Extended Abstract

Motivation Prior work has focused on the capability of language models (LMs) to solve Countdown, a puzzle where a target number has to be constructed by performing arithmetic operations on a set of given input numbers, in various ways and argued that search performance on Countdown can be generalized to more practical problems. However, to the best of our knowledge, no prior work has interfaced a classical Countdown solver with an LM where the solver acts as a tool and provides dense feedback to the LM during the search process.

Technique We take a preexisting solver for Countdown and interface it as a tool to our LM, Qwen-2.5 0.5B. To maintain the difficulty of the problem while obtaining actionable signal from the solver, we design the interface so that the solver is only able to respond in three ways: `GOOD`, `BAD`, or `END`, depending on whether the last operation chosen by the LM can lead to the target, cannot lead to the target, or has already reached the target, respectively.

We show that, with Chain of Thought (CoT) reasoning enabled, the model cannot be trained to generate well-formatted tool calls, while larger models were able to do so easily. However, we also show that by switching to a structured output format, tool calls can be successfully parsed and the model is able to learn from tool feedback. We describe our designs for both formats in detail.

Implementation We perform supervised finetuning (SFT) using our approach (with tool calling) and compare it to the same pretrained model with SFT on a different dataset meant for Countdown (without tool calling). Our evaluation shows that SFT on our training data augmented with tool calling significantly outperforms the other model.

Results To improve performance even more, we manually identify substitution errors as a common mistake made by our post-SFT model. In these cases, the model reached the target number but was unable to replace intermediate numbers in the answer expression by the operation that constructed them. For a completely correct answer, Countdown rules dictate that intermediate numbers do not occur in the answer. Partly to mitigate these errors, and partly to maximize LM performance, we experimented with Direct Preference Optimization (DPO) in several different configurations. All our experiments with DPO showed weak results. The only technique that resulted in a significant increase of performance over the SFT model was test-time compute (TTC) with best of N.

Discussion We hypothesize that DPO's ineffectiveness at mitigating substitution errors was due to the large amount of similarity in our preference pairs. In our setup, the chosen and rejected variants had the same trajectory leading up to the final answer. They only differed in the final answer itself: rejected variants had substitution errors whereas chosen variants did not. Other DPO configurations did not have this problem, but still yielded inconclusive results.

Conclusion We obtain three main results. One, that small LMs are unable to generate well-formatted tool calls with CoT enabled. Second, the same models can leverage tool calls if they are constrained to a structured output format. Lastly, that at least for Countdown, DPO is unable to meaningfully boost model performance over an SFT initialization.

Learning to Search with an Oracle: Finetuning for Countdown with a Classical Solver

Rupanshu Soi
Department of Computer Science
Stanford University
rsoi@stanford.edu

Masoud Charkhabi
Department of Computer Science
Stanford University
masoudc@stanford.edu

Abstract

Countdown has emerged as a fruitful problem for studying the capability of language models (LMs) to solve a reasoning problem. Countdown requires search, mainly because Countdown admits various strategies like backtracking and setting sub-goals, and a solution is easy to verify.

In fact, Countdown can be solved completely classically, by exhaustively searching the entire space. Open-source solvers for Countdown are readily available.

Based on this observation, we study whether a small LM (Qwen-2.5 0.5B) can be taught to solve Countdown by using such a solver as a tool. We find that with Chain of Thought (CoT) reasoning enabled, the LM is unable to consistently generate well-formatted tool calls. However, by representing search in a structured format, the LM is able to generate tool calls and can be taught to leverage tool feedback. By performing supervised finetuning using the structured format, the LM successfully solves 65% of Countdown problems in our experiments.

Furthermore, we study cases in which the LM is unable to find the solution and identify a kind of mistake it frequently makes. Partly to mitigate the occurrence of these mistakes, and partly to explore its potential to boost LM performance overall, we experiment with Direct Preference Optimization (DPO) with 4 different preference pair generation strategies. However, we find that DPO is unable to conclusively boost LM performance over the SFT initialization. We identify a concrete reason for this behavior for one of our strategies. In fact, we found test-time compute (TTC) to be the only technique that meaningfully increased LM performance over our SFT initialization. We conclude by discussing some potential reasons for DPO's ineffectiveness in our setting, and give directions for future work.

1 Introduction

Countdown is an old recreational mathematics problem where the goal is to construct, by means of the standard four arithmetic operations, a target number using a given set of input numbers. Each input number can be utilized at most once.

Countdown was introduced as a search problem for LMs by Gandhi et al. (2024). Their paper introduced a textual representation of the search process that could aid the LM in strategically reaching the target number. By a combination of pretraining and RL finetuning, they showed that LMs could achieve decent performance on Countdown problems with 3 or 4 input numbers.

Building on their work, we are interested in exploring whether an LM's search capabilities can be further enhanced by providing it access to expert guidance. In particular, we observe that Countdown is a problem that admits a simple, classical solution. For problems of small size, a brute force search over every possibility can be done in milliseconds. Therefore, we explore whether a small LM can be taught to effectively utilize feedback provided by such a solver.

A key design goal of our work is to ensure that our setup echoes more practical scenarios where the LM is performing a proof or program search equipped with expert guidance or human intuition. This guidance could, for example, allow the LM to prune parts of the space, or, encourage it to focus more on parts that are likely to contain the solution.

Therefore, the interface between the LM and the solver must be carefully designed. Since the solver can always just provide the final answer, we must restrict the feedback that can be obtained from it in such a way that maintains the difficulty of the original problem while containing useful signal about how to reach the solution.

In our design, every time the LM chooses the next operation, the solver—which is used via a tool call—responds in one of three possible ways:

- `GOOD`, if it is possible to reach the target after performing this operation
- `BAD`, if it is impossible to reach the target after performing this operation
- `END`, if this operation constructed the target

Note that the LM is still solely responsible for picking the operation and operands in each step, which can only use numbers that are available at that step, and constructing the final answer expression, which cannot use any numbers created in intermediate steps. Figure 1 depicts a full trajectory using our approach.

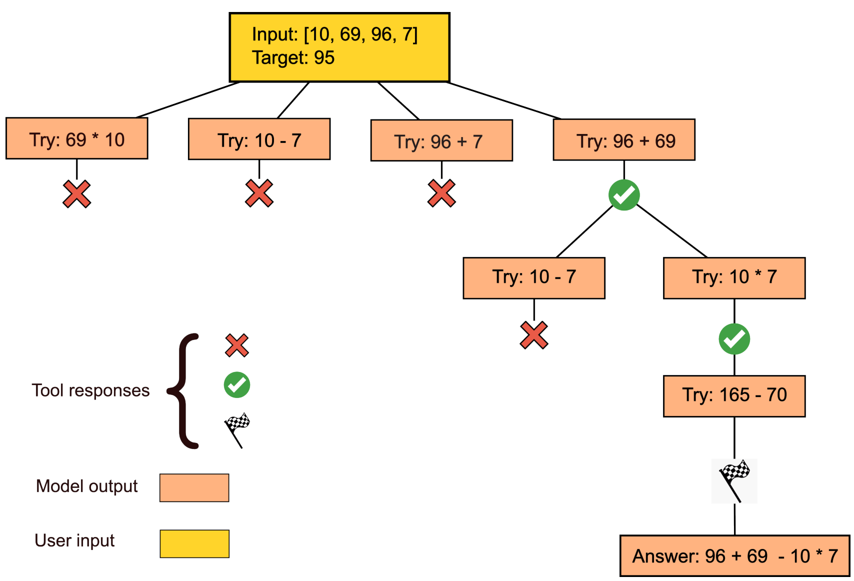

Figure 1: Visual depiction of our LM utilizing tool feedback to solve a Countdown problem. The search process is an abstraction of what the LM does to reach a solution, not a deterministic procedural search.

Our report is divided as follows. Section 2 briefly discusses the most relevant prior work. Section 3 presents our technique in more detail. We also present our designs that did not work, for others to learn from. Details of our experimental setup are discussed in Section 4. Section 5 presents the initial results, and discusses a common mistake that we observed in initial experiments. Section 6 discusses our efforts to mitigate those mistakes with DPO, discusses the inconclusive results we obtained with DPO, and presents results for test-time compute (TTC), the only technique that meaningfully increased performance over our SFT initialization. Section 7 concludes.

2 Related Work

Early efforts to teach language models (LMs) to search rather than merely predict optimal solutions introduced the Stream of Search (SoS) paradigm, in which the entire search trajectory is serialized as text and used for supervision (Gandhi et al., 2024). Subsequent work augmented SoS with supervision Moon et al. (2024).

A contemporaneous line of work is (Pan et al., 2025), called Adaptive Parallel Reasoning (APR), which lets an LM spawn multiple speculative reasoning branches and later join them, dynamically allocating inference-time compute to promising partial plans. APR demonstrates that modest models can benefit from explicit parallel exploration, but—like SoS—it still relies on the LM to evaluate every intermediate state without calling any external solver or verifier.

Both lines of research treat the LM itself as the sole search engine: the model proposes every intermediate move and judges its own progress from the textual state, without external computation. In contrast, our approach leverages a classical solver as a tool, providing dense reward at each step of the search process. To the best of our knowledge, no prior work on SoS or Countdown has interfaced an LM with a classical Countdown solver interactively.

3 Technique

We begin this section by describing our initial efforts in integrating CoT with tool calling, which relied on an unstructured textual format. After describing the challenges encountered with this approach, we move on to the structured format that is used in the final implementation.

3.1 Textual Format (Old)

A key design in our approach is the textual representation of the search used by the model while solving a problem.

Initially, we tried an unstructured format that allowed the model to output its chains of thought (CoTs) interspersed with tool calling. However, our model was unable to consistently generate well-formatted tool calls using this approach. This inconsistency could not be fixed even by using a custom, simpler tool calling format, and aggressive finetuning.

Figure 2 shows an example of CoT with an ill-formed tool call. The model hallucinated that the tool call format required XML tags, when in reality the standard format for Qwen requires JSON.

```
User: Input: [35, 77, 61]. Target: 19.

Assistant: Let's start by guessing an initial operation.
Given the numbers 35, 77, and 61, let's try adding two of them together.

<tools>
  <function>
    <name>ask_oracle</name>
    <parameters>
      <property name="operation">35 + 77</property>
    </parameters>
  </function>
</tools>
```

Figure 2: A tool call generated along with CoT. The call is completely misformatted.

We found this behavior to be quite variable as well. On subsequent runs, the model would remember to use JSON but would not be able to output well-formed JSON. At one point during development, we wrote custom parsing logic to attempt to handle every kind of mistake the model was making. The various cases we tried to handle can be seen in the regular expressions that were used at one point, shown in Figure 3. Still, the LM would manage to generate a tool call that would defeat our parsing logic.

```
patterns = [
    r"<tool_call>(.*?)</tool_call>",
    r"<tool_call>([^\n]*)",
    r"\n<tool_call>(.*?)</tool_call>",
    r"\n<tool_call>([^\n]*)",
    r"\n(.*?)<tool_call>",
    r"\n(.*?)\n",
]
```

Figure 3: Python code showing a list of regular expressions that were successively used to try to parse various kinds of misformatted tool calls. This approach is not used in the final design.

We tried the same approach with the 32B variant of Qwen-2.5 and found that it was easily able to generate tool calls with CoT, even without finetuning. We concluded that our 0.5B model was simply too small to consistently generate tool calls with CoT. We switched to a structured output format, similar to the Stream of Search paper, which we describe next.

3.2 Textual Format (New)

Before describing the format, let us discuss the high-level structure of each search trajectory.

Each trajectory is structured as a chat. The first message is a system prompt explaining the game of Countdown. The second message is from the user containing details about the problem. Starting from the third message, messages alternate between the assistant (containing the LM's choice of operation), and the solver (containing its feedback on the assistant's last message). Trajectories end if the LM outputs an answer in `<answer>...</answer>` tags or if the number of messages exceed a max limit.

Each LM response has three components:

1. Available: List of numbers available at the current step. In the beginning this is identical to the input numbers. The LM keeps updating this list as numbers are consumed and new ones are created. If this list is omitted (as it was in the initial design), the number of mistakes where the LM used an unavailable number was observed to sharply increase.
2. Right steps: List of operations (and the number created by them) that received `GOOD` from the solver. This list is meant to be used to construct the final answer by substituting in place of intermediate numbers the operation that created them.
3. Try: The next operation that the LM wishes to try.

As described in the introduction, each tool response can be either `GOOD`, `BAD`, or `END`. These responses are constructed by parsing the solver's actual output (see Section 3.4).

Figure 4 gives an abridged trajectory showing the system prompt, user input, model output, and tool feedback that successfully solves a given problem.

3.3 Synthetic Trajectory Generation

We wrote a Python script to generate training trajectories containing solver feedback using Countdown problems taken from the TinyZero dataset.

At each step, we randomly pick an operation from $+, -, \times, /$ and two numbers from the set of numbers available at that step. We output this operation as the model's output, and use it to call into the solver. The solver's response is parsed, and added to the conversation. This process is iterated until either the model outputs an answer or the max message limit is reached.

To improve LM performance, the trajectory generation code avoids picking operations that will never lead to the target number. In particular, no generated trajectory:

```
System: You will play the game of Countdown. The goal is to use the available
numbers to reach the target. At each step, look at the available numbers and
choose an arithmetic operation to do on two of them. The tool will give you
feedback on your choice.

User: Available: [17, 2, 49]. Target: 83.

Assistant: Available: [17, 2, 49]. Right steps: []. Try: 49 + 2
Tool: BAD. 49 + 2 does not lead to a solution.

[...][elided]

Assistant: Available: [17, 2, 49]. Right steps: []. Try: 17 * 2
Tool: GOOD. 17 * 2 leads to target. New available: [49, 34]

Assistant: Available: [49, 34]. Right steps: [17 * 2 => 34]. Try: 49 + 34
Tool: END. 49 + 34 reached the target!

Assistant: <answer>49 + 17 * 2</answer>
```

Figure 4: A trajectory with the structured output format that outputs the correct answer. Some intermediate steps are elided for brevity.

- Repeats an unsuccessful operation in the same state, or
- Subtracts a larger number from a smaller, as negative numbers need never be constructed in Countdown

These safeguards shrink the size of the search space and streamline the learning process for the LM. Indeed, in initial experiments LM performance was seen to improve after implementing these safeguards.

However, in line with Gandhi et al. (2024) who include unsuccessful trajectories in their training data, we still wanted to include some invalid operations in ours. Therefore, at each step, with a small probability, we pick a number that is not currently available. In this case the tool pipeline does not call the solver but instead preemptively returns an error indicating that an invalid number was used. This invalid operation was included because we observed that this mistake was quite common. We also generate synthetic data in the loop for our DPO variants that we describe in 6.

3.4 Solver

We use an open-source Countdown solver written in C panzi ([n. d.]). It supports Countdown problems of arbitrary size. For problems of size 3 or 4, it can instantly search all possibilities and provide all solutions.

Figure 5 shows the solver's output for a problem of size 3 that has a single solution, and figure 6 shows its output for a problem instance with no solutions. Note that the LM does not see this output.

```
$ ./build/numbers - 47 20 3 13
tasks = 12
target = 47
numbers = [3, 13, 20]

solutions:
  1: 3 * 20 - 13
```

Figure 5: Solver output for a Countdown problem of size 3. Note our LM does not see this full output, it only gets class level feedback.

```
$ ./build/numbers - 42 43 68 18
tasks = 12
target = 42
numbers = [18, 43, 68]

solutions:
  no solutions found
```

Figure 6: Solver output for a Countdown problem with no solutions.

4 Experimental Setup

All experiments were performed on a g6e.xlarge instance on AWS, which comes equipped with an NVIDIA L40S GPU with 48 GB of GPU memory. In total we spent over $500 in AWS credits.

We generated 5,000 training trajectories using Countdown problems from the TinyZero dataset. We used these trajectories to perform supervised finetuning (SFT) of Qwen-2.5 0.5B. SFT was done for 10 epochs with a batch size of 4. To simulate a larger batch size, we used gradient accumulation. Gradients were accumulated for 16 batches before every optimizer update, resulting in an effective batch size of 64. AdamW was used as the optimizer with a learning rate of 5e-6. The max message limit was set to 20.

To evaluate the effectiveness of our methodology, we trained via SFT a copy of the same pretrained model on the Warmstart dataset Gandhi et al. (2025). This dataset was created by sampling a much larger model, Qwen-2.5 3B, and contains various examples of "cognitive strategies" employed by the LM to solve Countdown problems, including backtracking, setting sub-goals, etc. Training was performed on the entire dataset (1,000 rows) with a batch size of 4 for 4 epochs. As before, effective batch size was 64 due to gradient accumulation being done every 16 batches. The same optimizer and learning rate were used.

HuggingFace `transformers` library was used to load and train models. PyTorch was used to take gradients and perform backpropagation. W&B was used to track experiment runs, and models and datasets were shared among the team via HuggingFace Hub.

5 Results

We plot the exponential moving average (EMA) of training loss versus time with smoothing factor 0.5 for training on our generated dataset. As we can see, SFT finished in approximately two hours and hyperparameter tuning was impactful in increasing slope and reducing variance 7.

Performance for both LMs was evaluated using the `compute_score` function taken from the original paper. Table 3 gives the average score for each LM, evaluated on 100 randomly selected Countdown problems from the test split of the TinyZero dataset. Each problem had 3 or 4 input numbers, and the target was a 2-digit number. We see that our model significantly outperforms the model trained with SFT on the Warmstart dataset.

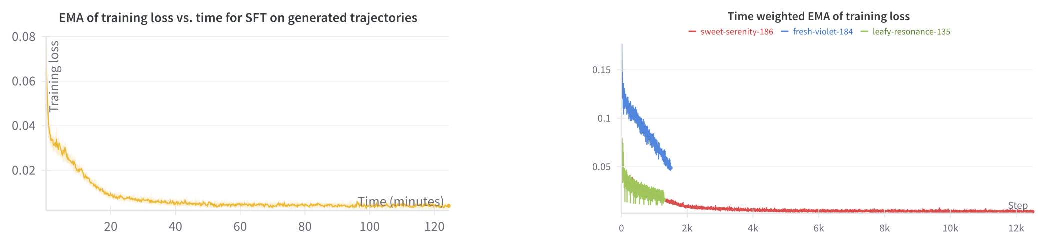

Figure 7: EMA of training loss during SFT over our generated dataset (left) and three training loss curves with hyperparameter tuning to accelerate and stabilize training runs. Notice improved slope and less jagged trajectories (right).

Training Method	Average Score
SFT (generated dataset) + Tool Use	0.65
SFT (Warmstart dataset)	0.30

Table 1: Comparison of average score for the two LMs. Both models are separate finetunes of pretrained Qwen-2.5 0.5B.

5.1 Substitution Errors

To understand why our model could not solve 30% of problems, we conduct error analysis on trajectories. We identified a frequent "substitution error". In trajectories containing this error, the model correctly reached the target, received `END` from the solver, but failed to construct the final answer expression: it did not replace an intermediate number by the operation that constructed it.

As an example, Figure 8 shows a trajectory that was performed by our finetuned model. Notice how the final answer, 52 + 32 - 63 is numerically equal to the target, 21, but uses an intermediate number, 32, which is not allowed. To form the correct answer it should replace 32 by its production, 90 - 58, resulting in 52 + 90 - 58 + 63, the correct answer.

```
System: You will play the game of Countdown. The goal is to use the available
numbers to reach the target. At each step, look at the available numbers and
choose an arithmetic operation to do on two of them. The tool will give you
feedback on your choice.

User: Available: [90, 63, 58, 52]. Target: 21.

Assistant: Available: [90, 63, 58, 52]. Right steps: []. Try: 90 * 58
Tool: BAD. 90 * 58 does not lead to a solution.

[...][elided]

Assistant: Available: [63, 84]. Right steps: [90 - 58 => 32, 52 + 32 => 84].
Try: 84 - 63
Tool: END. 84 - 63 reached the target!

Assistant: <answer>52 + 32 - 63</answer>
```

Figure 8: A trajectory containing a substitution error. The model has found the correct sequence of operations but fails to substitute 90 - 58 in place of the intermediate number 32 in the final answer. Some intermediate steps are elided for brevity.

Substitution errors are especially interesting for two reasons.

Training Method	Average Score
SFT + Tool Use (sft-syn-tool)	0.65
SFT + DPO v0 (subst. errors) + Tool Use	0.50

Table 2: Comparison of average score obtained via SFT and SFT + DPO.

- The LM remembers each successful operation and the intermediate number it generated in the “Right steps” list. So, all information required to construct the right answer are available. This design was intentional, of course. So it’s surprising to see that the LM did not learn this pattern after extensive finetuning.
- The scoring function applies a 90% penalty for answers that use an invalid number, regardless of whether the answer numerically computes the target or not. The score for a completely correct answer is 1.0, and for a properly formatted answer is 0.1, with nothing in between. We believe answers with substitution errors ought to be judged as “less wrong” than answers that don’t result in the target. However, we chose to not modify the scoring function for compatibility.

6 Direct Preference Optimization (DPO)

To mitigate the occurrence of substitution errors, we attempted DPO. Note that the original generated trajectories did not contain any substitution errors, therefore it’s possible that the LM did not receive negative feedback on committing a substitution error.

6.1 Synthetic Preference Pair Generation (DPO)

For DPO, we constructed a preference dataset on top of our training trajectories. Each example in the preference dataset had a chosen and rejected variant. The only difference in both variants was in the answer expression in the final message. In the chosen variants, the final answer was completely correct. In the rejected variants, the final answer had a substitution error. In particular, the final answer was equal to the last successful operation. This choice results in a variable number of substitution errors depending on the depth of the successful answer. In all other respects, both variants were identical.

6.2 DPO with distance-based reward variants

Next, we implemented DPO with reward-based preference generation to improve the distance between the preference pairs. Motivated by Gandhi et al. (2024) we used two reward functions: 1. sum-of-distance measuring distance from target and 2. reward-factor-distance measuring distance to the closest factor combined to automatically label response quality.

For each countdown problem, we generated 4 model responses at temperature 0.8, scored them using our reward functions, and created preference pairs where Chosen responses had rewards > 5 points higher than Rejected responses. The alternatives are sorted by reward, then pairs with max distance are created. We hypothesized that reward-based preference learning would guide the model toward mathematically valid solutions by preferentially training on responses that were closer to correct answers. We experimented with different reward and pair formulation set ups.

Reward-based DPO with standard distance to target showed modest improvement on the held-out test set (72% vs 70% baseline accuracy). Analysis revealed the primary limitation was incomplete text generation and mathematical hallucination in the base SFT model, rather than preference learning failure. The reward-based approach shows some promise in identifying better responses, but the underlying model’s generation quality remained the bottleneck.

6.3 Experimental Setup

We generated 2,000 preference pairs using this process. DPO was done for 2 epochs with a batch size of 1. Gradient accumulation was done every 32 epochs. β was set to 0.25. RMSprop was used as the optimizer with learning rate 1e-6. These hyperparameters were picked to resemble the choices made in the original DPO paper Rafailov et al. (2024).

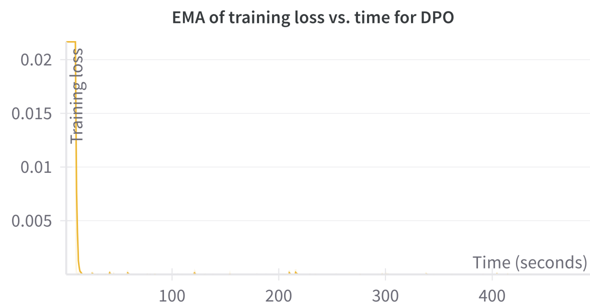

Figure 9: Plot of EMA of training loss during DPO v0 over generated preference pairs.

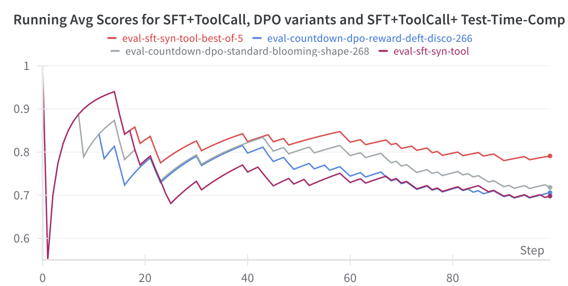

Figure 10: Average scores for SFT+ToolCall, DPO variants and SFT+ToolCall+Test-Time-Compute-Best-of-N over generated preference pairs.

Model	Test Set (Heldout) Score
SFT (generated dataset) + Tool Use (sft-syn-tool)	0.65
DPO v1 (dpo-standard, distance to target)	0.72
DPO v2 (dpo-reward, distance to factors)	0.71
SFT (generated dataset) + Tool Use + TTC N=5 (sft-syn-tool-best-of-5)	0.79

Table 3: Comparison of scores across the Test Set (A 1k sample offered by staff referred to as the "Heldout" data) for DPO variants.

6.4 Results

We saw initial DPO training loss sharply drops within a few batches (9). In fact, for most of the training run the loss was less than 1e-6. On evaluation of this LM, we saw that not only was it unable to reduce the frequency of occurrence of substitution errors, but it had worse overall performance than the SFT-only LM. Table 2 compares the performance of the initial DPO and SFT. Evaluation was done as described in Section 4. In Steam of Search (Gandhi et al., 2024) they suggest DPO pairs need to have reasonable distance between them. We then experimented with variants of rewards to guide pair formation and found creating four alternatives, scoring, sorting and selecting max distance pairs improved results. We gradually extended the number of runs to see if the gains hold: 10, 100, 250. The improvements are not trivial but not conclusive at a p-value <= 0.95. Finally, we were able to boost the SFT + tool call winning approach with best of 5 test time compute to 0.79.

6.5 Discussion

The loss of performance post DPO v0 was surprising. We validated our DPO implementation and have not found any issues. Moreover, we have separately performed DPO for a different problem (instruction following) using the same code and observed significant gains in performance. Therefore, we do not believe there is an issue with our DPO implementation.

Instead, we suspect the issue lied with the preference pairs. Perhaps the issue is that the difference between chosen and rejected variants is too small. As discussed previously, the only difference is in the final answer expression—one contains a substitution error, the other does not. The remaining trajectory, every choice of operation and tool feedback, is identical within each each preference pair. This hypothesis is corroborated by the minuscule values of the training loss. We then reformulated DPO to create variants, sort them by reward, and select pairs with maximum distance to target and factors. DPO performance improved from 0.50 in the first DPO pass to 0.72, but not conclusively over 0.70 with SFT and tool calling.

Therefore, we conclude that a different training methodology is needed to mitigate substitution errors, one that can extract non-trivial signal from preference pairs. We would like to explore this in future work.

7 Conclusion

We have presented a complete methodology for solving Countdown problems using Qwen-2.5 0.5B. We described a structured output format that is amenable to tool calling, and described how to interface a preexisting Countdown solver as a tool to guide the LM in search. We performed SFT and showed that our model outperformed the same pretrained model finetuned on a separate dataset for Countdown. We identified a common mistake committed by our finetuned LM, and implemented DPO in order to mitigate it. We ended with a discussion about potential reasons for DPO's ineffectiveness at mitigating these mistakes, and gave some directions for future work.

8 Team Contributions

- **Rupanshu Soi:**
  - Pre-extension: Data loading and construction, SFT on smoltalk, DPO on Ultrafeedback, evaluation for both.
  - Extension: generation of synthetic trajectories for SFT, design of tool calling format and pipeline, evaluation of SFT with tool calling, generation of preference pairs for DPO, evaluation of SFT + DPO with tool calling, writing of proposal, milestone, poster and final report.
- **Masoud Charkhabi:**
  - Generation of synthetic trajectories and preference pairs, evaluation, and reproducibility infrastructure.
  - DPO with reward learning on countdown. Reward formulations. Test-Time-Compute. Evaluations across SFT, DPO, Reward-Learning, Test-Time-Compute. Proposal, poster, and final report writing.

Changes from Proposal We were able to execute the main idea in our proposal, namely to leverage a classical solver for Countdown to improve performance of a small LM on this task. However, we did not implement test-time scaling as we achieved decent performance with tool calling alone.

References

Kanishk Gandhi, Ayush Chakravarthy, Anikait Singh, Nathan Lile, and Noah D. Goodman. 2025. Cognitive Behaviors that Enable Self-Improving Reasoners, or, Four Habits of Highly Effective STaRs. arXiv:2503.01307 [cs.CL] `https://arxiv.org/abs/2503.01307`

Kanishk Gandhi, Denise Lee, Gabriel Grand, Muxin Liu, Winson Cheng, Archit Sharma, and Noah D. Goodman. 2024. Stream of Search (SoS): Learning to Search in Language. arXiv:2404.03683 [cs.LG] `https://arxiv.org/abs/2404.03683`

Seungyong Moon, Bumsoo Park, and Hyun Oh Song. 2024. Guided Stream of Search: Learning to Better Search with Language Models via Optimal Path Guidance. *arXiv preprint arXiv:2410.02992* (2024).

Jiayi Pan, Xiuyu Li, Long Lian, Charlie Snell, Yifei Zhou, Adam Yala, Trevor Darrell, Kurt Keutzer, and Alane Suhr. 2025. Learning Adaptive Parallel Reasoning with Language Models. arXiv:2504.15466 [cs.AI] `https://arxiv.org/abs/2504.15466`

panzi. [n. d.]. GitHub - panzi/numbers-c: Countdown numbers game solver written in C. — github.com. `https://github.com/panzi/numbers-c`. [Accessed 26-04-2025].

Rafael Rafailov, Archit Sharma, Eric Mitchell, Stefano Ermon, Christopher D. Manning, and Chelsea Finn. 2024. Direct Preference Optimization: Your Language Model is Secretly a Reward Model. arXiv:2305.18290 [cs.LG] `https://arxiv.org/abs/2305.18290`

A Implementation Details

```
def run_dpo(pi, ref, tokenizer, dataloader, optimizer,
    epochs, grad_accum, save_path, save_every_epoch=True):
    pi.train()
    ref.eval()

    for epoch in tqdm(range(epochs), desc="Epochs"):
        optimizer.zero_grad()
        for idx, batch in enumerate(dataloader):
            chosen = batch["chosen"]
            rejected = batch["rejected"]

            pi_chosen_output = pi(**chosen)
            pi_chosen_logps = get_batch_logps(
                pi_chosen_output.logits, chosen["labels"]
            )

            pi_rejected_output = pi(**rejected)
            pi_rejected_logps = get_batch_logps(
                pi_rejected_output.logits, rejected["labels"]
            )

            with torch.no_grad():
                ref_chosen_output = ref(**chosen)
                ref_chosen_logps = get_batch_logps(
                    ref_chosen_output.logits, chosen["labels"]
                )

                ref_rejected_output = ref(**rejected)
                ref_rejected_logps = get_batch_logps(
                    ref_rejected_output.logits, rejected["labels"]
                )

            dpo_loss = -torch.nn.functional.logsigmoid(
                BETA
                * (
                    pi_chosen_logps
                    - ref_chosen_logps
                    - pi_rejected_logps
                    + ref_rejected_logps
                )
            ).mean()
            dpo_loss /= grad_accum

            if idx % grad_accum == grad_accum - 1:
                optimizer.step()
                optimizer.zero_grad()

        if save_every_epoch or (epoch == epochs - 1):
            pi.save_pretrained(save_path)
            tokenizer.save_pretrained(save_path)
```

Figure 11: The main DPO loop with gradient accumulation. Code for logging and checking loss has been removed.