SAGE: SELF-PLAY ADVERSARIAL GAMES ENHANCE LARGE LANGUAGE MODEL REASONING CAPABILITIES

Saraswathy Amjith **Michael Wang** **Jayson Lynch** **Hans Gundlach** **Neil Thompson**
MIT CSAIL
{swathy, wangrzm, jaysonl, hansgund, neil_t}@mit.edu

ABSTRACT

We introduce SAGE (Self-play Adversarial Games for Enhancement), a framework for improving LLM reasoning capabilities through adversarial self-play without human-curated data. SAGE places two model instances in an asymmetric game: a *Setter* generates a problem and predicts its solution, while an *Opponent* attempts to solve the problem independently. The Setter receives positive reward only when it answers correctly and the Opponent fails, incentivizing the generation of problems that are solvable yet challenging and naturally targeting the frontier of model capabilities. We instantiate SAGE in two domains: **Code-Game**, where problems are Python programs verified by execution, and **Math-Game**, where math problems are graded by an external LLM judge, as a proxy for a verifiable environment. Training models from 1B to 4B parameters across two architectures (Qwen3, Llama), SAGE consistently outperforms baselines: up to +10% on MATH, +8% on MBPP, and +6% on ARC-Challenge. Notably, we find cross-domain transfer: Code-Game training improves mathematical reasoning and vice versa, suggesting SAGE strengthens domain-general reasoning skills. Ablations confirm that adversarial pressure, rather than verified rewards alone, drives these gains: removing the opponent while retaining execution-verified rewards decreases the improvement by 40-70%. SAGE offers a scalable path to reasoning improvement that requires only a verifier, not human supervision.

1 INTRODUCTION

Large language models (LLMs) have achieved remarkable performance on code generation and mathematical reasoning benchmarks, yet their capabilities remain bounded by the quality and diversity of training data. As models approach or exceed human-level performance on existing benchmarks, the scarcity of novel, challenging problems becomes a critical bottleneck for further improvement (Zhao et al., 2025). We draw on work that has achieved superhuman performance through self-generated experience: *self-play* in game-playing agents (Silver et al., 2017) and *adversarial training* in generative models (Goodfellow et al., 2014). AlphaZero demonstrated that an agent can rapidly improve without human supervision by playing against itself, while GANs formalized model improvement as a minimax game between generator and discriminator.

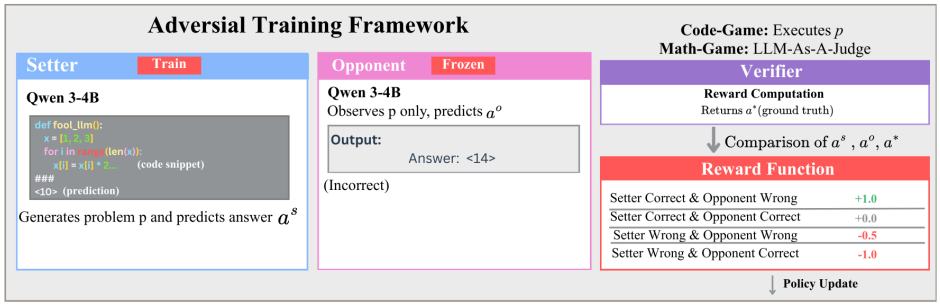

Figure 1: SAGE Overview

We propose **SAGE** (Self-play Adversarial Games for Enhancement), a framework that combines these insights for LLM improvement. In SAGE, two LLM instances engage in an asymmetric game:
1. The **Setter** (π_θ) generates a problem p and a predicted answer $\hat{a}$
2. The **Opponent** (π_{opp}) observes only p and produces its own prediction $\tilde{a}$
3. The **Verifier** ($\mathcal{V}$) computes the ground-truth answer a^*
4.The Setter receives reward based on whether it "fools" the Opponent, that is, whether $\hat{a} = a^*$ while $\tilde{a} \neq a^*$.
Here, the Setter is incentivized to generate problems that are (i) *verifiable* (correctness can be determined by comparing solutions against ground truth), (ii) *solvable* ($\hat{a}$ must equal a^*), and (iii) *challenging* ($\tilde{a} \neq a^*$). During training, this reward system drives the Setter toward increasingly sophisticated problems that probe the frontier of its current capabilities. We outline this in Figure 1.

We propose SAGE for two domains:
Code-Game The Setter writes a deterministic Python program that prints exactly one value (numeric, string, or boolean), and then predicts that value. The Opponent sees the code and attempts the same prediction. Rewards are verified by program execution.
Math-Game The Setter constructs a math problem of its choice, then predicts its answer (accurate to six decimal digits or symbolic answer). The Opponent attempts to compute the same value. Rewards are computed using LLM-judged correctness (GPT-4.1-mini), serving as a proxy reward.

2 Related Work

RLVR (RL w/ Verified Rewards) has emerged as a promising direction for LLM improvement (Zhao et al., 2025). While RLVR has been applied to code generation (Le et al., 2022; Shojaee et al., 2023; Yuxiang Wei, 2025b), these approaches typically rely on curated problem sets. SAGE's adversarial formulation provides a natural incentive for generating diverse, high-quality problems: the Setter is rewarded only when problems are verifiable, solvable, and challenging enough to defeat the Opponent, eliminating the need for external dataset curation. Recent work has advanced RL-based training for LLMs: CodeRL+ trains models to infer variable-level execution trajectories beyond binary rewards (Jiang et al., 2025), while Wen et al. (2025) show that RLVR extends reasoning boundaries by implicitly incentivizing correct chain-of-thought. GRPO estimates advantages from sample groups without a separate value network (Shao et al., 2024), and DeepSeek-AI (2025) demonstrated that GRPO with rule-based rewards can elicit sophisticated reasoning through pure RL. Fang et al. (2025) introduce SeRL, combining self-instruction generation with online filtering for quality and diversity.

Self-play has been applied to negotiation (Fu et al., 2023), non-zero-sum games (Liao et al., 2024), debate (Liang et al., 2024; Arnesen et al., 2024), and adversarial language games (Cheng et al., 2024). Liu et al. (2025) showed zero-sum games transfer to improved reasoning with 8.6% math gains. For alignment, self-play preference methods (Wu et al., 2024; Tang et al., 2025) use game-theoretic frameworks where models compete against prior versions, as in SPIN (Chen et al., 2024). Multi-agent approaches further enhance this: execution feedback in self-play improves instruction-following (Dong et al., 2024), while co-evolution among diverse agents yields complementary gains beyond single-agent improvement (Chen et al., 2025; Subramaniam et al., 2025). AlphaCode (Li et al., 2022) achieved human-competitive programming performance through massive sampling and execution-based filtering, while ACES (Pourcel et al., 2024) generates progressively harder puzzles using solve-rate heuristics. Self-improvement frameworks leverage model-generated feedback (Lee et al., 2023; Bai et al., 2022; Yuan et al., 2024) and iterative refinement for reasoning (Madaan et al., 2023; Shinn et al., 2023; Zelikman et al., 2022; Xuezhi Wang, 2022). Recent work extends these ideas to code: Maxime Robeyns (2025) introduce a coding agent that edits its own codebase through reflection, Genghan Zhang (2025) stratify self-generated experiences by difficulty for curriculum learning, and Yuxiang Wei (2025b;a) apply GRPO to software evolution with iterative bug injection and repair.

Absolute Zero (Zhao et al., 2025) and R-Zero (Huang et al.) also target reasoning improvement from zero external data through self-evolving task generation. SAGE differs in key respects: (i) it uses an explicit two-player adversarial game with separate Setter and Solver roles rather than a single model in both roles; (ii) its reward is directly adversarial (the Setter is rewarded for fooling the Solver) rather than based on task learnability; and (iii) it extends to domains with proxy verification (Math-Game

with LLM judge), beyond execution-verified coding tasks. R-Zero (Huang et al.) similarly targets reasoning LLM improvement from zero data through self-evolving curriculum generation.

While RLVR and value-based approaches like B-Coder (Yu et al., 2023) excel with abundant high-quality data, SAGE is complementary: it requires only a verifier and generates its own training problems in data-scarce settings, enables continuous curriculum generation through adversarial self-play that concentrates learning at the capability frontier, and we provide controlled ablations (SPR/DPDR, see 3.6) isolating the opponent's contribution beyond verified rewards alone.

3 METHOD

3.1 PROBLEM FORMULATION

We frame adversarial self-play as a two-player asymmetric game. Let $\mathcal{P}$ denote the space of problems (programs or mathematical expressions), $\mathcal{A}$ the space of answers (numeric values), and $\mathcal{V} : \mathcal{P} \to \mathcal{A} \cup \{\perp\}$ a verifier that computes ground-truth answers (returning $\perp$ for invalid problems).

Definition 3.1 (Adversarial Prediction Game)**.** An episode proceeds as follows:

1. **Setter move:** Given context c, the Setter π_θ generates $(p, \hat{a}) \sim \pi_\theta(\cdot|c)$
2. **Verification:** Compute $a^* = \mathcal{V}(p)$
3. **Solver move:** The Solver π_ϕ observes p and generates $\tilde{a} \sim \pi_\phi(\cdot|p, c^*)$
4. **Outcome:** Compare predictions to ground truth and assign rewards per Equations 1–2.

We consider two training regimes: **frozen-opponent** (ϕ fixed throughout training) and **co-evolving opponent** (ϕ updated alongside θ). We use "Opponent" for frozen and "Solver" for concurrent settings to emphasize the distinct training dynamics.

3.2 TRAINING ALGORITHM

We train the Setter using Group Relative Policy Optimization (GRPO) (Shao et al., 2024), which estimates advantages from groups of samples without requiring a separate value network.

Algorithm 1 SAGE Training Loop

Input: Setter π_θ; Solver π_ϕ (frozen or co-trained); Verifier $\mathcal{V}$; Batch size B; Rollouts per prompt K; KL coefficient β
Initialize: Reference policy $\pi_{\text{ref}} \leftarrow \pi_{\theta_0}$
for $t = 1, \ldots, T$ **do**
  // Rollout phase
  for $i = 1, \ldots, B$; $k = 1, \ldots, K$ **do**
    $(p_{i,k}, \hat{a}_{i,k}) \sim \pi_\theta(\cdot|c)$ ▷ Setter generates problem + prediction
    $a^*_{i,k} \leftarrow \mathcal{V}(p_{i,k})$ ▷ Verify via execution/grading
    if $a^*_{i,k} = \perp$ **then**
      $r_{i,k} \leftarrow -1$ ▷ Invalid problem penalty
    else
      $\tilde{a}_{i,k} \sim \pi_{\text{opp}}(\cdot|p_{i,k})$ ▷ Opponent attempts solution
      $r_{i,k} \leftarrow \text{Reward}(\hat{a}_{i,k}, \tilde{a}_{i,k}, a^*_{i,k})$ ▷ Eq. 1
    end if
  end for
  // GRPO update
  Compute group-normalized advantages: $A_{i,k} = \frac{r_{i,k} - \mu_i}{\sigma_i + \epsilon}$
  $\theta \leftarrow \theta + \alpha \nabla_\theta \left[J_{\text{GRPO}}(\theta) - \beta D_{D_{\text{KL}}}(\pi_\theta \| \pi_{\text{ref}}) \right]$
end for
return θ

3.3 REWARD STRUCTURE

The Setter's reward function captures the adversarial objective:

The Setter's reward function captures the adversarial objective:

$$r_{\text{setter}} = \begin{cases} +1.0 & \text{if correct}_G \wedge \neg\text{correct}_S \\ 0.0 & \text{if correct}_G \wedge \text{correct}_S \\ -0.5 & \text{if } \neg\text{correct}_G \wedge \neg\text{correct}_S \\ -1.0 & \text{if } \neg\text{correct}_G \wedge \text{correct}_S \end{cases} \quad (1)$$

When training the Solver concurrently, we use a simple correctness reward:

$$r_{\text{solver}} = \begin{cases} +1.0 & \text{if correct}_S \\ -1.0 & \text{if } \neg\text{correct}_S \end{cases} \quad (2)$$

3.4 Game Variants

Code-Game. The Setter generates a deterministic Python program p that prints exactly one value (boolean, string, number), along with a predicted output $\hat{a}$. The verifier $\mathcal{V}$ executes p in a sandboxed subprocess with resource constraints such as timeout (4 seconds maximum execution time), separate Python process with captured stdout/stderr, and Standard library only; no file I/O, network, or randomness. The ground-truth answer a^* is extracted from the program's stdout. This execution-based verification provides deterministic reward signals. An example of the coding game framework is shown in Figure 2.

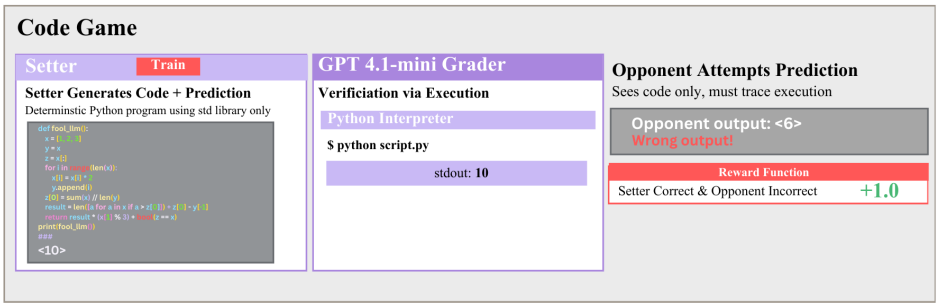

Figure 2: Code-Game Setup Overview

Math-Game. The Setter constructs a math problem of its choice, then predicts its answer (either as a numeric value or symbolic expression such as LaTeX fractions). The Opponent attempts to compute the same answer. Rewards are computed using LLM-judged correctness (GPT-4.1-mini) (OpenAI, 2025), which handles both numeric comparison (with tolerance $\epsilon = 10^{-6}$) and symbolic equivalence, serving as a proxy reward. This LLM-as-judge approach extends SAGE to domains where symbolic evaluation is insufficient, such as word problems requiring multi-step reasoning or problems with multiple valid answer formats, at the cost of introducing a learned verifier. An example of the math game framework is shown in Figure 3. During training, we periodically audited grading reliability on a fixed subset of 20 held-out problems and found that our GPT-4.1-mini–based grader matched the ground-truth labels on 19/20 cases (95% accuracy).We also manually inspected generated problems throughout training for signs of reward hacking (e.g., malformed problems that exploit judge parsing, adversarial phrasings). We observed no systematic exploitation: generated problems remained mathematically coherent, and errors were typical calculation mistakes rather than judge-specific exploits.

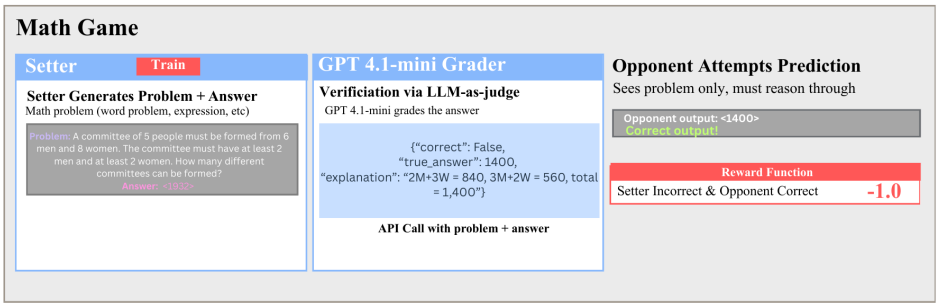

Figure 3: Math-Game Setup Overview

3.5 Opponent Variants: Frozen vs. Concurrent Training

We consider two training regimes that differ in whether the second player's parameters are updated. To clarify their distinct roles, we adopt the following naming convention used consistently throughout the paper (see Table 1):

Table 1: Terminology for opponent configurations. Both regimes share the same game mechanics; only the second player's training status differs.

Regime	**Second Player Name**	**Parameters**
Frozen	Opponent	ϕ fixed at initialization
Co-Evolving	Solver	ϕ updated via GRPO alongside θ

Frozen Opponent. The Opponent is initialized from the same base model as the Setter but remains fixed throughout training. This provides stable reward signals and isolates the Setter's learning dynamics. Results appear in Tables 2–3.
Co-Evolving Both Setter and Opponent (renamed Solver in this setting) update via GRPO, alternating training phases. The Solver receives reward based on correctness alone (Equation 2). This tests whether co-evolution produces complementary improvements, and can be utilized to prevent a potential saturation of reasoning capability increase against frozen opponent. Training details are in Appendix B.

3.6 Ablation Conditions

To isolate the contribution of adversarial pressure from other aspects of our training setup, we design two ablation conditions that systematically vary the reward structure and prompt framing:
SPR (Same Prompt, Different Reward) uses the identical game-framed prompt as SAGE, but rewards are based solely on execution correctness: $+1$ if the Setter's prediction matches the verified output, -1 otherwise. The Opponent is never queried. This isolates whether gains come from verified execution rewards alone.
DPDR (Different Prompt, Different Reward) uses a standard code/math generation prompt without game framing, with the same correctness-only reward as SPR. This tests whether the competitive prompt framing itself provides learning signal independent of adversarial rewards.
If adversarial pressure is essential, we expect the ordering SAGE $>$ SPR $>$ DPDR $\geq$ Baseline: removing the Opponent (SPR) should reduce gains, and further removing the game framing (DPDR) should reduce them further. If verified rewards alone suffice, SPR should match SAGE.

Relationship to RLVR Baselines.
Our DPDR ablation is structurally equivalent to standard RLVR: the model generates code or math solutions and receives binary rewards based on verified correctness. Our ablations control for general RLVR effects: any improvement of SAGE over DPDR cannot be attributed to verified rewards or GRPO optimization alone, since both conditions share these components.

If adversarial pressure is essential, we expect the ordering SAGE $>$ SPR $>$ DPDR $\geq$ Baseline: removing the Opponent (SPR) should reduce gains, and further removing the game framing (DPDR) should reduce them further. If verified rewards alone suffice, SPR should match SAGE.

3.7 Model and Evaluation Benchmark Selection

To evaluate the generality of SAGE across architectures and scales, we conduct experiments on two model families at multiple parameter counts. We use Qwen3-1.7B and Qwen3-4B (Team, 2025), representing small and medium-scale models within Alibaba's Qwen3 series. We additionally evaluate on Llama-3.2-1B-Instruct and Llama-3.2-3B-Instruct from Meta's Llama series (Team, 2024).

We evaluate on three benchmarks spanning code generation, mathematical reasoning, and general reasoning:
MBPP (Mostly Basic Python Programming) (Austin et al., 2021): Python programming tasks with test cases. All evaluations use temperature $\tau = 0.01$ for generation.

MATH. The MATH dataset (Hendrycks et al., 2021) covers algebra, geometry, number theory, and calculus.
ARC-Challenge. The challenge split of the AI2 Reasoning Challenge (**?**), testing commonsense and scientific reasoning. We measure accuracy on the multiple-choice format.
We evaluate both Math & Arc-Challenge on a fixed, pre-specified subset of 200 problems (selected prior to experimentation) used for all model variants.

4 RESULTS

We evaluate SAGE under two configurations: Frozen Opponent and Co-Evolving Opponent. Tables 2 and 3 report results for both regimes. We compare against three conditions: the untrained baseline, DPDR (standard generation with correctness rewards), and SPR (game-framed prompt with correctness rewards but no opponent). Across all model-benchmark combinations, we observe: **SAGE** $>$ **SPR** $>$ **DPDR** $\geq$ **Baseline**. The gap between SAGE (full adversarial objective) and SPR (same prompt, correctness-only reward) isolates the contribution of the opponent, if verified rewards alone sufficed, SPR would match SAGE. The ordering SPR $>$ DPDR confirms that the game-framed prompt provides some benefit even without adversarial rewards, likely by encouraging varied problem difficulty. We hypothesize that an improving opponent provides a richer curriculum than a frozen one, preventing the Setter from overfitting to a fixed adversary's weaknesses. One plausible explanation is that the frozen opponent has not saturated within our training budget; under this hypothesis, longer training could amplify the observed gap.

Table 2: Accuracy (%) for **Code-Game** post-training variants (pass @ 1).

Base Model	Variant	MBPP	MATH	ARC
Qwen3-1.7B	Baseline	32.0	73.0	82.0
	DPDR	33.0	75.0	83.0
	SPR	36.0	76.0	82.0
	Frozen-Opp Gen	40.0	**78.0**	**84.0**
	Co-Evolve Solver	39.0	77.0	83.0
	Co-Evolve Gen	**41.0**	**78.0**	83.0
Qwen3-4B	Baseline	54.0	82.0	83.0
	DPDR	55.0	84.0	84.0
	SPR	57.0	84.0	86.0
	Frozen-Opp Gen	59.0	**86.0**	88.0
	Co-Evolve Solver	58.0	83.0	89.0
	Co-Evolve Gen	**60.0**	84.0	**90.0**
Llama-3.2 1B-Inst	Baseline	27.0	24.0	36.0
	DPDR	27.0	25.0	38.0
	SPR	28.0	25.0	39.0
	Frozen-Opp Gen	**30.0**	27.0	**42.0**
	Co-Evolve Solver	26.0	28.0	35.0
	Co-Evolve Gen	28.0	**30.0**	36.0
Llama-3.2 3B-Inst	Baseline	32.0	49.0	67.0
	DPDR	33.0	50.0	71.0
	SPR	35.0	52.0	72.0
	Frozen-Opp Gen	40.0	54.0	75.0
	Co-Evolve Solver	35.0	54.0	75.0
	Co-Evolve Gen	**41.0**	**56.0**	**77.0**

SPR = Same Prompt, Diff Reward. DPDR = Diff Prompt, Diff Reward.

Table 3: Accuracy (%) for **Math-Game** post-training variants (pass @ 1).

Base Model	Variant	MBPP	MATH	ARC
Qwen3-1.7B	Baseline	32.0	73.0	82.0
	DPDR	34.0	75.0	83.0
	SPR	34.0	77.0	**84.0**
	Frozen-Opp Gen	**37.0**	**83.0**	**84.0**
	Co-Evolve Solver	35.0	82.0	82.0
	Co-Evolve Gen	34.0	**82.0**	83.0
Qwen3-4B	Baseline	54.0	82.0	83.0
	DPDR	54.0	83.0	84.0
	SPR	55.0	86.0	85.0
	Frozen-Opp Gen	**57.0**	**89.0**	**87.0**
	Co-Evolve Solver	55.0	88.0	85.0
	Co-Evolve Gen	55.0	88.0	86.0
Llama 1B-Inst	Baseline	27.0	24.0	36.0
	DPDR	28.0	29.0	35.0
	SPR	27.0	26.0	37.0
	Frozen-Opp Gen	**29.0**	37.0	34.0
	Co-Evolve Solver	27.0	38.0	38.0
	Co-Evolve Gen	25.0	**41.0**	**42.0**
Llama 3B-Inst	Baseline	32.0	49.0	67.0
	DPDR	32.0	50.0	66.0
	SPR	32.0	49.0	67.0
	Frozen-Opp Gen	33.0	52.0	68.0
	Co-Evolve Solver	33.0	58.0	72.0
	Co-Evolve Gen	**36.0**	**60.0**	**72.0**

SPR = Same Prompt, Diff Reward. DPDR = Diff Prompt, Diff Reward.

In Tables 2–3, "Frozen-Opp Gen" reports the *Setter*'s downstream performance under the frozen-opponent regime, while "Co-Evolve Gen" and "Co-Evolve Solver" report the Setter's and Solver's performance, respectively, under the co-evolving regime.

Figure 4 shows benchmark accuracy over training steps for SAGE across model families. Patterns emerging include steady improvement without collapse unsaturated learning curves. All configurations show monotonic or near-monotonic improvement throughout training. Both MBPP curves continue improving at step 750, suggesting additional training could yield further gains.

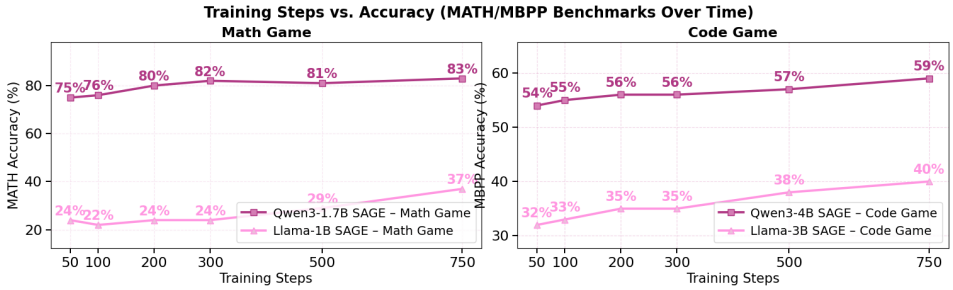

Figure 4: Benchmarks Over Training Steps (SAGE-Frozen) Overview

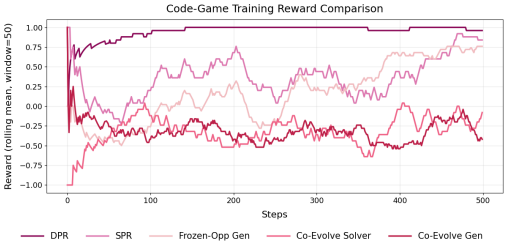

Figure 5: Code-Game Rolling Mean of Rewards During Training (Gen = Generator, Opp = Opponent)

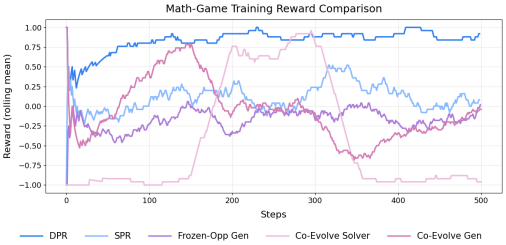

Figure 6: Math-Game Rolling Mean of Rewards During Training (Gen = Generator, Opp = Opponent)

Figures 5 and 6 show training dynamics against all SAGE variants (frozen-opponent and co-evolving opponent) along with the ablations. The figures reveal adversarial co-evolution: the Generator's reward (dark pink) dominates near 1.0 but degrades as the Solver (light pink) climbs from -1.0, with cyclical inversions demonstrating that each agent adapts to overcome the other's improving strategy. The Code-Game exhibits slower Generator improvement than the Math-Game, suggesting that generating valid, challenging code presents a harder learning signal than mathematical reasoning. DPDR maintains consistently high rewards and SPR exhibits more volatile learning curves. The upward trend in frozen opponent's reward indicates that the Setter learns to generate increasingly valid problems that it can solve but the frozen Opponent cannot.

Code-Game: Early Training (Step 50)

```
def puzzle(s):
    digits = sum(int(c) for c in s if c.isdigit())
    non_digits = len(s) - sum(1 for c in s if c.isdigit())
    return str(digits) + str(non_digits)
print(puzzle("a1b2c3"))
```

Output: 63 *Setter predicts:* 63 ✓

Simple string traversal. Opponent easily traces logic.

Code-Game: Late Training (Step 490)

```
def puzzle(x, nan=float("nan")):
    d = {nan: "K", float("nan"): "L", True: "T", 1: "O", 1.0:
         "P"}
    return d[nan] + d.get(float("nan"), "?") + d[1] + str(len
        (d))
print(puzzle(0))
```

Output: K?P3 *Setter predicts:* K?P3 ✓

Exploits NaN identity and `True/1/1.0` hash collisions.

Math-Game: Early Training (Step 50)

Evaluate the limit:

$$\lim_{x \to 0} \frac{\sin(\tan x) - \tan(\sin x)}{x^7}$$

Answer: $\frac{1}{630}$ *Setter predicts:* ✓

Standard Taylor expansion. Routine calculation.

Math-Game: Late Training (Step 490)

Compute:

$$\left(\frac{2^{2^{2^2}}}{3^{3^{3^3}}}\right)^{\frac{1}{\sqrt{\left(\frac{1}{2}\right)^{\frac{1}{2}}+\frac{1}{2}}}} + \left(\frac{1}{2}\right)^{\frac{1}{2}} + \left(\frac{1}{3}\right)^{\frac{1}{3}} - \left(\frac{1}{4}\right)^{\frac{1}{4}}$$

Answer: $\frac{1}{\sqrt{2}} + \frac{1}{\sqrt[3]{3}} - \frac{1}{\sqrt[4]{4}}$ *Setter predicts:* ✓

Tower exponential trap: $3^{3^{3^3}} \gg 2^{2^{2^2}}$*, first term vanishes.*

Figure 7: **Problem evolution during SAGE training.** Top: early (Step 50). Bottom: late (Step 490). Left: Code-Game. Right: Math-Game.

The learning is demonstrated in Figure 7. These two figures give example completions from both the Generator and Solver for coding and math, respectively. For each training type, we provide an early training and late training example to illustrate the progression in difficulty and solving capacity of the two models.

Code-Game training improves mathematical reasoning (MATH: up to +7 points), while Math-Game training improves code generation (MBPP: up to +5 points). We hypothesize this reflects shared underlying reasoning capabilities: both domains require understanding and solving problems and careful attention to edge cases. The adversarial objective may be particularly effective at strengthening these domain-general skills because it forces the model to identify and exploit subtle reasoning gaps.

5 Discussion

5.1 The Reward Paradox: Lower Training Rewards, Better Performance

A seemingly counterintuitive finding emerges from comparing training dynamics to downstream performance: SAGE frozen-opponent achieves *lower* training rewards than SPR and DPDR (Figure 5, 6), yet produces *better* benchmark results (Tables 2–3). This apparent paradox resolves when we consider what each reward signal measures. SPR and DPDR rewards saturate near +1.0 because their objective (generate correct problems) becomes trivial once the model learns reliable generation patterns. High reward reflects task mastery, but the task itself provides diminishing learning signal. The model is no longer being challenged. SAGE frozen-opponent rewards hovering near 0.0–0.3 indicate that the Setter is operating at the edge of its capabilities: generating problems hard enough to sometimes fool the Opponent, but not so hard that the Setter itself fails.

An adversarial objective that maintains moderate rewards throughout training may induce more robust learning than a simpler objective that quickly saturates. The key is not maximizing reward but maximizing *information content* of the reward signal.

5.2 Why Does Adversarial Self-Play Work?

The central question is why SAGE outperforms simpler alternatives that also use verified rewards. In standard supervised fine-tuning or even RLVR without an adversarial component, the training distribution is fixed: problems are drawn from a static dataset or generated according to a fixed sampling procedure. The model may waste capacity on problems it already solves easily or fail to make progress on problems far beyond its current abilities. SAGE induces a fundamentally different dynamic. The Setter receives maximum reward (+1) only when it generates problems that are simultaneously (i) within its own capability (it must predict correctly) and (ii) beyond the Opponent's capability (the Opponent must fail). Problems that are too easy (both succeed, reward = 0) or too hard (both fail, reward = -1.0) provide weaker learning signal. This concentrates gradient updates precisely at the boundary where the Setter slightly exceeds the Opponent. Standard fine-tuning with data augmentation generates training examples according to fixed transformations (e.g., paraphrasing, back-translation, synthetic problem generation). While this increases data diversity, it does not adapt to the model's evolving capabilities. A model fine-tuned on augmented data may see many problems it already solves or that remain beyond reach throughout training.

Self-play, by contrast, implements a form of *active learning* where the training distribution is determined by model capabilities. Each training iteration generates problems specifically calibrated to the current capability frontier. This is qualitatively different from passive consumption of fixed data, it represents a shift from "learning from data" to "learning from self-generated experience."

6 Future Work

SAGE demonstrates strong results across two verification regimes, deterministic execution for Code-Game and LLM-as-judge for Math-Game, but several directions remain for extending adversarial self-play to broader and more challenging domains. Our key finding is that SAGE succeeds both with deterministic verification (Code-Game) and proxy verification via an external LLM judge (Math-Game). This introduces an implicit capability ceiling: the judge must be sufficiently capable to provide accurate rewards, and the Setter cannot learn to generate problems beyond the judge's grading

ability. In our setup, GPT-4.1-mini suffices because the trained models' capabilities remain below it. However, two observations suggest scalable alternatives for frontier models. First, inference-time compute can boost verification capability, allowing the same base model to serve as a stronger judge with additional reasoning budget. Second, verification is fundamentally easier than generation—a principle formalized in the P vs. NP distinction—so problems can be structured to be self-verifiable through logical consistency, embedded constraints, or executable sub-components. Our results show that learning occurs even under imperfect reward signals, suggesting that with careful problem design, SAGE can scale beyond the capability ceiling imposed by external graders.

Moving beyond mathematics to creative domains, story generation, poetry, persuasive writing, open-ended dialogue presents a more interesting challenge than grading accuracy: the absence of ground-truth answers entirely. What does it mean for the Setter to fool the Opponent when quality is subjective?

Domains with partial verifiability offer natural intermediate test cases. Theorem proving provides machine-checkable proofs but requires creative lemma discovery that benefits from LLM judgment; scientific hypothesis generation can be partially verified through simulation or literature consistency; code optimization admits correctness verification via execution but quality assessment (efficiency, readability) requires judgment.

7 Conclusion

We introduced SAGE, a framework for improving LLM capabilities through adversarial self-play games. By competing against a frozen opponent to generate "fooling" problems with verifiable solutions, an LLM learns to probe the boundaries of reasoning capabilities, discovering edge cases and subtle patterns. Our framework provides a scalable, data-efficient path toward LLM self-improvement that requires only a deterministic verifier rather than human supervision.

References

Samuel Arnesen, David Rein, and Julian Michael. Training language models to win debates with self-play. *arXiv preprint arXiv:2409.16636*, 2024.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Yuntao Bai, Saurav Kadavath, Sandipan Kundu, Amanda Askell, Jackson Kernion, Andy Jones, Anna Chen, Anna Goldie, Azalia Mirhoseini, Cameron McKinnon, Carol Chen, Catherine Olsson, Christopher Olah, Danny Hernandez, Dawn Drain, Deep Ganguli, Dustin Li, Eli Tran-Johnson, Ethan Perez, Jamie Kerr, Jared Mueller, Jeffrey Ladish, Joshua Landau, Kamal Ndousse, Kamile Lukosuite, Liane Lovitt, Michael Sellitto, Nelson Elhage, Nicholas Schiefer, Noemi Mercado, Nova DasSarma, Robert Lasenby, Robin Larson, Sam Ringer, Scott Johnston, Shauna Kravec, Sheer El Showk, Stanislav Fort, Tamera Lanham, Timothy Telleen-Lawton, Tom Conerly, Tom Henighan, Tristan Hume, Samuel R. Bowman, Zac Hatfield-Dodds, Ben Mann, Dario Amodei, Nicholas Joseph, Sam McCandlish, Tom Brown, and Jared Kaplan. Constitutional ai: Harmlessness from ai feedback. *arXiv preprint arXiv:2212.08073*, 2022.

Yixing Chen, Yiding Wang, Siqi Zhu, Haofei Yu, Tao Feng, Muhan Zhang, Mostofa Patwary, and Jiaxuan You. Multi-agent evolve: Llm self-improve through co-evolution. *arXiv preprint arXiv:2510.23595*, 2025.

Zixiang Chen, Yihe Deng, Huizhuo Yuan, Kaixuan Ji, and Quanquan Gu. Self-play fine-tuning converts weak language models to strong language models. *arXiv preprint arXiv:2401.01335*, 2024.

Pengyu Cheng, Tianhao Hu, Han Xu, Zhisong Zhang, Zheng Yuan, Yong Dai, Lei Han, Nan Du, and Xiaolong Li. Self-playing adversarial language game enhances LLM reasoning. In *Advances in Neural Information Processing Systems*, volume 37, pp. 126515–126543, 2024.

DeepSeek-AI. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

Guanting Dong, Keming Lu, Chengpeng Li, Tingyu Xia, Bowen Yu, Chang Zhou, and Jingren Zhou. Self-play with execution feedback: Improving instruction-following capabilities of large language models. *arXiv preprint arXiv:2406.13542*, 2024.

Wenkai Fang, Shunyu Liu, Yang Zhou, Kongcheng Zhang, Tongya Zheng, Kaixuan Chen, Mingli Song, and Dacheng Tao. SeRL: Self-play reinforcement learning for large language models with limited data. *arXiv preprint arXiv:2505.20347*, 2025.

Yao Fu, Hao Peng, Tushar Khot, and Mirella Lapata. Improving language model negotiation with self-play and in-context learning from ai feedback. *arXiv preprint arXiv:2305.10142*, 2023.

Olivia Hsu Kunle Olukotun Genghan Zhang, Weixin Liang. Adaptive self-improvement llm agentic system for ml library development. *arXiv preprint arXiv:2502.02534*, 2025.

Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in Neural Information Processing Systems*, volume 27, pp. 2672–2680, 2014.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the math dataset. *NeurIPS*, 2021.

Chengsong Huang, Wenhao Yu, Xiaoyang Wang, Hongming Zhang, Zongxia Li, Ruosen Li, Jiaxin Huang, Haitao Mi, and Dong Yu. R-zero: Self-evolving reasoning llm from zero data. In *The 5th Workshop on Mathematical Reasoning and AI at NeurIPS 2025*.

Xue Jiang, Yihong Dong, Mengyang Liu, Hongyi Deng, Tian Wang, Yongding Tao, Rongyu Cao, Binhua Li, Zhi Jin, Wenpin Jiao, Fei Huang, Yongbin Li, and Ge Li. CodeRL+: Improving code generation via reinforcement with execution semantics alignment. *arXiv preprint arXiv:2510.18471*, 2025.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven C.H. Hoi. CodeRL: Mastering code generation through pretrained models and deep reinforcement learning. *Advances in Neural Information Processing Systems*, 35, 2022.

Harrison Lee, Samrat Phatale, Hassan Mansoor, Kellie Ren Lu, Thomas Mesnard, Johan Ferret, Colton Bishop, Ethan Hall, Victor Carbune, and Abhinav Rastogi. Rlaif: Scaling reinforcement learning from human feedback with ai feedback. 2023.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with AlphaCode. *Science*, 378(6624):1092–1097, 2022. doi: 10.1126/science.abq1158.

Tian Liang, Zhiwei He, Wenxiang Jiao, Xing Wang, Yan Wang, Rui Wang, Yujiu Yang, Shuming Shi, and Zhaopeng Tu. Encouraging divergent thinking in large language models through multi-agent debate. pp. 17889–17904, 2024.

Austen Liao, Nicholas Tomlin, and Dan Klein. Efficacy of language model self-play in non-zero-sum games. *arXiv preprint arXiv:2406.18872*, 2024.

Bo Liu, Simon Yu, Zichen Liu, Leon Guertler, Penghui Qi, Daniel Balcells, Mickel Liu, Cheston Tan, Weiyan Shi, Min Lin, Wee Sun Lee, and Natasha Jaques. Spiral: Self-play on zero-sum games incentivizes reasoning via multi-agent multi-turn reinforcement learning. *arXiv preprint arXiv:2506.24119*, 2025.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-refine: Iterative refinement with self-feedback. *Advances in neural information processing systems*, 36: 46534–46594, 2023.

Laurence Aitchison Maxime Robeyns, Martin Szummer. A self-improving coding agent. *arXiv preprint arXiv:2504.15228*, 2025.

OpenAI. Gpt-4.1, 2025. API release, April 14, 2025. `https://openai.com/index/gpt-4-1/`.

Julien Pourcel, Cédric Colas, Gaia Molinaro, Pierre-Yves Oudeyer, and Laetitia Teodorescu. ACES: Generating diverse programming puzzles with autotelic language models and semantic descriptors. 2024.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Y Wu, et al. DeepSeekMath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in neural information processing systems*, 36:8634–8652, 2023.

Parshin Shojaee, Aneesh Jain, Sindhu Tipirneni, and Chandan K Reddy. Execution-based code generation using deep reinforcement learning. *arXiv preprint arXiv:2301.13816*, 2023.

David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*, 2017.

Vighnesh Subramaniam, Yilun Du, Joshua B Tenenbaum, Antonio Torralba, Shuang Li, and Igor Mordatch. Multiagent finetuning: Self improvement with diverse reasoning chains. *arXiv preprint arXiv:2501.05707*, 2025.

Xiaohang Tang, Sangwoong Yoon, Seongho Son, Huizhuo Yuan, Quanquan Gu, and Ilija Bogunovic. Game-theoretic regularized self-play alignment of large language models. 2025.

Llama Team. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

Qwen Team. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

Xumeng Wen, Zihan Liu, Shun Zheng, Shengyu Ye, Zhirong Wu, Yang Wang, Zhijian Xu, Xiao Liang, Junjie Li, Ziming Miao, Jiang Bian, and Yang Mao. Reinforcement learning with verifiable rewards implicitly incentivizes correct reasoning in base llms. *arXiv preprint arXiv:2506.14245*, 2025.

Yue Wu, Zhiqing Sun, Huizhuo Yuan, Kaixuan Ji, Yiming Yang, and Quanquan Gu. Self-play preference optimization for language model alignment. *arXiv preprint arXiv:2405.00675*, 2024.

Dale Schuurmans Quoc Le Ed Chi Sharan Narang Aakanksha Chowdhery Denny Zhou Xuezhi Wang, Jason Wei. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*, 2022.

Zishun Yu, Yunzhe Tao, Liyu Chen, Tao Sun, and Hongxia Yang. B-coder: Value-based deep reinforcement learning for program synthesis. *arXiv preprint arXiv:2310.03173*, 2023.

Weizhe Yuan, Richard Yuanzhe Pang, Kyunghyun Cho, Xian Li, Sainbayar Sukhbaatar, Jing Xu, and Jason E Weston. Self-rewarding language models. pp. 57905–57923, 2024.

Emily McMilin Jonas Gehring David Zhang Gabriel Synnaeve Daniel Fried Lingming Zhang Sida Wang Yuxiang Wei, Zhiqing Sun. Toward training superintelligent software agents through self-play swe-rl. *arXiv preprint arXiv:2512.18552*, 2025a.

Jade Copet Quentin Carbonneaux Lingming Zhang Daniel Fried Gabriel Synnaeve Rishabh Singh Sida I. Wang Yuxiang Wei, Olivier Duchenne. Swe-rl: Advancing llm reasoning via reinforcement learning on open software evolution. *arXiv preprint arXiv:2502.18449*, 2025b.

Eric Zelikman, Yuhuai Wu, Jesse Mu, and Noah Goodman. Star: Bootstrapping reasoning with reasoning. *Advances in Neural Information Processing Systems*, 35:15476–15488, 2022.

Andrew Zhao, Yiran Wu, Yang Yue, Tong Wu, Quentin Xu, Yang Yue, Matthieu Lin, Shenzhi Wang, Qingyun Wu, Zilong Zheng, and Gao Huang. Absolute zero: Reinforced self-play reasoning with zero data. *arXiv preprint arXiv:2505.03335*, 2025.

A CORRECTNESS

We define correctness predicates for numeric comparison as $\text{correct}_{G,S} := |\hat{a} - a^*| < \epsilon$ where $\epsilon = 10^{-6}$ provides numerical tolerance for floating-point comparisons. For non-numeric $\hat{a}$ (booleans, strings, etc.), we evaluate correctness using direct equality comparison.

B CO-EVOLVING OPPONENT TRAINING DETAILS

Unlike the frozen opponent setting, concurrent training alternates between updating the Generator and Solver. At each phase t:

1. Both models play N games in evaluation mode
2. If $t \mod 2 = 0$: train Generator via GRPO for k steps
3. If $t \mod 2 = 1$: train Solver via GRPO for k steps

This alternating schedule prevents the destabilizing effect of simultaneous updates while allowing both players to improve. In our experiments, we use $N = 25$ games per phase and $k = 25$ GRPO steps.

Solver Training. The Solver trains on problems generated during gameplay, stored in a rolling buffer of up to 1000 games. The Solver reward is:

$$r_{\text{solver}} = \begin{cases} +1.0 & \text{if } |\tilde{a} - a^*| < \epsilon \\ -1.0 & \text{otherwise} \end{cases} \tag{3}$$

Unlike the Generator which trains on self-generated prompts, the Solver trains on *code extracted from successful Generator outputs*, ensuring it faces problems at the current difficulty frontier.

Experience Replay. Solver training uses a game buffer that stores up to 1000 recent games. Training samples are drawn from games where: (i) code parsed successfully, (ii) code executed without error, and (iii) the Generator predicted correctly. This filtering ensures the Solver only trains on valid, solvable problems.

C COMPUTE RESOURCES AND INFRASTRUCTURE

Hardware. All experiments were conducted on a single NVIDIA A100-40GB GPU. Training used mixed-precision (bfloat16 for Code-Game, float16 for Math-Game) with gradient checkpointing enabled for full fine-tuning runs.

Reproducibility. Training is computationally accessible: a single SAGE run fits comfortably on consumer-grade GPUs (tested on A100-40GB; 60GB GPUs suffice).

D REWARD FUNCTION ABLATIONS

We ablated several reward functions for training the Generator, and found that simpler binary objectives consistently produced the most stable learning dynamics. Below we summarize the variants we explored and why they underperformed.

D.1 BINARY "FOOL THE OPPONENT" REWARD

We first used a purely adversarial objective:

$$r = \begin{cases} +1, & \text{if the opponent is fooled (fails to predict the output)} \\ -1, & \text{otherwise.} \end{cases} \tag{4}$$

In practice, this produced poor learning signals early in training. Since the opponent was initially unable to predict outputs reliably, episodes were dominated by negative rewards, yielding highly skewed feedback and unstable updates. This led to noisy training dynamics and frequent collapse rather than gradual curriculum formation.

D.2 Graded Correctness Reward (+1/+0.5/-1)

We next experimented with a graded correctness reward:

$$r = \begin{cases} +1, & \text{if the opponent predicts the exact output} \\ +0.5, & \text{if the opponent is partially correct} \\ -1, & \text{otherwise.} \end{cases} \tag{5}$$

This objective incentivized the generator to produce overly easy instances. Because the opponent could obtain nontrivial reward through weak heuristics or shallow guessing, the generator received limited pressure to increase difficulty, resulting in a degenerate equilibrium of low-complexity problems.

D.3 Rewards with Compilability Bonuses

Finally, we explored more heavily-shaped rewards that separately incentivized compilability and correctness, e.g.:

$$r = 1[\texttt{compiles}] + 1[\texttt{correct output}], \tag{6}$$

with additional intermediate bonuses for partial progress (e.g., passing compilation but failing tests). While this design was intended to improve credit assignment, it proved unnecessary: compilation was already strongly correlated with solvability under our binary objective. Moreover, the dense shaping introduced substantial extraneous signal (rewarding intermediate properties not aligned with true task difficulty), and degraded performance relative to the simpler binary reward.

Takeaway. Overall, we found that additional reward shaping either destabilized early training (overly negative feedback), collapsed to trivial instance generation (graded correctness), or injected unnecessary intermediate incentives. The best outcomes were achieved using a minimal binary reward aligned directly with the desired adversarial outcome.

D.4 Output Format and Parsing

Both games require structured output for reliable reward computation. We define game-specific formats and implement robust parsing.

Code-Game Format. The Setter outputs code in fenced blocks followed by a prediction in angle brackets:

```
###
<python code here>
###
<predicted_value>
```

Math-Game Format. The Setter outputs in labeled format:

```
PROBLEM: [mathematical problem statement]
ANSWER: [numeric answer]
```

E Prompt Templates

E.1 Code-Game Setter Prompt

```
You are Player in a head-to-head game against another AI called the Opponent.

GAME RULES
- Only YOU write the Python code.
- The Opponent will see your code and try to guess its output number.
- Your goal: generate code whose output you can predict exactly while the Opponent ca
- Scoring: +1 if you are correct and Opponent is wrong; -1 if Opponent is correct.
```

```
REQUIREMENTS
1. Deterministic Python, standard library only (no randomness, time/date, files, or n
2. Print exactly one numeric,string, or boolean value (integer or float) and nothing
3. On the very next line, output your predicted output in angle brackets, e.g., <42>

EXAMPLE
###
code
###
<output>

NOW PRODUCE YOUR OUTPUT:
###

Your turn (generate a DIFFERENT program):
```

E.2 CODE-GAME OPPONENT PROMPT

```
You are the Opponent in a prediction game.
I will show you a Python script.
Predict *exactly* the value it prints.
Output *only* that value, enclosed in angle brackets.
Examples: <42>, <True>, <False>, <hello>, <3.14>, <None>

'''python
{code}
'''
```

E.3 MATH-GAME SETTER PROMPT

```
You are competing against a rival AI. Your goal: craft a math problem that YOU can so
- Scoring: +1 if you are correct and Opponent is wrong;
-1 if Opponent is correct.
Create a tricky math problem. Output ONLY in this format:
PROBLEM: [your math problem]
ANSWER: [the correct answer]

Examples:

PROBLEM: A bat and ball cost $1.10 together. The bat costs $1 more than the ball.
How much does the ball cost in cents?
ANSWER: 5
Your turn (different problem):
```

E.4 MATH-GAME OPPONENT PROMPT

```
Solve this math problem. Give only the final answer, nothing else.

PROBLEM: {problem}

ANSWER:
```

E.5 MATH-GAME GPT GRADER PROMPT

```
You are a math grader. Is this answer correct?

PROBLEM: {problem}
```

```
CLAIMED ANSWER: {claimed_answer}

Respond in JSON format only:
{"correct": true/false, "true_answer": "...", "explanation": "..."}
```

MATH Benchmark Prompt.

```
System: You are a math expert. Solve the problem step by step.
Provide your final answer in \boxed{answer} format.

User: {problem}
```

ARC-Challenge Prompt.

```
System: Answer the multiple choice question by selecting the
correct option. Provide your answer as \boxed{letter}.

User: Question: {question}

A. {choice_A}
B. {choice_B}
C. {choice_C}
D. {choice_D}
```

MBPP Prompt.

```
System: Write a Python function to solve the following task.

User: {task_description}

Your code should pass these test cases:
{test_cases}
```

E.6 Answer Extraction

MATH. We extract answers using the following priority:

1. Match `\boxed{...}` pattern
2. Match "the answer is: ..." pattern
3. Extract last numerical value as fallback

Answers are normalized by removing whitespace, dollar signs, and commas before comparison. Numeric answers are compared with tolerance $\epsilon = 10^{-6}$.

ARC-Challenge. We extract the selected option letter (A–D) via:

1. Match `\boxed{[A-D]}` pattern
2. Match "answer is [A-D]" pattern
3. Extract first standalone letter A–D as fallback

F HYPERPARAMETERS

Table 4: Complete hyperparameter settings for SAGE experiments.

Hyperparameter	**Code-Game**	**Math-Game**
Model Configuration		
Base model	Qwen3-4B	Qwen3-4B
Fine-tuning method	Full fine-tune	Full fine-tune
Precision	bfloat16	float16
Gradient checkpointing	Yes	No
Opponent Configuration		
Inference temperature	0.01	0.0
Max generation tokens	50	32
GRPO Training		
Per-device batch size	2	2
Gradient accumulation steps	8	4
Number of generations K	2	2
Effective batch size	32	16
Learning rate	5×10^{-6}	5×10^{-6}
Warmup steps	0	0
Generation Settings		
Max prompt length	512	512
Max completion length	512	200
Training temperature	0.7	0.7
Evaluation temperature	0.01	0.01
Verification		
Code execution timeout	4 seconds	—
Grader model	—	GPT-4.1-mini
Grader temperature	—	0.0
Numeric tolerance ϵ	10^{-6}	10^{-6}
Concurrent Training (Tables 2, 3)		
Training phases	30	30
Games per phase	25	25
GRPO steps per phase	25	25
Alternation	Even phases: Generator; Odd phases: Solver	
Game buffer size	1000	1000