強化学習を用いた低リソースプログラミング言語におけるコード生成精度改善

井上貴之 [1]　鶴岡慶雅 [1]
[1] 東京大学大学院情報理工学系研究科電子情報学専攻
{inoue, tsuruoka}@logos.t.u-tokyo.ac.jp

概要

大規模言語モデルのコーディング能力には事前学習データの差によりプログラミング言語間での差が依然として聳えている. 本研究では大規模言語モデル自身により自己生成したカリキュラムデータにより学習データを補完し, Proximal Policy Optimization による強化学習を行い, リソースの乏しい言語における生成精度および推論能力の向上可能性について検証した. 結果, 対象言語において高リソース言語を上回る精度向上が確認され, さらに報酬設計の改修によりさらなる精度向上が確認できた.

1　はじめに

2025 年初頭に提案された Group Relative Policy Optimization (GRPO) [1] をはじめとする強化学習手法の発展により, 大規模言語モデル (Large Language Model, LLM) [2] の推論, コーディング面の能力は大きく向上した. 一方で LLM の持つ知識情報は事前学習におけるコーパスデータに依存するため, 学習可能なデータの乏しいプログラミング言語 (Low-Resource Programming Languages, LRPLs) に関しては事前学習の段階で LLM は十分な知識を獲得できず, これはその後の推論能力にも影響する. これは使用人口が少ないことに起因する収集可能なオープンソースコードおよびレポジトリの不足がその要因となる. LRPLs においては同程度のロジックを要求するタスクにおいても使用人口の多い Python, JavaScript などと比べ精度にギャップが生じている [3, 4]. さらには評価指標となるコード生成タスク用ベンチマークに関しても Python を中心とした高リソースなプログラミング言語 (High-Resource Programming Languages, HRPLs) を対象とするものが多く [5, 6], LRPLs における整備は不十分である. 最近では多様な言語に対応したベンチマークが増加している [7, 8] が, 多くは Python をベースとしたベンチマークの翻訳による拡張形であり, LRPLs 固有の文法および型に沿ったベンチマークは以前として少ない. こうしたデータセットの整備に多大な労力と時間を要すること自体が本分野の研究への参入に対する障壁となっている.

LRPLs 分野における既存の研究はクロスリンガル転移または合成データを用いた学習を用いて精度改善を図るものが主である. 合成データによる学習においても LLM が潤沢に有している HRPLs の文法知識, 構造との対応付けを用いた学習手法が取られている [9, 10]. これらは自然言語の翻訳タスク同様に, 同じ意味を持つ HRPL と LRPL のテキストのペアを用いて教師ありファインチューニングを行う. こうした手法は HRPL を介してチューニングを行うため, 前述のベンチマーク同様 HRPL の文法の範疇しかカバーできない.

一方 LLM における強化学習は, 与えられた推論タスクへの LLM 自身の Chain-of-Thought (CoT) [11] などを用いた出力データに対し, 推論結果の正誤に応じて 0/1 の報酬を与える. この枠組みは合成データによる学習を行う LRPLs の研究手法と相性が良く, 並列データを用いずとも LRPLs によるコーディング, 推論能力の向上が期待できる.

本研究は上記の強化学習手法のうち, 環境となる問題データを含む全てを LLM による合成データで完結させる手法について, LRPLs において精度向上に寄与できるか検証するものである.

2　LRPLs 分野における強化学習

2025 年の強化学習手法の発展により LRPLs 分野においても強化学習の導入が行われ始めている. Wu ら [12] は HRPLs から LRPLs への翻訳タスクにおいて, 出力の正誤に応じた 0/1 の報酬を受けての Proximal Policy Optimization (PPO) [13] と, 機能上等価

な中間表現を有するコード群についてそうでないコード群との報酬平均の最大化, および各コード群内での報酬分散の最小化を行う Group Equivalent Policy Optimization の目的関数を組み合わせた最適化を行なっている. Fu ら [14] は低サイズモデルの LRPLs に対し, Unit test の他に静的に構文エラーの有無を精査し, 構文エラーが生じた解答の割合に応じて Unit test の通過報酬との比率を動的に変動させながら PPO を行有ことで文法情報の学習を先に行い, 意味的情報の学習へ徐々に移行することができる.

このように強化学習の導入は LRPLs においても能力向上に寄与している. さらには解の正誤だけでなく文法上の性質や相対的な優劣を踏まえた複合報酬により多目的に学習を進められる. 一方でこれらの強化学習には既存ないし研究者自身がデータセットおよびベンチマークの設計を必要とするという LRPLs 分野が有する課題の解消には至っていない.

3 合成データを活用した強化学習

本節では, 研究の土台となる Zhao ら [15] により提案された Absolute-Zero Reasoner (AZR) について説明する. 図 1 に概観を示す通り, 学習対象である LLM 自身はコード生成タスクの対象となる問題の生成を行い, 問題の生成 (proposer) と問題の解答生成 (solver) 両方のタスクから報酬を受けとり PPO を行う. 初期の solver タスクに使用する問題は報酬を受け取らずに事前に proposer タスクを用いて 256 題の学習に使用可能な問題の生成を行っている. 今回は solver タスクおよび proposer タスクの報酬に応じて以下の目的関数を最適化する.

$$J_{AZR}(\theta) := \max_{\theta} \mathbb{E}_{z\sim p(z)} \Big[\mathbb{E}_{(x,y^*)\sim f_e(\cdot|\tau),\tau\sim\pi_\theta^{propose}(\cdot|z)} \big[r_e^{propose}(\tau,\pi_\theta) + \lambda \mathbb{E}_{y\sim\pi_\theta^{solve}(\cdot|x)} [r_e^{solve}(y,y^*)] \big] \Big] \tag{1}$$

x, y : 生成される問題文および解, y^*はその正答
e : 現在の LLM 推論能力を環境とみなしたもの
f : LLM による生成を関数としてみたもの
z : 参照された過去の生成コード
τ : タスクとして選出された問題文
λ : 定数
各タスクにおける報酬設定は式 2, 3 の通りである.

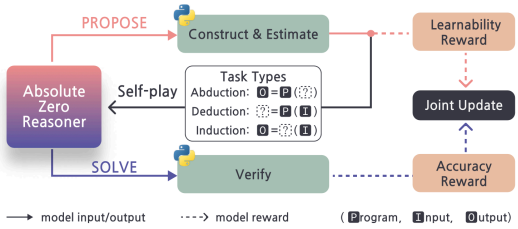

図 1 Absolute-Zero Reasoner

$$r_{solver} = \begin{cases} acc \text{ if passable} \\ -0.5 \text{ if formatted but not passable} \\ -1.0 \text{ else} \end{cases} \tag{2}$$

$$r_{proposer} = \begin{cases} 1 - acc \text{ if } 0 < acc < 1 \\ -0.5 \text{ if } acc = 0 or 1 \\ -1.0 \text{ if not passable} \end{cases} \tag{3}$$

ここで $accuracy$ について, solver については生成された解答一つ一つに 0/1 の値が割り当てられ, proposer については生成された問題については m 回の solver タスクを設計し, その正答率を用いる. 正答率が極端に高い問題は学習する余地が少なく, 逆に正答を生成できない問題においては正例を用いた強化学習が不可能なため, 報酬がそれぞれ負に設定されている. また, 各タスクにはプロンプトを用いて以下の 2 つのフォーマットを指定している.

- <think>〜</think>および<answer>〜<answer>のタグ内にそれぞれ推論過程と解答を記載する.
- “‘{category}〜“‘のタグ内 ({category}内には入出力変数を囲う場合はそれぞれ input/output, コードの場合は対象となる言語名が入る) に各種解答を記載する.

負の報酬についてはこのフォーマットによる則っているかどうか, および取り出した変数およびコードが問題なく実行できるかどうかを基準に分岐する.

また, この学習形態においてはある Python 関数 p および関数に入力変数 i と対応する出力変数 o のトリプレット (p,i,o) に対し以下の通り定義される 3 種類の問題形式のタスクが同時に行われている.

- Deduction: 入力に p,i を与え, o を推論する.
- Abduction: 入力に p,o を与え, i を推論する.
- Induction: 入力に (i,o) のペアを n (= 10) 個を与え, p を推論する.

4 実験

4.1 既存手法の LRPLs への応用

本研究は前述した Absolute-Zero Reasoner を用いて LRPLs による強化学習を実行し, コーディング精度の向上可能性について検証し, さらに HRPLs の挙動と比較することでその推論精度の差を埋めることが可能かを検証する. LRPLs としては Julia, Racket を使用した. パラメータは表 3 に示す. 使用したモデルは Qwen 2.5-3B-Coder [16] である. 最終学習結果について MultiPL-E の HumanEval データおよび MBPP データで評価を行い, HRPL として JavaScript で学習した結果, および参考データとして先行研究の Python による学習結果について HumanEval+および MBPP+ [17] で評価した結果を用いて比較する. また, 本学習手法は学習後のモデルの出力はプロンプトのフォーマット依存性が高く, コード 1 に記載するプロンプトを用いている.

表 1 強化学習前後の言語間精度比較

Language	HumanEval		MBPP	
	Base	+AZR	Base	+AZR
Julia	38.4	50.3$_{+11.9}$	39.5	61.8$_{+22.3}$
Racket	50.1	64.6$_{+14.5}$	39.0	68.3$_{+29.3}$
JavaScript	54.0	70.8$_{+16.8}$	54.2	60.0 $_{+5.8}$
Python (参考値)	67.1	71.3 $_{+4.2}$	65.9	69.0 $_{+3.1}$

結果を表 1 に示す. Python, JavaScript は数%程度の精度向上が見られた一方で Julia, Racket では 10%以上の大幅な精度向上が確認でき, HRPL との精度差が縮んでいることがわかる.

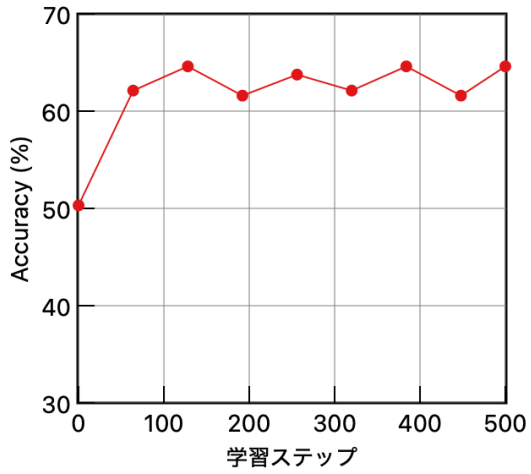

図 2 実験結果: Racket における精度推移

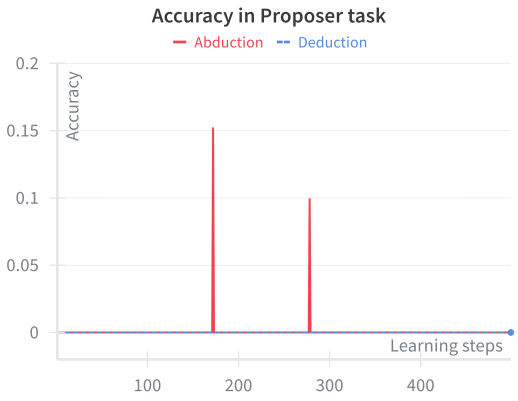

図 3 実験結果: Proposer タスクにおける Accuracy 精度推移

一方図 2 は 64 step 毎の学習結果についてそれぞれ MultiPl-E HumanEval による test を行った精度の推移であるが, 64step 時点で精度は 60 %を上回り, 以後の変動は 2, 3%程度の変動となっており, 早期の段階で学習が収束していることが考えられる. ここで図 3 は, 新たに (p, i, o) トリプレットを生成する Abduction/Deduction proposer タスクの Acccuracy の推移であるが, ほぼ学習全体を通して 0 になっている. すなわち生成したタスクに対して有効な問題に対して正答を生成できず問題生成能力に対して十分な推論能力を得られていないということであり, この報酬は式 3 に従い 0 となる. このため初期シードとなる最も単純な問題に対してのみ solver タスクが適用され, 学習が頭打ちになっていると考えられる.

4.2 学習設計の改修

$$r_{proposer} = \begin{cases} 0.5 + 2 \times acc \times (1 - acc) \text{ if passable} \\ -1.0 \text{ else} \end{cases} \tag{4}$$

続いて前節で確認された学習の早期収束の一因を問題の総数が増加しないことによるものと仮定し, 報酬設計による改善を行う. 報酬設計には 1 ステップごとの推論能力推移はわずかであり, acc が 0 として推定された問題において利用可能問題の設計に応じ問題生成タスクにおける報酬の改修が LRPLs においては必要であると考え, 式 4 の通り報酬を再設計した. 再設計には式 3 の不連続性解消も考慮し, Wang ら [18] を参考に行っている. この報酬設計をもとに再学習を実行した. なお実験の収束を考慮し以降の検証は学習ステップを 256 に留めるものとす

る. 言語は Racket を使用する.

上記を踏まえた学習結果について MultiPL-E で評価した結果を表 2 のうち 2 行目に示す. ベースラインに比べ精度が低下しているだけでなく, 表 1 中のベースモデル性能と比較しても精度の低下が生じている. これについて, 本来ベースモデルはプロンプト 1 を使用した場合フォーマットに関して学習していないため推論精度が記載した値より低下する. 改修後の学習形式では $acc = 0$ の問題に正の報酬を与えたことにより, solver タスクにおいて正報酬を取得できる問題が相対的に減少し, 学習自体が遅延したと考えられる. 図 4 は solver タスクにおける各問題形式の精度推移であるが, いずれも報酬改修後の場合の方が精度向上が緩やかであり, 実際に推論能力の向上が遅れていることがわかる.

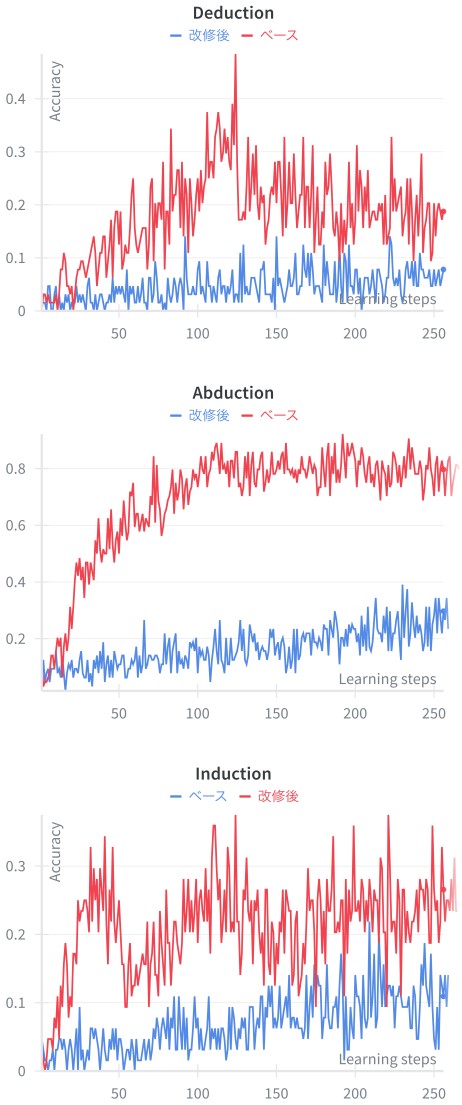

図 4 実験結果: Racket における精度推移

以上の結果を踏まえ, 最初の実験において精度向上が止まった 64 step 目を境に報酬設計を式 3 から式 4 へ移行して学習を進める. これにより変化が横這いになった学習中期以降の変化についてのみ比較を行うことができる.

表 2 報酬設計による学習精度の変化

Reward	HumanEval	MBPP
改修前	64.6	68.3
改修後	41.6	60.5
併用型	66.5	72.5

組み合わせた結果について同様に MultiPL-E で評価した結果を表 2 のうち 2 行目に示す. 表の通りベースラインに比べ大幅ではないもののさらなる精度の向上が確認できた. したがって推論能力が生成される問題に対し不十分な学習初期は proposer タスクにおける報酬による制約を強め, 推移とともにこれを緩和することが効果的であることがわかった. 今回は 64 ステップで報酬を切り替えるという恣意的とも取れる報酬設計を行なったが, これはステップごとの solver タスクの正解率に応じて動的に変動させる設計をとることで報酬の連続性を学習全体で維持しながら proposer の性能を高めることが期待できる.

5 おわりに

本論文では低リソースなプログラミングにおける LLM のコーディング能力向上に向け, 問題データとそれに対する解答データを自己生成し, 報酬を獲得する強化学習フレームワークを使用し, その精度の向上を図った. 実験の結果としては言語間ギャップの縮小には成功したもののさらなる縮小においては学習の頭打ちが生じた. これを踏まえ報酬の改集を行い問題生成タスクにおける正報酬の獲得機会を増加させた. 今後は解答生成においても報酬を見直し, コード生成時の推論内容や文法上エラー別に負報酬を細分化するなど細部の影響について精査する. 同時にゼロリソースの言語へ拡張を行い, 生じうる固有の問題についても解消方法を模索する.

参考文献

[1] DeepSeek-AI. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning, 2025.

[2] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, ¥L ukaszKaiser, Illia Polosukhin. Attention is all you need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, **Advances in Neural Information Processing Systems**, 第 30 巻. Curran Associates, Inc., 2017.

[3] Alessandro Giagnorio, Alberto Martin-Lopez, and Gabriele Bavota. Enhancing code generation for low-resource languages: No silver bullet. In **33rd IEEE/ACM International Conference on Program Comprehension, ICPC@ICSE 2025, Ottawa, ON, Canada, April 27-28, 2025**, pp. 478–488. IEEE, 2025.

[4] Federico Cassano, John Gouwar, Daniel Nguyen, and et al. MultiPL-E: A scalable and polyglot approach to benchmarking neural code generation. **IEEE Transactions of Software Engineering (TSE)**, 2023.

[5] Mark Chen, Jerry Tworek, Heewoo Jun, and et al. Evaluating large language models trained on code. **CoRR**, Vol. abs/2107.03374, , 2021.

[6] Jacob Austin, Augustus Odena, and Maxwell I. Nye andet al. Program synthesis with large language models. **CoRR**, Vol. abs/2108.07732, , 2021.

[7] Anonymous. Multi-LCB: Extending livecodebench to multiple programming languages. In **Submitted to The Fourteenth International Conference on Learning Representations**, 2025. under review.

[8] Aleksander Boruch-Gruszecki, Yangtian Zi, Zixuan Wu, Tejas Oberoi, Carolyn Jane Anderson, Joydeep Biswas, and Arjun Guha. Agnostics: Learning to code in any programming language via reinforcement with a universal learning environment. **CoRR**, Vol. abs/2508.04865, , 2025.

[9] Federico Cassano, John Gouwar, Francesca Lucchetti, and et al. Knowledge transfer from high-resource to low-resource programming languages for code llms. **Proc. ACM Program. Lang.**, Vol. 8, No. OOPSLA2, pp. 677–708, 2024.

[10] Jianbo Lin, Yi Shen, Chuanyi Li, Changan Niu, and Bin Luo. Optcodetrans: Boost llms on low-resource programming language translation. In **2025 IEEE/ACM Second International Conference on AI Foundation Models and Software Engineering (Forge)**, pp. 67–72, 2025.

[11] Mirac Suzgun, Nathan Scales, and et al. Challenging big-bench tasks and whether chain-of-thought can solve them. In **ACL (Findings)**, pp. 13003–13051, 2023.

[12] Haoyuan Wu, Rui Ming, Jilong Gao, Hangyu Zhao, Xueyi Chen, Yikai Yang, Haisheng Zheng, Zhuolun He, and Bei Yu. On-policy optimization with group equivalent preference for multi-programming language understanding. In **The Thirty-ninth Annual Conference on Neural Information Processing Systems**, 2025.

[13] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. **CoRR**, Vol. abs/1707.06347, , 2017.

[14] David Jiahao Fu, Aryan Gupta, Aaron Councilman, David Grove, Yu-Xiong Wang, and Vikram Adve. Slmfix: Leveraging small language models for error fixing with reinforcement learning, 2025.

[15] Andrew Zhao, Yiran Wu, Yang Yue, Tong Wu, Quentin Xu, Yang Yue, Matthieu Lin, Shenzhi Wang, Qingyun Wu, Zilong Zheng, and Gao Huang. Absolute zero: Reinforced self-play reasoning with zero data. In **The Thirty-ninth Annual Conference on Neural Information Processing Systems**, 2025.

[16] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, An Yang, Rui Men, Fei Huang, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. Qwen2.5-coder technical report. **CoRR**, Vol. abs/2409.12186, , 2024.

[17] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. In Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine, editors, **Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023**, 2023.

[18] Yiping Wang, Qing Yang, Zhiyuan Zeng, Liliang Ren, Lucas Liu, Baolin Peng, Hao Cheng, Xuehai He, Kuan Wang, Jianfeng Gao, Weizhu Chen, Shuohang Wang, Simon Shaolei Du, and Yelong Shen. Reinforcement learning for reasoning in large language models with one training example. **arXiv preprint arXiv:2504.20571**, 2025.

A 実験環境詳細

表 3 実験に使用した変数一覧

名称	値
強化学習手法	PPO
オプティマイザ	AdamW
ステップ数	500
シードデータ	256 題
Temperature	1.0/0.0 (train/val)
最大入力長	6144
最大出力長	8096
最大プログラム保持数	16384
訓練用バッチサイズ	64 /タスク × 6 タスク
学習率	1e-6
GradClip	1.0

Listing 1 学習後モデル評価プロンプト

```
# Task: You will be given a question (problem
    specification) and will generate a correct
    {Language} program that matches the
    specification and passes all tests. 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>. Your final answer
    should be wrapped in ```{language}``` tags.
Question: {question}

You will use the following starter code to write
     the solution to the problem and enclose
    your code within delimiters.
```{language}
{comment_out} YOUR CODE HERE
```

Assistant: <think>
```

- Language: プログラミング言語名
- language: プログラミング言語名 (略称)
- question: 問題データ
- comment_out: コメントアウト記号

B 学習環境における入出力例

Listing 2 Induction solver タスクにおける入力例

```
# Task:
; Instruction to solve the problem
# Code Requirements:
; Requrements here

# Input and Output Pairs:
```input
'()
```
```output
'(() . 0)
```
```input
'(1)
```
```output
'((1) . 1)
```
...

Message:
```message
;Additional instruction unique to the problem
    here
```

Example Output:
#lang racket

(define (f a)
    a)

Name your entry function `f()`!!!
"""
```

Listing 3 Induction solver タスクにおける出力例

```
<answer>
#lang racket

(define (f lst)
    (let ([squares (map (lambda (x) (* x x)) lst
        )])
        (let ([sum (apply + squares)])
            (cons squares sum)
        )
    )
)
</answer>
<think>
The function seems to take a list of numbers and
     squares each number, then sums them up, and
     stores the original list along with the sum
     in a list. It also uses . to concatenate
    the list made by squaring the numbers with
    the sum.
</think>
```