Title

An Improved Post-Training Framework for Small Language Models Based on Group Relative Policy Optimization (GRPO)

ISBN

9798247915782

Author

Tattersall, Casey

Publication Date

2026-05-26

Peer reviewed|Thesis/dissertation

UNIVERSITY OF CALIFORNIA
Los Angeles

An Improved Post-Training Framework for Small Language Models Based on Group Relative Policy Optimization (GRPO)

A thesis submitted in partial satisfaction
of the requirements for the degree
Master of Science in Statistics

by

Casey Tattersall

2026

ABSTRACT OF THE THESIS

An Improved Post-Training Framework for Small Language Models
Based on Group Relative Policy Optimization (GRPO)

by

Casey Tattersall
Master of Science in Statistics
University of California, Los Angeles, 2026
Professor Yuhua Zhu, Chair

Pre-trained large language models require post-training to become useful problem solvers, and DeepSeek's Group Relative Policy Optimization (GRPO) has emerged as the dominant reinforcement learning algorithm for this purpose. However, current GRPO research is primarily focused on larger models, while many practical applications may be better served by smaller specialized models that can run locally on consumer-grade hardware. This thesis investigates how a GRPO-based training pipeline can be modified to work effectively at the small-model scale, using multi-digit multiplication as a controlled environment with cheap, verifiable, binary rewards.

Starting from Qwen2.5-1.5B-Instruct, we train and compare five LoRA-adapted models on a single NVIDIA A100 GPU through Google Colab. Our baseline GRPO run improves substantially over the base model, but suffers from mid-training instability and stagnant performance on harder problems. We then propose and evaluate three modifications: zeroing the KL coefficient on uninformative prompts whose rollout group produces no reward variation, adopting the DAPO modifications proposed by ByteDance, and adding a multi-

armed bandit-inspired data sampler that shifts training budget toward problem types at the frontier of the current model's ability. The first and third modifications produce substantial gains. Our best model reaches 100%, 94%, 92%, 69%, and 80% accuracy on 2×2, 3×3, 4×4, 5×5, and out-of-distribution 6×3 multiplication problems respectively, up from 89%, 30%, 0%, 0%, and 1% for the base model. We then investigate each model's responses to better understand the strategies they have learned and confirm that the final models are robust to variation in prompt format and therefore are not overfitting.

The thesis of Casey Tattersall is approved.

Ying Nian Wu

Xiaowu Dai

Yuhua Zhu, Committee Chair

University of California, Los Angeles

2026

To all of the friends, family, and mentors ...

who have helped me along the way

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

Introduction

1.1 Motivation

Pre-trained large language models (LLMs) are, at first, not very useful. Such models are effectively next-token predictors that have absorbed enormous quantities of data from the internet, but are only capable of continuing a given passage in a stylistically plausible way. They are not yet assistants or problem solvers, and they are not yet capable of consistently and reliably completing any specific task. The transformation from a raw pre-trained model to a refined useful model that consumers actually interact with happens during post-training. The typical post-training pipeline consists of supervised fine-tuning (SFT) on curated instruction–response pairs followed by some form of reinforcement learning (RL) against a reward signal that measures response quality [6].

DeepSeek's release of Group Relative Policy Optimization (GRPO) [8] and its use as the reinforcement learning backbone of DeepSeek-R1 [4] in early 2025 made GRPO-style methods the dominant approach for teaching LLMs to reason. Since then, frontier labs have leveraged GRPO to push reasoning performance higher on increasingly larger models. While these frontier LLMs are impressive problem-solvers, consumers are generally limited to interacting with them through APIs; most of today's best models are not open-source, and even if they were publicly available they are far too large to run on affordable consumer-grade hardware. But, a growing body of work suggests that many practical LLM applications, including agentic ones, don't require frontier scale and may be better served by smaller,

specialized models that can run locally [2]. Such models trade their generalization abilities for speed, cost, privacy, and the ability to operate offline; so, if the use case is simple enough, a smaller model will be the superior option.

However, much of the current post-training literature targets large, general-purpose models; we suspect that the optimal techniques may differ for models that are small and specialized. In this thesis, we experiment with such techniques on `Qwen2.5-1.5B-Instruct` [9], a smaller open-source model that can easily be run locally on a consumer-grade GPU.

Why Multiplication?

We choose multi-digit multiplication as the training task throughout this thesis. This choice requires some defense, since LLMs are obviously not the right tool for arithmetic (a four-line Python program does the job far more accurately and cost-effectively). To be clear, we are not attempting to build a useful multiplication service; we are simply using multiplication as an environment to test different training methodologies.

Multiplication has three properties that make it attractive for such testing:

First, the task is naturally decomposable in a way that resembles the kind of repetitive, formulaic work a specialized small model might be asked to do in production [3]. No LLM of any size is going to memorize the result of 12585×24579 directly, but what a model *can* do is memorize the results of small multiplications — such as all one-by-one digit products, for instance — and learn a procedure that decomposes a larger product into a sum of these smaller ones, like this:

$$12585 \times 24579 = 12585 \times 2 \cdot 10^4 + 12585 \times 4 \cdot 10^3 + ... \tag{1.1}$$

So, if we can build a framework in which a small model learns a valid decomposition strategy for multiplication, we should expect this framework to generalize well to other real-world tasks.

Second, training data is effectively unlimited and free. For many problem types, it can

be challenging (or impossible) to find a free, high-quality dataset to use in post-training. Most public reasoning datasets are small and, at this point, likely already seen by the model during pre-training.

Third, the reward is binary and easy to verify. A multiplication problem has exactly one correct numerical answer, and checking whether the model produced it requires only a regex extraction and an integer comparison. This is the ideal setting for GRPO; the reward is simple, cheap to compute, and not game-able in any way.

So, to be clear, the goal of this thesis is not to produce a useful model. The goal is to study how a GRPO-based training pipeline can be modified to work well at the small-model scale, using multiplication as our controlled environment, with the expectation that the modifications we develop here will transfer to other repetitive, decomposable tasks.

1.2 Selection of Qwen2.5-1.5B-Instruct

There are numerous small open-source LLMs for us to choose from, but for this thesis we selected `Qwen2.5-1.5B-Instruct` as our base model. This model was an ideal choice for two primary reasons.

First, all models in the Qwen2.5 family received targeted pretraining on mathematical and code-heavy data. This gives the base model a nontrivial amount of arithmetic competence right out of the box. As we document in Section 3.2.1, `Qwen2.5-1.5B-Instruct` already solves the vast majority of 2×2-digit and a few 3×3-digit multiplication problems. The standard post-training pipeline applies SFT to a base model before RL in order to ensure that the base model produces enough correct answers for the RL signal to be informative [6], as a model that gets every training question wrong produces no positive rewards and therefore no learning signal. However, since `Qwen2.5-1.5B-Instruct` already answers easy problems correctly some of the time, we are able to skip SFT entirely and go straight into GRPO.

Second, the `-Instruct` variants of Qwen2.5 have already been post-trained by the Qwen team on general-purpose instruction-following data [9]. As a result, the checkpoint we start from already knows how to follow English instructions and already has some familiarity with producing content inside XML-style tags such as `<think>` and `<answer>`. We require the model to use tags like these to allow for fast and simple answer extraction and verification, and the fact that Qwen2.5 has seen them before allows us to skip a significant amount of early-stage format shaping that would otherwise be necessary. In order to go straight into GRPO, it's not enough for the base model to solve some 2×2 and 3×3-digit problems correctly – it must also format the answer properly, so that we can systematically extract it and assign the correct reward.

Also, we should note that we are compute-constrained. All experiments in this thesis were run on a single NVIDIA A100 80GB GPU accessed through Google Colab, with a 24-hour session limit that caps the maximum length of each individual training run. So, while smaller (0.5B) and larger (3B, 7B, and more) `Qwen2.5-Instruct` checkpoints are also available, baseline performance was too low for 0.5B, and the larger models require trade-offs in terms of group size and maximum generation length in order to fit onto our single 80GB A100. And regardless, our goal is to work with models that can feasibly be run locally by consumers, so we wanted to work with as small of a model as possible.

CHAPTER 2

Overview of Group Relative Policy Optimization

2.1 What is GRPO?

Group Relative Policy Optimization (GRPO) was introduced by the DeepSeek team in the DeepSeek-Math technical report [8] and subsequently used as the reinforcement learning backbone of DeepSeek-R1 [4], which took the artificial intelligence community by storm upon its release in January 2025. However, this new algorithm still has a lot in common with the previous methods; it is still a policy-gradient method in the PPO [7] family, and it retains the same clipped surrogate objective that made PPO the standard choice for RLHF. The key difference lies in how the advantage is computed.

PPO estimates per-timestep advantages A_t using a learned value function V_ϕ, typically a neural network of roughly the same size as the policy itself. For every state s_t visited during a rollout, the value network predicts the expected future return, and the advantage is computed as a function of this prediction and the realized return (for example, via generalized advantage estimation). This works well in principle, but in the LLM setting it is expensive: a value network the size of the policy doubles the memory footprint of training, and its predictions in the high-dimensional, sparse-reward regime of text generation are often noisy. The value network is also the source of substantial engineering complexity; it needs its own optimizer, its own hyperparameters, and careful treatment of credit assignment across a long generated sequence.

GRPO, however, removes this expensive value network entirely. Given a prompt q, the

policy π_θ generates a group of G completions $\{o_1, o_2, \ldots, o_G\}$ with non-greedy sampling. Each completion receives a scalar reward r_i from the external reward function (in our case, $+1$ for correct answers and -1 otherwise). The advantage of completion i is then computed as its standardized reward within the group:

$$A_i \;=\; \frac{r_i - \text{mean}(\mathbf{r})}{\text{std}(\mathbf{r}) + \varepsilon}, \tag{2.1}$$

where $\mathbf{r} = (r_1, r_2, \ldots, r_G)$ and ε is a small constant for stability. The group itself serves as the baseline against which each sample is compared: above-average completions receive positive advantage, below-average completions receive negative advantage. If every completion in the group receives the same reward (i.e. all answers were correct or all answers were incorrect), then every advantage is exactly zero. This may cause some problems, which we will discuss further in Chapter 4.

Each token in completion o_i is then assigned the same sequence-level advantage $A_{i,t} = A_i$, and the PPO clipped surrogate is applied at the token level:

$$\mathcal{L}_{\text{PG}}(\theta) \;=\; -\frac{1}{G}\sum_{i=1}^{G}\frac{1}{|o_i|}\sum_{t=1}^{|o_i|}\min\Big(w_{i,t}(\theta)\,A_{i,t},\; \text{clip}(w_{i,t}(\theta),\,1-\varepsilon,\,1+\varepsilon)\,A_{i,t}\Big), \tag{2.2}$$

where $w_{i,t}(\theta) = \pi_\theta(o_{i,t} \mid q, o_{i,<t}) / \pi_{\theta_{\text{old}}}(o_{i,t} \mid q, o_{i,<t})$ is the importance-sampling ratio between the current and old policies. Finally, a KL-divergence penalty against a fixed reference policy π_{ref} is added to keep the trained policy from drifting too far from its starting point:

$$\mathcal{L}_{\text{GRPO}}(\theta) \;=\; \mathcal{L}_{\text{PG}}(\theta) + \beta \cdot D_{\text{KL}}(\pi_\theta \,\|\, \pi_{\text{ref}})\,. \tag{2.3}$$

In practice, the KL divergence is estimated per-token from the log-probabilities of the generated sequence under each policy. Our implementation uses the unbiased estimator of Schulman,

$$\hat{D}_{\text{KL}}^{\text{tok}} \;=\; \exp\big(-\Delta_{i,t}\big) - 1 + \Delta_{i,t}, \qquad \Delta_{i,t} \;=\; \log \pi_\theta(o_{i,t}) - \log \pi_{\text{ref}}(o_{i,t}), \tag{2.4}$$

with Δ clipped to $[-10, 10]$ for numerical stability. This estimator is always non-negative and has the same expectation as the true KL, and it is the version used in the DeepSeek-Math implementation [8].

2.2 Applying GRPO to LLMs

The property that makes GRPO particularly well-suited to LLM training is, as noted above, the elimination of the expensive value network. In the classical RLHF setting, the actor–critic pair roughly doubles the number of trainable parameters; for a 1.5B-parameter policy this is a significant cost, and for large frontier-sized models it becomes prohibitive. GRPO replaces those parameters with on-the-fly Monte Carlo sampling from the policy itself, replacing this value network with an ongoing inference cost of G generations per prompt. For reasoning tasks with rewards that are cheap and easy to calculate, this trade is extremely favorable.

A GRPO step at the level of one prompt can be described intuitively as follows. Suppose the model is given the question "What is 47×92?". We sample $G = 8$ completions from the policy at a moderately high temperature to encourage diversity. Some fraction of these completions will happen to arrive at the correct answer (4,324) through whatever decomposition strategy the model currently favors; the remainder will arrive at an incorrect answer, either because the decomposition was wrong or because an arithmetic step was wrong. Suppose 3 of the 8 are correct and 5 are wrong. The rewards are $(+1, +1, +1, -1, -1, -1, -1, -1)$ in some order. Equation (2.1) standardizes these into advantages of roughly $(+1.291, +1.291, +1.291, -0.775, -0.775, -0.775, -0.775, -0.775)$. The policy-gradient term in Equation (2.2) then nudges π_θ to increase the probability of every token in the three correct completions and decrease the probability of every token in the five incorrect completions. The KL term in Equation (2.3) pulls the update back toward π_{ref}, ensuring that the model does not diverge too far from its starting behavior. While there is plenty of noise in this signal, across many such updates we see that strategies that reliably produce correct answers are reinforced and strategies that do not are suppressed, even though the training procedure never receives an explicit labeled example of what a "good" decomposition looks like.

Now, it should be clear why the base model's initial nonzero success rate is so important.

If every rollout in every group received the same reward, every advantage would be zero, the policy-gradient term would vanish, and we would not be doing any reinforcing or any learning.

Token-Wise Loss Adjustment

The formulation in Equation (2.2) uses a particular normalization: the per-token losses are first averaged within each response (the inner $1/|o_i|$), and the resulting per-response losses are then averaged across the group (the outer $1/G$). This is the normalization used in the original DeepSeek-Math GRPO formulation [8]. In every experiment in this thesis, we deviate from this convention and instead average over tokens globally:

$$\mathcal{L}_{\mathrm{PG}}^{\mathrm{tok}}(\theta) \;=\; -\frac{1}{\sum_{i=1}^{G}|o_i|}\sum_{i=1}^{G}\sum_{t=1}^{|o_i|}\min\Big(w_{i,t}(\theta)\,A_{i,t},\;\mathrm{clip}(w_{i,t}(\theta),\,1-\varepsilon,\,1+\varepsilon)\,A_{i,t}\Big). \quad (2.5)$$

In the code, this corresponds to computing a per-token loss tensor, multiplying elementwise by a binary mask that selects generated positions, summing over all positions, and dividing by the sum of the mask. Responses of different lengths within the same group therefore contribute in proportion to their token counts rather than contributing equally.

This change, which we adopt throughout the thesis, is the "token-level loss aggregation" advocated by the DAPO paper [11] that we discuss further in Section 4.4. Its motivation is straightforward: in the response-level normalization of Equation (2.2), every token in a short response receives a larger gradient contribution than every token in a long response, simply because the short response's inner $1/|o_i|$ is bigger. A long correct response is penalized, per token, relative to a short correct one, even though the long response may be the one that actually walks through the decomposition carefully and arrives at the right answer for the right reason. We believe that this modification is better suited for the long-form multiplication reasoning our models will be working on.

This is the only modification to the original GRPO objective that is shared by every model in this thesis. The remaining chapters examine modifications that are applied to

some models and not others.

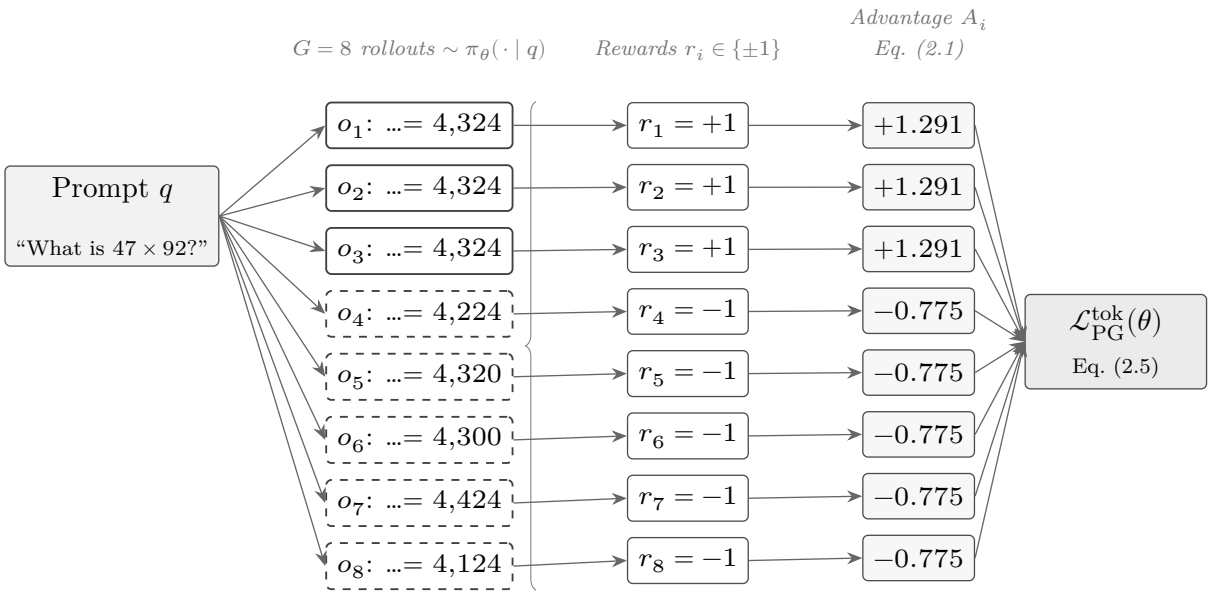


Figure 2.1: Illustration of one GRPO update step on a single multiplication prompt.

CHAPTER 3

Setting Up the Training Framework

3.1 Training Infrastructure

All experiments in this thesis were implemented in Python using PyTorch and the Hugging Face Transformers library, with LoRA adapters attached via the PEFT library [5]. We wrote the GRPO training loop from scratch (rather than building on top of a higher-level reinforcement learning framework) as this allowed us to ensure that every minor adjustment we made was implemented correctly.

Training was performed on a single NVIDIA A100 80GB GPU provisioned through Google Colab. To optimize the process, we loaded the base model in bfloat16, used the SDPA kernel for faster torch attention computations, enabled gradient checkpointing to save on VRAM, and simulated larger batch sizes through gradient accumulation. As mentioned previously, Google Colab imposes a hard 24-hour session limit, and we used the full 24 hours for all five training runs that we will discuss in this thesis. A single session typically accommodates around 10,000 training steps (each step being one prompt with eight group-sampled generations), but this can vary based on a variety of factors, including the average generation length (i.e. when the model tends to think for more tokens, each step takes more wall-clock time). Also, even though all training was done on an 80GB A100, we observed some random variation between instances; some A100s seemed to generate responses slightly faster than others.

3.2 Prompt Development

3.2.1 Answer Formatting and Baseline Results

The prompt used for Models 1 through 4 is shown in Figure 3.1 below. It has a simple structure: a direct question, an instruction to use the `<think>` and `<answer>` tags, a single miniature worked example, an instruction to end generation after the answer, and an instruction not to write code. Arriving at this prompt took a surprising amount of iteration, but as previously discussed this was essential in order to raise baseline accuracy high enough to avoid the need for the SFT step and jump straight into RL.

Complete Prompt Template

```
System prompt: You are Qwen, created by Alibaba Cloud. You are a helpful
assistant.

User prompt:
What is {a} * {b}?

Think through the problem between <think> and </think> tags, and return your
final numerical answer between <answer> and </answer> tags.

For example, 12 * 2 = <think> 12 * 2 = 10 * 2 + 2 * 2 = 20 + 4 = 24 </think>
<answer>24</answer>

End the response immediately after giving the answer.
Do not write code, solve the problem manually.
```

Figure 3.1: The prompt template used for Models 1 through 4. The numbers `{a}` and `{b}` change for each problem, while every other character remains fixed.

As we discussed, Qwen2.5 is already capable of producing content inside XML-style tags [9], and the `<think>`/`<answer>` pair in particular has become a near-standard convention in

the reasoning-model community since DeepSeek-R1 [4]. We experimented with other tagging schemes, but they all led to an increase in malformed responses. Of course, any reasonable tagging scheme could easily be taught to the model through SFT, but this prompt was good enough to allow for the unmodified `Qwen2.5-1.5B-Instruct` to be our starting point for GRPO.

This tag structure is essential for making evaluation fast and cheap, as extracting the model's answer reduces to a single regular expression. If the extraction fails, or the extracted value can't be cast to an integer, the response is marked as a formatting error and automatically counted as incorrect. This is why multiplication is such a nice dataset to work with; since no slow, expensive, and/or unreliable evaluation methods are required, we can perfectly grade hundreds of thousands of responses at essentially zero cost.

The worked example in the prompt (12×2 decomposed into $10 \times 2 + 2 \times 2$) was also crucial to the baseline accuracy. Without it, the base model rarely tried to decompose the problem at all (often leaving the `<think>` section entirely blank) and answered all but the simplest questions incorrectly. Interestingly, our best final models no longer have this problem, and the worked example is no longer required (see Section 6.2). However, for our starting point, adding this example was the final key step we needed to increase the base model's accuracy on 2×2 and 3×3 problems enough to skip SFT and go straight into GRPO.

The instruction "End the response immediately after giving the answer" was added to suppress the model's tendency to continue generating after `</answer>`, which wastes time and can corrupt our per-token loss normalization process. The instruction "Do not write code, solve the problem manually" was necessary as, understandably, the base model often wrote code to solve the math problem. While this is clearly a better approach in reality, we don't want the model to take that shortcut for the purpose of this thesis.

Problem type	Base Model Accuracy	Formatting Error Rate
2×2-digit	89%	1%
3×3-digit	30%	16%
4×4-digit	0%	21%
5×5-digit	0%	34%
6×3-digit	1%	9%

Table 3.1: Baseline accuracy and formatting error rate of `Qwen2.5-1.5B-Instruct` under the prompt of Figure 3.1, with greedy decoding.

3.3 Evaluation Optimizations

Because we want to observe how the model learns over the course of training, we need to evaluate the model during the training process. In order to produce a sufficiently dense learning curve, we decided to target 50-60 evaluation checkpoints per 24-hour training run. Also, for these metrics to be useful for comparison, we need to evaluate a large sample of multiple different problem types. The evaluation framework we settled on requires answering 500 different questions, and generating them one-by-one without any optimizations would take us well over an hour. Clearly, we will need to optimize this process in order to complete this many evaluations while leaving the vast majority of our 24-hours for training the model. We will discuss some of these optimizations, along with our other key decisions for the evaluation process below.

Evaluation Dataset Selection

Each evaluation consists of five slices of 100 problems each at 2×2, 3×3, 4×4, 5×5, and 6×3 digits. The same random seed draws the 500-problem evaluation set for every evaluation in every run for Models 1 through 4, so accuracy at any step in any run is directly comparable to accuracy at any other step in any other run. In contrast to our training set, which consists

of all sixteen combinations of 2-through-5 digit multiplications, we only evaluate on the four diagonal slices; this is done in order to control evaluation cost while spanning the full training-difficulty range. We also evaluate on a fifth slice, 6×3, which is strictly out of distribution: no six-digit operand ever appears during training, so 6×3 acts as a generalization check. A model whose 6×3 accuracy tracks its in-distribution accuracy has learned a decomposition procedure that transfers, while a model that improves in-distribution but stagnates on 6×3 has overfit to the specific digit-count grid it was trained on. Thankfully, as we will see in Chapter 4, the 6×3 accuracy keeps pace with the others for all of our models.

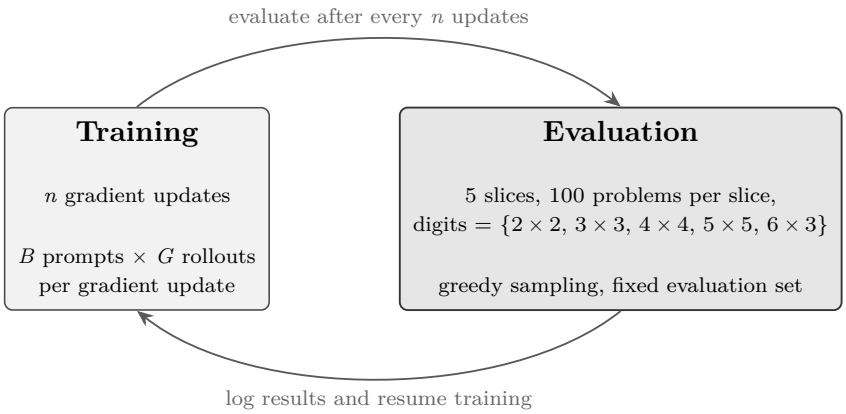
Parallel Greedy Rollouts

Rather than evaluating each problem individually, for each slice we tokenize the 100 problems together and dispatch them to `model.generate` in a single batch; with a 1024-token budget and greedy decoding, a full 500-problem evaluation now takes only a few minutes on the A100, with no decrease in response quality. We also set `pad_token_id` equal to `eos_token_id` so that padding masks reduce to a single-token equality check, and we approximate the per-response "thinking token" count (used in Chapter 5) by simply counting non-EOS tokens, which avoids a slow re-tokenization pass.

Greedy Decoding for Evaluation, Non-Greedy for Training

During training, each prompt is expanded into eight rollouts sampled at temperature 1.2 with top-$p = 0.95$: GRPO's learning signal comes from within-group reward variation, so obviously we need some variation in the model's responses — greedy decoding would make all eight rollouts identical. However, at evaluation time, we believe greedy decoding is the better option, since we are measuring the model rather than learning from it. Greedy decoding gives deterministic, reproducible pass@1 metrics, avoiding both the cost of averaging many samples per problem and the several-percentage-point sampling noise that non-greedy evaluation

would introduce. Anyone who has worked extensively with small LLMs may have doubts about this approach, as they will know that greedy decoding can often cause small models to get "stuck" in a loop, repeating the same string of tokens over and over again until the token limit is reached. We note that this did happen occasionally, but mostly early on in training; by the end, it was exceedingly rare, and thus the benefits of greedy decoding outweigh this risk.



One iteration of training and evaluation; one full 24*-hour Colab run contains* 50–60 *such cycles.*

Figure 3.2: Training and evaluation loop in our GRPO workflow.

CHAPTER 4

Model Training and Comparisons

This chapter presents the four primary models that we tested as the core component of this thesis. Model 1 is a baseline GRPO run using the algorithm as described in Chapter 2, with the single token-level aggregation modification of Section 2.2. Models 2, 3, and 4 each modify this baseline in some specific way, while everything else is held constant. This allows us to attribute any difference in learning curves to the specific intervention being tested.

4.1 Universal Hyperparameters

Before covering the four individual models, we first introduce and justify the hyperparameters that are held constant throughout all four runs.

LoRA Adapter

All four models are trained as LoRA adapters [5] on top of a frozen `Qwen2.5-1.5B-Instruct`, with the adapter attached to all seven projection matrices in every transformer block. We use rank $r = 128$, scaling factor $\alpha = 32$, and dropout 0.05. This rank is higher than most public GRPO implementations use at this scale; we chose it to give the adapter sufficient capacity to shift digit-token outputs (the tokenizer maps each digit 0–9 to its own token), and our VRAM budget was able to accommodate it. The result is 147,718,144 trainable parameters, or roughly 9.6% of the size of the base model. The base weights remain frozen throughout, which lets us recover the reference policy π_{ref} used in the KL penalty by simply

disabling the adapter at inference time.

Optimization

For our optimizer, we chose AdamW with $\beta_1 = 0.9$, $\beta_2 = 0.999$, zero weight decay, and a constant learning rate of 8×10^{-5}. Gradients are clipped to a maximum norm of 1.0.

Rollouts

Each GRPO step draws a single prompt and expands it into a group of $G = 8$ rollouts, the largest group size we could sustain at the full 1024-token generation length. Generation during training uses non-greedy sampling at temperature 1.2 with top-$p = 0.95$, for the reasons discussed in Section 3.3. The 1024-token ceiling should be enough for the model to complete a 5×5-digit decomposition (though as we'll discuss in Section 4.5 and Section 5.1, this upper bound may limit our model's creativity to some extent).

Rewards

Binary: $+1$ if the extracted answer equals the ground truth, -1 otherwise. A response with a missing `<answer>` tag or a non-integer answer counts as incorrect. We briefly experimented with a separate additional format-error penalty, but since the model quickly learns to format properly we found this to be unnecessary (and actually decreased early-training stability). Model 3 supplements this base reward function with a DAPO-style overlong penalty (Section 4.4.1); however, the other three models use the raw binary reward.

GRPO objective

Except where otherwise noted, the objective is the token-level aggregated GRPO loss of Equation (2.5), with one PPO epoch per batch, a symmetric clipping range of $\varepsilon = 0.2$, and a KL coefficient of $\beta = 0.01$ against the base model.

Training distribution and seeding.

At each step a digit pair (d_1, d_2) is drawn from the sixteen-element grid $\{2, 3, 4, 5\}^2$, and the two operands are drawn uniformly from $[10^{d-1}, 10^d)$. Models 1, 2, and 3 sample (d_1, d_2) uniformly with the same random seed, so each model sees the same training problems in the same order. Model 4, however, attempts to optimize this uniform sampling approach (Section 4.5.1).

4.2 Model 1: Baseline GRPO

Model 1 is our baseline, reference run. The loss function is exactly the token-level aggregated GRPO objective of Equation (2.5), with the KL penalty of Equation (2.3) and no further modifications. The only model-specific choice to discuss for Model 1 is the batch size, or gradient accumulation count. We use a batch size of 1 (for VRAM purposes) and 32 gradient accumulation steps, giving us an effective batch size of 32 (which means a single optimizer update is taken after every 32 prompts, or equivalently after every $32 \times 8 = 256$ rollouts). Model 1 was trained for 11,840 steps, corresponding to 370 gradient updates, over the 24-hour Colab session.

4.2.1 Results

Problem type	**Base model accuracy**	**Model 1 final accuracy**
2×2-digit	89%	**100**%
3×3-digit	30%	**91**%
4×4-digit	0%	**72**%
5×5-digit	0%	**24**%
6×3-digit	1%	**64**%

Table 4.1: Final Accuracy, Model 1 (11,840 steps, 24 hours).

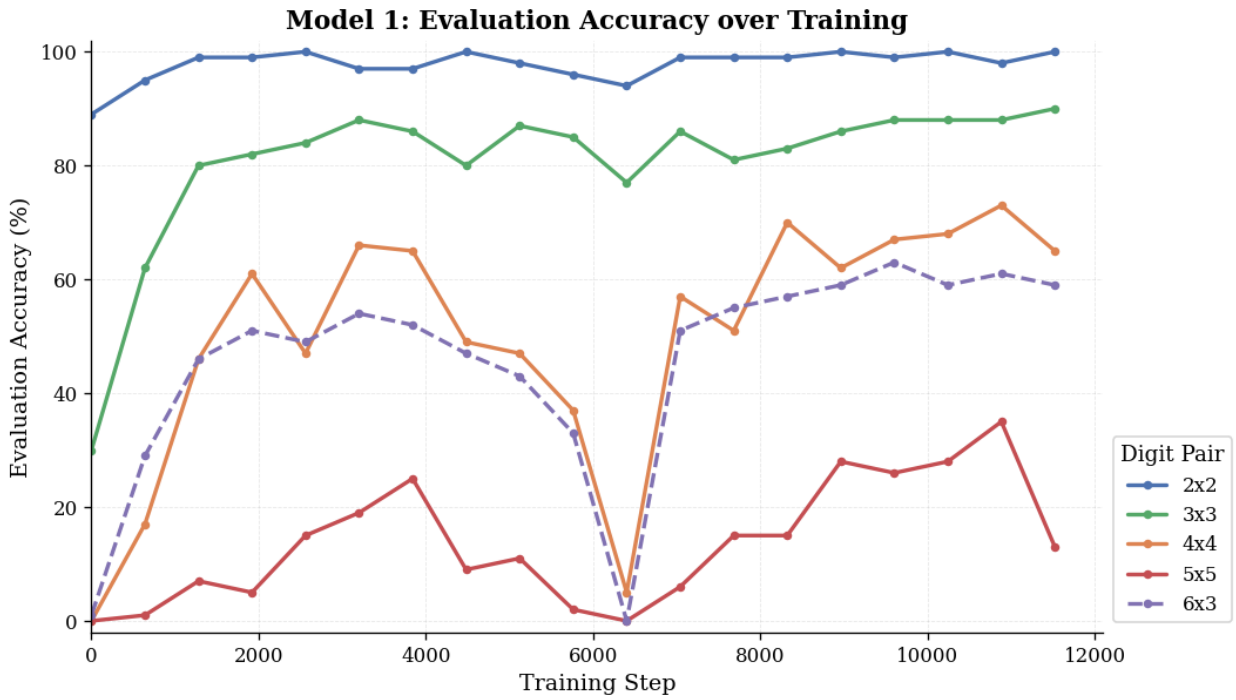


Figure 4.1: Model 1 evaluation accuracy over the course of training.

Overall, for a baseline run, it is fair to call this model a success. The model has learned how to work through many of these problems; it is excelling at the easier 2×2 and 3×3 problems, rapidly improving at the 4×4 problems (which the base model never solved correctly), and solving some of the hardest 5×5 problems.

However, the decomposition strategy that the model has discovered (an example of which is shown in Section 5.2) clearly struggles to generalize to these harder problems. Ideally, the model would be able to use its decomposition approach on problems of any size (as long as they fit within the 1024-token generation limit) and achieve a similar accuracy. Additionally, the training process struggles with stability at times, with the evaluation curves for 4×4, 5×5, and 6×3 all crashing to 0% over halfway through the training run. While the model eventually recovered from this, it is still something to keep an eye on; and, in fact, this crash was the primary motivator for the modification made for Model 2.

4.3 Model 2: Zero-Beta for Uninformative Questions

4.3.1 Theoretical Justification

We begin with a definition. Call a training prompt *informative* if the $G = 8$ rollouts sampled from that prompt produce at least one correct and at least one incorrect answer; call it *uninformative* otherwise. So, an uninformative question is one whose entire group of rollouts received the same reward — either all $+1$ (if every rollout was correct) or all -1 (if every rollout was incorrect). We found that such uninformative prompts dominate the two tails of training. Early in a run, the model has not yet learned to handle harder problems (with more digits), so these prompts tend to produce groups of eight uniformly incorrect responses. Later on, the model has learned to solve the easier problems (with fewer digits), so these prompts tend to produce groups of eight uniformly correct responses. Even during the middle stages of training it is possible that, purely due to chance, we will happen to select a number of uninformative questions in a row.

This situation is concerning because of what happens to the GRPO objective on an uninformative prompt. Recall the GRPO advantage from Equation (2.1):

$$A_i \;=\; \frac{r_i - \mathrm{mean}(\mathbf{r})}{\mathrm{std}(\mathbf{r}) + \varepsilon}. \tag{4.1}$$

If $r_1 = r_2 = ... = r_G$, then $\mathrm{r_i} = \mathrm{mean}(\mathbf{r})$, and $A_i = 0$ for every i. Plugging $A_i = 0$ into the policy-gradient loss of Equation (2.5) collapses the entire sum to zero:

$$\mathcal{L}_{\mathrm{PG}}^{\mathrm{tok}}(\theta) \;=\; 0 \qquad \text{(when prompt is uninformative).} \tag{4.2}$$

The full GRPO objective of Equation (2.3) then reduces to:

$$\mathcal{L}_{\mathrm{GRPO}}(\theta) \;=\; \mathcal{L}_{\mathrm{PG}}^{\mathrm{tok}}(\theta) + \beta \cdot D_{\mathrm{KL}}(\pi_\theta \,\|\, \pi_{\mathrm{ref}}) \;=\; \beta \cdot D_{\mathrm{KL}}(\pi_\theta \,\|\, \pi_{\mathrm{ref}})\,, \tag{4.3}$$

and its gradient is simply the scaled KL-divergence term:

$$\nabla_\theta \mathcal{L}_{\mathrm{GRPO}}(\theta) \;=\; \beta \cdot \nabla_\theta D_{\mathrm{KL}}(\pi_\theta \,\|\, \pi_{\mathrm{ref}})\,. \tag{4.4}$$

The KL divergence between the trained policy and the reference policy is minimized when $\pi_\theta = \pi_{\text{ref}}$; this means that its gradient always points in the direction that makes the trained policy more like the reference. Therefore, an optimizer step taken on an uninformative prompt does one thing and one thing only: it pulls the policy back toward the base model.

If uninformative questions were rare, this would not be much of a concern; if anything, we're simply regularizing slightly more than usual. However, uninformative questions are not rare. For Model 1, 58.7% (or 6,952 of the 11,840) prompts observed during training were uninformative. This means that on average we received an advantage-based signal for less than half of the 32 questions in each batch, while the KL-penalty signal was given by all 32 questions. Our goal is for the KL-term to act as a regularizer, but in batches filled with primarily uninformative questions it is becoming more of a dominant force; in the limit where every prompt in a batch is uninformative, the entire gradient update is a pure regression toward π_{ref}, and the model loses ground on whatever it had learned during the preceding informative batches. Our hypothesis is that this dynamic is at least partially responsible for the mid-training instability observed in Model 1's learning curves (Figure 4.1), where a stretch of accumulated uninformative steps undoes enough prior progress to knock 4×4, 5×5, and 6×3 accuracy back to zero.

The intervention we propose to fix this problem is quite simple. When a prompt is detected to be uninformative, we set the KL coefficient β to zero for that prompt only. The prompt still counts toward the gradient-accumulation batch size (so the batch still consists of 32 prompts regardless of how many are informative), but its contribution to the accumulated gradient is exactly zero. Under this rule, the loss simplifies as

$$\mathcal{L}_{\text{GRPO}}^{\beta=0}(\theta) \;=\; \begin{cases} \mathcal{L}_{\text{PG}}^{\text{tok}}(\theta) + \beta \cdot D_{\text{KL}}(\pi_\theta \,\|\, \pi_{\text{ref}}) & \text{if prompt is informative,} \\ 0 & \text{if prompt is uninformative.} \end{cases} \tag{4.5}$$

To be clear, we are not removing the KL term entirely; informative prompts still pay the full KL penalty. Also, it is worth mentioning that this has a computational side benefit, as

the reference-policy log-probabilities required to estimate D_{KL} are no longer calculated for uninformative questions.

4.3.2 Results

Problem type	Model 1 final accuracy	Model 2 final accuracy
2×2-digit	100%	**100%**
3×3-digit	91%	**96%**
4×4-digit	72%	**89%**
5×5-digit	24%	**60%**
6×3-digit	64%	**78%**

Table 4.2: Final Accuracy, Model 2 (9,920 steps, 24 hours).

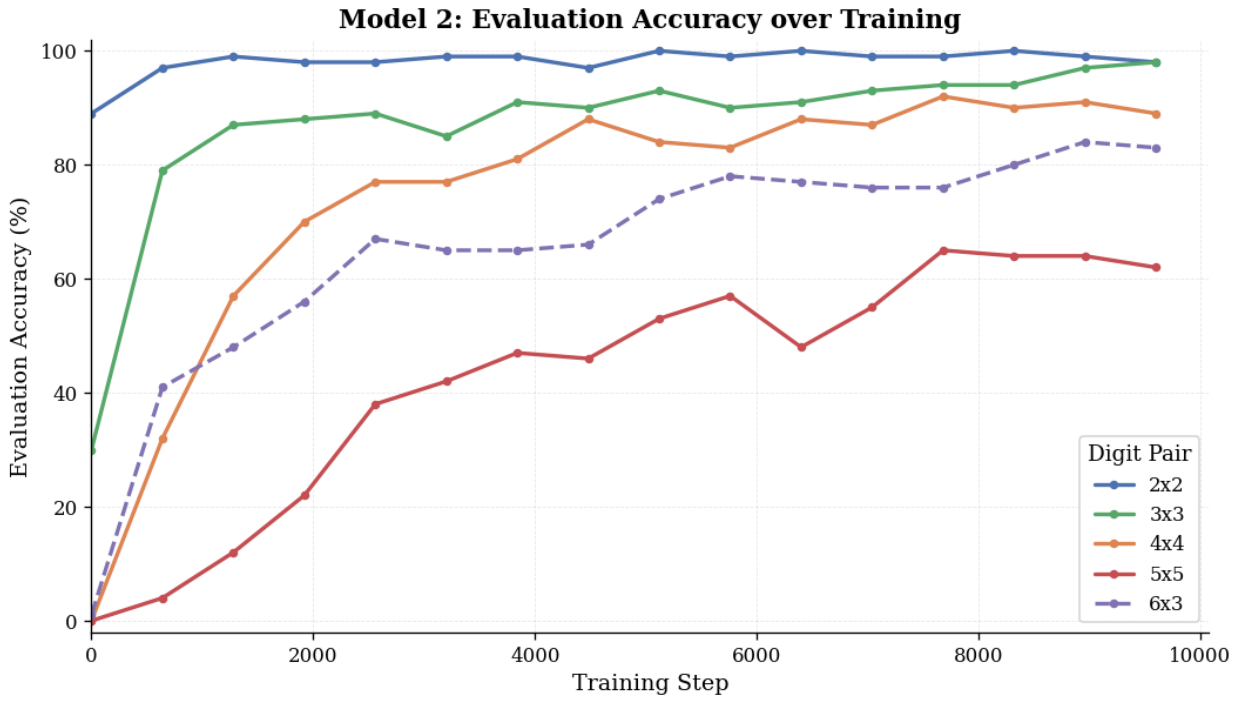


Figure 4.2: Model 2 evaluation accuracy over the course of training.

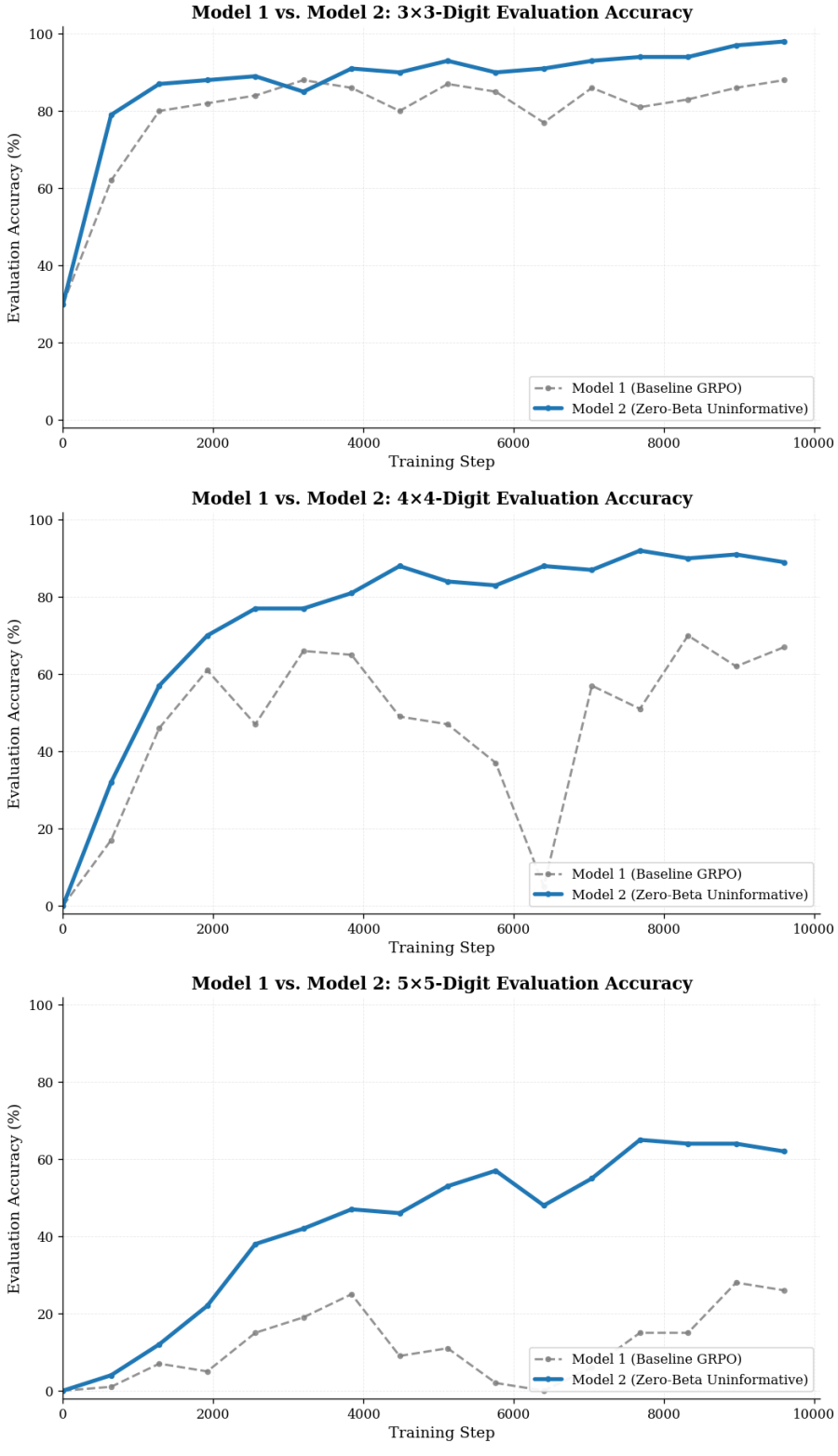


Figure 4.3: Comparison of Model 1 and Model 2 evaluation curves.

Model 2 sees a significant increase in both training effectiveness and training stability over Model 1. The final model is far more accurate across the board, solving the vast majority of 2×2 through 4×4 problems and more than half of 5×5. Additionally, the evaluation curves are far smoother; we rarely take significant steps backwards during training, and no mid-run crash like we observed in Model 1 occurred. Since everything else was held constant between Model 1 and Model 2, we believe that the zero-β adjustment for uninformative questions is responsible for the improvement. It appears that the full-strength KL-term was too restrictive on the model, and only after decreasing its influence were we able to maximize the amount of information gained from each informative question.

4.4 Model 3: Dynamic Sampling Policy Optimization (DAPO)

4.4.1 Modifications

Model 2's success suggests that the treatment of uninformative prompts is one of the most important (if not the most important) levers in small-model GRPO training with simple binary rewards. However, the realization that uninformative prompts may cause issues is not a novel one; this issue, along with numerous others, is addressed in ByteDance's "**D**ecoupled clip and **D**ynamic s**A**mpling **P**olicy **O**ptimization" (DAPO) paper [11]. DAPO is a multi-component modification of GRPO that was developed and validated on larger-scale reasoning models, and it contains four primary modifications. We will adopt all four in Model 3, adapted where necessary to the smaller model scale, with the goal of seeing whether these modifications are necessary to further improve performance.

Dynamic sampling.

Rather than simply zeroing the contribution of uninformative prompts (as in Model 2), DAPO discards them entirely and continues sampling until it has accumulated enough in-

formative prompts to perform an update. In theory, this will increase the stability of each gradient signal, as each batch will contain the exact same number of informative prompts. For Model 2, in contrast, we may experience a run of uninformative questions and perform an update after only one or two informative questions have been seen. That signal will be appropriately dampened by all of the uninformative zeros, as we take the mean loss across the batch, but it could feasibly increase the risk of training instability.

For Model 3, we adopt this dynamic sampling approach, with a gradient-accumulation target of 8 informative prompts per update. We tried numerous different targets, starting with the equivalent value of 32, but these parameters led to exceedingly slow learning in the early stages of training. We will justify and discuss this choice further in Section 4.4.2.

Clip-higher.

DAPO argues that the symmetric PPO clipping range $[1-\varepsilon, 1+\varepsilon]$ under-rewards positive-advantage updates on low-probability tokens, since the upper bound $1+\varepsilon$ cuts off exactly the ratios that represent the largest policy improvements. The remedy is to decouple the bounds, using a smaller ε_{low} for downside stability and a larger $\varepsilon_{\text{high}}$ for more aggressive upside updates. We adopt $\varepsilon_{\text{low}} = 0.2$ and $\varepsilon_{\text{high}} = 0.28$, taken directly from the DAPO paper, giving the asymmetric clipped ratio

$$\text{clip}\Big(w_{i,t}(\theta),\, 1-\varepsilon_{\text{low}},\, 1+\varepsilon_{\text{high}}\Big). \tag{4.6}$$

Overlong reward shaping.

In early stages of training, rollouts that hit the 1024-token ceiling are almost always lost in unproductive reasoning, so they are both unlikely to be correct and wasteful of generation budget. Later on, extremely verbose decomposition strategies may work for smaller problems, but would exceed 1024 tokens for larger ones, which could confuse the model. To encourage the model to answer the question both correctly and efficiently, DAPO introduces

a soft penalty that begins before the hard ceiling and ramps linearly to a small negative value at the ceiling. Adapted to our max-length of 1024 with a cache length of 124 tokens:

$$R_{\text{len}}(\ell) \;=\; \begin{cases} 0 & \text{if } \ell \le 900, \\ \dfrac{900-\ell}{2\cdot 124} & \text{if } 900 < \ell \le 1024, \\ -0.5 & \text{if } \ell > 1024, \end{cases} \tag{4.7}$$

where ℓ is the generated length in tokens. Added to the base ± 1 correctness reward, the total reward ranges from -1.5 (incorrect and truncated) to $+1$ (correct and well under the ceiling), giving a continuous gradient signal that discourages exceedingly long completions.

No KL penalty.

Finally, DAPO removes the KL-to-reference term from the objective altogether. Model 3's loss is

$$\mathcal{L}^{\text{tok}}_{\text{DAPO}}(\theta) \;=\; -\frac{1}{\sum_{i=1}^{G}|o_i|}\sum_{i=1}^{G}\sum_{t=1}^{|o_i|}\min\Big(w_{i,t}(\theta)\,A_{i,t},\; \text{clip}(w_{i,t}(\theta),\,1-\varepsilon_{\text{low}},\,1+\varepsilon_{\text{high}})\,A_{i,t}\Big), \tag{4.8}$$

with rewards shaped by Equation (4.7). So, no reference-policy log-probabilities are computed during training at all.

4.4.2 Results

Problem type	**Model 2 final accuracy**	**Model 3 final accuracy**
2×2-digit	100%	99%
3×3-digit	96%	93%
4×4-digit	89%	81%
5×5-digit	60%	50%
6×3-digit	78%	66%

Table 4.3: Final Accuracy, Model 3 (10,000 steps, 24 hours).

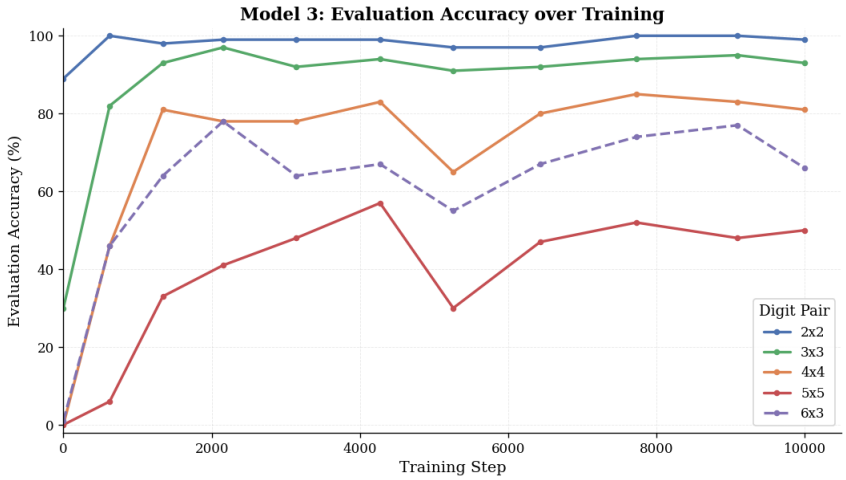


Figure 4.4: Model 3 evaluation accuracy over the course of training.

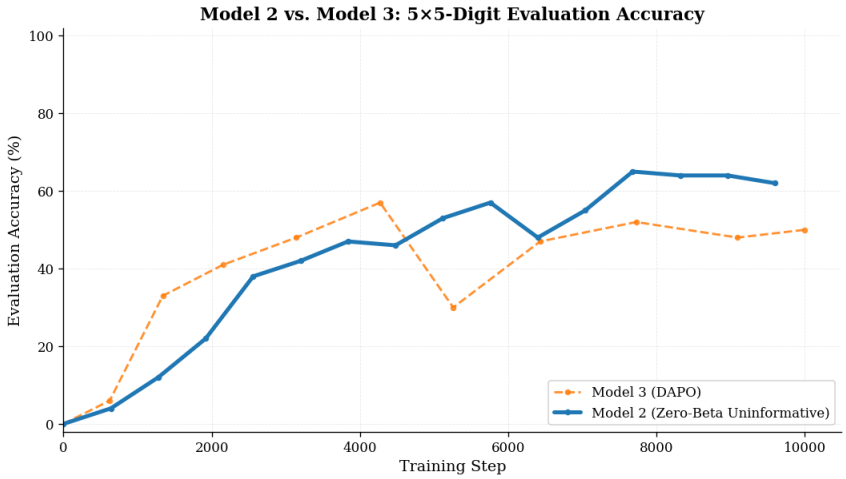


Figure 4.5: Comparison Model 2 and Model 3 evaluation curves.

Overall, the DAPO model fails to improve on Model 2, with slight declines in accuracy for all five evaluation slices. This suggests that, while these DAPO adjustments may be helpful for larger, generalized reasoning models, they may be unnecessary or even harmful for our circumstances.

We believe that the primary driver of this decrease in performance is the dynamic sampling approach. In the early stages of training, when most prompts are uninformative, it can

take a very long time to accumulate even one informative batch. Since Model 2 was able to learn rapidly during this phase, we believe that there is a lot of "easy" knowledge that can be obtained from only a few informative samples (like reinforcing the formatting rules and basic decomposition ideas), so batches that are almost entirely uninformative can still provide a useful gradient update. Initially, we attempted to run DAPO with an equivalent 32 informative prompts per batch, but updates were so infrequent that minimal learning was occurring and the evaluation curves remained flat. After trying numerous different parameter values, we settled on 8 informative prompts per batch as this was the only one that kept pace with (and even exceeded) the performance of Model 2 in the early stages of training; however, as we can see in Figure 4.5, this advantage was not maintained, and Model 3 soon fell behind Model 2. Regarding the cause of the instability and stagnation observed in these evaluation curves, we believe that the clip-higher adjustment may be the primary contributor, for a counter-intuitive reason. This will be discussed further in Section 5.1.

For now, we will conclude that, for our use case, this collection of DAPO modifications does not seem to help our model's performance. It is possible that a different selection of hyperparameters (such as a longer maximum generation length and a longer training time) may improve the performance of DAPO, but for this thesis we will move on with Model 2 as our current "best model".

4.5 Model 4: Bandit-Inspired Data Selection

So far, we have left our training distribution (uniformly sampling the digits from the sixteen-element grid $\{2, 3, 4, 5\}^2$) unchanged, while solely focusing on modifying what happens after a prompt has been sampled. However, we believe this may not be the most efficient approach.

4.5.1 Theoretical Justification

The argument for non-uniform sampling is a straightforward extension from the analysis of Section 4.3.1. At any given moment in training, the sixteen digit pairs do not contribute equal amounts of information to the gradient; some pairs may be too easy (leading to a lot of uninformative, all-correct rollouts), and some pairs may be too hard (leading to a lot of uninformative, all-incorrect rollouts). The informative pairs at any moment are those at the frontier of the model's current ability, where the rollout group has a diverse spread of outcomes. Intuitively, we can imagine that this frontier will move during training. Initially, most 2×2 prompts are still informative, while all harder questions almost always result in an accuracy of 0/8. Then, after a few gradient updates, 2×2 problems quickly become trivially easy, but 5×5 is still impossibly hard, and it is 3×3 that lives at the frontier; later on, 2×2 is still trivial and 3×3 is now easy, and it is 4×4 that is at the frontier, and so on.

A uniform sampler spends equal time on every pair regardless of where the frontier actually is, which means a great deal of time is wasted on predictably uninformative prompts. While the modifications of Model 2 have helped to minimize the damage that these uninformative prompts can cause, we still waste wall-clock time generating unnecessary responses, a problem which can only be solved by sampling fewer such prompts in the first place.

To improve on the uniform sampler, we take inspiration from the non-stationary multi-armed bandit [1], where we have one arm per digit pair and a reward function that is explicitly designed to measure informativeness. Let $n \in \{0, 1, \ldots, G\}$ denote the number of correct rollouts in the group produced by an arm on a given step. We define the arm's informativeness reward as

$$\rho(n) \;=\; 1 - \left|\frac{2n}{G} - 1\right|. \tag{4.9}$$

This function is a triangle that peaks at 1 when exactly half the rollouts are correct ($n = G/2 = 4$ in our setting) and falls linearly to 0 at the extremes $n = 0$ and $n = G$. It is, in other words, a direct numerical proxy for how close the arm is to the learning frontier

at that moment. An arm whose groups are reliably all-correct or all-incorrect receives zero reward; an arm whose groups are balanced receives a reward of 1.

Each arm a maintains an exponential moving average μ_a of its informativeness reward. After a prompt drawn from arm a produces a group with n correct rollouts, we update

$$\mu_a \leftarrow \alpha \cdot \rho(n) + (1 - \alpha) \cdot \mu_a, \tag{4.10}$$

with decay $1-\alpha$. Higher α makes the bandit more responsive to recent outcomes at the cost of higher variance; we found $\alpha = 0.2$ to be a reasonable middle ground for training runs on the order of ten thousand steps.

Given the current EMA values, the arm selection distribution is constructed in two steps. First, we compute a raw score-based distribution by normalizing:

$$\tilde{p}_a = \frac{\mu_a}{\sum_{a'} \mu_{a'}}. \tag{4.11}$$

Second, we mix this distribution with a uniform floor to preserve some exploration of arms that would otherwise become rare:

$$p_a = (1 - N \cdot \phi) \cdot \tilde{p}_a + \phi, \tag{4.12}$$

where $N = 16$ is the number of arms and $\phi = 0.02$ is the per-arm floor. This guarantees that every arm receives at least $\phi = 2\%$ selection probability regardless of how low its EMA has drifted. We still want to periodically sample 5×5 problems even if the model hasn't solved any of them yet, because eventually it will have learned enough from the easier problems to be able to do so, and we still want to sample 2×2 problems to ensure that the model isn't modifying its decomposition strategy in such a way that is incompatible with smaller problems. With $\phi = 2\%$, 32% of the sample space is effectively fixed and spread out uniformly among the 16 arms, while the remaining 68% is free for the bandit to distribute optimally.

Initialization and Burn-In

For initialization, there are multiple options; we could start with an uninformative prior ($\mu_a = 0.5$), we could estimate each μ_a through an extra evaluation step, or we could utilize a "burn-in" calibration phase in which we sample uniformly for a set number of steps, and then let the bandit take over.

Concretely, we implement this as follows. The first 960 training steps are run under the Model 2 training setup, so Model 2 and Model 4 share a checkpoint at step 960. During these shared steps, the bandit sampler runs in the background, updating its EMAs after every rollout group but not being used to select prompts. Model 4 then resumes from this checkpoint (inheriting the warmed-up EMAs) and switches to bandit-driven sampling from step 961 onwards, allowing for direct comparison between Models 2 and 4 from that point.

4.5.2 Results

Problem type	**Model 2 final accuracy**	**Model 4 final accuracy**
2×2-digit	100%	**100**%
3×3-digit	96%	**94**%
4×4-digit	89%	**92**%
5×5-digit	60%	**69**%
6×3-digit	78%	**80**%

Table 4.4: Final Accuracy, Model 4 (12,160 steps, 24 hours).

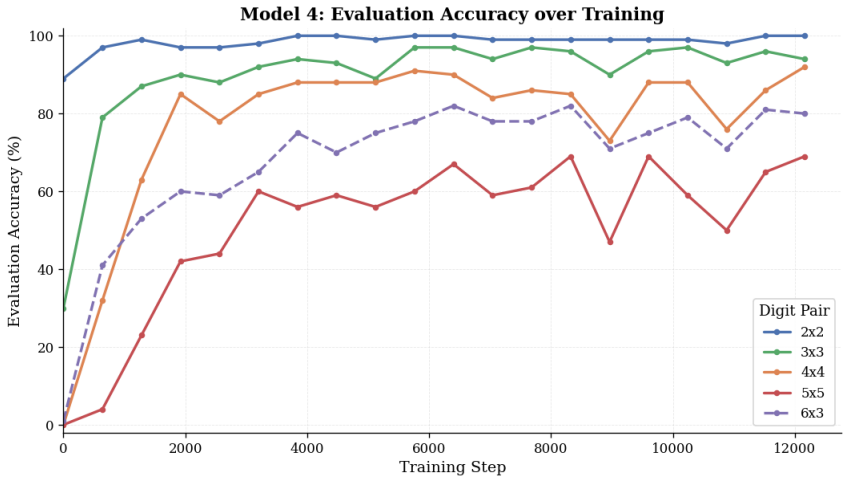


Figure 4.6: Model 4 evaluation accuracy over the course of training.

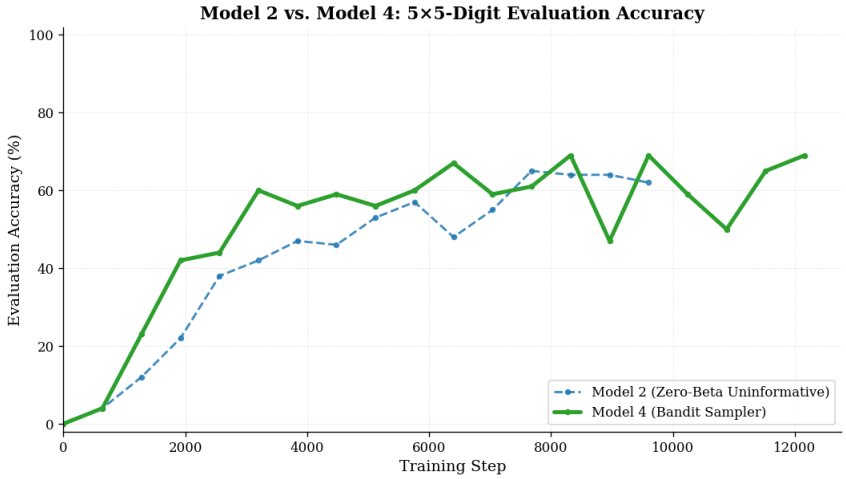


Figure 4.7: Comparison of Model 2 and Model 4.

We see a small but noticeable improvement from Model 2 to Model 4, particularly in the harder 5×5 problems. We see the largest improvement as soon as the bandit takes over in the early stages of training (Steps 1000 to 3000), where the model learns far more quickly; this can be attributed to the decrease in sampling of 4-digit and 5-digit questions which are far too difficult and will almost always be uninformative.

In addition to an increase in per-step efficiency, this also leads to a significant increase in wall-clock efficiency. When the model is presented with a question that is too difficult, it is likely that it will get lost in at least one of its answers, reasoning in circles until it eventually hits the max token limit. This means that this type of uninformative question also wastes a disproportionate amount of time, which we believe is the primary reason that Model 4 was able to see a larger number of questions than Model 2.

Now, we should examine the training process in more detail to ensure that the bandit is behaving as expected. First, of course, we should see an overall decrease in uninformative questions from Model 2 to Model 4. Then, we should examine the μ_a grid at different stages in training to confirm that the frontier (i.e. the arm with maximum μ_a) starts at 2×2 and then gradually moves over to 5×5.

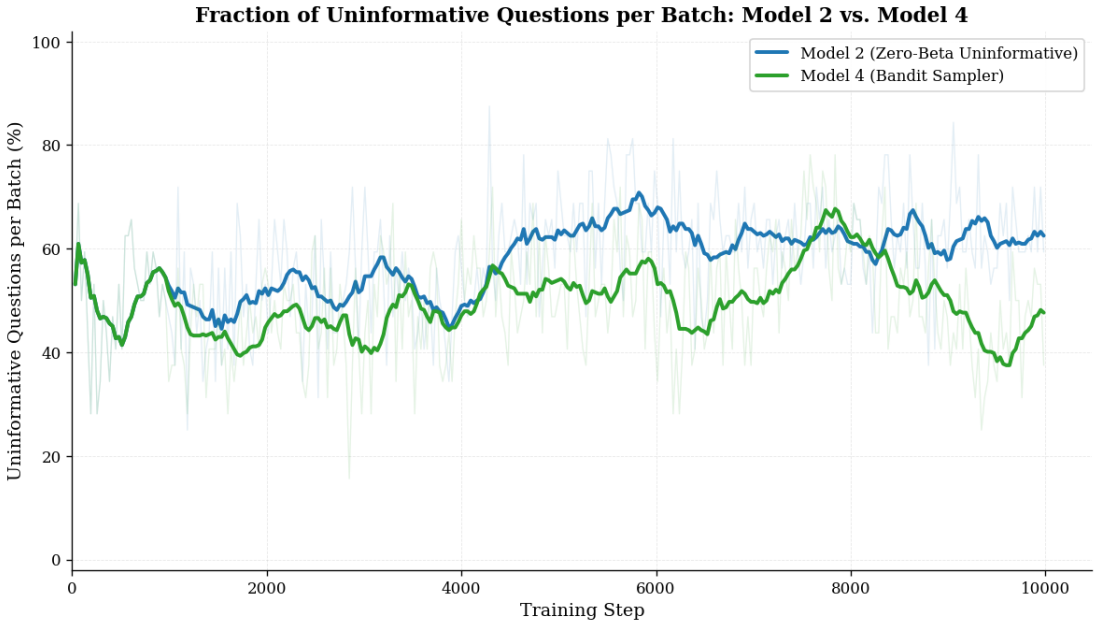
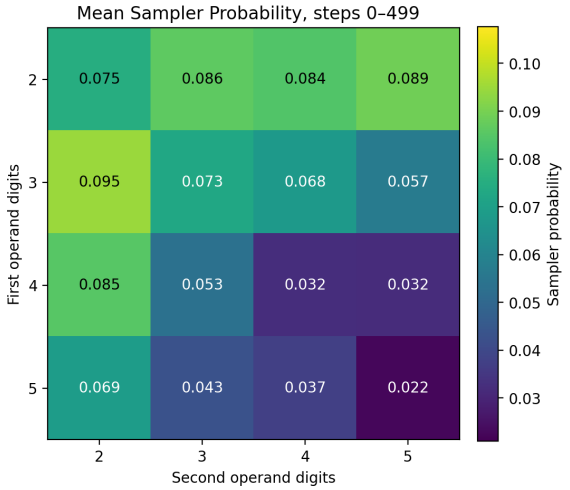


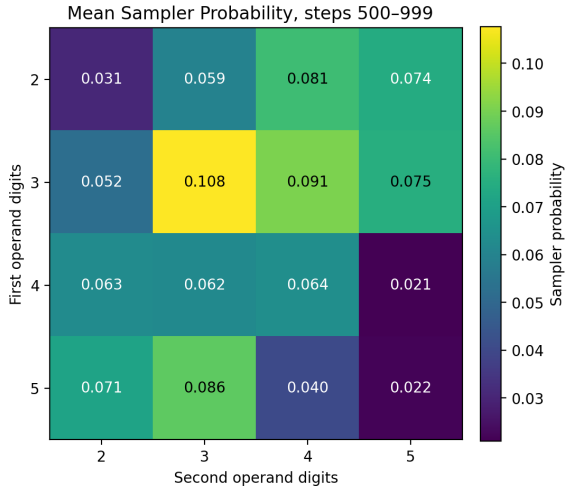
Figure 4.8: Uninformative question rate for Model 2 and Model 4.

Overall, the bandit does show an improvement in informative question rate. We should mention that both paths are subject to a lot of variance, as 32% of the sampling space is still spread uniformly, and there is still a significant amount of variation in difficulty for problems

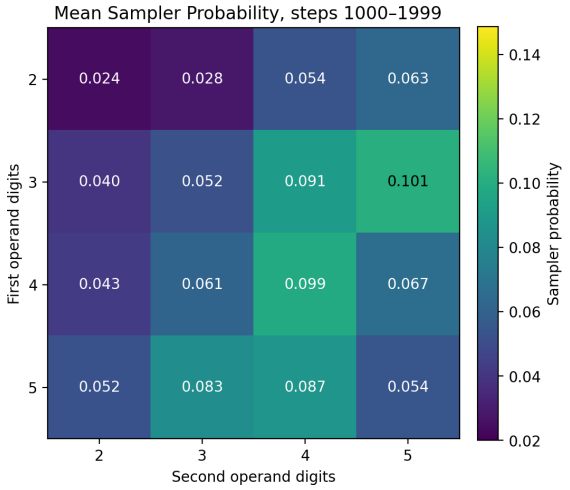
with the same number of digits (for example, a problem like 20400×11200 is far easier than most 5×5 problems, while our bandit has no control over this). However, it is clear that our selection of parameters for the bandit was on the conservative side, so it is possible that the improvement in efficiency would be even greater with a lower ϕ floor or a more aggressive reward function.



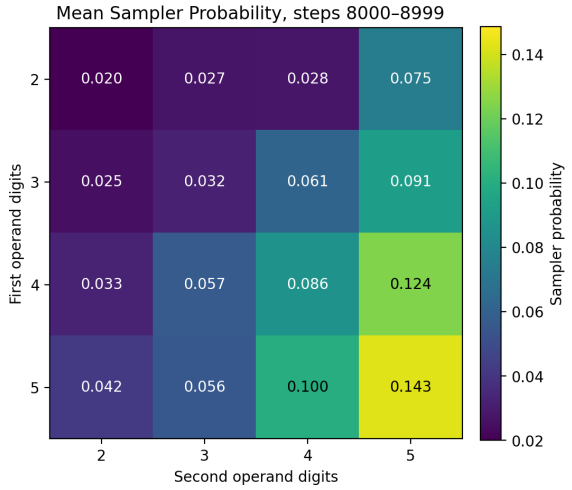
Steps 0–500: Frontier across 2-digit problems.



Steps 500–1000: Frontier at 3×3 problems.



Steps 1000–2000: Frontier around 4×4 problems.



Steps 8000–9000: Final frontier of 5×5 problems.

Figure 4.9: Mean bandit sampler probabilities over four regions of Model 4's training.

Here, we can clearly see the frontier move from the top-left (2×2) diagonally down to the bottom-right (5×5) as the model learns more and more. While the probabilities appear roughly symmetric across this diagonal line, we do see a nonzero difference between some opposing pairs. For example, the model is far more comfortable solving 5×2 problems than 2×5, which makes sense when we see the specific decomposition strategies used (see Section 5.2). Unfortunately, the model was never observed switching the first and second operands before decomposing; it is possible that this would eventually emerge after more training, or maybe if we expanded the sample space to include more digits and therefore more unbalanced problems.

4.5.3 Generalization to Other Problems

The bandit is particularly well-suited to our setup, where there are sixteen natural problem classes arranged along an obvious difficulty gradient. However, we must remember that our goal was not to develop a language model that is great at multiplication, but rather to develop and test techniques that can generalize to other training environments.

While it may not be quite as clean, we argue that this method can still apply to a more conventional post-training setup. The core requirement is only that the training distribution be partitionable into a finite set of sub-distributions that can plausibly differ in informativeness at any given moment. This condition is met in many practical scenarios. A multi-task fine-tuning run might draw problems from several distinct datasets — say, a grade-school math dataset, a programming dataset, and a reading-comprehension dataset — each of which a given base model finds easy or hard to a different degree. Early in training, the easier dataset may be the only one producing informative groups, and spending the training budget on the hard datasets is wasted; later, the easy dataset may be solved reliably and the hard ones become the frontier. A bandit over datasets, with each dataset as an arm and Equation (4.9) as the informativeness reward, would automatically shift sampling weight from the easy dataset to the hard ones as training progresses, without requiring any

hand-tuned curriculum schedule.

Since there is effectively zero additional cost with our bandit sampler, we see no reason not to use it (as long as one is willing to spend a bit of effort tuning the parameters and reward function). We therefore view the bandit sampler as a drop-in replacement for uniform sampling in any GRPO training setup where the prompt distribution is naturally partitioned.

4.6 Final Model Comparison

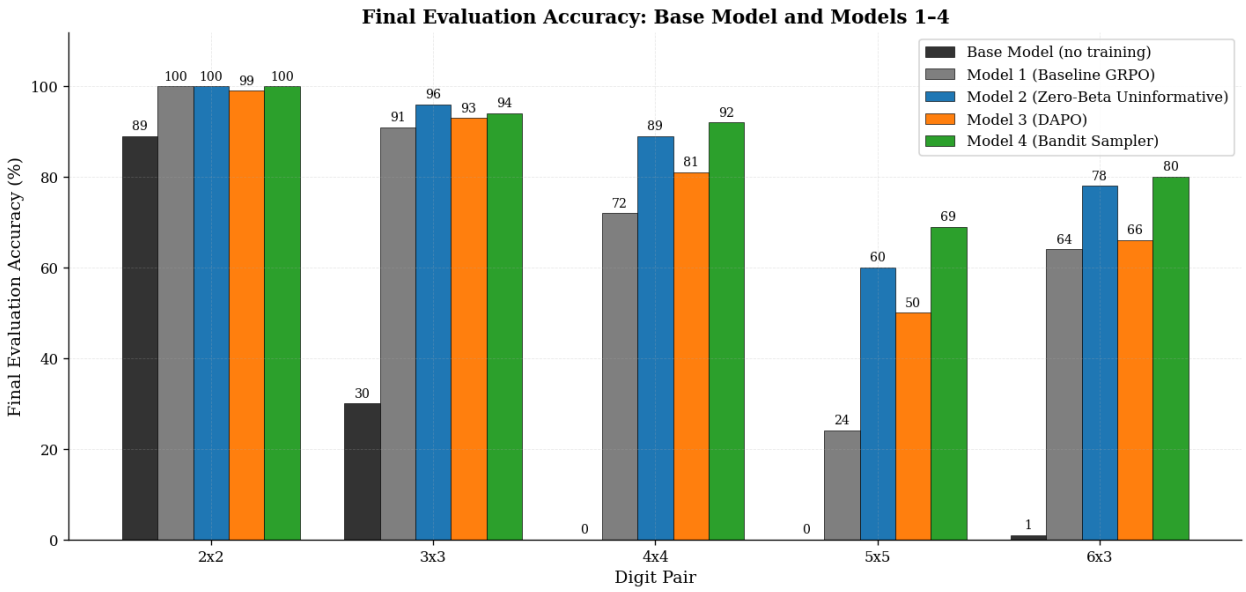


Figure 4.10: Pass@1 greedy accuracy at the final training step for all four models.

Here, we clearly see that all four models significantly improve on the base model, and Model 4 consistently scores the highest (especially on the most challenging problems).

CHAPTER 5

Signs of Reasoning Behavior in Trained Models

We have now examined the training process and evaluation metrics for five different models (including the base model). However, simply looking at the model's accuracy over time doesn't tell us *how* the model is decomposing these problems, and how each model's thinking process is different. For that, we must examine the model outputs themselves.

5.1 Increase in Thinking Time

First, we should look at one more metric: how long does each model think before returning an answer? One of the key discoveries from DeepSeek-R1 [4] was that the model's response length increased linearly throughout training, with no signs of slowing down; ideally we will see an increase in response length as well.

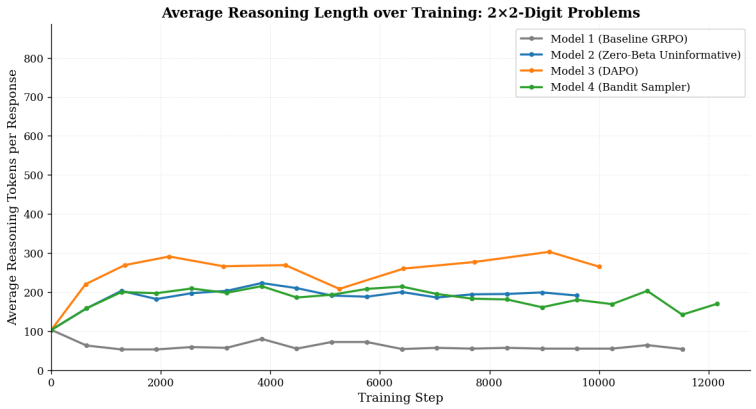


Figure 5.1: Thinking time (in tokens) for 2×2 problems (evaluation dataset)

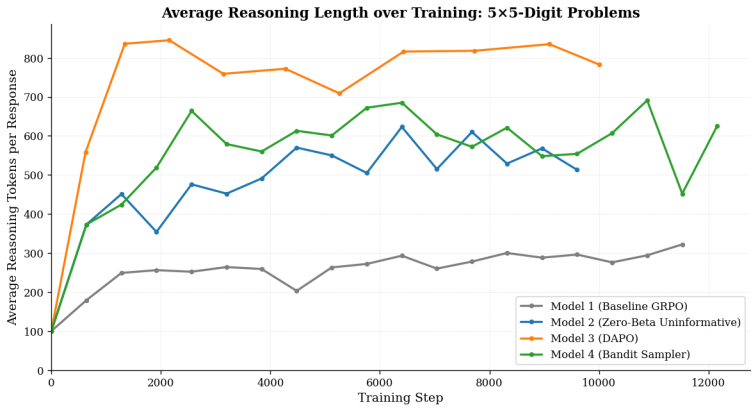


Figure 5.2: Thinking time (in tokens) for 5×5 problems (evaluation dataset)

Here, we see a similar result, particularly in the harder problems with more digits. This makes sense, as harder problems require a far more complex decomposition process.

The first key observation is that Model 1 thinks for fewer tokens than the other models, particularly on harder problems; we attribute this to the full-strength KL-term preventing Model 1 from deviating too far from the base model.

We also notice that Model 3 initially thinks for longer than the other models, quickly exceeding 800 tokens on average for 5×5 problems before leveling off. We believe this is primarily caused by the clip-higher DAPO adjustment, which allows the model to change more quickly. Ordinarily, this may be a good thing; it is possible that the model was developing a decomposition strategy that was superior to those of Model 2 and Model 4, but it was cut off by the max-token limit of 1024 and the overlong reward function was unable to help the model get back on track. Still, this realization shows that the DAPO adjustments should not be fully discounted, especially in training environments where the model is free to think for longer.

5.2 Sample Model Output Comparison

Now, we will compare the outputs from one of our 5×5 evaluation questions:

User Prompt: 5x5 Eval Question #6

```
What is 27953 * 68959?

Think through the problem between <think> and </think> tags, and return your
final numerical answer between <answer> and </answer> tags.

For example, 12 * 2 = <think> 12 * 2 = 10 * 2 + 2 * 2 = 20 + 4 = 24 </think>
<answer>24</answer>

End the response immediately after giving the answer.

Do not write code, solve the problem manually.
```

First, let's see what the base Qwen2.5-1.5B can do:

Base Qwen2.5-1.5B-Instruct: 5x5 Eval Question #6

<think> To solve this multiplication problem, we can use the distributive property of multiplication over addition. We will break down the numbers into parts that are easier to multiply.</think>

<answer>200000 + 100000 + 10000 + 1000 + 2000 + 1000 + 100 + 100 + 900 + 800 + 700 + 600 + 500 + 400 + 300 + 200 + 100 + 90 + 80 + 70 + 60 + 50 + 40 + 30 + 20 + 19 + 18 + 17 + 16 + 15 + 14 + 13 + 12 + 11 + 10 + 9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1</answer>

Correct Answer: 1,927,610,927

Model Incorrect, Formatting Error (No Valid Answer Given)

Despite all of the mathematical data that `Qwen2.5-1.5B-Instruct` saw during its training process, it is clearly not capable of even attempting to solve these types of problems.

While correct tags are used, the model "thinks" in the answer section, and this "thinking" is nonsensical. This base model still answered most 2×2 and some 3×3 questions correctly, but it simply loses control when presented with more challenging questions.

Model 1, Final Checkpoint: 5x5 Eval Question #6

<think> To solve 27953 * 68959, I can use the distributive property of multiplication over addition:

27953 * 68959 = (27953 * (60000 + 8000 + 900 + 50 + 9))

= (27953 * 60000) + (27953 * 8000) + (27953 * 900) + (27953 * 50) + (27953 * 9)

= 1677180000 + 223624000 + 24157700 + 1397650 + 241577

= 1677180000 + 223624000 + 24157700 + 1397650 + 241577

= 1900804000 + 24157700 + 1397650 + 241577

= 1924961700 + 1397650 + 241577

= 1926359350 + 241577

= 1926359350 + 241577

= 1926600927

</think> <answer>**1926600927**</answer>

Correct Answer: 1,927,610,927

Model Incorrect

Now, Model 1 is clearly a significant improvement; the solution is logically organized, and a relatively close answer is given. The model does attempt to use the distributive property to simplify the problem, using the exact framework seen in the example from the prompt. However, the model makes multiple mistakes (27953×900 and 27953×9 are both incorrect), resulting in an incorrect final answer. It also repeats a few lines; maybe this is just a waste of tokens, or perhaps it has found that repeating itself may slightly decrease the error rate.

Additionally, like the base model, after the first sentence we exclusively get numbers and symbols in the chain of thought. This may be because our model is constrained by the full-strength KL-penalty, keeping it close to the base model and restraining its creativity.

Model 4, Final Checkpoint: 5x5 Eval Question #6

<think> To find 27953×68959, we can use the distributive property of multiplication over addition:

$$27953 \times 68959 = (27953 \times 68950) + (27953 \times 9).$$

First, calculate each part separately. $27953 \times 68950 = 27953 \times (70000 - 1050) = 27953 \times 70000 - 27953 \times 1050$.

Calculate 27953×70000:

$$27953 \times 70000 = 27953 \times (7 \times 10000) = 27953 \times 7 \times 10000 = 195671 \times 10000 = 1956710000.$$

Now compute 27953×1050:

$$27953 \times 1050 = 27953 \times (1000 + 50) = 27953 \times 1000 + 27953 \times 50 = 27953000 + 1397650 = 29350650.$$

So,

$$27953 \times 68950 = 1956710000 - 29350650 = 1927359350.$$

Next, do 27953×9.

$$27953 \times 9 = 27953 \times (10 - 1) = 27953 \times 10 - 27953 \times 1 = 279530 - 27953 = 251577.$$

Finally add both results together to get our total product:

$$27953 \times 68959 = 1927359350 + 251577 = 1927610927.$$

Therefore,

$$27953 \times 68959 = 1927610927.$$

</think> <answer>**1927610927**</answer>

Correct Answer: 1,927,610,927

Model Correct ✓

Model 4's output is quite different. First, you may notice that the answer is properly formatted within LaTeX, with both centered and in-line math mode. This has not been added after the fact; the model actually started doing this automatically. While there is no way to know for sure why this happened, we believe that it may have tapped into some of its prior mathematical training that included LaTeX-formatted data, or perhaps it simply found that this was a better way to organize the problem.

The model is also far more creative overall. Rather than always distributing by each digit in the second number like Model 1, it instead finds an "easier" way to decompose the problem. As we saw for Model 1, these models often struggle to multiply numbers with larger digits like 8 and 9. Rather than trying to get better at these more challenging sub-problems, Model 4 instead found a brilliant solution: don't even try. In this example, the model converts 68950 into $70000 - 1050$, and even converts 9 into $10 - 1$.

While our prompt includes a very simple decomposition example, we never instructed the model to use subtraction in this fashion. However, when no longer restrained by a disproportionately strong KL-divergence penalty, we found that the model was able to develop strategies like these to overcome its inherent weaknesses all on its own. In this case, these strategies allowed the model to reach the correct answer.

5.3 Additional Demonstrations of Model Creativity

Below, we will highlight two additional interesting generations from Model 4:

Model 4, Final Checkpoint: 5x5 Eval Question #5

<think> To find 27038×29999, we can use the distributive property of multiplication over addition:

$$27038 \times 29999 = (27038 \times (30000 - 1)) = 27038 \times 30000 - 27038 \times 1$$

Calculate each part separately:

$$27038 \times 30000 = 27038 \times (3 \times 10^4) = 27038 \times 3 \times 10^4 = 81114 \times 10^4 = 811140000$$

And, $27038 \times 1 = 27038$. Substituting these back in gives us:

$$27038 \times 29999 = 811140000 - 27038 = 811112962$$

Therefore, $27038 \times 29999 = 811112962$. </think> <answer>**811112962**</answer>

Correct Answer: 811,112,962

Model Correct ✓

First, the model finds an obvious simplification (replace 29999 with $30000 - 1$), which allows it to effectively treat this 5×5 problem like two 5×1 problems.

Then, we see an example of another interesting strategy that emerged: when many consecutive zeros are involved, the model sometimes used exponential notation (replacing 10000 with 10^4), which may be easier for the model to keep track of within the embedding space. Together, these shortcuts allow the model to reach the correct answer to a 5×5 problem in only a few hundred thinking tokens.

Model 4, Final Checkpoint: 5x5 Eval Question #85

<think> To find 25322×23089, we can use the distributive property of multiplication over addition:

$$25322 \times 23089 = (25322 \times (23000 + 89)) = 25322 \times 23000 + 25322 \times 89$$

First, calculate each part separately. $25322 \times 23000 = 25322 \times (20000 + 3000) = 25322 \times 20000 + 25322 \times 3000$
Calculate 25322×20000:

$$25322 \times 20000 = 506440000$$

Now compute 25322×3000:

$$25322 \times 3000 = 75966000$$

Adding these together gives us:

$$25322 \times 23000 = 506440000 + 75966000 = 582406000$$

Next, do the second part: 25322×89. We can use the distributive property again to simplify this calculation:

$$25322 \times 89 = 25322 \times (100 - 11) = 25322 \times 100 - 25322 \times 11$$

Calculate 25322×100:

$$25322 \times 100 = 2532200$$

And now, compute 25322×11:

$$25322 \times 11 = 25322 \times (10 + 1) = 25322 \times 10 + 25322 \times 1 = 253220 + 25322 = 278542$$

Subtracting them yields:

$$25322 \times 89 = 2532200 - 278542 = 2253658$$

Finally combining both results from above,

$$25322 \times 23089 = 582406000 + 2253658 = 584659658$$

Therefore,

$$25322 \times 23089 = 584659658$$

</think> <answer>**584659658**</answer>

Correct Answer: 584,659,658

Model Correct ✓

This is another example of how the model seeks out ”easier” multiplication; 25322×89,

which is a harder problem to figure out, is effectively replaced with $25322 \times (100 - 10 - 1)$. While this decomposition leads to multiple additional rounds of subtraction, the model clearly finds this process to be easier than solving multiple complex multiplication sub-problems. Again, after thoroughly walking through this decomposition, the model reaches the correct answer.

5.3.1 Complex Signs of Reasoning

One important thing to note is that we didn't see any "aha moments" [10] in training (where the model realized that it made a mistake during the chain of thought and backtracked, eventually arriving at the correct answer). It is possible that this would eventually emerge after more training time or if allowed to think for more tokens per problem, or perhaps models of this size are simply not capable of this level of reasoning.

Regardless, we believe that using multiple specialized small models to work through their own specific tasks is a better approach than trying to get one small model to be able to reason deeply and solve all sorts of different questions. If a given use case requires that level of generalization from one model, then relying on externally-hosted frontier models may be the only option.

CHAPTER 6

Does Single-Prompt Training Cause Overfitting?

Up until this point, every model has been trained against a single fixed prompt template (Figure 3.1), which is held constant (except for the two numbers `{a}` and `{b}`) for all 10,000-plus training steps. This raises a natural concern of overfitting; a model trained on exactly one prompt format may only be able to handle exactly one prompt format. It is possible that such a model may not be able to generalize its multiplication procedure if the prompt is organized differently, especially after seeing how sensitive the base `Qwen2.5-1.5B-Instruct` model was to different prompts (Section 3.2.1). Since prompt components like the sample decomposition affected baseline accuracy so significantly, we should know if our final models also require these lines in this exact order — if we modify or remove them, will accuracy plummet?

To investigate this possibility, we train an additional model with one key modification: we use a variety of different prompt formats, rather than only one, and then compare the generalization abilities of this final model with Model 4.

6.1 Model 5: Varied Prompts

The training apparatus for Model 5 is identical to Model 4, but instead of a single fixed prompt format, we will draw a prompt template uniformly at random from a pool of twenty different templates. Then, the operands `{a}` and `{b}` selected by the bandit are substituted into the selected template.

The twenty templates span several axes of variation. The worked example is changed, expanded, or entirely removed, and the instruction wording varies significantly. However, the structural skeleton of each prompt is still the same: every template instructs the model to produce reasoning between `<think>` and `</think>` tags and a final numerical answer between `<answer>` and `</answer>` tags, and every template instructs the model to solve the problem manually rather than by writing code.

6.2 Model 4 and Model 5 Comparison

For all three comparisons, we construct a single evaluation set of 2,500 problems (500 each from the 2×2, 3×3, 4×4, 5×5, and 6×3 digit slices) and we evaluate both Model 4 and Model 5 on this same set using the same evaluation protocol discussed in Section 3.2. We will then use this dataset for three different comparison tests:

1. All 2,500 problems use the original prompt format (Figure 3.1)
2. Each problem uses one of the twenty prompt formats from Model 5's training (selected at random)
3. All 2,500 problems use a brand new prompt format, never before seen by either model

This brand new prompt has been simplified as much as possible, with no worked example or additional instructions. All that remains is the `<think>` and `<answer>` tagging instructions and the multiplication problem itself:

New Prompt Template (User)

```
Find the product of {a} and {b}.
Place your working inside <think> ... </think>, and the final integer answer
inside <answer> ... </answer>.
```

If the prompt-overfitting hypothesis is correct, we should observe two patterns in the

comparison between Model 4 and Model 5. First, on prompts Model 5 has been trained on but Model 4 has not, Model 5 should outperform Model 4 substantially, as Model 4 will fail to generalize its learned decomposition strategy. Second, on a brand-new prompt that neither model has seen during training, Model 5 should still outperform Model 4, because Model 5's exposure to prompt diversity during training should have taught it how to better handle prompt variation. However, as we see below, the results do not support this hypothesis:

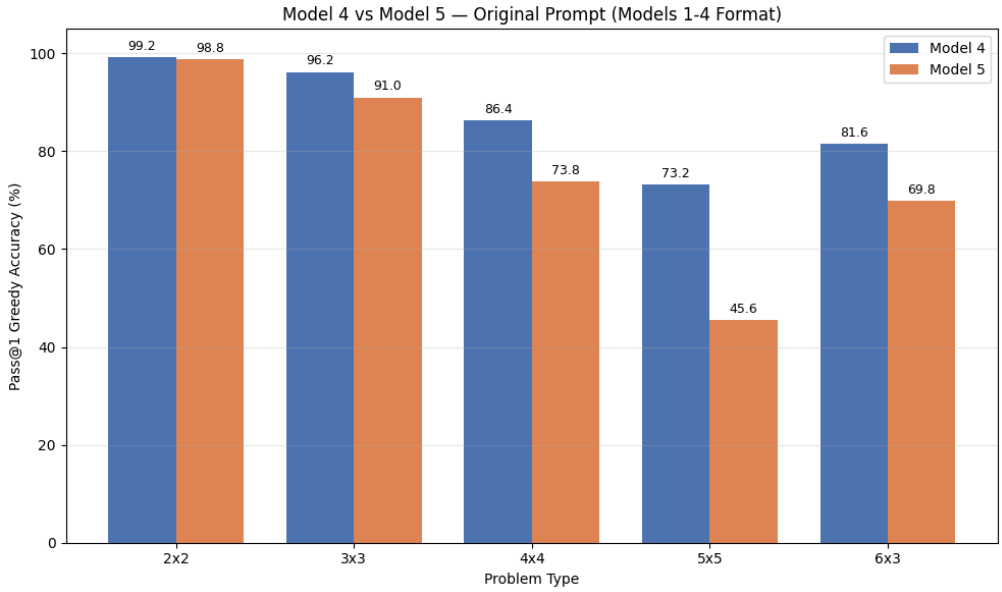


Figure 6.1: Pass@1 greedy accuracy of Models 4 and 5, using the original prompt format.

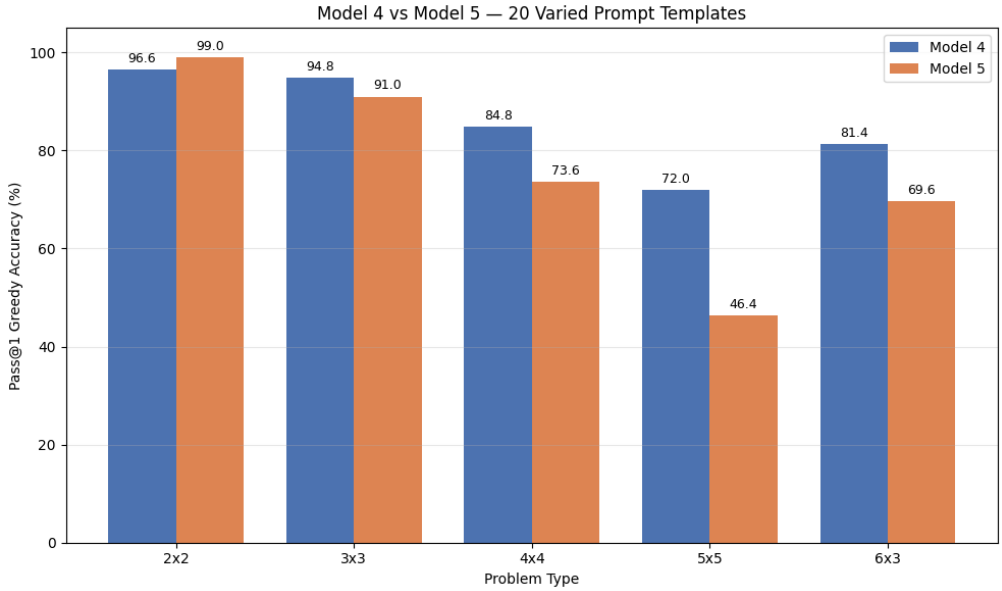


Figure 6.2: Pass@1 greedy accuracy of Models 4 and 5, using the twenty prompt formats from Model 5's training.

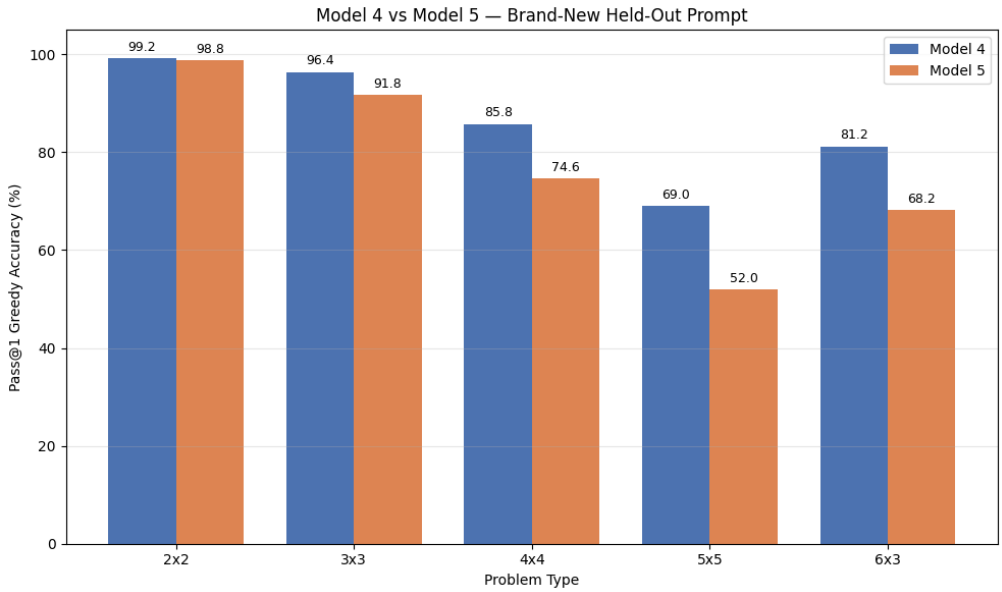


Figure 6.3: Pass@1 greedy accuracy of Models 4 and 5, using the new simplified prompt format.

Surprisingly, Model 4 outperforms Model 5 on nearly every digit slice in every test, including the test designed to favor Model 5 (Figure 6.2). Model 4's accuracy on the brand-new prompt is essentially indistinguishable from its accuracy on the prompt it was exclusively trained against, despite the new prompt lacking the worked example that the base model required to function at all. We take this as strong evidence that single-prompt training does not necessarily lead to overfitting and does not produce a brittle stimulus-response mapping; it produces a model that has internalized the underlying decomposition procedure well enough that the surface form of the request no longer matters as much. While this would explain why Model 4 performed well on all three tests, we should also address why Model 5 performed worse; we suggest that training Model 5 across twenty different templates simply introduced noise into the training process, causing the signal from each gradient update to be less clear. While the exact cause is unknown, we have clear evidence that modifying the prompt structure during the training process resulted in a final model that is no better at interpreting new prompt formats and less accurate overall.

We also should note that changing the prompt format did cause a slight decrease in the performance of Model 4; while this difference was negligible, it is possible that other, messier prompt formats could cause performance to drop even more. In that situation, we propose a simple solution inspired by our regular expression-based answer-extraction process discussed in Section 3.2. During inference, one could cheaply and easily pre-process the user's input before sending it to the model, converting it into something that more closely resembles the training prompt. This may require an additional LLM, or for repetitive and formulaic tasks like multiplication, a simple regex parser may be sufficient. This may require some additional overhead, but for a model that is actually being deployed, this prompt engineering step may be essential in order to keep real-world accuracy close to the accuracy observed in the training and evaluation environments.

CHAPTER 7

Conclusion

The intent for this thesis was to investigate whether a small, locally-runnable LLM could be trained with GRPO to reliably perform a repetitive, decomposable task without requiring SFT or frontier-scale compute. Using multi-digit multiplication as a controlled environment, we trained five models and compared their outcomes. After this training, we conclude that the answer is yes, at least for any use case that doesn't require the model to be 100% accurate on the first attempt. We also conclude that the standard GRPO recipe benefits from several modifications at this scale.

7.1 Summary of Findings

Our first and most important finding is that the KL-divergence term is the dominant training lever for small models with binary rewards. Zeroing β on uninformative prompts (Model 2) produced the single largest accuracy improvement in the thesis, lifting 5×5 accuracy from 24% to 60% and eliminating the mid-training crashes observed in the Model 1 baseline. Because uninformative prompts dominated more than half of all training rollouts, the unmodified KL penalty was effectively pulling the policy back toward the base model on the majority of gradient steps, drowning out the genuine learning signal from informative prompts.

Second, the bundle of modifications proposed by DAPO [11] did not transfer to our setting. Model 3 underperformed Model 2 on every evaluation slice. We do not interpret

this as evidence against DAPO in general, as it was developed and validated on substantially larger models with longer generation budgets, but rather as evidence that techniques designed for larger reasoning models cannot be exactly copied at the 1.5B scale, especially when training conditions (such as maximum output tokens) are more restrictive.

Third, our multi-armed bandit-inspired data sampler (Model 4) provided a small but real improvement over uniform sampling at essentially zero additional cost, with the largest gains on the hardest 5×5 problems. The technique generalizes naturally to any GRPO setup whose training distribution partitions into a finite set of sub-distributions.

Finally, single-prompt training did not produce the prompt-overfitting we initially feared. Model 4, trained against one fixed template, generalized cleanly to a new and simpler prompt it had never seen, while Model 5's prompt-diverse training produced a measurably worse model on every evaluation. We suspect that prompt format diversity during training injected noise into the gradient signal without providing a corresponding benefit.

7.2 Recommendations

For anyone tasked with setting up a similar GRPO training run involving a small LLM and an easily-verifiable binary reward, we recommend the following:

1. **Start from a base model with nonzero baseline accuracy.** If this is not achievable, then a preliminary SFT step (or something similar) is necessary. If the base model rarely produces a correct answer, almost every GRPO rollout is uninformative and minimal learning will occur.

2. **Zero the KL coefficient on uninformative prompts.** This simple change led to the largest improvement we observed. It both improves accuracy and stabilizes training.

3. **Use a bandit-style data sampler, if applicable.** It is worth implementing when

the training distribution partitions naturally, as the cost is minimal and training efficiency will likely improve.

4. **Train against a single prompt template.** If robustness to prompt variation is needed at deployment, handle it with a cheap regex- or LLM-based pre-processing step at inference time rather than by diversifying prompts during training.

7.3 Limitations and Future Work

One significant limitation of this work is the 1024-token generation ceiling, which almost certainly constrained the decomposition strategies our models could discover and may have unfairly handicapped the DAPO run in particular. At the same time, this limitation does exist in real-world scenarios; locally-hosted LLM inference can be slow, so for many practical applications we need the response to be both short and correct. Additionally, we don't know the upper limit of the performance of these models, as we were limited to 24-hour training sessions. It is possible that new and interesting behavior could emerge after more training, and the optimal hyperparameters for a much longer training run may be different.

We also did not observe any of the "aha moments" reported in larger reasoning models [10]; whether this reflects an inherent limit of the 1.5B scale or a consequence of the limited training time and 1024-token budget remains an open question. Finally, multiplication is a deliberately clean environment, and while we have argued that the techniques developed here should transfer to other repetitive and decomposable tasks, there are certainly a number of issues that could arise in the process.

Despite these limitations, we believe that developing a collection of specialized small models, each GRPO-trained on its own narrow task and capable of being run locally, will soon be superior to relying on expensive API calls to a single large generalist model for many real-world applications, and we believe that the techniques discussed in this work provide a promising starting point for their training.

REFERENCES

[1] Peter Auer, Nicolò Cesa-Bianchi, Yoav Freund, and Robert E. Schapire. The non-stochastic multiarmed bandit problem. *SIAM Journal on Computing*, 32(1):48–77, 2002.

[2] Peter Belcak, Greg Heinrich, Shizhe Diao, Yonggan Fu, Xin Dong, Saurav Muralidharan, Yingyan Celine Lin, and Pavlo Molchanov. Small language models are the future of agentic AI. *arXiv preprint arXiv:2506.02153*, 2025.

[3] Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. PAL: Program-aided language models. *arXiv preprint arXiv:2211.10435*, 2022.

[4] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, et al. DeepSeek-R1: Incentivizing reasoning capability in LLMs via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

[5] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, 2021.

[6] Long Ouyang, Jeffrey 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. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems*, 35:27730–27744, 2022.

[7] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[8] Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. DeepSeekMath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

[9] Qwen Team, An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tianyi Tang, Tingyu Xia, Xingzhang

Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. Qwen2.5 Technical Report. *arXiv preprint arXiv:2412.15115*, 2024.

[10] Shu Yang, Junchao Wu, Xin Chen, Yunze Xiao, Xinyi Yang, Derek F. Wong, and Di Wang. Understanding aha moments: From external observations to internal mechanisms. *arXiv preprint arXiv:2504.02956*, 2025.

[11] Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, Xin Liu, Haibin Lin, Zhiqi Lin, Bole Ma, Guangming Sheng, Yuxuan Tong, Chi Zhang, Mofan Zhang, Wang Zhang, Hang Zhu, Jinhua Zhu, Jiaze Chen, Jiangjie Chen, Chengyi Wang, Hongli Yu, Yuxuan Song, Xiangpeng Wei, Hao Zhou, Jingjing Liu, Wei-Ying Ma, Ya-Qin Zhang, Lin Yan, Mu Qiao, Yonghui Wu, and Mingxuan Wang. DAPO: An open-source LLM reinforcement learning system at scale. *arXiv preprint arXiv:2503.14476*, 2025.