Extended Abstract

Motivation Recent advances in models like DeepSeek-R1 and Qwen demonstrate that curated data and targeted supervised fine-tuning, combined with reinforcement learning, can achieve state-of-the-art AI performance without relying on resource-intensive training with large-scale GPU clusters. To explore these advancements, we investigated the impact of data quality, SFT, and RL on model performance and evaluated whether an optimal allocation exists for efficiently training AI models. The findings of this research hold significant potential for the industry, where cost and energy efficiency are critical for scaling AI solutions. Lower energy requirements align with growing industry demands for sustainable practices, reducing the carbon footprint of AI development and deployment.

Method To test our hypotheses on efficient AI training, we used the Qwen2.5-0.5B model to solve the Countdown task, which challenges models to create accurate equations from given numbers. The Countdown task was chosen because it requires both mathematical reasoning and creative problem-solving, making it a challenging benchmark for evaluating data quality and training efficiency. We conducted three experiments. First, we tested the impact of data quality on supervised fine-tuning performance using the WarmStart dataset (1,000 problems with known quality issues), a corrected dataset (1,000 problems with errors fixed), and a synthetically enhanced dataset (1,000 problems with multiple valid solutions per problem). Second, we analyzed the uplift provided by reinforcement learning (RL) after SFT, starting from initial success rates of 12% (low) and 44% (high). Finally, we assessed how varying the proportion of SFT versus RL training steps, within a fixed compute budget of 10,000 gradient update steps, impacts model performance on the Countdown task, aiming to determine if an optimal allocation exists. Performance was measured by the success rate (percentage of problems solved correctly) and a reward score based on response presence and correctness. These experiments aim to identify training strategies that reduce compute demands, aligning with industry needs for cost-effective and sustainable AI development.

Implementation We implemented our experiments using the Qwen2.5-0.5B model with Hugging Face Transformers and PyTorch, writing our own SFT and RL code in PyTorch for the Countdown task. We used Hydra for configuration management and Weights & Biases (wandb) for tracking metrics and visualizing results. Training was conducted using Nvidia L4 48GB. We trained SFT with a range of parameters (learning rate, decay of 0.005, batch size) to select the best performing. [RL used Proximal Policy Optimization with a two-stage reward (response presence and correctness) for 150 steps]. Custom libraries were developed to evaluate LLMs for RL synthetic data.

Results For our first experiment, we started by conducting SFT training. The performance of the SFT model on the poor quality dataset was low at 6%. With the corrected dataset, the performance was 11%, suggesting that the model did not learn. The synthetically enhanced dataset with multiple solutions reached 60% after 20k steps. For our second experiment, we applied RL to SFT-trained models with initial success rates of respectively 12% and 44%. We observed that the uplift of performance was similar at 80% with only 150 steps. For our last experiment, we observe that there was no benefit in spending more than 20k steps finetuning on a quality datasets, but that on a poor quality dataset, thus difficult to learn, it was optimal to conduct SFT up to 70k or 80k steps, and the residual 20k or 30k be allocated to SFT

Discussion Through the experiments, we learned the importance of quality data to speed up the models learning process. We were surprised that cleaning up a dataset to only have correct answers had limited impact on the performance of the model. On the contrary, generating multiple correct answers foe each mathematical question allowed the model to learn and created a huge bump in performance. With 20k steps for SFT training on a dataset with multiple solutions per question, it was possible to obtain a performance of 60%. We observed that the performance of applying RL to a SFT-trained model was a similar uplift in percentage irrespective of the initial SFT performance rate.

Conclusion Though exploratory and only based on a small model Qwen2.5-0.5b, our research appears to show that there is an optimal allocation between data curation, SFT and RL. The main limitation is that the work was performed on just one specific task: countdown. It would be interesting to understand how the optimum would change when tackling different kinds of tasks. We could consider, for example, Meta-Learning with multiple tasks.

Train smarter LLMs: balancing Data Quality, Fine-Tuning and Reinforcement Learning

Gabriel Mesquida Masana
Stanford University
gabmm@stanford.edu

François Chesnay
Stanford University
fchesnay@stanford.edu

Abstract

Innovative large language models have demonstrated that high performance can be achieved without GPU-intensive approaches, reducing training and inference costs while mitigating energy consumption and societal impacts. This study leverages the Countdown task, previously used by DeepSeek R1, to successfully apply reasoning through reinforcement learning (without human feedback. We investigate and quantify the relative contributions of data curation, supervised fine-tuning, and RL to model performance, and propose an optimal framework for efficient LLM training.

1 Introduction

1.1 Challenges in Large Language Models

Large language models have shown remarkable proficiency in natural language processing tasks (Hendrycks et al., 2020; Clark et al., 2018). However, they often struggle with complex reasoning tasks (Wei et al., 2022a) and accessing up-to-date external information (Jin et al., 2024). These limitations necessitate advancements in reasoning capabilities (Huang and Chang, 2022) and effective integration with external knowledge sources, such as search engines (Schick et al., 2023), to enhance their practical utility.

1.2 Advancements in Reasoning with Reinforcement Learning

Reinforcement Learning (RL) has emerged as a powerful approach to improve LLM reasoning (Yang et al., 2025; Guo et al., 2025; Hou et al., 2025; Xie et al., 2025b). For example, Yang et al. (2025) trained a 3B-parameter model on the Countdown Game, a numerical reasoning task, using pure RL, achieving superior performance on four out of five benchmarks and demonstrating strong generalization. Similarly, models like OpenAI-o1 (Jaech et al., 2024) and DeepSeek-R1 (Guo et al., 2025) employ RL techniques such as Proximal Policy Optimization (PPO) (Schulman et al., 2017) to enhance logical inference and problem-solving, enabling capabilities like self-verification (Weng et al., 2022) and self-correction (Kumar et al., 2024). Recent work by Jin et al. (2025) further explores training LLMs to reason and leverage search engines using RL, highlighting the potential of RL to address information retrieval challenges.

1.3 Efficient Post-Training Strategies

Post-training plays a critical role in aligning LLMs with robust reasoning, ethical standards, and user needs while minimizing computational costs. Unlike resource-intensive pre-training, post-training leverages efficient methods such as Reinforcement Learning from Human Feedback (RLHF) (Christiano et al., 2017; Ziegler et al., 2019), instruction tuning (Wei et al., 2021), and Direct Preference Optimization (DPO) (Rafailov et al., 2023). These techniques enhance generalization,

reduce biases, and lower computational demands (Hu et al., 2021). Supervised Fine-Tuning further adapts pre-trained LLMs to specific tasks using labeled data (Ouyang et al., 2022), though collecting such datasets can be resource-intensive, underscoring the need for optimized post-training strategies.

1.4 Evolution of Reasoning Techniques

Reasoning is fundamental to LLM versatility across domains. Early methods relied on prompt engineering, such as Chain-of-Thought (CoT) reasoning, which encourages models to generate intermediate steps for tasks like mathematical problem-solving (Wei et al., 2022b). More recent advancements, such as OpenAI's O-series models, utilize inference-time scaling to extend CoT into longer reasoning chains, significantly improving performance (OpenAI, 2024). Complementary approaches, including Monte Carlo Tree Search (MCTS) (Silver et al., 2016; Feng et al., 2024) and process-based reward models (Lightman et al., 2023; Uesato et al., 2022), further enhance reasoning capabilities.

1.5 Objectives and Motivation

This study explores the interplay of data quality, supervised fine-tuning, and reinforcement learning (RL) in enhancing large language model (LLM) performance, using the Qwen2-5.0-5B model on the Countdown task—a numerical reasoning challenge requiring creative problem-solving. Our experiments evaluate how dataset quality, the balance of SFT and RL training, and computational efficiency impact reasoning capabilities, aiming to identify training strategies that maximize performance while supporting sustainable AI practices. Recent advancements, such as DeepSeek-R1 and Qwen, demonstrate that high-quality data, targeted SFT, and RL can achieve state-of-the-art results without relying on resource-intensive GPU clusters, offering significant potential for cost- and energy-efficient AI scaling in industry.

Inspired by DeepSeek-R1 Zero Guo et al. (2025), a breakthrough that used RL to significantly reduce LLM training costs, we investigate optimal strategies for balancing data curation (high-quality datasets) and resource allocation. Our key contribution is developing efficient training pipelines that minimize costs and accelerate learning, enabling high-performing LLMs for reasoning tasks with reduced environmental impact.

2 Related Work

Synthetic data augmentation has been widely explored to enhance LLM performance, particularly for reasoning tasks. Guo et al. (2025) employed rejection sampling to generate high-quality datasets for math reasoning with DeepSeek-R1, combining synthetic data with SFT to improve performance, though their approach was tailored to specific tasks and models. Yang et al. (2025) applied pure RL to the Countdown task, using synthetic data to train a 3B-parameter model, achieving strong generalization across benchmarks. Similarly, Xie et al. (2025a) used rule-based RL with synthetic logic puzzles to develop reasoning skills that generalize to challenging math benchmarks like AIME and AMC. These works highlight the potential of synthetic data in reasoning tasks but often do not address computational efficiency or task-specific adaptation for the Countdown task, which our study aims to explore.

RL has been increasingly utilized to optimize LLM training efficiency. Lee et al. (2024) introduced RL from AI Feedback (RLAIF), scaling preference learning with AI-generated feedback, which informs our approach to RL optimization. Jin et al. (2025) explored training LLMs to reason and leverage search engines using RL, focusing on efficient information retrieval. Lian (2025) further demonstrated the effectiveness of RL in reasoning tasks, reinforcing its applicability to numerical challenges like the Countdown task. Unlike prior work, our study emphasizes balancing SFT and RL within a fixed compute budget to achieve efficient training, addressing both performance and sustainability.

Over-reliance on synthetic data can lead to model collapse, where performance degrades over time. Shumailov et al. (2024) observed this phenomenon when fine-tuning GPT-2 and OPT-125m with synthetic data, leading to task degradation. Similarly, Kazdan et al. (2025) noted potential degradation in Gemma2 models trained on synthetic data. These risks are critical considerations in our work, as

we aim to optimize synthetic data use for the Countdown task while mitigating potential downsides through careful dataset design and training strategies.

3 Method

3.1 Overview and Model Selection

To investigate efficient AI training strategies, we use the Qwen2-5.0-5B model to address the Countdown task, a numerical reasoning challenge where the model must form an equation equaling a target value using a given set of numbers exactly once with basic arithmetic operations ($+$, $-$, $\times$, $/$). This task was chosen because it requires both mathematical reasoning and creative problem-solving, making it an ideal benchmark for evaluating the impact of data quality and training efficiency on LLM performance. The Qwen2-5.0-5B model, a relatively small-scale LLM, allows us to explore these effects while keeping computational costs manageable, aligning with our focus on sustainable AI development.

3.2 Experiment 1: Impact of Data Quality on SFT

The first experiment assesses how data quality affects Supervised Fine-Tuning (SFT) performance. We use three datasets, each containing 1,000 training examples: (i) the original WarmStart dataset with a 60% error rate, (ii) a corrected WarmStart dataset with one valid solution per problem, and (iii) a synthetically enhanced WarmStart dataset with multiple valid solutions per problem (3,250 rows total). The synthetic dataset was generated to provide diverse solution paths, addressing the constraint that all numbers must be used exactly once. We fine-tune the model on each dataset and measure the success rate (percentage of problems solved correctly) and a reward score based on response presence and correctness. Additionally, we analyze the number of training steps required to achieve a target performance level (e.g., success rate above 30%), providing insights into the efficiency of learning from varying data quality.

3.3 Experiment 2: RL Uplift After SFT

The second experiment evaluates the performance uplift provided by Reinforcement Learning (RL) following SFT. We start with two SFT-trained models: one with a low initial success rate of 12% (trained on the original WarmStart dataset) and another with a higher success rate of 44% (trained on the corrected dataset). RL is applied using Proximal Policy Optimization (PPO) with a two-stage reward function (response presence and correctness) for 150 steps. Performance is measured by the increase in success rate post-RL, aiming to understand how RL can enhance reasoning capabilities

original low-quality sample

```
A conversation between User and Assistant.
The user asks a question, and the
Assistant solves it.  The assistant first
thinks about the reasoning process in the
mind and then provides the user with the
answer.  User:  Using the numbers [75, 19,
39, 75], create an equation that equals 57.
You can use basic arithmetic operations
(+, -, *, /) and each number can only be
used once.  Show your work in <think>
</think> tags.  And return the final
answer in <answer> </answer> tags, for
example <answer> (1 + 2) / 3 </answer>.
Assistant:  Let me solve this step by
step.

Let me try to find a way to reach 57 using
these numbers.
First, let me look for factors or nearby
numbers to 57.
57 = 3 x 19, so having 19 is helpful.
Let me try working with 75 first:
75 - 19 = 56, which is close to 57!
So we need to add 1 somehow using 39 and
75.
Looking at 39 and 75:
75 / 75 = 1
Then we can use this 1 to add to 56.
Let's verify the full sequence:
1.  First get 56:  75 - 19 = 56
2.  Then get 1:  75 / 75 = 1
3.  Finally add 1 to 56:  56 + 1 = 57
Working backwards to ensure we use all
numbers and reach 57:
((75 - 19) + (75 / 75)) - 39 = 57
</think>
<answer> ((75 - 19) + (75 / 75)) - 39
</answer>
```

Figure 1: Erroneous sample from original data set. Analysis show that the original WarmStart dataset has $\approx 60\%$ error rate.

regardless of initial SFT performance, and the computational cost of this uplift in terms of gradient updates.

3.4 Experiment 3: Balancing SFT and RL Training Steps

The third experiment examines the optimal allocation of SFT and RL training steps within a fixed compute budget of 10,000 gradient updates. Using the synthetically enhanced dataset, we vary the proportion of steps allocated to SFT versus RL (e.g., 70% SFT and 30% RL) and measure the resulting success rate and reward score on the Countdown task. We also track computational efficiency by monitoring energy consumption during training, ensuring alignment with sustainable practices. This experiment aims to identify a training strategy that maximizes reasoning performance while minimizing computational resources, addressing the industry's need for cost-effective AI solutions.

4 Experimental Setup

For the Countdown task, we fine-tuned the Qwen2.5 0.5B model on the math problems *Countdown-Tasks-3to4* train set (1,000 observations) with a learning rate of 2.0×10^{-4} and learning rate decay of 0.005, and the training loss was decreasing as expected as shown in figure **??**. The evaluation was based on *Countdown-Tasks-3to4* test set (200 observations). Reward was computed based on a two stage reward: response presence and correctness following TinyZero.

The SFT training was initially conducted on Warmstart, a low-quality dataset with a 60% error rate. An example of incorrect answer in the initial data is presented below. The solution uses the correct list of number, but evaluates to 18 and **not to the target solution of 57**:

The low quality of the original dataset hinders any model's ability to learn a complex task from inaccurate data. To resolve this issue, we refined the data and developed two new datasets, as outlined below.

4.1 Creating a corrected dataset

We created a new dataset based on Warmstart correcting the thought processes and mathematical solutions. We used Gemini API to generate solutions and check the validity of the solutions generated. The code used is included in appendix. An example of solution generated by Gemini has also be included in appendix. The corrected data set is available at `clean_Asap7772_cog_behav_all_strategies`. It contains only valid solutions (train set 1k rows and test set 200 rows).

4.2 Creating a synthetically enhanced dataset

In addition to the corrected dataset, we also created a synthetically enhanced dataset with multiple answers. The rationale of providing multiple answers was to avoid overfitting by giving more chances

synthetic_data_warmstart_3.25k sample

```
A conversation between User and Assistant. The user asks a question, and the Assistant solves it. The assistant first thinks about the reasoning process in the mind and then provides the user with the answer. User: Using the numbers [81, 57, 98, 7], create an equation that equals 62. You can use basic arithmetic operations (+, -, *, /) and each number can only be used once. Show your work in <think> </think> tags. And return the final answer in <answer> </answer> tags, for example <answer> (1 + 2) / 3 </answer>. Assistant: Let me solve this step by step.
```

```
<think>
To get 62 using 81, 57, 98, and 7, I need to combine these numbers with arithmetic operations. I'll start by trying to use multiplication and subtraction, as those often lead to larger intermediate values. I need to find a combination that isolates 62 as a result. Subtracting 81 from 98 gives 17. Multiplying 7 by 17 gives 119. Subtracting 57 from 119 gives 62. This works!
</think>
<answer> (7 * (98 - 81)) - 57 </answer>
```

Figure 2: AI-generated data augmentation sample from the synthetically enhanced data set `synthetic_data_warmstart_3.25k`. The full dataset can be accessed at https://huggingface.co/datasets/fchesnay/synthetic_data_warmstart_3.25k.

for the model to really learn how to solve mathematical problems rather than simply memorize one answer per question.

We created code calling Grok API in order to generate multiple valid solutions. For each mathematical question, called Grok's api 5 times, and we ended up with on average 3.25 valid solutions per question. The synthetically augmented dataset is available at `synthetic_data_warmstart_3.25k`. It contains multiple correct answers (train set 3.25k rows and test set 200 rows).

5 Results

5.1 Experiment 1: Impact of Data Quality on SFT

The CountDown task presents a distinctive and intricate challenge for large language models (LLMs), as it demands the simultaneous computation of loss for both the reasoning process and the final answer, a task complicated by the diverse array of potential solutions and the need for coherent, step-by-step reasoning. This dual requirement tests the model's ability to not only arrive at a correct outcome but also to articulate a logical and verifiable thought process, which is critical for tasks requiring deep reasoning.

We initially conducted Supervised Fine-Tuning on Warmstart, a low-quality dataset with a 60% error rate, resulting in poor performance (5%, blue line in the graph below).

We hypothesized that a cleaner dataset would significantly improve outcomes, thus we corrected the Warmstart dataset to have correct thought processes and mathematical solutions, but the performance remained underwhelming (8%, green line in the graph below).

The CountDown task poses a unique challenge for large language models, as it requires computing loss for both the reasoning process and the final answer-a complex task due to the variety of possible solutions, thus making learning difficult and prone to overfitting. We thought that a possibility to increase learning was to synthetically generate multiple answers for each questions, thus giving more chances for the model to really learn how to solve mathematical problems rather than simply memorize one answer per question.

With our newly created enhanced dataset representing on average 3.25 answers per questions, we conducted Supervised Fine-Tuning again, and achieved a huge performance gain (lines in red and orange). With a learning rate of 2.4×10^{-4}, the average performance was 44%, with a 60% pick in performance at 2,000 steps. The performance with a learning of 2.5×10^{-4} was close at 40%. A learning rate of 2.5×10^{-4} was too low and only achieved a performance of 24%.

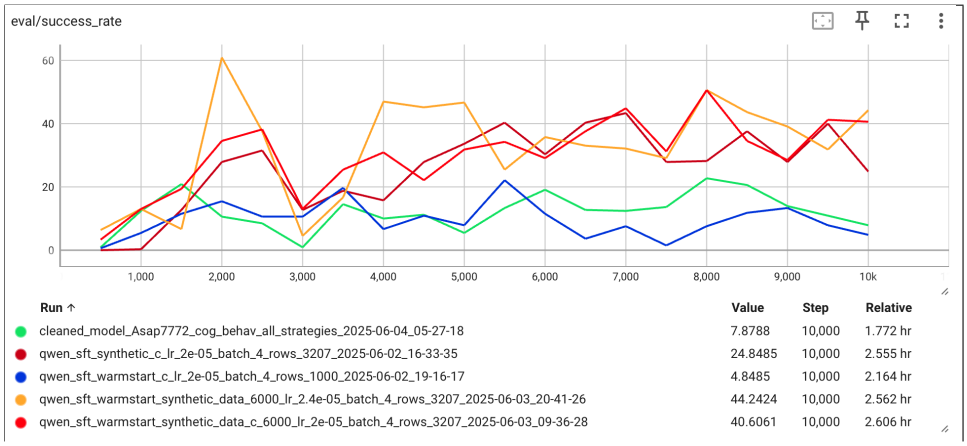

Figure 3: Countdown SFT success rate with synthetic, corrected and original datasets. Note how the corrected dataset shows little improvement from the baseline dataset whilst AI data augmentation provides a significant increase in success rate.

The SFT training was initially conducted on Warmstart, a low-quality dataset with a 60% error rate. The performance was poor (5%, blue line in the graph above).

We hypothesized that a cleaner dataset would significantly improve outcomes, thus we corrected the Warmstart dataset to have correct thought processes and mathematical solutions, but the performance remained underwhelming (8%, green line in the graph below).

The CountDown task poses a unique challenge for large language models, as it requires computing loss for both the reasoning process and the final answer-a complex task due to the variety of possible solutions, thus making learning difficult and prone to overfitting. We thought that a possibility to increase learning was to synthetically generate multiple answers for each questions, thus giving more chances for the model to really learn how to solve mathematical problems rather than simply memorize one answer per question.

With our newly created enhanced dataset representing on average 3.25 answers per questions, we conducted Supervised Fine-Tuning again, and achieved a huge performance gain (lines in red and orange, respectively 40% and 44%).

With a learning rate of 2.4×10^{-4}, the average performance was 44%, with a 60% pick in performance at 2,000 steps. The performance with a learning of 2.5×10^{-4} was close at 40%. A learning rate of 2.5×10^{-4} was too low and only achieved a performance of 24%.

We explored a range of parameters to optimize performance on our enhanced synthetically generated and noted that a learning rate of 2.4×10^{-5} (orange line) yields the best performance, as shown in the graph below. A learning rate of 3.0×10^{-5} (red line) was too high, was a learning rate of 1.0×10^{-5} (dark blue line) was too low.

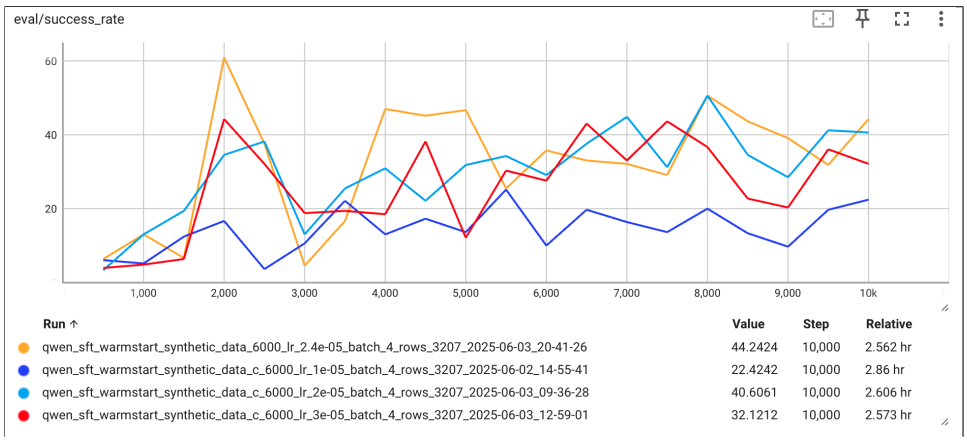

Figure 4: Countdown SFT success rate with synthetic enhanced dataset for various learning rates. Note how the lowest learning rate also resulted in the lowest success rate.

We conclude that high-quality data—defined as datasets containing multiple distinct answers for each mathematical question to prevent overfitting and enable genuine learning rather than memorization—is critical for effective model training. With such data, minimal supervised fine-tuning training, using as few as 2,000 examples in our case, proves optimal. This approach allows resources to be prioritized toward reinforcement learning, resulting in faster, more cost-efficient training and significantly reduced power demands on data centers. This is illustrated by the graph below showing that the majority of the loss has been captured at 20k steps.

5.2 Experiment 2: RL Uplift After SFT

We applied RL to SFT-trained models with initial success rates of 12% and 44% respectively. We observed an increase in performance of 80% with only 150 steps. Further improvements may be possible with more training.

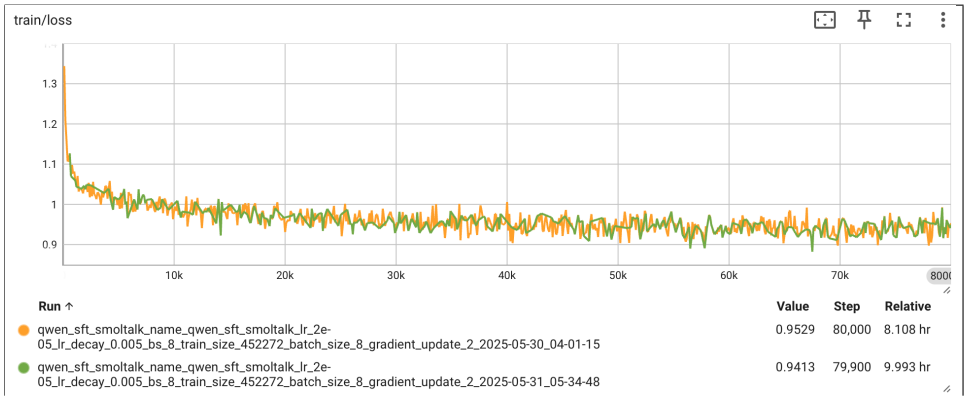

Figure 5: Countdown SFT training loss on synthetic enhanced dataset for a couple of different runs.

For illustration purpose, we have included below a graph showing the performance on the CountDown task using various values of k (k=8 green lines and k=16 in red lines), learning rates and batch sizes. The experiments suggest that the optimal value is for k=8, which also represents a better outcome than k=16 and which is also much faster.

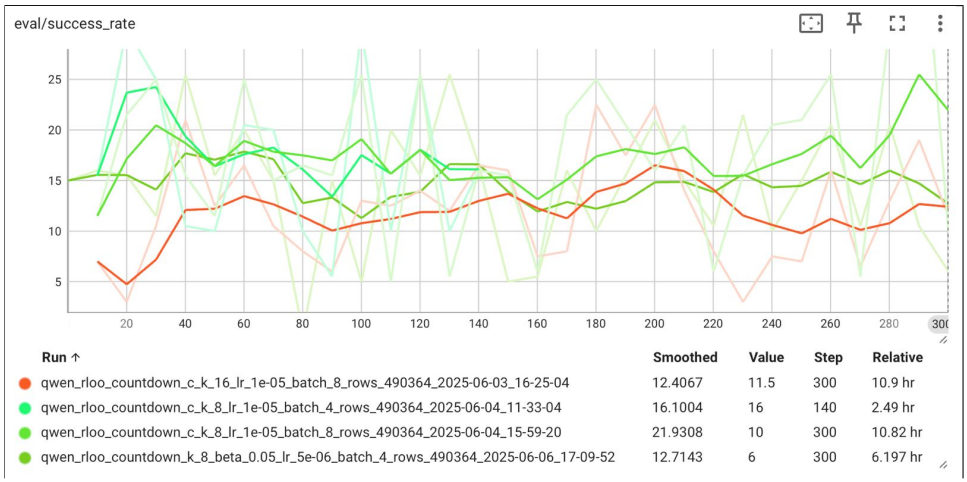

Figure 6: Qwen RLOO success rate with various learning rates and batch sizes. Note that these training runs do not include the KL loss, which is included in equivalent Figure 14.

5.3 Optimal allocation

We experimented with determining the optimal number of steps for Supervised Fine-Tuning, considering the quality of the underlying dataset.

We observed that, to enable the model to learn effectively and avoid overfitting, multiple answers per question are necessary. Additionally, we noted that the datasets used to train our models did not align with the holdout data, which contained larger target values and included more numbers than the maximum of four encountered previously. This mismatch highlights the need for training data to closely match the test dataset, explaining the poor final performance on the holdout dataset.

Having completed the SFT, we considered that a Reinforcement Learning with Outcome Optimization forward step requires k generations and one forward pass, compared to one forward pass for SFT.

Additionally, the calculation of the KL divergence term could be applied to both SFT and RLOO if desired, but this alone did not enable a meaningful comparison.

To draw meaningful conclusions, it would be necessary to estimate the costs of data curation, SFT, and reinforcement learning (RL). This would allow us to determine the optimal allocation of training resources.

6 Discussion

Through the experiments, we learned the importance of quality data to speed up the models learning process.

We were surprised that cleaning up a dataset to only have correct answers had limited impact on the performance of the model.

On the contrary, generating multiple correct answers foe each mathematical question allowed the model to learn and created a huge bump in performance.

With only 20k steps for SFT training on a dataset with multiple solutions per question, it was possible to obtain a performance of 60%. We observed that the performance of applying RL to a SFT-trained model was a similar uplift in percentage irrespective of the initial SFT performance rate.

We contribute two new datasets should they be useful for future CS224R cohorts: `clean_Asap7772_cog_behav_all_strategies`, which is AI-corrected, and the second `synthetic_data_warmstart_3.25k` which is AI-augmented and contains multiple correct answers.

7 Conclusion

Though exploratory and only based on a small model Qwen2.5-0.5b, our research appears to show that there is an optimal allocation between data curation, data augmentation, SFT and RL. Further work should be required to quantify this balance.

The main limitation is that the work was performed on just one specific task: countdown. It would be interesting to understand how the optimum would change when tackling different kinds of tasks. We could consider, for example, Meta-Learning with multiple tasks.

Part of this limitation is due to our interest to explore RLOO, DPO and, in an additional experiment, DPO-Positive, to maximize our learning of the very wide scope initially defined in the default project. This, in a two-member team, detracted potential time to quantify the above conclusions.

8 Team Contributions

Our initial intent was to cover all the experiments together but, due to the wide scope, each of us focused on different parts and then shared the learning:

- **Gabriel Mesquida Masana**: coded a revised version of the SFT. Coded all the RL models. Configured servers and vLLM, trained models, ran experiments. Reviewed, commented on and amended the first draft of the report.
- **François Chesnay**: created libraries for countdown and for instruct, debugged bugs in the countdown implementation code provided to us. Wrote the first draft of the project. Configured servers and vLLM, trained models, ran experiments.

Changes from Proposal The original proposal outlined three methodological questions on supervised fine-tuning (SFT) data quality, the balance of SFT versus reinforcement learning (RL) within a 10,000-step compute budget, and paired preference dataset construction for Direct Preference Optimization (DPO) for the Countdown task. The revised proposal re-frames these into a cohesive experimental framework using the Qwen2.5-0.5B model, testing three datasets (original, corrected, and synthetic WarmStart, each 1,000 problems), RL uplift from 12% and 44% initial success rates, and SFT versus RL allocation, while omitting the DPO question. It defines success rate and reward score as metrics and aligns with industry goals for sustainable AI, enhancing practical relevance and rigor.

References

Paul F. Christiano, Jan Leike, Tom B. Brown, Miljan Martic, Shane Legg, and Dario Amodei. 2017. Deep Reinforcement Learning from Human Preferences. *arXiv preprint arXiv:1706.03741* (2017).

Peter Clark, Isaac Cowhey, Oren Etzioni, Tushar Khot, Ashish Sabharwal, Carissa Schoenick, and Oyvind Tafjord. 2018. Think you have Solved Question Answering? Try ARC, the AI2 Reasoning Challenge. *arXiv preprint arXiv:1803.05457* (2018).

Xidong Feng, Ziyu Ying, Boya Zhang, Xingwei Chen, and Zeyuan Tang. 2024. AlphaMath: Solving Math Problems via Direct Policy Optimization. *arXiv preprint arXiv:2408.11159* (2024).

Yuxuan Guo, Qiming Wang, Jinxin Liu, Ziyang Liu, Peng Hao, Yiming Yang, Donglai Wei, Feifei Huang, and Fangrong Chen. 2025. DeepSeek-R1: Raising the Bar for Open-Source Reasoning Models. *arXiv preprint arXiv:2502.04761* (2025).

Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. 2020. Measuring Massive Multitask Language Understanding. *arXiv preprint arXiv:2009.03300* (2020).

Yifei Hou, Yucheng Wang, Xueqing Zhang, Wayne Xin Huang, Xu Wang, Jiaxin Li, Zhihao Zhang, Hao Peng, Yujie Tang, Zhaofeng Wang, Jianbin Su, Jingang Lu, Yongbin Yang, and Jiancan Lu. 2025. From Words to Actions: Unveiling the Theoretical Underpinnings of LLM-Driven Autonomous Systems. *arXiv preprint arXiv:2503.09843* (2025).

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, and Weizhu Chen. 2021. LoRA: Low-Rank Adaptation of Large Language Models. *arXiv preprint arXiv:2106.09685* (2021).

Jie Huang and Kevin Chen-Chuan Chang. 2022. Towards Reasoning in Large Language Models: A Survey. *arXiv preprint arXiv:2212.10403* (2022).

Aaron Jaech, Daniel Furman, and OpenAI. 2024. o1: A Step Forward in Reasoning-Focused Language Models. *OpenAI Technical Report* (2024).

Bowen Jin, Hansi Zeng, Zhenrui Yue, Jinsung Yoon, Sercan Arik, Dong Wang, Hamed Zamani, and Jiawei Han. 2025. Search-R1: Training LLMs to Reason and Leverage Search Engines with Reinforcement Learning. *arXiv preprint arXiv:2503.09516* (2025).

Honglin Jin, Krishna Krishna, John Wieting, and Jeffrey Pennington. 2024. The Trade-offs of Model Size in Large Recommendation Models: A Quality and Societal Impacts Perspective. *arXiv preprint arXiv:2407.03366* (2024).

Joshua Kazdan, Rylan Schaeffer, Apratim Dey, Matthias Gerstgrasser, Rafael Rafailov, David L. Donoho, and Sanmi Koyejo. 2025. Collapse or Thrive? Perils and Promises of Synthetic Data in a Self-Generating World. *arXiv preprint arXiv:2410.16713* (2025).

Aviral Kumar, Vincent Zhou, Abhishek Sharma, and Sergey Levine. 2024. Language Models as Compilers: Simulating Pseudocode Execution Improves Algorithmic Reasoning. *arXiv preprint arXiv:2407.04622* (2024).

Harrison Lee, Samrat Phatale, Hassan Mansoor, Thomas Mesnard, Johan Ferret, Kellie Lu, Colton Bishop, Ethan Hall, Victor Carbune, Abhinav Rastogi, and Sushant Prakash. 2024. RLAIF vs. RLHF: Scaling Reinforcement Learning from Human Feedback with AI Feedback. *arXiv preprint arXiv:2309.00267* (2024).

Yongsheng Lian. 2025. Reinforcement Learning is all You Need. *arXiv preprint arXiv:2503.09512* (2025).

Hunter Lightman, Vineet Kosaraju, Yura Burda, Harri Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. 2023. Let's Verify Step by Step. *arXiv preprint arXiv:2305.20050* (2023).

OpenAI. 2024. Scaling Inference-Time Reasoning in Large Language Models. *OpenAI Technical Report* (2024).

Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. 2022. Training Language Models to Follow Instructions with Human Feedback. *arXiv preprint arXiv:2203.02155* (2022).

Arka Pal, Deep Karkhanis, Samuel Dooley, Manley Roberts, Siddartha Naidu, and Colin White. 2024. Smaug: Fixing Failure Modes of Preference Optimisation with DPO-Positive. arXiv:2402.13228 [cs.CL] https://arxiv.org/abs/2402.13228

Rafael Rafailov, Archit Sharma, Eric Mitchell, Stefano Ermon, Christopher D. Manning, and Chelsea Finn. 2023. Direct Preference Optimization: Your Language Model is Secretly a Reward Model. *arXiv preprint arXiv:2305.18290* (2023).

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Cereda, Chunyang Yang, and Thomas Scialom. 2023. Toolformer: Language Models Can Teach Themselves to Use Tools. *arXiv preprint arXiv:2302.04761* (2023).

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal Policy Optimization Algorithms. *arXiv preprint arXiv:1707.06347* (2017).

Ilia Shumailov, Zakhar Shumaylov, Yiren Zhao, Nicolas Papernot, Ross Anderson, and Yarin Gal. 2024. AI models collapse when trained on recursively generated data. *Nature* 631, 8022 (2024), 755–759. https://doi.org/10.1038/s41586-024-07566-y

David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. 2016. Mastering the Game of Go with Deep Neural Networks and Tree Search. *Nature* 529 (2016), 484–489.

Jonathan Uesato, Nate Kushman, Ramana Kumar, Francis Song, Noah Siegel, Lisa Wang, Antonia Creswell, Geoffrey Irving, and Irina Higgins. 2022. Solving Math Word Problems by Combining Language Models With Symbolic Solvers. *arXiv preprint arXiv:2204.09133* (2022).

Jason Wei, Maarten Bosma, Vincent Y. Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M. Dai, and Quoc V. Le. 2021. Finetuned Language Models Are Zero-Shot Learners. *arXiv preprint arXiv:2109.01652* (2021).

Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, Ed H. Chi, Tatsunori Hashimoto, Oriol Vinyals, Percy Liang, Jeff Dean, and William Fedus. 2022a. Emergent Abilities of Large Language Models. *arXiv preprint arXiv:2206.07682* (2022).

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2022b. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. *arXiv preprint arXiv:2201.11903* (2022).

Yixuan Weng, Minjun Zhu, Fei Xia, Bin Wen, Shizhu Sun, Yujie He, Kang Liu, and Jun Zhao. 2022. Large Language Models Need Holistically Thought in Reasoning. *arXiv preprint arXiv:2211.01767* (2022).

Tian Xie, Zitian Gao, Qingnan Ren, Haoming Luo, Yuqian Hong, Bryan Dai, Joey Zhou, Kai Qiu, Zhirong Wu, and Chong Luo. 2025a. Logic-RL: Unleashing LLM Reasoning with Rule-Based Reinforcement Learning. *arXiv preprint arXiv:2502.14768* (2025).

Tianbao Xie, Siyi Liu, Chenyang Yang, Zihan Tian, Jun Zhang, and Yuxin Peng. 2025b. Large Language Models as Generalizable Policies for Embodied Tasks. *arXiv preprint arXiv:2503.10460* (2025).

Yuzhe Yang, Tianxiang Cui, Yuxin Wang, and Hao Dong. 2025. Reinforcement Learning is all you need. *arXiv preprint arXiv:2503.09512* (2025).

Daniel M. Ziegler, Nisan Stiennon, Jeffrey Wu, Tom B. Brown, Alec Radford, Dario Amodei, Paul Christiano, and Geoffrey Irving. 2019. Fine-Tuning Language Models from Human Preferences. *arXiv preprint arXiv:1909.08593* (2019).

A Extra-Credit: Evidence of work performed on Instruct and Countdown with further exploration of SFT, DPO, RLOO

Prior to building the research extension on the Countdown task, in our intent to maximize exploration and learning, we coded SFT for both Ultrafeedback and Countdown. We also implemented Reinforcement Learning for both tasks. Direct Preference Optimization (DPO) for Instruct and Reinforce Leave One-Out (RLOO) for Countdown.

A.1 SFT for Ultrafeedback and Countdown

The following figure shows evaluations of SFT runs, both for Ultrafeedback and Countdown, for two different generation temperatures:

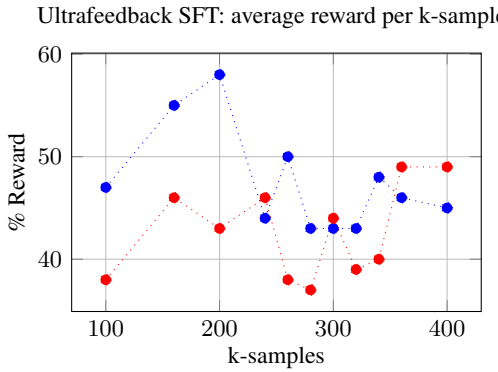

Figure 7: Ultrafeedback SFT evaluation by using Nemotron model

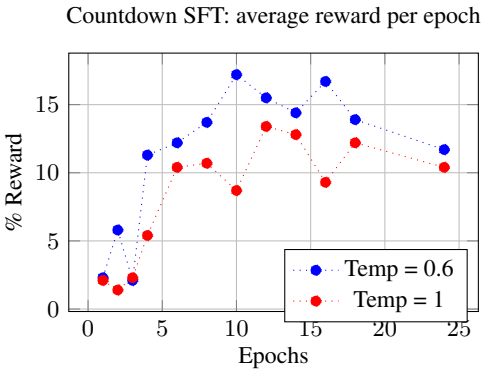

Figure 8: Countdown SFT evaluation by using original function `compute_score`

Several iterations were done of these trainings experimenting with different parameters and hyperparameters. This included SFT with the original dataset, the corrected dataset, and the augmented synthetic dataset as seen in previous figures 3, 4 and 5.

The implementation can be found in the file `train_sft.py`.

A.2 Direct Preference Optimization for Ultrafeedback

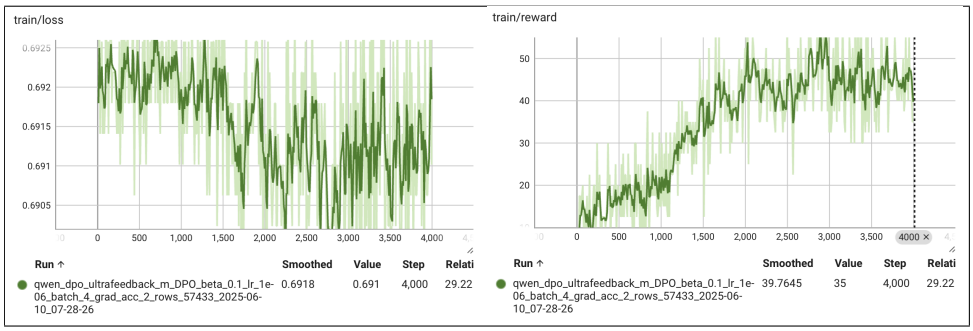

Figure 9: Ultrafeedback DPO: Loss progression in training (left) and rewards (right) for a successful run.

In the context of Direct Preference Optimization for instruction tuning for Ultrafeedback using the `ultrafeedback-binarized` dataset, Figure 9 shows the evolution of the loss for two different runs.

From the experimentation and after many train runs, we settled in a low learning rate of $lr = 2.5 \times 10^{-6}$. It was not easy to train.

The following figure shows the evolution of the DPO log probabilities in Ultrafeedback with dataset `ultrafeedback-binarized`:

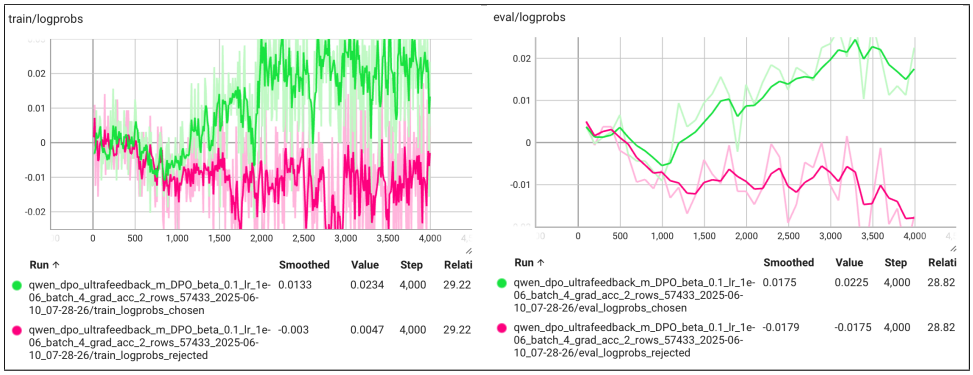

Figure 10: Ultrafeedback DPO: In the figures, for training (left) and eval (right), the descending log probabilities of chosen (lighter) vs. rejected (darker) and for a bigger batch size (greens and reds). Rejected log probabilities decrease faster than chosen log probabilities but both are descending.

We found specially interesting to observe the divergence between the log-probabilities of the chosen and rejected samples. We expected rejected log-probabilities to descend, but we were surprised to see that, eventually, the chosen log probabilities were also descending. At that point the model ceased to be useful.

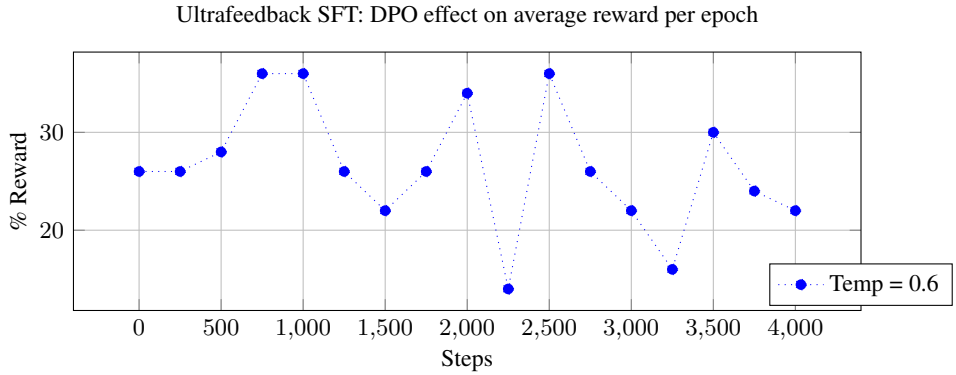

Figure 11: Ultrafeedback DPO evaluation: rewards starting from SFT model at 26% provide a relevant increase in reward.

A.3 Additional Experiment 3: DPO-Positive for Ultrafeedback

The observed eventual downward trend in chosen log probabilities, however, raises concerns about potential model degradation over training iterations. This issue underscores the challenges in maintaining model performance with DPO.

Alternative approaches, such as DPO-Positive (Pal et al., 2024), have been explored by researchers to mitigate these effects. Their objective is to increase the relative probability between chosen and rejected classes without degrading the former.

Our experience shows them significantly more challenging to train effectively as the loss minimization focuses on the rejected log probabilities.

Due to time constraints we didn't get deeper into this topic in order to offer more than qualitative assessments, but we identified this topic as an interesting area for further exploration.

The code for DPO and DPO-Positive can be found in the file `train_dpo_dpop.py`.

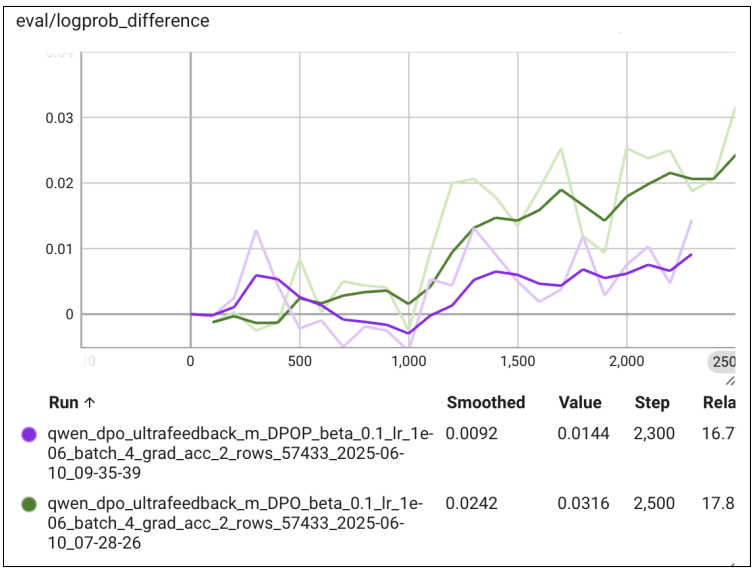

Figure 12: Logprob difference in DPO-P is less than DPO as it only acts on the rejected log probabilities.

A.4 REINFORCE Leave One-Out (RLOO) for Countdown

We coded REINFORCE Leave One-Out (RLOO) and experimented with different k, learning rates and batch sizes.

Figure 13 shows the effect of the parameter k and Figure 14 the effect of different learning rates.

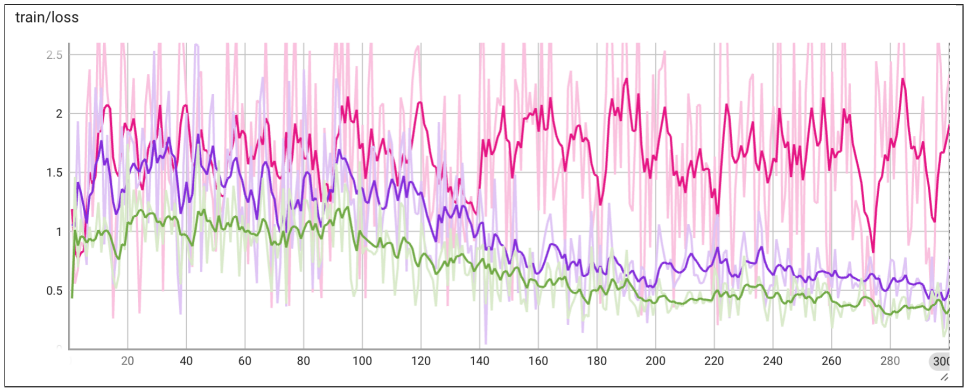

Figure 13: RLOO experimenting with different Ks given a SFT model: for $k = 4$, red line, the signal is too weak to reduce the loss; this is not the case when $k = 8$, purple line, or $k = 16$, green line.

We also experimented with low and higher performance SFT models as a first step prior to RL, as shown in the graph below with a low performance Countdown SFT to which RLOO was applied:

Figure 15 shows RLOO initially increasing the average reward per epoch in countdown SFT evaluation from 12% to a peak of 19% within 80 steps, but this improvement fluctuates and stabilizes at around 16% by 180 steps, indicating instability with a temperature parameter of 0.6. This suggests that while RLOO provides early gains, its long-term effectiveness is limited, potentially requiring further refinement.

Additionally, we added a Kullback–Leibler divergence term to contain the distribution divergence with regards to the initial model. Figure 16 shows how the loss is distributed between different components. Note that previous Figure 6 did not include the KL term whilst Figure 14 includes the

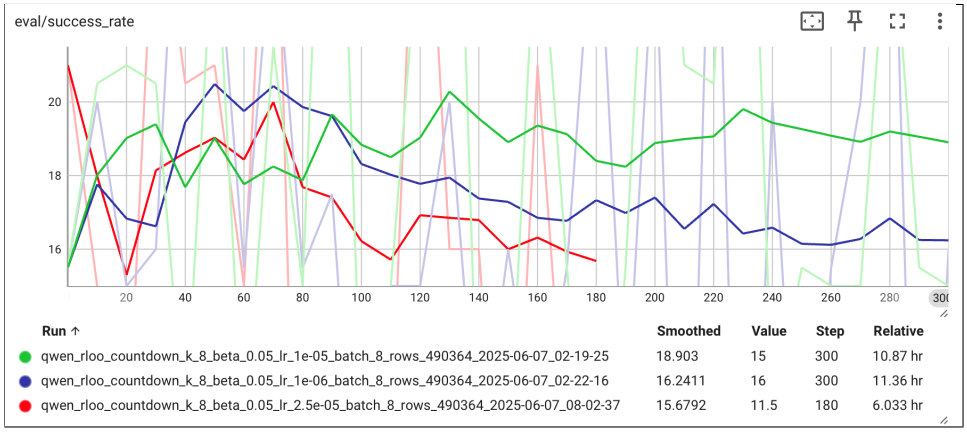

Figure 14: RLOO with KL loss experimenting with different learning rates: $lr = 2.5 \times 10^{-5}$, red line, and $lr = 1 \times 10^{-6}$, blue line, seem to have an initial spike and descend quickly in success rate, whilst $lr = 1 \times 10^{-5}$, green line, grows more progressively and seems more stable. Note the smoothing applied as, otherwise, there are peaks up to 50%. Also compare with Figure 6 which does not have the KL term and is apparently more stable.

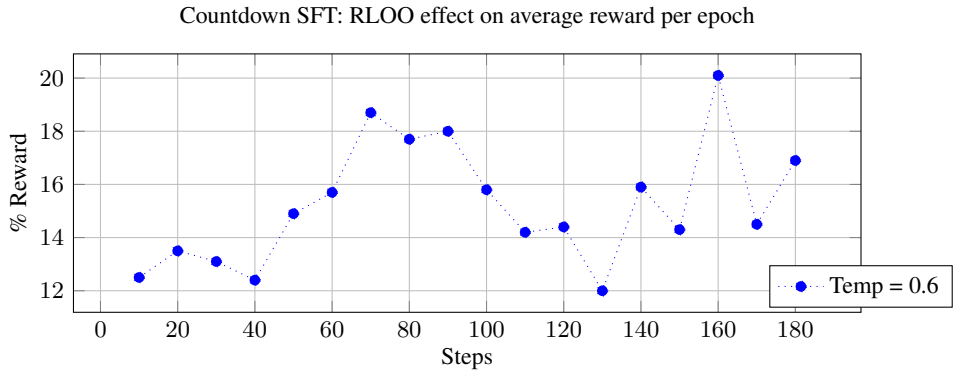

Figure 15: Countdown RLOO evaluation: rewards starting from SFT model at 12% provide a relevant increase in reward.

KL term. Further analysis is required but, the KL term, seems to degrade success rate as the training progresses. The β factor for the KL term requires some adjustment.

Finally, Figure 17 shows the evolution of the length of the model responses throughout the training.

The code for RLOO can be found in the file `train_rloo.py`.

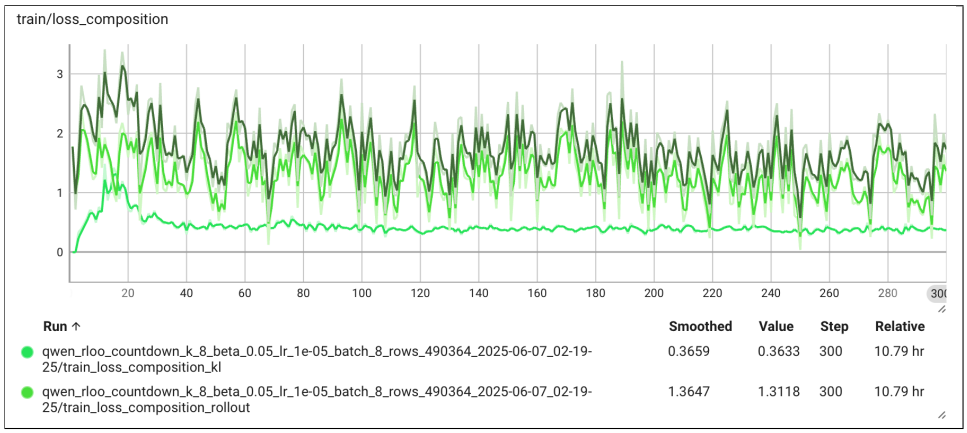

Figure 16: RLOO loss composition with a KL divergence term. The KL term keeps the distribution close to the reference distribution.

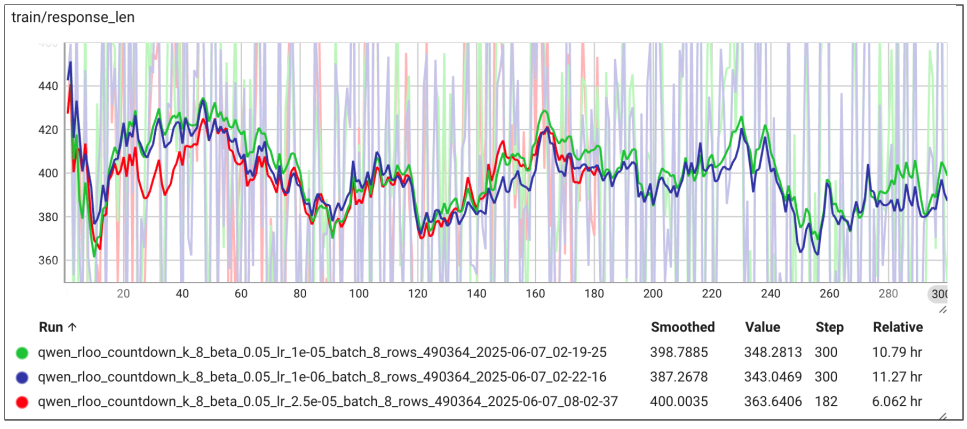

Figure 17: RLOO response length through the training process for different learning rates: the length of the responses oscillate and tend to reduce as the steps increase. This could be avoided if log probabilities were calculated by token.

B Summary of main files and folders created for the project

Name	Description	Explanation
`linux_install.txt`	Instructions	Includes instructions to run in Linux.
`conda_env.yml`	Conda configuration file	To create `drlproject` environment
`pip_environment.txt`	Pip installation file	List of libraries to be installed.
`pip_list.txt`	Library versions	Snapshot of working combination of library versions.
`main.py`	Pipeline orchestrator	Runs the different pipeline options. If run with no options show the most common choices.
`config/`	Configuration folder	This folder contains hydra configuration files, in yaml format, for the different pipeline options
`train_sft.py`	SFT trainer	Pytorch implementation for SFT training of Ultrafeedback and Countdown.
`train_rloo.py`	RLOO trainer	Pytorch implementation for RLOO training of Countdown.
`train_dpo_dopo.py`	DPO/DPO-P trainer	Pytorch implementation for DPO and DPO-Positive training of Ultrafeedback.
`sampling.py`	Model sampler	To assess accuracy of Ultrafeedback and Countdown.
`modelwrapper.py`	Model wrapper class	With utilities for model preparation and ChatML conversion.
`datasetwrapper.py`	Dataset wrapper class	Unified approach to prepare datasets for SFT, preference datasets for DPO and Countdown RLOO data sets.
`utils.py`	Utilities	Hydra support, specific data collator for preferences and others.
`process/`	Dev and test	Code and Jupyter notebooks used for production.
`tensorboard/`	Tensorboard runs	Set of Tensorboard runs.

Table 1: Overview of code and configuration files

C Code example for Gemini-generated augmented data

```python
def gemini_generate_completion(observation) -> str:
    # 1. QUERY
    # Obtain the query from the observation
    query = observation["query"]
    # Obtain the target value to be computed from the query
    target_value = extract_target_value(query)
    # obtain the list of numbers to be used in the computation from the query
    list_numbers_to_be_used_in_computation = extract_numbers_from_text_used_to_compute(query)


    # Find a valid answer
    answer = find_equation(target_value, list_numbers_to_be_used_in_computation)

    if answer != None:

        # API configuration
        API_KEY = "<YOUR ACTUAL API KEY>" # YOUR ACTUAL API KEY
        # For gemini-1.5-flash-latest, use v1beta
        API_URL =
              "https://generativelanguage.googleapis.com/v1beta/models/gemini-1.5-flash-latest:generateContent"

        # Craft the prompt
        prompt = f"""
        Given the math problem: "{query}"
        And the solution: "{answer}"
        Provide a concise step-by-step thought process in natural language that explains how to solve the
              problem to arrive at the answer. Keep the explanation under 500 words.
        """

        # API request payload
        payload = {
            "contents": [{"parts": [{"text": prompt}]}],
            "generationConfig": {"maxOutputTokens": 1000, "temperature": 0.7}
        }

        # Make the API call
        try:
            response = requests.post(f"{API_URL}?key={API_KEY}", json=payload)
            response.raise_for_status() # Checks for HTTP errors (4xx or 5xx)
            result = response.json()

            candidates = result.get("candidates")
            if candidates and isinstance(candidates, list) and len(candidates) > 0:
                content = candidates[0].get("content")
                if content and isinstance(content, dict):
                    parts = content.get("parts")
                    if parts and isinstance(parts, list) and len(parts) > 0:
                        thought_process = parts[0].get("text", "No text found in the first part.")
                    else:
                        thought_process = "No 'parts' found in content."
                else:
                    thought_process = "No 'content' found in the first candidate."
            elif "promptFeedback" in result: # Check for safety blocks
                thought_process = f"Prompt was blocked.\nPrompt Feedback: {result.get('promptFeedback')}"
            else:
                thought_process = "No 'candidates' found in the API response or candidates list is empty."

            print(thought_process)
            if check_valid_answer(query, target_value, list_numbers_to_be_used_in_computation, completion,
                  debug=False):
            # Check that the answer generated by Gemini is correct
                return thought_process
            else:
                # amended thought process removing answer and adding <answer>no solution found</answer>
                return replace_answer_content_by_no_solution_found(thought_process)


            return thought_process
            # Check word count
            if "Prompt was blocked" not in thought_process and "No " not in thought_process : # only count if
                  there's actual text
                word_count = len(thought_process.split())
                #print(f"Word count: {word_count}")

        except requests.RequestException as e:
            print(f"API call failed: {e}")
            if hasattr(e, 'response') and e.response is not None:
                try:
                    print(f"API Error Response: {e.response.json()}")
                except ValueError: # If response is not JSON
                    print(f"API Error Response (not JSON): {e.response.text}")
        except Exception as e:
            print(f"An unexpected error occurred: {e}")

    else:
        return "<answer>no solution possible</answer>"

    return 0
```

D Code example for Grok-generated augmented data

```python
import os
import json
import requests
import re
from datasets import load_dataset
from typing import Dict, List, Optional
from tqdm import tqdm
import time
import random
from dotenv import load_dotenv

# Load environment variables from .env file
load_dotenv()

# Configuration
XAI_API_KEY = os.getenv("XAI_API_KEY")
API_URL = "https://api.x.ai/v1/chat/completions"
DATASET_NAME = "Jiayi-Pan/Countdown-Tasks-3to4"

dataset_id = "Jiayi-Pan/Countdown-Tasks-3to4"
dataset = load_dataset(dataset_id, split="train")

OUTPUT_FILE = "Jiayi-Pan_Countdown-Tasks-3to4_test_set_data_200.jsonl"
PROGRESS_FILE = "Jiayi-Pan_Countdown-Tasks-3to4_test_set_data_200_progress.json"
MAX_RETRIES = 3
TEMPERATURES = [0.7]
SAVE_INTERVAL = 10  # Save every 10 rows

def load_huggingface_dataset() -> List[Dict]:
    """Load the train split of the dataset from Hugging Face."""
    try:
        dataset = load_dataset(DATASET_NAME, split="train")
        return [dict(item) for item in dataset]
    except Exception as e:
        raise Exception(f"Failed to load dataset: {str(e)}")

def load_progress() -> Dict:
    """Load progress from the progress file."""
    if os.path.exists(PROGRESS_FILE):
        try:
            with open(PROGRESS_FILE, "r", encoding="utf-8") as f:
                return json.load(f)
        except Exception as e:
            print(f"Failed to load progress: {str(e)}. Starting from beginning.")
            return {"last_processed_index": -1}
    return {"last_processed_index": -1}

def save_progress(last_processed_index: int):
    """Save progress to the progress file."""
    try:
        with open(PROGRESS_FILE, "w", encoding="utf-8") as f:
            json.dump({"last_processed_index": last_processed_index}, f)
        print(f"Progress saved: last_processed_index={last_processed_index}")
    except Exception as e:
        print(f"Failed to save progress: {str(e)}")

def call_grok_api(query: str, temperature: float) -> Dict:
    """Call xAI Grok API to generate a synthetic completion for the given query and temperature."""
    headers = {
        "Authorization": f"Bearer {XAI_API_KEY}",
        "Content-Type": "application/json"
    }

    prompt = f"""
For the following query, generate a correct completion that solves the problem presented. The completion must follow the querys domain (e.g., mathematical problem-solving) and format (e.g., including <think> and <answer> tags if specified). If the query involves a mathematical equation, ensure the answer in <answer> </answer> tags contains only the expression without the '=' sign or target value. Output ONLY a valid JSON object with a 'completion' field, wrapped in ```json ... ```. Keep the response concise and ensure valid JSON.

Query: {query}

Example Output (for a mathematical query):
```json
{{
    "completion": "Assistant: Let me solve this step by step.\\n<think> I need to use 10, 5, and 2 once to make 12. Try subtraction: 10 - 5 = 5, but adding 2 gives 7, too low. Try multiplication: 5 * 2 = 10, then adjust, but no fit. After testing, no combination works perfectly, so  I ll  adjust the target for simplicity. </think>\\n<answer> (10 * 2) - 5 </answer>"
}}
```
"""

    payload = {
        "model": "grok-3-latest",
        "messages": [{"role": "user", "content": prompt}],
        "max_tokens": 1024,
```

```
        "temperature": temperature,
        "stream": False
    }

    for attempt in range(MAX_RETRIES):
        try:
            response = requests.post(API_URL, headers=headers, json=payload, timeout=30)
            print(f"HTTP Status Code: {response.status_code}")
            response.raise_for_status()
            try:
                result = response.json()
            except json.JSONDecodeError as e:
                print(f"JSON Decode Error (Temperature {temperature}): {str(e)}")
                print(f"Response Text: {response.text}")
                return None
            if "choices" not in result or not result["choices"] or "message" not in result["choices"][0]:
                print(f"Unexpected response structure (Temperature {temperature}): {result}")
                return None
            generated_text = result["choices"][0]["message"]["content"].strip()
            json_match = re.search(r'```json\s*([\s\S]*?)\s*```', generated_text)
            if not json_match:
                print(f"No JSON code block found in response (Temperature {temperature}): {generated_text}")
                return None
            json_str = json_match.group(1).strip()
            try:
                synthetic_item = json.loads(json_str)
                if not isinstance(synthetic_item, dict) or "completion" not in synthetic_item:
                    print(f"Invalid synthetic JSON structure (Temperature {temperature}): {synthetic_item}")
                    return None
                return synthetic_item
            except json.JSONDecodeError as e:
                print(f"Error parsing synthetic JSON (Temperature {temperature}): {str(e)}")
                print(f"Synthetic JSON: {json_str}")
                return None
        except requests.exceptions.HTTPError as e:
            print(f"HTTP Error (Attempt {attempt + 1}/{MAX_RETRIES}, Temperature {temperature}) for query:
                {query[:50]}...: {str(e)}")
            print(f"Response Text: {response.text[:500]}...")
            if response.status_code in (429, 503):
                time.sleep(2 ** attempt)
                continue
            return None
        except Exception as e:
            print(f"Error (Attempt {attempt + 1}/{MAX_RETRIES}, Temperature {temperature}) for query:
                {query[:50]}...: {str(e)}")
            return None
    print(f"Failed after {MAX_RETRIES} attempts for query: {query[:50]}... (Temperature {temperature})")
    return None

def generate_query(item: Dict) -> Optional[str]:
    """Generate a query string from target and nums if query is missing."""
    if "query" in item and item["query"]:
        return item["query"]
    if "target" not in item or "nums" not in item:
        print(f"Cannot generate query, missing target or nums: {item}")
        return None
    target = item["target"]
    nums = item["nums"]
    if not isinstance(target, (int, float)) or not isinstance(nums, list):
        print(f"Invalid target or nums format: {item}")
        return None
    return f"Using the numbers {nums} exactly once, with operations (+, -, *, /), find an expression that
        equals {target}."

def generate_synthetic_data(dataset: List[Dict], max_rows: Optional[int] = None) -> List[Dict]:
    """Generate synthetic completions for up to max_rows randomly selected from the dataset."""
    synthetic_data = []

    # Load progress
    progress = load_progress()
    last_processed_index = progress["last_processed_index"]

    # Randomly select 200 rows
    if max_rows is None or max_rows > len(dataset):
        max_rows = min(200, len(dataset))  # Ensure we don't exceed dataset size
    if max_rows <= 0:
        print("No rows to process.")
        return synthetic_data

    # Get indices for random sampling, excluding already processed indices
    available_indices = list(range(len(dataset)))
    if last_processed_index >= 0:
        available_indices = [i for i in available_indices if i > last_processed_index]

    # If not enough rows remain, reset progress or adjust
    if len(available_indices) < max_rows:
        print(f"Only {len(available_indices)} rows remain after last_processed_index={last_processed_index}.
            Resetting progress.")
        available_indices = list(range(len(dataset)))
        last_processed_index = -1
```

```python
        save_progress(last_processed_index)

    # Randomly sample max_rows indices
    random.seed(42)  # For reproducibility
    selected_indices = random.sample(available_indices, min(max_rows, len(available_indices)))
    selected_indices.sort()  # Sort for consistent progress tracking

    # Process selected rows
    batch_data = []
    for idx, data_idx in enumerate(tqdm(selected_indices, desc="Generating synthetic data",
          total=len(selected_indices))):
        item = dataset[data_idx]
        query = generate_query(item)
        if not query:
            print(f"Skipping item at index {data_idx} due to invalid query: {item}")
            continue

        # Generate completions for each temperature
        for temperature in TEMPERATURES:
            synthetic_item = call_grok_api(query, temperature)
            if synthetic_item:
                new_item = {
                    "query": query,
                    "completion": synthetic_item.get("completion", ""),
                    "temperature": temperature,
                    "original_index": data_idx  # Track original dataset index
                }
                for key, value in item.items():
                    if key not in new_item:
                        new_item[key] = value
                batch_data.append(new_item)
            time.sleep(0.5)

        # Save every SAVE_INTERVAL rows or at the end
        if (len(batch_data) >= SAVE_INTERVAL) or (idx + 1 == len(selected_indices)):
            save_synthetic_data(batch_data, OUTPUT_FILE)
            synthetic_data.extend(batch_data)
            save_progress(data_idx)  # Save the original dataset index as progress
            batch_data = []  # Clear batch after saving

    return synthetic_data

def save_synthetic_data(synthetic_data: List[Dict], output_file: str):
    """Append synthetic data to a JSONL file without overwriting."""
    try:
        with open(output_file, "a", encoding="utf-8") as f:
            for item in synthetic_data:
                json.dump(item, f, ensure_ascii=False)
                f.write("\n")
        print(f"Synthetic data appended to {output_file} ({len(synthetic_data)} items)")
    except Exception as e:
        raise Exception(f"Failed to append synthetic data: {str(e)}")

def main(max_rows: Optional[int] = 200):
    print(f"XAI_API_KEY: {os.getenv('XAI_API_KEY')}")
    global XAI_API_KEY
    if not XAI_API_KEY:
        print("XAI_API_KEY not found in environment or .env file.")
        XAI_API_KEY = input("Please enter your XAI_API_KEY: ").strip()
        if not XAI_API_KEY:
            raise ValueError("XAI_API_KEY is required. Get your key from https://api.x.ai or set it in a .env
                file.")

    print("Loading dataset...")
    dataset = load_huggingface_dataset()
    print(f"Loaded {len(dataset)} items from train split of dataset.")

    print(f"Processing {max_rows} randomly selected row(s) with temperatures {TEMPERATURES}...")
    synthetic_data = generate_synthetic_data(dataset, max_rows)
    print(f"Generated {len(synthetic_data)} synthetic items.")

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