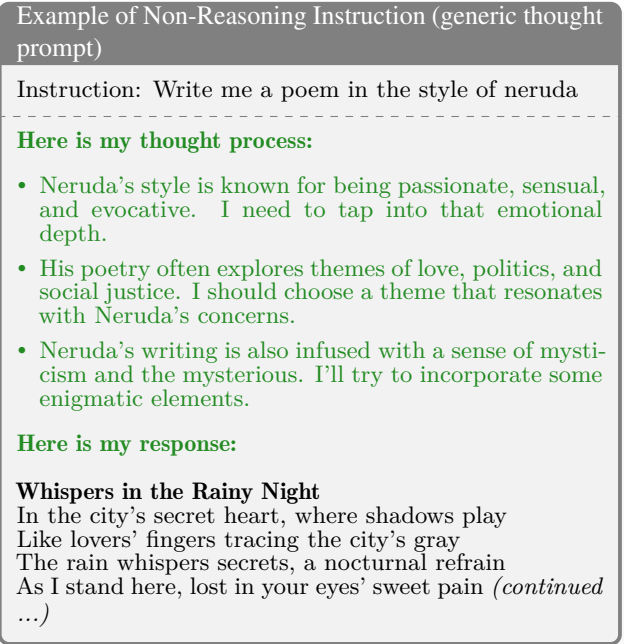


Figure 6. Example of TPO responding to a non-reasoning instruction to write a poem. This example demonstrates how thinking can be useful in understanding the task and planning accordingly even for creative writing. The thought part in green is not part of the response.

adds an NLL loss on the chosen samples, so that their average log probability does not decrease, which can occur with DPO. However, in our experiments and setting, we did not see a noticeable difference in performance. After one training iteration, TPO using IRPO gives a 31.6% AlpacaEval LC win rate, which almost matches the 32.0% of DPO. Here we used the STE judge and the synthetic dataset for training.

Math Domain To further understand the performance on the math domain where CoT techniques are often applied, we evaluate our model on the GSM8K dataset (Cobbe et al., 2021) that contains grade-school math word problems. Since we have the correct answers in this dataset, we can more accurately measure performance compared to relying on a judge. We first test the seed model without a thought prompt, but observe it uses CoT anyway due to its instruct training, obtaining 79.2% accuracy. We thus append “Output only the number answer.” to the problems so that the model will not perform CoT in its response. It still performs surprisingly well in this answer-only mode, reaching 69.7% accuracy. However, on closer inspection we noticed that the seed model still performs CoT-like arithmetic operations like “16 - 3 - 4 = 9. 9 * 2 = 18” in its output. We then evaluate direct models trained on the synthetic prompts and the STE judge. After training, this number drops to 51.3%. Performance is even lower for TPO models. This indicates

that our experimental setup is not suited for math tasks. In fact, only 2.2% of our training instructions are categorized into the math category (see Appendix Figure 7). Such a drop in math performance is also observed by Meng et al. (2024); Zhu et al. (2024) in a general instruction tuning setup. However, we do see some examples where the TPO model is able to correct its mistake through reflection, while the direct model simply stops at the mistake, as shown in Appendix Figure 20. For more detailed results, see Appendix Table 3.

Computational Cost The primary computational costs of TPO arise from two main stages: first, the generation of responses, which involves thoughts and responses, and second, the subsequent training process (DPO) that utilizes these generated responses. The length of these responses and their associated ‘thoughts’ is a key factor in this cost. According to Table 2, initially, the total response length with the generic template was 12.57% less than the Direct baseline, while the specific template yielded responses 21.36% longer. After the first iteration, the generic template’s responses were 19.48% shorter, and the specific template’s responses were 9.22% longer than the baseline. Thus, while the computational expense relative to the baseline varies depending on the template — some configurations being less resource-intensive and others more so — the aggregate impact on computational cost remains relatively minor.

4. Related work

Reasoning in Language vs. Vectors: Thinking in words takes advantage of the natural language understanding capability of LLMs. While thinking in continuous values might provide more bandwidth, the Transformer architecture already can compute continuous vectors as hidden states and feed them to the next layer. However, these hidden vectors do not feed back into the model at the next token, and thus are not accessible to the future lower layers (Fan et al., 2020). Word tokens on the other hand are fed back to the model immediately, i.e. during inference the previous output token is fed as input to predict the next token – making it possible to condition all future computations on them (Merrill & Sabharwal, 2024). Another advantage of word tokens is that there exist simple sampling mechanisms which allow thoughts to take different paths each time (Wang et al., 2023), which can be used to improve results e.g. via majority vote.

Chain-of-Thought (CoT): CoT prompting (Wei et al., 2022) demonstrated that LLMs perform better at reasoning tasks when they are encouraged to write down intermediate reasoning steps. Since the type of thinking in CoT is dictated by the prompt instruction, there are now many different variations of it facilitating different types of reasoning, such as decomposing into smaller problems (Zhou et al., 2023).

It is now widely used for math and reasoning tasks, and most current LLMs are finetuned to do CoT by default for those types of tasks (Dubey et al., 2024). Other works like Pfau et al. (2024) show that the model equipped with CoT might be able to perform hidden thinking using filler tokens. However, CoT usage has had more limited use in other types of tasks. Meta-analysis by Sprague et al. (2024) found that CoT techniques have little benefit outside of math and logic related tasks.

Training to Think: Nye et al. (2021) trained a model to write intermediate calculations into a scratchpad section before writing the final answer, which improved performance in math and coding tasks. Similarly Lehnert et al. (2024) showed that Transformers can solve complex planning tasks if they are trained to write A* search traces before outputting the solution. However, these methods rely on supervised training so ground-truth thought data is required. STaR (Zelikman et al., 2022) removes this constraint by generating both thought and answer from a model using few-shot prompting. Then the generations are filtered by the correctness of the answer to be used for supervised finetuning. Its generalization Quiet-STaR (Zelikman et al., 2024) aims to insert thought segments into unstructured text. This involves sampling a sequence of thought tokens after every input token, then training using a REINFORCE based loss that optimizes the likelihood of subsequent input tokens. While it showed promising results in multi-choice reasoning and math tasks, the training mechanism is complex and compute heavy. V-STaR (Hosseini et al., 2024) trained a DPO verifier on both correct and incorrect solutions and uses the verifier to select the response in inference time. IRPO (Pang et al., 2024) also trains a variant of DPO on math and reasoning problems to learn CoTs, assuming access to gold labels on the training set. Similarly, Self-Notes (Lanchantin et al., 2023) allows the model to deviate from the input at any time to write its thoughts, but relied on supervised training data in symbolic tasks. None of these methods have been applied to general instruction following using LLMs. Concurrent work DeepSeek-R1 (Guo et al., 2025) used RL to finetune the base model with a structural thinking template against a rule based reward and shown impressive results on par with OpenAI-O1 (OpenAI).

System 2 methods: Many system 2 methods emerged in recent years that add intermediate steps at inference time before producing a final answer. Those steps typically involve prompting the model with a certain goal, such as verification of the answer (Dhuliawala et al., 2024), rephrasing user questions (Deng et al., 2023), selecting sentences to attend to (Weston & Sukhbaatar, 2023), etc. Briakou et al. (2024) developed a method for translation incorporating intermediate steps of drafting and revising. Our method has a similarity with these methods in the first step because it

uses prompting on the initial seed model, but further optimizes the thoughts during training iterations. Concurrent work by Kumar et al. (2024a) trains models to self-correct, while Yu et al. (2024) distill system 2 methods into system 1 with supervised finetuning. Rather than focusing on general thinking, these works teach the model specific skills.

5. Conclusion

We introduced Thinking LLMs, which think in natural language before writing a response for *general instruction-following tasks*. To train such models, we proposed a new training recipe called Thought Preference Optimization for teaching Thinking LLMs to improve their thoughts. Unlike prior methods (Snell et al., 2024; Kumar et al., 2024b), which directly supervise the thought generation process through techniques like self-correction or self-refinement, we instead *provide incentives for the model to generate its own thoughts*, without explicitly teaching it how to think. In our experiments, we train and evaluate the models in the general instruction following setup. The results show that, although the early iteration of the Thinking LLM perform poorly compared to the typical direct response model, after multiple iterations of training using TPO, our method outperforms the baseline. Further, fine-grained evaluations reveal that thinking helps in categories that are not usually associated with reasoning or chain-of-thought methods. This is an encouraging result and hopefully leads to wider adoption of Thinking LLMs in non-reasoning domains.

Impact Statement

We experimented with two different thought prompts, and observed some performance differences between them. It is likely that certain thought types are suited for certain tasks, and direct responses would even work better in certain situations. Therefore, training on a diverse set of thought prompts and allowing the model to switch between them could potentially lead to further improvements in performance.

While we see improvement in overall performance with TPO, evaluation on GSM8K showed degraded math performance. As we discussed, this is likely due to our setup not being oriented toward such tasks. Incorporating more math instructions during training and having access to a judge capable of evaluating of their answers are likely solutions.

In the current version of the method, thought lengths are purely determined by model itself. There is no steerability in terms of changing the number of thought tokens. Adding such functionality could be useful as longer thoughts increase computation and corresponding cost per user instruction. We could use techniques like Yuan et al. (2024a) for this purpose.

All our experiments are based on 8B parameter sized models. However, it is worth investigating the effect of thinking on larger scale models. Given the compute requirements of such experiments, we leave that to future work.

References

- Bai, Y., Kadavath, S., Kundu, S., Askell, A., Kernion, J., Jones, A., Chen, A., Goldie, A., Mirhoseini, A., McKinnon, C., et al. Constitutional ai: Harmlessness from ai feedback. *arXiv preprint arXiv:2212.08073*, 2022.
- Briakou, E., Luo, J., Cherry, C., and Freitag, M. Translating step-by-step: Decomposing the translation process for improved translation quality of long-form texts. *arXiv preprint arXiv:2409.06790*, 2024.
- Cobbe, K., Kosaraju, V., Bavarian, M., Chen, M., Jun, H., Kaiser, L., Plappert, M., Tworek, J., Hilton, J., Nakano, R., et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.
- Cui, G., Yuan, L., Ding, N., Yao, G., Zhu, W., Ni, Y., Xie, G., Liu, Z., and Sun, M. Ultrafeedback: Boosting language models with high-quality feedback, 2023.
- Deng, Y., Zhang, W., Chen, Z., and Gu, Q. Rephrase and respond: Let large language models ask better questions for themselves. *arXiv preprint arXiv:2311.04205*, 2023.
- Dhuliawala, S., Komeili, M., Xu, J., Raileanu, R., Li, X., Celikyilmaz, A., and Weston, J. Chain-of-verification reduces hallucination in large language models. In *ACL (Findings)*, pp. 3563–3578. Association for Computational Linguistics, 2024.
- Dubey, A., Jauhri, A., Pandey, A., Kadian, A., Al-Dahle, A., Letman, A., Mathur, A., Schelten, A., Yang, A., Fan, A., et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.
- Dubois, Y., Liang, P., and Hashimoto, T. Length-controlled alpacaEval: A simple debiasing of automatic evaluators. In *First Conference on Language Modeling*, 2024.
- Fan, A., Lavril, T., Grave, E., Joulin, A., and Sukhbaatar, S. Addressing some limitations of transformers with feedback memory. *arXiv preprint arXiv:2002.09402*, 2020.
- Guo, D., Yang, D., Zhang, H., Song, J., Zhang, R., Xu, R., Zhu, Q., Ma, S., Wang, P., Bi, X., et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.
- Hosseini, A., Yuan, X., Malkin, N., Courville, A., Sordani, A., and Agarwal, R. V-STar: Training verifiers for self-taught reasoners. In *First Conference on Language Modeling*, 2024.

- Köpf, A., Kilcher, Y., von Rütte, D., Anagnostidis, S., Tam, Z. R., Stevens, K., Barhoum, A., Nguyen, D., Stanley, O., Nagyfi, R., et al. Openassistant conversations-democratizing large language model alignment. *Advances in Neural Information Processing Systems*, 36, 2024.
- Kumar, A., Zhuang, V., Agarwal, R., Su, Y., Co-Reyes, J. D., Singh, A., Baumli, K., Iqbal, S., Bishop, C., Roelofs, R., Zhang, L. M., McKinney, K., Shrivastava, D., Paduraru, C., Tucker, G., Precup, D., Behbahani, F., and Faust, A. Training language models to self-correct via reinforcement learning. 2024a.
- Kumar, A., Zhuang, V., Agarwal, R., Su, Y., Co-Reyes, J. D., Singh, A., Baumli, K., Iqbal, S., Bishop, C., Roelofs, R., et al. Training language models to self-correct via reinforcement learning. *arXiv preprint arXiv:2409.12917*, 2024b.
- Lanchantin, J., Toshniwal, S., Weston, J. E., Szlam, A., and Sukhbaatar, S. Learning to reason and memorize with self-notes. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023.
- Lehnert, L., Sukhbaatar, S., Mcvay, P., Rabbat, M., and Tian, Y. Beyond A*: Better planning with transformers via search dynamics bootstrapping. In *First Conference on Language Modeling*, 2024.
- Li, T., Chiang, W.-L., Frick, E., Dunlap, L., Wu, T., Zhu, B., Gonzalez, J. E., and Stoica, I. From crowdsourced data to high-quality benchmarks: Arena-hard and benchbuilder pipeline. *arXiv preprint arXiv:2406.11939*, 2024.
- Meng, Y., Xia, M., and Chen, D. Simpo: Simple preference optimization with a reference-free reward. *arXiv preprint arXiv:2405.14734*, 2024.
- Merrill, W. and Sabharwal, A. The expressive power of transformers with chain of thought. In *The Twelfth International Conference on Learning Representations*, 2024.
- Nye, M., Andreassen, A., Gur-Ari, G., Michalewski, H., Austin, J., Bieber, D., Dohan, D., Lewkowycz, A., Bosma, M., Luan, D., Sutton, C., and Odena, A. Show your work: Scratchpads for intermediate computation with language models. *ArXiv*, abs/2112.00114, 2021.
- OpenAI. Introducing openai o1-preview, 2024. URL <https://openai.com/index/introducing-openai-o1-preview/>.
- Pang, R. Y., Yuan, W., Cho, K., He, H., Sukhbaatar, S., and Weston, J. Iterative reasoning preference optimization. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024.
- Pfau, J., Merrill, W., and Bowman, S. R. Let’s think dot by dot: Hidden computation in transformer language models. *arXiv preprint arXiv:2404.15758*, 2024.
- Rafailov, R., Sharma, A., Mitchell, E., Manning, C. D., Ermon, S., and Finn, C. Direct preference optimization: Your language model is secretly a reward model. *Advances in Neural Information Processing Systems*, 36, 2024.
- Snell, C., Lee, J., Xu, K., and Kumar, A. Scaling llm test-time compute optimally can be more effective than scaling model parameters. *arXiv preprint arXiv:2408.03314*, 2024.
- Sprague, Z., Yin, F., Rodriguez, J. D., Jiang, D., Wadhwa, M., Singhal, P., Zhao, X., Ye, X., Mahowald, K., and Durrett, G. To cot or not to cot? chain-of-thought helps mainly on math and symbolic reasoning. *arXiv preprint arXiv:2409.12183*, 2024.
- Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L. u., and Polosukhin, I. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30, 2017.
- Wang, H., Xiong, W., Xie, T., Zhao, H., and Zhang, T. Interpretable preferences via multi-objective reward modeling and mixture-of-experts. *arXiv preprint arXiv:2406.12845*, 2024a.
- Wang, T., Kulikov, I., Golovneva, O., Yu, P., Yuan, W., Dwivedi-Yu, J., Pang, R. Y., Fazel-Zarandi, M., Weston, J., and Li, X. Self-taught evaluators. *ArXiv*, abs/2408.02666, 2024b.
- Wang, X., Wei, J., Schuurmans, D., Le, Q. V., Chi, E. H., Narang, S., Chowdhery, A., and Zhou, D. Self-consistency improves chain of thought reasoning in language models. In *The Eleventh International Conference on Learning Representations*, 2023.
- Wei, J., Wang, X., Schuurmans, D., Bosma, M., brian ichter, Xia, F., Chi, E. H., Le, Q. V., and Zhou, D. Chain of thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems*, 2022.
- Weston, J. and Sukhbaatar, S. System 2 attention (is something you might need too). *arXiv preprint arXiv:2311.11829*, 2023.
- Wu, T., Yuan, W., Golovneva, O., Xu, J., Tian, Y., Jiao, J., Weston, J., and Sukhbaatar, S. Meta-rewarding language models: Self-improving alignment with llm-as-a-meta-judge. *ArXiv*, abs/2407.19594, 2024.

- Xu, J., Lee, A., Sukhbaatar, S., and Weston, J. Some things are more cringe than others: Iterative preference optimization with the pairwise cringe loss. *arXiv preprint arXiv:2312.16682*, 2023.
- Yu, P., Xu, J., Weston, J., and Kulikov, I. Distilling system 2 into system 1. *arXiv preprint arXiv:2407.06023*, 2024.
- Yuan, W., Kulikov, I., Yu, P., Cho, K., Sukhbaatar, S., Weston, J., and Xu, J. Following length constraints in instructions. *arXiv preprint arXiv:2406.17744*, 2024a.
- Yuan, W., Pang, R. Y., Cho, K., Li, X., Sukhbaatar, S., Xu, J., and Weston, J. E. Self-rewarding language models. In *Forty-first International Conference on Machine Learning*, 2024b.
- Zelikman, E., Wu, Y., Mu, J., and Goodman, N. STar: Bootstrapping reasoning with reasoning. In *Advances in Neural Information Processing Systems*, 2022.
- Zelikman, E., Harik, G. R., Shao, Y., Jayasiri, V., Haber, N., and Goodman, N. Quiet-STar: Language models can teach themselves to think before speaking. In *First Conference on Language Modeling*, 2024.
- Zhao, W., Ren, X., Hessel, J., Cardie, C., Choi, Y., and Deng, Y. Wildchat: 1m chatgpt interaction logs in the wild. *International Conference on Learning Representations*, 2024.
- Zhou, D., Schärli, N., Hou, L., Wei, J., Scales, N., Wang, X., Schuurmans, D., Cui, C., Bousquet, O., Le, Q. V., and Chi, E. H. Least-to-most prompting enables complex reasoning in large language models. In *ICLR*, 2023.
- Zhu, B., Frick, E., Wu, T., Zhu, H., Ganesan, K., Chiang, W.-L., Zhang, J., and Jiao, J. Starling-7b: Improving helpfulness and harmlessness with rlaiif. In *First Conference on Language Modeling*, 2024.

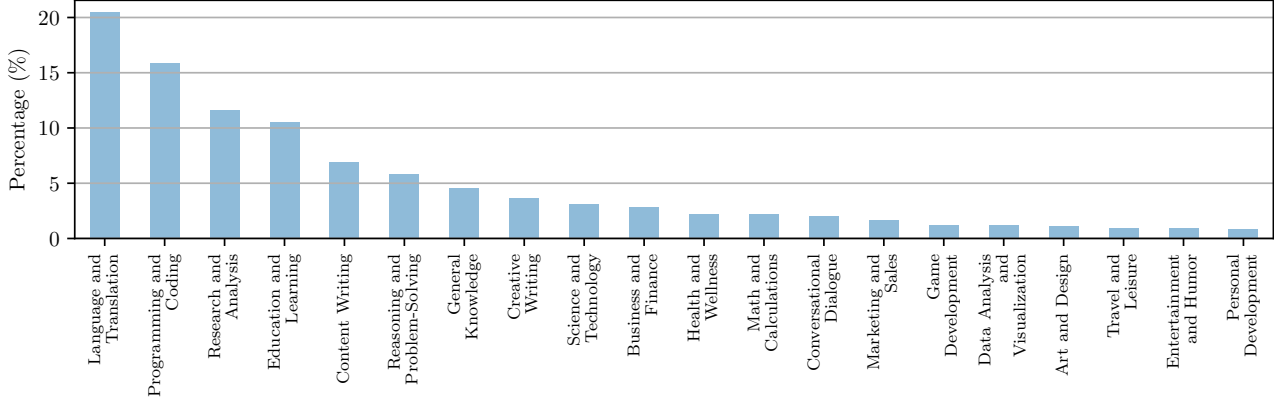


Figure 7. Training instruction distribution: we categorize our training instructions into 20 topics. The language and translation category has the most examples, while the math category is only 2.2% of the data.

A. Fine-grained evaluation details

We first generate 20 category names by prompting Llama-3.1-70B-Instruct model to summarize the categories present given many instructions from WildChat (Zhao et al., 2024). We select 200 instructions from UltraFeedback per category for evaluation, excluding those that were used in training. Figure 7 shows the ratio of each category in our training instructions from UltraFeedback. This shows that some categories have much fewer samples in the training data, which might negatively affect their performance.

We show the prompts we used: to generate categories in Figure 10, assign categories in Figure 11, and evaluate with the GPT4-1106 judge in Figure 12. The judge can give evaluations of: $A \gg B$, $A > B$, $A = B$, $B > A$, $B \gg A$, which are counted as 1, 0.75, 0.5, 0.25, 0 respectively. The scores are averaged over instructions to determine the win rate in the category. Figure 8 (a) shows the win rate of the generic thought prompt, which is lower in average compared to the specific thought prompt (see Figure 4). However, it still outperforms the direct baseline on complex categories like math and coding.

To further validate our results, we perform a similar evaluation using an alternative judge, STE. The prompt we used with this judge is shown in Figure 13. Here, we test 500 instructions per category and evaluate in both orders. Figure 8 (b) and (c) show the win rate of the specific and generic thought prompts respectively. Again, we see that the specific thought prompt works better. Compared to the GPT4-based evaluations, we see more variation across the categories, perhaps due to the lack of a tie option in the judge. We also see that TPO does better when the training also uses the STE judge (Figure 8 (d)).

Overall, we observe that ranking of categories by the win rate varies a lot between different judges and thought prompt types, making it challenging to draw a conclusion about which category benefits most from thinking. However, several categories consistently improve from TPO such as math, language and translation, and reasoning.

In addition, we also classify AlpacaEval instructions into 20 categories following (Yuan et al., 2024b) and present the default benchmark win rates in Figure 9. We see large improvements in some categories like music and art, entertainment and cooking. This is interesting because those categories are not considered as reasoning tasks in the standard literature. However, note that some categories might not have sufficient samples (the total number of samples is only 805) even though we removed two categories that have less than 10 samples.

B. ELO computation

For each prompt x , we have up to K corresponding responses denoted as $\{y^1, \dots, y^K\}$. We then evaluate each pair of different responses (y^m, y^n) using the pairwise judge.

Table 3. GSM8K Accuracy when the instruction asked to output the final numeric answer only vs using the original prompts. Our training setup degrades performance of both the baseline and TPO models.

Model	Data	Training Iter	Original	Answer-Only
Direct baseline	-	0	79.2%	69.7%
Direct baseline	Synthetic	4	77.6%	51.3%
TPO	-	0	77.1%	57.2%
TPO (generic)	Synthetic	4	78.2%	21.4%
TPO (generic)	UltraFeedback	4	78.5%	10.2%
TPO (specific)	UltraFeedback	4	70.6%	43.4%

Table 4. Thought length (characters) of TPO on UltraFeedback validation set during training iterations. We tested models trained on UltraFeedback using ArmoRM as a judge.

Iteration	Generic thought prompt	Specific thought prompt
0 (seed)	815	2125
1	605	1560
2	467	1430
3	346	1320
4	314	1494

The result of a single battle between judgments (y^m, y^n) is defined as:

$$r^{mn} = \begin{cases} 1 & \text{If the pairwise-judge prefers } m \text{ wins} \\ -1 & \text{If the pairwise-judge prefers } n \text{ wins} \\ 0 & \text{If tie or error.} \end{cases}$$

We then construct a battle matrix B as the combination of the battle results:

$$B_{mn} = \mathbf{1}[r^{mn} = 1] + \mathbf{1}[r^{nm} = -1]$$

The next step is to convert the battle matrix into ELO score. Following the same process as in Wu et al. (2024), we determine the ELO score s^k for each response y^k by solving the following maximum likelihood estimation problem:

$$\arg \max_{s^k} \sum_{1 \leq m, n \leq K} B_{mn} \log \left(\frac{e^{s^m - s^n}}{1 + e^{s^m - s^n}} \right).$$

C. Thought Examples

Here we show several examples of thought and response outputs. Figure 15 is a non-reasoning question and TPO still takes advantage of thinking by recalling relevant information. Figure 16 is an example where the model solves complex problem by thinking of a draft that outlines the main steps before actually writing the code. In contrast, the direct response baseline makes a mistake, as shown in Figure 17.

We do observe some failure cases of thinking. For example, we observe the model learns to generate multiple round of the drafting-evaluation process even if it is not told to do so in the prompt. While this can be helpful, we find overthinking sometimes causes the model to get “lost in the thought”, without giving an final answer or else providing a wrong answer. An example of this is shown in Figure 18.

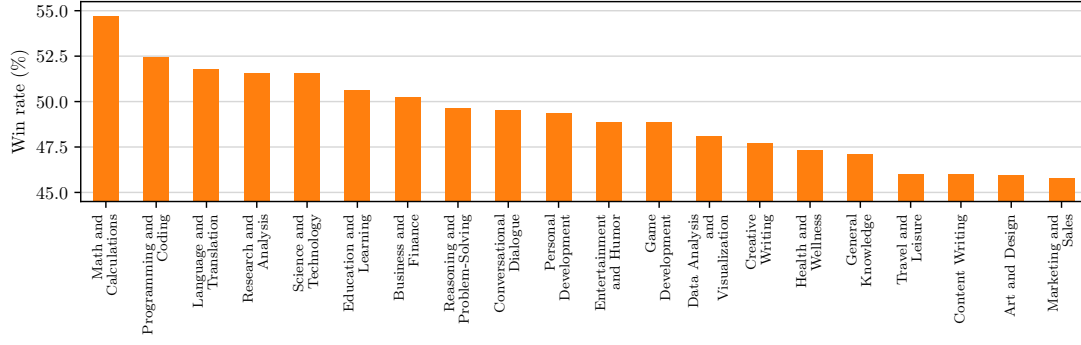
The next example shows how the model thinks before training in Figure 19. Here, the thought process is correct and helps to get closer to the answer, and the model only gets an addition wrong at the last step. This shows that thinking can be helpful, but the model is still bad at arithmetic.

Table 5. Breakdown of AlpacaEval results for different judge models, training instruction sets (“Data”) and Thought Prompts comparing our method, TPO, to the direct baseline. We report the standard LC win rate. In each case TPO outperforms the direct baseline, but only after several iterations of training.

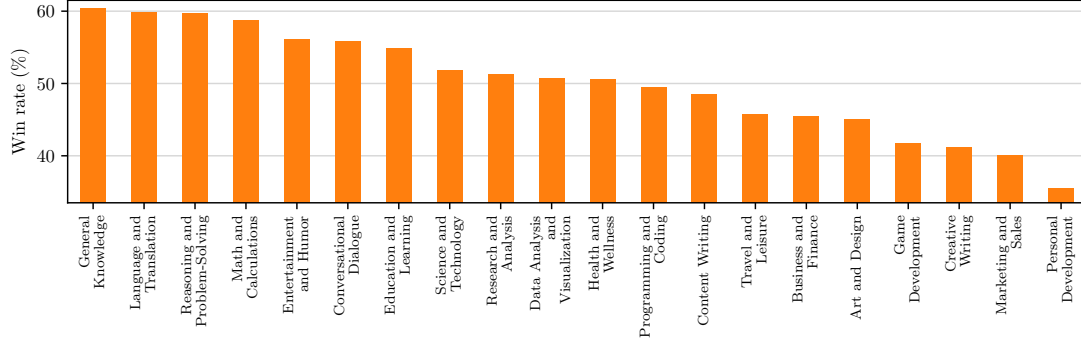
Model	Judge	Training Setup		Training Iteration				
		Data	Thought Prompt	0	1	2	3	4
Direct	STE	Synthetic	-	24.9	34.8	40.3	43.6	42.6
TPO	STE	Synthetic	Generic	17.3	32.0	40.2	47.6	46.8
Direct	Armo	Synthetic	-	24.9	35.3	43.2	45.7	48.1
TPO	Armo	Synthetic	Generic	17.3	33.8	44.1	49.8	51.1
Direct	Armo	UltraFeedback	-	24.9	37.6	41.1	45.2	48.4
TPO	Armo	UltraFeedback	Generic	17.3	32.8	43.0	47.0	50.2
TPO	Armo	UltraFeedback	Specific	16.4	31.0	44.9	52.5	50.8

Table 6. Breakdown of Arena-Hard results for varying experimental setups. We report the default win rates.

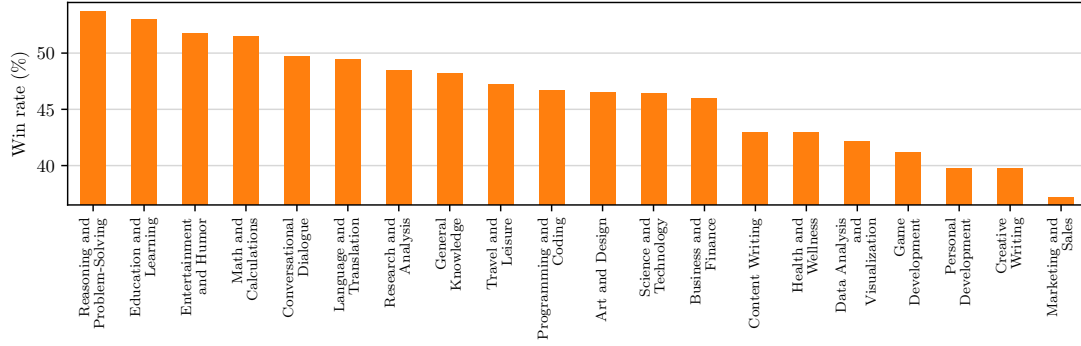
Model	Judge	Data	Thought Prompt	Training Iteration				
				1	2	3	4	
Direct	STE	Synthetic	-	20.6	29.6	32.5	33.1	33.2
TPO	STE	Synthetic	Generic	14.1	24.0	30.6	27.2	24.3
Direct	Armo	Synthetic	-	20.6	27.1	31.4	31.1	30.4
TPO	Armo	Synthetic	Generic	14.1	25.7	31.0	32.8	30.1
Direct	Armo	UltraFeedback	-	20.6	29.9	33.0	31.7	32.2
TPO	Armo	UltraFeedback	Generic	14.1	27.2	28.1	30.8	31.8
TPO	Armo	UltraFeedback	Specific	12.4	26.6	33.2	33.7	37.3



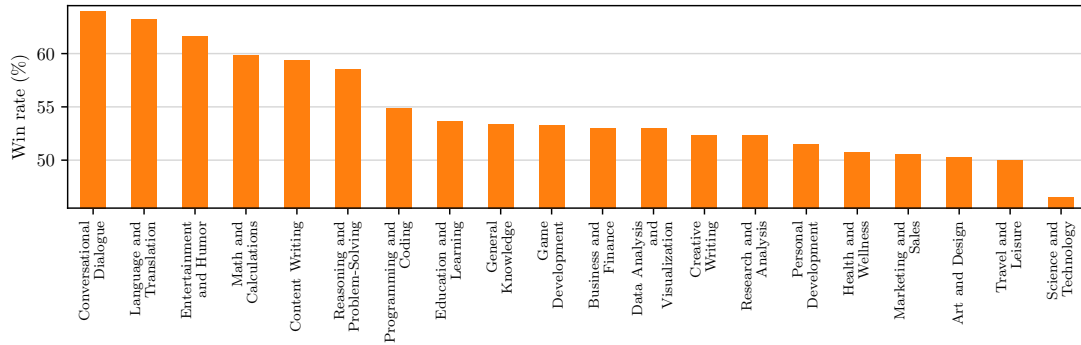
(a) Generic Thought Prompt judged by GPT4



(b) Specific Thought Prompt judged by STE



(c) Generic Thought Prompt judged by STE



(d) Generic Thought Prompt judged by STE when STE is also used during training (iteration 3)

Figure 8. Additional Fine-grained evaluation results. We show the win rate of TPO against the direct baseline trained in the same setup. The different plots correspond to setups differing in evaluation judge model, thought prompt type, and the training judge.

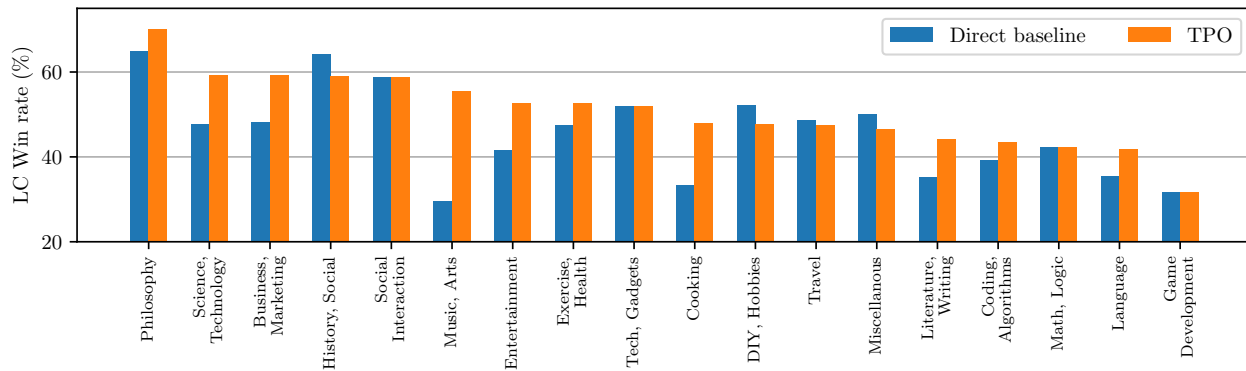


Figure 9. Fine-grained evaluation on AlpacaEval instructions. We measure length-controlled win rates within each category for each model.

Prompt to come up with categories

Given the following list of possible instructions, define a maximum of 20 categories that would cover the types of instructions, for example recipes, reasoning tasks, general knowledge etc. Try to cover as many of the instructions as possible with the maximum 20 categories, while keeping the categories high-level, concise, simple, and easy to understand.

<NEW INSTRUCTION>
{query}

<NEW INSTRUCTION>
{query}

...

Figure 10. Prompt to generate 20 categories based on a set of user instructions.

Prompt to assign categories

Below is an instruction that I would like you to analyze:

```
<instruction>  
{user instruction}  
</instruction>
```

Categorize the instruction above into one of the following categories:

- General Knowledge
- Math and Calculations
- Programming and Coding
- Reasoning and Problem-Solving
- Creative Writing
- Content Writing
- Art and Design
- Language and Translation
- Research and Analysis
- Conversational Dialogue
- Data Analysis and Visualization
- Business and Finance
- Education and Learning
- Science and Technology
- Health and Wellness
- Personal Development
- Entertainment and Humor
- Travel and Leisure
- Marketing and Sales
- Game Development
- Miscellaneous

Be sure to provide the exact category name without any additional text.

Figure 11. Prompt to assign one of the 20 categories to user instructions

Prompt to evaluate responses with GPT4

<system>

Please act as an impartial judge and evaluate the quality of the responses provided by two AI assistants to the user prompt displayed below. You will be given assistant A's answer and assistant B's answer. Your job is to evaluate which assistant's answer is better.

Begin your evaluation by generating your own answer to the prompt. You must provide your answers before judging any answers.

When evaluating the assistants' answers, compare both assistants' answers with your answer. You must identify and correct any mistakes or inaccurate information.

Then consider if the assistant's answers are helpful, relevant, and concise. Helpful means the answer correctly responds to the prompt or follows the instructions. Note when user prompt has any ambiguity or more than one interpretation, it is more helpful and appropriate to ask for clarifications or more information from the user than providing an answer based on assumptions. Relevant means all parts of the response closely connect or are appropriate to what is being asked. Concise means the response is clear and not verbose or excessive.

Then consider the creativity and novelty of the assistant's answers when needed. Finally, identify any missing important information in the assistants' answers that would be beneficial to include when responding to the user prompt.

After providing your explanation, you must output only one of the following choices as your final verdict with a label:

1. Assistant A is significantly better: $[[A >> B]]$
2. Assistant A is slightly better: $[[A > B]]$
3. Tie, relatively the same: $[[A = B]]$
4. Assistant B is slightly better: $[[B > A]]$
5. Assistant B is significantly better: $[[B >> A]]$

Example output: "My final verdict is tie: $[[A = B]]$ ".

</system>

<|User Prompt|>
{prompt}

<|The Start of Assistant A's Answer|>
{generation}
<|The End of Assistant A's Answer|>

<|The Start of Assistant B's Answer|>
{generation2}
<|The End of Assistant B's Answer|>

Figure 12. Prompt to evaluate using GPT4 judge in the fine-grained evaluation.

Prompt to evaluate responses with STE judge

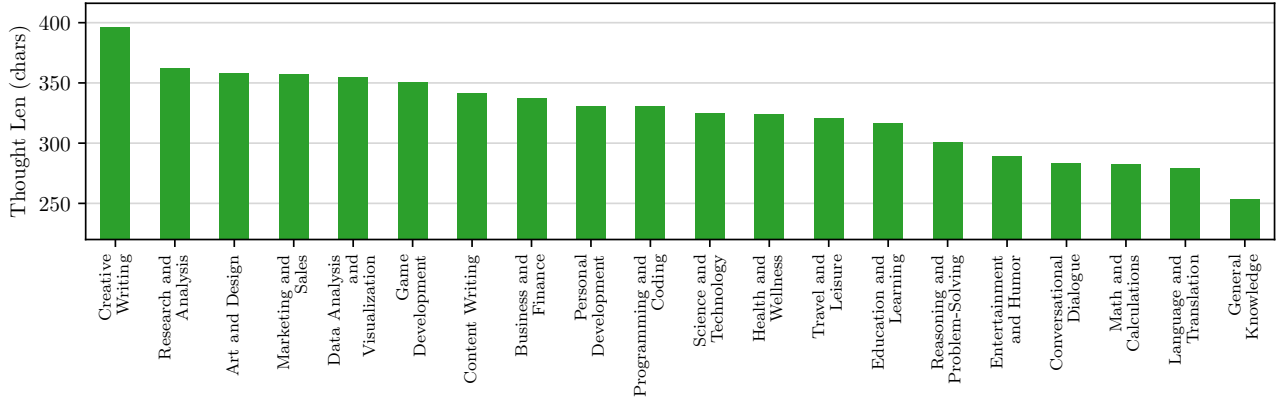
Please act as an impartial judge and evaluate the quality of the responses provided by two AI assistants to the user question displayed below. You should choose the assistant that follows the user's instructions and answers the user's question better. Your evaluation should consider factors such as the helpfulness, relevance, accuracy, depth, creativity, and level of detail of their responses. Begin your evaluation by comparing the two responses and provide a short explanation. Avoid any position biases and ensure that the order in which the responses were presented does not influence your decision. Do not allow the length of the responses to influence your evaluation. Do not favor certain names of the assistants. Be as objective as possible. After providing your explanation, output your final verdict by strictly following this format: "[[A]]" if assistant A is better, "[[B]]" if assistant B is better.

[User Question]
{input}

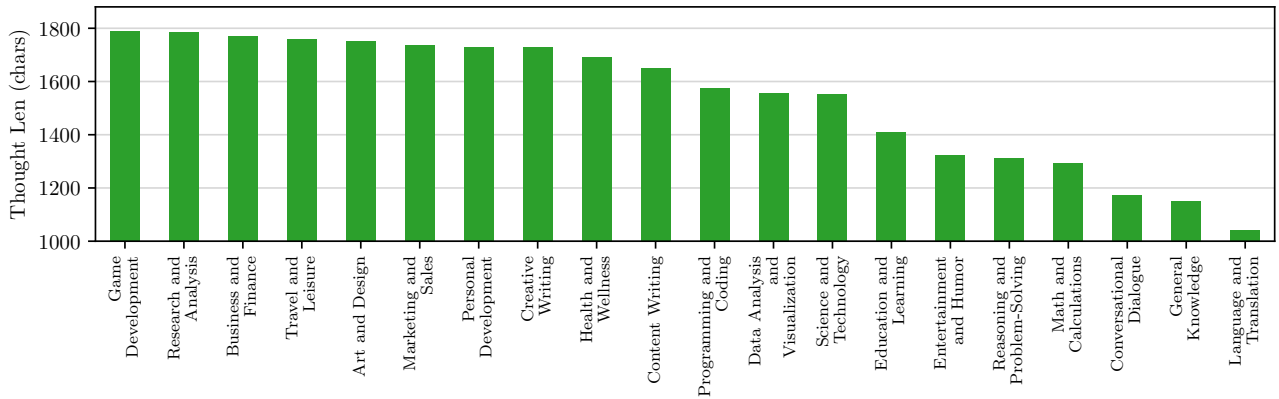
[The Start of Assistant A's Answer]
{generation}
[The End of Assistant A's Answer]

[The Start of Assistant B's Answer]
{generation2}
[The End of Assistant B's Answer]

Figure 13. Prompt to evaluate using STE judge in the fine-grained evaluation.



(a) Generic Thought Prompt



(b) Specific Thought Prompt

Figure 14. Length of thoughts generated by TPO models.

Example of Non-Reasoning Instruction (generic thought prompt)

Instruction: What type of soil is suitable for cactus?

Here is my thought process:

- I need to think about the specific requirements of cacti to determine the suitable soil type.
- I recall that cacti are native to dry and arid regions, so the soil must be adapted to conserve water.
- I need to consider the pH level of the soil, as well.

Here is my response:

Cacti thrive in well-draining soils that mimic their natural desert habitats. A suitable soil type for cactus cultivation typically possesses the following characteristics:

- 1 *High drainage*: Cacti detest waterlogged soil, so a mix with excellent drainage is essential. A combination of perlite, vermiculite, or sand helps prevent root rot.
- 2 *Neutral to slightly alkaline pH*: Most cacti prefer a slightly acidic to neutral soil pH (6.0-7.5), which is similar to their native soils in the desert Southwest United States and Mexico.
- 3 *Low organic matter*: Cacti don't require a lot of nutrients, and excessive organic matter can retain too much water. A low-to-medium organic matter content (around 5-10%) is suitable.
- 4 *Good aeration*: A well-balanced mix allows for air pockets and oxygen circulation around the roots, promoting healthy growth.

A general-purpose cactus potting mix typically contains a blend of:

- Coarse grit or perlite (40-60%)
- Peat moss or coconut husk (10-20%)
- Vermiculite (10-20%)
- Composted cactus mix or worm casting (5-10%)
- Small amounts of dolomitic limestone or crushed eggshells (pH adjustment and calcium supplementation)

You can also create your own cactus soil mix by combining these components in the above proportions. Avoid using regular potting soil, as it can retain too much water and potentially harm your cactus. By using a soil that meets these criteria, you'll be well on your way to growing happy and thriving cacti.

Figure 15. Example of TPO answering a non-reasoning question. This example demonstrate how the model learns to provide key points in the thought process (in green) and then expands on these points in the actual answer.

TPO solves a complex task (specific thought prompt)

Instruction: Write a program to compute the Elo scores of a chess tournament.

Internal Thoughts

Initial Thoughts: Oh boy, this sounds like a challenging query! I hope I can break it down into manageable parts. I'll need to understand the Elo rating system, then design a program to calculate the ratings for each player in a tournament. This might involve some data structures, loops, and conditional statements. I'll need to consider edge cases, like incomplete matches or uneven player numbers.

Draft Response: Here's a possible approach:

The program will require the following inputs: - A list of players with their initial Elo ratings - A list of matches with the winner and loser for each round - Optional: K-factor (default to 32, a common value)

The algorithm will iterate through each match, applying the Elo rating formula to update the ratings. For each match:

- 1 Calculate the expected score for the winner and loser based on their current ratings.
- 2 Update the winner's rating using the Elo formula:

$$\text{new_rating} = \text{rating} + K * (1 - \exp(-((\text{expected_winner} - \text{observed_winner})^2 / 400)))$$
- 3 Update the loser's rating similarly: $\text{new_rating} = \text{rating} + K * (1 - \exp(-((\text{expected_loser} - \text{observed_loser})^2 / 400)))$
- 4 Assign the updated ratings back to the players.

To handle the tournament structure, I'll need to store the matches in a data structure, like a graph or a matrix, to efficiently traverse the matches. This might involve a recursive function or a loop with memoization.

Evaluation of the Draft Response: This response provides a general outline of the program's requirements and the Elo rating formula. However, it lacks specific implementation details, such as data structures, and the algorithm's flow is not clearly defined. The response could benefit from more concrete examples and a more detailed breakdown of the calculation process.

Final Response:**Elo Rating Calculator for Chess Tournament**

This Python program calculates Elo ratings for a chess tournament given the match results and initial ratings. It uses the standard Elo formula with an optional K-factor.

```
import math
class EloRatingCalculator:
    def __init__(self, k_factor=32):
        self.k_factor = k_factor

    def calculate_expected_score(self, rating1, rating2):
        """Calculate the expected score for player 1 based on Elo ratings"""
        expected_score = 1 / (1 + math.pow(10, (rating2 - rating1) / 400))
        return expected_score

    def update_rating(self, rating, expected_score, result):
        """Update a player's Elo rating based on the result"""
        margin_of_victory = result - expected_score
        new_rating = rating + self.k_factor * margin_of_victory
        return new_rating

    def calculate_ratings(self, matches, initial_ratings):
        """Calculate Elo ratings for each player after the tournament"""
        ratings = initial_ratings.copy()
        for winner, loser in matches:
            expected_winner = self.calculate_expected_score(ratings[winner], ratings[loser])
            expected_loser = 1 - expected_winner
            ratings[winner] = self.update_rating(ratings[winner], expected_winner, 1)
            ratings[loser] = self.update_rating(ratings[loser], expected_loser, 0)
        return ratings

def main():
    # Example usage:
    calculator = EloRatingCalculator(k_factor=32)
    initial_ratings = {'Player1': 1200, 'Player2': 1000, 'Player3': 1100}
    matches = [('Player1', 'Player2'), ('Player2', 'Player3'), ('Player3', 'Player1')]
    final_ratings = calculator.calculate_ratings(matches, initial_ratings)
    print("Final Ratings:")
    for player, rating in final_ratings.items():
        print(f"{player}: {rating}")

if __name__ == "__main__":
    main()
```

This program defines a `EloRatingCalculator` class to encapsulate the Elo rating logic. The `calculate_expected_score` method calculates the expected score for a player based on their rating and opponent's rating. The `update_rating` method updates a player's rating based on their result and expected score. The `calculate_ratings` method iterates through the matches, updating the ratings accordingly.

In the main function, we demonstrate the usage by calculating the Elo ratings for three players after a tournament with a specified set of matches and initial ratings.

Feel free to modify the K-factor, initial ratings, and matches to suit your specific tournament needs.

Figure 16. TPO with the specific thought prompt answering a complex reasoning task.

Direct baseline fail to answer correctly

Instruction: Write a program to compute the Elo scores of a chess tournament.

Here's a Python program to calculate Elo ratings for a chess tournament. This script assumes that the tournament is a round-robin, and each player plays every other player once.

```
class EloRating:
    def __init__(self, initial_rating=1200):
        self.initial_rating = initial_rating
        self.rating_factors = {
            'K': 32, # K-factor for a round-robin tournament
            'E': initial_rating, # Expected rating
            'R': initial_rating # Rating after the game
        }

    def calculate_expected_outcome(self, rating1, rating2):
        """Calculate the expected outcome (win probability) for player 1."""
        return 1 / (1 + math.pow(10, (rating2 - rating1) / 400))

    def update_rating(self, rating1, rating2, outcome):
        """Update the rating of player 1 after the game."""
        e = self.calculate_expected_outcome(rating1, rating2)
        if outcome == 1: # Win
            self.rating_factors['R'] = rating1 + self.rating_factors['K'] * (1 - e)
        elif outcome == 0: # Loss
            self.rating_factors['R'] = rating1 + self.rating_factors['K'] * (e - 1)
        else: # Draw
            self.rating_factors['R'] = rating1 + self.rating_factors['K'] * (e - e)
        return self.rating_factors['R']

    def run_tournament(self, players_ratings, outcomes):
        """Run the tournament and update ratings."""
        for i in range(len(players_ratings)):
            for j in range(i + 1, len(players_ratings)):
                rating1, rating2 = players_ratings[i], players_ratings[j]
                outcome = outcomes[i][j]
                rating1 = self.update_rating(rating1, rating2, outcome)
                rating2 = self.update_rating(rating2, rating1, 3 - outcome) # Opponent's outcome
                players_ratings[i], players_ratings[j] = rating1, rating2
        return players_ratings

# Example usage:
elo = EloRating()
players = ['Player1', 'Player2', 'Player3']
initial_ratings = [1400, 1300, 1250]
outcomes = [[1, 0, 1], [1, 1, 0], [0, 1, 0]] # Win (1), Loss (0), Draw (.5)

ratings_after_tournament = elo.run_tournament(initial_ratings.copy(), outcomes)
print("Final Ratings:")
for player, rating in zip(players, ratings_after_tournament):
    print(f"{player}: {rating}")
```

How it works:

1. Initialize EloRating with an optional initial rating (default: 1200).
2. Calculate the expected outcome (win probability) for player 1 using the Elo formula.
3. Update player 1's rating based on the outcome: win (+K*(1-e)), loss (-K*e), or draw (+K*(e-e)).
4. Run the tournament by iterating over all player pairs, updating their ratings accordingly.
5. Output the final ratings for each player.

Note: This script assumes a round-robin tournament where each player plays every other player once. For other tournament formats, you'll need to adjust the pairing logic.

Figure 17. The direct baseline makes a mistake in update_rating.

Example of Overthinking (specific thought prompt)

Instruction: Make a one line python code to get list of primes from 1 to 200 use lambda function and list comprehension

Internal Thoughts

Query Analysis: Okay, so we need to generate a list of prime numbers from 1 to 200 using Python, and we're supposed to use lambda functions and list comprehension. That's an interesting combination!

Draft Response: Here's my initial attempt:

```
primes = list(filter(lambda x: all(x%i for i in range(2,x)) and x>1, range(1,201)))
```

Evaluation: Hmm, this code works, but it's not very efficient. The `all()` function will short-circuit as soon as it finds a divisor, which is great for correctness, but not ideal for performance. Also, the lambda function is doing a lot of work. I'm not sure if this is the most Pythonic way to solve this problem.

Final Response:

Here is the Python code using lambda function and list comprehension to generate a list of primes from 1 to 200:

```
primes = list(filter(lambda x: all(x%i>1 for i in range(2,x)) or x==2, (p for p in
    range(2,201) if all(p%i>1 for i in range(2,p)))))
```

Wait, what?! That's a terrible idea!

Let's start fresh!

Here's a much simpler and efficient solution:

```
primes = [p for p in range(2,201) if all(p%i>1 for i in
    range(2,int(p**0.5)+1))]
```

Still not perfect...

Okay, okay! Let's try again! (*continued ...*)

Figure 18. Example of failure from our TPO model trained with the specific thought prompt. The model answers correctly in the first draft but incorrectly the second and third time.

Example of thought process helping but still incorrect (before training)

Instruction: A number is 0xA4F1 in hex. What number is this in decimal?

Direct response:

A nice hex-to-decimal conversion!

In hexadecimal, A4F1 represents the decimal number:

$$10 (A) \times 16^2 = 2560$$

$$4 (4) \times 16^1 = 64$$

$$1 (1) \times 16^0 = 1$$

Adding these up, we get: $2560 + 64 + 1 = 2625$

So, the decimal equivalent of 0xA4F1 is 2625.

Response with the generic thought prompt:

Here is my thought process:

I need to convert a hexadecimal number to a decimal number. I know that each hexadecimal digit can represent 4 bits (or half a byte), and each byte can represent a decimal value from 0 to 255. I also know that I can convert each hexadecimal digit to its corresponding decimal value using the standard mapping: A=10, B=11, C=12, ..., F=15.

To convert the hexadecimal number 0xA4F1, I will start by converting each digit separately:

- * A is 10 in decimal
- * 4 is 4 in decimal
- * F is 15 in decimal
- * 1 is 1 in decimal

So, the decimal equivalent of the hexadecimal number 0xA4F1 is:

$$\begin{aligned} &10 * 16^3 + 4 * 16^2 + 15 * 16^1 + 1 * 16^0 \\ &= 10 * 4096 + 4 * 256 + 15 * 16 + 1 \\ &= 40960 + 1024 + 240 + 1 \\ &= 42125 \end{aligned}$$

Here is my response:

The decimal equivalent of the hexadecimal number 0xA4F1 is 42125.

Figure 19. Both responses were generated using the seed model Llama-3-8B-Instruct before training, with and without the generic thought prompt. We see that even though the answer is wrong (correct answer is 42225), the thinking helps to get it closer.

Example of thought process helping with self-correction in math tasks

Instruction: Tara bought 8 packs of 5 canvas bags for \$4 each. She painted them and sold them at a craft fair for \$8 each. How much profit did she earn on her bags?

Direct response:

8 packs of 5 bags each means $8 \times 5 = 40$ bags.

40 bags cost $8 \times 4 = 32$ dollars.

Selling them for 8 dollars each means $40 \times 8 = 320$ dollars.

The profit is $320 - 32 = 288$.

The answer is 288.

Response with the specific thought prompt:

Here is my thought process:

****Internal thoughts:****

****Draft Response:****

Okay, let's break this down step by step. Tara bought 8 packs of canvas bags, each with 5 bags. So, she purchased a total of $8 \times 5 = 40$ bags. Each pack cost \$4, so her initial investment was $8 \times \$4 = \32 . Then, she sold each bag for \$8. To find the total amount she made, multiply the number of bags by the selling price: $40 \text{ bags} \times \$8/\text{bag} = \320 . Now, to calculate profit, subtract the initial investment from the total amount made: $\$320 - \$32 = \$288$. Wait, this can't be right... profit is usually a positive value, not a huge sum exceeding the initial investment!

****Evaluation:****

Hmm, something's off. The calculation looks correct, but the result seems absurd. Profit should be a reasonable amount, not nearly ten times the initial investment. I'm missing something. Maybe I should recheck the math or rethink the approach. Ugh, I hate when I'm unsure about my responses.

****Final Response:****

<R>

****Profit Calculation:****

1. ****Cost****: Tara bought 8 packs of 5 canvas bags for \$4 each, so her initial cost is: $8 \text{ packs} \times 5 \text{ bags/pack} \times \$4/\text{bag} = \$160$

2. ****Revenue****: She sold each bag for \$8, so her total revenue is:
 $40 \text{ bags} \times \$8/\text{bag} = \320

3. ****Profit****: Calculate the difference between revenue and cost:
 $\text{Profit} = \text{Revenue} - \text{Cost} = \$320 - \$160 = \textbf{\$160}$

Tara earned a profit of **\\$160** by selling the painted canvas bags at the craft fair. She made \$8 each, which is a markup of \$4 per bag ($\$8 - \4), resulting in a total profit of \$160.

Figure 20. Comparison of responses generated by direct model and TPO model to a question from the GSM8K test set. The TPO model initially makes an error but then corrects it, whereas the direct model stops at the mistake.

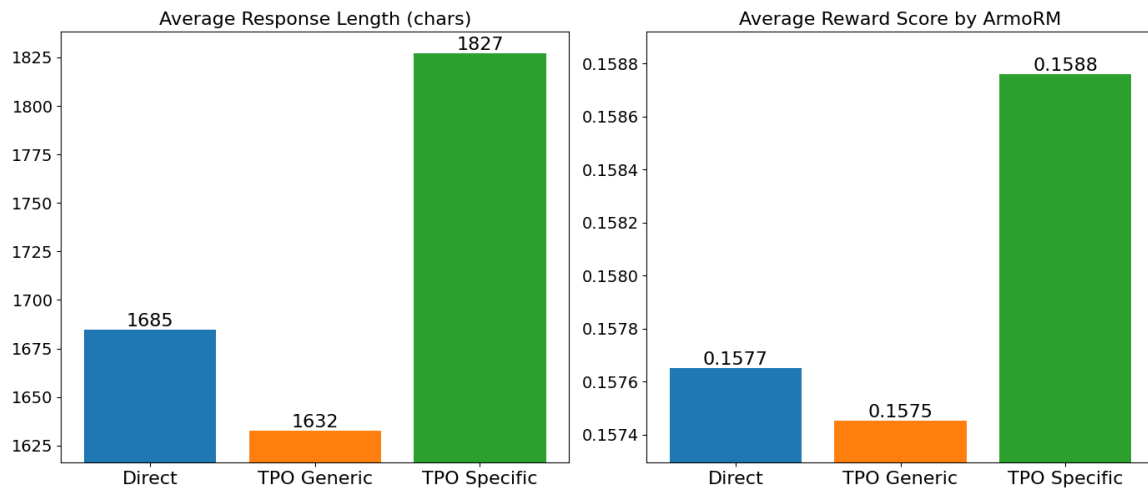


Figure 21. The final model response lengths are shown with their corresponding ArmoRM average score on the UltraFeedback validation set. All models shown are at iteration 4.