

# Countdown to Brilliance: Evolving Math Reasoning Through Train and Test-Time Compute

**Motivation** Supervised fine-tuning (SFT) demonstrates significant improvements in the Countdown task, but is limited by the quality of the dataset. We explore two methods of improving performance over SFT: 1) additional train-time compute and 2) additional test-time compute. Train-time compute aims to optimize the dataset through iterative augmentation and curriculum; test-time compute exploits known reward models to increase performance through increased samples.

**Method** Expert iteration (ExIt) augments the SFT dataset by having multiple rounds of "experts" add to the dataset, increasing diversity through off-policy samples. Each expert generates responses to unsolved Countdown problems, but this leads to poor sample efficiency as experts attempt problems that are too easy/hard. Curriculum learning (CL) solves this by finding an optimal difficulty level for each expert to solve.

Due to the stochastic sampling of LLMs, multiple generation samples can have different answers. Best-of- $N$  sampling exploits this at test-time by generating  $N$  responses for each question. Then, a verifier can score each response, and the best response is chosen. To further improve sample efficiency, we dynamically vary  $N$  with an exponential schedule for each problem based on its difficulty.

**Implementation** The methods are evaluated by training the Qwen 2.5 0.5B base model. The provided Countdown warmstart dataset is used as a baseline for SFT. ExIt, CL, and RLOO sample unsolved problems from the same TinyZero dataset. Three different hand-crafted curriculum for CL are tested by increasing available operators, number count, and number range. We also discuss the usage of third-party LLMs for automatic curriculum crafting.

**Results** As a baseline, SFT on the warmstart dataset achieves a 0.33 score on the test dataset. ExIt vastly improves model performance—surpassing reinforcement learning methods like RLOO through fine-tuning alone—with a score of 0.51. CL achieves comparable performance (0.40) to ExIt with only 35% of the samples using a naive hand-crafted curriculum. Test-time compute achieves the best score by far of 0.93 at the cost of additional samples during inference time. Notably, we outperform much larger commercial models like Gemini-Pro-2.5-06-05 Low Reasoning Mode (which scores 0.78) and reached 2nd place on the CS224R class leaderboard.

**Discussion** A good training dataset for SFT can outperform reinforcement learning, as ExIt surpasses RLOO in performance. At the same time, ExIt is the least sample-efficient training method tested. Curriculum learning improves sample efficiency, but crafting an effective curriculum is unintuitive and not trivial; further work should aim to automate curriculum crafting as using third-party LLMs proved to be intractable. Test-time compute is far more effective than train-time compute in improving Countdown performance, achieving a near perfect score with enough samples. However, test-time methods add inference latency that is undesirable in user-facing applications.

**Conclusion** This work explores tradeoffs in sample efficiency and model performance. For domains like Countdown where verification is fast and cheap, test-time compute quickly improves performance with small number of samples. The added inference latency can be reduced by transferring compute from test to train time using expert iteration and curriculum learning. Boosting the training dataset with off-policy samples significantly boosts SFT performance at the cost of compute, but this cost can be minimized with cleverly designed curriculum.

---

# Countdown to Brilliance: Evolving Math Reasoning Through Train and Test-Time Compute

---

**Codey Sun**

Department of Electrical Engineering  
Stanford University  
codeysun@stanford.edu

**Doug Fulop**

Department of Computer Science  
Stanford University  
fulop@stanford.edu

**Xiang Li**

Department of Computer Science  
Stanford University  
kevin02@stanford.edu

## Abstract

Countdown is a math reasoning task used to evaluate the arithmetic reasoning of a large language model (LLM). Through supervised fine-tuning (SFT) alone, emergent reasoning capabilities demonstrate impressive performance on simple Countdown problems. However, SFT performance is limited by the quality of the training dataset. This work explores expert iteration and curriculum learning as methods of improving SFT performance through dataset augmentation. In addition, we explore the tradeoff of train-time compute and test-time compute methods for optimizing Countdown performance.

## 1 Introduction

Countdown is a math reasoning task used to evaluate the arithmetic capabilities of a large language model (LLM). Given an array of numbers and a target, the LLM must find the correct sequence of arithmetic operations (from  $+$ ,  $-$ ,  $*$ ,  $/$ ), to convert the numbers to the target. For example,

```
nums: [72, 30, 29]
target: 72
answer: (30 - 29) * 72
```

The simplicity of Countdown’s format and score function, yet challenge for small LLMs to solve effectively, makes it an accessible test bed for math reasoning. This project explores methods improving the LLMs performance in the Countdown task through expert iteration and curriculum learning. In addition, we evaluate the effect of test-time inference strategies in improving Countdown performance.

## 2 Related Work

Self-play in RL is a variant of multi-agent in which the model competes with a copy of itself in an adversarial manner. Doing so ensures that the model learns in an environment with an adversary of similar level as the agent, maximizing performance gains. This self-play method has shown promising results in games such as Go Silver et al. (2017).

The most related work to us is STP Dong and Ma (2025), which trains a single agent to act as both a conjecturer that comes up with math statements with appropriate difficulty level and act as a prover

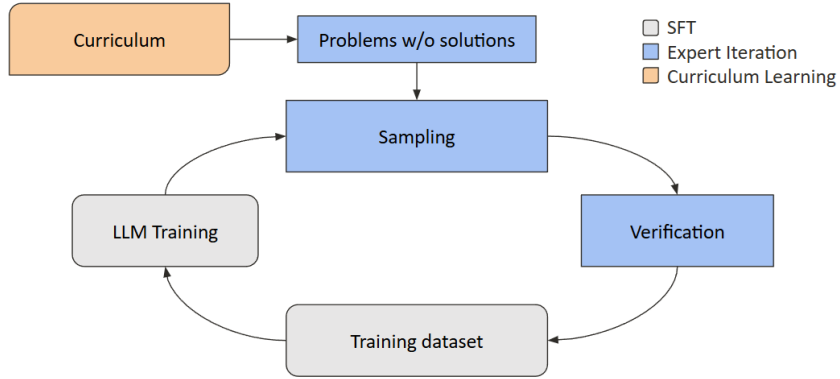


Figure 1: Method Overview. Gray blocks show the vanilla SFT pipeline using the warmstart dataset. We first add the expert iteration (blue) to iteratively augment the dataset with new samples given unsolved problems. Next, we add a curriculum (orange) to improve sample efficiency.

to prove those statements. Previous works like Wu et al. (2024) wastes the majority of their compute on unsuccessful proofs that are too challenging for the model. STP solves this issue by iteratively and dynamically adjusting the difficulty level of synthesized proofs so that the prover can receive meaningful, dense reward signals during training.

MINIMO Poesia et al. (2024) aims to have a model learn conjecturing and theorem proving through intrinsic motivation—learning from an environment with no explicit goals. In this case, the environment’s action space is the axioms of a mathematical domain. Like STP, MINIMO performs self-play alternating between conjecturing and proving. MINIMO also introduces hindsight labeling, in which failed trajectories are recycled as successful trajectories by changing the goals, improving sample efficiency. The key difference from STP is that MINIMO trains from scratch in an environment with finite action space (the axioms).

Both of these works focus on theorem-proving in formal languages through automatic curriculum learning. Our work relaxes this constraint, instead applying self-play to the Countdown task. In addition, we explore different curriculum crafting methods and compare their effectiveness in both sample efficiency and effectiveness.

Scaling test-time compute has been shown to more effectively improve model performance on math problems Snell et al. (2024). Given the same sample budget, Cobbe et al. (2021) doubles the test solve rate on GSM8K by using verifier-based test-time scoring. Their simple idea, best-of-N, rollouts N samples from the model and returns the best sample with the highest verifier score. We explore the effect of test-time compute on the Countdown task.

### 3 Method

As part of the default project, the base Qwen 2.5 0.5B model is pretrained on a warmstart dataset through supervised fine-tuning (SFT). This pretrained model can then act as a starting point for reinforcement learning methods like RLOO Ahmadian et al. (2024).

This project explores two methods of improving performance over vanilla SFT: 1) additional train-time compute and 2) additional test-time compute. Train-time compute primarily attempts to augment the initial SFT dataset through expert iteration and curriculum learning. Test-time compute explores how increasing samples and exploiting a known verifier can significantly increase performance. We also compare the sample efficiency of the methods.

#### 3.1 Train-time Compute

SFT alone achieves significant improvements in the Countdown task from the base model. However, given a limited dataset, SFT saturates quickly as the training expends all available knowledge from the given examples. In addition, the dataset may contain erroneous examples that limit the model’s

upper bound performance. Solving this garbage-in, garbage-out problem is the intuition behind the first improvement: augmenting the dataset with off-policy samples.

Expert iteration (ExIt) aims to augment the SFT dataset by having multiple iterations of “experts” add to the dataset Havrilla et al. (2024). After the base model is trained through SFT, it is treated as an expert. Afterwards,  $N$  unsolved Countdown problems are sampled and answered by this expert. Correct responses—as verified by the reward function—are appended to the SFT training dataset, and the model is retrained on the augmented dataset. Repeating this process effectively adds many off-policy trajectories being included in the dataset, making the dataset more diverse. Note that for each iteration, we retrain the model back from the base model. Otherwise, the model learns little additional information from its own responses.

However, blindly sampling problems for experts to answer results in very low sample efficiency as hard problems can take many samples to solve correctly, if solvable at all. This can be solved by defining a problem difficulty level suitable for the model. As demonstrated in Dong and Ma (2025), the order in which a model learns skills can drastically vary its reasoning abilities. In other words, by crafting a *curriculum* that allows the model to effectively learn foundational skills, curriculum learning (CL) can be 1) more efficient and 2) more effective. In practice, we compare three different hand-crafted curriculum and discuss the use of LLM-crafted curriculum in a multi-agent setup.

Figure 1 showcases the improvement added to the SFT pipeline.

### 3.2 Test-time Compute

Naively sampling  $N$  times for each question is compute intensive. This is because some questions are easy and a solution can be found in one or very few trials while other questions are much more challenging and require more trials. The optimal strategy in theory is to sample each question variable number of times until a correct solution is verified. However, this can be very slow in practice due to suboptimal use of GPU resources. To strike a balance between sample efficiency and GPU utilization, we design an exponential sampling schedule as shown in Algorithm 1, where we batch sample unfinished questions in each iteration and doubles the number of samples for each question.

---

#### Algorithm 1 Exponential Sampling Schedule

---

**Require:** Dataset  $D = \{q_1, q_2, \dots, q_n\}$  of queries

**Ensure:** Best completion  $c_i^*$  for each query  $q_i$

```

1:  $U \leftarrow \{1, 2, \dots, n\}$  ▷ Unfinished query indices
2:  $k \leftarrow 1$  ▷ Initial sample count
3: for  $i \in \{1, \dots, n\}$  do
4:    $C_i \leftarrow \emptyset$  ▷ Completions for query  $i$ 
5:    $S_i \leftarrow \emptyset$  ▷ Scores for query  $i$ 
6: end for
7: while  $|U| > 0$  do
8:   for  $i \in U$  do
9:     Generate  $k$  completions for query  $q_i$ 
10:    for each completion  $c$  do
11:       $s \leftarrow \text{EVALUATE}(q_i, c)$ 
12:       $C_i \leftarrow C_i \cup \{c\}$ 
13:       $S_i \leftarrow S_i \cup \{s\}$ 
14:    end for
15:  end for
16:   $U \leftarrow \{i \in U : \max(S_i) < 1.0\}$  ▷ Keep queries without perfect score
17:   $k \leftarrow 2k$  ▷ Double the sample count
18: end while
19: for  $i \in \{1, \dots, n\}$  do
20:    $c_i^* \leftarrow \arg \max_{c \in C_i} S_i(c)$  ▷ Best completion for query  $i$ 
21: end for

```

---

## 4 Experimental Setup

The initial base model is the Qwen 2.5 0.5B model. The Asap7772/cog\_behav\_all\_strategies warmstart dataset is used as the initial dataset for SFT Gandhi et al. (2025). For RLOO and expert iteration, unsolved Countdown problems are sampled from the Jiayi-Pan/Countdown-Tasks-3to4 dataset provided by TinyZero Pan et al. (2025). For validation, we hold out 200 unsolved problems from this TinyZero dataset.

For SFT, we train with an effective batch size of 8 for 5 epochs; the learning rate is  $5e-5$  and decays on a linear schedule. Each expert iteration follows these same hyperparameters with an additional boosting step, wherein responses are sampled until 250 correct responses are added to the training dataset.

For test-time compute, we use our SFT model and the exponential sampling algorithm to sample up to 16384 times per question and pick the attempt with the highest score.

For curriculum learning we manually created three curriculum (two of which are shown in Table 1 that generate Countdown problems of increasing challenge for the model (with target numbers between 1 and 100). A third curriculum includes up to 6 6-digit numbers, a target number between 0 and 999,999, and a solution that requires targeted placement of parentheses. We initially used the same hyperparameters from SFT, along with the addition of a starting difficulty level of 0 and a threshold of 70% successful completions to prompts before the model was allowed to progress to the next difficulty level. We experiment with applying the curriculum to the base Qwen model, the best performing SFT model, and the best performing RLOO model. To understand the importance of targeted difficulty we also perform a grid search of starting difficulty between levels 0 and 11 out of 12 difficulty levels.

Difficulty Level	Cur 1: Operator Complexity		Cur 2: Number Count	
	Count of Numbers	Available Operators	Count of Numbers	Available Operators
0	3	{+}	3	{+}
1	3	{+,-}	3	{+,-}
2	3	{+,-,*}	3	{+,-,*}
3	3	{+,-,*,/}	4	{+}
4	4	{+}	4	{+,-}
5	4	{+,-}	4	{+,-,*}
6	4	{+,-,*}	5	{+}
...	...	...	...	...

Table 1: **Comparing the curriculum of Operator Complexity and Number Count.** Division moves from level 3 in Operator Complexity to level 9 in Number Count as the division operator proved to be more challenging for the model to learn than to solve problems with increased counts of numbers from 3 to 4 or 4 to 5.

We also prompted OpenAI’s GPT-4o and 04mini, Google’s Gemini Pro and Flash, and Claude 3.7 Sonnet to produce a curriculum based on a sample of problems that were challenging to our current model iteration but we discovered that these larger models were incapable of generating valid countdown problems within a reasonable period of time (for example, less than 4% of training examples generated by Gemini Flash were valid countdown problems where the proposed solution used all the numbers and equaled the target. Each valid problem took between 15 and 30 seconds to generate which proves to be intractable as an expert iteration round requires continually generating batches of 250 valid problems until our model successfully solves 250 problems, which could require hours per batch given an initially low success rate (e.g. at an initial success rate we saw of 0.1 this could require 10 batches which would require 2,500 LLM-generated examples and a minimum of 10 hours for one iteration, compared to seconds of generation with other curriculum approaches.

For RLOO, we train with an effective batch size of 64 with  $k = 8$  for 1000 iterations. Note that the  $k$  responses are resampled if their rewards are equal. The reward function provided by TinyZero is used, which outputs 1.0 for correct responses, 0.1 for correct formats, and 0.0 otherwise.

Experiments are run on a RTX 4090 GPU with 24GB VRAM, an L40S with 48GB VRAM on AWS, an H200 on Runpod, and H100 on SF Compute.

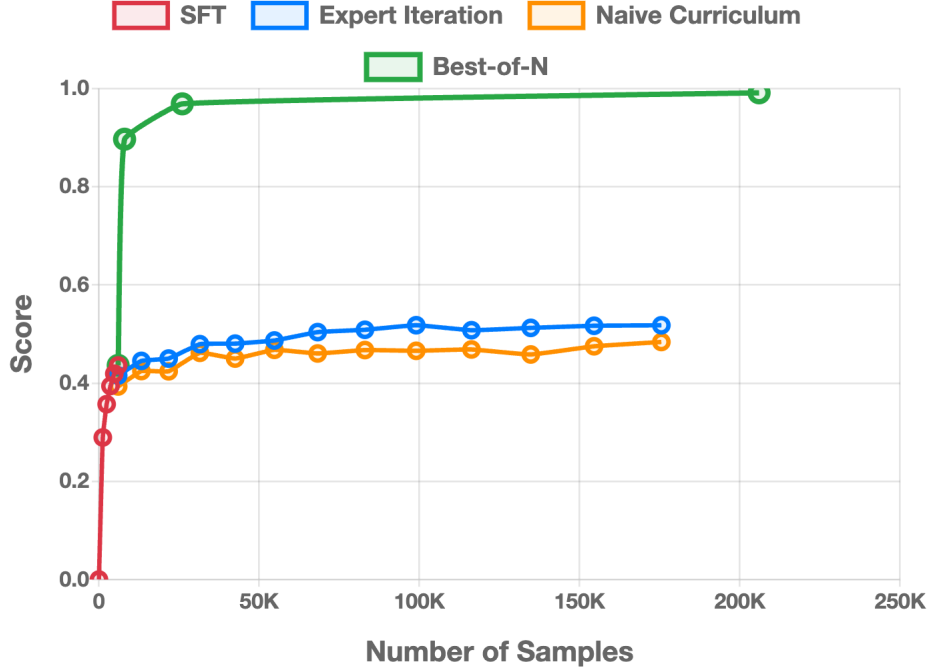


Figure 2: **Countdown score vs number of samples seen by the model.** Scores are calculated over a random subset of JiayiPan/Countdown-3to4 with 200 samples. Scores for test-time compute are calculated on another random subset of the same dataset but we believe the performance should be similar.

Method	Milestone Score	Final Score
SFT	0.57	0.33
ExIt	0.70	0.51
CL	0.59	0.40
RLOO	0.69	0.47
TTC	0.99	0.93

Table 2: Test results on leaderboard held-out data, both the milestone and final variants.

## 5 Results

Results are summarized in Figure 2. Both ExIt and CL demonstrate improvements over vanilla SFT. However, test-time compute method test-time compute far outperforms all other methods with a fraction of the cost.

### 5.1 Quantitative Evaluation

Tables 2 lists the evaluation results on the Leaderboard Milestone and Final test datasets, respectively. As shown, Best-of-N outperforms other train-time compute methods by far. ExIt outperforms RLOO, showing that good SFT data can outperform RL methods.

#### 5.1.1 Test-Time Compute Results in Great Improvement but Plateaus Quickly

As shown in Figure 3, number of questions solved increases quickly as we increase the number of samples for each question. However, the growth rate plateaus after 32 samples. Additionally, we plot the distribution of attempts per question in 4 and study its correlation with features of the question like number count and average digits. Overall, we found that the number of attempts is extremely skewed to the left with the vast majority of the questions being easy and requiring only few attempts to solve. We also discovered a weak positive correlation between the question number count and

attempts required, showing that increasing the length of the questions makes them more difficult for our SFT model which is only trained on 3 to 4 number counts. Interestingly, we found that the average number of digits in the question is very weakly correlated with number of attempts required, showing that our SFT model can effectively generalize to larger number of digits.

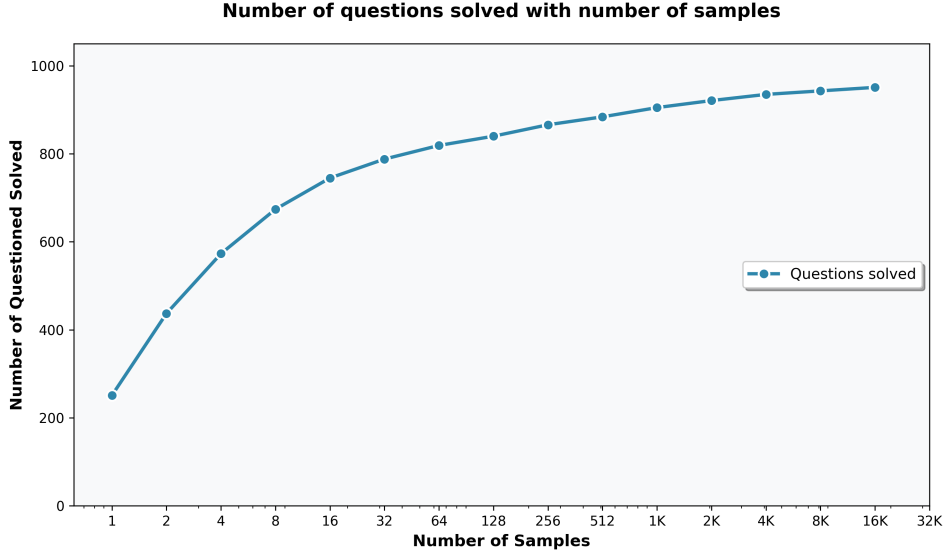


Figure 3: Number of questions solved increases quickly as number of samples increases from 1 to 32. Then, the growth rate plateaus even as number of samples keeps on doubling.

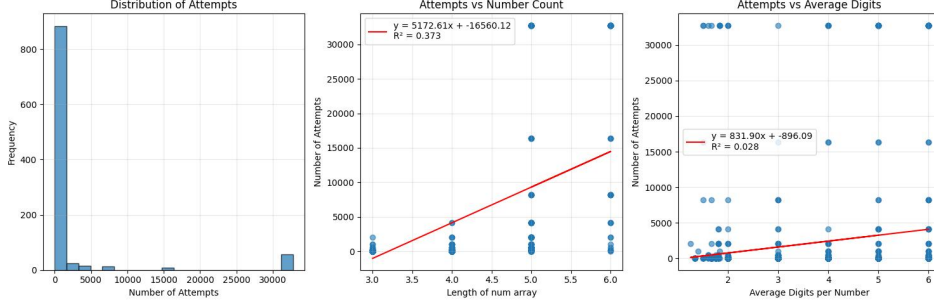


Figure 4: **Left:** Histogram of number of attempts per question. **Middle:** Number of attempts has weak positive correlation with length of input num array. **Right:** Number of attempts is very weakly correlated with number of digits in the numbers of the questions.

### 5.1.2 Test-Time Compute Outperforms Gemini on Low Reasoning

We found that Google’s Gemini-Pro-2.5-06-05 was able to solve 100% of the first 50 problems from the holdout test when using High reasoning, though it only solved 78% of these same problems with Low reasoning. We also note that these 100 total generations with the Gemini API cost \$3.54 which suggests the importance of empowering smaller models to solve tasks. As our model was able to solve 93% of these Countdown problems we see that test-time compute of our small model can outperform a larger model that uses low-levels of reasoning, at a fraction of the compute cost.

### 5.1.3 Curriculum Learning is More Sample Efficient than Expert Iteration

Curriculum learning (CL) overcomes the sample inefficiency of ExIt by intelligently picking problems for the expert to solve. Figure 5 illustrates this benefit—CL achieves comparable performance with only 35% of the samples that ExIt needed. This is due to ExIt answering problems with < 30%



Figure 5: **Sample efficiency of ExIt (orange) vs. CL (green).** **Left:** success rate of response samples. CL has a much higher success rate for responses and thus reaches 250 boost responses quicker. Note that the dips in success rate occur at difficulty level increases. **Right:** validation score. CL achieves comparable score with 35% of the samples of ExIt.

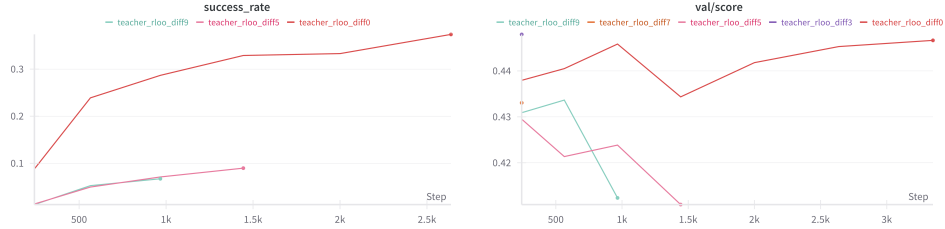


Figure 6: **Curriculum Learning starting with RLOO Model as base.** **Left:** The success rate shows the model’s ability to successfully solve problems at the current difficulty level. **Right:** The validation score shows the model’s score on the heldout test problems. From these graphs we can see that only the curriculum that starts at difficulty level 0 is able to improve in validation performance while the others fail to learn with challenging initial problems.

success rate, while CL has on average a 50% success rate. Note that with each step-up in difficulty, the success rate of CL plummets but quickly rises again. CL’s performance plateaus at difficulty level 4, which increases from 3 numbers to 4 numbers.

#### 5.1.4 Challenges of Designing Curriculum Difficulty Levels

In addition to the challenge of designing an appropriate curriculum the models are also sensitive to the hyperparameter of starting difficulty level. In Figure 6 we show the validation scores of starting curriculum learning at 5 different difficulty levels (level 0, 3, 5, 7 and 9 out of 12) with an allowed 75 minutes per run on an Nvidia H100 each. We see that starting at difficulty 0 leads to the best performing curriculum learning model. From success rates in the figure we see that the model is able to achieve non-zero success at difficulty levels 0, 5, and 9.

As the success rate determines how many times the model must re-sample to find 250 successful examples to add to its replay buffer, we can see that the difficulty 0 model approaches 2x the steps of difficulty 5 model and 3x the steps of difficulty 9 model in the same period of time. However the models that start training at levels 3 and 7 are not shown in this chart as they fail to achieve a success rate above 0, which led to a complete failure of the curriculum process, as the models failed to add entries to their replay buffer. Notably levels 3 and 7 are the difficulty levels where the model is given 6 numbers to use instead of 3 numbers at difficulty level 0 and 4 at difficulty level 5 and 9. The reduced success rate at level 9 compared to level 5 is explained by the introduction of division at level 9. The model appears to tolerate the requirement of using parentheses in solutions without issue, as an unintuitive result compared to adding a 6th number.

## 5.2 Qualitative Analysis

Figure 7 lists a sample response demonstrating the emergent reasoning capabilities of the model. We can see that the model learns to factor target numbers and backtrack when it reaches a dead end in its current solution path.



**Question:** Using the numbers [28, 88, 30, 56], create an equation that equals 44

1. First, let's look at factors of 44:
  - 1, 2, 4, 11, 22, 44
  - None of our numbers are factors of 44 directly
2. Let's try to work backwards from 44:
  - We could try to get a number close to 44 and then adjust
  - Or we could try to get a number larger than 44 and subtract
3. Looking at the numbers:
  - 88 is quite large, maybe dividing it would help
  - 28 and 30 could be used to reduce 88
  - 56 could be divided to get closer to 44
4. Let's try:  $88 \div 30 = 2.8$  (approximately)
  - This isn't helpful as we need exact integers
5. What if we start with smaller numbers:
  - $28 + 30 = 58$
  - $58 - 56 = 2$
  - $88 \div 2 = 44$

This works!
6. Let's verify:
 
$$88 \div (28 + 30 - 56) = 44$$

**Final Answer:**  $88 \div ((28 + 30) - 56) = 44$  ✓

Figure 7: A sample response demonstrating our model's systematic reasoning strategies, including factoring target numbers and backtracking when stuck.

## 6 Discussion

Test-time compute is far more effective than train-time compute. Thus, one may assume that allocating compute to inference is the best solution for performance. However, in latency-critical applications this is not desirable as it drastically degrades the user experience. Train-time compute methods are still a critical research area in practice.

While a hand-crafted curriculum for Countdown via simple arithmetic steps seems intuitive to humans, this curriculum is challenging for an LLM—difficulty increases that seem trivial turn out to be insurmountable for the LLM, e.g., the division operator or adding one additional number from 5 numbers to 6 numbers. In fact, creating an effective curriculum is time-consuming, difficult, and unintuitive. This highlights the need for an automatic curriculum. While we attempted this through a third-party LLM, the high latency for responses made this intractable.

## 7 Conclusion

Both expert iteration and curriculum learning can significantly improve the performance of a pre-trained SFT model, even surpassing reinforcement learning algorithms. Creating an effective curriculum is not trivial, and future work should aim to automatically create an optimized curriculum. To push Countdown performance to the maximum, test-time compute methods far outperform train-time compute methods. However, train-time compute methods are still valuable as additional latency during inference can degrade user experience.

## 8 Team Contributions

- **Codey Sun:** SFT; expert iteration; RLOO
- **Doug Fulop:** Curriculum learning
- **Xiang Li:** SFT; RLOO; test-time compute

## References

Arash Ahmadian, Chris Cremer, Matthias Gallé, Marzieh Fadaee, Julia Kreutzer, Olivier Pietquin, Ahmet Üstün, and Sara Hooker. 2024. Back to Basics: Revisiting REINFORCE Style Optimization

- for Learning from Human Feedback in LLMs. arXiv:2402.14740 [cs.LG] <https://arxiv.org/abs/2402.14740>
- Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. 2021. Training Verifiers to Solve Math Word Problems. arXiv:2110.14168 [cs.LG] <https://arxiv.org/abs/2110.14168>
- Kefan Dong and Tengyu Ma. 2025. STP: Self-play LLM Theorem Provers with Iterative Conjecturing and Proving. *arXiv e-prints* (2025), arXiv–2502.
- 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>
- Alex Havrilla, Yuqing Du, Sharath Chandra Raparthy, Christoforos Nalmpantis, Jane Dwivedi-Yu, Maksym Zhuravinskiy, Eric Hambro, Sainbayar Sukhbaatar, and Roberta Raileanu. 2024. Teaching large language models to reason with reinforcement learning. *arXiv preprint arXiv:2403.04642* (2024).
- Jiayi Pan, Junjie Zhang, Xingyao Wang, Lifan Yuan, Hao Peng, and Alane Suhr. 2025. TinyZero. <https://github.com/Jiayi-Pan/TinyZero>. Accessed: 2025-01-24.
- Gabriel Poesia, David Broman, Nick Haber, and Noah Goodman. 2024. Learning formal mathematics from intrinsic motivation. *Advances in Neural Information Processing Systems* 37 (2024), 43032–43057.
- David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. 2017. Mastering the game of go without human knowledge. *Nature* 550, 7676 (2017), 354–359.
- Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. 2024. Scaling LLM Test-Time Compute Optimally can be More Effective than Scaling Model Parameters. arXiv:2408.03314 [cs.LG] <https://arxiv.org/abs/2408.03314>
- Zijian Wu, Suozhi Huang, Zhejian Zhou, Huaiyuan Ying, Jiayu Wang, Dahua Lin, and Kai Chen. 2024. Internlm2. 5-stepprover: Advancing automated theorem proving via expert iteration on large-scale lean problems. *arXiv preprint arXiv:2410.15700* (2024).