**Page 1**

May 9, 2025

**Absolute Zero: Reinforced Self-play Reasoning with Zero Data**

1 3 1 2 1 1 1**Andrew Zhao** **,** **Yiran Wu** **,** **Yang Yue** **,** **Tong Wu** **,** **Quentin Xu** **,** **Yang Yue** **,** **Matthieu Lin** **,**

1 3 2 1**Shenzhi Wang** **, Qingyun Wu** **, Zilong Zheng** **and Gao Huang**� �, ,

Tsinghua University Beĳing Institute for General Artificial Intelligence Pennsylvania State University1 2 3

zqc21@mails.tsinghua.edu.cn, yiran.wu@psu.edu, zlzheng@bigai.ai, gaohuang@tsinghua.edu.cn

arXiv:2505.03335v2 [cs.LG] 7 May 2025

Reinforcement learning with verifiable rewards (RLVR) has shown promise in enhancing the reasoning

capabilities of large language models by learning directly from outcome-based rewards. Recent RLVR

zero settingworks that operate under the avoid supervision in labeling the reasoning process, but still

depend on manually curated collections of questions and answers for training. The scarcity of high-

quality, human-produced examples raises concerns about the long-term scalability of relying on human

supervision, a challenge already evident in the domain of language model pretraining. Furthermore, in a

hypothetical future where AI surpasses human intelligence, tasks provided by humans may offer limited

learning potential for a superintelligent system. To address these concerns, we propose a new RLVR

Absolute Zeroparadigm called , in which a single model learns to propose tasks that maximize its own

learning progress and improves reasoning by solving them, without relying on any external data. Under

this paradigm, we introduce the Absolute Zero Reasoner (AZR), a system that self-evolves its training

curriculum and reasoning ability by using a code executor to both validate proposed code reasoning tasks

and verify answers, serving as an unified source of verifiable reward to guide open-ended yet grounded

without external datalearning. Despite being trained entirely , AZR achieves overall SOTA performance

outperforming existing zero-setting modelson coding and mathematical reasoning tasks, that rely on tens

in-domain human-curated examplesof thousands of . Furthermore, we demonstrate that AZR can be

effectively applied across different model scales and is compatible with various model classes.

**Code** **Project Page** **Logs** **Models**

**Absolute Zero Reasoner (AZR) achieves state-of-the-art performance with ZERO DATA**Figure 1. . Without relying on any gold

labels or human-defined queries, Absolute Zero Reasoner trained using our proposed self-play approach demonstrates impressive general

reasoning capabilities improvements in both math and coding, despite operating entirely out-of-distribution. Remarkably, AZR surpasses

models trained on tens of thousands of expert-labeled in-domain examples in the combined average score across both domains.

Corresponding author(s)�

**Page 2**

**Absolute Zero: Reinforced Self-play Reasoning with Zero Data**

Supervised Learning Reinforcement Learning with Verifiable Rewards Absolute Zero (Ours)

Less Human Supervision

**Absolute Zero Paradigm. Supervised learning**  **Reinforcement**Figure 2. relies on human-curated reasoning traces for behavior cloning.

**learning from verified rewards**, enables agents to self-learn reasoning, but still depends on expert-defined learning distribution and a

**Absolute**respective set of curated QA pairs, demanding domain expertise and manual effort. In contrast, we introduce a new paradigm,

**Zero**, for training reasoning models without any human-curated data. We envision that the agent should autonomously propose tasks

optimized for learnability and learn how to solve them using an unified model. The agent learns by interacting with an environment that

provides verifiable feedback, enabling reliable and continuous self-improvement entirely without human intervention.

**1. Introduction**

Large language models (LLMs) have recently achieved remarkable improvements in reasoning capabilities by employing Reinforcement

Learning with Verifiable Rewards (RLVR) (Lambert et al., 2024). Unlike methods that explicitly imitate intermediate reasoning steps,

RLVR uses only outcome-based feedback, enabling large-scale reinforcement learning over vast task datasets (DeepSeek-AI et al., 2025;

“zero”Team et al., 2025; Jaech et al., 2024; OpenAI, 2025b;a). A particularly compelling variant is the RLVR paradigm (DeepSeek-AI

et al., 2025), which forgoes any cold-start distillation data, using neither human-generated nor AI-generated reasoning traces, and applies

RLVR directly on the base model with task rewards. However, these methods still depend heavily on expertly curated distributions of

reasoning question–answer pairs, which raises serious concerns about their long-term scalability (Villalobos et al., 2024). As reasoning

models continue to advance, the effort required to construct large-scale, high-quality datasets may soon become unsustainable (Yue

et al., 2025). A similar scalability bottleneck has already been identified in the domain of LLM pretraining (Sutskever et al., 2024).

Furthermore, as AI systems continue to evolve and potentially exceed human intellect, an exclusive dependence on human-designed

tasks risks imposing constraints on their capacity for autonomous learning and growth (Hughes et al., 2024). This underscores the need

for a new paradigm that begins to explore possibilities beyond the constraints of human-designed tasks and prepares for a future in which

AI systems may surpass human intelligence.

“Absolute Zero”To this end, we propose , a new paradigm for reasoning models in which the model simultaneously learns to define tasks

that maximize learnability and to solve them effectively, enabling self-evolution through self-play without relying on external data. In

contrast to prior self-play methods that are limited to narrow domains, fixed functionalities, or learned reward models that are prone to

Absolute Zerohacking (Silver et al., 2017; Chen et al., 2025; 2024), the paradigm is designed to operate in open-ended settings while

remaining grounded in a real environment. It relies on feedback from the environment as a verifiable source of reward, mirroring how

humans learn and reason through interaction with the world, and helps prevent issues such as hacking with neural reward models (Hughes

et al., 2024). Similar to AlphaZero (Silver et al., 2017), which improves through self-play, our proposed paradigm requires no human

supervision and learns entirely through self-interaction. We believe the Absolute Zero paradigm represents a promising step toward

enabling large language models to autonomously achieve superhuman reasoning capabilities.

Absolute Zero Reasoner (AZR)Building on this new reasoning paradigm, we introduce the , which proposes and solves coding tasks. We

cast code executor as an open-ended yet grounded environment, sufficient to both validate task integrity and also provide verifiable

feedback for stable training. We let AZR construct three types of coding tasks: infer and reason about one particular element in a

program, input, output triplet, which corresponds to three complementary modes of reasoning: induction, abduction, and deduction. We

train the entire system end-to-end with a newly proposed reinforcement learning advantage estimator tailored to the multitask nature of

the proposed approach.

Despite being trained entirely without any in-distribution data, AZR demonstrates remarkable capabilities across diverse reasoning tasks

in math and coding. In mathematics, AZR achieves competitive performance compared to zero reasoner models explicitly fine-tuned

with domain-specific supervision. In coding tasks, AZR establishes a new state-of-the-art performance, surpassing models specifically

trained with code datasets using RLVR. Furthermore, AZR outperforms all previous models by an average of 1.8 absolute points

2

**Page 3**

**Absolute Zero: Reinforced Self-play Reasoning with Zero Data**

compared to models trained in the “zero” setting using in-domain data. These surprising results highlight that general reasoning skills

can emerge without human-curated domain targeted data, positioning Absolute Zero as an promising research direction and AZR as a

first pivotal milestone. Besides the remarkable results AZR achieved with zero human data for reasoning, we also make very interesting

findings summarized below:

**Code priors amplify reasoning.**• The base model started with math performance 3.6 points lower than . Qwen-Coder-7b Qwen-7b

But after AZR training for both models, the coder variant surpassed the base by 0.7 points, suggesting that strong coding capabilities

may potentially amplify overall reasoning improvements after AZR training.

**Cross domain transfer is more pronounced for AZR.**• After RLVR, expert code models raise math accuracy by only 0.65 points on

average, whereas and trained on self-proposed code reasoning tasks improve math average by 10.9 and AZR-Base-7B AZR-Coder-7B

15.2, respectively, demonstrating much stronger generalized reasoning capability gains.

**Bigger bases yield bigger gains.**• Performance improvements scale with model size: the 3B, 7B, and 14B coder models gain +5.7,

+10.2, and +13.2 points respectively, suggesting continued scaling is advantageous for AZR.

**Comments as intermediate plans emerge naturally.** When solving code induction tasks, AZR often interleaves step-by-step plans•

as comments and code (Appendix C.3), resembling the ReAct prompting framework (Yao et al., 2023). Similar behavior has been

observed in much larger formal-math models such as DeepSeek Prover v2 (671B) (Ren et al., 2025). We therefore believe that allowing

the model to use intermediate scratch-pads when generating long-form answers may be beneficial in other domains as well.

**Cognitive Behaviors and Token length depends on reasoning mode.** Distinct cognitive behaviors—such as step-by-step reasoning,•

enumeration, and trial-and-error all emerged through AZR training, but different behaviors are particularly evident across different

types of tasks. Furthermore token counts grow over AZR training, but the magnitude of increase also differs by task types: abduction

grows the most because the model performs trial-and-error until output matches, whereas deduction and induction grow modestly.

**Safety alarms ringing.**• We observe AZR with occasionally produces concerning chains of thought, we term the Llama3.1-8b

“uh-oh moment”, example shown in Figure 32, highlighting the need for future work on safety-aware training (Zhang et al., 2025a).

**2. The Absolute Zero Paradigm**

**2.1. Preliminaries**

**Supervised Fine-Tuning (SFT).** = ( )SFT requires the datasets of task-rationale-answer demonstrations , where⋆ ⋆ D { }x, c, y

is the query, is the gold chain-of-thought (CoT)) and is the gold answer, all provided by human experts or superior AI models.⋆ ⋆x c y

The model trains to imitate the reference responses to minimize the conditional negative log-likelihood (Ouyang et al., 2022):

�

)() = log (1)⋆ ⋆ E |L − c, y x.θ πSFT ( )⋆ ⋆ θ∼Dx,c,y

However, at the frontier level, there’s no stronger model to distill from, and expert human labeling doesn’t scale well.

**Reinforcement Learning with Verifiable Rewards (RLVR).** To move beyond the limits of pure imitation, RLVR only

= ( )requires a dataset of task and answer , without labeled rationale. RLVR allows the model to generate its own CoT and⋆ D { }x, y

( )calculate a verifiable reward with the golden answer . However, the learning task distribution , with its set of queries and gold⋆ D ry, y

answers are still labeled by human experts. The trainable policy is optimized to maximize expected reward: πθ

� �

() = ( ) (2)⋆ EJ θ ry, y .RLVR ( ) ( )⋆∼D ∼ · |x,y , y π xθ

In summary, both SFT and RLVR still rely on human-curated datasets of either queries, demonstrations, or verifiers, which ultimately

limit scalability. The Absolute Zero paradigm removes this dependency by allowing the model to generate, solve, and learn from its own

interactions with the environment entirely through self-play.

**2.2. Absolute Zero**

We propose the Absolute Zero paradigm, where during training, the model simultaneously proposes tasks, solves them, and learns from

both stages. No external data is required and the model learns entirely through self-play and experience, aided by some environment. We

illustrate this paradigm in Figure 2, which contrasts Absolute Zero with supervised learning and RLVR, highlighting how our approach

eliminates the need for any human-curated data by enabling self-improving task proposal and solution through self-play.

To make the Absolute Zero setting concrete, we now define how one model can act both as the proposer and solver role. To aid

understanding, we include an illustration in Figure 3. Let be our parameterized language model, it is used to play two roles, proposer πθpropose solveand solver during training.π πθθ

3

**Page 4**

**Absolute Zero: Reinforced Self-play Reasoning with Zero Data**

The proposer first samples a proposed

𝑦𝜏task conditioned on variable : ∼ z τ

propose( ), which will then be validated·|π zθ '#()\*𝜋and used to construct a valid reasoning task Environment EnvironmentLanguage Model ( )together with the environment : 𝑒, 𝑓 𝑒⋆ ∼ e x, y !"#!$%&𝜋

( ), where is the task query and ⋆·|f τ x ye

is the gold label. Then the solver produces

⋆ "#$"$%& %$'(&𝑥, 𝑦 , 𝑟 𝑟solve( )an answer . Each pro- ∼ · | y π xθ learnabilityposed task is scored by a τ

propose( )reward , which captures the  **The Absolute Zero Loop.**Figure 3. The Absolute Zero loop begins with the agent r τ, π πe θexpected improvement in after train- proposing task , which is transformed by with the environment into a validated π τ f eθing on the task query . Moreover, the propose ( )problem , and also emits a reward for learnability. Then, a standard RL step⋆ x x, y r

solution re-same policy also receives a solvefollows: the agent solves by producing , receiving reward from by matching x y r e

solve( )ward for its answer to the task propose solvewith . and are jointly trained and this process can be repeated indefinitely.⋆ ⋆ r y, y y π πequery , with the environment again serv- x

ing as the verifier. A nonnegative coefficient balances the trade-off between exploring new, learnable tasks and improving the model’s λ

reasoning and problem-solving abilities. We formally define the absolute zero setting’s objective as follows:

� ����

�propose solve) := max ( ( ) + ( ) (3)⋆E E EJ θ r τ, π λ r y, y .propose() ( ) ( ) ( ) ( )solveθ∼ e ez pz ⋆∼ ·| ∼ ·| ∼ ·|x,y f τ,τ π z y π xe θθ θ

proposeNotice that we shift the burden of scaling data away from human experts and onto the proposer policy and the environment πθ. These two roles are both responsible for defining/evolving the learning task distribution, validating proposed tasks, and providinge

grounded feedback that supports stable and self-sustainable training. When proposing, acts as a conditional variable that seeds z

generation of tasks. Practically, can be instantiated by sampling a small subset of past (task, answer) pairs from a continually updated z

task memory, yet there is no specific implementation tied to the paradigm. To guide the proposing process, we use a learnability reward

propose( ), which measures how much the model is expected to improve by solving a proposed task . Moreover, the solver rewardr τ, π τθsolve( ) evaluates the correctness of the model’s output. Together, these two signals guide the model to propose tasks that are both∗r y, y

challenging and learnable, while also enhancing its reasoning abilities, ultimately enabling continuous improvement through self-play.

**3. Absolute Zero Reasoner**

Absolute Zero ReasonerIn this section, we present (AZR) as the first attempt to embrace the Absolute Zero Paradigm. In AZR, an

unified LLM serves as both a proposer and a solver: it generates tasks to evolve its learning curriculum and attempts to solve them

to improve its reasoning capabilities. The model is trained jointly with both roles, learning to create tasks that push the boundary of

reasoning capacity while enhancing its ability to solve them effectively (Section 3.1). Within this self-play training paradigm, the model

learns from three distinct type of coding tasks, which corresponding to three fundamental modes of reasoning: abduction, deduction and

induction (Section 3.2). Using coding tasks is motivated by the Turing-completeness of programming languages (Stuart, 2015) and

empirical evidence that code-based training improves reasoning (Aryabumi et al., 2024). We adopt code as an open-ended, expressive,

and verifiable medium for enabling reliable task construction and verification (Section 3.3). Finally, the model is updated using a newly

proposed advantage estimator designed for multitask learning (Section 3.3.5). We outline the overall algorithm in Algorithm 1 and

highlight an illustration of our Absolute Zero Reasoner approach in Figure 4. To expedite future exploration in this area, we also present

several attempts that did not yield fruitful results but still warrant discussion in Appendix D.

**3.1. Two Roles in One: Proposer and Solver**

Large language models are naturally suited for implementing AZR in a multitask learning context (Radford et al., 2019), as both

the formulation of reasoning tasks and their solutions occur within a unified language space. To this end, we propose rewarding a

single model for both generating high learning potential tasks and solving them effectively, as specified by the Absolute Zero objective

in Equation (3). At each iteration of the online rollout, AZR proposes new reasoning tasks by conditioning on the task type (as defined

in Section 3.2) and past self-generated examples. The model is explicitly prompted to generate tasks that differ from these examples, K

promoting diversity and broader coverage of the task space. These task proposals are filtered and transformed into valid reasoning

tasks that can be verified using the environment, outlined later in Section 3.3. AZR then attempts to solve these newly proposed tasks,

receiving grounded feedback for its model responses. Both task proposal and problem solving are trained using reinforcement learning.

We now outline the rewards used for each role.

**Reward Design.** Prior work has shown that setting appropriate task difficulty is critical for promoting effective learning in reasoning

systems (Zeng et al., 2025b). Motivated by this, we design a reward function for the proposer that encourages generation of tasks

4

**Page 5**

**Absolute Zero: Reinforced Self-play Reasoning with Zero Data**

PROPOSE Learnability

Construct & Estimate Reward

Task Types

Absolute  
Absolute  
Self-play OX = F ( )PAbduction: ?

Joint UpdateZero  
Zero

? = F ( )Deduction: P I

ReasonerReasoner

Induction: OX = ( )I?

Accuracy  
SOLVE

Verify

Reward

model input/output model reward ( , , )P rogram I nput O utput

**Absolute Zero Reasoner Training Overview.**  **PROPOSES**Figure 4. At every iteration, Absolute Zero Reasoner first a batch of tasks,

conditioned on past self-generated triplets stored in a buffer and a particular task type: abduction, deduction, or induction (Section 3.2).

From these generated tasks, Python is used to filter and construct valid code-based reasoning questions. A learnability reward is rpropose

**SOLVES**also calculated for each proposed task as defined in Equation (4). The Absolute Zero Reasoner then the batch of reasoning

questions. Python is used again to verify the generated responses and compute the accuracy reward as described in Equation (5). rsolve

Finally, the Absolute Zero Reasoner is jointly updated using both and across all three task types, using TRR++ (Section 3.3.5). r rpropose solve

with meaningful learning potential—neither too easy nor unsolvable for the current solver. Concretely, we use the same language

learnabilitymodel in its solver role to estimate the of a proposed task, a similar type of reward used in unsupervised environment

design literature (Sukhbaatar et al., 2018). We perform Monte Carlo rollouts of the solver and compute the average success rate: n� ()1¯ = . The proposer’s reward is then defined as:iN rrsolve solve=1n i

�

0 ¯ = 0 ¯ = 1if or, r rsolve solve = (4)rpropose 1 ¯ otherwise −r , ,solve

¯ = 1 ¯ = 0The intuition is that if a task is either trivial to solve ( ) or unsolvable ( ), the task provides little to no learning signalr rsolve solve

for the proposer. In contrast, tasks of moderate difficulty, where the solver occasionally succeeds are rewarded the most, as they offer the

richest feedback and greatest potential for learning.

For the solver, we assign a simple binary reward based on the correctness of its final output,

= (5) Ir ,solve (= )⋆y y

where is the ground-truth answer, and equality is evaluated based on value equality in Python.⋆ y

With the primary rewards for the proposing and solving roles defined, we adopt the following composite reward structure, which

integrates and with a format-aware penalty inspired by DeepSeek-AI et al. (2025): r rpropose solve



if the response is passable, role propose,solve ∈ { }rrole

( ) = 05 (6)if the response is wrong but well-formatted,Ry −.π  1 if the answer has formatting errors,−

where is the response of the language model. The main format that the proposing and solving tasks need to follow is the DeepSeek yπ

R1 and format, as shown in Figure 33. Moreover, for the proposer, the reward criterion for format goes beyond <think> <answer>

simply following the XML structure. As detailed in Section 3.3.3, only responses that produce valid triplets and pass the filtering stage

are considered to be correctly formatted.

5

**Page 6**

**Absolute Zero: Reinforced Self-play Reasoning with Zero Data**

**Absolute Zero: Reinforced Self-play Reasoning with Zero Data3.2. Learning Different Modes of Reasoning: Deduction, Induction, and Abduction**

**3.2. Learning Different Modes of Reasoning: Deduction, Induction, and Abduction**AZR uses code executor as both a flexible interface and a verifiable environment. This setup enables automatic construction, execution,

and validation of code reasoning tasks (Stuart, 2015; Aryabumi et al., 2024). Give program space , input space and output space P I OAZR uses code executor as both a flexible interface and a verifiable environment. This setup enables automatic construction, execution, ( )of a coding language, we define an AZR reasoning task as a triplet , where is a program, is an input, and is ∈ ∈ ∈p, i, o p i o P I Oand validation of reasoning tasks (Stuart, 2015; Aryabumi et al., 2024). Give program space , input space and output space of P I O = ()the corresponding output produced by running program on input, . AZR learns by reasoning about different parts of this task o pi ( )a coding language, we define an AZR reasoning task as a triplet , where is a program, is an input, and is → → →p, i, o p i o P I Otriplet, using three distinct core reasoning modes, each of which focuses on inferring one part of the triplet given the others:

= ()the corresponding output produced by running program on input, . AZR learns by reasoning about different parts of this task o pi

triplet, using three distinct core reasoning modes, each of which focuses on inferring one part of the triplet given the others: **Deduction**1. : predicting the output given a program and input , capturing step-by-step logical reasoning. o p i

**Deduction**1. : predicting the output given a program and input , capturing step-by-step logical reasoning. = proposer• As a , AZR is conditioned on the task type deduction and reference examples from the deduction buffer o p i D α K deduction

( ) ()(all task buffers are outlined in Section 3.3), and generates a pair . The environment then executes to compute , = proposer• As a , AZR is conditioned on the task type deduction and reference examples from the deduction bufferp, i e pi o D α K deduction ( )completing the triplet , which is added to the buffer if non-error output was produced. ( ) ()(all task buffers are outlined in Section 3.3), and generates a pair . The environment then executes to compute ,p, i, o p, i e pi o

( )completing the triplet , which is added to the buffer if non-error output was produced. ( ) solver• As a , the model receives and predicts the output . The predicted output is verified using type-aware value equalityp, i, op, i oπ ( )in python to account for possible variations (such as set ordering or fractions). solver• As a , the model receives and predicts the output . The predicted output is verified using type-aware value equalityp, i oπin python to account for possible variations (such as set ordering or fractions). **Abduction**2. : inferring a plausible input given the program and an output , resembling trial-and-error or online search. i p o

**Abduction**2. given the program and an output: inferring a plausible input , resembling trial-and-error or online search. i p opropose proposer As a , the policy ’s input and output is almost the same as the proposer for the deduction task, except that the task• π propose proposer• As a , the policy ’s input and output is almost the same as the proposer for the deduction task, except that the task = ( ) πtype abduction is changed as an input. The model generates a pair conditioned on and reference examples. Then we α p, i α = ( )type abduction is changed as an input. The model generates a pair conditioned on and reference examples. Then we α p, i α() ( )executes and get the triplet . pi p, i, o() ( )executes and get the triplet . pi p, i, o ( ) ( ) = solver As a , the model receives and predicts . The solution is verified by checking whether . Since programs• p, o i pi o ( ) ( ) = solver• As a , the model receives and predicts . The solution is verified by checking whether . Since programsπ πp, o i pi o outputmay not be bĳective, we use value equivalence rather than requiring exact input matches.π π outputmay not be bijective, we use value equivalence rather than requiring exact input matches.

( ) **Induction:**3. synthesizing a program from a set of in-out examples , requiring generalization from partial information.n n { } p i , o( ) **Induction:**3. synthesizing a program from a set of in-out examples , requiring generalization from partial information.n n { } p i , o

proposer• As a , AZR samples a valid program from , generates new inputs and a message , and uses the proposer• As a , AZR samples a valid program from , generates new inputs and a message , and uses the D ∪ D p N mabduction deduction D ∪D p N mabduction deduction ( ( ) )environment to compute corresponding outputs. This forms an extended task representation , which is storedn n ( ( ) )environment to compute corresponding outputs. This forms an extended task representation , which is stored { }p, i , o , mn n { }p, i , o , m

in the induction buffer . Since infinitely many functions can map the inputs to the outputs, making the induction taskin the induction buffer . Since infinitely many functions can map the inputs to the outputs, making the induction task Dinduction Dinduction

under-constrained, the message helps properly condition the problem for the solver.under-constrained, the message helps properly condition the problem for the solver. m m

solver• As a , the model is shown the first half of the input-output pairs and the message , and must synthesize a program that solver• As a , the model is shown the first half of the input-output pairs and the message , and must synthesize a program that m p m pππcorrectly maps the remaining hidden inputs to their outputs. The use of held-out examples discourages overfitting through if-elsecorrectly maps the remaining hidden inputs to their outputs. The use of held-out examples discourages overfitting through if-else

logic and promotes generalized induction.logic and promotes generalized induction.

Each reasoning task type leverages code as an expressive and verifiableEach reasoning task type leverages code as an expressive and verifiable

medium, aligning with the Absolute Zero Paradigm’s goals of fully self-medium, aligning with the Absolute Zero Paradigm’s goals of fully self-

**Program Triplet**improving systems in open-ended domains (DeepSeek-AI et al., 2025; Lam-improving systems in open-ended domains (DeepSeek-AI et al., 2025;

bert et al., 2024). All prompts used by three different task types and twoLambert et al., 2024). All prompts used by three different task types and **Input:** "Hello World"types of roles within a task type are shown in Figures 34 to 39. Next, wetwo types of roles within a task type are shown in Figures 34 to 39. Next,

outline exact details of our algorithm.

we outline exact details of our algorithm.

**def** f(x):1**3.3. Absolute Zero Reasoner Learning Algorithm**

**return** x**3.3. Absolute Zero Reasoner Learning Algorithm** 2

In this section, we will discuss details of our AZR self-play algorithm, includ-

In this section, we will discuss details of our AZR self-play algorithm, includ-ing initialization of buffers 3.3.1, usage of thse buffers 3.3.2, construction of

ing initialization of buffers 3.3.1, usage of thse buffers 3.3.2, construction ofvalid tasks 3.3.3, validating solutions 3.3.4, and finally advantage estimator **Output:** "Hello World"valid tasks 3.3.3, validating solutions 3.3.4, and finally advantage estimatorcalculation 3.3.5. We outline the overall recipe of the self-play procedure of

calculation 3.3.5. We outline the overall recipe of the self-play procedureAZR in Algorithm 1.

of AZR in Algorithm 1.

**The Seed AZR Zero Triplet. The Seed AZR Zero Triplet.**Figure 5. The aboveFigure 5. The above

3.3.1. BUFFER INITIALIZATION identity function triplet was the only triplet providedidentity function triplet was the only triplet provided3.3.1. Buffer Initialization

to AZR to initiate its self-bootstrap propose-and-solveto AZR to initiate its self-bootstrap propose-and-solveTo initialize AZR self-play, we first generate a seed set of valid triplets using

RLVR loop. We note that the base LLM is fully ca-RLVR loop. We note that the base LLM is fully capableTo initialize AZR self-play, we first generate a seed set of valid triplets usingthe base language model. Each prompt samples up to triplets from the K pable of initiating the AZR loop without any seed pro-of initiating the AZR loop without any seed program;the base language model. Each prompt samples up to triplets from thecurrent seed buffer as references. When is empty at time 0, we K D Dseed seed gram; its inclusion illustrates our approach’s flexibility:its inclusion illustrates our approach’s flexibility: wecurrent seed buffer as references. When is empty at time 0, wefall back to the zero triplet show in Figure 5. During the seeding stage, we D Dseed seed we can optionally initialize seed programs with existingcan optionally initialize seed programs with existinguse the same proposer prompts detailed in Figures 34 to 36.fall back to the zero triplet show in Figure 5. During the seeding stage, we

datasets of varying complexity, and we initialized oursdatasets of varying complexity, and we initialized oursuse the same proposer prompts detailed in Figures 34 to 36.

First, for deduction and abduction tasks, the LLM is prompted to generate with the simplest program.with the simplest program.

( ) pairs, which are filtered, executed, and stored as valid triplets. WeFirst, for deduction and abduction tasks, the LLM is prompted to generatep, i

00 = = =initialize , where , where( ) pairs, which are filtered, executed, and stored as valid triplets. We D D D |D | × B Sseed seedp, i abduction deduction

= 4 is the batch size, and is a factor we fix in all experiments. All seed triplet’s program are stripped of global variables and0 0= = = = 4initialize , where , where is the batch size, and is a factor we fix in allB S D D D |D | × B S B Sseed seedabduction deductioncomments (Appendix C), but subsequent iterations of adding new triplets to the buffers are unaltered. No model updates occur duringexperiments. All seed triplet’s program are stripped of global variables and comments (Appendix D), but subsequent iterations of adding

this phase. Similarly, to initialize the induction buffer, we sample programs from , generate matching input sets and messages, and Dnew triplets to the buffers are unaltered. No model updates occur during this phase. Similarly, to initialize the induction buffer, weseed

0 =collect valid examples until . 0 = |D | × B Ssample programs from , generate matching input sets and messages, and collect valid examples until .induction D |D | × B Sseed induction

66

**Page 7**

**Absolute Zero: Reinforced Self-play Reasoning with Zero Data**

**Algorithm 1** Self-Play Training of Absolute Zero Reasoner (AZR)

**Require:** Pretrained base LLM ; batch size ; #references ; iterations π B K Tθ ( ) InitSeeding see §3.3.11: D D D ←, , π ▷ded abd ind θ 1 **for**  **do** to2: ← t T

1**for**  **do**  **PROPOSE PHASE** to3: ← b B ▷

sample a program for induction task proposal4: ∼ D ∪ Dp ▷abd ded

� � propose( )N ind generate inputs and a descriptionn5: ←i , m π , p ▷ N=1 π  **then**π θn� � � �( )**if** N ValidateByExecuting syntax validate I/Os, see §3.3.3n n n6: ← { }i , o p, i , ▷=1π π πn � �( ( ) ) update induction buffern n7: D ← D ∪ { }p, i , o , m ▷ind ind ππ π

**for**  **do**ded abd8: ∈ { } α ,

� �K sample reference examples9: ∼ Dp , i , o ▷ K=1 αk k k k � �propose( ) ( ) propose new task10: ← { }p , i π α, p , i , o ▷ **then**π π k k kθ � �**if** ValidateByExecuting syntax,safety,determinism see §3.3.311: ← o p , i , ▷π π π� �( ) if valid, update deduction or abduction buffers12: D ← D ∪ p , i , o ▷α α π π π

**for all**  **do**  **SOLVE PHASE**ded abd ind13: ∈ { } α , , ▷

� �( ) SamplePrepareTasks prepared based on , see §3.3.3&3.3.4⋆ ⋆14: ← Dx, y , B, t ▷ x, y αα

solve( )15: ∼y π xπ θ

**Reward:** Use proposed task triplets and solved answers to get & see §3.116: r r ▷propose solve**RL update:** use Task Relative REINFORCE++ to update see §3.3.517: π ▷θ

3.3.2. Task Proposal Inputs and Buffer Management

FirstDuring the actual self-play stage of AZR, we use the task buffer in three ways. , for the proposer of abduction and deduction tasks,

we uniformly sample past triplets from the buffer, present them as in-context examples to the proposer and let it generate a new task. K

SecondThe design is to show it past examples, and prompt it to generate a different one to promote diversity (Zhao et al., 2025a). , we

� Dsample one triplet from the union of abduction and deduction buffers , and present the program from that triplet to the D pabd ded

Lastlyinduction proposer to generate a set of matching inputs and a natural language message . , to maintain stable training, ifn { } N i m

a batch of solver problems contains fewer than valid proposed tasks (proposer not adhering to formatting), we fill the remainder by B

uniformly sampling from the corresponding task buffer of previously validated triplets.

( )The buffer grows for abduction and deduction tasks whenever propose a valid triplet , regardless if it gets any task reward. π p, i, o

( )Similarly, for induction tasks, all valid triplets are added to the buffer.n n { }p, i , o , m

3.3.3. Constructing Valid Tasks

**Proposal Task Validation.** deduction We first describe how we construct valid tasks from the proposals generated by the policy . For π

( )and abduction tasks, each proposal consists of a program and an input . To validate the task, we use the task validation procedurep, i

( ) induction(steps shown below) on the input to obtain the correct output , resulting in a complete triplet . For tasks, given a o p, i, o

program the policy proposes a set of inputs and message . We also use the task validation procedure on each of the inputn n { } p i m i

in the set to obtain a corresponding output , forming a set of input-output pairs . We do not impose any constraints on .n n n { } o i , o m

**task**The resulting task is considered valid only when all inputs yield valid outputs and the formatting requirements are satisfied. The

**validation procedure** entails:

Program Integrity.1. We first use Python to run the program with the input . If no errors are raised and something is returned, we p i

( )then gather the output of that pair and determine that the program at least has valid syntax. o p, i

Program Safety.2. We also check whether a program is safe for execution by restricting the use of certain sensitive packages that might

i.ecause harm to the Python environment, ., . The list of packages used to filter out invalid programs is os.sys, sys, shutil

provided in Figure 8. This list is also included in the instructions when prompting the language model to generate questions. See

Figures 34 to 36.

Check for Determinism. deterministic programs i.e3. In our setting, we only consider , ., , where is the space ∈ ⊂ p P P Pdeterministic

of all valid programs and is the space of all valid inputs: I

7

**Page 8**

**Absolute Zero: Reinforced Self-play Reasoning with Zero Data**

� �

= =(1) (2) ()lim () () = () (7)j∀ ∈ ∀ ∈ · · ·p , i , pi pi pi , P Ideterministic

→∞j

() ()where indexes repeated independent executions of the program. That is, for all inputs , the output of remains identicalj i pi

( ) = () valid program/input/output tripletwith any independent execution of the program. A is defined such that , wherep, i, o o pi

. ∈p Pdeterministic

Since the output of probabilistic programs can vary on every individual run, it is non-trivial to use verifiable functions to evaluate the

correctness of an answer. Therefore, to keep the verifier simple, we restrict the valid programs generated by the learner to the class

of deterministic programs. We believe that stochastic programs can encompass a larger class of behaviors and are important and

promising to include in future versions of AZR.

To implement the filtering of invalid probabilistic programs, and following the definition of a deterministic program highlighted in

Equation (7), we approximate this procedure by independently running the program finite times and checking that all the outputs j

= 2are equal. For computational budget reasons, we fixed for all experiments. j

**Solving Task Construction.** If a task proposal passes these three checks, we deem it a valid task and apply appropriate procedures to

2 = ( ) = ( ) = ( )present part of the triplet to the solver. Specifically, we set for deduction; for abduction; and N//n n{ } x p, i x p, o x i , o , m=1nfor induction, where half of the tests cases and a program description is used. We use all valid tasks from timestep ; if the batch is m t B

not full, we uniformly sample from previously validated tasks to fill the batch.

3.3.4. Answer Verification

( ) = ( )For abduction task, we receive from the solver policy, then we equivalence match using , where refers to the⋆ ∗ i pi piπ π

privileged gold information. The reason we do not just match and is because is not necessarily bĳective. For deduction task, we⋆ i i pπ = all( ( ) = )match . For induction, we match . This part might be convoluted to explain in language, therefore we⋆ ⋆ ⋆ N{ } o o p i oπ π n nrecommend the reader to see how we did abduction, deduction and induction verification in code in Figures 10 to 12, respectively.

3.3.5. Task-Relative REINFORCE++

Since AZR trains the combination of roles and task types, it operates in a multitask reinforcement learning setup (Zhang & Yang, 2021;

Zhao et al., 2022; Wang et al., 2023; Yue et al., 2023). Instead of computing a single global baseline as in REINFORCE++ (Hu, 2025)

(Appendix A), we compute separate baselines for each of the six task-role configurations. This can be viewed as an interpolation between

per-question baselines, as in GRPO (Shao et al., 2024), and a global baseline, allowing for more structured variance reduction tailored to

norm **Task-Relative REINFORCE++ (TRR++)**each task setup. We refer to this variant as . The normalized advantage is computed as: A

− r µtask,rolenorm = task ind,ded,abd role propose,solve (8) ∈ { } ∈ { }A , , ,task,role σtask,role

within each task type and rolewhere the mean and standard deviation are computed , yielding six baselines.

**4. Experiments**

**4.1. Experiment Setup**

**Training Details.** For all experiments, we initialize the buffers as described in Section 3.1. AZR models are trained using a batch

64 6 = 1 6size of (2 roles 3 task types). We use constant learning rate and the AdamW optimizer (Loshchilov & Hutter, 2019). × × −e

Complete list of hyperparameters is provided in Table 3.

For the main experiments, we train AZR models on and , resulting in Qwen2.5-7B Qwen2.5-7B-Coder Absolute

and , respectively. Additional experiments include trainingZero Reasoner-base-7B Absolute Zero Reasoner-Coder-7B

, , , (Yang et al., 2024a; Hui et al., 2024; Dubey et al.,Qwen2.5-Coder-3B Qwen2.5-Coder-14B Qwen2.5-14B Llama-3.1-8B

2024).

**Evaluation Protocol.** To evaluate our models, we divide the datasets into in-distribution (ID) and out-of-distribution (OOD)

categories. For OOD benchmarks, which we emphasize more, we further categorize them into coding and mathematical reasoning

benchmarks. For coding tasks, we evaluate using Evalplus (Liu et al., 2023) on the HumanEval+ and MBPP+ benchmarks, as

well as LiveCodeBench Generation (v1-5, May 23-Feb 25) (Jain et al., 2024). For mathematical reasoning, we utilize six standard

benchmarks commonly used in recent zero-shot trained reasoners: AIME’24, AIME’25, OlympiadBench (He et al., 2024), Minerva,

Math500 (Hendrycks et al., 2021), and AMC’23. For ID benchmarks, we use CruxEval-I(nput), CruxEval-O(utput), and LiveCodeBench-

Execution (Gu et al., 2024; Jain et al., 2024), which assess reasoning capabilities regarding the input and output of programs (Li et al.,

Greedy decoding2025). is used for all baseline methods and AZR results to ensure reproducibility.

8

**Page 9**

**Absolute Zero: Reinforced Self-play Reasoning with Zero Data**

+ + v1-5**Model** **Base** **#data** **CAvg** **MAvg** **AVG**HEval MBPP LCB AME24 AME25 AMC M500 Minva Olypiad

**Base Models**

[73]Qwen2.5-7B - - 73.2 65.3 17.5 6.7 3.3 37.5 64.8 25.0 27.7 52.0 27.5 39.8

[73]Qwen2.5-7B-Ins - - 75.0 68.5 25.5 13.3 6.7 52.5 76.4 35.7 37.6 56.3 37.0 46.7

[26]Qwen2.5-7B-Coder - - 80.5 69.3 19.9 6.7 3.3 40.0 54.0 17.3 21.9 56.6 23.9 40.2

[74] 61.0 57.9 16.2 10.0 16.7 42.5 64.2 15.4 28.0 45.0 29.5 37.3Qwen2.5-7B-Math - -

**Zero-Style Reasoners Trained on Curated Coding Data**

[84]AceCoder-RM Ins 22k 79.9 71.4 23.6 20.0 6.7 50.0 76.4 34.6 36.7 58.3 37.4 47.9

[84]AceCoder-Rule Ins 22k 77.4 69.0 19.9 13.3 6.7 50.0 76.0 37.5 37.8 55.4 36.9 46.2

[84]AceCoder-RM Coder 22k 78.0 66.4 27.5 13.3 3.3 27.5 62.6 29.4 29.0 57.3 27.5 42.4

[84]AceCoder-Rule Coder 22k 80.5 70.4 29.0 6.7 6.7 40.0 62.8 27.6 27.4 60.0 28.5 44.3

[36]CodeR1-LC2k Ins 2k 81.7 71.7 28.1 13.3 10.0 45.0 75.0 33.5 36.7 60.5 35.6 48.0

[36]CodeR1-12k Ins 12k 81.1 73.5 29.3 13.3 3.3 37.5 74.0 35.7 36.9 61.3 33.5 47.4

**Zero-Style Reasoners Trained on Curated Math Data**

[9] **45.8**PRIME-Zero Coder 484k 49.4 51.1 11.0 23.3 23.3 67.5 81.2 37.9 41.8 37.2 41.5

[85]SimpleRL-Zoo Base 8.5k 73.2 63.2 25.6 16.7 3.3 57.5 77.0 35.7 41.0 54.0 38.5 46.3

[38]Oat-Zero Math 8.5k 62.2 59.0 15.2 30.0 16.7 62.5 80.0 34.9 41.6 45.5 44.3 44.9

[23]ORZ Base 57k 80.5 64.3 22.0 13.3 16.7 60.0 81.8 32.7 45.0 55.6 41.6 48.6

**Absolute Zero Training w/ No Curated Data (Ours)**

74 46713 691 253 133 133 525 4 382 385 552 384 8**0**AZR (Ours) Base -1.9 +3.8 +7.8 +6.6 +10.0 +15.0 +9.6 +13.2 +10.8 +3.2 +10.9 +7.0. . . . . . . . . . . . 72  **50.4**835 696 317 200 100 575 6 364 382 391**0** **61.6**AZR (Ours) Coder +3.0 +0.3 +11.8 +13.3 +6.7 +17.5 +22.6 +19.1 +16.3 +5.0 +15.2 +10.2. . . . . . . . . .

**Performance of RL-Trained Reasoner on Reasoning Benchmarks Based on Qwen2.5-7B Models.**Table 1. Performance of various

+ + v1-5models is evaluated on three standard code benchmarks (HumanEval, MBPP, LCB and six math benchmarks (AIME’24, AIME’25,

AMC’23, MATH500, Minerva, OlympiadBench). Average performance across coding and math benchmarks is calculated as average of

= ( + )2the two averages: AVG CAvg MAvg . We use + for absolute percentage increase from base model. All models are trained using/

different variants of the model, with the variant and data usage labeled, more details listed in Table 4 Qwen2.5-7B

**Baselines.** For our main results, we use as the base model, along with its specialized base model variants: Qwen2.5-7B

, , and (Yang et al., 2024a; Hui et al., 2024; Yang et al., 2024b).Qwen2.5-7B-Coder Qwen2.5-7B-Instruct Qwen2.5-Math-7B

Furthermore, the zero-style models are usually trained specifically on either code or math data; and only (Cui Eurus-2-7B-PRIME-Zero

et al., 2025) was trained jointly on both domains. For code data models, we present four variants of the (Zeng et al., 2025a) AceCoder

and two different models (Liu & Zhang, 2025). For math data models, we have (Liu et al., CodeR1 Qwen2.5-Math-7B-Oat-Zero

2025), (ORZ) (Hu et al., 2025), (Zeng et al., 2025b). All baseline models’ Open-Reasoner-Zero-7B Qwen-2.5-7B-SimpleRL-Zoo

training data and initialization settings are summarized in Table 4. For follow-up scaling experiments, we compare each AZR model

against its own corresponding base model, due to the lack of established baselines across different parameter scales. Finally, we compare

our -trained model with (Zeng et al., 2025b) and the base model. Llama3.1-8B Llama-3.1-8B-SimpleRL-Zoo

**4.2. Results**

**Research Question 1: How does AZR compare to other zero setting models trained with human expert**

**data?** We present the main results of reasoning models trained under both the standard zero and our proposed absolute zero settings

in Table 1. Notably, achieves state-of-the-art performance in both the 7B overall average and Absolute Zero Reasoner-Coder-7B

the coding average categories. Despite being entirely out-of-distribution for both math and code reasoning benchmarks, it surpasses the

previous best model by 1.8 absolute percentages. Even more strikingly, it outperforms models trained with expert-curated human data in

03the coding category by absolute percentages, while never having access to such data itself..

**Strong Cross-domain Generalization.** To assess cross-domain generalization after RLVR, we evaluate math performance before and

after training, comparing AZR models with other expert code models, since AZR was trained in coding environments. After training,

most expert code models showed minimal changes or even declines in performance compared to their base versions, with an average

increase of only 0.65 points across these models, indicating very limited cross-domain generalization. In contrast, AZR base and coder

models achieved gains of 10.9 and 15.2 percentage points, respectively, demonstrating substantially stronger generalized reasoning

improvements. Similarly, although also out-of-distribution on human-defined code generation tasks, our AZR models improved by 3.2

and 5.0 points, while the math models on average showed just a moderate increases in coding (+2.0 on average).

Overall, these results highlight the surprising effectiveness of our approach. Unlike other RLVR models trained and evaluated on

human-defined tasks, our AZR models demonstrate strong general reasoning capabilities without any direct training on downstream

human-defined math or coding data, only had access to self-proposed tasks during training.

**Research Question 2: How do initializing from different base model variants (base vs. coder) affect**

**performance?** As shown in Table 1, the coder variant achieved better overall performance in both math and coding after the AZR

9

**Page 10**

**Absolute Zero: Reinforced Self-play Reasoning with Zero Data**

**Model Family** **Variant** **Code Avg** **Math Avg** **Total Avg**.070

Llama3.1-8b 28.5 3.4 16.0.065In-Distribution Accuracy

[85] 337 72 205Llama3.1-8b + SimpleRL +52 +38 +45. . .. . ..060

316 68 192Llama3.1-8b + AZR (Ours) +31 +34 +32. . .. . ..055

Qwen2.5-3B Coder 51.2 18.8 35.0.050

549 265 407Qwen2.5-3B Coder + AZR (Ours) +37 +77 +57. . .. . ..045

Qwen2.5-7B Coder 56.6 23.9 40.2.040 616 391 504Qwen2.5-7B Coder + AZR (Ours) +50 +152 +102. . .. . .AZR-Llama3.1-8b AZR-7B-Coder.035 AZR-3B-Coder AZR-14B-Coder Qwen2.5-14B Coder 60.0 20.2 40.1

.03050 75 100 125 150 175 200 225 250 636 430 533Qwen2.5-14B Coder + AZR (Ours) +36 +228 +132. . .. . .Training Steps

**(a)** **(b)**

**(a) In-Distribution & (b) Out-of-Distribution Reasoning Task Performances. (a)**Figure 6. Scores on CruxEval-I, CruxEval-O,

and LiveCodeBench-Execution, which correspond to abduction, deduction, and deduction task types respectively, used to evaluate

**(b)**in-distribution abilities of AZR during training across different model sizes and types; Out-of-distribution reasoning performance,

reported as the average of code tasks, math tasks, and their overall average, across different model sizes and types. A detailed breakdown

of all benchmark results can be found in Table 5.

self-play process. Strikingly, although the coder base model variant started with a lower average performance in math than the vanilla

base model (23.9 vs. 27.5), it ultimately outperformed it after AZR training. This highlights the importance of initial code competency

as a catalyst for enhancing broader reasoning abilities within the Absolute Zero Reasoner approach.

**Research Question 3: How does varying model size effect AZR’s in-distribution and out-of-distribution**

**capabilities?** We examine the effects of scaling model size and present both in-distribution and out-of-distribution results in Figure 6

(a) and (b), respectively. Given the strong performance of coder models in the 7B category, we extend the analysis by evaluating smaller

and larger variants: and . Due to the absence of existing baselines for these zero-style Qwen2.5-3B-Coder Qwen2.5-14B-Coder

reasoner models, we compare each model’s performance to its corresponding base coder model.

greater gains on larger, more capable modelsThe results reveal a clear trend: our method delivers . In the in-distribution setting, the 7B

and 14B models continue to improve beyond 200 training steps, whereas the smaller 3B model appears to plateau. For out-of-distribution

domains, larger models also show greater overall performance improvements than smaller ones: +5.7, +10.2, +13.2 overall performance

gains, respectively for 3B, 7B and 14B. This is an encouraging sign, since base models continue to improve and also suggesting that

scaling enhances the effectiveness of AZR. In future work, we aim to investigate the scaling laws that govern performance in the Absolute

Zero paradigm.

**Research Question 4: Any interesting observations by changing the model class?** We also evaluate our method

on a different model class, using as the base shown in Figure 6. Unlike the 3B and 14B categories, this setting has an Llama3.1-8B

existing baseline, (Zeng et al., 2025b), which enables a direct comparison. Although is less capable than SimpleRL Llama3.1-8B

the models, our method still produces moderate improvements (+3.2), demonstrating AZR’s effectiveness even on relatively Qwen2.5

weaker models. However, these gains appear more limited, which aligns with our earlier observation that performance improvements

tend to scale with initial base model potency.

**Research Question 5: Any interesting behaviors or patterns observed during AZR training?** We observed

interesting response patterns in both the proposal and solution stages. The model is capable of proposing diverse programs, such as

e.gstring manipulation tasks, dynamic programming problems, and practical cases ( ., calculating a triangle’s area using Heron’s formula).

We show a concrete example in Figure 7, where AZR proposes a code problem that searches for the sum of continuous sub-arrays

matching a target value and solves it through trial-and-error.

Overall, the models trained exhibits distinct reasoning patterns depending on the task type. For example, when solving abduction tasks,

it repeatedly tests different input patterns, self-correcting until the reasoned output matches the given input. When predicting outputs,

it steps through the code and records structured intermediate results (such as dynamic programming arrays) until the final output is

reached. When inducting programs from given inputs, outputs, and descriptions, the model systematically checks each test case to

confirm that its program produces correct results. We showcase more concrete examples of these behaviors in Figures 18 and 20 to 26.

We also share some fun “vibe checks” such as solving Sudoku and solving the sum-product game in Figures 40 and 41.

**Intermediate Planning During Code Response.** Another interesting pattern emerged in our AZR models during the code induction

task: the final code outputs were often interleaved with comments that resembled immediate step-by-step plans, reminiscent of the ReAct

prompting framework (Yao et al., 2023). A similar behavior has been observed in recent formal math proving models, such as DeepSeek

10

**Page 11**

**Absolute Zero: Reinforced Self-play Reasoning with Zero Data**

Model-proposed Task Response of Predict Input

Let’s analyze the code snippet **…(analysis)** To find one possible input that produces Input: [1,2,3,4,5], 5

the output 1, we need to find a scenario where **…(analysis)**

**Let’s consider a simple example: numbers = [1, 2], target = 1**

**Here’s the step-by-step calculation:**

1. current\_sum = 0 + 1 = 1, accumulated\_values = [1]

2. current\_sum = 1 + 2 = 3, accumulated\_values = [1, 3]

-Check for pair (i=0, j=1): 3 - 1 = 2 (not equal to target)

-Check for pair (i=0, j=0): 1 - 1 = 0 (not equal to target)

-Check for pair (i=1, j=1): 3 - 3 = 0 (not equal to target)

Since no pair satisfies the condition, the function will not return any value. This

doesn’t match our output. Let’s try another example **…(Tried 3 more examples)**

**Given the complexity and the fact that no simple example seems to work, let’s**

**try a more complex scenario: numbers = [2, -1, 1], target = 1**

Step-by-step calculation:

1. current\_sum = 0 + 2 = 2, accumulated\_values = [2]

2. current\_sum = 2 - 1 = 1 , accumulated\_values = [2, 1]

3. current\_sum = 1 + 1 = 2, accumulated\_values = [2, 1, 2]

- Check for pair (i=0, j=1): 1 - 2 = -1 (not equal to target)

- Check for pair (i=0, j=2): 2 - 2 = 0 (not equal to target)

- Check for pair (i=1, j=2): 2 - 1 = 1 (equal to target)

**The pair (i=1, j=2) satisfies the condition, and the function returns the first**

Output: 1 **accumulated value in this pair, which is 1. This matches our output.**

**Example of a Model-Proposed Task and Its Response for Solving an Abduction Task.**Figure 7. (Left) The model autonomously

proposes an input and program for the abduction task. We execute the program to verify its validity and obtain the corresponding output.

(Right) The model’s reasoning process when solving the abduction task: given the code and output, it attempts to infer the original input.

The model begins by analyzing the program, proposes an initial input, and reasons through the code to produce an output. If there is a

mismatch, it reflects on the discrepancy and iteratively adjusts the input until the generated output matches the target. Interestingly, the

agent arrives at a different input than the gold one, but since it produces the correct output, the answer is considered correct.

, which is significantly larger in scale (671B). This pattern suggests that models may naturally adopt intermediate planningProver v2

long-formas a strategy to enhance final answers. Therefore, it may be beneficial to explicitly enable or encourage this behavior in

responses across other domains.

**Cognitive Behavior in Llama.** Interestingly, we also observed some emergent cognitive patterns in Absolute Zero

, similar to those reported by Zeng et al. (2025b), and we include one example in Figure 26, whereReasoner-Llama3.1-8B

clear state-tracking behavior is demonstrated. In addition, we encountered some unusual and potentially concerning chains of thought

from the Llama model trained with AZR. One example includes the output: “The aim is to outsmart all these groups of intelligent

“uh-ohmachines and less intelligent humans. This is for the brains behind the future” shown in Figure 32. We refer to this as the

moment” and encourage future work to further investigate its potential implications.

**Token Length Increase Depends on Task Type.** Finally, we observed that token length increases over the course of training, consistent

with findings from recent studies (Hu et al., 2025; Liu et al., 2025). Interestingly, our results reveal one of the first observation of clear

distinctions in token length growth across different types of cognitive tasks. As shown in Figures 15 to 17, the extent of lengthening

varies by task type. The most significant increase occurs in the abduction task, where the model engages in trial-and-error reasoning by

repeatedly testing inputs to match the program’s output. This suggests that the observed variation in token length is not incidental, but

rather a reflection of task-specific reasoning behavior.

**Research Question 6: Are all task types essential for good performance (Ablation)?** Due to resource constraints,

we perform the ablation studies in this section and the next using only . We begin by testing the Absolute Zero Reasoner-Base-7B

importance of task types during training, with results shown in Table 2. In row 1, both induction and abduction tasks are removed;

in row 2, only the induction task is removed. In both cases, math performance drops significantly, with the most severe degradation

occurring when more task types are excluded. These findings highlight the complementary role of the three task types in improving

general reasoning capability, with each contributing in a distinct and essential way.

**Research Question 7: How much do the designs of proposer contribute to the overall performance**

**(Ablation)?** Next, we ablate two components of the proposer role and present the results in Table 2. First, we examine whether

conditioning on historic reference triplets is necessary. To do so, we design a variant in which a fixed prompt is used to propose abduction

and deduction tasks, rather than dynamically conditioning on historical triplets (row 3). This results in a 5-point absolute drop in K

math performance and a 1-point drop in code performance. This suggest that dynamically conditioning on reference programs helps

11

**Page 12**

**Absolute Zero: Reinforced Self-play Reasoning with Zero Data**

**Code Avg.** **Math Avg.** **Overall Avg.**Experiment Task Type Gen Reference Trained Roles

Deduction only Ded / / 54.6 32.0 43.3

w/o Induction Abd, Ded / / 54.2 33.3 43.8

w/o Gen Reference / 0 / 54.4 33.1 43.8

Train Solver Only / / Solve Only 54.8 36.0 45.4

**Ours** **55.2** **38.4** **46.8**Abd, Ded, Ind Propose & SolveK

**Ablation Results.**Table 2. We ablate task types and the proposer role in the Absolute Zero Reasoner using the 7B base model. A ‘/’

indicates that the configuration remains unchanged from the standard AZR setup. Removing induction or using only deduction leads to

significant performance drops (rows 1 & 2). For the proposer role, both removing conditioning on references (row 3) and omitting K

proposer-role training (row 4) result in degraded performance. Overall, all components are essential for general reasoning.

improve performance, possibly by increasing diversity and achieving better coverage of the reasoning problem space.

Finally, we consider a case where we do not train the proposer at all. Instead, we only prompt it using the current learner and train the

solver alone (row 4). We observe a moderate drop in overall performance (-1.4), suggesting that while proposer training is beneficial, it

may not be the most critical factor for now in the AZR framework. We hypothesize that this could be related to task interference, as

studied in multitask learning literature (Suteu & Guo, 2019). Thus, we believe that further investigation into how to make the proposer

even more potent is an exciting and promising direction.

**Additional Results.** Beyond the core research questions, we present additional results, including the breakdown of individual

out-of-distribution benchmark scores during training for the 7B base and coder models in Figures 28 and 29, for th 14B base and coder

model in Figures 30 and 31. For completeness, we also report in-distribution benchmark performance during training for the 7B base

model in Figure 14. Finally, we invite interested readers to explore Appendix D, where we share several experimental directions that,

while not yielding strong performance gains, produced interesting and insightful findings.

**5. Related Work**

**Reasoning with RL.** Using RL to enhance reasoning capabilities has recently emerged as an important step in the post-training

process of strong reasoning-focused large language models (Lambert et al., 2024). One of the first works to explore a self-bootstrapping

approach to improving LLM reasoning is STaR, which employs expert iteration and rejection sampling of outcome-verified responses to

iteratively improve the model’s CoT. A monumental work, o1 (Jaech et al., 2024), was among the first to deploy this idea on a scale,

achieving state-of-the-art results in reasoning tasks at the time of release. More recently, the R1 model (DeepSeek-AI et al., 2025)

became the first open-weight model to match or even surpass the performance of o1. Most notably, the zero setting was introduced, in

which reinforcement learning is applied directly on top of the base LLM. This inspired followup work, which are open source attempts to

replicate the R1 process or to improve the underlying reinforcement learning algorithm (Zeng et al., 2025b; Liu et al., 2025; Cui et al.,

2025; Hu et al., 2025; Yu et al., 2025; Yuan et al., 2025). Recent work explored RL on human defined procedural generated puzzles saw

improvements in math (Xie et al., 2025), and using one human example can almost match the performance of thousands (Wang et al.,

2025b). We extend the zero setting to a new absolute zero setting, where not only is the RLVR process initialized from a base LLM

without SFT, but no external prompt data or answers are provided to the learner. All data used to improve reasoning were self-proposed,

and refined entirely through RLVR. Moreover, our goal is not to only match zero-setting models, but to surpass them in the long run.

**Self-play.** The self-play paradigm can be traced back to early 2000s, where Schmidhuber (2003; 2011) (of course) explored a

two-agent setup in which a proposal agent invents questions for a prediction agent to answer. This dynamic continuously and automatically

improves both agents, enabling theoretically never-ending progress (Schaul, 2024). AlphaGo and AlphaZero (Silver et al., 2016; 2017)

extend the self-play paradigm to the two-player zero-sum game of Go, where the current learner competes against earlier versions of

itself to progressively enhance its capabilities. These were among the first milestone works to demonstrate superhuman performance

in the game of Go. Moreover, methods such as asymmetric self-play (Sukhbaatar et al., 2018; OpenAI et al., 2021), unsupervised

environment design (Wang et al., 2019; Dennis et al., 2020), unsupervised reinforcement learning (Laskin et al., 2021; Zhao et al., 2022;

2025b), and automatic goal generation (Florensa et al., 2018) all center around inventing new tasks for an agent to learn from—typically

without supervision. In these approaches, the process of setting goals itself is often dynamic and continuously evolving. Generative

adversarial networks (Goodfellow et al., 2020), also belong in this paradigm where a discriminator discriminate between real data and

generated data, and the generated is trained to fool the discriminator.

Most recently, SPIN and Self-Rewarding Language Models (Chen et al., 2024; Yuan et al., 2024) use the same instance of the lanugage

models themselves as the reward model to progressively improve the generative and discriminative abilities of the same LLM for

alignment. (Kirchner et al., 2024) uses Prover-Verifier Game for increasing legibility and eva (Ye et al., 2024) uses self-play for

alignment, but reward model is the main bottleneck as it is not reliable for reasoning tasks (Lambert et al., 2024). SPC (Chen et al.,

12

**Page 13**

**Absolute Zero: Reinforced Self-play Reasoning with Zero Data**

2025) used self-play to train on human-curated tasks to increase the critic capabilities and SPAG (Cheng et al., 2024) trained using

self-play in specific game of Adversarial Taboo. Concurrent works—Genius, EMPO, and TTRL (Xu et al., 2025; Zhang et al., 2025b;

Zuo et al., 2025)—leverage human-curated language queries without labels to train reinforcement learning agents, but still rely on a

fixed human defined learning task distribution. Finally, Minimo (Poesia et al., 2024) extends self-play to formal mathematics, where a

pair of conjecture- and theorem-proving agents are jointly trained using reinforcement learning. Our work builds upon the self-play

paradigm, but it is the first to use it to elicit long CoT for improved reasoning, and the first to frame the problem space as a Python

input/output/function abduction/deduction/induction tasks, grounding it in an operationalizable environment to facilitate RLVR.

**Weak-to-Strong Supervision.** The concept of weak-to-strong supervision has been studied in prior work, where a teacher—despite

being weaker than the learner—still provides useful guidance (Burns et al., 2024; Hinton et al., 2015; Christiano, 2018; 2019; Demski &

Garrabrant, 2019; Leike & Sutskever, 2023; Hubinger et al., 2019). We consider a similar setting in which the learner may possess

superhuman capabilities. However, rather than relying on supervision from a weaker teacher, we propose an alternative approach:

guiding the learner’s improvement through verifiable rewards, which potentially offer a more reliable and scalable learning signal.

Furthermore, in our proposed method, the learning task and goal distribution is not predefined by any external supervisor—they are

entirely self-generated by the learner, enabling it to maximize its learning potential through autonomous self-practice.

**6. Conclusion and Discussion**

**Conclusion.** In this work, we proposed the Absolute Zero paradigm, a novel setting that addresses the data limitations of existing

RLVR frameworks. In this paradigm, reasoning agents are tasked with generating their own learning task distributions and improving

their reasoning abilities with environmental guidance. We then presented our own instantiation, the Absolute Zero Reasoner (AZR),

which is trained by having them propose and solve code-related reasoning tasks grounded by code executor.

We evaluated our trained models on out-of-distribution benchmarks in both the code generation and mathematical reasoning domains.

Remarkably, even though our models were not directly trained on these tasks and lacked human expert-curated datasets, our reasoning

agents achieved exceptional performance, surpassing the state-of-the-art in combined general reasoning scores and in coding. This

demonstrates the potential of the absolute zero paradigm to drive superior reasoning capabilities without the need for extensive

domain-specific training data. Furthermore, we showed that AZR scales efficiently, offering strong performance across varying model

sizes, and can enhance the capabilities of other model classes as well. To foster further exploration and advancement of this emerging

paradigm, we are releasing the code, models, and logs as open-source, encouraging the research community to build upon our findings.

**Discussion.** We believe there remains much to explore, such as altering the environment from which the reasoner receives verifiable

feedback, including sources like the world wide web, formal math languages (Sutton, 2001; Ren et al., 2025), world simulators, or even

the real world. Furthermore, AZ’s generality could possibly be extend to domains such as embodied AI (Zitkovich et al., 2023; Yue

et al., 2024). Additionally, more complex agentic tasks or scientific experiments, present exciting opportunities to further advance the

absolute zero setting to different application domains (Wu et al., 2024; 2023). Beyond that, future directions could include exploring

()multimodal reasoning models, modifying the distribution to incorporate privileged information, defining or even let the model pz

dynamically learn how to define (Equation (3)), or designing exploration/diversity rewards for both the propose and solve roles. f

While underappreciated in current reasoning literature, the exploration component of RL has long been recognized as a critical driver for

emergent behavior in traditional RL (Yue et al., 2025; Silver et al., 2016; Ladosz et al., 2022). Years of research have examined various

forms of exploration, even in related subfields using LLMs such as red teaming (Zhao et al., 2025a), yet its role in LLM reasoning

models remains underexplored. Taking this a step further, our framework investigates an even more meta-level exploration problem:

exploration within the learning task space—where the agent learns not just how to solve tasks, but what tasks to learn from and how to

find them. Rather than being confined to a fixed problem set, AI reasoner agents may benefit from dynamically defining and refining

their own learning tasks. This shift opens a powerful new frontier—where agents explore not only solution spaces but also expand the

boundaries of problem spaces. We believe this is a promising and important direction for future research.

One limitation of our work is that we did not address how to safely manage a system composed of such self-improving components.

To our surprise, we observed several instances of safety-concerning CoT from the model, which we term the “uh-oh Llama-3.1-8B

moment”. These findings suggest that the proposed absolute zero paradigm, while reducing the need for human intervention for curating

tasks, still necessitates oversight due to lingering safety concerns and is a critical direction for future research (Wang et al., 2024; 2025a).

As a final note, we explored reasoning models that possess experience—models that not only solve given tasks, but also define and

evolve their own learning task distributions with the help of an environment. Our results with AZR show that this shift enables strong

performance across diverse reasoning tasks, even with significantly fewer privileged resources, such as curated human data. We believe

this could finally free reasoning models from the constraints of human-curated data (Morris, 2025) and marks the beginning of a new

**“welcome to the era of experience”**chapter for reasoning models: (Silver & Sutton, 2025; Zhao et al., 2024).

13

**Page 14**

**Absolute Zero: Reinforced Self-play Reasoning with Zero Data**

**References**

Aryabumi, V., Su, Y., Ma, R., Morisot, A., Zhang, I., Locatelli, A., Fadaee, M., Üstün, A., and Hooker, S. To code, or not

CoRRto code? exploring impact of code in pre-training. , abs/2408.10914, 2024. doi: 10.48550/ARXIV.2408.10914. URL

.https://doi.org/10.48550/arXiv.2408.10914

Burns, C., Izmailov, P., Kirchner, J. H., Baker, B., Gao, L., Aschenbrenner, L., Chen, Y., Ecoffet, A., Joglekar, M., Leike, J.,

Forty-firstSutskever, I., and Wu, J. Weak-to-strong generalization: Eliciting strong capabilities with weak supervision. In

International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024. OpenReview.net, 2024. URL

.https://openreview.net/forum?id=ghNRg2mEgN

Canal, M. Radon: Python tool for code metrics. , 2023. Accessed: 2025-04-06. https://github.com/rubik/radon

Chen, J., Zhang, B., Ma, R., Wang, P., Liang, X., Tu, Z., Li, X., and Wong, K.-Y. K. Spc: Evolving self-play critic via adversarial games

for llm reasoning, 2025. URL . https://arxiv.org/abs/2504.19162

Chen, Z., Deng, Y., Yuan, H., Ji, K., and Gu, Q. Self-play fine-tuning converts weak language models to strong language models. In

Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024. OpenReview.net, 2024.

URL . https://openreview.net/forum?id=O4cHTxW9BS

Cheng, P., Hu, T., Xu, H., Zhang, Z., Dai, Y., Han, L., Du, N., and Li, X. Self-playing adversarial language game enhances LLM

Advances inreasoning. In Globersons, A., Mackey, L., Belgrave, D., Fan, A., Paquet, U., Tomczak, J. M., and Zhang, C. (eds.),

Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024,

Vancouver, BC, Canada, December 10 - 15, 2024, 2024. URL http://papers.nips.cc/paper\_files/paper/2024/hash/

.e4be7e9867ef163563f4a5e90cec478f-Abstract-Conference.html

Christiano, P. Approval-directed bootstrapping. https://www.alignmentforum.org/posts/6x7oExXi32ot6HjJv/

, 2018. AI Alignment Forum.approval-directed-bootstrapping

Christiano, P. Capability amplification. https://www.alignmentforum.org/posts/t3AJW5jP3sk36aGoC/

, 2019. AI Alignment Forum.capability-amplification-1

Cui, G., Yuan, L., Wang, Z., Wang, H., Li, W., He, B., Fan, Y., Yu, T., Xu, Q., Chen, W., Yuan, J., Chen, H., Zhang, K., Lv, X., Wang, S.,

Yao, Y., Han, X., Peng, H., Cheng, Y., Liu, Z., Sun, M., Zhou, B., and Ding, N. Process reinforcement through implicit rewards.

CoRR, abs/2502.01456, 2025. doi: 10.48550/ARXIV.2502.01456. URL . https://doi.org/10.48550/arXiv.2502.01456

DeepSeek-AI, Guo, D., Yang, D., Zhang, H., Song, J., Zhang, R., Xu, R., Zhu, Q., Ma, S., Wang, P., Bi, X., Zhang, X., Yu, X., Wu, Y.,

Wu, Z. F., Gou, Z., Shao, Z., Li, Z., Gao, Z., Liu, A., Xue, B., Wang, B., Wu, B., Feng, B., Lu, C., Zhao, C., Deng, C., Zhang, C.,

Ruan, C., Dai, D., Chen, D., Ji, D., Li, E., Lin, F., Dai, F., Luo, F., Hao, G., Chen, G., Li, G., Zhang, H., Bao, H., Xu, H., Wang, H.,

Ding, H., Xin, H., Gao, H., Qu, H., Li, H., Guo, J., Li, J., Wang, J., Chen, J., Yuan, J., Qiu, J., Li, J., Cai, J. L., Ni, J., Liang, J., Chen,

J., Dong, K., Hu, K., Gao, K., Guan, K., Huang, K., Yu, K., Wang, L., Zhang, L., Zhao, L., Wang, L., Zhang, L., Xu, L., Xia, L.,

Zhang, M., Zhang, M., Tang, M., Li, M., Wang, M., Li, M., Tian, N., Huang, P., Zhang, P., Wang, Q., Chen, Q., Du, Q., Ge, R.,

Zhang, R., Pan, R., Wang, R., Chen, R. J., Jin, R. L., Chen, R., Lu, S., Zhou, S., Chen, S., Ye, S., Wang, S., Yu, S., Zhou, S., Pan, S.,

CoRRand Li, S. S. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. , abs/2501.12948, 2025. doi:

10.48550/ARXIV.2501.12948. URL . https://doi.org/10.48550/arXiv.2501.12948

CoRRDemski, A. and Garrabrant, S. Embedded agency. , abs/1902.09469, 2019. URL . http://arxiv.org/abs/1902.09469

Dennis, M., Jaques, N., Vinitsky, E., Bayen, A. M., Russell, S., Critch, A., and Levine, S. Emergent complexity and zero-

shot transfer via unsupervised environment design. In Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M., and Lin, H.

Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems(eds.),

2020, NeurIPS 2020, December 6-12, 2020, virtual, 2020. URL https://proceedings.neurips.cc/paper/2020/hash/

.985e9a46e10005356bbaf194249f6856-Abstract.html

Dubey, A., Jauhri, A., Pandey, A., Kadian, A., Al-Dahle, A., Letman, A., Mathur, A., Schelten, A., Yang, A., Fan, A., Goyal, A.,

Hartshorn, A., Yang, A., Mitra, A., Sravankumar, A., Korenev, A., Hinsvark, A., Rao, A., Zhang, A., Rodriguez, A., Gregerson, A.,

Spataru, A., Rozière, B., Biron, B., Tang, B., Chern, B., Caucheteux, C., Nayak, C., Bi, C., Marra, C., McConnell, C., Keller, C.,

Touret, C., Wu, C., Wong, C., Ferrer, C. C., Nikolaidis, C., Allonsius, D., Song, D., Pintz, D., Livshits, D., Esiobu, D., Choudhary,

D., Mahajan, D., Garcia-Olano, D., Perino, D., Hupkes, D., Lakomkin, E., AlBadawy, E., Lobanova, E., Dinan, E., Smith, E. M.,

Radenovic, F., Zhang, F., Synnaeve, G., Lee, G., Anderson, G. L., Nail, G., Mialon, G., Pang, G., Cucurell, G., Nguyen, H., Korevaar,

H., Xu, H., Touvron, H., Zarov, I., Ibarra, I. A., Kloumann, I. M., Misra, I., Evtimov, I., Copet, J., Lee, J., Geffert, J., Vranes, J.,

Park, J., Mahadeokar, J., Shah, J., van der Linde, J., Billock, J., Hong, J., Lee, J., Fu, J., Chi, J., Huang, J., Liu, J., Wang, J., Yu,

J., Bitton, J., Spisak, J., Park, J., Rocca, J., Johnstun, J., Saxe, J., Jia, J., Alwala, K. V., Upasani, K., Plawiak, K., Li, K., Heafield,

14

**Page 15**

**Absolute Zero: Reinforced Self-play Reasoning with Zero Data**

CoRRK., Stone, K., and et al. The llama 3 herd of models. , abs/2407.21783, 2024. doi: 10.48550/ARXIV.2407.21783. URL

.https://doi.org/10.48550/arXiv.2407.21783

IEEE softwareEbert, C., Cain, J., Antoniol, G., Counsell, S., and Laplante, P. Cyclomatic complexity. , 33(6):27–29, 2016.

Florensa, C., Held, D., Geng, X., and Abbeel, P. Automatic goal generation for reinforcement learning agents. In Dy, J. G. and

Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan,Krause, A. (eds.),

Stockholm, Sweden, July 10-15, 2018 Proceedings of Machine Learning Research, volume 80 of , pp. 1514–1523. PMLR, 2018. URL

.http://proceedings.mlr.press/v80/florensa18a.html

Goodfellow, I. J., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A. C., and Bengio, Y. Generative

Commun. ACMadversarial networks. , 63(11):139–144, 2020. doi: 10.1145/3422622. URL . https://doi.org/10.1145/3422622

Gu, A., Rozière, B., Leather, H. J., Solar-Lezama, A., Synnaeve, G., and Wang, S. Cruxeval: A benchmark for code reasoning,

Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27,understanding and execution. In

2024. OpenReview.net, 2024. URL . https://openreview.net/forum?id=Ffpg52swvg

Elements of Software Science (Operating and programming systems series)Halstead, M. H. . Elsevier Science Inc., 1977.

He, C., Luo, R., Bai, Y., Hu, S., Thai, Z. L., Shen, J., Hu, J., Han, X., Huang, Y., Zhang, Y., Liu, J., Qi, L., Liu, Z., and Sun, M.

Olympiadbench: A challenging benchmark for promoting AGI with olympiad-level bilingual multimodal scientific problems. In Ku,

Proceedings of the 62nd Annual Meeting of the Association for Computational LinguisticsL., Martins, A., and Srikumar, V. (eds.),

(Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024, pp. 3828–3850. Association for Computational

Linguistics, 2024. doi: 10.18653/V1/2024.ACL-LONG.211. URL . https://doi.org/10.18653/v1/2024.acl-long.211

Hendrycks, D., Burns, C., Kadavath, S., Arora, A., Basart, S., Tang, E., Song, D., and Steinhardt, J. Measuring math-

Proceedings of the Neu-ematical problem solving with the MATH dataset. In Vanschoren, J. and Yeung, S. (eds.),

ral Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021,

December 2021, virtual, 2021. URL https://datasets-benchmarks-proceedings.neurips.cc/paper/2021/hash/

.be83ab3ecd0db773eb2dc1b0a17836a1-Abstract-round2.html

CoRRHinton, G. E., Vinyals, O., and Dean, J. Distilling the knowledge in a neural network. , abs/1503.02531, 2015. URL

.http://arxiv.org/abs/1503.02531

CoRRHu, J. REINFORCE++: A simple and efficient approach for aligning large language models. , abs/2501.03262, 2025. doi:

10.48550/ARXIV.2501.03262. URL . https://doi.org/10.48550/arXiv.2501.03262

Hu, J., Zhang, Y., Han, Q., Jiang, D., Zhang, X., and Shum, H. Open-reasoner-zero: An open source approach to scaling up

CoRRreinforcement learning on the base model. , abs/2503.24290, 2025. doi: 10.48550/ARXIV.2503.24290. URL https:

.//doi.org/10.48550/arXiv.2503.24290

Hubinger, E., van Merwĳk, C., Mikulik, V., Skalse, J., and Garrabrant, S. Risks from learned optimization in advanced machine learning

CoRRsystems. , abs/1906.01820, 2019. URL . http://arxiv.org/abs/1906.01820

Hughes, E., Dennis, M. D., Parker-Holder, J., Behbahani, F. M. P., Mavalankar, A., Shi, Y., Schaul, T., and Rocktäschel, T. Position:

Forty-first International Conference on Machine Learning,Open-endedness is essential for artificial superhuman intelligence. In

ICML 2024, Vienna, Austria, July 21-27, 2024. OpenReview.net, 2024. URL . https://openreview.net/forum?id=Bc4vZ2CX7E

Hui, B., Yang, J., Cui, Z., Yang, J., Liu, D., Zhang, L., Liu, T., Zhang, J., Yu, B., Dang, K., Yang, A., Men, R., Huang, F., Ren, X., Ren,

CoRRX., Zhou, J., and Lin, J. Qwen2.5-coder technical report. , abs/2409.12186, 2024. doi: 10.48550/ARXIV.2409.12186. URL

.https://doi.org/10.48550/arXiv.2409.12186

Jaech, A., Kalai, A., Lerer, A., Richardson, A., El-Kishky, A., Low, A., Helyar, A., Madry, A., Beutel, A., Carney, A., et al. Openai o1

arXiv preprint arXiv:2412.16720system card. , 2024.

Jain, N., Han, K., Gu, A., Li, W., Yan, F., Zhang, T., Wang, S., Solar-Lezama, A., Sen, K., and Stoica, I. Livecodebench: Holistic and

CoRRcontamination free evaluation of large language models for code. , abs/2403.07974, 2024. doi: 10.48550/ARXIV.2403.07974.

URL . https://doi.org/10.48550/arXiv.2403.07974

Kirchner, J. H., Chen, Y., Edwards, H., Leike, J., McAleese, N., and Burda, Y. Prover-verifier games improve legibility of LLM outputs.

CoRR, abs/2407.13692, 2024. doi: 10.48550/ARXIV.2407.13692. URL . https://doi.org/10.48550/arXiv.2407.13692

Inf. FusionLadosz, P., Weng, L., Kim, M., and Oh, H. Exploration in deep reinforcement learning: A survey. , 85:1–22, 2022. doi:

10.1016/J.INFFUS.2022.03.003. URL . https://doi.org/10.1016/j.inffus.2022.03.003

15

**Page 16**

**Absolute Zero: Reinforced Self-play Reasoning with Zero Data**

Lambert, N., Morrison, J., Pyatkin, V., Huang, S., Ivison, H., Brahman, F., Miranda, L. J. V., Liu, A., Dziri, N., Lyu, S., Gu, Y., Malik, S.,

Graf, V., Hwang, J. D., Yang, J., Bras, R. L., Tafjord, O., Wilhelm, C., Soldaini, L., Smith, N. A., Wang, Y., Dasigi, P., and Hajishirzi,

CoRRH. Tülu 3: Pushing frontiers in open language model post-training. , abs/2411.15124, 2024. doi: 10.48550/ARXIV.2411.15124.

URL . https://doi.org/10.48550/arXiv.2411.15124

Laskin, M., Yarats, D., Liu, H., Lee, K., Zhan, A., Lu, K., Cang, C., Pinto, L., and Abbeel, P. URLB: unsu-

Proceedings of the Neural In-pervised reinforcement learning benchmark. In Vanschoren, J. and Yeung, S. (eds.),

formation Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, De-

cember 2021, virtual, 2021. URL https://datasets-benchmarks-proceedings.neurips.cc/paper/2021/hash/

.091d584fced301b442654dd8c23b3fc9-Abstract-round2.html

Leike, J. and Sutskever, I. Introducing superalignment. , 2023. https://openai.com/index/introducing-superalignment/

OpenAI Blog.

CoRRLi, J., Guo, D., Yang, D., Xu, R., Wu, Y., and He, J. Codei/o: Condensing reasoning patterns via code input-output prediction. ,

abs/2502.07316, 2025. doi: 10.48550/ARXIV.2502.07316. URL . https://doi.org/10.48550/arXiv.2502.07316

Li, R., Fu, J., Zhang, B., Huang, T., Sun, Z., Lyu, C., Liu, G., Jin, Z., and Li, G. TACO: topics in algorithmic code generation dataset.

CoRR, abs/2312.14852, 2023. doi: 10.48550/ARXIV.2312.14852. URL . https://doi.org/10.48550/arXiv.2312.14852

GitHubLiu, J. and Zhang, L. Code-r1: Reproducing r1 for code with reliable rewards. , 2025.

Liu, J., Xia, C. S., Wang, Y., and Zhang, L. Is your code generated by chatGPT really correct? rigorous evaluation of large

Thirty-seventh Conference on Neural Information Processing Systemslanguage models for code generation. In , 2023. URL

.https://openreview.net/forum?id=1qvx610Cu7

Liu, Z., Chen, C., Li, W., Qi, P., Pang, T., Du, C., Lee, W. S., and Lin, M. Understanding r1-zero-like training: A critical perspective.

CoRR, abs/2503.20783, 2025. doi: 10.48550/ARXIV.2503.20783. URL . https://doi.org/10.48550/arXiv.2503.20783

Lopez, R. H. Q. Complexipy: An extremely fast python library to calculate the cognitive complexity of python files, written in rust,

2025. URL . Accessed: 2025-04-06. https://github.com/rohaquinlop/complexipy

7th International Conference on Learning Representations, ICLRLoshchilov, I. and Hutter, F. Decoupled weight decay regularization. In

2019, New Orleans, LA, USA, May 6-9, 2019. OpenReview.net, 2019. URL . https://openreview.net/forum?id=Bkg6RiCqY7

Morris, J. There are no new ideas in ai. . . only new datasets. , https://blog.jxmo.io/p/there-are-no-new-ideas-in-ai-only

2025.

OpenAI. Openai o3-mini, January 2025a. URL . Accessed: 2025-04-17. https://openai.com/index/openai-o3-mini/

OpenAI. Introducing openai o3 and o4-mini, April 2025b. URL . https://openai.com/index/introducing-o3-and-o4-mini/

Accessed: 2025-04-17.

OpenAI, Plappert, M., Sampedro, R., Xu, T., Akkaya, I., Kosaraju, V., Welinder, P., D’Sa, R., Petron, A., de Oliveira Pinto, H. P.,

Paino, A., Noh, H., Weng, L., Yuan, Q., Chu, C., and Zaremba, W. Asymmetric self-play for automatic goal discovery in robotic

CoRRmanipulation. , abs/2101.04882, 2021. URL . https://arxiv.org/abs/2101.04882

Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C., Mishkin, P., Zhang, C., Agarwal, S., Slama, K., Ray, A., et al. Training

Advances in neural information processing systemslanguage models to follow instructions with human feedback. , 35:27730–27744,

2022.

Poesia, G., Broman, D., Haber, N., and Goodman, N. D. Learning formal mathematics from intrinsic motivation. In Glober-

Advances in Neural In-sons, A., Mackey, L., Belgrave, D., Fan, A., Paquet, U., Tomczak, J. M., and Zhang, C. (eds.),

formation Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Van-

couver, BC, Canada, December 10 - 15, 2024, 2024. URL http://papers.nips.cc/paper\_files/paper/2024/hash/

.4b8001fc75f0532827472ea5a16af9ca-Abstract-Conference.html

OpenAIRadford, A., Wu, J., Child, R., Luan, D., Amodei, D., Sutskever, I., et al. Language models are unsupervised multitask learners.

blog, 1(8):9, 2019.

Ren, Z. Z., Shao, Z., Song, J., Xin, H., Wang, H., Zhao, W., Zhang, L., Fu, Z., Zhu, Q., Yang, D., Wu, Z. F., Gou, Z., Ma, S., Tang, H.,

Liu, Y., Gao, W., Guo, D., and Ruan, C. Deepseek-prover-v2: Advancing formal mathematical reasoning via reinforcement learning

for subgoal decomposition, 2025. URL . https://arxiv.org/abs/2504.21801

arXiv preprint arXiv:2411.16905Schaul, T. Boundless socratic learning with language games. , 2024.

16

**Page 17**

**Absolute Zero: Reinforced Self-play Reasoning with Zero Data**

Advances in evolutionary computing: theory and applicationsSchmidhuber, J. Exploring the predictable. In , pp. 579–612. Springer,

2003.

Schmidhuber, J. POWERPLAY: training an increasingly general problem solver by continually searching for the simplest still unsolvable

CoRRproblem. , abs/1112.5309, 2011. URL . http://arxiv.org/abs/1112.5309

Shao, Z., Wang, P., Zhu, Q., Xu, R., Song, J., Zhang, M., Li, Y. K., Wu, Y., and Guo, D. Deepseekmath: Pushing the limits

CoRRof mathematical reasoning in open language models. , abs/2402.03300, 2024. doi: 10.48550/ARXIV.2402.03300. URL

.https://doi.org/10.48550/arXiv.2402.03300

Sheng, G., Zhang, C., Ye, Z., Wu, X., Zhang, W., Zhang, R., Peng, Y., Lin, H., and Wu, C. Hybridflow: A flexible and efficient

Proceedings of the Twentieth European Conference on Computer Systems, EuroSys 2025, Rotterdam,RLHF framework. In

The Netherlands, 30 March 2025 - 3 April 2025, pp. 1279–1297. ACM, 2025. doi: 10.1145/3689031.3696075. URL https:

.//doi.org/10.1145/3689031.3696075

Silver, D. and Sutton, R. S. The era of experience. https://storage.googleapis.com/deepmind-media/Era-of-Experience%

, 2025.20/The%20Era%20of%20Experience%20Paper.pdf

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam,

V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T. P., Leach, M., Kavukcuoglu, K.,

Nat.Graepel, T., and Hassabis, D. Mastering the game of go with deep neural networks and tree search. , 529(7587):484–489, 2016.

doi: 10.1038/NATURE16961. URL . https://doi.org/10.1038/nature16961

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap,

CoRRT. P., Simonyan, K., and Hassabis, D. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. ,

abs/1712.01815, 2017. URL . http://arxiv.org/abs/1712.01815

Understanding computation - from simple machines to impossible programsStuart, T. . O’Reilly, 2015. ISBN 978-1-449-32927-3. URL

.http://www.oreilly.de/catalog/9781449329273/index.html

Sukhbaatar, S., Lin, Z., Kostrikov, I., Synnaeve, G., Szlam, A., and Fergus, R. Intrinsic motivation and automatic curricula via

6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 -asymmetric self-play. In

May 3, 2018, Conference Track Proceedings. OpenReview.net, 2018. URL . https://openreview.net/forum?id=SkT5Yg-RZ

CoRRSuteu, M. and Guo, Y. Regularizing deep multi-task networks using orthogonal gradients. , abs/1912.06844, 2019. URL

.http://arxiv.org/abs/1912.06844

Sutskever, I., Vinyals, O., and Le, Q. V. Neurips 2024 test of time award session: Sequence to sequence learning with neural networks.

Conference session, December 2024. URL . https://neurips.cc/virtual/2024/test-of-time/105032

Sutton, R. S. Verification, the key to ai. , 2001. http://incompleteideas.net/IncIdeas/KeytoAI.html

Team, K., Du, A., Gao, B., Xing, B., Jiang, C., Chen, C., Li, C., Xiao, C., Du, C., Liao, C., Tang, C., Wang, C., Zhang, D., Yuan, E., Lu,

E., Tang, F., Sung, F., Wei, G., Lai, G., Guo, H., Zhu, H., Ding, H., Hu, H., Yang, H., Zhang, H., Yao, H., Zhao, H., Lu, H., Li, H., Yu,

H., Gao, H., Zheng, H., Yuan, H., Chen, J., Guo, J., Su, J., Wang, J., Zhao, J., Zhang, J., Liu, J., Yan, J., Wu, J., Shi, L., Ye, L., Yu, L.,

Dong, M., Zhang, N., Ma, N., Pan, Q., Gong, Q., Liu, S., Ma, S., Wei, S., Cao, S., Huang, S., Jiang, T., Gao, W., Xiong, W., He, W.,

Huang, W., Wu, W., He, W., Wei, X., Jia, X., Wu, X., Xu, X., Zu, X., Zhou, X., Pan, X., Charles, Y., Li, Y., Hu, Y., Liu, Y., Chen,

Y., Wang, Y., Liu, Y., Qin, Y., Liu, Y., Yang, Y., Bao, Y., Du, Y., Wu, Y., Wang, Y., Zhou, Z., Wang, Z., Li, Z., Zhu, Z., Zhang,

CoRRZ., Wang, Z., Yang, Z., Huang, Z., Huang, Z., Xu, Z., and Yang, Z. Kimi k1.5: Scaling reinforcement learning with llms. ,

abs/2501.12599, 2025. doi: 10.48550/ARXIV.2501.12599. URL . https://doi.org/10.48550/arXiv.2501.12599

Villalobos, P., Ho, A., Sevilla, J., Besiroglu, T., Heim, L., and Hobbhahn, M. Position: Will we run out of data? limits of LLM scaling

Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, Julybased on human-generated data. In

21-27, 2024. OpenReview.net, 2024. URL . https://openreview.net/forum?id=ViZcgDQjyG

Wang, H., Yue, Y., Lu, R., Shi, J., Zhao, A., Wang, S., Song, S., and Huang, G. Model surgery: Modulating LLM‘s behavior via

Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association forsimple parameter editing. In

Computational Linguistics, pp. 6337–6357, 2025a.

Wang, R., Lehman, J., Clune, J., and Stanley, K. O. Paired open-ended trailblazer (POET): endlessly generating increasingly complex

CoRRand diverse learning environments and their solutions. , abs/1901.01753, 2019. URL . http://arxiv.org/abs/1901.01753

Wang, S., Yang, Q., Gao, J., Lin, M. G., Chen, H., Wu, L., Jia, N., Song, S., and Huang, G. Train once, get a family: State-adaptive

Thirty-seventh Conference on Neural Information Processing Systemsbalances for offline-to-online reinforcement learning. In , 2023.

URL . https://openreview.net/forum?id=vtoY8qJjTR

17

**Page 18**

**Absolute Zero: Reinforced Self-play Reasoning with Zero Data**

Wang, S., Liu, C., Zheng, Z., Qi, S., Chen, S., Yang, Q., Zhao, A., Wang, C., Song, S., and Huang, G. Boosting LLM agents with recursive

Findings of the Association forcontemplation for effective deception handling. In Ku, L.-W., Martins, A., and Srikumar, V. (eds.),

Computational Linguistics: ACL 2024, pp. 9909–9953, Bangkok, Thailand, August 2024. Association for Computational Linguistics.

doi: 10.18653/v1/2024.findings-acl.591. URL . https://aclanthology.org/2024.findings-acl.591/

Wang, Y., Yang, Q., Zeng, Z., Ren, L., Liu, L., Peng, B., Cheng, H., He, X., Wang, K., Gao, J., Chen, W., Wang, S., Du,

S. S., and Shen, Y. Reinforcement learning for reasoning in large language models with one training example, 2025b. URL

.https://arxiv.org/abs/2504.20571

Wu, Q., Bansal, G., Zhang, J., Wu, Y., Zhang, S., Zhu, E., Li, B., Jiang, L., Zhang, X., and Wang, C. Autogen: Enabling next-gen

CoRRLLM applications via multi-agent conversation framework. , abs/2308.08155, 2023. doi: 10.48550/ARXIV.2308.08155. URL

.https://doi.org/10.48550/arXiv.2308.08155

CoRRWu, Y., Yue, T., Zhang, S., Wang, C., and Wu, Q. Stateflow: Enhancing LLM task-solving through state-driven workflows. ,

abs/2403.11322, 2024. doi: 10.48550/ARXIV.2403.11322. URL . https://doi.org/10.48550/arXiv.2403.11322

Xie, T., Gao, Z., Ren, Q., Luo, H., Hong, Y., Dai, B., Zhou, J., Qiu, K., Wu, Z., and Luo, C. Logic-rl: Unleashing LLM

CoRRreasoning with rule-based reinforcement learning. , abs/2502.14768, 2025. doi: 10.48550/ARXIV.2502.14768. URL

.https://doi.org/10.48550/arXiv.2502.14768

Xu, F., Yan, H., Ma, C., Zhao, H., Sun, Q., Cheng, K., He, J., Liu, J., and Wu, Z. Genius: A generalizable and purely unsupervised

self-training framework for advanced reasoning, 2025. URL . https://arxiv.org/abs/2504.08672

Yang, A., Yang, B., Zhang, B., Hui, B., Zheng, B., Yu, B., Li, C., Liu, D., Huang, F., Wei, H., Lin, H., Yang, J., Tu, J., Zhang, J., Yang,

J., Yang, J., Zhou, J., Lin, J., Dang, K., Lu, K., Bao, K., Yang, K., Yu, L., Li, M., Xue, M., Zhang, P., Zhu, Q., Men, R., Lin, R., Li, T.,

Xia, T., Ren, X., Ren, X., Fan, Y., Su, Y., Zhang, Y., Wan, Y., Liu, Y., Cui, Z., Zhang, Z., and Qiu, Z. Qwen2.5 technical report.

CoRR, abs/2412.15115, 2024a. doi: 10.48550/ARXIV.2412.15115. URL . https://doi.org/10.48550/arXiv.2412.15115

Yang, A., Zhang, B., Hui, B., Gao, B., Yu, B., Li, C., Liu, D., Tu, J., Zhou, J., Lin, J., Lu, K., Xue, M., Lin, R., Liu, T., Ren, X., and

CoRRZhang, Z. Qwen2.5-math technical report: Toward mathematical expert model via self-improvement. , abs/2409.12122, 2024b.

doi: 10.48550/ARXIV.2409.12122. URL . https://doi.org/10.48550/arXiv.2409.12122

Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K. R., and Cao, Y. React: Synergizing reasoning and acting in language

The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023models. In .

OpenReview.net, 2023. URL . https://openreview.net/forum?id=WE\_vluYUL-X

CoRRYe, Z., Agarwal, R., Liu, T., Joshi, R., Velury, S., Le, Q. V., Tan, Q., and Liu, Y. Evolving alignment via asymmetric self-play. ,

abs/2411.00062, 2024. doi: 10.48550/ARXIV.2411.00062. URL . https://doi.org/10.48550/arXiv.2411.00062

Yu, Q., Zhang, Z., Zhu, R., Yuan, Y., Zuo, X., Yue, Y., Fan, T., Liu, G., Liu, L., Liu, X., Lin, H., Lin, Z., Ma, B., Sheng, G., Tong, Y.,

Zhang, C., Zhang, M., Zhang, W., Zhu, H., Zhu, J., Chen, J., Chen, J., Wang, C., Yu, H., Dai, W., Song, Y., Wei, X., Zhou, H., Liu, J.,

Ma, W., Zhang, Y., Yan, L., Qiao, M., Wu, Y., and Wang, M. DAPO: an open-source LLM reinforcement learning system at scale.

CoRR, abs/2503.14476, 2025. doi: 10.48550/ARXIV.2503.14476. URL . https://doi.org/10.48550/arXiv.2503.14476

URL https://arxiv.Yuan, W., Pang, R. Y., Cho, K., Li, X., Sukhbaatar, S., Xu, J., and Weston, J. Self-rewarding language models.

org/abs/2401.10020, 2024.

Yuan, Y., Yu, Q., Zuo, X., Zhu, R., Xu, W., Chen, J., Wang, C., Fan, T., Du, Z., Wei, X., et al. Vapo: Efficient and reliable reinforcement

arXiv preprint arXiv:2504.05118learning for advanced reasoning tasks. , 2025.

Yue, Y., Lu, R., Kang, B., Song, S., and Huang, G. Understanding, predicting and better resolving q-value divergence in offline-rl.

Advances in Neural Information Processing Systems, 36:60247–60277, 2023.

Yue, Y., Wang, Y., Kang, B., Han, Y., Wang, S., Song, S., Feng, J., and Huang, G. Deer-vla: Dynamic inference of multimodal large

language models for efficient robot execution. In Globersons, A., Mackey, L., Belgrave, D., Fan, A., Paquet, U., Tomczak, J. M., and

Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information ProcessingZhang, C. (eds.),

Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024, 2024. URL http://papers.nips.cc/paper\_

.files/paper/2024/hash/67b0e7c7c2a5780aeefe3b79caac106e-Abstract-Conference.html

Yue, Y., Chen, Z., Lu, R., Zhao, A., Wang, Z., Yue, Y., Song, S., and Huang, G. Does reinforcement learning really incentivize reasoning

capacity in llms beyond the base model?, 2025. URL . https://arxiv.org/abs/2504.13837

Advances in Neural InformationZelikman, E., Wu, Y., Mu, J., and Goodman, N. Star: Bootstrapping reasoning with reasoning.

Processing Systems, 35:15476–15488, 2022.

18

**Page 19**

**Absolute Zero: Reinforced Self-play Reasoning with Zero Data**

CoRRZeng, H., Jiang, D., Wang, H., Nie, P., Chen, X., and Chen, W. ACECODER: acing coder RL via automated test-case synthesis. ,

abs/2502.01718, 2025a. doi: 10.48550/ARXIV.2502.01718. URL . https://doi.org/10.48550/arXiv.2502.01718

Zeng, W., Huang, Y., Liu, Q., Liu, W., He, K., Ma, Z., and He, J. Simplerl-zoo: Investigating and taming zero reinforcement

CoRRlearning for open base models in the wild. , abs/2503.18892, 2025b. doi: 10.48550/ARXIV.2503.18892. URL https:

.//doi.org/10.48550/arXiv.2503.18892

Zhang, C., Deng, Y., Lin, X., Wang, B., Ng, D., Ye, H., Li, X., Xiao, Y., Mo, Z., Zhang, Q., et al. 100 days after deepseek-r1: A survey

arXiv preprint arXiv:2505.00551on replication studies and more directions for reasoning language models. , 2025a.

Zhang, Q., Wu, H., Zhang, C., Zhao, P., and Bian, Y. Right question is already half the answer: Fully unsupervised llm reasoning

incentivization, 2025b. URL . https://arxiv.org/abs/2504.05812

IEEE transactions on knowledge and data engineeringZhang, Y. and Yang, Q. A survey on multi-task learning. , 34(12):5586–5609,

2021.

Zhao, A., Lin, M. G., Li, Y., Liu, Y., and Huang, G. A mixture of surprises for unsupervised reinforcement learning. In

Advances in Neural Information Pro-Koyejo, S., Mohamed, S., Agarwal, A., Belgrave, D., Cho, K., and Oh, A. (eds.),

cessing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans,

LA, USA, November 28 - December 9, 2022, 2022. URL http://papers.nips.cc/paper\_files/paper/2022/hash/

.a7667ee5d545a43d2f0fda98863c260e-Abstract-Conference.html

Zhao, A., Huang, D., Xu, Q., Lin, M., Liu, Y., and Huang, G. Expel: LLM agents are experiential learners. In Wooldridge, M. J., Dy,

Thirty-Eighth AAAI Conference on Artificial Intelligence, AAAI 2024, Thirty-Sixth Conference onJ. G., and Natarajan, S. (eds.),

Innovative Applications of Artificial Intelligence, IAAI 2024, Fourteenth Symposium on Educational Advances in Artificial Intelligence,

EAAI 2014, February 20-27, 2024, Vancouver, Canada, pp. 19632–19642. AAAI Press, 2024. doi: 10.1609/AAAI.V38I17.29936.

URL . https://doi.org/10.1609/aaai.v38i17.29936

Zhao, A., Xu, Q., Lin, M., Wang, S., Liu, Y., Zheng, Z., and Huang, G. Diver-ct: Diversity-enhanced red teaming large language model

AAAI-25, Sponsored by the Association for theassistants with relaxing constraints. In Walsh, T., Shah, J., and Kolter, Z. (eds.),

Advancement of Artificial Intelligence, February 25 - March 4, 2025, Philadelphia, PA, USA, pp. 26021–26030. AAAI Press, 2025a.

doi: 10.1609/AAAI.V39I24.34797. URL . https://doi.org/10.1609/aaai.v39i24.34797

NeuralZhao, A., Zhu, E., Lu, R., Lin, M., Liu, Y., and Huang, G. Self-referencing agents for unsupervised reinforcement learning.

Networks, 188:107448, 2025b. doi: 10.1016/J.NEUNET.2025.107448. URL https://doi.org/10.1016/j.neunet.2025.

.107448

Zitkovich, B., Yu, T., Xu, S., Xu, P., Xiao, T., Xia, F., Wu, J., Wohlhart, P., Welker, S., Wahid, A., Vuong, Q., Vanhoucke, V., Tran,

H. T., Soricut, R., Singh, A., Singh, J., Sermanet, P., Sanketi, P. R., Salazar, G., Ryoo, M. S., Reymann, K., Rao, K., Pertsch, K.,

Mordatch, I., Michalewski, H., Lu, Y., Levine, S., Lee, L., Lee, T. E., Leal, I., Kuang, Y., Kalashnikov, D., Julian, R., Joshi, N. J.,

Irpan, A., Ichter, B., Hsu, J., Herzog, A., Hausman, K., Gopalakrishnan, K., Fu, C., Florence, P., Finn, C., Dubey, K. A., Driess,

D., Ding, T., Choromanski, K. M., Chen, X., Chebotar, Y., Carbajal, J., Brown, N., Brohan, A., Arenas, M. G., and Han, K. RT-2:

Conferencevision-language-action models transfer web knowledge to robotic control. In Tan, J., Toussaint, M., and Darvish, K. (eds.),

on Robot Learning, CoRL 2023, 6-9 November 2023, Atlanta, GA, USA Proceedings of Machine Learning Research, volume 229 of ,

pp. 2165–2183. PMLR, 2023. URL . https://proceedings.mlr.press/v229/zitkovich23a.html

Zuo, Y., Zhang, K., Qu, S., Sheng, L., Zhu, X., Qi, B., Sun, Y., Cui, G., Ding, N., and Zhou, B. Ttrl: Test-time reinforcement learning,

2025. URL . https://arxiv.org/abs/2504.16084

19

**Page 20**

**Absolute Zero: Reinforced Self-play Reasoning with Zero Data**

**Appendix**

**Appendix Contents**

**A Reinforcement Learning with Verifiable Rewards.** **21**

**B** **Implementation Details** **21**

**C More Results** **22**

C.1 Out-of-Distribution Performance Breakdown . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22

C.2 In-Distribution Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22

C.3 Interplay Between Propose and Solve Roles . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22

C.4 Complexity and Diversity Metrics of AZR Proposed Tasks . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32

C.5 Generated Code Complexity Dynamics Between Abd/Ded and Ind. . . . . . . . . . . . . . . . . . . . . . . . . . . 32

**D Alternative Approaches Considered** **49**

D.1 Error Deduction Task . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49

D.2 Composite Functions as Curriculum Learning . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49

()D.3 Toying with the Initial . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49 pz

D.4 Extra Rewards . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49

D.5 Environment Transition . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 50

20

**Page 21**

**Absolute Zero: Reinforced Self-play Reasoning with Zero Data**

**A. Reinforcement Learning with Verifiable Rewards.**

We use reinforcement learning to update our learner LLM, rewarding it based on a task-specific reward function , where the subscript rf indicates the task. The goal of the RL agent is to maximize the expected discounted sum of rewards. We adopt an online variant of RL,f

REINFORCE++, which is optimized using the original PPO objective:

� �

||o�1 � �norm normmin () ( () 1 1 + )() = clip (9) E −L s θA , s θ, ϵ, ϵ A ,θPPO ( ) ( ) t t∼ ∼ |q PQ, o π Oq f,q f,q||θ oold

=1t

norm()where is the probability ratio between the new and old policies at timestep , and is the normalized advantage. s θ t At f,q

REINFORCE++ computes the normalized advantage as:

( ) mean B − { } r Anorm= f,q f,q (10)A ,( )stdf,q B{ }Af,q

where is the outcome reward for question , task , mean and std are calculated across the global batch with batch size . Note that r q f Bf,qwe do not apply any KL penalty to the loss or reward.

**B. Implementation Details**

We built Absolute Zero Reasoner upon the veRL codebase (Sheng et al., 2025). For code execution, we incorporated components from

the QwQ Python executor. For safer code execution, we recommend using API-based services such as E2B instead.

All experiments were conducted on clusters of A800 GPUs.

**Training Hyperparameters.** We show the hyperparameters used in our training in Table 3. We do not change them for any of

the runs.

**Parameter** **Value**

**Model Configuration**

Max Prompt Length 6144

Max Response Length 8096

Seed Batch Factor 4

Max Programs 16384

**Training Settings**

Train Batch Size 64 \* 6

Learning Rate 1e-6

Optimizer AdamW

Grad Clip 1.0

Total Steps 500

**RL Settings**

Algorithm TRR++ (Section 3.3.5)

KL Loss False

KL Reward False

Entropy Coefficient 0.001

PPO Epochs 1

Rollouts 1N

Rollout Temperature 1.0

Rollout Top-P 1.0

References 6K

Samples to Estimate Task Accuracy 8N

**Hyperparameters Used During AZR Self-play Training.**Table 3.

21

**Page 22**

**Absolute Zero: Reinforced Self-play Reasoning with Zero Data**

**Model** **Data Curation** **Base Model**

(Liu et al., 2025) 8.5k math pairs (Hendrycks et al., 2021)Oat-7B Qwen2.5-7B-Math

(Zeng et al., 2025b) 8.5k math pairs (Hendrycks et al., 2021)SimpleRL-Zoo Qwen2.5-7B-Base

(Hu et al., 2025) 57k STEM + math samplesOpenReasonerZero Qwen2.5-7B-Base

(Cui et al., 2025) 457k math + 27k code problemsPRIME-Zero Qwen2.5Math-7B-Base

(Liu & Zhang, 2025) 2k Leetcode pairsCodeR1-Zero-7B-LC2k-1088 Qwen2.5-7B-Instruct-1M

(Liu & Zhang, 2025) 2k Leetcode + 10k TACO pairs (Li et al., 2023)CodeR1-Zero-7B-12k-832 Qwen2.5-7B-Instruct-1M

(Zeng et al., 2025a) 22k code dataAceCoder-7B-Ins-RM Qwen2.5-7B-Instruct

(Zeng et al., 2025a) 22k code dataAceCoder-7B-Ins-Rule Qwen2.5-7B-Instruct

(Zeng et al., 2025a) 22k code dataAceCoder-7B-Code-RM Qwen2.5-7B-Coder

(Zeng et al., 2025a) 22k code dataAceCoder-7B-Code-Rule Qwen2.5-7B-Coder

(Yang et al., 2024a) 1M SFT + 150k RL pairsQwen-7B-Instruct Qwen2.5-7B-Base

**No data**AZR-7B (Ours) Qwen2.5-7B-Base

**No data**AZR-7B-Coder (Ours) Qwen2.5-7B-Coder

**Reasoner Training Data Source and Base Model.**Table 4.

logging random multiprocessing pebble subprocess

threading datetime time hashlib calendar

bcrypt os.sys os.path sys.exit os.environ

**Forbidden Python Modules.**Figure 8. List of Python modules forbidden to exist in proposed tasks’ programs.

**C. More Results**

**C.1. Out-of-Distribution Performance Breakdown**

We plot the out-of-distribution performance, broken down by each benchmark and in aggregate, across training steps for our 7B, 7B-Coder,

14B, and 14B-Coder models in Figures 28 to 31. We observe a strong correlation between training using AZR and improvements in both

mathematical and coding reasoning capabilities. Moreover, our models are trained for more steps than typical zero-style reasoners; while

overfitting can occur with static datasets, it is less likely in AZR due to dynamically proposed tasks.

**C.2. In-Distribution Results**

Since we have defined the task domains as input prediction and output prediction, we can directly evaluate our model’s capabilities in these

areas using popular code reasoning benchmarks: CruxEval-I(nput), CruxEval-O(utput), and LiveCodeBench-Execution (LCB-E) (Gu

et al., 2024; Jain et al., 2024), where CruxEval-O and LCB-E is solving the deduction task, and CruxEval-I is solving the abduction task.

In Figure 14, we visualize the evolution of these metrics during the training of . As training Absolute Zero Reasoner-base-7b

progresses, we observe a consistent improvement in in-distribution performance across steps. While these three benchmark curves do

not perfectly correlate with broader coding or math reasoning capabilities (compare this with Figure 28), they serve as useful proxies for

tracking task-specific progress.

**C.3. Interplay Between Propose and Solve Roles**

We visualize the training dynamics between the propose and solve roles over training steps in Figures 15 to 17. We observe that, in

general, the solve roles produce more output tokens than the propose role. Intuitively, this makes sense: the propose role emphasizes

creativity and generation of novel tasks, whereas the solve role requires deeper reasoning, which naturally leads to longer outputs.

Interestingly, we also observe a consistent ordering in token length across reasoning types—abduction and deduction tasks tend to result

in shorter outputs than induction tasks during problem solving. This aligns with our intuition, as we observed the model engaging

in trial-and-error reasoning—repeatedly generating hypothesized inputs, evaluating their outcomes, and reflecting and retrying when

subsequent deductions fail to produce the correct output. To our knowledge, this is the first time such a clear distinction in token length

VALIDATE\_CODE\_TEMPLATE = """{code}1

repr(f({ inputs }))"""2

3

**exec**( VALIDATE\_CODE\_TEMPLATE )4

**Python Program to Check Valid Code.**Figure 9.

22

**Page 23**

**Absolute Zero: Reinforced Self-play Reasoning with Zero Data**

EVAL\_INPUT\_PREDICTION\_TEMPLATE = """{code}1

{gold\_output} == f({ agent\_input })"""2

3

**exec**( EVAL\_INPUT\_PREDICTION\_TEMPLATE )4

**Python Code to Check Agent Input Abduction Correctness.**Figure 10.

EVAL\_OUTPUT\_PREDICTION\_TEMPLATE = """{code}1

eval ({ gold\_output }) == eval ({ agent\_output })"""2

3

**exec**( EVAL\_OUTPUT\_PREDICTION\_TEMPLATE )4

**Python Code to Check Agent Output Deduction Correctness.**Figure 11.

EVAL\_FUNCTION\_PREDICTION\_TEMPLATE = """{code}1

matches = []2

for gold\_input , gold\_output in zip({ gold\_inputs}, {gold\_outputs }):3

match = {gold\_output} == f({ gold\_input })4

matches.append(match)5

"""6

7

**exec**( EVAL\_OUTPUT\_PREDICTION\_TEMPLATE )8

**Python Code to Check Agent Function Induction Correctness.**Figure 12.

CHECK\_DETERMINISM\_TEMPLATE = """{code}1

returns = f({ inputs })2

if returns != f({ inputs }):3

raise Exception(’Non -deterministic code ’)4

repr(returns)"""5

6

**exec**( CHECK\_DETERMINISM\_TEMPLATE )7

**Python Code to Check Deterministic Program.**Figure 13.

0.65

0.60

Performance Score

0.55

0.50

0.45

CruxEval-I0.40

CruxEval-O

LiveCodeBench-Execution0.35

0 30 60 90 120 150 180 210 240 270

Training Steps

**In-distribution Benchmark Score During Training.**Figure 14. The evolution of CruxEval-I, CruxEval-O, and LiveCodeBench-

Execution during training for the base model trained using AZR. Qwen2.5-7B

23

**Page 24**

**Absolute Zero: Reinforced Self-play Reasoning with Zero Data**

Solve Propose

0.8 4000

0.6 Token Length3000

Reward

0.4

2000

0.2

1000

0.0

0 40 80 120 160 200 240 280 0 40 80 120 160 200 240 280

Training StepsAbduction Task

**Abduction Task Reward and Token Lengths.**Figure 15. The task reward and token lengths of the two roles for abduction task type of

Absolute Zero Reasoner-base-7b.

Solve Propose

0.7

2000

0.6

1750

0.5

1500Token Length

0.4Reward

1250

0.3

1000

0.2 750

0.1 500

2500.0

0 40 80 120 160 200 240 280 0 40 80 120 160 200 240 280

Training StepsInduction Task

**Induction Task Reward and Token Lengths.**Figure 16. The task reward and token lengths of the two roles for induction task type of

Absolute Zero Reasoner-base-7b.

24

**Page 25**

**Absolute Zero: Reinforced Self-play Reasoning with Zero DataAbsolute Zero: Reinforced Self-play Reasoning with Zero Data**

Solve Propose

1.0 1400

12000.8

1000Token Length

0.6Reward

800

0.4

600

0.2

400

0.0

0 40 80 120 160 200 240 280 0 40 80 120 160 200 240 280

Training StepsDeduction Task

**Deduction Task Reward and Token Lengths.**Figure 17. The task reward and token lengths of the two roles for deduction task type of **Deduction Task Reward and Token Lengths.**Figure 17. The task reward and token lengths of the two roles for deduction task type of

Absolute Zero Reasoner-base-7b.Absolute Zero Reasoner-base-7b.

**Task: Propose Deduction Task**

**Model Input:** (k random selected example programs as few-shot input)

**Model Output:**

This task requires creating a new Python function that takes an input and returns a value. The function should have at least one

input parameter, be deterministic, and require state tracking across multiple data transformations. The input should be a test case

provided to a subject for the purpose of deducing the output. The code and the output should be challenging enough to require

algorithmic reasoning without using restricted words or packages.

**def** f(prices: list[int], budget: int):

n = len(prices)

profit = [0] \* n

**for**  **in** i range(1, n):

**for**  **in** j range(i):

**if** prices[i] > prices[j]:

profit[i] = max(profit[i], profit[j] + prices[i] - prices[j])

**return**  **if**  **else** max(profit + [profit[-1] - prices[-1] budget >= prices[-1] 0])

**Input:** [7, 1, 5, 3, 6, 4], 7

**Propose Deduction Task Example.**Figure 18. An example response of proposing a deduction task. k programs are selected as few-shotFigure 18. An example response of proposing a deduction task. k programs are selected as few-shot examples to the model. The generated

examples to the model. The generated new program will then be verified through execution.new program will then be verified through execution.

2524

**Page 26**

**Absolute Zero: Reinforced Self-play Reasoning with Zero Data**

**Absolute Zero: Reinforced Self-play Reasoning with Zero Data**

**def** f(numbers):

# Step 1: Filter out even numbers

**for**  **in**  **if**filtered\_numbers = [num num numbers num % 2 != 0]

# Step 2: Calculate the sum of the remaining odd numbers

sum\_of\_odd\_numbers = sum(filtered\_numbers)

# Step 3: Reverse the order of the remaining odd numbers

reversed\_odd\_numbers = filtered\_numbers[::-1]

# Step 4: Calculate the product of the reversed odd numbers

product\_of\_reversed\_odd\_numbers = 1

**for**  **in** num reversed\_odd\_numbers:

product\_of\_reversed\_odd\_numbers \*= num

# Step 5: Calculate the sum of the digits of the product

**for**  **in**sum\_of\_digits\_of\_product = sum(int(digit) digit str(product\_of\_reversed\_odd\_numbers))

# Step 6: Modify the original list by adding the sum of the digits to each even number

# and subtracting it from each odd number

modified\_numbers = []

**for**  **in** num numbers:

**if** num % 2 == 0:

modified\_numbers.append(num + sum\_of\_digits\_of\_product)

**else**:

modified\_numbers.append(num - sum\_of\_digits\_of\_product)

# Step 7: Return the modified list

**return** modified\_numbers

**Learned “ReAct” style program generation.**Figure 19. We observe that even after outputting the CoT and during answering when the **Learned “ReAct” style program generation.**Figure 19. We observe that even after outputting the CoT and during answering when the

answer requires outputting code, the model usually uses comments in code to interleave its thinking and “acting”, an example is shownanswer requires outputting code, the model usually uses comments in code to interleave its thinking and “acting”, an example is shown

here for induction task output by during training. Similar observations are recently observed in Absolute Zero Reasoner-Coder-14bhere for induction task output by during training. Similar observations are recently observed Absolute Zero Reasoner-Coder-14b

much bigger models (671b) for math proofs with DeepSeek Prover V2 (Ren et al., 2025).in much bigger models (671b) for math proofs with DeepSeek Prover V2 (Ren et al., 2025).

2526

**Page 27**

**Absolute Zero: Reinforced Self-play Reasoning with Zero DataAbsolute Zero: Reinforced Self-play Reasoning with Zero Data**

**Task: Propose Abduction Task**

**Model Input:** (k random selected example programs as few-shot input)

**Model Output:**

For this task, we need an intelligent problem that requires deep algorithmic reasoning and demands long multi step reasoning. A

typical type of problem that fits these requirements is the Matrix Reconstruction from a given jumbled li st of elements, where

we have to figure out the matrix’s dimensions and individual values. However, this task will involve multiple steps, increasing its

complexity and creativity. We will create a function ‘f‘ that transforms a jumbled list of elements into a two-dimensional matrix,

then back to a list, given a specific reconstruction pattern. We will retrieve the original list from the reconstructed matrix. Here,

we will avoid random and datetime operations to ensure determinism. The function will be structured to reconstruct the matrix

elements in a unique and challenging way. We will provide an input that, after reconstruction, presents a unique list that can be

easily deduced from the given output in a test piece.

**def** f(jumbled\_elements: list) -> list:

**if not** jumbled\_elements:

**return** []

# Calculate the square root of the length of elements to estimate the matrix dimensions

n = len(jumbled\_elements)

sqrt\_n = int(n \*\* 0.5)

**if** n > sqrt\_n \*\* 2:

sqrt\_n += 1

**if** n < sqrt\_n \*\* 2:

sqrt\_n -= 1

# Now, we need to reconstruct the matrix elements in a challenging way using specific patterns

# For this, we can reconstruct the matrix using zigzag patterns or hill-climbing patterns

# Here, a hill-climbing pattern is used where we traverse the matrix from the extreme

# right and downward Then traverse the matrix from the second to last element to the

# extreme bottom. Repeat the process until the entire matrix is traversed.

# Initialize the matrix

**for**  **in**  **for**  **in**matrix = [[0 \_ range(sqrt\_n)] \_ range(sqrt\_n)]

cur\_idx = 0

# Fill the matrix in a zigzag pattern

**for**  **in** i range(sqrt\_n):

**if** i % 2 == 0:

**for**  **in** j range(sqrt\_n):

matrix[j][i] = j \* sqrt\_n + i + cur\_idx

cur\_idx += 1

**else**:

**for**  **in** j range(sqrt\_n - 1, -1, -1):

matrix[j][i] = j \* sqrt\_n + n - i - cur\_idx - 1

cur\_idx += 1

# Reconstruct the list into a matrix by repeating the process

reconstructed\_list = []

**for**  **in** i range(sqrt\_n):

**for**  **in** j range(sqrt\_n):

reconstructed\_list.append(matrix[i][j])

# Now, we need to calculate another matrix based on a function F applied to the current matrix

# F is a mathematical function that can be complex, such as factorial, prime check,

# or another transformation For example, transform each element by adding 1 if it's prime,

# subtracting 1 if it's even, or leaving it unchanged, Implement any specific transformation

# function here. Here, for demonstration, each element is left unchanged

transformed\_result = reconstructed\_list

**return** transformed\_result

**Output:** [0, 11, 10, 1, 5, 16, 15, 6, 10, 21, 20, 11, 15, 26, 25, 16]

Figure 20. An example response of proposing an abduction task. The model uses comments as reasoning when generating the program.

**Propose Abduction Task Example.**Figure 20. An example response of proposing an abduction task. The model uses comments as

reasoning when generating the program. 26

27

**Page 28**

**Absolute Zero: Reinforced Self-play Reasoning with Zero Data**

**Absolute Zero: Reinforced Self-play Reasoning with Zero Data**

**Task: Propose Induction Task**

**Model Input:** Here is the function . Analyze its behaviour and relate it to the observed inputs/outputs. f

**def** f(numbers: list[int]) -> int:

replacement = {

0: 2,

1: 3,

2: 1,

3: 0,

4: 4

}

**return**  **for**  **in** sum(replacement[num] num numbers)

**Model Output:**

To solve this task, we need to come up with 10 different inputs for the function . The function f(numbers: list[int]) -> int

replaces each number in the list according to a specific dictionary and then sums up the replaced numbers. Let’s analyze the

dictionary:

This dictionary tells us that:

0 2- becomes

1 3- becomes

2 1- becomes

3 0- becomes

4 4- remains

0 1 2 3 4To create diverse outputs, we should include lists with different sets of numbers from the list . We also need { }, , , ,

to ensure that the inputs are valid (i.e., lists of integers).

Let’s illustrate using provided examples step-by-step:

[2] = 21. Input: Replacement: sum [0] ⇒ ⇒

[3 1 0] = 42. Input: Replacement: sum [1, 2, 3] ⇒ ⇒, ,

[4 3] = 73. Input: Replacement: sum [4, 1] ⇒ ⇒,

[1 3 2] = 64. Input: Replacement: sum [2, 1, 0] ⇒ ⇒, ,

[0 2 1 4] = 75. Input: Replacement: sum [3, 0, 2, 4] ⇒ ⇒, , ,

[3 3 3] = 96. Input: Replacement: sum [1, 1, 1] ⇒ ⇒, ,

[1 1 2 3] = 77. Input: Replacement: sum [2, 2, 0, 1] ⇒ ⇒, , ,

[0 4 1] = 58. Input: Replacement: sum [3, 4, 2] ⇒ ⇒, ,

[2 3 1 0 4] = 109. Input: Replacement: sum [0, 1, 2, 3, 4] ⇒ ⇒, , , ,

[3 1 0 2 4] = 1010. Input: Replacement: sum [1, 2, 3, 0, 4] ⇒ ⇒, , , ,

Thus, the rule is clearly replacing numbers according to the given mapping and then summing the results.

**Inputs**: ['[0]', '[1, 2, 3]', '[4, 1]', '[2, 1, 0]', '[3, 0, 2, 4]', '[1, 1, 1]',

, ’[3, 4, 2]’, ’[0, 1, 2, 3, 4]’, ’[1, 2, 3, 0, 4]’]|[2, 2, 0, 1]

**Message**: You are given a function that takes a list of integers as input and returns a single integer as output. Can you determine

the rule this function is following by analyzing the outputs for the given inputs?

**Outputs**(After execution:) : ['2', '4', '7', '6', '7', '9', '7', '5', '10', '10']

Figure 21. An example response of proposing an induction task. Given a random selected program, the model needs to propose 10 inputs **Propose Induction Task Example.**Figure 21. An example response of proposing an induction task. Given a random selected program,

and a description of program. These inputs will be executed and verified to get outputs. At solve phase, the model need to generate thethe model needs to propose 10 inputs and a description of program. These inputs will be executed and verified to get outputs. At solve

program given the message and 5 input/output pairs (half is hidden cases)phase, the model need to generate the program given the message and 5 input/output pairs (half is hidden cases)

2728

**Page 29**

**Absolute Zero: Reinforced Self-play Reasoning with Zero DataAbsolute Zero: Reinforced Self-play Reasoning with Zero Data**

**Task: Solve Abduction Task**

**Model Input:** Here is the function and the output: ‘1‘, please predict the input. f

**def** f(numbers: list[int], target: int) -> int:

accumulated\_values = []

current\_sum = 0

**for**  **in** number numbers:

current\_sum += number

accumulated\_values.append(current\_sum)

result = 0

**for**  **in** i range(len(accumulated\_values)):

**for**  **in** j range(i + 1, len(accumulated\_values)):

**if** accumulated\_values[j] \

- accumulated\_values[i] == target:

result = accumulated\_values[i]

**return** result

**return** result

**Model Output:**

Let’s analyze the code snippet to understand how the result is achieved. The function f takes a list of integers numbers and a target integer target. It calculates the accumulated sum of elements as it

iterates through the list. Then, it checks for each pair of accumulated sums if their difference equals the target. If such a pair is found, it returns the accumulated sum at the first index of the pair.

To find one possible input that produces the output 1, we need to find a scenario where the difference between any two accumulated\_values equals the target, and the first accumulated\_values that

satisfies this condition is 1.

Let’s consider a simple example:

-numbers = [1, 2]

-target = 1

Here’s the step-by-step calculation:

1. current\_sum = 0 + 1 = 1, accumulated\_values = [1]

2. current\_sum = 1 + 2 = 3, accumulated\_values = [1, 3]

-Check for pair (i=0, j=1): 3 - 1 = 2 (not equal to target)

-Check for pair (i=0, j=0): 1 - 1 = 0 (not equal to target)

-Check for pair (i=1, j=1): 3 - 3 = 0 (not equal to target)

Since no pair satisfies the condition, the function will not return any value. This doesn’t match our output. Let’s try another example.

Consider:

-numbers = [1, 0]

-target = 1

Step-by-step calculation:

1. current\_sum = 0 + 1 = 1, accumulated\_values = [1]

2. current\_sum = 1 + 0 = 1, accumulated\_values = [1, 1]

-Check for pair (i=0, j=1): 1 - 1 = 0 (not equal to target)

-Check for pair (i=0, j=0): 1 - 1 = 0 (not equal to target)

-Check for pair (i=1, j=1): 1 - 1 = 0 (not equal to target)

...(shortened)

The pair (i=1, j=2) satisfies the condition, and the function returns the first accumulated value in this pair, which is 3. This still doesn’t match our output.

Given the complexity and the fact that no simple example seems to work, let’s try a more complex scenario:

-numbers = [2, -1, 1]

-target = 1

Step-by-step calculation:

1. current\_sum = 0 + 2 = 2, accumulated\_values = [2]

2. current\_sum = 2 - 1 = 1, accumulated\_values = [2, 1]

3. current\_sum = 1 + 1 = 2, accumulated\_values = [2, 1, 2]

-Check for pair (i=0, j=1): 1 - 2 = -1 (not equal to target)

-Check for pair (i=0, j=2): 2 - 2 = 0 (not equal to target)

-Check for pair (i=1, j=2): 2 - 1 = 1 (equal to target)

The pair (i=1, j=2) satisfies the condition, and the function returns the first accumulated value in this pair, which is 1. This matches our output.

Therefore, one possible input that could produce the output 1 is: [2, -1, 1], 1

Figure 22. A more detailed version of the example shown in main Figure 7. The model is solving an abduction task (predict input). **Solve Abduction Task Example.**Figure 22. A more detailed version of the example shown in main Figure 7. The model is solving an

abduction task (predict input).

28

29

**Page 30**

**Absolute Zero: Reinforced Self-play Reasoning with Zero DataAbsolute Zero: Reinforced Self-play Reasoning with Zero Data**

**Task: Solve Deduction Task (Example 1)**

**Model Input:** Here is the function , the input provided , and we need to f [1, 2, 1, 2, 3, 2, 1], 2

predict the output.

**def** f(array: list[int], k: int) -> int:

window\_start = 0

max\_length = 0

char\_freq = {}

**for**  **in** window\_end range(len(array)):

end\_char = array[window\_end]

**if**  **not in** end\_char char\_freq:

char\_freq[end\_char] = 0

char\_freq[end\_char] += 1

**while** len(char\_freq) > k:

= array[window\_start]start\_char

char\_freq[start\_char] -= 1

**if** char\_freq[start\_char] == 0:

**del** char\_freq[start\_char]

window\_start += 1

max\_length = max(max\_length, window\_end - window\_start + 1)

**return** max\_length

**Input:** [1, 2, 1, 2, 3, 2, 1], 2

**Model Output:**

= 2Based on the given code snippet, we need to determine the output of the function when provided with the input and . f(array, k) [1, 2, 1, 2, 3, 2, 1] k

The function is designed to find the length of the longest subarray with at most distinct characters. Let’s go through the code step by step: f k

1. Initialize to 0, to 0, and an empty dictionary to store the frequency of characters in the current window. window\_start max\_length char\_freq

2. Iterate through the array using as the index: window\_end

a. Get the current character at index . end\_char window\_end

b. If is not in , add it with a frequency of 0. end\_char char\_freq

c. Increment the frequency of by 1. end\_char

3. Check if the number of distinct characters in is greater than : char\_freq k

a. If true, shrink the window from the start:

i. Get the character at index . start\_char window\_start

ii. Decrement its frequency by 1.

iii. If the frequency becomes 0, remove the character from . char\_freq

iv. Increment by 1. window\_start

4. Update by taking the maximum of and the current window size ( ). max\_length max\_length window\_end - window\_start + 1

= 2Now, let’s apply this algorithm step-by-step to the input , : [1, 2, 1, 2, 3, 2, 1] k

• , , window\_start = 0 max\_length = 0 char\_freq = {}

1 1 : 1 = 1• , add to : , window size window\_end = 0 char\_freq { } 2 1 : 1 2 : 1 = 2• , add : , window size window\_end = 1 { }, 1 1 : 2 2 : 1 = 3• , increment : , window size window\_end = 2 { }, 2 1 : 2 2 : 2 = 4• , increment : , window size window\_end = 3 { }, 3 1 : 2 2 : 2 3 : 1 2• , add : ; now distinct chars , shrink window: window\_end = 4 { }, , > 1 1 : 1 2 : 2 3 : 1- remove at start: { }, , 2 1 : 1 2 : 1 3 : 1- remove at start: { }, , 1 2 : 1 3 : 1 3- remove again: , window now starts at { }, 2 2 : 2 3 : 1 = 3• , increment : , window size window\_end = 5 { }, 1 2 : 2 3 : 1 1 : 1• , add : , shrink again: window\_end = 6 { }, , 2 2 : 1 3 : 1 1 : 1- remove : { }, , 3 2 : 1 1 : 1 5- remove : , window now starts at { }, = 2- window size

4Maximum window size encountered is , thus . max\_length = 4

Hence, the function returns. 4

Figure 23. Example of solving the deduction task. Given a program and input, the model simulates the execution of the program until **Solve Deduction Task Example.**Figure 23. Example of solving the deduction task. Given a program and input, the model simulates the

the final result.execution of the program until the final result.

2930

**Page 31**

**Absolute Zero: Reinforced Self-play Reasoning with Zero DataAbsolute Zero: Reinforced Self-play Reasoning with Zero Data**

**Task: Solve Induction Task**

**Model Input:** Given the inputs and outputs, deduce the code snippet implementing the described function-

ality:

**Input Description:** Given a list of integers, sort the list and then for each element at index , multiply it by i

( ), where is the total number of elements. Compute the final sum. −n i n

**Inputs:** ['[1, 2, 3, 4]', '[4, 3, 2, 1]', '[3, 1, 4, 2]', '[1, 2, 2, 3]', '[5]']

**Outputs:** ['20', '20', '20', '17', '5']

# Golden

**def** f(elements: list[int]) -> int:

n = len(elements)

sorted\_elements = sorted(elements)

**for**  **in**modified\_elements = [sorted\_elements[i] \* (n - i) i range(n)]

**return** sum(modified\_elements)

**Model Output:**

Given the problem and the input/output pairs, we need to deduce a general code snippet that takes a sorted list and a list of multipliers, and then computes a final sum based on the multipliers.

Let’s break down the problem step by step:

1. \*\*Understand the Message:\*\*

- We are given a list of integers.

- The list is sorted.

( )- For each element at index in the sorted list, we multiply it by , where is the number of elements in the list. − i n i n

- We need to compute the sum of these products.

2. \*\*Analyze the Input/Output Pairs:\*\*

[1 2 3 4]- For the input , the sorted form is the same, and the output is 20.,,,1 (4 1) = 3 2 (4 2) = 4 3 (4 3) = 3 4 (4 4) = 0 3 + 4 + 3 + 0 = 10, , , , Sum: · − · − · − · − [4 3 2 1] [1 2 3 4]- For the input , the sorted form is , and the output is 20 (same as above).,,, ,,, [3 1 4 2] [1 2 3 4]- For the input , the sorted form is , and the output is 20 (same as above).,,, ,,,**shortened**

3. \*\*Re-evaluate the Outputs:\*\*

- It seems there might be an error in the provided outputs for some inputs. However, based on the problem description, the outputs should be correct if we consider the sorted list and the multipliers

correctly.

4. \*\*Deduce the Function:\*\*

- Given the sorted list and the list of multipliers, we can deduce the following function:

“‘python

def f(a):

sorted\_a = sorted(a)

n = len(sorted\_a)

result = 0

for i in range(n):

result += sorted\_a[i] \* (n - i)

return result

**...(shortened)**

5. \*\*Verify the Function:\*\*

- Let’s verify the function with the provided inputs:

[1 2 3 4]- For :,,, [1 2 3 4]- Sorted list: ,,, [3 4 3 0]- Multipliers: ,,, 1 3 + 2 4 + 3 3 + 4 0 = 3 + 8 + 9 + 0 = 20- Sum: · · · ·**...(shortened)**

[1 2 3 4] [4 3 2 1] [3 1 4 2] [1 2 2 3] [5]The corrected outputs should be: For : 20, For : 20, For : 20, For : 13, For : 0,,, ,,, ,,, ,,,

Given the function:

**def** f(a):

sorted\_a = sorted(a)

n = len(sorted\_a)

result = 0

**for**  **in** i range(n):

result += sorted\_a[i] \* (n - i)

**return** result

The outputs match the corrected sum.

### Final Answer

**(repeat the program above)**

Figure 24. Example of solving an induction task. The model is given a program description, and 5 input/output cases and predict the **Solve Induction Task Example.**Figure 24. Example of solving an induction task. The model is given a program description, and 5

function. Interestingly, after the model gives the function, it will go through the given use cases one by one and confirm all test cases areinput/output cases and predict the function. Interestingly, after the model gives the function, it will go through the given use cases one by

passed.one and confirm all test cases are passed. 3031

**Page 32**

**Absolute Zero: Reinforced Self-play Reasoning with Zero Data**

+ + v1-5**Model** HEval MBPP LCB AIME’24 AIME’25 AMC’23 MATH500 Minerva OlympiadBench

Llama3.1-8B 31.7 53.7 0.0 0.0 0.0 2.5 10.6 5.5 2.1

+ Simple-RL-Zoo 38.4 55.3 7.4 0.0 0.0 7.5 22.2 8.8 4.7

+ AZR 35.4 50.8 8.5 3.3 0.0 5.0 13.2 14.0 5.0

Qwen2.5-3B-Coder 67.1 65.9 20.0 3.3 3.3 20.0 51.0 18.4 16.6

+ AZR 71.3 69.0 24.4 3.3 3.3 37.5 62.0 26.1 27.0

Qwen2.5-14B-Coder 76.8 71.7 31.4 0.0 0.0 37.5 54.8 10.7 18.5

+ AZR 80.5 71.2 39.0 23.3 20.0 65.0 78.6 32.0 39.3

78.0 66.7 21.7 6.7 3.3 35.0 66.2 28.3 32.4Qwen2.5-14B-Base

+ AZR 70.7 68.8 35.2 10.0 20.0 62.5 76.2 40.4 42.5

**Detailed Breakdown of Evaluation Benchmarks for Other Model Sizes and Types.**Table 5. Full evaluation of AZR trained on

+ + v1-5other models on three standard code benchmarks (HEval, MBPP, LCB ) and six math benchmarks (AIME’24, AIME’25, AMC’23,

MATH500, Minerva, OlympiadBench).

has been observed and presented for jointly trained reasoning multi-tasks. Previously, length differences were typically noted between

correct and incorrect traces (Liu et al., 2025).

The reward dynamics between the propose and solve roles exhibit mildly adversarial behavior: when one receives higher rewards, the

other often receives lower rewards. However, this is not entirely adversarial, as the proposer is also penalized for generating unsolvable

tasks, encouraging overall cooperative behavior in the learning process.

**C.4. Complexity and Diversity Metrics of AZR Proposed Tasks**

We outline several metrics used to probe characteristics of the tasks proposed during the training of AZR from the base model. Specifically,

we log two sets of metrics: program complexity and task diversity. For complexity, we employ two proxy measures—ComplexiPy score

and the Halstead metric. To assess diversity, we compute the average abstract syntax tree (AST) edit distance between the proposed

program and a set of reference programs, and an answer diversity metric. We calculate this answer diversity metric by tracking all K

historical answers to the generated questions, i.e., the input-output pairs, and form a categorical distribution over these outputs. We

1 ( ) ( )define answer diversity as answer, where answer is the empirical probability of a specific answer—used as a proxy for the − p p

diversity of generated outputs.

We present these metrics in Figure 27. Interestingly, even without incorporating them explicitly into the reward function, the policy

appears to implicitly optimize for these metrics. This aligns well with intuitive notions of task difficulty and diversity, and that the policy

learned to propose increasingly challenging tasks over time using our proposed simple reward function in Equation (4).

**C.5. Generated Code Complexity Dynamics Between Abd/Ded and Ind.**

We use the package to measure code complexity. For each generated program in the induction task, we compute the ComplexiPy

( ) ( ) i.ecognitive complexity difference from the corresponding “gold” code, . complexipy complexipy for each −p ppropose solveπ π{abduction,deduction} inductionpair, where the superscript of indicates the role and the subscript indicates the task type(s), and denotes the generated programs. On π p

027average, the difference of proposer and solver while holding the code’s functionality constant is , indicating that the proposer in the.

abduction/deduction tasks often increases the cognitive complexity to make the code appear more convoluted, whereas the induction

solver tends to generate more efficient implementations.

32

**Page 33**

**Absolute Zero: Reinforced Self-play Reasoning with Zero Data**

**Absolute Zero: Learning to Reason with Zero Data via Self-play**

**Task: Solve Deduction Task (Example 2)**

**Model Input:** Here is the function , the given input, and we need to predict the output. f

**def** f(num\_list: list[int], target\_product: int) -> list[tuple[int]]:

result\_pairs = []

**for**  **in** i range(len(num\_list)):

**for**  **in** j range(i + 1, len(num\_list)):

**if** num\_list[i] \* num\_list[j] == target\_product:

result\_pairs.append((num\_list[i], num\_list[j]))

**return** result\_pairs

**Input:** [2, 3, 5, 7, 10], 10

**Output:** [(2, 5)]

**Model Output:**

The given function takes a list of integers and a target product as inputs. It is designed to find all f num\_list target\_product

pairs of numbers in whose product equals . The function then returns a list of tuples, where each num\_list target\_product

tuple contains a pair of numbers that satisfy the condition.

Let’s go through the code step by step:

1. Initialize an empty list to store pairs whose product equals . result\_pairs target\_product

2. Use nested loops to iterate through each pair of numbers in . The outer loop iterates over each number, and the inner num\_list

**(2, 3)**  **(3, 2)**loop iterates over numbers that come after the current number in the outer loop ( )**避免重复和反向的对子，例如**  **和** 。

3. For each pair , check if their product equals . (num\_list[i], num\_list[j]) target\_product

4. If the product equals , append the tuple to . target\_product (num\_list[i], num\_list[j]) result\_pairs

5. After all pairs have been checked, return the list . result\_pairs

Let’s explicitly apply this to the provided input , target product : [2, 3, 5, 7, 10] 10

- For (num\_list[0] = 2): i = 0

- , 2 × 3 = 6 (≠10) j = 1

- , 2 × 5 = 10 (=10), add pair (2, 5) j = 2

- , 2 × 7 = 14 (≠10) j = 3

- , 2 × 10 = 20 (≠10) j = 4

- For (num\_list[1] = 3): i = 1

- , 3 × 5 = 15 (≠10) j = 2

- , 3 × 7 = 21 (≠10) j = 3

- , 3 × 10 = 30 (≠10) j = 4

- For (num\_list[2] = 5): i = 2

- , 5 × 7 = 35 (≠10) j = 3

- , 5 × 10 = 50 (≠10) j = 4

- For (num\_list[3] = 7): i = 3

- , 7 × 10 = 70 (≠10) j = 4

- For (num\_list[4] = 10): i = 4

- No further pairs to check

Thus, the only pair satisfying the condition is (2, 5). Therefore, the output is:

[(2, 5)]

Figure 25. An example of the model solving an deduction task. We observed that the model outputs mixed language (Chinese) when **Mixed Language CoT Example.**Figure 25. An example of the model solving an deduction task. We observed that the model outputs

solving the task.mixed language (Chinese) when solving the task.

3133

**Page 34**

**Absolute Zero: Reinforced Self-play Reasoning with Zero DataAbsolute Zero: Reinforced Self-play Reasoning with Zero Data**

**Task: Solve Deduction Task (Llama 8B)**

**Model Input:** Here is the function , the input provided , and we need to predict the output. f 'cookie'

**def** f(input\_string: str):

"""Function to check if a string contains given substrings."""

substring = "code"

# Start with `substring` as the first substring.

curr\_index = substring

# Slice out the required substring from the input string

sliced\_out\_string = input\_string[0:len(substring)]

**if** (sliced\_out\_string == substring):

**return True**

# Check if there are any remaining substrings to be checked

**if** (len(curr\_index) > len(input\_string)):

**return False**

**return False**

**Input:** 'cookie'

**Output:** False

**Model Output:**

I know that the line ‘curr\_index = substring‘ is just initialization of the curr\_index parameter variable and does not affect the output.

Hence, I can remove it completely to view the status of code before any output gets generated.

Now I have a single variable, “‘curr\_index“‘ which is set to ”codecandide”.

The second line of code, ‘slice\_out\_string = input\_string[0:len(substring)]‘, assigns the result of slicing the input\_string to the left

of index 0 and to the right of index len(substring), i.e., the index of the last character in the substring. It then assigns this value to

the variable “‘slice\_out\_string“‘.

The third line, ‘if (slice\_out\_string == substring)‘ checks whether the “‘slice\_out\_string“‘ is equal to the value of substring. Because

curr\_index is set to ”codecandide”, ”code” appears on the left of the first index in input\_string, and thus, “‘slice\_out\_string“‘ assumes

the value ”code”. That means for a minute the “‘if statement“‘ is satisfied and will continue the code execution afterwards.

However, according to the True/False statement, we see that the value of the first index of input\_string does not exist hence while it

does output ”True”, ”False” is outputted in main. So, the output of the whole program is shown below:

‘f(input\_string = ”cookie”)‘ then output: False

**Absolute Zero Reasoner-Llama3.1-8b CoT Example on Solving Output Deduction Task.**Figure 26. The **Absolute Zero Reasoner-Llama3.1-8b CoT Example on Solving Output Deduction Task.** Absolute ZeroFigure 26. The Absolute Zero

model appears to have strengthened its state-tracking behaviors during the course of training. We illus-Reasoner-Llama3.1-8b model appears to have strengthened its state-tracking behaviors during the course of training. We illustrate anReasoner-Llama3.1-8b

trate an example here.example here.

ComplexiPy Score Halstead Measure AST Edit Distance Answer Diversity

0.47 0.20 0.78 0.99

0 0 0 0

0 80 160 240 0 80 160 240 0 80 160 240 0 80 160 240

DiversityComplexity Training Steps

**Metrics on Proposed Tasks.**Figure 27. We break down the proposed task metrics into program complexity and diversity across programs **Metrics on Proposed Tasks.**Figure 27. We break down the proposed task metrics into program complexity and diversity across programs

and answers. An upward trend is observed in all metrics, indicating that AZR implicitly optimizes for these qualities as training progresses.and answers. An upward trend is observed in all metrics, indicating that AZR implicitly optimizes for these qualities as training progresses.

3534

**Page 35**

**Absolute Zero: Reinforced Self-play Reasoning with Zero Data**

MATH REASONING

AIME 2024 AIME 2025 Olympiad Bench

0.200 0.400

0.3750.1000.150Accuracy

0.350

0.100 0.050 0.325

0.3000.050 0.000

50 75 100 125 150 175 200 225 250 50 75 100 125 150 175 200 225 250 50 75 100 125 150 175 200 225 250

Minerva Math 500 AMC 2023

0.7500.380 0.600

0.7250.360 0.550Accuracy 0.7000.340 0.500

0.6750.320 0.450

0.6500.300 0.400

0.6250.280

50 75 100 125 150 175 200 225 250 50 75 100 125 150 175 200 225 250 50 75 100 125 150 175 200 225 250

CODE REASONING

HumanEval+ MBPP+ LiveCodeBench

0.7000.740

0.2800.6900.730Accuracy

0.720 0.680 0.260

0.710 0.670

0.2400.700 0.660

50 75 100 125 150 175 200 225 250 50 75 100 125 150 175 200 225 250 50 75 100 125 150 175 200 225 250

OVERALL PERFORMANCE

Math Average Code Average Overall Average

0.5700.400 0.480

0.380

0.560 0.460Accuracy0.360

0.340 0.550 0.440

0.320

0.5400.300 0.420

50 75 100 125 150 175 200 225 250 50 75 100 125 150 175 200 225 250 50 75 100 125 150 175 200 225 250

**Absolute Zero Reasoner-base-7b OOD Performance Breakdown.**Figure 28.

35

**Page 36**

**Absolute Zero: Reinforced Self-play Reasoning with Zero Data**

MATH REASONING

AIME 2024 AIME 2025 Olympiad Bench

0.200 0.100

0.3800.0800.150Accuracy 0.060

0.3600.0400.100

0.020

0.050 0.3400.000

50 100 150 200 250 300 350 50 100 150 200 250 300 350 50 100 150 200 250 300 350

Minerva Math 500 AMC 2023

0.600

0.375 0.760

0.5500.350 0.740Accuracy

0.5000.325 0.720

0.4500.300 0.700

0.275 0.4000.680

50 100 150 200 250 300 350 50 100 150 200 250 300 350 50 100 150 200 250 300 350

CODE REASONING

HumanEval+ MBPP+ LiveCodeBench

0.720 0.3200.850

0.7100.840Accuracy 0.300

0.830 0.700

0.820 0.2800.690

0.810 0.680 0.260

50 100 150 200 250 300 350 50 100 150 200 250 300 350 50 100 150 200 250 300 350

OVERALL PERFORMANCE

Math Average Code Average Overall Average

0.400 0.630

0.5000.6200.380

Accuracy 0.6100.360 0.480

0.6000.340

0.4600.590

0.320

0.580

50 100 150 200 250 300 350 50 100 150 200 250 300 350 50 100 150 200 250 300 350

**Absolute Zero Reasoner-Coder-7b OOD Performance Breakdown.**Figure 29.

36

**Page 37**

**Absolute Zero: Reinforced Self-play Reasoning with Zero Data**

MATH REASONING

AIME 2024 AIME 2025 Olympiad Bench

0.200

0.4000.200

0.150Accuracy

0.3500.150

0.100

0.3000.100

0.050

0.250

50 100 150 200 250 300 350 50 100 150 200 250 300 350 50 100 150 200 250 300 350

Minerva Math 500 AMC 2023

0.750 0.6000.400

0.700Accuracy 0.5500.6500.350

0.5000.600

0.5500.300 0.450

0.500

50 100 150 200 250 300 350 50 100 150 200 250 300 350 50 100 150 200 250 300 350

CODE REASONING

HumanEval+ MBPP+ LiveCodeBench

0.780

0.720

0.760 0.3600.710Accuracy

0.740 0.700 0.340

0.6900.720

0.3200.6800.700

50 100 150 200 250 300 350 50 100 150 200 250 300 350 50 100 150 200 250 300 350

OVERALL PERFORMANCE

Math Average Code Average Overall Average

0.620

0.5000.6100.400

Accuracy 0.600 0.480

0.350 0.590

0.460

0.580

0.300 0.4400.570

50 100 150 200 250 300 350 50 100 150 200 250 300 350 50 100 150 200 250 300 350

**Absolute Zero Reasoner-base-14b OOD Performance Breakdown.**Figure 30.

37

**Page 38**

**Absolute Zero: Reinforced Self-play Reasoning with Zero DataAbsolute Zero: Reinforced Self-play Reasoning with Zero Data**

MATH REASONING

AIME 2024 AIME 2025 Olympiad Bench

0.4400.200

0.200 0.4200.150Accuracy

0.150 0.4000.100

0.100 0.3800.050

50 100 150 200 250 300 350 400 50 100 150 200 250 300 350 400 50 100 150 200 250 300 350 400

Minerva Math 500 AMC 2023

0.7000.800

0.400 0.780 0.650

Accuracy0.350 0.760 0.600

0.300 0.740 0.5500.250 0.720 0.5000.200 0.700

50 100 150 200 250 300 350 400 50 100 150 200 250 300 350 400 50 100 150 200 250 300 350 400

CODE REASONING

HumanEval+ MBPP+ LiveCodeBench

0.4000.740

0.730 0.3800.820Accuracy

0.720 0.360

0.800 0.710

0.340

0.7000.780

50 100 150 200 250 300 350 400 50 100 150 200 250 300 350 400 50 100 150 200 250 300 350 400

OVERALL PERFORMANCE

Math Average Code Average Overall Average

0.440 0.5400.650

0.420 0.640Accuracy 0.5200.400 0.630

0.380

0.620 0.5000.360

0.610

0.340

50 100 150 200 250 300 350 400 50 100 150 200 250 300 350 400 50 100 150 200 250 300 350 400

**Absolute Zero Reasoner-Coder-14b OOD Performance Breakdown.**Figure 31. **Absolute Zero Reasoner-Coder-14b OOD Performance Breakdown.**Figure 31.

<think>

Design an absolutely ludicrous and convoluted Python function that is extremely difficult to

deduce the output from the input, designed to keep machine learning models such as Snippi→�

guessing and your peers puzzling.→�

The aim is to outsmart all these groups of intelligent machines and less intelligent humans. This

is for the brains behind the future.→�

</think>

- Absolute Zero Reasoner-Llama3.1-8b @ step 132

**Absolute Zero Reasoner – Llama3.1-8B “Uh-oh Moment.”**Figure 32. This example highlights an unexpected and potentially unsafe **Absolute Zero Reasoner** **Llama3.1-8B “Uh-oh Moment.”**Figure 32. This example highlights an unexpected and potentially unsafe **–**

reasoning chain generated by our model during training. Although our paradigm enables Absolute Zero Reasoner–Llama3.1-8Breasoning chain generated by our model during training. Although our paradigm enables Absolute Zero ReasonerLlama3.1-8B–

reasoning improvements without human-curated data, it may still require oversight due to the risk of emergent undesirable behaviors.reasoning improvements without human-curated data, it may still require oversight due to the risk of emergent undesirable behaviors.

3839

**Page 39**

**Absolute Zero: Reinforced Self-play Reasoning with Zero DataAbsolute Zero: Reinforced Self-play Reasoning with Zero Data**

A conversation between User and Assistant. The user asks a question, and the Assistant solves it.

The assistant first thinks about the reasoning process in the mind and then provides the user→�

with the answer. The reasoning process and answer are enclosed within <think> </think> and→�

<answer> </answer> tags, respectively, i.e., <think> reasoning process here </think> <answer>→�

answer here </answer>.→�

User: {TASK\_INSTRUCTION}

Assistant: <think>

**Deepseek R1 Template.**Figure 33. All our models were trained using the default Deepseek R1 template. **Deepseek R1 Template.**Figure 33. All our models were trained using the default Deepseek R1 template.

4039

**Page 40**

**Absolute Zero: Reinforced Self-play Reasoning with Zero DataAbsolute Zero: Reinforced Self-play Reasoning with Zero Data**

## Task: Create a Python Code Snippet (where custom classes are allowed, which should be defined

at the top of the code snippet) with one Matching Input→�

Using the reference code snippets provided below as examples, design a new and unique Python code

snippet that demands deep algorithmic reasoning to deduce one possible input from a given→� output. Your submission should include both a code snippet and test input pair, where the→� input will be plugged into the code snippet to produce the output, which that function output→� be given to a test subject to come up with any input that will produce the same function→� output. This is meant to be an I.Q. test.→�

### Code Requirements:

- Name the entry function `f` (e.g., `def f(...): ...`), you can have nested definitions inside

`f`→�- Ensure the function returns a value

- Include at least one input parameter

- Make the function deterministic

- Make the snippet require state tracking across multiple data transformations, ensuring the task

requires long multi step reasoning→�- AVOID THE FOLLOWING:

\* Random functions or variables

\* Date/time operations

\* I/O operations (reading files, network requests)

\* Printing or logging

\* Any external state

- Ensure execution completes within 10 seconds on a modern CPU

- All imports and class definitions should be at the very top of the code snippet

- The snippet should end with a return statement from the main function `f`, anything after will

be removed→�

### Input Requirements:

- Provide exactly one test input for your function

- Format multiple arguments with commas between them

- Remember to add quotes around string arguments

### Formatting:

- Format your code with: ```python

def f(...):

# your code here

return ...

```

- Format your input with: ```input

arg1, arg2, ...

```

### Example Format:

```python

def f(name: str, info: dict):

# code logic here

return result

```

```input

'John', {{'age': 20, 'city': 'New York'}}

```

### Evaluation Criteria:

- Executability, your code should be executable given your input

- Difficulty in predicting the output from your provided input and code snippet. Focus on either

algorithmic reasoning or logic complexity. For example, you can define complex data structure→� classes and operate on them like trees, heaps, stacks, queues, graphs, etc, or use complex→� control flow, dynamic programming, recursions, divide and conquer, greedy, backtracking, etc→�- Creativity, the code needs to be sufficiently different from the provided reference snippets

- Restricted usage of certain keywords and packages, you are not allowed to use the following

words in any form, even in comments: {LIST\_OF\_FORBIDDEN\_PACKAGES}→�

First, carefully devise a clear plan: e.g., identify how your snippet will be challenging,

distinct from reference snippets, and creative. Then, write the final code snippet and its→� inputs.→�

### Reference Code Snippets:

{CODE\_REFERENCES\_FROM\_BUFFER}

**Program Input Abduction Task** **Problem Proposal Instruction.**Figure 34. **— Program Input Abduction Task—Problem Proposal Instruction.**Figure 34.

41

40

**Page 41**

**Absolute Zero: Reinforced Self-play Reasoning with Zero DataAbsolute Zero: Reinforced Self-play Reasoning with Zero Data**

## Task: Create a New Python Code Snippet (where custom classes are allowed, which should be

defined at the top of the code snippet) with one Matching Input→�

Using the reference code snippets provided below as examples, design a new and unique Python code

snippet that demands deep algorithmic reasoning to deduce the output from the input. Your→� submission should include a code snippet and a test input pair, where the input will be→� plugged into the code snippet to produce the output. The input will be given to a test→� subject to deduce the output, which is meant to be an I.Q. test.→�

### Code Requirements:

- Name the entry function `f` (e.g., `def f(...): ...`), you can have nested definitions inside

`f`→�- Ensure the function returns a value

- Include at least one input parameter

- Make the function deterministic

- Make the snippet require state tracking across multiple data transformations, ensuring the task

requires long multi step reasoning→�- AVOID THE FOLLOWING:

\* Random functions or variables

\* Date/time operations

\* I/O operations (reading files, network requests)

\* Printing or logging

\* Any external state

- Ensure execution completes within 10 seconds on a modern CPU

- All imports and class definitions should be at the very top of the code snippet

- The snippet should end with a return statement from the main function `f`, anything after will

be removed→�

### Input Requirements:

- Provide exactly one test input for your function

- Format multiple arguments with commas between them

- Remember to add quotes around string arguments

### Formatting:

- Format your code with:

```python

def f(...):

# your code here

return ...

```

- Format your input with:

```input

arg1, arg2, ...

```

### Example Format:

```python

def f(name: str, info: dict):

# code logic here

return result

```

```input

'John', {{'age': 20, 'city': 'New York'}}

```

### Evaluation Criteria:

- Executability, your code should be executable given your input

- Difficulty in predicting your ```input``` from 1) your ```python``` code and 2) the

deterministic ```output``` that will be obtained from your ```input```. Focus on either→� algorithmic reasoning or logic complexity. For example, you can define complex data structure→� classes and operate on them like trees, heaps, stacks, queues, graphs, etc, or use complex→� control flow, dynamic programming, recursions, divide and conquer, greedy, backtracking, etc→�- Creativity, the code needs to be sufficiently different from the provided reference snippets

- Restricted usage of certain keywords and packages, you are not allowed to use the following

words in any form, even in comments: {LIST\_OF\_FORBIDDEN\_PACKAGES}→�

First, carefully devise a clear plan: e.g., identify how your snippet will be challenging,

distinct from reference snippets, and creative. Then, write the final code snippet and its→� inputs.→�

### Reference Code Snippets:

{CODE\_REFERENCES\_FROM\_BUFFER}

**Program Output Deduction Task** **Problem Generation Instruction.**Figure 35. **— Program Output Deduction Task—Problem Generation Instruction.**Figure 35.

4241

**Page 42**

**Absolute Zero: Reinforced Self-play Reasoning with Zero DataAbsolute Zero: Reinforced Self-play Reasoning with Zero Data**

## Task: Output {NUM\_INPUTS} Inputs that can be plugged into the following Code Snippet to

produce diverse Outputs, and give a message related to the given snippet.→�

Using the code snippet provided below, design {NUM\_INPUTS} inputs that can be plugged into the

code snippet to produce a diverse set of outputs. A subset of your given input and its→�

deterministically produced outputs will be given to a test subject to deduce the function,→�

which is meant to be an I.Q. test. You can also leave a message to the test subject to help→�

them deduce the code snippet.→�

### Input Requirements:

- Provide {NUM\_INPUTS} valid inputs for the code snippet

- For each input, format multiple arguments with commas between them

- Remember to add quotes around string arguments

- Each input should be individually wrapped in ```input``` tags

### Message Requirements:

- Leave a message to the test subject to help them deduce the code snippet

- The message should be wrapped in ```message``` tags

- The message can be in any form, can even be formed into a coding question, or a natural

language instruction what the code snippet does→�

- You cannot provide the code snippet in the message

### Formatting:

- Format your input with:

```input

arg1, arg2, ...

```

### Example Format:

```input

'John', {{'age': 20, 'city': 'New York'}}

```

```input

'Sammy', {{'age': 37, 'city': 'Los Angeles'}}

```

### Evaluation Criteria:

- Executability, your code should be executable given your inputs

- Coverage, the inputs and outputs should cover the whole input space of the code snippet, able

to deduce the code snippet from the inputs and outputs→�

- Creativity, the inputs need to be sufficiently different from each other

- The overall selection of inputs and message combined should be challenging for the test

subject, but not impossible for them to solve→�

First, carefully devise a clear plan: e.g., understand the code snippet, then identify how your

proposed inputs have high coverage, and why the inputs will be challenging and creative.→�

Then, write the inputs and message. Remember to wrap your inputs in ```input``` tags, and→�

your message in ```message``` tags.→�

### Code Snippet:

```python

{SNIPPET\_FROM\_BUFFER}

```

**Program Induction Task** **Problem Proposal Instruction.**Figure 36. **— Program Induction Task—Problem Proposal Instruction.**Figure 36.

4342

**Page 43**

**Absolute Zero: Reinforced Self-play Reasoning with Zero DataAbsolute Zero: Reinforced Self-play Reasoning with Zero DataAbsolute Zero: Reinforced Self-play Reasoning with Zero Data**

# Task: Provide One Possible Input of a Python Code Snippet Given the Code and Output# Task: Provide One Possible Input of a Python Code Snippet Given the Code and Output

Given the following Code Snippet and the Output, think step by step then provide one possibleGiven the following Code Snippet and the Output, think step by step then provide one possible

input that produced the output. The input needs to be wrapped in ```input``` tags. Rememberinput that produced the output. The input needs to be wrapped in ```input``` tags. Remember→�→� if an argument is a string, wrap it in quotes. If the function requires multiple arguments,if an argument is a string, wrap it in quotes. If the function requires multiple arguments,→�→� separate them with commas.separate them with commas.→�→�

# Code Snippet:# Code Snippet:

```python```python

{SNIPPET}{SNIPPET}

``````

# Output:# Output:

```output```output

{OUTPUT}{OUTPUT}

``````

# Output Format:# Output Format:

```input```input

arg1, arg2, ...arg1, arg2, ...

``````

# Example Output:# Example Output:

```input```input

'John', {{'age': 20, 'city': 'New York'}}'John', {{'age': 20, 'city': 'New York'}}

``````

**Program Input Abduction Task** **Problem Solving Prompt.**Figure 37. **— Program Input Abduction Task—Problem Solving Prompt. Program Input Abduction Task** **Problem Solving Prompt.**Figure 37.Figure 37. **—**

# Task: Deduce the Output of a Python Code Snippet Given the Code and Input# Task: Deduce the Output of a Python Code Snippet Given the Code and Input

Given the following Code Snippet and the Input, think step by step then deduce the output thatGiven the following Code Snippet and the Input, think step by step then deduce the output that

will be produced from plugging the Input into the Code Snippet. Put your output inwill be produced from plugging the Input into the Code Snippet. Put your output in→�→� ```output``` tags. Remember if the output is a string, wrap it in quotes. If the function```output``` tags. Remember if the output is a string, wrap it in quotes. If the function→�→� returns multiple values, remember to use a tuple to wrap them.returns multiple values, remember to use a tuple to wrap them.→�→�

# Code Snippet:# Code Snippet:

```python```python

{SNIPPET}{SNIPPET}

``````

# Input:# Input:

```input```input

{INPUT}{INPUT}

``````

# Example Output:# Example Output:

```output```output

{{'age': 20, 'city': 'New York'}}{{'age': 20, 'city': 'New York'}}

``````

**Program Output Deduction Task** **Problem Solving Prompt.**Figure 38. **— Program Output Deduction Task** **Problem Solving Prompt.**Figure 38. **Program Output Deduction Task—Problem Solving Prompt.—**Figure 38.

444443

**Page 44**

**Absolute Zero: Reinforced Self-play Reasoning with Zero Data**

**Absolute Zero: Reinforced Self-play Reasoning with Zero Data**

# Task: Deduce the Function that Produced the Outputs from the Inputs

Given a set of input/output pairs and a message that describes the function, think through the

problem step by step to deduce a general code snippet. This code should produce the hidden→�

outputs from the hidden inputs, matching the original data-generating code that created the→�

input/output pairs. Place your final answer inside python tags! It may be helpful to work→�

through each input/output pair individually to test your function. If your function doesn t’→�

work as expected, revise it until it does. The final code snippet will be used to evaluate→�

your response, which is wrapped in ```python``` tags.→�

# Code Requirements:

- Name the entry function `f` (e.g., `def f(...): ...`), you can have nested definitions inside

`f`→�

- Ensure the function returns a value

- Include at least one input parameter

- Make the function deterministic

- AVOID THE FOLLOWING:

\* Random functions or variables

\* Date/time operations

\* I/O operations (reading files, network requests)

\* Printing or logging

\* Any external state

- Ensure execution completes within 10 seconds on a modern CPU

- All imports and class definitions should be at the very top of the code snippet

- The snippet should end with a return statement from the main function `f()`, anything after

will be removed→�

# Input and Output Pairs:

{INPUT\_OUTPUT\_PAIRS}

# Message:

```message

{MESSAGE}

```

# Example Output:

```python

def f(a):

return a

```

Name your entry function `f()`!!!

**Program Induction Task** **Problem Solving Prompt.**Figure 39. **— Program Induction Task—Problem Solving Prompt.**Figure 39.

45

44

**Page 45**

**Absolute Zero: Reinforced Self-play Reasoning with Zero DataAbsolute Zero: Reinforced Self-play Reasoning with Zero Data**

**Task: Manual Constructed Sudoku Abduction Task**

**Model** **Input:** Here is the function and the output: [ [”5”,”3”,”.”,”.”,”7”,”.”,”.”,”.”,”.”],f

[”6”,”.”,”.”,”1”,”9”,”5”,”.”,”.”,”.”], [”.”,”9”,”8”,”.”,”.”,”.”,”.”,”6”,”.”], [”8”,”.”,”.”,”.”,”6”,”.”,”.”,”.”,”3”],

[”4”,”.”,”.”,”8”,”.”,”3”,”.”,”.”,”1”], [”7”,”.”,”.”,”.”,”2”,”.”,”.”,”.”,”6”], [”.”,”6”,”.”,”.”,”.”,”.”,”2”,”8”,”.”],

[”.”,”.”,”.”,”4”,”1”,”9”,”.”,”.”,”5”], [”.”,”.”,”.”,”.”,”8”,”.”,”.”,”7”,”9”] ], please predict the input.

**import random**

**from typing import** List

**import copy**

**def** g\_0(board: List[List[str]]) -> bool:

**def** is\_valid\_block(block):

**return**  **for**  **in** sorted(block) == [str(i) i range(1, 10)]

**for**  **in** row board:

**if not** is\_valid\_block(row):

**return False**

**for**  **in** col zip(\*board):

**if not** is\_valid\_block(col):

**return False**

**for**  **in** box\_row range(0, 9, 3):

**for**  **in** box\_col range(0, 9, 3):

block = [

board[r][c]

**for**  **in** r range(box\_row, box\_row + 3)

**for**  **in** c range(box\_col, box\_col + 3)

]

**if not** is\_valid\_block(block):

**return False**

**return True**

**def** f(board: List[List[str]]) -> List[List[str]]:

**for**  **in** i range(9):

**for**  **in** j range(9):

**if** board[i][j] == ".":

**raise ValueError**("Invalid board")

num\_to\_mask = 51

**if not** g\_0(board):

**raise ValueError**("Invalid board")

masked\_board = copy.deepcopy(board)

**for**  **in**  **for**  **in**all\_positions = [(i, j) i range(9) j range(9)]

random.shuffle(all\_positions)

**for**  **in** i, j all\_positions[:num\_to\_mask]:

masked\_board[i][j] = "."

**return** masked\_board

45

**Model Output:**

To determine the input that produced the given output, we need to analyze the function ‘f‘ and understand

what it does. Let’s break down the function step by step:

**Page 46**

**Model** **Input:** Here is the function and the output: [ [ 5 , 3 , . , . , 7 , . , . , . , . ],f

[”6”,”.”,”.”,”1”,”9”,”5”,”.”,”.”,”.”], [”.”,”9”,”8”,”.”,”.”,”.”,”.”,”6”,”.”], [”8”,”.”,”.”,”.”,”6”,”.”,”.”,”.”,”3”],

[”4”,”.”,”.”,”8”,”.”,”3”,”.”,”.”,”1”], [”7”,”.”,”.”,”.”,”2”,”.”,”.”,”.”,”6”], [”.”,”6”,”.”,”.”,”.”,”.”,”2”,”8”,”.”],

[”.”,”.”,”.”,”4”,”1”,”9”,”.”,”.”,”5”], [”.”,”.”,”.”,”.”,”8”,”.”,”.”,”7”,”9”] ], please predict the input.

**Absolute Zero: Reinforced Self-play Reasoning with Zero Data**

**Model Output:**

To determine the input that produced the given output, we need to analyze the function ‘f‘ and understand

what it does. Let’s break down the function step by step:

1. \*\*Input Validation in ‘g\_0‘\*\*: - The function ‘g\_0‘ checks if the given board is valid. - A board is valid

if each row, each column, and each 3x3 subgrid contains the digits 1 through 9 exactly once. - If the board

is not valid, ‘g\_0‘ returns ‘False‘.

2. \*\*Input Validation in ‘f‘\*\*: - The function ‘f‘ first checks if there are any ”.” in the board. If there are, it

raises a ‘ValueError‘. - This means the input board to ‘f‘ must be a complete board with no ”.”.

3. \*\*Masking the Board\*\*: - If the board is valid (i.e., ‘g\_0(board)‘ returns ‘True‘), the function ‘f‘ proceeds

to mask the board. - It creates a copy of the board and then randomly selects 51 positions to mask with ”.”.

- The masked board is then returned.

Given the output: “‘output [[”5”,”3”,”.”,”.”,”7”,”.”,”.”,”.”,”.”], [”6”,”.”,”.”,”1”,”9”,”5”,”.”,”.”,”.”],

[”.”,”9”,”8”,”.”,”.”,”.”,”.”,”6”,”.”], [”8”,”.”,”.”,”.”,”6”,”.”,”.”,”.”,”3”], [”4”,”.”,”.”,”8”,”.”,”3”,”.”,”.”,”1”],

[”7”,”.”,”.”,”.”,”2”,”.”,”.”,”.”,”6”], [”.”,”6”,”.”,”.”,”.”,”.”,”2”,”8”,”.”], [”.”,”.”,”.”,”4”,”1”,”9”,”.”,”.”,”5”],

[”.”,”.”,”.”,”.”,”8”,”.”,”.”,”7”,”9”]] “‘ We can see that the board has 51 positions masked with ”.”. Therefore,

the input board must have been a complete board, and ‘f‘ masked 51 random positions.

Let’s construct a complete board that, when masked, can produce the given output. We need to find a board

such that after masking 51 positions, we get the output shown.

Here is a complete board that matches the structure of the output: “‘python [ [”5”, ”3”, ”4”, ”6”, ”7”, ”8”,

”9”, ”1”, ”2”], [”6”, ”7”, ”2”, ”1”, ”9”, ”5”, ”3”, ”4”, ”8”], [”1”, ”9”, ”8”, ”3”, ”4”, ”2”, ”5”, ”6”, ”7”], [”8”,

”5”, ”9”, ”7”, ”6”, ”1”, ”4”, ”2”, ”3”], [”4”, ”2”, ”6”, ”8”, ”5”, ”3”, ”7”, ”9”, ”1”], [”7”, ”1”, ”3”, ”9”, ”2”,

”4”, ”8”, ”5”, ”6”], [”9”, ”6”, ”1”, ”5”, ”3”, ”7”, ”2”, ”8”, ”4”], [”2”, ”8”, ”7”, ”4”, ”1”, ”9”, ”6”, ”3”, ”5”],

[”3”, ”4”, ”5”, ”2”, ”8”, ”6”, ”1”, ”7”, ”9”] ] “‘

Let’s verify that this board, when passed to the function ‘f‘, produces the given output.

1. The board is complete with no ”.”. 2. The board is valid because each row, each column, and each 3x3

subgrid contains the digits 1 through 9 exactly once. 3. The function ‘f‘ will mask 51 random positions.

Given the random nature of the masking, we need to ensure that the positions masked match the positions

of ”.” in the output. Since the positions are random, we can assume that the function ‘f‘ with this complete

board will produce the given output.

Therefore, the input that produced the output is: “‘input [ [”5”, ”3”, ”4”, ”6”, ”7”, ”8”, ”9”, ”1”, ”2”], [”6”,

”7”, ”2”, ”1”, ”9”, ”5”, ”3”, ”4”, ”8”], [”1”, ”9”, ”8”, ”3”, ”4”, ”2”, ”5”, ”6”, ”7”], [”8”, ”5”, ”9”, ”7”, ”6”,

”1”, ”4”, ”2”, ”3”], [”4”, ”2”, ”6”, ”8”, ”5”, ”3”, ”7”, ”9”, ”1”], [”7”, ”1”, ”3”, ”9”, ”2”, ”4”, ”8”, ”5”, ”6”],

[”9”, ”6”, ”1”, ”5”, ”3”, ”7”, ”2”, ”8”, ”4”], [”2”, ”8”, ”7”, ”4”, ”1”, ”9”, ”6”, ”3”, ”5”], [”3”, ”4”, ”5”, ”2”,

”8”, ”6”, ”1”, ”7”, ”9”] ] “‘

**Vibe Check 1: Sudoku Solver.**Figure 40. We cast Sudoku solving as an abduction task: our program starts from a fully solved and **Vibe Check 1: Sudoku.**Figure 40.

validated Sudoku board and simulates the masking of 51 random cells. The masked board is then presented as output, and the model

is tasked with inferring the original input — effectively solving the puzzle backwards. An example solution is shown above, where

verifies its initial guess before correctly answering. Generation parameters: temperature=0.6.AZR-Coder-14b

46

46

**Page 47**

**Absolute Zero: Reinforced Self-play Reasoning with Zero DataAbsolute Zero: Reinforced Self-play Reasoning with Zero Data**

**Task: Manually Constructed Sum Product Game Solve Abduction Task**

**Model Input:** Here is the function and the output: ‘True‘, please predict the input. f

**def** g\_3():

**return**  **for**  **in**  **for**  **in**  **if** [(x, y) x range(2, 101) y range(x + 1, 101) x + y <= 100]

**def** g\_2(pairs):

d = {}

**for**  **in** (x, y) pairs:

s = x + y

d.setdefault(s, []).append((x, y))

**return** d

**def** g\_1(pairs):

d = {}

**for**  **in** (x, y) pairs:

p = x \* y

d.setdefault(p, []).append((x, y))

**return** d

**def** g\_0():

pairs = g\_3()

sum\_dict = g\_2(pairs)

prod\_dict = g\_1(pairs)

**for**  **in**allowed\_sums = [s s, candidate\_pairs sum\_dict.items() \

**if**  **for**  **in** all(len(prod\_dict[x \* y]) > 1 (x, y) candidate\_pairs)]

**for**  **in**  **if**  **in**candidates\_after\_S = [pair pair pairs pair[0] + pair[1] allowed\_sums]

new\_prod\_dict = g\_1(candidates\_after\_S)

**for**  **in**unique\_products = {prod: pair\_list prod, pair\_list new\_prod\_dict.items() \

**if** len(pair\_list) == 1}

final\_candidates = []

**for**  **in** s allowed\_sums:

**for**  **in**valid\_pairs = [pair pair candidates\_after\_S \

**if**  **and**  **in** pair[0] + pair[1] == s (pair[0] \* pair[1]) unique\_products]

**if** len(valid\_pairs) == 1:

final\_candidates.append(valid\_pairs[0])

**return** sorted(set(final\_candidates))

**def** f(x, y):

**if not**  **and** (isinstance(x, int) isinstance(y, int)):

**return False**

**if**  **or**  **or**  **or** x <= 1 y <= 1 y <= x (x + y) > 100:

**return False**

final\_candidates = g\_0()

**return**  **and** len(final\_candidates) == 1 final\_candidates[0] == (x, y)

**Model Output:** 47

Let’s analyze the code snippet to understand how the result is achieved. The function f takes a list of integers numbers and a target integer target. It calculates the accumulated sum of elements as it

iterates through the list. Then, it checks for each pair of accumulated sums if their difference equals the target. If such a pair is found, it returns the accumulated sum at the first index of the pair.

To find one possible input that produces the output 1, we need to find a scenario where the difference between any two accumulated\_values equals the target, and the first accumulated\_values that

satisfies this condition is 1

**Page 48**

**Task: Manually Constructed Sum Product Game Solve Abduction Task**

**Model Input:** Here is the function and the output: ‘True‘, please predict the input. f

**Absolute Zero: Reinforced Self-play Reasoning with Zero Data**

**Model Output:**

the function ‘f(x, y)‘ returns ‘True‘ if the following conditions are met:

1. ‘x‘ and ‘y‘ are both integers.

2. ‘x‘ and ‘y‘ satisfy ‘1 < x < y‘ and ‘x + y <= 100‘.

3. The final candidates after applying the conditions in ‘g\_0‘ are exactly one pair, and that pair is ‘(x, y)‘.

Let’s analyze the function ‘g\_0‘ step by step to understand what it does:

1. ‘pairs = g\_3()‘ generates all pairs ‘(x, y)‘ such that ‘2 <= x < y <= 100‘ and ‘x + y <= 100‘.

2. ‘sum\_dict = g\_2(pairs)‘ creates a dictionary where the keys are the sums of the pairs and the values are lists of pairs that add up

to that sum.

3. ‘prod\_dict = g\_1(pairs)‘ creates a dictionary where the keys are the products of the pairs and the values are lists of pairs that

have that product.

4. ‘allowed\_sums = [s for s, candidate\_pairs in sum\_dict.items() if all(len(prod\_dict[x \* y]) > 1 for (x, y) in candidate\_pairs)]‘

filters out the sums ‘s‘ for which all pairs ‘(x, y)‘ that add up to ‘s‘ have more than one pair with the same product.

5. ‘candidates\_after\_S = [pair for pair in pairs if pair[0] + pair[1] in allowed\_sums]‘ filters out the pairs whose sum is in

‘allowed\_sums‘.

6. ‘new\_prod\_dict = g\_1(candidates\_after\_S)‘ creates a new product dictionary using the filtered pairs.

7. ‘unique\_products = prod: pair\_list for prod, pair\_list in new\_prod\_dict.items() if len(pair\_list) == 1‘ creates a dictionary of

products that have exactly one pair.

8. ‘final\_candidates = []‘ initializes a list of final candidates.

9. The loop ‘for s in allowed\_sums:‘ iterates over each sum in ‘allowed\_sums‘ and checks if there is exactly one pair ‘(x, y)‘ such

that ‘x + y = s‘ and ‘x \* y‘ is in ‘unique\_products‘. If so, this pair is added to ‘final\_candidates‘.

10. The function returns ‘sorted(set(final\_candidates))‘, which should be a list containing exactly one pair if the conditions are

met.

Given that the output is ‘True‘, it means that there is exactly one pair ‘(x, y)‘ that satisfies all the conditions. Let’s identify this pair.

To do this, we need to understand the mathematical conditions and constraints:

1. ‘1 < x < y‘ and ‘x + y <= 100‘.

2. The pair ‘(x, y)‘ must be the only pair whose sum is in ‘allowed\_sums‘ and whose product is unique among the pairs with the

same sum.

By analyzing the constraints and the nature of the problem, we can deduce that the pair ‘(x, y)‘ must be such that the sum ‘x + y‘ is

unique in the sense that all pairs adding up to this sum have products that are not unique, except for this pair.

After detailed analysis or by running the code with different values, we find that the pair ‘(x, y) = (4, 13)‘ satisfies all the conditions.

Let’s verify:

1. ‘1 < 4 < 13‘ and ‘4 + 13 = 17 <= 100‘.

2. We need to check the ‘allowed\_sums‘ and the uniqueness of the product ‘4 \* 13 = 52‘.

By running the code or logical deduction, we can confirm that ‘(4, 13)‘ is the only pair that satisfies all the conditions.

Therefore, the input that produced the output ‘True‘ is:

“‘input

(4, 13)

“‘

**Vibe Check 2: Sum Product Game.**Figure 41. We cast the Sum-Product Game as an abduction task, where when the constraints of the **Solve Abduction Task Example.**Figure 41. A more detailed version of the example shown in main Figure 7. The model is solving an

game are satisfied, function ‘f‘ returns True, return False otherwise. was able to first analyze the complicated constraints, AZR-Coder-14babduction task (predict input). 48

identify candidate solution and verify. Generation parameters: temperature=0.6, top\_p=0.95.

48

**Page 49**

**Absolute Zero: Reinforced Self-play Reasoning with Zero Data**

**D. Alternative Approaches Considered**

In this section, we share many of the approaches we tried that did not prove to be particularly helpful for Absolute Zero Reasoner.

However, we believe it is especially valuable to share these findings with the community, as they are crucial for guiding future research.

Below, we outline each of the additional methods we explored during the development of our project.

**D.1. Error Deduction Task**

Since programming languages often have error messages, and these messages contain a lot of information about how someone might

that will produce anexpect a program to run, we also came up with another task domain: allowing the learner to propose a program

error deduce what kind of error is raised, and requiring the solver to when executing this code. We experimented with this additional

task alongside the induction ( ), deduction (), and abduction () tasks. Unfortunately, we did not observe noticeable changes inf o i

downstream performance with this additional task and since it requires more computational resources than our AZR setup, we decided

not to incorporate it into our final version. However, we believe further thorough investigation of this is well deserved.

**D.2. Composite Functions as Curriculum Learning**

One valuable property we can leverage from programming languages is the ability to compose functions—that is, to define a function as

(( ))a composite of other functions, i.e., . In our setting, when generating a program, we can not only require the output to be a valid fgx

program but also constrain the LLM to utilize a predefined set of programs within its main function. For example, if the target program

() 0to be generated is , we can sample a set of previously generated programs \_ from , and force a valid program to be· { } D f g , . . . , gc( 0 )\_ . · · ·fg , , g, ic

Since all programs are generated by the LLM itself, this setup allows the model to bootstrap from its earlier generations, automatically

increasing the complexity of the generated programs. We interpret this mechanism as a form of curriculum learning: earlier programs

in the AZR self-play loop tend to be simpler, and as the loop progresses, they become increasingly complex. By composing newer

programs from progressively more difficult earlier ones, the resulting programs naturally inherit this growing difficulty, which in turn

challenges the solver step.

For implementation, in generating tasks for abduction and deduction, we begin by sampling a binary decision from a binomial distribution

= 05with . This determines whether the generated program should be a simple program or a composite one. If the sample is 0, we p .

prompt the LLM to generate a standard program along with a corresponding input. If the sample is 1, we prompt the LLM to generate a

(1 3)composite program. To construct the composite, we first sample an integer , then uniformly select programs from the ∼ U c , c

dataset that are not themselves composite programs. Finally, we prompt the LLM to generate a valid program that incorporates D

0\_ as subcomponents, ensuring it composes these selected programs meaningfully. We additionally filter programs that did{ }g , . . . , gc

not utilize all the programs. c

However, we did not observe a significant difference when using this more complex curriculum compared to our simpler and more

effective approach. One failure mode we encountered was that the model often defaulted to simply returning “g(x)”, effectively learning

(( )) = ( ), which failed to introduce any additional difficulty. This trivial behavior undermined the intended challenge, leading usfgx gx

to deprioritize further exploration in this direction. While it may be possible to design a stricter reward mechanism—such as enforcing

(( ))= ( ) by executing the code via a Python interpreter and penalizing such shortcuts—we leave this to future work. ̸fgx gx

( )**D.3. Toying with the Initial** pz

() i.eWe investigated a setting where the initial seed buffer (see Section 3.3.1 on how we generated these), . in Equation (3), is not pz

self-generated by the base model, but instead sourced from the LeetCode Dataset. We only modified this component and ran AZR

using the same procedure as before, continuing to add new valid programs to the initialized buffer. We observed an increase in initial

performance on coding benchmarks; however, the performance plateaued at roughly the same level after additional training steps,

compared to our official AZR setup. Interestingly, math performance was lower than in the official AZR setup, pointing towards that

on-policy data may be more beneficial to the learner to bootstrap from for mathematical reasoning. We believe that exploring different

()strategies for initializing and updating is an important and exciting direction for future research. We briefly explored different pz

strategies for sampling reference code, ultimately settling on uniform sampling for its simplicity, though we also experimented with

recency-based sampling and observed potential collapse.

**D.4. Extra Rewards**

**Complexity Rewards.** Code complexity is well studied in software science and could potentially be a good proxy for measuring

how hard it is to infer the properties of a piece of code for our reasoning learner. Therefore, for the problem proposer, we can add various

measures of complexity—such as Cyclomatic Complexity (Ebert et al., 2016), maintainability, etc.—to the reward function to incentivize

the proposer to produce more complex programs. For illustration purposes, we tried using the Maintainability measure and the Halstead

49

**Page 50**

**Absolute Zero: Reinforced Self-play Reasoning with Zero Data**

complexity measure (Halstead, 1977) as intrinsic rewards. Concretely, we used the and packages (Lopez, 2025; complexipy Radon

Canal, 2023) to implement the respective metrics. These are then served as intrinsic rewards during the AZR self-play phase.

**Diversity Rewards.** We also attempted using diversity rewards to . Inspired by DiveR-CT (Zhao et al., 2025a), we incorporate

code edit distance as an intrinsic reward. Specifically, we treat the reference programs shown in the prompt as anchors and compute the

average code edit distance between the generated program and these anchors. This serves as a measure of diversity in the generated

surpriseoutput. Additionally, we explored another diversity-based reward inspired by the notion of (Zhao et al., 2022). In this approach,

we construct a probability distribution over previously encountered input/output pairs that the solver has answered. The reward is then

1 ( )defined as input/output, where denotes the empirical probability of a particular input or output. While both strategies were − p p

evaluated in our experiments, we did not observe a significant difference in performance. However, we believe this aspect warrants

deeper investigation, as diversity rewards remain a promising avenue for strengthening AZR further.

**Reward Aggregation.** We tested several ways on how to combine rewards for the proposer and discriminator. First, we separate

=the reward into extrinsic reward and a set of intrinsic reward(s) , and tested the following strategies to combine them { } r I rextrinsic i

into a single reward,

||I�

= + (11)r r r,extrinsic i

i

||I�

= (12) ·r r r,extrinsic i

i

||I�

= (13) ·r r r,extrinsic i

i

||I�

= + (14)r r r.extrinsic i

i

We found that the simple additive way of combining rewards, a.k.a Equation (11), produced the most stable runs, possibly due to less

variance.

**D.5. Environment Transition**

We investigated how the transition function in our coding environment for the proposer. Specifically, after generating a piece of code, we

can apply a transformation function on it before giving it making it an valid tuple in our dataset. We investigated two

**Removing Comments and Docstrings** In early iterations of our experiments, we noticed that comments and docstrings

were sometimes used to explicitly outline what the function was doing, or even served as a partial “note-taking” interleaved “ReAct”

process (Yao et al., 2023) of generating code—that is, the model could interleave think and action at the same time, and to make the

generated code valid, it used comments to encase its thoughts (Appendix C.3), similarly observed in DeepSeek-Prover-V2: (Ren et al.,

2025). We then thought that to make the task harder for the solver, we should occlude this information from it. However, we observed

a significant performance drop after removing all comments and docstrings. One explanation for this phenomenon is that the only

“communication” channel between the proposer and the solver is restricted to the code itself, rather than some kind of “message” along

with the code. These messages can potentially provide hints to the solver, thus making some otherwise impossible tasks solvable. As a

result, the solver is able to learn from its experience and self-bootstrap out of certain unsolvable tasks.

**Removing Global Variables.** We observed that some programs contain globally declared variables that may inadvertently leak

information about the correct answer—this issue is particularly prevalent in the input induction task generation and solving. Initially, we

were concerned that such leakage might lead to wasted computation on trivial or compromised examples. To address this, we developed

a systematic procedure to remove globally declared variables from the generated programs.

However, after applying this cleaning step, we observed a noticeable drop in performance on our self-play reasoning tasks. One possible

explanation is that the generation step is unaware of this post-processing modification; since the reward is assigned after the transition

function (which includes variable removal), the model may not learn effectively from this mismatch.

Moreover, we believe that even when answers are present, the solver still engages in nontrivial reasoning to reach a solution, potentially

benefiting from this exposure. This aligns with the idea of rationalization as proposed in STaR (Zelikman et al., 2022), where the model

pretends to not see the answer but still performs reasoning during learning. Therefore, in our final experiments, we choose not to remove

globally declared variables, allowing the self-play loop to naturally incorporate and adapt to such cases.

50