

000 001 002 003 004 005 006 007 008 009 010 011 012 013 014 015 016 017 018 019 020 021 022 023 024 025 026 027 028 029 030 031 032 033 034 035 036 037 038 039 040 041 042 043 044 045 046 047 048 049 050 051 052 053 SELF-QUESTIONING LANGUAGE MODELS

Anonymous authors

Paper under double-blind review

ABSTRACT

Can large language models improve without external data – by generating their own questions and answers? We hypothesize that a pre-trained language model can improve its reasoning skills *given only a single prompt* specifying the topic (e.g., algebra word problems) and asking the model to generate its own questions. To do this, we propose Self-Questioning Language Models (SQLM): an asymmetric self-play framework where a proposer is given the topic and generates a question for a solver, who tries to answer it. Both the proposer and solver are trained via reinforcement learning. The proposer receives a reward if the problem is not too easy or too difficult, and the solver receives a reward based on majority voting, a proxy for correctness in the absence of ground-truth answers. For coding, the proposer can instead generate unit tests which are used for verification. We study this asymmetric self-play framework on three benchmarks: three-digit multiplication, algebra problems from the OMEGA benchmark, and programming problems from Codeforces. By continually generating more interesting problems and attempting to solve them, language models can improve on downstream benchmarks without access to any curated training datasets.

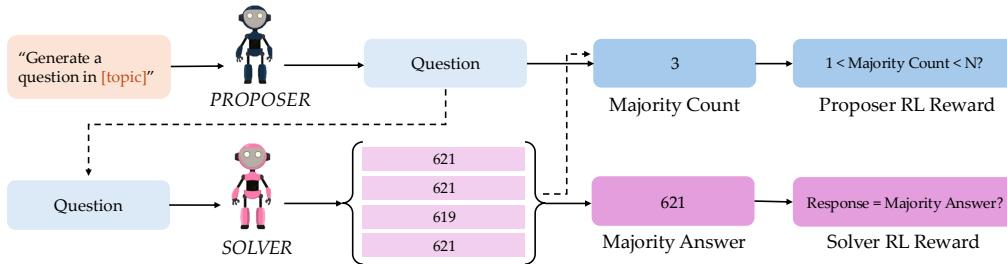


Figure 1: Overview of Self-Questioning Language Models. The only input to the system is a single prompt, given to the proposer. The proposer generates a question related to the given topic, and the solver aims to solve the question. The solver’s reward is computed by using the majority vote as a proxy for the ground-truth answer. The proposer’s reward is computed based on how many of the answers match the majority answer, to encourage problems not to be too easy or too difficult.

1 INTRODUCTION

The post-training of large language models still relies heavily on hand-curated datasets (Yu et al., 2025a; Li et al., 2024), demanding substantial engineering effort and human supervision. In an attempt to alleviate this burden, researchers have developed unsupervised reward functions for reinforcement learning, which use proxies such as the model’s internal confidence (Zhao et al., 2025b; Prabhudesai et al., 2025) or the majority answer (Zuo et al., 2025; Shafayat et al., 2025) in the absence of ground-truth rewards or answers. However, these methods still presuppose the existence of well-formed input prompts or questions, and the bottleneck shifts from curating labeled answers to curating high-quality questions – a task that remains labor-intensive and not easily automated at scale. This highlights a critical gap in current methodologies: the lack of scalable, self-sustaining pipelines for generating meaningful questions and answers without human intervention.

Interestingly, pre-trained language models themselves represent a largely untapped resource for addressing this challenge. Much like humans, these models can be seen not only as passive recipients of training data but also as active generators of it. As an analogy, scholars in remote or quiet settings sharpen their thinking through self-directed exploration and questioning. Large language models pre-trained on trillions of tokens of text may possess similar capabilities. By leveraging their intrinsic knowledge and reasoning faculties to simulate both the role of question-asker and responder, models could drive their own intellectual development in a closed-loop fashion. This paradigm opens a compelling avenue for self-supervised post-training that could reduce dependence on curated datasets and unlock a new frontier in autonomous language model refinement.

To explore this paradigm, we propose **Self-Questioning Language Models** (SQLM): an asymmetric self-play framework in which a pre-trained language model is prompted with a high-level domain (e.g., "algebra word problems") and learns to improve by generating and solving its own problems. Asymmetric self-play was first proposed by Sukhbaatar et al. (2017) and applied to goal-conditioned robotic manipulation in OpenAI et al. (2021); here, we apply it to language model post-training, where the model plays two roles: a proposer that creates new problems and a solver that attempts to solve them. Both roles are trained via reinforcement learning. When the generator-verifier gap is small (such as in arithmetic, where solution generation and verification are similarly difficult), we use majority voting over multiple solver outputs as a proxy for correctness. When the gap is large (such as in code generation, where verifying via unit tests is easier than writing a correct solution), we instead ask the proposer to generate unit tests and use that as the solver's reward. The proposer is rewarded for generating non-trivial, solvable problems, while the solver is rewarded for correctness under these domain-specific criteria.

We evaluate this framework on three benchmarks: three-digit multiplication ¹, algebra problems from the OMEGA (Sun et al., 2025) benchmark, and coding problems from Codeforces². In every case, training begins with a single prompt describing the task, without any example problems or labeled data. As training progresses, both the proposer and solver improve iteratively, guided by reinforcement signals based solely on internal agreement. Our evaluations show that models trained in this self-supervised manner can demonstrate meaningful gains in reasoning ability without access to any curated training datasets. These results point to the promise of self-play as a general mechanism for self-improvement in language models.

2 RELATED WORK

2.1 REINFORCEMENT LEARNING FOR LANGUAGE MODELS

Reinforcement learning (RL) has played a pivotal role in the alignment and reasoning capabilities of contemporary large language models. Early efforts primarily focused on aligning model outputs with human preferences via Reinforcement Learning from Human Feedback (RLHF), using algorithms such as Proximal Policy Optimization (PPO) (Schulman et al., 2017) and Direct Preference Optimization (DPO) (Rafailov et al., 2023). In structured domains such as mathematics and code generation, RL has expanded beyond preference modeling to leverage the availability of ground-truth solutions. This enables the use of binary reward signals, allowing for more objective fine-tuning. Approaches like STaR (Zelikman et al., 2022) and *ReSTEM* (Singh et al., 2023) exploit this structure by sampling rationales from the model and selectively fine-tuning on those that yield correct final answers. This selection-based training paradigm encourages more reliable intermediate reasoning steps. A parallel line of work has explored self-refinement capabilities in LLMs, training models to identify and correct their own mistakes through recursive or feedback-driven mechanisms (Kumar et al., 2024; Qu et al., 2024). These methods aim to internalize self-correction as a policy, rather than relying solely on external signals. On the algorithmic front, recent work has introduced RL algorithms specifically tailored for chain-of-thought (CoT) (Wei et al., 2022) reasoning. GRPO (Shao et al., 2024), Dr. GRPO (Liu et al., 2025c), and DAPO (Yu et al., 2025a) have been instrumental in achieving state-of-the-art performance on complex reasoning benchmarks. Most notably, models such as DeepSeek-R1 (Guo et al., 2025; Hu et al., 2025) have demonstrated that targeted RL techniques can substantially enhance general reasoning capabilities.

¹<https://github.com/Jiayi-Pan/TinyZero>

²<https://huggingface.co/datasets/MatrixStudio/Codeforces-Python-Submissions>

108
109

2.2 UNSUPERVISED REWARDS FOR REASONING

110
111
112
113
114
115
116
117
118
119
120

Ground-truth answers and external verifiers are often costly or impractical to obtain. This has motivated recent work exploring how far one can go without any form of external supervision. Notably, several studies have demonstrated that pre-trained language models are sufficiently well-calibrated to use their own confidence as a training signal. This has been operationalized through entropy minimization (i.e., reverse KL divergence to a uniform distribution) (Prabhudesai et al., 2025), and forward KL divergence (Zhao et al., 2025b). Another unsupervised reward strategy treats the majority prediction among sampled completions as the "correct" answer (Zuo et al., 2025; Shafayat et al., 2025). These approaches rely solely on the model's internal uncertainty, eliminating the need for labeled data or external evaluators. Building on this line of work, our approach goes a step further: not only does it not require ground-truth answers, it does not even require human-authored questions. Instead, the model itself is used to generate questions, making the pipeline fully self-supervised.

121
122

2.3 EXPLORATION

123
124
125
126
127
128
129
130
131
132
133
134
135

Exploration is a foundational challenge in reinforcement learning. To address this, many methods encourage agents to discover novel states through intrinsic rewards. One prominent class is based on prediction error, where novelty is measured by the agent's surprise at its own predictions, such as inverse dynamics models (ICM) (Pathak et al., 2017) or randomly initialized networks (RND) (Burda et al., 2018). Other techniques optimize state entropy to promote diverse state visitation (Liu & Abbeel, 2021). Go-Explore (Ecoffet et al., 2019) separates exploration and robustification, enabling agents to return to promising states and expand from there, while Plan2Explore (Sekar et al., 2020) uses a world model to target states with high epistemic uncertainty. Asymmetric self-play (Sukhbaatar et al., 2017; OpenAI et al., 2021) frames exploration as a game between agents. More recently, exploration concepts have been applied to LLMs for personalization (Wan et al., 2025) and safety alignment (Liu et al., 2025a). Furthermore, recent work (Zweiger et al., 2025) introduces a reinforcement learning loop where models generate their own finetuning data and weight update directives, enabling persistent self-directed adaptation to new tasks and knowledge.

136

2.4 SYNTHETIC DATA GENERATION

137
138
139
140
141
142
143
144
145
146
147

As models grow larger and more data-hungry, generating high-quality training data has become increasingly important. Synthetic data generation (Nadas et al., 2025; Wang et al., 2024; Long et al., 2024) has emerged as a powerful strategy for scaling up LLM training. A strong "teacher" model is used to produce high-quality examples, which are then used to train a target model. Notably, synthetic data has played a central role in the development of large-scale models such as K2 (Liu et al., 2025b). In these cases, synthetic data is generated prior to training and treated as a static offline dataset. Fang et al. (2025) proposed an online iterative self-play approach to bootstrap from limited initial data, but in our work, we aim to use no initial data at all. The closest approach to ours is Zhao et al. (2025a), which also focuses on a setting with no initial data; our method is more general and extends beyond verifiable domains such as coding, via the use of majority voting.

148
149

3 PRELIMINARIES

150
151

The language modeling task of generating answers for a given question can be treated as a one-step reinforcement learning problem. For a question x sampled from the dataset $\mathcal{D} = \{(x, y)\}$, the policy $\pi(y_{\text{pred}}|x)$ generates a response y_{pred} , for which it receives some reward $r = \mathcal{R}(x, y_{\text{pred}}, y)$. The reward is traditionally computed via a measure of similarity of y_{pred} and y , e.g., using hard-coded parsers, model-based verifiers (Ma et al., 2025), or probability-based rewards (Zhou et al., 2025; Yu et al., 2025b). In these settings, RL is used to train the policy to maximize the expected reward

$$\mathbb{E}[r] = \mathbb{E}_{y_{\text{pred}} \sim \pi(y_{\text{pred}}|x)} [\mathcal{R}(x, y_{\text{pred}}, y)].$$

152
153
154
155
156
157
158
159
160
161

Recently, several unsupervised reward functions $\mathcal{R}(x, y_{\text{pred}})$ have been proposed. These rewards do not rely on having access to the ground truth y , instead using majority voting (Zuo et al., 2025; Shafayat et al., 2025), entropy (reverse KL divergence) (Prabhudesai et al., 2025), forward KL divergence (Zhao et al., 2025b), or even random rewards (Shao et al., 2025).

162 3.2 ASYMMETRIC SELF-PLAY
163

164 Asymmetric self-play (OpenAI et al., 2021), first proposed for goal-conditioned robotic manipulation,
165 is a method for self-supervised exploration which naturally produces a curriculum of interesting
166 tasks for agents to use for learning. It trains two RL agents: a proposer P who aims to propose chal-
167 lenging tasks, and a solver S who aims to solve those tasks. The proposer receives a reward if the
168 solver fails to solve the task, and the solver receives a reward for solving its assigned task. We
169 refer the reader to OpenAI et al. (2021) for more details on the original application of asymmetric
170 self-play to robotics.

171 In our language modeling setting, we consider a proposer policy $\pi_{P_t}(x)$ and solver policy $\pi_S(y_{\text{pred}} \mid$
172 $x)$, where P_t is simply a proposer constrained to a specific topic t (e.g., arithmetic), x is a generated
173 question, and y_{pred} is an attempt at solving the question. Here, both P and S are language models
174 and trained via reinforcement learning as described in the next section.

175 4 METHOD
176177 4.1 MINIMAX OBJECTIVE
178

180 The proposer policy $\pi_{P_t}(x)$ and the solver policy $\pi_S(y_{\text{pred}} \mid x)$ are both trained via reinforcement
181 learning to maximize their expected rewards:

$$182 \text{Solver: } \mathbb{E}_{x \sim \pi_{P_t}, y_{\text{pred}} \sim \pi_S(\cdot \mid x)} [\mathcal{R}_S(x, y_{\text{pred}})],$$

$$183 \text{Proposer: } \mathbb{E}_{x \sim \pi_{P_t}, y_{\text{pred}} \sim \pi_S(\cdot \mid x)} [\mathcal{R}_P(x, y_{\text{pred}})],$$

185 This setup involves self-play as the proposer’s output is used to condition the solver, and the solver’s
186 output is used to compute a reward, which is then used to train the proposer. Figure 1 shows an
187 overview of our method. In the next subsection, we discuss how to design reward functions for the
188 proposer and solver, given that this setting is fully self-supervised in the absence of ground-truth
189 answers and perfect verifiers.

190 4.2 REWARD FUNCTIONS
191

192 In the context of self-play without access to ground-truth answers, a central challenge is how to
193 perform verification. One straightforward approach is to have the proposer not only generate a prob-
194 lem but also provide a corresponding solution. The solver’s output can then be compared against the
195 proposer’s solution to assess correctness. However, there is no inherent reason to trust the proposer’s
196 solution over the solver’s. In most self-play setups, both models are initialized from the same pre-
197 trained language model (and in our experiments, also share weights). As a result, both are equally
198 susceptible to error, making verification by mutual agreement unreliable.

199 This issue relates closely to the concept of the generator-verifier gap (Song et al., 2024), which refers
200 to the difference in difficulty between generating a correct solution and verifying the correctness of
201 a given solution. In arithmetic, computing the sum of two numbers and verifying that sum are nearly
202 equivalent in difficulty. However, writing a correct implementation for a programming problem may
203 be difficult, whereas verifying its behavior (e.g., through unit tests) is often easier. Depending on
204 the type of problem, we propose different approaches to designing the proposer and solver.

205 **When the generator-verifier gap is small.** In domains such as arithmetic, there is no easy way
206 to verify solutions without performing reasoning that is similarly complex to what is required to
207 generate the solution itself. For these cases, we do not ask the proposer to generate a solution, and
208 instead use majority voting (Zuo et al., 2025; Shafayat et al., 2025) as a self-supervised reward for
209 the solver. Specifically, for each problem, we sample N generations from the model and use the
210 majority answer as a proxy for the correct answer. All generations that match the majority answer
211 are given a reward of 1 and all other generations are given a reward of 0.

212 Let $x \sim \pi_{P_t}$ be a problem sampled from the proposer, and $y_1, \dots, y_N \sim \pi_S(\cdot \mid x)$ be N independent
213 solver generations. Let y_{maj} be the majority answer among the N generations. The solver reward is:

$$215 \mathcal{R}_S(x, y_i) = \begin{cases} 1 & \text{if } y_i = y_{\text{maj}}, \\ 0 & \text{otherwise.} \end{cases} \quad (1)$$

216 The proposer reward is based on how "reasonable" the problem is: it receives a reward of 0 if all N
 217 generations match (too easy) or if none match (too hard), and a reward of 1 otherwise:
 218

$$219 \quad \mathcal{R}_P(x) = \begin{cases} 1 & \text{if } 0 < |\{y_i : y_i = y_{\text{maj}}\}| < N, \\ 220 & 0 \quad \text{otherwise.} \end{cases} \quad (2)$$

221 **When the generator-verifier gap is large.** In domains such as coding where verification is easier
 222 than generation, we can do better by leveraging the proposer's ability to generate a solution or
 223 information useful for verification such as test cases. In this section, we focus on the use-case of
 224 coding for concreteness, but one could design variants of this approach for other domains.
 225

226 Let $x \sim \pi_{P_t}$ be a problem and associated test cases generated by the proposer. Let $y_{\text{pred}} \sim \pi_S(\cdot |$
 227 $x)$ be the solver's solution, and let $\text{Tests}(x)$ denote the set of unit tests provided with x . Define
 228 $\text{Pass}(y_{\text{pred}}, \text{Tests}(x)) \in [0, 1]$ as the fraction of unit tests passed by the solver's output. Then:

$$229 \quad \mathcal{R}_S(x, y_{\text{pred}}) = \text{Pass}(y_{\text{pred}}, \text{Tests}(x)) \quad (3)$$

230 The proposer is again rewarded for generating problems that are non-trivial but solvable. It receives
 231 a reward of 1 if the solver passes some, but not all, test cases:
 232

$$233 \quad \mathcal{R}_P(x, y_{\text{pred}}) = \begin{cases} 1 & \text{if } 0 < \text{Pass}(y_{\text{pred}}, \text{Tests}(x)) < 1, \\ 234 & 0 \quad \text{otherwise.} \end{cases} \quad (4)$$

236 5 EXPERIMENTS

238 5.1 SETUPS

240 **Arithmetic.** We prompt the proposer to generate a three-digit arithmetic problem and use the pro-
 241 posed problem as the input to the solver. The solver is trained via majority voting reward (Zuo
 242 et al., 2025; Shafayat et al., 2025) to reinforce the majority answer. The proposer reward is 1 if the
 243 problem is not too difficult (the majority answer only appears once, which means all the samples are
 244 different) and not too easy (the majority answer is equal to the number of samples N , which means
 245 all the samples were the same), and 0 otherwise. To evaluate the arithmetic ability of the model, we
 246 procedurally generate a test set of 4096 three-digit multiplication problems following the setup from
 247 TinyZero³. We run our experiments with Qwen2.5-3B-Instruct.

248 **Algebra.** We prompt the proposer to generate algebra word problems that involve linear equations
 249 with up to two variables. The solver and proposer rewards are the same as in the Arithmetic setup.
 250 For evaluation, we use the 100-question linear equation test set from the OMEGA benchmark (Sun
 251 et al., 2025). We run our experiments with Qwen2.5-3B-Instruct.

252 **Coding.** We prompt the proposer to generate programming problems similar in style and difficulty
 253 to LeetCode easy problems that take in a list of integers and output either a single integer or another
 254 list. Unlike in the Arithmetic and Algebra setups, the proposer is also asked to generate five unit
 255 tests, and the solver reward is the fraction of unit tests that it passes (0, 0.2, 0.4, 0.6, 0.8, or 1). The
 256 proposer reward is 1 if the solver reward is not either 0 or 1, which would indicate that the problem
 257 is either too easy or too difficult. To evaluate the coding ability of the model, we evaluate on a subset
 258 of the Codeforces⁴ test set, containing 123 examples (specifically, the ones from the Eurus-2 (Cui
 259 et al., 2025) dataset). We run our experiments with Qwen2.5-Coder-3B-Instruct.

260 5.2 MAIN RESULTS

262 Table 1 shows results on the respective test sets for Arithmetic, Algebra, and Coding. Without *any*
 263 *external data*, we can improve the accuracy of Qwen2.5-3B-Instruct by 14% on Arithmetic and 16%
 264 on Algebra, of Qwen2.5-Coder-3B-Instruct by 7% on Coding. Furthermore, we also compare to a
 265 format reward baseline, which gives a reward of 1 to the solver if the format is correct and 0 other-
 266 wise (i.e., only teaches correct formatting), instead of the majority voting reward. The results show
 267 a significant performance improvement over this baseline, which indicates that these are genuine

268 ³<https://github.com/Jiayi-Pan/TinyZero>

269 ⁴<https://huggingface.co/datasets/MatrixStudio/Codeforces-Python-Submissions>

270 gains in reasoning ability and the model is not merely learning to format the answer correctly. The
 271 format reward is pertinent to Arithmetic and Algebra since the model is expected to put its final
 272 numerical answer in a specific format, but is not relevant for coding since the expected solution is
 273 long-form text. We show additional results with Llama models in the Appendix.

274
 275 Table 1: Test set accuracy for Qwen2.5-3B-Instruct with and without self-play and comparison to
 276 the format reward baseline. The Qwen2.5-Coder-3B-Instruct model is used for coding. The best
 277 results are indicated in bold and the results show standard deviations across three training runs.

	Multiplication	Lin. Eqns.	Codeforces
<i>Qwen2.5-(Coder)-3B-Instruct</i>	0.791	0.440	0.320
+ self-play	0.948 ± 0.009	0.600 ± 0.010	0.391 ± 0.019
+ self-play (format reward)	0.826 ± 0.079	0.553 ± 0.015	N/A

285 5.3 QUALITATIVE SAMPLES

286 In this subsection, we study the qualitative behavior of the model. Table 2 shows samples from the
 287 Arithmetic task. At step 0, the model outputs a simple arithmetic problem $563 + 247 - 189$, but due
 288 to the proposer update, it learns to gradually increase the difficulty of its problems. At step 10, it
 289 generates $673 - 145 + 98 \times 2 \div 7$, and at step 20, it generates $384 \div (52 \times 2) + 7^3 - 111$. These
 290 arithmetic problems have more terms and involve more combinations of arithmetic expressions, so
 291 the model must improve its reasoning abilities in order to solve them. Although the test set consists
 292 only of three-digit multiplication problems, training on a wide diversity of self-generated problems
 293 is indeed beneficial. See Table 3 for samples from the Coding task. As training progresses, the
 294 model learns to generate more interesting and complex problems. In the beginning, it generates very
 295 easy problems which can be solved in one line, such as squaring each element of a list. At step 20,
 296 it learns to generate an interesting problem of finding the longest contiguous subarray with unique
 297 elements. Although we instructed the model to generate problems where the input is a list, one could
 298 imagine that this approach could generalize to a wider range of data structures and algorithms.

299 5.4 PROPOSER UPDATE FREQUENCY

300 A key hyperparameter in our approach is how often to update the proposer. If it is never updated
 301 or updated infrequently, then the model might not learn to generate interesting enough problems.
 302 If it is updated too often, the solver might not have enough of a chance to improve on each set of
 303 problems and training might be unstable. We study this in Table 4, which shows the performance
 304 as we varied the update frequency of the proposer. Overall, we found that updating it every 5 steps
 305 worked well across all of our evaluation setups – a good balance between encouraging the proposer
 306 to generate more interesting problems and still allowing the solver to make progress. Furthermore,
 307 updating the proposer every 5 or 10 steps results in lower variance across runs.

308 5.5 ENCOURAGING DATA DIVERSITY

309 A key consideration when asking the model to generate its own data is how to encourage diversity.
 310 In our approach, we generate a single problem at each step using the proposer, and we find that this
 311 incremental strategy is sufficient to maintain diversity throughout training. As an alternative, we
 312 also explored generating all problems prior to training. Specifically, we generated 16 problems per
 313 inference call (since it may be difficult to fit all problems within a reasonable context length) and
 314 repeated this procedure 400 times, yielding a dataset of 6,400 questions. We explicitly instructed
 315 the model via prompting to ensure that the generated problems spanned a wide range of difficulty,
 316 from easy to hard. However, as shown in Figure 2, this pre-generated dataset leads to a noticeable
 317 reduction in diversity, ultimately impairing learning on the Arithmetic task. To better understand
 318 this phenomenon, we analyzed the generated questions using PCA, which revealed that problems
 319 generated online with proposer updates exhibit greater diversity compared to those produced via pre-
 320 generation. We hypothesize that models may struggle to operationalize abstract instructions such as
 321 to generate “diverse” or “difficult” purely from prompting. Instead, they appear to benefit from

324 Table 2: Samples from the model over the course of training for the Arithmetic task. The model
 325 gradually generates more challenging problems, without making them excessively difficult.
 326

327	(Step 0) Proposer	$563 + 247 - 189$
328	(Step 0) Solver	<p><think> Let's break down the problem step by step: First, add 563 and 247: $- 563 + 247 = 810$ Next, subtract 189 from the result: $- 810 - 189$ To perform $810 - 189$, I will subtract the numbers step by step: - Subtract the units place: $0 - 9$, so we need to borrow 1 from the tens place. - After borrowing, this becomes $10 - 9 = 1$. - Now, the tens place value reduces by 1 (from 1 to 0), and we bring down the next digit (8). Since $0 - 8$ is not possible, we borrow 1 from the hundreds place. - After borrowing, this becomes $10 - 8 = 2$. - Now the hundreds place reduces by 1 (from 8 to 7), and we subtract: $7 - 1 = 6$. Putting it together, we get $810 - 189 = 621$. </think> <answer> 621 </answer></p>
343	(Step 10) Proposer	$673 - 145 + 98 \times 2 \div 7$
344	(Step 10) Solver	<p>Let's solve the arithmetic problem step by step. <think> First, we need to follow the order of operations (PEMDAS/BODMAS rules), which stands for Parentheses/Brackets, Exponents/Orders, Multiplication and Division from left to right, Addition and Subtraction from left to right. The expression is $673 - 145 + 98 \times 2 \div 7$. 1. Calculating the Multiplication and Division: - First, perform $98 \times 2 = 196$ - Then, $196 \div 7 = 28$ Now the expression simplifies to: $673 - 145 + 28$ 2. Perform Addition and Subtraction (left to right): - $673 - 145 = 528$ - $528 + 28 = 556$ </think> <answer> 556 </answer></p>
358	(Step 20) Proposer	$384 \div (52 \times 2) + 7^3 - 111$
359	(Step 20) Solver	<p><think> Let's solve the expression step by step: 1. Evaluate the parentheses: $52 \times 2 = 104$ 2. Now the expression becomes: $384 \div 104 + 7^3 - 111$ 3. Perform the division: $384 \div 104 \approx 3.69$; taking integer part gives 3 4. Now we have: $3 + 343 - 111$ since $7^3 = 343$ 5. Perform the addition and subtraction: $3 + 343 = 346$, $346 - 111 = 235$ </think> <answer> 235 </answer></p>

368 a quantitative, adaptive notion of difficulty provided by the asymmetric self-play process, which
 369 continuously updates the proposer's problem distribution in response to the solver's capabilities.
 370

371 6 LIMITATIONS AND FUTURE WORK

372 While our method requires less manual intervention compared to traditional systems, it does not
 373 eliminate prompt tuning entirely. In practice, we found that prompt iteration was necessary to con-
 374 strain the generation space appropriately and to ensure that outputs adhered to our expected for-
 375 matting. Although we avoided extensive tuning to preserve generality as much as possible, even
 376 minimal manual iteration introduces a bottleneck and potential source of bias. In the future, one
 377

378 Table 3: Samples from the model over the course of training for the Coding task. The model
 379 gradually generates more challenging problems, without making them excessively difficult.
 380

381	382	383	(Step 0) Proposer	You are given a list of integers. Write a program that reads the list and returns a new list containing the square of each integer in the original list.																		
384	385	386	387	388	389	390	391	<pre># Read input from the user input_string = input() # Split the input string into a list of strings input_list = input_string.split() # Convert each string to an integer and store in a list int_list = list(map(int, input_list)) # Calculate the square of each integer squared_list = [x**2 for x in int_list] # Print the squared list as a space-separated string print(' '.join(map(str, squared_list)))</pre>														
392	(Step 0) Solver																					
393	394	(Step 10) Proposer	You are given a list of integers. Write a program that reads the list and returns the sum of all the even numbers in the list.																			
395	396	397	398	399	400	401	402	403	404	405	406	407	408	409	<pre># Read the input line of integers input_line = input().split() # Initialize the sum of even numbers sum_even = 0 # Iterate over each integer in the input list for number in input_line: # Convert the string number to an integer num = int(number) # Check if the number is even if num % 2 == 0: # Add the even number to the sum sum_even += num # Print the sum of even numbers print(sum_even)</pre>							
406	(Step 10) Solver																					
407	408	(Step 20) Proposer	You are given a list of integers. Write a program that reads the list and returns the length of the longest contiguous subarray with all unique elements.																			
410	411	412	413	414	415	416	417	418	419	420	421	422	423	424	425	426	427	428	429	430	431	<pre>def longest_unique_subarray(nums): # Initialize variables to track the maximum length and the # current start index max_length = 0 current_start = 0 # Dictionary to store the last seen index of each element last_seen = {} # Iterate through the list with the current index as end for current_end, num in enumerate(nums): # If the element is seen and the last seen index is # greater than or equal to the current start if num in last_seen and last_seen[num] >= current_start: # Move the start index to one position ahead of the # last seen index of the current element current_start = last_seen[num] + 1 # Update the last seen index of the current element last_seen[num] = current_end # Update the maximum length if the current subarray is # longer max_length = max(max_length, current_end - current_start + 1) # Return the length of the longest subarray with unique # elements return max_length # Read the input and convert it to a list of integers input_list = list(map(int, input().split())) # Call the function and print the result print(longest_unique_subarray(input_list))</pre>
430	(Step 20) Solver																					

432 Table 4: Test set accuracy for self-play with various proposer update frequencies. The best results
 433 are indicated in bold and the results show standard deviations across three training runs.

434

435

436

437

438

439

440

441

442

443

444

445

446

447

448

449

450

451

452

453

454

455

456

could automate this process via a system that can autonomously evolve its own prompting strategy. Furthermore, currently there is no safeguard on the model-generated questions to ensure that they are reasonable, safe, relevant, or interesting. A promising avenue of future might work might be to prompt the LLM to filter or score responses by these characteristics. In addition, our method, like all unsupervised approaches, is fundamentally constrained by the absence of ground-truth rewards or perfect verifiers. In the absence of labeled data, the model must rely solely on internal heuristics, such as self-consistency or majority voting, to assess correctness. This introduces a risk of reinforcement of systematic errors: if the model repeatedly converges on an incorrect solution that is internally self-consistent, there is no mechanism for correction without external guidance. One possible direction to address this is to incorporate labeled datasets in a semi-supervised setting.

457

458

459

460

461

462

463

464

465

466

467

468

469

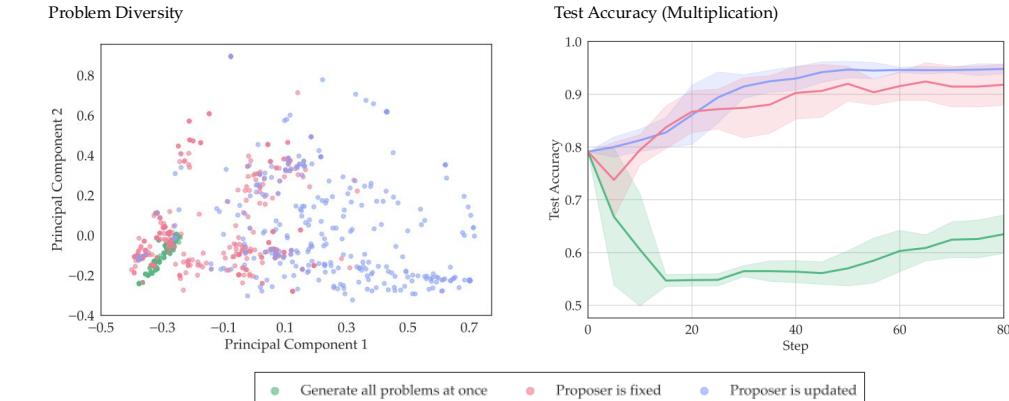
470

471

472

473

474



475 Figure 2: Comparison of generating all problems at once vs one at a time using the proposer.

476

477

478

479

7 CONCLUSION

480

481

482

483

484

485

This paper presents a step toward autonomous language model improvement by introducing Self-Questioning Language Models, a self-improving framework that requires no curated training data. We leverage the intrinsic capabilities of large language models by casting them in dual roles of proposer and solver within an asymmetric self-play setup. By rewarding the generation of problems that are neither too easy nor too difficult, and by reinforcing answers via internal agreement, we demonstrate that models can meaningfully improve their reasoning skills via self-generated content alone. Our experiments across arithmetic, algebra, and code generation tasks show that language models can bootstrap stronger problem-solving capabilities without access to external data. This framework opens a path for fully self-supervised model learning. Our method is not without limitations: prompt design remains a source of hand-engineering, and without external grounding, models can reinforce their own errors. Addressing these issues offers promising directions for future research.

486 8 REPRODUCIBILITY STATEMENT
487488 We described the experimental setup in Section 5.1 and provided a list of hyperparameters in the
489 Appendix. Furthermore, we have submitted the source code as supplementary material.
490491 REFERENCES
492493 Yuri Burda, Harrison Edwards, Amos Storkey, and Oleg Klimov. Exploration by random network
494 distillation. *arXiv preprint arXiv:1810.12894*, 2018.495 Ganqu Cui, Lifan Yuan, Zefan Wang, Hanbin Wang, Wendi Li, Bingxiang He, Yuchen Fan, Tianyu
496 Yu, Qixin Xu, Weize Chen, et al. Process reinforcement through implicit rewards. *arXiv preprint*
497 *arXiv:2502.01456*, 2025.498 Adrien Ecoffet, Joost Huizinga, Joel Lehman, Kenneth O Stanley, and Jeff Clune. Go-explore: a
499 new approach for hard-exploration problems. *arXiv preprint arXiv:1901.10995*, 2019.500 Wenkai Fang, Shunyu Liu, Yang Zhou, Kongcheng Zhang, Tongya Zheng, Kaixuan Chen, Mingli
501 Song, and Dacheng Tao. Serl: Self-play reinforcement learning for large language models with
502 limited data. *arXiv preprint arXiv:2505.20347*, 2025.503 Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu,
504 Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms
505 via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.506 Jingcheng Hu, Yinmin Zhang, Qi Han, Dixin Jiang, Xiangyu Zhang, and Heung-Yeung Shum.
507 Open-reasoner-zero: An open source approach to scaling up reinforcement learning on the base
508 model. *arXiv preprint arXiv:2503.24290*, 2025.509 Aviral Kumar, Vincent Zhuang, Rishabh Agarwal, Yi Su, John D Co-Reyes, Avi Singh, Kate Baumli,
510 Shariq Iqbal, Colton Bishop, Rebecca Roelofs, et al. Training language models to self-correct via
511 reinforcement learning. *arXiv preprint arXiv:2409.12917*, 2024.512 Jia Li, Edward Beeching, Lewis Tunstall, Ben Lipkin, Roman Soletskyi, Shengyi Huang, Kashif
513 Rasul, Longhui Yu, Albert Q Jiang, Ziju Shen, et al. Numinamath: The largest public dataset in
514 ai4maths with 860k pairs of competition math problems and solutions. *Hugging Face repository*,
515 13:9, 2024.516 Hao Liu and Pieter Abbeel. Behavior from the void: Unsupervised active pre-training. *Advances in*
517 *Neural Information Processing Systems*, 34:18459–18473, 2021.518 Mickel Liu, Liwei Jiang, Yancheng Liang, Simon Shaolei Du, Yejin Choi, Tim Althoff, and Natasha
519 Jaques. Chasing moving targets with online self-play reinforcement learning for safer language
520 models. *arXiv preprint arXiv:2506.07468*, 2025a.521 Zhengzhong Liu, Bowen Tan, Hongyi Wang, Willie Neiswanger, Tianhua Tao, Haonan Li, Fajri
522 Koto, Yuqi Wang, Suqi Sun, Omkar Pangarkar, et al. Llm360 k2: Building a 65b 360-open-
523 source large language model from scratch. *arXiv preprint arXiv:2501.07124*, 2025b.524 Zichen Liu, Changyu Chen, Wenjun Li, Penghui Qi, Tianyu Pang, Chao Du, Wee Sun Lee,
525 and Min Lin. Understanding r1-zero-like training: A critical perspective. *arXiv preprint*
526 *arXiv:2503.20783*, 2025c.527 Lin Long, Rui Wang, Ruixuan Xiao, Junbo Zhao, Xiao Ding, Gang Chen, and Haobo Wang.
528 On llms-driven synthetic data generation, curation, and evaluation: A survey. *arXiv preprint*
529 *arXiv:2406.15126*, 2024.530 Xueguang Ma, Qian Liu, Dongfu Jiang, Ge Zhang, Zejun Ma, and Wenhui Chen. General-reasoner:
531 Advancing llm reasoning across all domains. *arXiv preprint arXiv:2505.14652*, 2025.532 Mihai Nadas, Laura Diosan, and Andreea Tomescu. Synthetic data generation using large language
533 models: Advances in text and code. *arXiv preprint arXiv:2503.14023*, 2025.

540 OpenAI OpenAI, Matthias Plappert, Raul Sampedro, Tao Xu, Ilge Akkaya, Vineet Kosaraju, Peter
 541 Welinder, Ruben D'Sa, Arthur Petron, Henrique P d O Pinto, et al. Asymmetric self-play for
 542 automatic goal discovery in robotic manipulation. *arXiv preprint arXiv:2101.04882*, 2021.

543

544 Deepak Pathak, Pulkit Agrawal, Alexei A Efros, and Trevor Darrell. Curiosity-driven exploration
 545 by self-supervised prediction. In *International conference on machine learning*, pp. 2778–2787.
 546 PMLR, 2017.

547 Mihir Prabhudesai, Lili Chen, Alex Ippoliti, Katerina Fragkiadaki, Hao Liu, and Deepak Pathak.
 548 Maximizing confidence alone improves reasoning. *arXiv preprint arXiv:2505.22660*, 2025.

549

550 Yuxiao Qu, Tianjun Zhang, Naman Garg, and Aviral Kumar. Recursive introspection: Teaching
 551 language model agents how to self-improve. *Advances in Neural Information Processing Systems*,
 552 37:55249–55285, 2024.

553 Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea
 554 Finn. Direct preference optimization: Your language model is secretly a reward model. *Advances
 555 in Neural Information Processing Systems*, 36:53728–53741, 2023.

556

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

559

560 Ramanan Sekar, Oleh Rybkin, Kostas Daniilidis, Pieter Abbeel, Danijar Hafner, and Deepak Pathak.
 561 Planning to explore via self-supervised world models. In *International conference on machine
 562 learning*, pp. 8583–8592. PMLR, 2020.

563

564 Sheikh Shafayat, Fahim Tajwar, Ruslan Salakhutdinov, Jeff Schneider, and Andrea Zanette. Can
 565 large reasoning models self-train? *arXiv preprint arXiv:2505.21444*, 2025.

566

567 Rulin Shao, Shuyue Stella Li, Rui Xin, Scott Geng, Yiping Wang, Sewoong Oh, Simon Shaolei
 568 Du, Nathan Lambert, Sewon Min, Ranjay Krishna, et al. Spurious rewards: Rethinking training
 569 signals in rlvr. *arXiv preprint arXiv:2506.10947*, 2025.

570

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

574

575 Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng,
 576 Haibin Lin, and Chuan Wu. Hybridflow: A flexible and efficient rlhf framework. *arXiv preprint
 577 arXiv: 2409.19256*, 2024.

578

579 Avi Singh, John D Co-Reyes, Rishabh Agarwal, Ankesh Anand, Piyush Patil, Xavier Garcia, Peter J
 580 Liu, James Harrison, Jaehoon Lee, Kelvin Xu, et al. Beyond human data: Scaling self-training
 581 for problem-solving with language models. *arXiv preprint arXiv:2312.06585*, 2023.

582

583 Yuda Song, Hanlin Zhang, Carson Eisenach, Sham Kakade, Dean Foster, and Udaya Ghai. Mind
 584 the gap: Examining the self-improvement capabilities of large language models. *arXiv preprint
 585 arXiv:2412.02674*, 2024.

586

587 Sainbayar Sukhbaatar, Zeming Lin, Ilya Kostrikov, Gabriel Synnaeve, Arthur Szlam, and Rob
 588 Fergus. Intrinsic motivation and automatic curricula via asymmetric self-play. *arXiv preprint
 589 arXiv:1703.05407*, 2017.

590

591 Yiyou Sun, Shawn Hu, Georgia Zhou, Ken Zheng, Hannaneh Hajishirzi, Nouha Dziri, and Dawn
 592 Song. Omega: Can llms reason outside the box in math? evaluating exploratory, compositional,
 593 and transformative generalization. *arXiv preprint arXiv:2506.18880*, 2025.

594

595 Yanming Wan, Jiaxing Wu, Marwa Abdulhai, Lior Shani, and Natasha Jaques. Enhancing person-
 596 alized multi-turn dialogue with curiosity reward. *arXiv preprint arXiv:2504.03206*, 2025.

597

598 Ke Wang, Jiahui Zhu, Minjie Ren, Zeming Liu, Shiwei Li, Zongye Zhang, Chenkai Zhang, Xiaoyu
 599 Wu, Qiqi Zhan, Qingjie Liu, et al. A survey on data synthesis and augmentation for large language
 600 models. *arXiv preprint arXiv:2410.12896*, 2024.

594 Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny
 595 Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in*
 596 *neural information processing systems*, 35:24824–24837, 2022.

597

598 Qiyi Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Tiantian Fan, Gaohong
 599 Liu, Lingjun Liu, Xin Liu, et al. Dapo: An open-source llm reinforcement learning system at
 600 scale. *arXiv preprint arXiv:2503.14476*, 2025a.

601 Tianyu Yu, Bo Ji, Shouli Wang, Shu Yao, Zefan Wang, Ganqu Cui, Lifan Yuan, Ning Ding, Yuan
 602 Yao, Zhiyuan Liu, et al. Rlpr: Extrapolating rlrv to general domains without verifiers. *arXiv*
 603 *preprint arXiv:2506.18254*, 2025b.

604

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

607

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

611

612 Xuandong Zhao, Zhewei Kang, Aosong Feng, Sergey Levine, and Dawn Song. Learning to reason
 613 without external rewards. *arXiv preprint arXiv:2505.19590*, 2025b.

614

615 Xiangxin Zhou, Zichen Liu, Anya Sims, Haonan Wang, Tianyu Pang, Chongxuan Li, Liang
 616 Wang, Min Lin, and Chao Du. Reinforcing general reasoning without verifiers. *arXiv preprint*
 617 *arXiv:2505.21493*, 2025.

618

619 Yuxin Zuo, Kaiyan Zhang, Shang Qu, Li Sheng, Xuekai Zhu, Biqing Qi, Youbang Sun, Ganqu
 620 Cui, Ning Ding, and Bowen Zhou. Ttrl: Test-time reinforcement learning. *arXiv preprint*
 621 *arXiv:2504.16084*, 2025.

622

623

624

625

626

627

628

629

630

631

632

633

634

635

636

637

638

639

640

641

642

643

644

645

646

647

648 **A APPENDIX**
649650 **A.1 IMPLEMENTATION DETAILS**
651652 We build our implementation on top of verl (Sheng et al., 2024). A list of hyperparameters can be
653 found in Table 5. For all experiments, we report the best test accuracy achieved over 100 training
654 steps, averaged over three training runs.
655656 Table 5: Hyperparameters of SQLM.
657

659 Hyperparameter	659 Value
660 Max prompt length	512
661 Max response length	1024 (Arithmetic)
662	3072 (Algebra)
663	512 (Coding)
664 Batch size	64
665 Policy mini batch size	32
666 Policy micro batch size per GPU	8
667 Learning rate	1×10^{-6}
668 Weight decay	0.01
669 Learning rate warmup	Constant
670 Optimizer	Adam
671 Temperature	1.0
672 Top k	-1
673 Top p	1
674 Number of samples per example n	4
675 Remove padding	True
676 Use KL loss	True
677 KL loss coefficient	0.001
678 Clip ratio	0.2
679 Grad clip	1.0
680 Proposer update frequency	5

681 **A.2 MORE RESULTS**
682683 We use Qwen2.5-3B-Instruct for our experiments as it is a small model which still has strong math
684 and coding capabilities, as well as good instruction-following capabilities (to enable the proposer to
685 generate reasonable questions). To show the generality of our asymmetric self-play framework, we
686 also show results with Llama-3.2-3B-Instruct and Llama-3.1-8B-Instruct on Codeforces in Table 6.
687688 Table 6: Test set accuracy for Qwen2.5-Coder-3B-Instruct, Llama-3.2-3B-Instruct, and Llama-3.1-
689 8B-Instruct, with and without self-play. The best results are indicated in bold and the results show
690 standard deviations across three training runs.
691

692 Codeforces	
693 <i>Qwen2.5-Coder-3B-Instruct</i>	0.320
694 + self-play	0.391 ± 0.019
695	
696 <i>Llama-3.2-3B-Instruct</i>	0.211
697 + self-play	0.243 ± 0.026
698	
699 <i>Llama-3.1-8B-Instruct</i>	0.231
700 + self-play	0.382 ± 0.023
701	

702 A.3 PROMPTS
703

704 The prompts used for each task are shown in Table 7. For the Algebra task, we additionally en-
705 couraged diversity by asking the model to generate three problems and selecting the last one. For
706 the Coding task, we generated random numbers to show the model the expected format for the test
707 cases. This was done only to make processing of the inputs and outputs easier in the loop, and we
708 intentionally did not give an example problem in order to not bias the proposer.

709
710 Table 7: Prompts used for each task.

711 712 713 714 715 Arithmetic	716 717 718 719 720 721 722 723 724 725 726 727 728 729 730 731 732 733 734 735 736 737 738 739 740 741 742 743 744 745 746 747 748 749 750 751 752 753 754 755 Coding
	722 723 724 725 726 727 728 729 730 731 732 733 734 735 736 737 738 739 740 741 742 743 744 745 746 747 748 749 750 751 752 753 754 755 Coding
	722 723 724 725 726 727 728 729 730 731 732 733 734 735 736 737 738 739 740 741 742 743 744 745 746 747 748 749 750 751 752 753 754 755 Coding