

O1-CODER: AN O1 REPLICATION FOR CODING

Yuxiang Zhang, Shangxi Wu, Yuqi Yang, Jiangming Shu, Jinlin Xiao, Chao Kong & Jitao Sang *

School of Computer Science and Technology

Beijing Jiaotong University

Beijing, China

{yuxiangzhang, wushangxi, yqyang, jiangmingshu, jinlinx, 23120361, jtsang}@bjtu.edu.cn

ABSTRACT

The technical report introduces O1-CODER, an attempt to replicate OpenAI’s o1 model with a focus on coding tasks. It integrates reinforcement learning (RL) and Monte Carlo Tree Search (MCTS) to enhance the model’s System-2 thinking capabilities. The framework includes training a Test Case Generator (TCG) for standardized code testing, using MCTS to generate code data with reasoning processes, and iteratively fine-tuning the policy model to initially produce pseudocode, followed by the generation of the full code. The report also addresses the opportunities and challenges in deploying o1-like models in real-world applications, suggesting transitioning to the System-2 paradigm and highlighting the imperative for environment state updates. Updated model progress and experimental results will be reported in subsequent versions. All source code, curated datasets, as well as the derived models will be disclosed at <https://github.com/ADaM-BJTU/O1-CODER>.

1 INTRODUCTION

OpenAI recently introduced the o1 model (OpenAI, 2024), which has demonstrated impressive system-2 thinking capabilities. This model represents a significant advancement in AI’s ability to perform complex reasoning tasks that require higher-order cognitive functions. Following its release, numerous analysis and replication efforts have emerged, highlighting the growing interest and potential of o1-like models. Notable works include g1 (Benjamin Klieger, 2024), OpenO1 (ope, 2024), O1-Journey (GAIR-NLP, 2024), OpenR (Team, 2024), LLaMA-O1 (SimpleBerry, 2024), LLaMA-Berry (Zhang et al., 2024), Steiner (Ji, 2024), Thinking Claude (Richards Tu, 2024), LLaVA-o1 (Xu et al., 2024), and several industrial releases such as k0-math, DeepSeek-R1-Lite, Macro-o1 (Zhao et al., 2024), Skywork o1, QwQ (Qwen Team, 2024), and InternThinker (Shanghai AI Lab, 2024) (illustrated in Fig. 1).

Prior to the o1 model, large language models (LLMs) primarily exhibited System-1 capabilities, characterized by fast, intuitive responses. These models were trained on datasets consisting mainly of question-answer (Q, A) pairs, lacking the intermediate reasoning steps that involve deliberate and analytical processing. This stems from the fact that humans rarely record their thought processes on the internet or elsewhere. Traditionally, techniques such as Chain-of-Thought (CoT) prompting were used to guide models in generating step-by-step reasoning before arriving at an answer. However, a more direct and effective way is to create datasets including the reasoning sequences, e.g., $(Q, \dots, S_i, \dots, A)$, where S_i represents an individual reasoning step leading to the final answer.

It is widely believed that o1 addresses the lack of reasoning data by combining reinforcement learning with pretraining. Reinforcement learning (RL) is well known for its ability to explore and discover new strategies rather than relying on predefined data. Looking back at key developments in machine learning, we can see that deep learning and large-scale pretraining have driven transformations in model architecture and the requirements for labeled data, respectively. In contrast, reinforcement learning addresses a different aspect of transformation on the objective function. In

*Corresponding author.

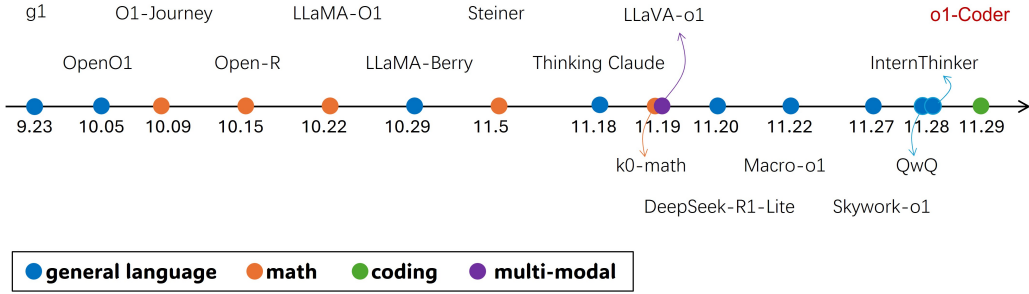


Figure 1: o1 replication efforts: upper part from academic institutions and open-source communities, and lower part from the industry.

situations where explicit guidance or clear goals are absent, RL exploits exploration to search for new knowledge and solutions. Combining pretraining with RL creates a powerful synergy of learning and search, where pretraining compresses existing human knowledge, and RL enables the model to explore new possibilities.

We chose coding tasks to explore how to employ RL to generate and refine reasoning data. Coding is a typical task that requires System-2 thinking, involving careful, logical, and step-by-step problem-solving. Moreover, coding can serve as a foundational skill for solving many other complex problems. This technical report presents our attempt to replicate o1 with a specific focus on coding tasks. The approach integrates RL and Monte Carlo Tree Search (MCTS) to enable self-play, allowing the model to continually generate reasoning data and enhance its System-2 capabilities.

2 FRAMEWORK OVERVIEW

There are two main challenges to address for self-play RL applied to code generation. The first challenge is result evaluation, i.e., assessing the quality of the generated code. Unlike tasks such as Go or mathematics, where results can be directly evaluated based on game rules or correct answers, evaluating code requires running the generated code within a testing environment and verifying it against test cases. We cannot assume that code datasets will always provide sufficient test cases. The second challenge involves defining the thinking and search behaviors, i.e., determining the object and granularity of process rewards. For code generation, the key question is how to design the reasoning process and the space of policies to guide the model’s behavior effectively.

To address the first challenge, we propose training a Test Case Generator (TCG), which automatically generates test cases based on the question and the ground-truth code¹. This approach will help build a standardized code testing environment, providing result rewards for reinforcement learning.

For the second challenge, two possible approaches can be considered. One is “think before acting”, where the model first forms a complete chain of thought and then generates the final answer all at once. The other approach, “think while acting” (Zelikman et al., 2024), involves generating parts of the answer while simultaneously reasoning through the task. We chose the former approach. For code generation, this means first thinking through and writing out a detailed pseudocode, which is then used to generate the final executable code. The advantages are twofold: adaptability, as the same pseudocode can lead to different concrete code implementations; and controllable granularity, as adjusting the level of detail in the pseudocode can be adjusted to control the granularity of the reasoning/search behavior.

The complete framework pseudocode is provided in Algorithm 1, which consists of six steps. (1) The first step is training the test case generator (TCG) γ_{TCG} , which is responsible for automatically generating test cases based on the question. (2) In the second step, we run MCTS on the original code dataset to generate code data with reasoning processes $\mathcal{D}_{\text{process}}$, including a validity indicator to

¹We also propose an alternative approach where test cases are generated based solely on the question. In addition to utilizing code datasets only provide questions, it can also be applied during the inference phase, enabling online reasoning without the need for predefined ground-truth code.

distinguish between correct and incorrect reasoning steps. (3) Once we have data that includes the reasoning process, the third step is to fine-tune the policy model π_θ , training it to behave in a “think before acting” manner. (4) The reasoning process data can also be used to initialize the process reward model (PRM) ρ_{PRM} , which evaluates the quality of reasoning steps. (5) The fifth step is the most crucial: with PRM ρ_{PRM} providing process rewards and TCG γ_{TCG} provides result rewards, the policy model π_θ is updated with reinforcement learning and MCTS. (6) In the 6th step, based on the updated policy model, new reasoning data can be generated. This new data can then be used to fine-tune the PRM again (4th step). Therefore, steps 4, 5, and 6 form an iterative cycle, where self-play continues to drive model improvements. The flow between the six steps is illustrated in Fig. 2. The following section will introduce each step in detail.

Algorithm 1 Self-Play+RL-based Coder Training Framework

Require:

- $\mathcal{D}_{\text{code}}$: A dataset containing problems Q_i and solution code C_i .
- π_θ : Initial policy model
- γ_{TCG} : Test Case Generator(TCG) to create problem-oriented test samples
- ρ_{PRM} : Process Reward Model(PRM) to evaluate the quality of intermediate reasoning steps
- ϕ : Aggregation function combining result-based and process-based rewards

Ensure:

Optimized policy model π_θ^*

- 1: Train γ_{TCG} on $\mathcal{D}_{\text{code}}$ to maximize diversity and correctness of generated test cases $\{(I_i, O_i)\}$.
▷ ① Train the Test Case Generator (TCG)
 - 2: Based on $\mathcal{D}_{\text{code}} = \{Q_i, C_i\}$, use MCTS to generate $\mathcal{D}_{\text{process}} = \{(Q_i, \dots, S_i^j, v_i^j, \dots, C_i') | j = 1, \dots, m\}$, where S_i^j represents a reasoning step and $v_i^j \in \{0, 1\}$ is a validity indicator with $v_i^m = 1$ when the generated code pass the test cases.
▷ ② Synthesize Reasoning-enhanced Code Dataset
 - 3: Finetune π_θ with SFT on valid steps $\mathcal{D}_{\text{process}}^+ = \{(Q_i, S_i^j, C_i') \mid (Q_i, S_i^j, v_i^j, C_i') \in \mathcal{D}_{\text{process}}, \mathbb{I}(C_i') = 1\}$.
▷ ③ Finetune the Policy Model
 - 4: **while** not converged **do**
▷ ④ Initialize/Finetune the Process Reward Model (PRM)
 - 5: Train/Finetune PRM using SFT on $\mathcal{D}_{\text{process}}$ with point-wise loss, or using DPO with pair-wise loss.
 - 6: Initialize $r_i = 0$.
▷ ⑤ Improve the Policy Model with Reinforcement Learning
 - 7: **for** $j = 1, 2, \dots, m$ **do**
 - 8: Generate reasoning step $S_i^j \sim \pi_\theta(S_i^j \mid Q_i, S_i^{1:j-1})$.
 - 9: Use PRM to compute process-based reward $r_i^j = \rho_{\text{PRM}}(Q_i, S_i^{1:j})$.
 - 10: **end for**
 - 11: Based on Q_i and the complete reasoning sequence $S_i^{1:m}$, generate the final code C_i' .
 - 12: Use TCG to generate test cases (I_i, O_i) for each problem Q_i with the ground-truth code C_i .
 - 13: Execute generated code C_i' on inputs I_i to produce outputs O_i' .
 - 14: Compute result-based reward:

$$R_i = \begin{cases} \tau_{\text{pass}}, & \text{if } O_i' = O_i, \\ \tau_{\text{fail}}, & \text{otherwise.} \end{cases}$$
 - 15: Update π_θ using a reinforcement learning method guided by the aggregated reward $\phi(R_i, r_i^{1:m})$.
▷ ⑥ Generate New Reasoning Data
 - 16: Generate new reasoning data $\mathcal{D}'_{\text{process}}$ using the updated π_θ .
 - 17: Update dataset: $\mathcal{D}_{\text{process}} \leftarrow \mathcal{D}_{\text{process}} \cup \mathcal{D}'_{\text{process}}$.
 - 18: **end while**
 - 19: **return** Optimized policy model π_θ^*
-

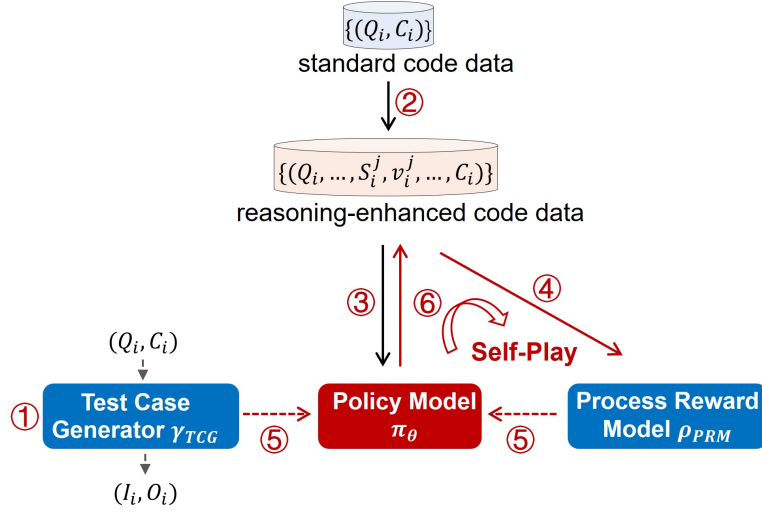


Figure 2: Self-Play+RL training framework.

3 METHOD AND INTERMEDIATE RESULTS

3.1 TEST CASE GENERATOR TRAINING

3.1.1 OBJECTIVE

A Test Case Generator is a tool designed to automate the creation of input-output test cases, which plays a critical role in supporting program verification in code generation tasks.

During the training phase, the correctness of the generated code is typically assessed with standard input-output test cases. The pass rate of these test cases serves as a key metric for evaluating the quality of the generated code and acts as an outcome reward signal to guide the training of the policy model. This reward signal helps the model refine its generation strategy, thereby enhancing its capability to produce accurate and functional code.

In the inference phase, when the trained model is tasked with code generation, standard test cases are often not available to verify the correctness of the generated code. The test case generator mitigates this limitation by providing a self-validation mechanism for the policy model, which allows the policy model to evaluate before final generation. As a result, the policy model is able to select the optimal output path based on the validation results.

3.1.2 TRAINING

The training process is divided into two distinct phases: Supervised Fine-Tuning (SFT) and Direct Preference Optimization (DPO) (Rafailov et al., 2024). We denote the generator which is not fine-tuned as $\gamma_{TCG_{base}}$.

The primary objective of the SFT phase is to ensure that the generator’s output adheres to a predefined format, enabling the accurate parsing and extraction of the generated test cases. The training data for this phase is derived from the TACO dataset (Li et al., 2023), which follows the format $\{question, solution, test_case\}$. To standardize the model’s input and output, we developed a template format, as detailed below:

Template format for TCG SFT

```

### Instruction
Please complete the task in the code part and generate some test case in the test part that can
be used to test the quality of the generated code.
### Problem
{question}
### Code Part
{randomly select one solution from the provided solutions}
### Test Part
[Generate 3 test cases here to validate the code]
{sample 3 test_cases with each formatted as input and output}

```

Figure 3: Template format for TCG SFT

The generator is denoted as $\gamma_{\text{TCG}_{sft}}$ after SFT.

The goal of the DPO phase is to guide the model in generating test cases that align with specific preferences, thereby enhancing both the performance and reliability of the test case generator. In this study, we employ the DPO method with artificially constructed sample pairs to improve the model’s ability to align with desired preferences by constructing a preference dataset. Our DPO fine-tuning relies on a pre-constructed preference dataset $D_{pref} = \{x, y_w, y_l\}$, where x is prompt that includes instruction, question, and code; y_w is positive example, i.e., test cases that align with the preference; and y_l is negative example, i.e., test cases that do not align with the preference. We adopt the following rules to construct preference data: for y_w , we directly use the three sampled test cases that are completely matched as positive examples; for y_l , we shuffle the outputs of the three sampled test cases and then concatenate the original inputs so that the input-output pairs of the three test cases do not completely match, and use the three incompletely matched test cases as negative examples. The training objective aims to optimize $\gamma_{\text{TCG}_\theta}$ based on initial SFT model $\gamma_{\text{TCG}_{sft}}$, while incorporating implicit reward modeling with the reference model $\gamma_{\text{TCG}_{ref}}$, which represents the initial SFT model $\gamma_{\text{TCG}_{sft}}$. The objective function is as follows:

$$\mathcal{L}_{\text{DPO}}(\gamma_{\text{TCG}_\theta}; \gamma_{\text{TCG}_{ref}}) = -\mathbb{E}_{(x, y_w, y_l) \sim D_{pref}} \left[\log \sigma \left(\beta \log \frac{\gamma_{\text{TCG}_\theta}(y_w|x)}{\gamma_{\text{TCG}_{ref}}(y_w|x)} - \beta \log \frac{\gamma_{\text{TCG}_\theta}(y_l|x)}{\gamma_{\text{TCG}_{ref}}(y_l|x)} \right) \right], \quad (1)$$

where $\sigma(x)$ is the sigmoid function and β represents a scaling factor used to adjust the contrast strength between the positive and negative examples during training. The generator is denoted as $\gamma_{\text{TCG}_{dpo}}$ after DPO, which represents the final generator γ_{TCG} .

3.1.3 EXPERIMENTS

We utilize DeepSeek-1.3B-Instruct (Guo et al., 2024) as the base model for the test case generator, followed by SFT and DPO. The fine-tuning phase employs QLoRA technology (Dettmers et al., 2023) with a rank parameter $r = 1$ to adapt the following modules: *q_proj*, *o_proj*, *k_proj*, *v_proj*, *gate_proj*, *up_proj*, *down_proj*. The learning rate is set to 5×10^{-4} to balance training stability and convergence speed. The training data is derived from a subset of the TACO train dataset, which adheres to the ACM competition format and contains approximately 10,000 samples. Similarly, the test data is obtained from a subset of the TACO test dataset, also conforming to the ICPC competition format, and consists of 314 samples.

We tested the quality of the generated test cases at different stages of the TACO test. After the SFT phase, the pass rate of test cases generated by $\gamma_{\text{TCG}_{sft}}$ on the standard code was 80.8%, demonstrating the generator’s ability to efficiently produce test cases following preliminary fine-tuning. Furthermore, $\gamma_{\text{TCG}_{dpo}}$ achieved a performance of 89.2%, reflecting a notable improvement compared to $\gamma_{\text{TCG}_{sft}}$. This indicates that preference optimization, by refining the model’s decision-making process, significantly enhanced the generator’s ability to produce more reliable test cases.

In practical scenarios, the generator’s performance has generally met the requirements for assessing code correctness. Looking ahead, we plan to explore the DPO method, where the model autonomously generates data during inference, potentially optimizing the generator’s performance further.

Additionally, we are considering the incorporation of self-play in the TCG’s training. In this setup, the policy model would generate code intended to pass the test cases produced by the TCG, while the TCG would aim to generate progressively more challenging test cases. This adversarial interaction could foster mutual improvements in both the policy model and the test case generator.

Pseudocode Prompt

Instruction

Please refer to the given task description and provide a thought process in the form of step-by-step pseudocode refinement.

A curious user has approached you with a programming question. You should give step-by-step solutions to the user’s questions. For each step you can choose one of the following three actions:

<Action 1> Defining algorithm Structures Using pseudocode
Description: Outline the core functions and overall structure of the solution without getting into implementation details. Define inputs, outputs, and the main tasks each function will perform.

<Action 2> Refine part of the pseudocode
Description: Add more details to the pseudocode, specifying the exact steps, logic, and operations each function will carry out. This prepares the pseudocode for actual coding.

<Action 3> Generate python code from the pseudocode
Description: Translate the refined pseudocode into executable Python code, making sure to handle inputs, outputs, and ensure correctness in the implementation.

Note:

- You can choose one of the three actions for each step.
- Provide a detailed explanation of the reasoning behind each step.
- Try to refer to the reference code as much as possible, but you can also modify it if needed (e.g. change variable names, add some comments, etc.).

Examples

{examples}

Question

{question}

Figure 4: Pseudocode Prompt for Step-by-Step Refinement

3.2 REASONING-ENHANCED CODE DATA SYNTHESIS

3.2.1 PSEUDOCODE-BASED REASONING PROCESS

The definition of the reasoning process is crucial. As mentioned in the *Introduction*, we explore a pseudocode-based prompting approach designed to guide large language models in deep reasoning for complex code tasks. Pseudocode, serving as an intermediate representation between natural language descriptions and actual code, offers a more abstract and concise way to express the logical flow of algorithms or programs. To integrate pseudocode reasoning into step-level Chain-of-Thought

Model	Qwen2.5-1.5B		Qwen2.5-3B		Qwen2.5-7B		Qwen2.5-Coder-7B	
	Vanilla	Pseudocode	Vanilla	Pseudocode	Vanilla	Pseudocode	Vanilla	Pseudocode
Pass@1(%)	55.8	46.7(-9.1)	56.3	51.3(-5.0)	59.8	50.1(-9.7)	57.7	58.2(+0.5)
ASPR(%)	49.9	54.5(+4.6)	52.0	70.6(+18.6)	66.4	78.1(+11.7)	49.3	74.9(+25.6)

Table 1: Pseudocode-based code generation results on the MBPP Benchmark. *Pass@1* indicates the overall pass rate. *ASPR* (Average Sampling Pass Rate) indicates the average success rate of reaching the correct reasoning path on the last step.

(CoT), as illustrated in Fig. 4, we define three key behavioral actions infused with pseudocode reasoning:

- *Action 1: Defining Algorithm Structures using Pseudocode:* In this action, the model outlines the structure and interface of the main functions, without delving into implementation details. The aim is to enable the model to grasp the overall task structure, including the inputs, outputs, and core functionalities of each primary function.
- *Action 2: Refining the Pseudocode:* In this action, the model iteratively refines the pseudocode defined in Action 1, progressively clarifying the steps, logic, and operations of each function in preparation for the final code implementation.
- *Action 3: Generating Code from the Pseudocode:* The goal of this action is to accurately translate the structure and logic of the pseudocode into executable code, ensuring that the generated code meets the task requirements.

These actions ensure that the model employs pseudocode as a cognitive tool during the reasoning process, enhancing its reasoning capability for complex code generation tasks. It is important to note that these three actions do not imply that the reasoning chain is limited to only these steps. As demonstrated in Fig. 5, the model may need to repeatedly invoke Action 2 throughout the reasoning process to iteratively refine the pseudocode until it is sufficiently developed for the final code generation.

To evaluate the effectiveness of the step-level CoT with pseudocode reasoning, we conducted experiments using the Qwen series of open-source models (Yang et al., 2024) and the Mostly Basic Python Problems (MBPP) dataset (Austin et al., 2021) as the benchmark. In the experiment, we employed a sampling strategy based on Monte Carlo Tree Search (MCTS) and compared Pass@1 for regular CoT and CoT with pseudocode reasoning, as well as the Average Sampling Pass Rate (ASPR) of the last step on the correct reasoning path. Our results indicate that incorporating pseudocode significantly improves the quality of the generated code when the reasoning is correct.

Table 1 presents the results. While the Pass@1 metric generally decreases with pseudocode-based reasoning, we observed a significant increase in ASPR, indicating that pseudocode enhances the overall reasoning process, particularly in refining the path toward the correct final output. This suggests that accurate pseudocode highly contributes to the final correct code. However, vanilla LLMs still face challenges in generating effective pseudocode, which is precisely the goal of the subsequent SFT initialization and Self-Play+RL enhancement.

3.2.2 REASONING PROCESS DATA SYNTHESIS

We use Monte Carlo Tree Search (MCTS) (Kocsis & Szepesvári, 2006; Feng et al., 2023; Qi et al., 2024) to construct step-level process reward data in the form of $\mathcal{D}_{\text{process}} = \{(Q_i, \dots, S_i^j, v_i^j, \dots, C_i')\}$, where v_i^j represents the evaluation of the reasoning path up to step S_i^j , and C_i' is the executable code derived from the final step S_i^m . In this process, we employ the standard MCTS rollout strategy for path exploration. For each problem Q_i , we apply the pseudocode prompt strategy defined earlier to guide the reasoning process. When a terminal node S_i^m is reached, a complete pseudocode reasoning path $(Q_i, S_i^1, \dots, S_i^m)$ is formed. The reward value v_i^m for the terminal node S_i^m is computed based on two key metrics:

Question
<p>There are two n-element arrays of integers, A and B. Permute them into some A' and B' such that the relation $A'[i] + B'[i] \geq k$ holds for all i where $0 \leq i < n$. There will be q queries consisting of A, B, and k. For each query, return YES if some permutation A', B' satisfying the relation exists. Otherwise, return NO.</p> <p>The first line contains an integer q, the number of queries. The next q sets of 3 lines are as follows:</p> <p>The first line contains two space-separated integers n and k, the size of both arrays A and B, and the relation variable.</p> <p>The second line contains n space-separated integers $A[i]$.</p> <p>The third line contains n space-separated integers $B[i]$.</p> <p>Sample Input</p> <pre>1 3 10 2 1 3 7 8 9</pre> <p>Sample Output</p> <pre>YES</pre>
Response
<p>Step 1: Defining High-Level Structure (Action 1)</p> <p>Goal:</p> <p>Define the overall structure of the solution by describing the steps needed to solve the problem:</p> <ol style="list-style-type: none"> 1. Read the number of queries. 2. For each query, read the sizes of the arrays and the integer k. 3. Read and sort both arrays. 4. Check if the condition can be met by comparing corresponding elements from A' and B'. <p>Pseudocode:</p> <pre>Function twoArrays(q): For each query from 1 to q: Read n and k Read array A of size n Read array B of size n Sort array A in ascending order Sort array B in descending order If for every index i, A[i] + B[i] >= k: Return "YES" Else: Return "NO"</pre> <p>Step 2: Reading Input and Sorting (Action 2)</p> <p>Goal:</p> <p>Implement the input reading and array sorting functionality. Ensure that the sizes and contents of the arrays are correctly processed.</p> <p>Pseudocode:</p> <pre>Function readInputAndSort(): Initialize a list for results For each query: Read n and k Read array A Read array B Sort A in ascending order Sort B in descending order Append sorted arrays to results Return results</pre> <p>Step 3: Checking Conditions (Action 2)</p> <p>Goal:</p> <p>Create a function to check if the condition $A[i] + B[i] \geq k$ holds for all indices after sorting.</p> <p>Pseudocode:</p> <pre>Function checkValidity(A_sorted, B_sorted, k): For i from 0 to n-1: If A_sorted[i] + B_sorted[i] < k: Return "NO" Return "YES"</pre> <p>Step 4: Putting It All Together (Action 2)</p> <p>Goal:</p> <p>Combine the input reading, sorting, and checking into a single coherent function for easier overall management.</p> <p>Pseudocode:</p> <pre>Function twoArrays(q): results = [] For i from 1 to q: Read n and k Read arrays A and B Sort A ascending Sort B descending If checkValidity(A_sorted, B_sorted, k) is "YES": results.append("YES") Else: results.append("NO") Return results</pre> <p>Step 5: Implementing Python Code (Action 3)</p> <p>Python Code:</p> <p>The defined structure is implemented in Python as follows:</p> <pre>def twoArrays(q): results = [] for _ in range(q): n, k = map(int, input().split()) a_array = list(map(int, input().split())) b_array = list(map(int, input().split())) # Sort A in ascending order and B in descending order a_array.sort() b_array.sort(reverse=True) # Check validity valid = True # Assume valid until proven otherwise for i in range(n): if a_array[i] + b_array[i] < k: valid = False # Found a case that fails the condition break results.append("YES" if valid else "NO") return results # Read number of queries q = int(input()) # Process and print results for all queries results = twoArrays(q) for res in results: print(res)</pre> <p>This code accurately reads multiple queries, sorts the arrays, checks the required conditions, and outputs the results as specified.</p>

Figure 5: Generated example code with pseudocode CoT

- *Compilation success rate (compile)*: This metric determines whether the generated code can successfully compile. The value *compile* is binary, with *compile* = 1 indicating success and *compile* = 0 indicating failure.
- *Test case pass rate (pass)*: Given a successful compilation, we further evaluate whether the generated code passes the test cases. The pass rate is calculated as $pass = \frac{\text{Num}_{\text{passed}}}{\text{Num}_{\text{test_case}}}$, where $\text{Num}_{\text{passed}}$ is the number of passed test cases and $\text{Num}_{\text{test_case}}$ is the total number of test cases used for validation.

The reward value for the terminal node S_i^m is calculated as a weighted sum of these two metrics:

$$v_i^m = \alpha \cdot \text{compile} + (1 - \alpha) \cdot \text{pass},$$

where α is a hyperparameter controlling the relative importance of compilation success and test pass rate.

Once the reward value v_i^m is computed for the terminal node, we backpropagate this value to all preceding nodes along the path, assigning a reward value v_i^j to each step (S_i^j, v_i^j) . Due to the multiple rollouts in the MCTS process, the cumulative reward for a node v_i^j during backpropagation may exceed 1. Therefore, we normalize the reward values for each node along the path using the following formula to obtain the final step validity value.

When constructing the reasoning process dataset, for each problem Q_i , if a correct answer is found through the search, we are guaranteed to obtain at least one terminal node (S_i^m, v_i^m) with $v_i^m = 1$. After completing the search, we select the full reasoning path from the correct terminal node $(Q_i, S_i^1, \dots, S_i^m, v_i^m)$, $v_i^m = 1$ to form the initialization dataset for the policy model. This dataset is denoted as:

$$\mathcal{D}_{\text{process}}^+ = \{(Q_i, S_i^j, C'_i) \mid (Q_i, S_i^j, v_i^j, C'_i) \in \mathcal{D}_{\text{process}}, \mathbb{I}(C'_i) = 1\},$$

where $\mathbb{I}(\cdot)$ is an indicator function that returns 1 if the generated code C'_i passes all the test cases.

3.3 POLICY MODEL INITIALIZATION

After completing the reasoning data synthesis tasks described in Section 3.2, we use each complete reasoning solution in the dataset to initialize the policy model π_θ . This step aims to help π_θ better understand the task requirements and follow the expected action behavior, providing an optimal starting point for subsequent iterative training.

Given the question Q_i , the specific reasoning step content generated by the policy model π_θ at step j can be expressed as $\pi_\theta(S_i^j \mid Q_i, S_i^{1:j-1})$, where $S_i^j = (w_1, w_2, \dots, w_k)$. Here, S_i^j represents the content of a reasoning step, delimited by specific separators, with w denoting the tokens generated by π_θ at each decoding step. $S_i^{1:j-1}$ represents the context formed by the outputs of the previous reasoning steps.

The policy model π_θ is then initialized using the set of verified, correct reasoning solutions $\mathcal{D}_{\text{process}}^+$. This initialization is performed by optimizing the following training objective:

$$\mathcal{L}_{\text{SFT}} = - \sum_{(Q_i, S_i^j, C'_i) \in \mathcal{D}_{\text{process}}^+} \log \pi_\theta(S_i^{1:m} \circ C'_i \mid Q_i), \quad (2)$$

where \circ denotes the concatenation of the reasoning steps $S_i^{1:m}$ and the final code C'_i . The initialized policy model π_θ^{SFT} will then serve as the foundation for subsequent training stages.

3.4 PRM TRAINING

Given a problem Q_i and a solution prefix corresponding to the current state, the Process Reward Model (PRM), denoted as $Q \times S \rightarrow \mathbb{R}^+$, assigns a reward value to the current step S_i^j to estimate its contribution to the final answer. Based on the tree search approach used during data synthesis

in Section 3.2, two formats of data organization can be used for training the process reward model, referred to as point-wise and pair-wise, are described in detail below.

Point-wise In this format, data collected from the search tree are organized as $D = \{(Q_i, S_i^{1:j-1}, S_i^j, v_i^j) \mid i = 1, 2, \dots, N\}$, where N is the number of samples, and v_i^j represents the value label assigned to step S_i^j during the tree search process. Depending on the processing method, this label can be used to derive either hard or soft estimates. Following the approach in (Wang et al., 2024), the PRM is trained using the objective:

$$\mathcal{L}_{\text{PRM}}^{\text{point-wise}} = -\mathbb{E}_{(Q_i, S_i^{1:j-1}, S_i^j, v_i^j) \sim D} \left[v_i^j \log r(Q_i, S_i^{1:j}) + (1 - v_i^j) \log (1 - r(Q_i, S_i^{1:j})) \right], \quad (3)$$

where $r(Q_i, S_i^{1:j})$ is the normalized prediction score assigned by the PRM.

Pair-wise In the pair-wise format, for a node n^d at depth d of the search tree, with its child nodes represented as $\sum_i n_i^{d+1}$, preference pair data are organized as $D_{\text{pair}} = \{(Q_i, S_i^{1:j-1}, S_i^{j_{\text{win}}}, S_i^{j_{\text{lose}}}) \mid i = 1, 2, \dots, N\}$. Here, $S_i^{j_{\text{win}}}$ represents the reasoning step that achieved a higher value estimate during the tree search compared to $S_i^{j_{\text{lose}}}$.

Following the Bradley-Terry model (Bradley & Terry, 1952), the PRM is trained using the following objective:

$$\mathcal{L}_{\text{PRM}}^{\text{pair-wise}} = -\mathbb{E}_{(Q_i, S_i^{1:j-1}, S_i^{j_{\text{win}}}, S_i^{j_{\text{lose}}}) \sim D_{\text{pair}}} \left[\log \left(\sigma(r(Q_i, S_i^{1:j-1}, S_i^{j_{\text{win}}}) - r(Q_i, S_i^{1:j-1}, S_i^{j_{\text{lose}}})) \right) \right], \quad (4)$$

where $\sigma(x)$ denotes the sigmoid function. Unlike the point-wise setting, the scores r here are not normalized. This enables the model to focus on learning relative preferences between actions rather than absolute value predictions.

3.5 RL-BASED POLICY MODEL IMPROVEMENT

We model the code generation task as a language-augmented Markov Decision Process (MDP), formally represented as $\mathcal{M} = (\mathcal{V}, \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \phi)$ (Team, 2024; Carta et al., 2023). In this framework, \mathcal{V} denotes the vocabulary, and $w \in \mathcal{V}$ represents an individual token generated by the model. The action space $\mathcal{A} \subseteq \mathcal{V}^N$ and the state space $\mathcal{S} \subseteq \mathcal{V}^N$ are sets of token sequences, meaning that both actions and states are sequences of tokens. In this framework, s_0 represents the question, and the action a_i is considered a reasoning step (referring to the S_i in algorithm 1), which consists of both the type of action and its corresponding chain of thought. The state transition function $\mathcal{T} : \mathcal{S} \times \mathcal{A} \rightarrow \mathcal{S}$ defines how the current state $s_t \in \mathcal{S}$ changes when an action $a_t \in \mathcal{A}$ is taken. Specifically, the action a_t appends tokens to the current state, forming a new state $s_{t+1} = \mathcal{T}(s_t, a_t)$. This process continues until the model generates the final solution. The reward function $\mathcal{R} : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}^+$ evaluates the quality of intermediate steps, such as the reasoning process or generated code fragments. The function ϕ combines process-based and outcome-based rewards to produce a final reward signal.

At each step, the model selects an action $a_t \in \mathcal{A}$, which transitions the system to a new state $s_{t+1} = \mathcal{T}(s_t, a_t)$. After executing the action, the model receives a process reward $r^t = \rho_{\text{PRM}}(s_{t-1}, a_t)$ from PRM. This process repeats until the model either generates the final code or reaches the predefined maximum depth.

Once the model generates the final code or completes the search process, the outcome reward R_i is evaluated by testing the generated code against a series of test cases. We propose a reward aggregation function that incorporates both time-dependent weights and a discount factor:

$$\phi(R_i, r_i^{1:m}) = \alpha(t) \cdot R_i + (1 - \alpha(t)) \cdot \frac{1}{m} \sum_{j=1}^m \gamma^j r_i^j,$$

where $\alpha(t)$ is a time-varying factor that adjusts the balance between the final reward R_i and the cumulative intermediate rewards $r_i^{1:m}$ over time. For instance, $\alpha(t)$ may decrease over time, gradually placing more weight on the intermediate rewards as the model refines its solution, while reducing the emphasis on the final reward as the model approaches the optimal policy. $r_i^{1:m}$, with $\alpha(t)$ typically following schedules such as linear or logarithmic decay. The parameter $\gamma \in [0, 1]$ is the discount

factor, which determines the importance of future rewards relative to immediate rewards. The aggregated reward signal is employed to refine the model’s policy, typically through the implementation of reinforcement learning algorithms such as PPO (Ziegler et al., 2019) and iterative DPO (Rafailov et al., 2024).

With this setup, we define a reinforcement learning environment tailored for the code generation task. The model’s actions are driven by both process-based rewards, which encourage intermediate reasoning steps, and outcome-based rewards, which reflect the correctness of the final code. This dual reward structure helps the model improve its code generation ability over time.

3.6 NEW REASONING DATA GENERATION AND SELF-PLAY

In step 6, the updated policy model π_θ is used to generate new reasoning data, denoted as $\mathcal{D}'_{\text{process}}$. This data is created by reasoning through new problem instances Q_i , generating step-by-step reasoning paths $\{S_i^1, S_i^2, \dots, S_i^m\}$, with each path culminating in a final code output C'_i . The reasoning steps are generated iteratively, where each step S_i^j is conditioned on the previous steps.

Once the new reasoning data is generated, it is added to the existing dataset $\mathcal{D}_{\text{process}}$ to form an updated dataset $\mathcal{D}_{\text{process}} \leftarrow \mathcal{D}_{\text{process}} \cup \mathcal{D}'_{\text{process}}$. This update increases the diversity and quality of the reasoning examples, providing more comprehensive training material for subsequent steps.

This new data generation process enables the iterative self-play training loop. After adding the new reasoning data, the model undergoes further fine-tuning, starting with updating PRM as described in the 4th step. The PRM, in turn, adjusts the policy model with RL described in the 5th step. This iterative cycle of data generation, reward model updating, and policy improvement ensures sustained improvement in the system’s reasoning ability.

4 DISCUSSIONS

4.1 BITTER LESSON: DATA IS ALL YOU NEED

Over the last decade, the AI field has been developing along a central line towards maximizing computation-intelligence conversion efficiency, which is to efficiently convert the ever-increasing computing power into higher intelligence levels. Along this line, as illustrated at the top of Fig. 6, early advancements prioritized improvements on the model side: from SVM to DNN and then to Transformer, scalable model architectures were designed to fully leverage computational power.

In recent years, the focus has shifted towards the data side. Techniques such as Semi-Supervised Learning (SSL) in pre-training and Reinforcement Learning (RL) in post-training have aimed to harness data more effectively. The o1 model continues this line. It moves from SFT, which leverages high-quality supervised data, to RLHF, which utilizes environmental feedback to access theoretically unlimited data, and finally to o1’s innovative approach of supervising the generation process through reward signals derived from the generated reasoning process itself.

This progression suggests that, with Transformer architectures now capable of scaling to handle vast amounts of data and training models of sufficient size, the only remaining challenge converges to acquiring adequate data. One approach is to seek data wherever it is lacking, such as reasoning data for system-2 abilities or physical world trajectories for embodied intelligence. Another approach is to explore data types that do not yet exist in the human world, which requires further exploration of techniques like RL and Self-Play.

4.2 SWEET LESSON: BEYOND HUMAN DATA

A common criticism of LLM is its reliance on existing human-recorded data, which inherently limits their potential. As Wittgenstein stated, “The limits of my language mean the limits of my world.” The finite scope and depth of human language records constrain the cognitive capabilities of LLMs. However, the success of o1 demonstrates that we can now explore the underlying thought processes behind these recorded data through RL. This advancement signifies a pivotal shift in AI development, moving from mere imitation of human language to the autonomous generation of novel cognitive processes.

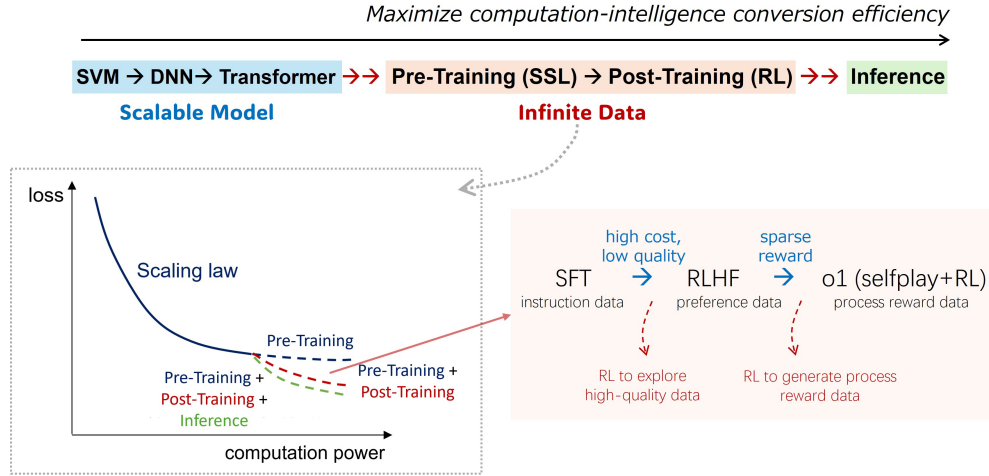


Figure 6: The trend towards maximizing computation-intelligence conversion efficiency.

More interestingly, these thought process data do not necessarily be confined to natural language. As highlighted in a recent Nature paper (Fedorenko et al., 2024), “language serves primarily as a tool for communication rather than the essence of thought.” In our observations, some of the thought chains generated by o1 contain nonsensical text, suggesting that the thinking tokens may not correspond to discrete natural language words. If the model has developed itself a more efficient form of internal representation for thinking, this will significantly elevate the efficiency of thought processes and problem-solving mechanisms, not only transcending the limitations imposed by human language data but also further unlocking the potential of model capabilities.

4.3 OPPORTUNITIES

The self-play+RL framework provides a viable solution for exploring underlying data, which opens up the possibility of exploring System-2 solutions for many tasks that were previously reliant on System 1 capabilities. By integrating more thoughtful, step-by-step processes into task execution, we believe that this approach can yield positive results across a wide range of domains (Kant et al., 2024; Ganapini et al., 2021; Valmeekam et al., 2024; Lowe, 2024). Tasks traditionally solved using System 1 capabilities, such as reward modeling (Mahan et al., 2024), machine translation (Zhao et al., 2024), retrieval-augmented generation (RAG) (Li et al., 2024), and multimodal QA (Islam et al., 2024), have already benefited from the deeper reasoning capabilities enabled by System-2 thinking.

The o1 model’s system card demonstrates notable improvements in model safety. Inspired by this, we have recently explored the concept of *System-2 Alignment*, which involves guiding models to thoroughly evaluate inputs, consider potential risks, and correct biases in their reasoning (Wang & Sang, 2024). We introduced three methods to realize System-2 alignment: prompt engineering, supervised fine-tuning, and reinforcement learning with process supervision. We will apply the Self-Play+RL framework presented in this report to System-2 alignment, aiming to further enhance the model’s ability to think deliberately and reduce vulnerabilities in complex scenarios.

4.4 CHALLENGES

The released o1-preview and o1-mini currently lack multimodal capabilities and functional call features, which are claimed by OpenAI to be included in its complete version. Beyond multimodal and functional call, another critical feature for improvement in o1-like inference models is the optimization of inference time. This includes enhancing inference efficiency—achieving higher performance per unit of time—and enabling adaptive inference time adjustments. Specifically, this involves dynamically adjusting the System 2 reasoning process based on task complexity and achieving a more human-like ability to seamlessly switch between System 1 and System 2 reasoning modes.

For o1-like inference models to be deployed across broader real-world applications, two major challenges need to be addressed, both involving with the RL environments. The first challenge concerns reward function generalization. This has been already discussed in the community. For example, leveraging the enhanced ability of inference models to understand high-level natural instructions, approaches like Constitutional AI (Bai et al., 2022) might directly define reward functions in natural language. Another strategy focuses on improving coding capability and transforming the other tasks into coding problems for resolution.

Another less mentioned challenge concerns environment state update. Unlike classic model-free RL methods, such as Q-learning, where state transitions are not explicitly modeled, o1-like models rely on behavior simulation and forward search, requiring knowledge of the updated state following an action. This shifts the paradigm towards model-based RL. In well-defined domains such as programming, mathematics, and Go, the environment often has deterministic rules. For example, programming uses compiler-defined language specifications, mathematics adheres to axiomatic logic, and Go operates under fixed game rules. These deterministic frameworks allow precise computation of state transition probabilities $p(\text{state}_{i+1} \mid \text{state}_i, \text{action}_i)$ following specific actions.

However, in many real-world applications, such as Retrieval-Augmented Generation (RAG), device usage (), and embodied agents, obtaining state updates requires interaction with external environments or simulators. This introduces significant computational and time costs. For example, in device use, behaviors like clicking, inputting, or scrolling must be simulated in a way that involves page rendering, state updates, and sometimes complex backend interactions like network requests. Moreover, o1-like models face the limitation of not being able to perform online behavior simulation during inference, which prevents the model from validating or correcting its actions by returning to a previous state. This leads to inability to backtrack and refine decisions.

Therefore, one of the key directions is to attempt explicit modeling of the environment by developing a world model for state transition prediction. The world model takes as input the current and past states and actions, and produces the next state as output. This allows the model to interact with its internal world model, rather than directly with the real environment or a simulator. We recognize that one of the ongoing challenges in RL when building such world models is ensuring their accuracy. As a result, world models have typically been applied to environments where the dynamics are relatively simple and well-understood. The good news is, recent rapid advancements in generative games (Sang, 2024) offer promising progress that could facilitate more accurate and practical environment modeling for inference models in real-world applications.

Prospects. The o1 model is clearly influenced by AlphaGo: AlphaGo utilized imitation learning to initialize the policy network, reinforcement learning to fine-tune the policy and learn the value network, and MCTS as an online search strategy, which parallels LLM’s pre-training, post-training, and inference. AlphaGoZero took a more advanced approach by not relying on historical data, which exactly mirrors current trends in LLM development increasingly emphasizing the post-training stage. If we follow the evolution of the Alpha series, we can anticipate similar developments in o1-like inference models. Initially, the Alpha series developed towards generalization: AlphaZero was applied to Go, Chess, and Shogi, while MuZero achieved human-level performance across 57 Atari games. The other goal besides generalization, however, is to apply these models to more complex, real-world tasks. This progression is evident in AlphaFold’s leap to AlphaCode and AlphaGeometry, as well as the extension of AI to physical environments, such as the 3D virtual agents in SIMA or the embodied intelligence in RT-X.

ACKNOWLEDGEMENTS

We thank Yuhang Wang and Jing Zhang for their fruitful discussions and participation.

REFERENCES

Open o1: A model matching proprietary power with open-source innovation. <https://github.com/Open-Source-O1/Open-O1/>, 2024.

-
- Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.
- Yuntao Bai, Saurav Kadavath, Sandipan Kundu, Amanda Askell, Jackson Kernion, Andy Jones, Anna Chen, Anna Goldie, Azalia Mirhoseini, Cameron McKinnon, et al. Constitutional ai: Harmlessness from ai feedback. *arXiv preprint arXiv:2212.08073*, 2022.
- Benjamin Klieger. g1: Using Llama-3.1 70b on Groq to create o1-like reasoning chains. <https://github.com/bklieger-groq/g1>, 2024.
- Ralph Allan Bradley and Milton E Terry. Rank analysis of incomplete block designs: I. the method of paired comparisons. *Biometrika*, 39(3/4):324–345, 1952.
- Thomas Carta, Clément Romac, Thomas Wolf, Sylvain Lamprier, Olivier Sigaud, and Pierre-Yves Oudeyer. Grounding large language models in interactive environments with online reinforcement learning. In *International Conference on Machine Learning*, pp. 3676–3713. PMLR, 2023.
- Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. Qlora: Efficient finetuning of quantized llms. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (eds.), *Advances in Neural Information Processing Systems*, volume 36, pp. 10088–10115. Curran Associates, Inc., 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/1feb87871436031bdc0f2beaa62a049b-Paper-Conference.pdf.
- Evelina Fedorenko, Steven T. Piantadosi, and Edward A. Gibson. Language is primarily a tool for communication rather than thought. *Nature*, 615:75–82, 2024.
- Xidong Feng, Ziyu Wan, Muning Wen, Stephen Marcus McAleer, Ying Wen, Weinan Zhang, and Jun Wang. Alphazero-like tree-search can guide large language model decoding and training. *arXiv preprint arXiv:2309.17179*, 2023.
- GAIR-NLP. O1 replication journey: A strategic progress report, 2024.
- Marianna Bergamaschi Ganapini, Murray Campbell, Francesco Fabiano, Lior Horesh, Jon Lenchner, Andrea Loreggia, Nicholas Mattei, Francesca Rossi, Biplav Srivastava, and Kristen Brent Venable. Thinking fast and slow in ai: the role of metacognition, 2021.
- Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, et al. Deepseek-coder: When the large language model meets programming—the rise of code intelligence. *arXiv preprint arXiv:2401.14196*, 2024.
- Mohammed Saidul Islam, Raian Rahman, Ahmed Masry, Md Tahmid Rahman Laskar, Mir Tafseer Nayeem, , and Enamul Hoque. Are large vision language models up to the challenge of chart comprehension and reasoning? *Findings of the Association for Computational Linguistics: EMNLP 2024*, 2024(November):3334–3368, 2024.
- Yichao Ji. A small step towards reproducing openai o1: Progress report on the steiner open source models, October 2024. URL <https://medium.com/@peakji/b9a756a00855>.
- Manuj Kant, Marzieh Nabi, Manav Kant, Preston Carlson, and Megan Ma. Equitable access to justice: Logical llms show promise. *arXiv preprint arXiv:2410.09904*, 2024.
- Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *European conference on machine learning*, pp. 282–293. Springer, 2006.
- Huayang Li, Pat Verga, Priyanka Sen, Bowen Yang, Vijay Viswanathan, Patrick Lewis, Taro Watanabe, and Yixuan Su. Alr2: A retrieve-then-reason framework for long-context question answering. *arXiv preprint arXiv:2410.03227*, 2024.
- Rongao Li, Jie Fu, Bo-Wen Zhang, Tao Huang, Zhihong Sun, Chen Lyu, Guang Liu, Zhi Jin, and Ge Li. Taco: Topics in algorithmic code generation dataset. *arXiv preprint arXiv:2312.14852*, 2023.

-
- Scott C. Lowe. System 2 reasoning capabilities are nigh, 2024.
- Dakota Mahan, Duy Van Phung, Rafael Rafailov, Chase Blagden, Nathan Lile, Louis Castricato, Jan-Philipp Fränken, Chelsea Finn, and Alon Albalak. Generative reward models, 2024.
- OpenAI. Learning to reason with large language models. <https://openai.com/index/learning-to-reason-with-llms/>, 2024.
- Zhenting Qi, Mingyuan Ma, Jiahang Xu, Li Lyna Zhang, Fan Yang, and Mao Yang. Mutual reasoning makes smaller llms stronger problem-solvers. *arXiv preprint arXiv:2408.06195*, 2024.
- Qwen Team. QwQ-32b-preview. <https://qwenlm.github.io/zh/blog/qwq-32b-preview/>, 2024.
- Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. Direct preference optimization: Your language model is secretly a reward model. *Advances in Neural Information Processing Systems*, 36, 2024.
- Richards Tu. Thinking Claude. <https://github.com/richards199999/Thinking-Claude/tree/main>, 2024.
- Jitao Sang. A note on generative games: Positioning, progress and prospects. *arXiv*, 2024.
- Shanghai AI Lab. InternThinker. <https://internlm-chat.intern-ai.org.cn>, 2024.
- SimpleBerry. Llama-o1: Open large reasoning model frameworks for training, inference and evaluation with pytorch and huggingface. <https://github.com/SimpleBerry/LLaMA-O1>, 2024. Accessed: 2024-11-25.
- OpenR Team. Openr: An open source framework for advanced reasoning with large language models. <https://github.com/openreasoner/openr>, 2024.
- Karthik Valmeekam, Kaya Stechly, Atharva Gundawar, and Subbarao Kambhampati. Planning in strawberry fields: Evaluating and improving the planning and scheduling capabilities of lrm o1, 2024.
- Peiyi Wang, Lei Li, Zhihong Shao, Runxin Xu, Damai Dai, Yifei Li, Deli Chen, Yu Wu, and Zhi-fang Sui. Math-shepherd: Verify and reinforce LLMs step-by-step without human annotations. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 9426–9439, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.510. URL <https://aclanthology.org/2024.acl-long.510>.
- Yuhang Wang and Jitao Sang. Don’t command, cultivate: An exploratory study of system-2 alignment, 2024.
- Guowei Xu, Peng Jin, Li Hao, Yibing Song, Lichao Sun, and Li Yuan. Llava-o1: Let vision language models reason step-by-step, 2024. URL <https://arxiv.org/abs/2411.10440>.
- An Yang, Baosong Yang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Zhou, Chengpeng Li, Chengyuan Li, Dayiheng Liu, Fei Huang, et al. Qwen2 technical report. *arXiv preprint arXiv:2407.10671*, 2024.
- Eric Zelikman et al. Quiet-star: Language models can teach themselves to think before speaking. *arXiv preprint arXiv:2403.09629*, 2024. URL <https://arxiv.org/abs/2403.09629>.
- Di Zhang, Jianbo Wu, Jingdi Lei, Tong Che, Jiatong Li, Tong Xie, Xiaoshui Huang, Shufei Zhang, Marco Pavone, Yuqiang Li, Wanli Ouyang, and Dongzhan Zhou. Llama-berry: Pairwise optimization for o1-like olympiad-level mathematical reasoning. *arXiv preprint arXiv:2410.02884*, 2024. URL <https://arxiv.org/abs/2410.02884>. Accessed: 2024-11-25.
- Yu Zhao, Huifeng Yin, Bo Zeng, Hao Wang, Tianqi Shi, Chenyang Lyu, Longyue Wang, Weihua Luo, and Kaifu Zhang. Marco-o1: Towards open reasoning models for open-ended solutions, 2024.

Daniel M Ziegler, Nisan Stiennon, Jeffrey Wu, Tom B Brown, Alec Radford, Dario Amodei, Paul Christiano, and Geoffrey Irving. Fine-tuning language models from human preferences. *arXiv preprint arXiv:1909.08593*, 2019.