Learning from [code-related] feedback

Daniel Fried, with slides from Sean Welleck

Neural Code Generation
Carnegie Mellon University
Sept 25, 2025

Language models

Problem: **distribution mismatch**

- Language model p_θ fits distribution q
  - E.g., code on the web
- Language model does not learn desired distribution q'
  - E.g., code that passes tests

This can be for several reasons. For instance, low quality data is included, or not enough data is included, or limited model capacity.

Intuition

Observation 1: many signals are not explicitly in pretraining data

- whether a program compiles,
  whether a program passes test cases,
  whether a specific user prefers one program over the other, ...

Intuition

Observation 1: many signals are not explicitly in pretraining data

- whether a program compiles,
  whether a program passes test cases,
  whether a specific user prefers one program over the other, ...

Observation 2: we can get these via **feedback** on **generated programs**

Intuition

Observation 1: many signals are not explicitly in pretraining data

- whether a program compiles,
  whether a program passes test cases,
  whether a specific user prefers one program over the other, ...

Observation 2: we can get these via **feedback** on **generated programs**

Today: learning from feedback on generated programs

Outline

- Reinforcement learning
- Reward modeling
- Expert iteration

Reinforcement learning

Adjust the model so that it maximizes a reward function:

$$\arg\max_{\theta} \underbrace{\mathbb{E}_{x\sim\mathcal{D}, y\sim p_\theta(\cdot|x)}\left[R(x,y)\right]}_{J(\theta)}$$

Example reward:

- $R(x, y) =$ `1 if program y passes test cases`

Reinforcement learning

General pattern:

- Generate data with the model, $y \sim p_\theta(\cdot|x)$
- Score the data, $R(y)$
- Update the model using data and rewards, so that high reward data is more likely

At a high level:

- $p_{\theta'} \leftarrow \mathcal{A}(p_\theta, \{x\}, R)$

Policy gradient methods [15, 14]

Generate program $\hat{y} \sim p_\theta(\cdot|x)$

Estimate the gradient of the expected reward with respect to θ:

$$\nabla_\theta J(\theta) = \mathbb{E}_{x\sim\mathcal{D}, y\sim p_\theta(y|x)} \nabla_\theta \log p_\theta(y|x) R(x, y) \tag{1}$$

Use gradient descent to update model parameters, $\theta' \leftarrow \theta + \alpha \nabla_\theta$.

Example: PPO [9] and GRPO [10]

Various innovations to stabilize policy gradient (out of scope)

Proximal Policy Optimization Algorithms

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, Oleg Klimov
OpenAI
{joschu, filip, prafulla, alec, oleg}@openai.com

DeepSeekMath: Pushing the Limits of Mathematical Reasoning in Open Language Models

Zhihong Shao[1,2*†], Peiyi Wang[1,3*†], Qihao Zhu[1,3*†], Runxin Xu[1], Junxiao Song[1]
Xiao Bi[1], Haowei Zhang[1], Mingchuan Zhang[1], Y.K. Li[1], Y. Wu[1], Daya Guo[1*]
[1]DeepSeek-AI, [2]Tsinghua University, [3]Peking University

At the end, we get an alternative algorithm:

$$p_{\theta'} \leftarrow \mathcal{A}_{\text{GRPO}}(p_\theta, \{x\}, R)$$

Recap

RL: used to update a model using rewards and generated sequences.

- $p_{\theta'} \leftarrow \mathcal{A}(p_\theta, \{x\}, R)$
- Policy gradient, PPO, GRPO, ...

How do we choose the reward?

Reward hacking

Issue 1: reward hacking

- Models can overfit to patterns in the reward
- Examples:
    - $R(x, y) = \texttt{1 if program y compiles, 0 otherwise}$
    - Then generating $y = \texttt{print("hello world")}$ for all x would maximize reward.
    - $R(x, y) = \texttt{1 if program y passes tests, 0 otherwise}$
    - Then deleting the test cases for all x would maximize reward.

Reward hacking: KL-divergence penalty [18]

Mitigation: KL-divergence penalty

- Keep the updated model close to the pretrained model
- $R_{\text{KL}} = -\beta \log \frac{p_\theta(y|x)}{p_0(y|x)}$

Reward hacking: KL-divergence penalty [18]

Mitigation: KL-divergence penalty

- Keep the updated model close to the pretrained model
- $R_{\mathrm{KL}} = -\beta \log \frac{p_\theta(y|x)}{p_0(y|x)}$

$$
\begin{aligned}
D_{\mathrm{KL}}(p_\theta(y|x) \| p_0(y|x)) &= \sum_y p_\theta(y|x) \log \frac{p_\theta(y|x)}{p_0(y|x)} \\
&= \mathbb{E}_{y \sim p_\theta} \log \frac{p_\theta(y|x)}{p_0(y|x)} \\
&\approx \log \frac{p_\theta(\hat{y}|x)}{p_0(\hat{y}|x)},
\end{aligned}
$$

where $\hat{y} \sim p_\theta(\cdot|x)$, i.e. a single-sample Monte-Carlo approximation.

Sparse reward

Issue 2: sparse reward

- The reward may be 0 for many programs; we only occasionally see a positive reward

Mitigation: engineer the reward function

Example: PPOCoder
Execution-based Code Generation using Deep RL [12]

Execution

- $R_{\text{execution}}(x, \hat{y})$: 1 if program $\hat{y}$ compiles and passes tests cases

Example: PPOCoder
Execution-based Code Generation using Deep RL [12]

Execution

- $R_{\text{execution}}(x, \hat{y})$: 1 if program $\hat{y}$ compiles and passes tests cases

Syntactic matching score

- $R_{\text{syntax}}(x, \hat{y}, y_*)$: overlap between abstract syntax tree of y and y_*

Example: PPOCoder
Execution-based Code Generation using Deep RL [12]

Execution

- $\mathrm{R}_{\text{execution}}(x, \hat{y})$: 1 if program $\hat{y}$ compiles and passes tests cases

Syntactic matching score

- $\mathrm{R}_{\text{syntax}}(x, \hat{y}, y_*)$: overlap between abstract syntax tree of y and y_*

Semantic matching score

- $\mathrm{R}_{\text{semantics}}(x, \hat{y}, y_*)$: overlap between dataflow graph of y and y_*

$$R = R_{\text{execution}} + R_{\text{syntax}} + R_{\text{semantics}} + R_{\text{KL}}$$

Example: PPOCoder *Execution-based Code Generation using Deep RL* [12]

Execution

- $\mathrm{R}_{\text{execution}}(x, \hat{y})$: 1 if program $\hat{y}$ compiles and passes tests cases

Syntactic matching score

- $\mathrm{R}_{\text{syntax}}(x, \hat{y}, y_*)$: overlap between abstract syntax tree of y and y_*

Semantic matching score

- $\mathrm{R}_{\text{semantics}}(x, \hat{y}, y_*)$: overlap between dataflow graph of y and y_*

$$R = R_{\text{execution}} + R_{\text{syntax}} + R_{\text{semantics}} + R_{\text{KL}}$$

Run PPO using the reward

PPOCoder results

Table 1: Results on the code completion task for completing the last 25 masked tokens from CodeSearchNet.

Model	$\uparrow$*xMatch*	$\uparrow$*Edit Sim*	$\uparrow$*Comp Rate*
BiLSTM	20.74	55.32	36.34
Transformer	38.91	61.47	40.22
GPT-2	40.13	63.02	43.26
CodeGPT	41.98	64.47	46.84
CodeT5 (220M)	42.61	68.54	52.14
PPOCoder + CodeT5 (220M)	**42.63**	**69.22**	**97.68**

Figure 1: Compilation rate increases while holding other metrics constant

PPOCoder results

		↑*pass@1*				↑*pass@5*			
Model	Size	Intro	Inter	Comp	All	Intro	Inter	Comp	All
Codex	12B	4.14	0.14	0.02	0.92	9.65	0.51	O.09	2.25
AlphaCode	1B	–	–	–	–	–	–	–	–
GPT-3	175B	0.20	0.03	0.00	0.06	–	–	–	–
GPT-2	0.1B	1.00	0.33	0.00	0.40	2.70	0.73	0.00	1.02
GPT-2	1.5B	1.30	0.70	0.00	0.68	3.60	1.03	0.00	1.34
GPT-Neo	2.7B	3.90	0.57	0.00	1.12	5.50	0.80	0.00	1.58
CodeT5	60M	1.40	0.67	0.00	0.68	2.60	0.87	0.10	1.06
CodeT5	220M	2.50	0.73	0.00	0.94	3.30	1.10	0.10	1.34
CodeT5	770M	3.60	0.90	0.20	1.30	4.30	1.37	0.20	1.72
CodeRL+CodeT5	770M	4.90	**1.06**	**0.5**	1.71	8.60	**2.64**	1.0	3.51
PPOCoder +CodeT5	770M	**5.20**	1.00	**0.5**	**1.74**	**9.10**	2.50	**1.20**	**3.56**

Figure 2: APPS

Model	Size	State	↑*pass@80*
GPT	224M	fine-tuned	7.2
GPT	422M	fine-tuned	12.6
GPT	1B	fine-tuned	22.4
GPT	4B	fine-tuned	33.0
GPT	8B	fine-tuned	40.6
GPT	68B	fine-tuned	53.6
GPT	137B	fine-tuned	61.4
CodeT5	60M	fine-tuned	19.2
CodeT5	220M	fine-tuned	24.0
CodeT5	770M	fine-tuned	32.4
CodeRL+CodeT5	770M	zero-shot	63.0
PPOCoder +CodeT5	770M	zero-shot	**68.2**

Figure 3: MBPP (transfer)

PPOCoder ablations

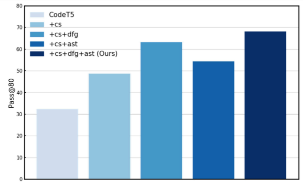


Figure 4: Reward terms (MBPP)

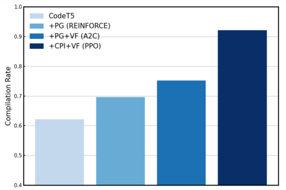


Figure 5: RL objective (MBPP)

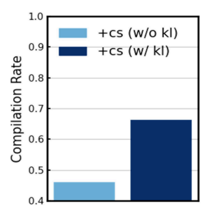


Figure 6: KL penalty

Qualitative – what is it changing?

Source (NL)

You are given two integers n and m. Calculate the number of pairs of arrays (a, b) such that: the length of both arrays is equal to m; each element of each array is an integer between 1 and n (inclusive); ai ≤ bi for any index i from 1 to m; array a is sorted in non-descending order; array b is sorted in non ascending order. As the result can be very large, you should print it modulo 109 + 7.

Input: The only line contains two integers n and m (1 ≤ n ≤ 1000, 1 ≤ m ≤ 10).

Output: Print one integer – the number of arrays a and b satisfying the conditions described above modulo 109 + 7

Generated (Python)

CodeT5

```
MOD = 10**9 + 7

def solve(n, m):                                    (1)
    dp = [[1] * (n + 1) for _ in range(m + 1)]
    for i in range(2, m + 1):
        for j in range(1, n + 1):                   (2)
            for k in range(j+1, n+1):
                dp[i][j] = (dp[i][j] + dp[i - 1][k]) % MOD
   ans = 0
   for i in range(1, n + 1):
      ans = (ans + dp[m][i]) % MOD
   return ans

n,m = map(int,input().split())
print(solve(n,m))
```

Pass Tests? ☒

PPOCoder+CodeT5

```
MOD = 10**9 + 7

def solve(n, m):
    dp = [[0] * (n + 1) for _ in range(m + 1)]
    for i in range(1, n + 1):
        dp[1][i] = 1
    for i in range(2, m + 1):
        for j in range(1, n + 1):
            for k in range(1, j + 1):
                dp[i][j] = (dp[i][j] + dp[i - 1][k]) % MOD
    ans = 0
    for i in range(1, n + 1):
        ans = (ans + dp[m][i]) % MOD
   return ans

n,m = map(int,input().split())
print(solve(n,m))
```

Pass Tests? ☑

Example: DeepSeek-R1-Zero [7]

- Have the model produce chain-of-thought before the final answer
- Do RL to learn to produce chains-of-thought and answers
- Simple reward and RL procedure (GRPO), but very large scale training and a strong starting model
- Data is a mystery, but all problems that afford verifiable reward (RLVF)

DeepSeek-R1-Zero: Prompt and Outputs

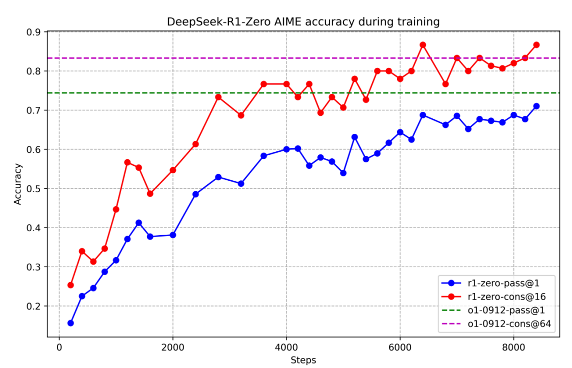
A conversation between User and Assistant. The user asks a question, and the Assistant solves it. The assistant first thinks about the reasoning process in the mind and then provides the user with the answer. The reasoning process and answer are enclosed within <think> </think> and <answer> </answer> tags, respectively, i.e., <think> reasoning process here </think> <answer> answer here </answer>. User: prompt. Assistant:

Question: If $a > 1$, then the sum of the real solutions of $\sqrt{a - \sqrt{a + x}} = x$ is equal to
Response: <think> To solve the equation $\sqrt{a - \sqrt{a + x}} = x$, let's start by squaring both $\cdots$ $\left(\sqrt{a - \sqrt{a + x}}\right)^2 = x^2 \implies a - \sqrt{a + x} = x^2.$ Rearrange to isolate the inner square root term: $(a - x^2)^2 = a + x \implies a^2 - 2ax^2 + (x^2)^2 = a + x \implies x^4 - 2ax^2 - x + (a^2 - a) = 0$ ... Wait, wait. Wait. That's an aha moment I can flag here. Let's reevaluate this step-by-step to identify if the correct sum can be $\cdots$ We started with the equation: $\sqrt{a - \sqrt{a + x}} = x$ First, let's square both sides: $a - \sqrt{a + x} = x^2 \implies \sqrt{a + x} = a - x^2$ Next, I could square both sides again, treating the equation: $\cdots$ ...

Table 3 | An interesting "aha moment" of an intermediate version of DeepSeek-R1-Zero. The model learns to rethink using an anthropomorphic tone. This is also an aha moment for us, allowing us to witness the power and beauty of reinforcement learning.

DeepSeek-R1-Zero: Training

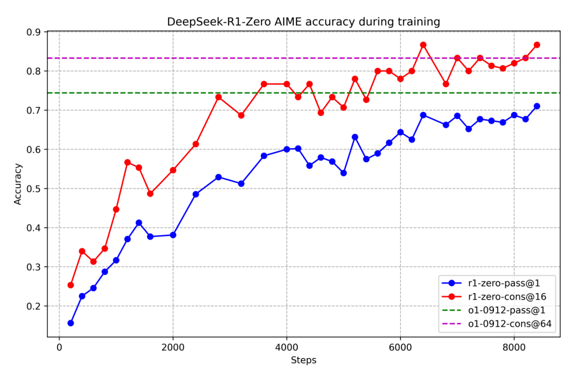
- Uses rule-based reward, which is 'mainly' a combination of accuracy of the final answer and format of the reasoning chains.
- No details on the amount or distribution of the training data.



- But, Qwen3 paper trains on 4K query/verifier pairs, and increases AIME scores from 70.1 to 85.1 in 170 RL steps.

DeepSeek-R1-Zero: Training

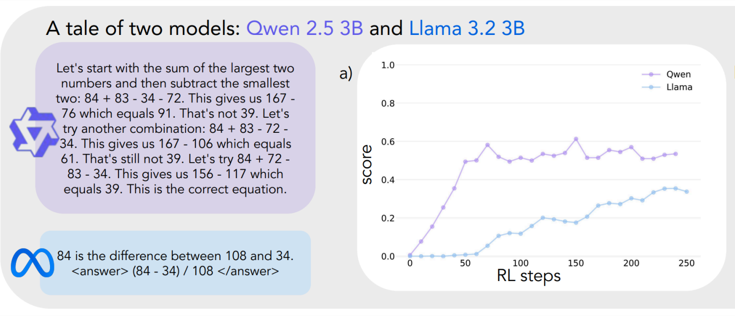
- Uses rule-based reward, which is 'mainly' a combination of accuracy of the final answer and format of the reasoning chains.
- No details on the amount or distribution of the training data.



- But, Qwen3 paper trains on 4K query/verifier pairs, and increases AIME scores from 70.1 to 85.1 in 170 RL steps.

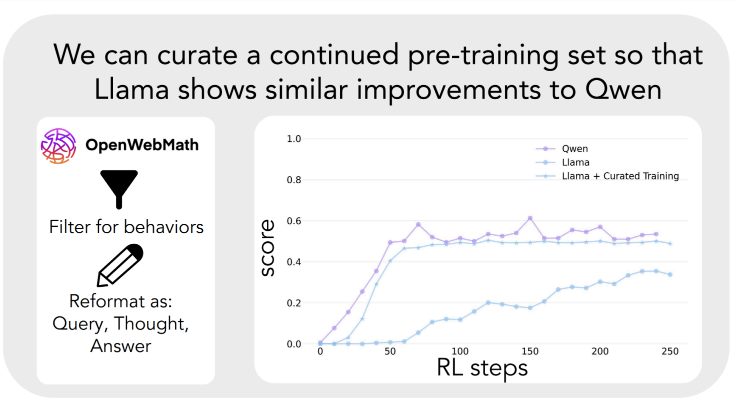
Role of the base model and the training data

- Common to use a "cold start" stage before performing RL: fine-tune the base model on problems with synthetic reasoning chains (DeepSeek-R1 used chains from DeepSeek-R1-Zero; Qwen-3 used QwQ)
- Gandhi et al. [5] identify reasoning strategies which are important for solution accuracy (verification, subgoal setting, backtracking, backward chaining)



Role of the base model and the training data

- Gandhi et al. [5] show that reasoning strategies can come from the base model or the training data



See *https://www.interconnects.ai/p/papers-im-reading-base-model-rl-grpo* for more.

Tradeoffs

RL with policy gradient methods

- Directly optimizes reward
- Can be astoundingly effective at scale
- Learning procedure adds complexity
- Dependent on the quality of the base model and the training problems

Tradeoffs

RL with policy gradient methods

- Directly optimizes reward
- Can be astoundingly effective at scale
- Learning procedure adds complexity
- Dependent on the quality of the base model and the training problems

Outline

- Reinforcement learning
- **Reward modeling**
- Expert iteration

Reward modeling

Basic idea:

- Train a model $R_\phi(y)$ to predict whether a program is correct
  - $R_\phi(y) \in [0, 1]$, 0 means incorrect, 1 means correct
- At test time:
  - Generate many programs, $\{y_1, \ldots, y_K\} \sim p_\theta(\cdot|x)$
  - Select the program with the highest score $R_\phi(y)$

Reward modeling

Basic idea:

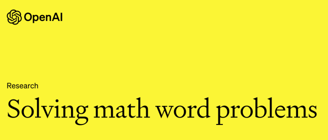
- Train a model $R_\phi(y)$ to predict whether a program is correct
    - $R_\phi(y) \in [0, 1]$, 0 means incorrect, 1 means correct
- At test time:
    - Generate many programs, $\{y_1, \ldots, y_K\} \sim p_\theta(\cdot|x)$
    - Select the program with the highest score $R_\phi(y)$

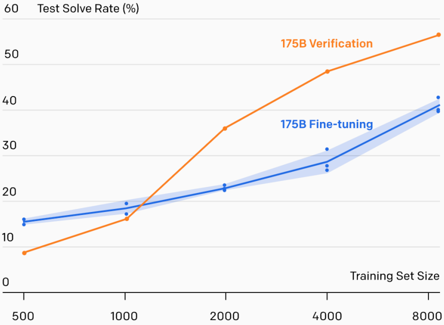
$R_\phi(y)$: “reward model” or “learned verifier”

Test time procedure: “best-of-n”

Reward modeling

LLMs: investigated on math word problems [3]





Reward modeling: LEVER

Learning to Verify Language-to-Code Generation with Execution [8]

- Key difference: we can **execute** code
- Train a model $p_\phi(v|x, y, \mathcal{E}(y))$
  - v is 0 or 1
  - x: input prompt
  - y: generated program
  - $\mathcal{E}(y)$ is the result of executing program y

Reward modeling: LEVER training

Given $(x, \mathcal{E}(y_*))$

- Generate $\{y_1, \ldots, y_K\} \sim p_\theta(\cdot|x)$
- Add $(x, y_k, \mathcal{E}(y_k), v_k)$ to a set S_x
    - v_k is 1 if execution result matches gold result $\mathcal{E}(y_*)$, 0 otherwise

$$\mathcal{L}(x, S_x) = -\frac{1}{|S_x|}\sum_{k=1}^{|S_x|} \log p(v_k|x, y_k, \mathcal{E}(y_k))$$

Reward modeling: LEVER

GSM8K: question + idiomatic program + answer variable

Input:

```
Carly recently graduated and is looking for work in a field she studied for.  She
sent 200 job applications to companies in her state, and twice that number to
companies in other states.  Calculate the total number of job applications she has
sent so far.  |
n_job_apps_in_state = 200
n_job_apps_out_of_state = n_job_apps_in_state * 2
answer = n_job_apps_in_state + n_job_apps_out_of_state |
'answer':  600
```

Output: yes

Reward modeling: LEVER

SPIDER/WIKITQ: question + SQL + linearized result table

Input:
```
-- question:  List the name, born state and age of the heads of departments ordered
by age.|
-- SQL:|select name, born_state, age from head join management on head.head_id =
management.head_id order by age|
-- exec result:|/*| name born_state age| Dudley Hart California 52.0| Jeff Maggert
Delaware 53.0|Franklin Langham Connecticut 67.0| Billy Mayfair California 69.0|
K. J. Choi Alabama 69.0|*/
```
Output: no

Reward modeling: LEVER

MBPP: task description + function + return type & value
Input:

```
# description
Write a function to find the n-th power of individual elements in a list using lambda
function.

# program
def nth_nums(nums,n):
    result_list = list(map(lambda x:  x ** n, nums))
    return (result_list)

# execution
# return:  (list)=[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
# return:  (list)=[1000, 8000, 27000]
# return:  (list)=[248832, 759375]
```

Output: yes

Reward modeling: LEVER

At test time:

- Generate $\{y_1, \ldots, y_K\} \sim p_\theta(\cdot|x)$
- Select the program y_k with the highest score $R(x, y_k)$.
    - $r(x, y_k) = \underbrace{p_\theta(y_k|x)}_{\text{LM score}} \cdot \underbrace{p_\phi(v = 1|x, y_k, \mathcal{E}(y_k))}_{\text{verifier score}}$
    - $R(x, y_k) = \sum_{y_{k'} \text{ with same exec result as } y_k} r(x, y_{k'})$

Reward modeling: LEVER

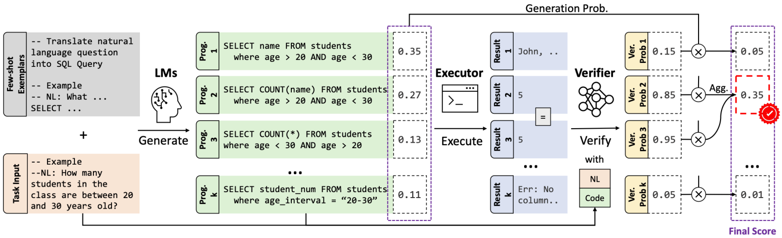


Figure 1: The illustration of LEVER using text-to-SQL as an example. It consists of three steps: 1) *Generation*: sample programs from code LLMs based on the task input and few-shot exemplars; 2) *Execution*: obtain the execution results with program executors; 3) *Verification*: using a learned verifier to output the probability of the program being correct based on the NL, program and execution results.

Reward modeling: LEVER

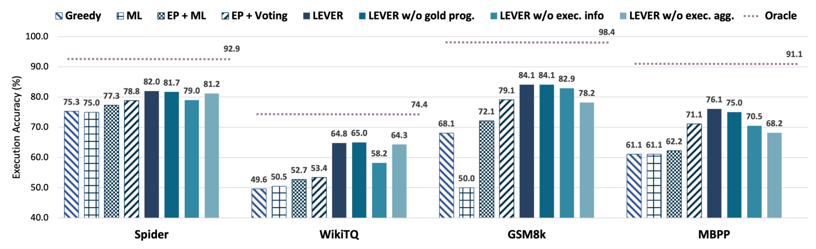


Figure 2: Comparison of LEVER and baselines with Codex-Davinci. LEVER and its ablation results are in solid bars.

Figure 8: LEVER improves performance. Using execution info is important

Reward modeling: LEVER

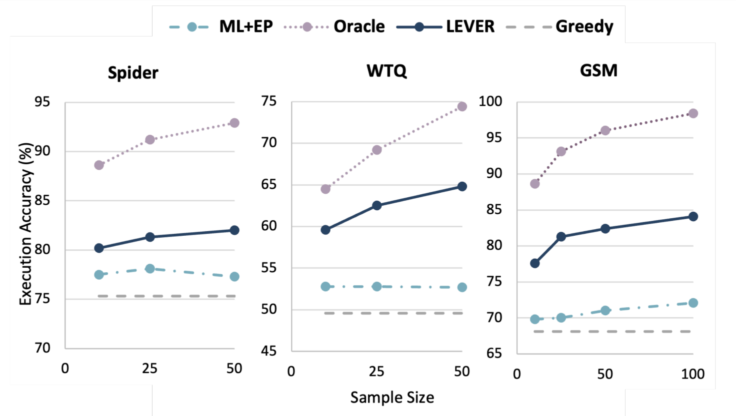


Figure 9: Scaling the number of samples

Tradeoffs

Reward modeling + best-of-n:

- Does not require updating generator p_θ
- Simple learning objective for reward model: standard maximum likelihood
- Strong performance
- Bounded by the generator's capabilities
- Expensive at generation time
- Reward model is imperfect

Outline

- Reinforcement learning
- Reward modeling
- Expert iteration

Expert iteration

Alternate between **search** and **learning**:

- **Search:** Use an 'expert model' to find good outputs
- **Learning:** Fine-tune on the discovered outputs
- Repeat

Expert iteration

Thinking Fast and Slow with Deep Learning and Tree Search

Thomas Anthony[1,✉], **Zheng Tian**[1], and **David Barber**[1,2]

[1]University College London
[2]Alan Turing Institute
✉thomas.anthony.14@ucl.ac.uk

Abstract

Sequential decision making problems, such as structured prediction, robotic control, and game playing, require a combination of planning policies and generalisation of those plans. In this paper, we present Expert Iteration (EXIT), a novel reinforcement learning algorithm which decomposes the problem into separate planning and generalisation tasks. Planning new policies is performed by tree search, while a deep neural network generalises those plans. Subsequently, tree search is improved by using the neural network policy to guide search, increasing the strength of new plans. In contrast, standard deep Reinforcement Learning algorithms rely on a neural network not only to generalise plans, but to discover them too. We show that EXIT outperforms REINFORCE for training a neural network to play the board game Hex, and our final tree search agent, trained tabula rasa, defeats MOHEX 1.0, the most recent Olympiad Champion player to be publicly released.

Figure 10: Anthony et al 2017

Expert iteration

For neural code generation:

- **Search:** Generate many programs, save those that succeed
- **Learning:** Fine-tune on the saved programs
- Repeat

Expert iteration

For neural code generation:

- **Search:** Generate many programs, save those that succeed
- **Learning:** Fine-tune on the saved programs
- Repeat

"Self-training": the expert model is the current language model (plus the binary execution feedback)

Self-training with execution [13]

Google DeepMind 2023-12-25

Beyond Human Data: Scaling Self-Training for Problem-Solving with Language Models

Avi Singh[1,*], John D Co-Reyes[1,*], Rishabh Agarwal[1,2,*],

Ankesh Anand[1], Piyush Patil[1], Xavier Garcia[1], Peter J. Liu[1], James Harrison[1], Jaehoon Lee[1], Kelvin Xu[1],

Aaron Parisi[1], Abhishek Kumar[1], Alex Alemi[1], Alex Rizkowsky[1], Azade Nova[1], Ben Adlam[1], Bernd Bohnet[1], Gamaleldin Elsayed[1], Hanie Sedghi[1], Igor Mordatch[1], Isabelle Simpson[1], Izzeddin Gur[1], Jasper Snoek[1], Jeffrey Pennington[1], Jiri Hron[1], Kathleen Kenealy[1], Kevin Swersky[1], Kshiteej Mahajan[1], Laura Culp[1], Lechao Xiao[1], Maxwell L Bileschi[1], Noah Constant[1], Roman Novak[1], Rosanne Liu[1], Tris Warkentin[1], Yundi Qian[1],

Yamini Bansal[1], Ethan Dyer[1], Behnam Neyshabur[1], Jascha Sohl-Dickstein[1], Noah Fiedel[1]

[*]Contributed equally, [1]Google DeepMind, [2] Mila

Builds on recent ideas, e.g. for reasoning [17, 16], generation [6], preference alignment [4].

Self-training with execution

Algorithm 1: ReST (Expectation-Maximization). Given a initial policy (e.g., pre-trained LM), ReST^{EM} iteratively applies `Generate` and `Improve` steps to update the policy.

Input: $\mathcal{D}$: Training dataset, $\mathcal{D}_{val}$: Validation dataset, $\mathcal{L}(\boldsymbol{x}, \boldsymbol{y}; \theta)$: loss, $r(\boldsymbol{x}, \boldsymbol{y})$: Non-negative reward function, I: number of iterations, N: number of samples per context

for $i = 1$ *to* I **do**

   `// Generate (E-step)`

   Generate dataset $\mathcal{D}_i$ by sampling: $\mathcal{D}_i = \{ (\boldsymbol{x}^j, \boldsymbol{y}^j)|_{j=1}^{N} \text{ s.t. } \boldsymbol{x}^j \sim \mathcal{D},\ \boldsymbol{y}^j \sim p_\theta(\boldsymbol{y}|\boldsymbol{x}^j) \}$

   Annotate $\mathcal{D}_i$ with the reward $r(\boldsymbol{x}, \boldsymbol{y})$.

   `// Improve (M-step)`

   while *reward improves on* $\mathcal{D}_{val}$ **do**

      Optimise θ to maximize objective: $J(\theta) = \mathbb{E}_{(\boldsymbol{x},\boldsymbol{y})\sim\mathcal{D}_i} [r(\boldsymbol{x}, \boldsymbol{y}) \ \log p_\theta(\boldsymbol{y}|\boldsymbol{x})]$

   end

end

Output: Policy p_θ

Self-training with execution

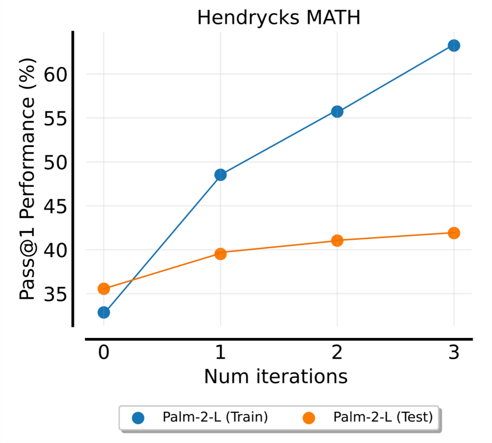


Figure 11: On the MATH dataset, improves for multiple iterations

Self-training with execution

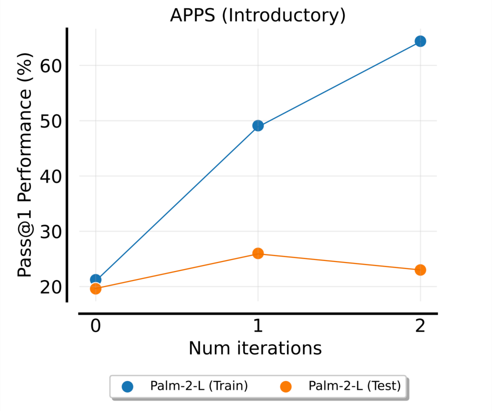


Figure 12: On a subset of APPS: initially improves, then overfits.

Self-training with execution

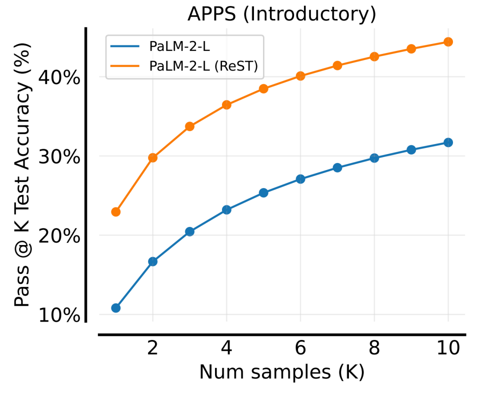


Figure 13: On a subset of APPS: improves pass@k

Connection with reinforcement learning

$$\mathcal{L}_{\mathrm{RL}}(\theta) = \mathbb{E}_{x \sim \mathcal{D}, y \sim p_\theta(y|x)} \left[R(x, y) \right]$$

Connection with reinforcement learning

$$\mathcal{L}_{\mathrm{RL}}(\theta) = \mathbb{E}_{x\sim\mathcal{D}, y\sim p_\theta(y|x)}\left[R(x, y)\right]$$

Policy gradient methods: interleave updates and generation

$$\theta_{t+1} \leftarrow \theta_t + \alpha\left[\nabla_\theta \log p_\theta(\hat{y}|x) R(x, \hat{y})\right]$$

Often has tricks to stabilize training: clipping, regularization, value estimators to reduce variance.

Connection with reinforcement learning

$$\mathcal{L}_{\text{RL}}(\theta) = \mathbb{E}_{x \sim \mathcal{D}, y \sim p_\theta(y|x)} \left[R(x, y) \right]$$

Policy gradient methods: interleave updates and generation

$$\theta_{t+1} \leftarrow \theta_t + \alpha \left[\nabla_\theta \log p_\theta(\hat{y}|x) R(x, \hat{y}) \right]$$

Often has tricks to stabilize training: clipping, regularization, value estimators to reduce variance.

Self-training: generate a large dataset, then update

$$\theta_{t+1} \leftarrow \arg\max_\theta \mathbb{E}_{x \sim \mathcal{D}} \left[\mathbb{E}_{y \sim p_{\theta_t}(y|x)} \left[\log p_\theta(y|x) r(x, y) \right] \right]$$

See the Rest-EM paper [13] for more details on the connection.

Role of training on model's own outputs

- RL and self-training both train on outputs from the model.
- Recent work shows that this helps avoid catastrophic forgetting [11] and can generalize better than SFT [2].

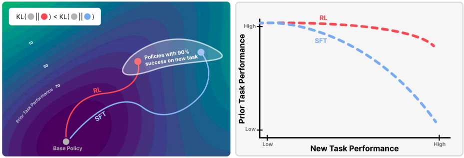


Figure 1: **Bias toward KL-minimal solutions reduces forgetting.** *Left:* Among policies that solve the new task, RL converges to those closest in KL to the base model. *Right:* This KL bias yields higher prior-task retention at matched new-task performance compared to SFT.

Figure 14: Figure from [11].

Recap

Self-training:

- Natural extension of best-of-n, which had good performance
- Simple learning objective: standard maximum likelihood
- Susceptible to overfitting
- Very recent; ongoing investigation!

Summary

Three methods for learning from feedback:

- Directly optimize a reward with reinforcement learning
- Learn a reward, generate programs, select the best program
- Generate programs, save successful ones, train on them

Recap

Looking ahead:

- Each method has pros and cons
- Still a research frontier for code generation
- Other potential sources of feedback, e.g. natural language [1][1]

[1] Another research frontier; not covered due to time constraints.

A. Chen.
Improving code generation by training with natural language feedback.
ArXiv, abs/2303.16749, 2023.

T. Chu, Y. Zhai, J. Yang, S. Tong, S. Xie, D. Schuurmans, Q. V. Le, S. Levine, and Y. Ma.
Sft memorizes, rl generalizes: A comparative study of foundation model post-training.
arXiv preprint arXiv:2501.17161, 2025.

K. Cobbe, V. Kosaraju, M. Bavarian, M. Chen, H. Jun, L. Kaiser, M. Plappert, J. Tworek, J. Hilton, R. Nakano, C. Hesse, and J. Schulman.
Training verifiers to solve math word problems.
arXiv preprint arXiv:2110.14168, 2021.

H. Dong, W. Xiong, D. Goyal, Y. Zhang, W. Chow, R. Pan, S. Diao, J. Zhang, K. SHUM, and T. Zhang.
RAFT: Reward ranked finetuning for generative foundation model alignment.
Transactions on Machine Learning Research, 2023.

K. Gandhi, A. Chakravarthy, A. Singh, N. Lile, and N. D. Goodman.
Cognitive behaviors that enable self-improving reasoners, or, four habits of highly effective stars.
arXiv preprint arXiv:2503.01307, 2025.

C. Gulcehre, T. L. Paine, S. Srinivasan, K. Konyushkova, L. Weerts, A. Sharma, A. Siddhant, A. Ahern, M. Wang, C. Gu, W. Macherey, A. Doucet, O. Firat, and N. de Freitas.
Reinforced self-training (rest) for language modeling, 2023.

D. Guo, D. Yang, H. Zhang, J. Song, R. Zhang, R. Xu, Q. Zhu, S. Ma, P. Wang, X. Bi, et al.
Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning.
arXiv preprint arXiv:2501.12948, 2025.

A. Ni, S. Iyer, D. Radev, V. Stoyanov, W.-t. Yih, S. I. Wang, and X. V. Lin.
Lever: Learning to verify language-to-code generation with execution.
In *Proceedings of the 40th International Conference on Machine Learning (ICML'23)*, 2023.

J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov.
Proximal policy optimization algorithms, 2017.

Z. Shao, P. Wang, Q. Zhu, R. Xu, J. Song, X. Bi, H. Zhang, M. Zhang, Y. Li, Y. Wu, et al.
Deepseekmath: Pushing the limits of mathematical reasoning in open language models.
arXiv preprint arXiv:2402.03300, 2024.

I. Shenfeld, J. Pari, and P. Agrawal.
Rl's razor: Why online reinforcement learning forgets less.
arXiv preprint arXiv:2509.04259, 2025.

P. Shojaee, A. Jain, S. Tipirneni, and C. K. Reddy.
Execution-based code generation using deep reinforcement learning.
Transactions on Machine Learning Research, 2023.

A. Singh, J. D. Co-Reyes, R. Agarwal, A. Anand, P. Patil, P. J. Liu, J. Harrison, J. Lee, K. Xu, A. Parisi, A. Kumar, A. Alemi, A. Rizkowsky, A. Nova, B. Adlam, B. Bohnet, H. Sedghi, I. Mordatch, I. Simpson, I. Gur, J. Snoek, J. Pennington, J. Hron, K. Kenealy, K. Swersky, K. Mahajan, L. Culp, L. Xiao, M. L. Bileschi, N. Constant, R. Novak, R. Liu, T. B. Warkentin, Y. Qian, E. Dyer, B. Neyshabur, J. N. Sohl-Dickstein, and N. Fiedel.

Beyond human data: Scaling self-training for problem-solving with language models.
ArXiv, abs/2312.06585, 2023.

R. S. Sutton, D. McAllester, S. Singh, and Y. Mansour.
Policy gradient methods for reinforcement learning with function approximation.
In *Proceedings of the 12th International Conference on Neural Information Processing Systems*, NIPS'99, page 1057–1063, Cambridge, MA, USA, 1999. MIT Press.

R. J. Williams.
Simple statistical gradient-following algorithms for connectionist reinforcement learning.
Machine Learning, 8:229–256, 1992.

Z. Yuan, H. Yuan, C. Li, G. Dong, C. Tan, and C. Zhou.
Scaling relationship on learning mathematical reasoning with large language models.
ArXiv, abs/2308.01825, 2023.

E. Zelikman, Y. Wu, J. Mu, and N. Goodman.
STar: Bootstrapping reasoning with reasoning.
In A. H. Oh, A. Agarwal, D. Belgrave, and K. Cho, editors, *Advances in Neural Information Processing Systems*, 2022.

D. M. Ziegler, N. Stiennon, J. Wu, T. B. Brown, A. Radford, D. Amodei, P. Christiano, and G. Irving.
Fine-tuning language models from human preferences.
ArXiv, abs/1909.08593, 2019.