Extended Abstract

Motivation The quality and diversity of training data limits the final performance of reinforcement learning (RL) finetuned large language models (LLMs), particularly for reasoning tasks [1]. Dong and Ma showed that the challenge of scarcity of high-quality training data for the complex reasoning task of theorem proving can be overcome by generating synthetic data for supervised finetuning (SFT) in an adversarial self-play process [2]. In this paper, we adapt Dong and Ma's approach for Countdown, a simple, multi-step mathematical reasoning problem expressed in natural language [3]. We show that encouraging the generation of barely solvable problems while training the policy on a mixture of real and synthetic data improves performance across the entire pass@k curve at each self-play iteration.

Method We replicate the five steps of Dong and Ma's Self-play Theorem Prover (STP), with adaptations for Countdown. We initialize the finetuned Qwen model with two roles, one for generating candidate problems, and another to generate solutions. We filter valid problem and solution pairs using a deductive verifier, and we finetune the generator and solver models on such pairs using cross-entropy loss. On top of this basic loop we add two interventions: weighting the solver's loss by problem difficulty (the inverse of each problem's solve rate), and replaying a buffer of our original Countdown data alongside the synthetic data at each iteration. We iterate over these steps and evaluate performance of the solver at each iteration.

Implementation To isolate the effect of each intervention, we organize our experiments as a 2×2 design, yielding a vanilla run (uniform loss, synthetic data), a difficulty-weighted run, a replay run, and a combined run that applies both. We finetune the Qwen 2.5 0.5B model once on 2,048 warm-up records to produce a shared generator warm-up checkpoint, and launch all four runs from this identical checkpoint. Every run is executed for three self-play iterations. At each iteration we generate 1,000 verified problems and sample 8 candidate solver rollouts per problem, retaining the single, shortest correct rollout per verified problem. The difficulty-weighted cells scale each record's CE loss by the inverse of its problem's solve rate under the current policy. The replay cells mix verified-correct original Countdown pairs into the synthetic training set at an approximately 70:30 synthetic-to-real ratio. We evaluate by drawing 128 samples per test problem under a fixed evaluation seed in vLLM, and report pass@k for $k \in \{1, 4, 8, 16\}$.

Results With just 1,000 synthetic rows of SFT data generated per iteration over three iterations, every configuration improved pass@1 over the baseline, but they differed vastly across the rest of the pass@k curve. The vanilla algorithm raised pass@1 by $+17.7\%$ but fell below the baseline at pass@8 (-1.6%) and pass@16 (-2.8%), and difficulty weighting alone behaved similarly, improving pass@1 by $+13.2\%$ while dropping further at pass@16 (-5.0%). In contrast, replaying real data improved the entire curve (pass@1 $+17.9\%$, pass@16 $+2.1\%$), and the combined configuration was best at every k, improving pass@1 by $+19.6\%$, pass@4 by $+6.6\%$, pass@8 by $+4.1\%$, and pass@16 by $+3.2\%$.

Discussion We read pass@1 as the reliability of the policy's single best answer and pass@k at large k as the diversity of the correct reasoning paths it can reach. Vanilla self-play, and difficulty weighting in particular, sharpen the dominant mode, raising pass@1 by reinforcing solutions the policy already favors while leaving large-k pass@k flat or lower. Replaying real data acts as a diversity counterweight that anchors the policy to the original data distribution and prevents mode collapse, lifting the entire curve. Combining it with difficulty weighting it is best at every k, turning aggressive learning on hard problems from a weakness at large k into a gain. Our main limitation is scale: we generated only 1,000 synthetic rows per iteration and ran at most three iterations under a tight compute budget.

Conclusion With as few as 1,000 records of synthetic data generated per iteration, our self-play algorithm can measurably improve performance at Countdown. Our central finding is that vanilla self-play only sharpens, raising pass@1 while leaving large k pass@k flat or lower. Mixing real and synthetic data at each iteration breaks this behavior, lifting the entire pass@k curve and adding reasoning paths. Combined with difficulty-weighting, we observe additive improvements in our model's test accuracy.

A Self-Play Algorithm for Countdown

Nikhil Raman
Department of Computer Science
Stanford University
nraman25@stanford.edu

Nadim Isaac
Department of Computer Science
Stanford University
nadisaac@stanford.edu

Abstract

Large language models (LLMs), when prompted to "think step by step", are capable of solving simple reasoning problems [4]. These emergent reasoning capabilities are often honed through reinforcement learning (RL) finetuning techniques in multiple stages [1]. The quality and diversity of training data can pose challenges to maximizing reasoning performance of LLMs, because poor quality training data can result in a weak initial policy, and lack of diversity can result in mode collapse [1]. Dong and Ma's Self-play Theorem Prover (STP), an algorithm capable of improving continually, overcomes the scarcity of training data through an adversarial data generation process, gated by a formal verifier [2]. In this paper, we adapt the STP for Countdown, a multi-step mathematical reasoning problem [2, 3]. We show that training the model with supervised finetuning (SFT) on a mixture of real and synthetic problems, using a cross-entropy (CE) loss weighted by difficulty, lifts the entire pass@k curve and improves pass@1 by up to 19.6% over just three iterations.

1 Introduction

LLMs display an improved ability to reason when prompted to "think step by step", as measured by performance on popular reasoning benchmarks [4]. This emergent reasoning capability of LLMs can be honed through multi-stage post-training, involving SFT on chain-of-thought (CoT) data to learn an initial policy, followed by preference and policy-gradient optimization techniques [1]. But the initial policy can only be as good as data used to train it; lack of diverse, high quality training data can constrain the initial policy, which in turn can constrain the performance of the final model. Dong and Ma showed that the lack of diverse, high quality training data can be overcome by generating synthetic data in an adversarial process of generating candidate problems and solutions, and retraining the LLM based on verified problem and solution pairs, while ensuring the quality and diversity of the generated data through de-duplication, random sampling, difficulty filters, and sample weighting [2].

Our primary goal is to adapt Dong and Ma's STP (originally developed for theorem proving, an abstract reasoning problem) for Countdown, a multi-step mathematical reasoning problem, the object of which is to show how a provided target value can be obtained by performing basic arithmetic operations (+, -, *, /) on a provided set of numbers [1, 3]. Our secondary goal is to measure the impact of applying select features, including difficulty based filters and sample weights on pass@k, a performance metric that allows the model k attempts to correctly solve a problem.

We introduce a novel modification to the Dong and Ma's approach to minimize the compute resources needed: After several self-play iterations, Dong and Ma retrain their algorithm on the union of the real and synthetic dataset, a highly compute-intensive process, to mitigate the effect of the changing synthetic data distribution. Instead, at each self-play step, we sample batches of data from the real and synthetic datasets, with equal probability to mitigate the same effect.

Stanford CS224R 2025 Final Report

2 Related Work

Dong and Ma introduced the STP, a formal theorem proving algorithm, which overcomes the scarcity of high-quality training data by mimicking how mathematicians push the boundaries of their field through an iterative process of trial and error, whereby they generate candidate theorems by extending and combining existing ones, and subsequently attempt to solve them [2]. Concretely, the STP consists of five steps, namely conjecturing, proving, verifying, assigning reward, and training, which are repeated: In the conjecturing step, an SFT finetuned model is used to generate a new conjecture, given a seed conjecture and proof pair. In the proving step, the model is used to generate possible proofs of the conjecture from the previous step. In the verifying step, pairs of conjectures and candidate proofs are passed to a formal verifier, which selects valid pairs. In the assigning reward step, the valid pairs are further filtered based on a set of criteria such as the difficulty of the conjecture and the elegance of the proof. In the training step, the LLMs are trained using the valid, filtered pairs of conjectures and proofs. After the desired number of self-play iterations, the base model is retrained on a combination of real and synthetic data to mitigate the effects of the changing distribution of the synthetic data. The STP algorithm set the new state-of-the-art performance baseline on the LeanWorkbook dataset, proving 28.5% of the statements in the dataset, doubling the previous best result [2].

Poesia et al. introduced Mathematics from Intrinsic Motivation (MINIMO), a self-play algorithm, similar to STP, which learns to pose and solve challenging mathematical conjectures, using a language model, trained from scratch, that encodes the proof search policy, the value function and the conjecturer [5]. They report that their algorithm gets progressively better at proving theorems with each iteration [5].

Yang and Band introduced EntiGraph, a data generation algorithm that extends a small, domain-specific corpus by extracting the entities described in the corpus, creating a knowledge graph, and generating text descriptions of the relationships represented in the graph using an LLM [6]. Yong and Ma use EntiGraph to generate additional training data for the task of question answering on the QuALITY dataset. They report log-linear increases in task performance as they increase the number of synthetically generated training tokens [6].

3 Method

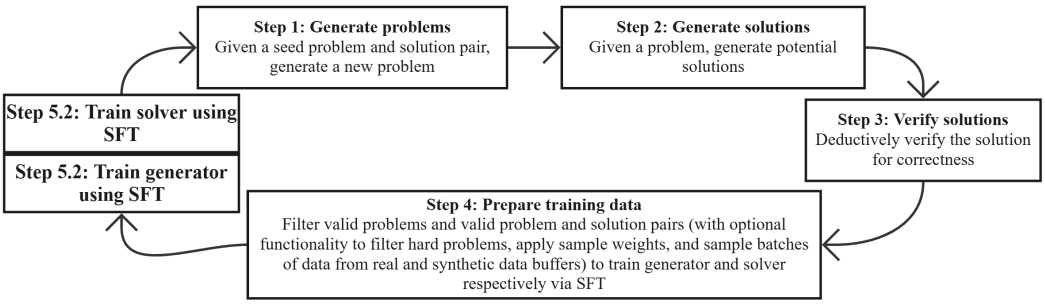

Figure 1: This is a high-level visualization of our adaptation of the Self-play Theorem Prover by Dong and Ma for Countdown [2].

We begin by replicating the five steps of the STP algorithm with adaptations for Countdown, as depicted in Figure 1. We start by finetuning the Qwen 2.5 0.5B model using a warm-start dataset for Countdown. We then initialize this with two roles, generator and solver. We use the former to generate candidate problems, given seed problem and solution pairs. To maximize the quality of the generated problems, we finetune the generator on a dataset we construct by extracting (seed problem, seed solution, new solution) tuples from the warm-start dataset, closely mirroring the approach taken by Dong and Ma [2]. Next, we extract the numbers and the targets from the generated problems and apply a brute-force algorithm that determines whether the generated problem is theoretically solvable. This algorithm permutes the numbers and applies all the possible arithmetic operations (/, *,

+, -). This step does away with the need to run the solver on invalid problems. We then use the solver to generate multiple candidate solutions for each valid problem. We identify the correct solutions, using a simple arithmetic evaluation function, and extract (valid problem, valid solution) tuples for our synthetic dataset. We filter difficult problems, based on their solvability rates under the current policy. We sample batches of data from the real and synthetic data buffers, with equal probability, and finetune the model using cross-entropy loss, weighted by the inverse of the solvability rates, as described in the equation below:

$$L_{CE}(\theta) = \mathbb{E}_{x,y\in D}\left[-\frac{1}{s_\pi(x)}\sum_{t=1}^{|y|}\log \pi(y_t|x, y_{<t})\right],$$

where x is the prompt, and y is the completion, and $s_\pi(x)$ is the solve rate of prompt x under the policy π over a given number of sampled responses. Lastly, we iterate over the above steps for the desired number of iterations, and evaluate the performance of the generator and solver at each iteration.

4 Experimental Setup

We study two interventions on top of our basic self-play adaptation of the STP for Countdown: first weighting the solver's cross-entropy (CE) loss by problem difficulty, and second replaying real data alongside the synthetic data at each training step. To isolate the effect of each intervention and to measure whether they interact, we organize our experiments as a 2×2 design over these two axes, summarized in Table 1. The four cells are: a *vanilla* run (uniform loss, synthetic data), a *difficulty-weighted* run, a *replay* run, and a *combined* run that applies both interventions as a capstone.

	Synthetic only	Real + synthetic replay
Uniform CE loss	Vanilla	Replay
Difficulty-weighted CE loss	Difficulty-weighted	Combined

Table 1: Our 2×2 design. Rows vary the solver loss weighting; columns vary the training data composition. The *Combined* cell is our capstone configuration.

Shared protocol. A central weakness of our early experiments was that each run used a different generator warm-up, making baselines vary. This has the possibility of reducing the meaningfulness of cross-run comparisons. We address this by fixing a single protocol across all four cells. We finetune the Qwen 2.5 0.5B model once on 2,048 warm-up records to produce a *shared* generator warm-up checkpoint, and all four ablations are launched from this identical checkpoint. Every run is executed for three self-play iterations. At each iteration we generate 1,000 verified problems and sample 8 candidate solver rollouts per problem. To hold the size of the SFT training set fixed at 1,000 records per iteration across all ablations, we retain the single, shortest correct rollout per verified problem. We evaluate by drawing 128 samples per test problem under a fixed evaluation seed in vLLM, and we report pass@k for $k \in \{1, 4, 8, 16\}$. Because the warm-up checkpoint and evaluation seed are shared, the four runs produce near-identical baselines (within 2.3% at pass@1, and within 1.6% across all k).

Hyperparameter	Value
Learning rate (solver)	2e-6
Learning rate (generator)	5e-6
Epochs per iteration	2
Gradient accumulation steps	8
Batch size	64
Self-play iterations	3
Verified problems per iteration	1,000
Solver rollouts per problem	8
Evaluation samples per problem	128

Table 2: Hyperparameter settings, held fixed across all four cells of the 2×2 design.

Difficulty weighting. In the two difficulty-weighted cells, we scale each solver training record's CE loss by the inverse of its problem's solve rate under the current policy. Therefore, harder, barely-solvable problems contribute more to the gradient. The two uniform cells weight every record equally.

Real + synthetic replay. In the two replay cells, we mix a buffer of real Countdown data into the synthetic training set at each iteration, rather than retraining on the union of all data only once at the end as Dong and Ma do [2]. From the original dataset we admit only verified-correct pairs (404 available), and combine them with the 1,000 synthetic records, yielding approximately 70:30 synthetic-to-real ratio. In the *Combined* cell, where difficulty weighting would otherwise distort this ratio, we rescale the replay weights so that the real and synthetic portions retain their intended 70:30 contribution to the loss. This allows the *Combined* cell to demonstrate the performance of a simultaneous difficulty-weighting and replay buffer approach.

5 Results

5.1 Quantitative Evaluation

We report pass@k (for $k \in \{1, 4, 8, 16\}$) on the held-out Countdown test set at the end of each self-play iteration, alongside the baseline measured after generator warm-up. Baselines across all four runs coincide to within 2.3% at pass@1 (and within 1.6% for higher k), so the differences we report below can be attributed to the interventions rather than to differing starting points.

5.1.1 Vanilla self-play algorithm for Countdown

Below we present the results of the vanilla cell of our design, where we measured the performance of our adaptation of the STP for Countdown in its most basic form. With uniform cross-entropy loss and synthetic data only, as described in the experimental setup section above.

Iteration	pass@1	pass@4	pass@8	pass@16
Baseline	31.61	57.19	64.73	69.68
1	34.36	58.50	64.98	69.54
2	35.19	57.51	63.30	67.32
3	37.20	58.24	63.69	67.70

Table 3: Performance of our vanilla adaptation of STP for Countdown. All figures are in percentages.

Pass@1 improved at each self-play iteration, ending +17.7% greater than the baseline by the third iteration. Pass@4 improved slightly (+1.8%), while pass@8 and pass@16 fell below the baseline, by -1.6% and -2.8% respectively.

5.1.2 Self-play algorithm with difficulty-weighted CE loss

Below we present the results of the difficulty-weighted cell of our design, where we trained the generator and solver using SFT with cross-entropy loss weighted by the inverse of the solve rate of the problems, while keeping the data synthetic-only.

Iteration	pass@1	pass@4	pass@8	pass@16
Baseline	31.58	58.01	65.52	70.12
1	33.42	58.45	65.29	69.60
2	35.52	59.09	65.33	69.39
3	35.73	57.94	63.18	66.62

Table 4: Performance of the difficulty-weighted variant of our self-play algorithm for Countdown, where each solver record's cross-entropy loss is weighted by the inverse of its problem's solve rate. All figures are in percentages.

By iteration 3, pass@1 improved by +13.2%, while pass@4 was essentially unchanged (-0.1%) and pass@8 and pass@16 fell by -3.6% and -5.0% respectively.

5.1.3 Self-play algorithm trained on real and synthetic data (replay buffer)

Below we present the results of the replay cell of our design, where the generator and solver were trained at each iteration on a mixture of the synthetic data and a buffer of verified real Countdown problems, at an approximately 70:30 synthetic to real ratio, with uniform cross-entropy loss.

Iteration	pass@1	pass@4	pass@8	pass@16
Baseline	31.25	57.33	64.88	69.37
1	34.84	59.52	66.43	71.07
2	35.92	59.17	65.35	69.70
3	36.83	60.67	66.84	70.85

Table 5: Performance of the replay variant of our self-play algorithm for Countdown, where generator and solver were trained on a mixture of synthetic and real data. All figures are in percentages.

By iteration 3, every point on the measured pass@k curve improved over the baseline: pass@1 by +17.9%, pass@4 by +5.8%, pass@8 by +3.0%, and pass@16 by +2.1%.

5.1.4 Combined self-play algorithm (difficulty weighting and replay buffer)

Finally, we present our capstone configuration, the combined cell, which applies both interventions at once: difficulty-weighted cross-entropy loss together with the real and synthetic replay buffer. Because difficulty weighting would distort the data mixture, we rescale the replay weights so that the 70:30 synthetic to real ratio is preserved.

Iteration	pass@1	pass@4	pass@8	pass@16
Baseline	30.89	57.23	64.62	69.06
1	33.98	58.96	65.70	70.10
2	36.17	60.29	66.64	70.64
3	36.95	61.03	67.24	71.29

Table 6: Performance of the combined variant of our self-play algorithm for Countdown, applying both difficulty-weighted cross-entropy loss and the real-and-synthetic replay buffer. All figures are in percentages.

By iteration 3, the combined configuration improved pass@1 by +19.6%, pass@4 by +6.6%, pass@8 by +4.1%, and pass@16 by +3.2%, the largest improvement of any cell at every value of k. This trend is visualized in the full pass@k curve against the baseline and the vanilla algorithm in Figure 2.

5.2 Qualitative Analysis

5.2.1 Quality of synthetic problems

As we show in Table 7, the synthetic problems were largely well formatted and novel, but only a fraction were solvable in theory. The generator learned to generate problems that were well formatted much faster than it learned to generate solvable problems. The solvability rate improved at each iteration, indicating that additional finetuning helped improve the performance of the generator model.

Iteration	Valid	Novel	Solvable
1	92.62	99.02	11.44
2	95.52	99.18	20.48
3	96.59	99.22	31.73

Table 7: Additional insights into the synthetic data generated at each iteration. We define problems that are formatted correctly as valid problems, problems that are unique as novel problems, and problems that the brute-force algorithm was able to find a solution for as solvable. All figures are in percentages.

Below we show some examples of synthetic problems generated by our model.

Solvable	Numbers	Target
No	99, 4, 94	98
No	42, 39, 32	54
Yes	91, 74, 59	76
Yes	38, 70, 13, 32	51

Table 8: Examples of problems generated by our model.

5.2.2 Pass@k across configurations

The table below plots the full pass@k curve, for continuous k from 1 to 16, of the warmed-up SFT checkpoint (the shared baseline), the vanilla self-play algorithm, and our combined capstone configuration, each at their final self-play iteration.

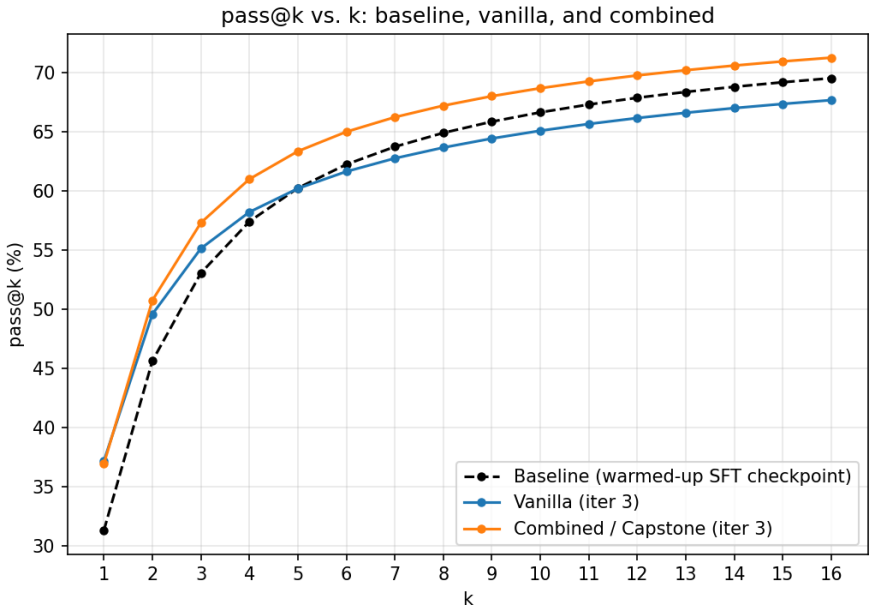

Figure 2: Pass@k as a function of k for the shared warmed-up SFT baseline, the vanilla self-play algorithm (final iteration), and the combined capstone configuration (final iteration). The vanilla curve rises above the baseline at small k but crosses below it near $k = 5$, whereas the capstone curve stays above the baseline across the entire range of k.

6 Discussion

6.1 Interpreting pass@k

Our 2×2 design separates two mechanisms. We read pass@1 as the reliability of the policy's single best answer, and pass@k at large k as the diversity of the correct reasoning paths it can reach. An intervention can either concentrate probability on the dominant mode, raising pass@1 but lowering pass@k for large k, or learn several correct modes, lifting the whole curve. The four cells fall cleanly between this single-mode versus multi-mode axis.

6.2 Vanilla self-play sharpens

The vanilla algorithm is self-distilling, in which each iteration fine-tunes the solver on its own verified-correct rollouts, and since we keep one solution per problem, training reinforces solutions the policy already favors. This sharpens the dominant mode, so pass@1 rose the most of any synthetic-only cell ($+17.7\%$) while pass@8 and pass@16 dropped below baseline (-1.6% and -2.8%), revealed in the crossing of curves near $k = 5$ in Figure 2. This is the sharpening mechanism of self-improvement: training on self-verified solutions concentrates mass on high-quality sequences, limited to reasoning paths the model can already produce. Because no new paths are added, large k pass@k cannot rise, and lifting the whole curve instead requires wider data coverage.

6.3 Difficulty weighting amplifies sharpening

Difficulty weighting on synthetic data alone amplifies this sharpening. Weighting each loss by the inverse solve rate lets hard, barely-solvable problems dominate the gradient and concentrate the distribution even more. This became clear when its high k loss was the worst of any cell (pass@16 -5.0%). It also spends the gradient budget on the hardest problems rather than the easy and medium ones that drive greedy accuracy, so its pass@1 gain ($+13.2\%$) trailed even vanilla. Without a diversity counterweight, emphasizing difficulty alone hurts both ends of the curve.

6.4 Replay adds coverage

The replay buffer works as that counterweight. At each step we mix in roughly 30% verified pairs from the original dataset that trained the initial SFT checkpoint, reusing our own earlier training data to anchor the policy to that broad, original data distribution. Without this, the model compoundingly drifts into the distribution of its own outputs, and leads to a model-collapse failure mode [7]. This collapse is avoidable: Accumulating real data alongside synthetic data, rather than replacing it, fixes it [8]. Implementing this strategy keeps the diverse paths that high k needs, which was observed in the replay cell. We lifted the entire curve (pass@1 $+17.9\%$ to pass@16 $+2.1\%$) with no collapse. This drives why we replay on every iteration rather than retraining once on the union at the end, as Dong and Ma do [2]. The real-data anchor is the mechanism that prevents compounding error and mode collapse.

6.5 The combined capstone wins at every k

The two interventions are complementary. Difficulty weighting drives aggressive learning on the problems the model is weakest at, while replay supplies the diversity that keeps this aggression from collapsing coverage. The capstone is best at every k, beating replay-only alone and flipping difficulty weighting from a weak high k performance (-5.0% alone) into a benefit ($+3.2\%$ combined). Only the replay cells gained at large k, suggesting replay is what turns self-play from sharpening the base model's existing solutions into expanding the reasoning paths it can reach [9].

6.6 Limitations and future work

We were constrained by a tight compute budget, hence we generated only 1,000 synthetic rows per iteration, ran at most 3 iterations, and trained only with SFT. Future work should scale both the data and the number of iterations, and extend the self-play loop beyond SFT to preference-based and policy-gradient methods such as IPO and RLOO.

7 Conclusion

We succeeded in our primary goal of building a self-play algorithm capable of improving continually for Countdown, replicating Dong and Ma's Self-play Theorem Prover for a simplified problem. As little as 1,000 records of synthetic data can measurably improve performance at Countdown. Our central finding is that vanilla self-play only sharpens, raising pass@1 while leaving large-k pass@k flat or lower. Mixing real and synthetic data at each iteration breaks this behavior, lifting the entire pass@k curve and adding reasoning paths. Combined with difficulty-weighting, we observe additive improvements in our model's test accuracy as the replay buffer protects coverage and diversity.

8 Team Contributions

Nikhil and Nadim contributed equally to the extension. We started development of the extension together, over a call. Nadim led the development of the generator and the algorithm to determine the solvability of the generated problems, and Nikhil led the development of the solver and the SFT trainer. We each implemented our own working versions of the self-play algorithm, before consolidating our work to make sure that we had reproducible results. Both Nikhil and Nadim collaboratively devised the list of features to build and experiments to run. Nikhil and Nadim met regularly to discuss what was and was not working to align on changes needed to keep the project on track.

9 Changes from Proposal

Our original project proposal entailed generating synthetic data for SFT as well as IPO. However, we realized after some experimentation that the compute requirements for training SFT and IPO over several self-play iterations would exceed our budget. Consequently, we pivoted to focusing on generating synthetic data for SFT, and implemented the novel modification to the self-play algorithm of training the generator and solver on a mixture of real and synthetic data at each self-play iteration.

References

[1] Stanford University. Rl fine-tuning of language models. In *Stanford University: https://cs224r.stanford.edu/material/CS224R_Default_Project_Guidelines.pdf*, 2026.

[2] Kefan Dong and Tengyu Ma. Stp: Self-play llm theorem provers with iterative conjecturing and proving. In *arXiv: https://arxiv.org/pdf/2502.00212*, 2025.

[3] Kanishk Gandhi. Stream of search (sos): Learning to search in language. In *arXiv: https://arxiv.org/pdf/2404.03683*, 2024.

[4] Aske Plaat. Multi-step reasoning with large language models, a survey. In *arXiv: https://arxiv.org/pdf/2407.11511*, 2025.

[5] Gabriel Poesia. Learning formal mathematics from intrinsic motivation. In *arXiv: https://arxiv.org/pdf/2407.00695*, 2024.

[6] Zitong Yang and Niel Band. Synthetic continued pretraining. In *arXiv: https://arxiv.org/pdf/2409.07431*, 2024.

[7] Ilia Shumailov, Zakhar Shumaylov, Yiren Zhao, Yarin Gal, Nicolas Papernot, and Ross Anderson. The curse of recursion: Training on generated data makes models forget. In *arXiv: https://arxiv.org/pdf/2305.17493*, 2024.

[8] Matthias Gerstgrasser, Rylan Schaeffer, Apratim Dey, Rafael Rafailov, Henry Sleight, John Hughes, Tomasz Korbak, Rajashree Agrawal, Dhruv Pai, Andrey Gromov, Daniel A. Roberts, Diyi Yang, David L. Donoho, and Sanmi Koyejo. Is model collapse inevitable? breaking the curse of recursion by accumulating real and synthetic data. In *arXiv: https://arxiv.org/pdf/2404.01413*, 2024.

[9] Yang Yue. Does reinforcement learning really incentivize reasoning capacity in llms beyond the base model? In *arXiv: https://arxiv.org/pdf/2504.13837*, 2025.

A AI Use Disclosure

We used Claude Code to implement the scaffolding for the self-play loop, the brute-force algorithm that determines whether a generated problem is theoretically solvable, the dataloaders for the generator, the metrics logging, end-to-end smoke tests, and experiment configurations to run on Modal. We implemented important elements of the self-play algorithm, including the SFT trainer and the batch sampler by hand.