Learning from [code-related] feedback

Sean Welleck

Neural Code Generation
Carnegie Mellon University
January 18, 2024
Part I: Foundations

- Learning
- Evaluation
- Inference
- Data
Language model learning pipeline

- Pretraining
  - Gives a “foundation model”
- Adaptation
  - Continued pretraining
  - Fine-tuning
  - Learning from feedback
  - In-context learning / prompting
Language models

Problem: **distribution mismatch**

- Language model $p_\theta$ 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, not enough data, not diverse enough data, limited model capacity.
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 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
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
• Reinforcement learning
• Reward modeling
• Expert iteration
Adjust the model so that it maximizes a reward function:

\[
\arg \max_{\theta} \mathbb{E}_{x \sim D, y \sim p_{\theta}(\cdot|x)} [R(x, y)]
\]

Example reward:

\[ R(x, y) = 1 \text{ if program } y \text{ passes test cases} \]
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

At a high level:

- \( p_{\theta'} \leftarrow A(p_\theta, \{x\}, R) \)
Policy gradient methods [11, 10]

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

Estimate the gradient of the expected reward with respect to $\theta$:

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

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

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

Abstract

We propose a new family of policy gradient methods for reinforcement learning, which alternate between sampling data through interaction with the environment, and optimizing a “surrogate” objective function using stochastic gradient ascent. Whereas standard policy gradient methods perform one gradient update per data sample, we propose a novel objective function that enables multiple epochs of minibatch updates. The new methods, which we call proximal policy optimization (PPO), have some of the benefits of trust region policy optimization (TRPO), but they are much simpler to implement, more general, and have better sample complexity (empirically). Our experiments test PPO on a collection of benchmark tasks, including simulated robotic locomotion and Atari game playing, and we show that PPO outperforms other online policy gradient methods, and overall strikes a favorable balance between sample complexity, simplicity, and wall-time.

At the end, we get an alternative algorithm:

\[ p_{\theta'} \leftarrow A_{\text{PPO}}(p_{\theta}, \{x\}, R) \]
Recap

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

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

How do we choose the reward?
Issue 1: reward hacking

- Models can overfit to patterns in the reward
- Example:
  - \( R(x, y) = 1 \) if program \( y \) compiles, 0 otherwise

Then generating \( y = \text{print("hello world")] \) for all \( x \) would maximize reward.
Mitigation: KL-divergence penalty

- Keep the updated model close to the pretrained model
- $R_{KL} = -\beta \log \frac{p_{\theta}(y|x)}{p_0(y|x)}$
**Reward hacking: KL-divergence penalty [14]**

Mitigation: KL-divergence penalty

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

\[
D_{KL}(p_\theta(y|x)\|p_0(y|x)) = \sum_y p_\theta(y|x) \log \frac{p_\theta(y|x)}{p_0(y|x)} \\
= \mathbb{E}_{y \sim p_\theta} \log \frac{p_\theta(y|x)}{p_0(y|x)} \\
\approx \log \frac{p_\theta(\hat{y}|x)}{p_0(\hat{y}|x)},
\]

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
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 [8]

Execution

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

Syntactic matching score

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

Execution-based Code Generation using Deep RL [8]

Execution

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

Syntactic matching score

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

Semantic matching score

- $R_{\text{semantics}}(x, \hat{y}, y_\ast)$: overlap between dataflow graph of $y$ and $y_\ast$

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

*Execution-based Code Generation using Deep RL* [8]

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 \( y_* \)

Semantic matching score

- \( R_{\text{semantics}}(x, \hat{y}, y_*) \): 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
Table 1: Results on the code completion task for completing the last 25 masked tokens from CodeSearchNet.

<table>
<thead>
<tr>
<th>Model</th>
<th>↑\textit{xMatch}</th>
<th>↑\textit{Edit Sim}</th>
<th>↑\textit{Comp Rate}</th>
</tr>
</thead>
<tbody>
<tr>
<td>BiLSTM</td>
<td>20.74</td>
<td>55.32</td>
<td>36.34</td>
</tr>
<tr>
<td>Transformer</td>
<td>38.91</td>
<td>61.47</td>
<td>40.22</td>
</tr>
<tr>
<td>GPT-2</td>
<td>40.13</td>
<td>63.02</td>
<td>43.26</td>
</tr>
<tr>
<td>CodeGPT</td>
<td>41.98</td>
<td>64.47</td>
<td>46.84</td>
</tr>
<tr>
<td>CodeT5 (220M)</td>
<td>42.61</td>
<td>68.54</td>
<td>52.14</td>
</tr>
<tr>
<td>PPOCoder + CodeT5 (220M)</td>
<td><strong>42.63</strong></td>
<td><strong>69.22</strong></td>
<td><strong>97.68</strong></td>
</tr>
</tbody>
</table>

**Figure 1:** Compilation rate increases while holding other metrics constant
PPOCoder results

**Figure 2: APPS**

<table>
<thead>
<tr>
<th>Model</th>
<th>Size</th>
<th>↑pass@1</th>
<th>Intro</th>
<th>Inter</th>
<th>Comp</th>
<th>All</th>
<th>Intro</th>
<th>Inter</th>
<th>Comp</th>
<th>All</th>
</tr>
</thead>
<tbody>
<tr>
<td>Codex</td>
<td>12B</td>
<td>4.14</td>
<td>0.14</td>
<td>0.02</td>
<td>0.92</td>
<td>9.65</td>
<td>0.51</td>
<td>0.09</td>
<td>2.25</td>
<td></td>
</tr>
<tr>
<td>AlphaCode</td>
<td>1B</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>GPT-3</td>
<td>175B</td>
<td>0.20</td>
<td>0.03</td>
<td>0.00</td>
<td>0.06</td>
<td>2.70</td>
<td>0.73</td>
<td>0.00</td>
<td>1.02</td>
<td></td>
</tr>
<tr>
<td>GPT-2</td>
<td>0.1B</td>
<td>1.00</td>
<td>0.33</td>
<td>0.00</td>
<td>0.40</td>
<td>2.70</td>
<td>0.73</td>
<td>0.00</td>
<td>1.02</td>
<td></td>
</tr>
<tr>
<td>GPT-2</td>
<td>1.5B</td>
<td>1.30</td>
<td>0.70</td>
<td>0.00</td>
<td>0.68</td>
<td>3.60</td>
<td>1.03</td>
<td>0.00</td>
<td>1.34</td>
<td></td>
</tr>
<tr>
<td>GPT-Neo</td>
<td>2.7B</td>
<td>3.90</td>
<td>0.57</td>
<td>0.00</td>
<td>1.12</td>
<td>5.50</td>
<td>0.80</td>
<td>0.00</td>
<td>1.58</td>
<td></td>
</tr>
<tr>
<td>CodeT5</td>
<td>60M</td>
<td>1.40</td>
<td>0.67</td>
<td>0.00</td>
<td>0.68</td>
<td>2.60</td>
<td>0.87</td>
<td>0.10</td>
<td>1.06</td>
<td></td>
</tr>
<tr>
<td>CodeT5</td>
<td>220M</td>
<td>2.50</td>
<td>0.73</td>
<td>0.00</td>
<td>0.94</td>
<td>3.30</td>
<td>1.10</td>
<td>0.10</td>
<td>1.34</td>
<td></td>
</tr>
<tr>
<td>CodeT5</td>
<td>770M</td>
<td>3.60</td>
<td>0.90</td>
<td>0.20</td>
<td>1.30</td>
<td>4.30</td>
<td>1.37</td>
<td>0.20</td>
<td>1.72</td>
<td></td>
</tr>
<tr>
<td>CodeRL+CodeT5</td>
<td>770M</td>
<td>4.90</td>
<td>1.06</td>
<td>0.5</td>
<td>1.71</td>
<td>8.60</td>
<td>2.64</td>
<td>1.0</td>
<td>3.51</td>
<td></td>
</tr>
<tr>
<td>PPOCoder +CodeT5</td>
<td>770M</td>
<td>5.20</td>
<td>1.00</td>
<td>0.5</td>
<td>1.74</td>
<td>9.10</td>
<td>2.50</td>
<td>1.20</td>
<td>3.56</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 3: MBPP (transfer)**

<table>
<thead>
<tr>
<th>Model</th>
<th>Size</th>
<th>State</th>
<th>↑pass@80</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPT</td>
<td>224M</td>
<td>fine-tuned</td>
<td>7.2</td>
</tr>
<tr>
<td>GPT</td>
<td>422M</td>
<td>fine-tuned</td>
<td>12.6</td>
</tr>
<tr>
<td>GPT</td>
<td>1B</td>
<td>fine-tuned</td>
<td>22.4</td>
</tr>
<tr>
<td>GPT</td>
<td>4B</td>
<td>fine-tuned</td>
<td>33.0</td>
</tr>
<tr>
<td>GPT</td>
<td>8B</td>
<td>fine-tuned</td>
<td>40.6</td>
</tr>
<tr>
<td>GPT</td>
<td>68B</td>
<td>fine-tuned</td>
<td>53.6</td>
</tr>
<tr>
<td>GPT</td>
<td>137B</td>
<td>fine-tuned</td>
<td>61.4</td>
</tr>
<tr>
<td>CodeT5</td>
<td>60M</td>
<td>fine-tuned</td>
<td>19.2</td>
</tr>
<tr>
<td>CodeT5</td>
<td>220M</td>
<td>fine-tuned</td>
<td>24.0</td>
</tr>
<tr>
<td>CodeT5</td>
<td>770M</td>
<td>fine-tuned</td>
<td>32.4</td>
</tr>
<tr>
<td>CodeRL+CodeT5</td>
<td>770M</td>
<td>zero-shot</td>
<td>63.0</td>
</tr>
<tr>
<td>PPOCoder +CodeT5</td>
<td>770M</td>
<td>zero-shot</td>
<td><strong>68.2</strong></td>
</tr>
</tbody>
</table>
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); \( a_i \leq b_i \) 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 \( m \) \((1 \leq n \leq 1000, 1 \leq m \leq 10)\).

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

---

**Generated (Python)**

### CodeT5

```python
MOD = 10**9 + 7

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

```python
MOD = 10**9 + 7

def solve(n, m):
    dp = [[0] * (n + 1) for _ in range(m + 1)]
    for i in range(2, m + 1):
        for j in range(1, n + 1):
            for k in range(1, j + 1):
                dp[j][i] = (dp[j][i] + dp[j - 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? **✓
Qualitative – what is it changing?

Figure 8: Error types (APPS), from *RLTF: Reinforcement Learning from Unit Test Feedback* [5]
Qualitative – what is it changing?

Figure 8: Error types (APPS), from *RLTF: Reinforcement Learning from Unit Test Feedback* [5]

Figure 9: RLTF: analogous RL method with similar performance

programs that pass, especially for problems with introductory difficulty levels. The observed increase in failure rate stems from the fixing of error codes, resulting in either pass or fail outcomes. This illustrate that RLTF is more effective in addressing runtime and compiler errors compared to semantic errors, which remain more challenging. Figure 2b illustrates the percentages of different sub-errors among erroneous results before and after applying the RLTF method. Most errors show a decline in proportion after using the RLTF method, particularly syntax errors. It is also noteworthy that the proportion of timeout errors exhibits a minor increase, which can be attributed to the correction of other grammar-related errors resulting in timeout errors.
Tradeoffs

RL with policy gradient methods

• Directly optimizes reward
• Susceptible to reward hacking; requires good reward design
• Learning procedure adds complexity
• So far, improvements may be explained by syntax/index fixes
RL with policy gradient methods

- Directly optimizes reward
- Susceptible to reward hacking; requires good reward design
- Learning procedure adds complexity
- So far, improvements may be explained by syntax/index fixes
Outline

• Reinforcement learning
• Reward modeling
• Expert iteration
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 $R_\phi(y)$
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”
LLMs: investigated on math word problems [2]
Reward modeling: LEVER

Learning to Verify Language-to-Code Generation with Execution [6]

• 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 $y$
Given \((x, \mathcal{E}(y_\star))\)

- 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 \(\mathcal{E}(y_\star)\), 0 otherwise

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

```python
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
**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: |
<table>
<thead>
<tr>
<th>name</th>
<th>born_state</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dudley Hart</td>
<td>California</td>
<td>52.0</td>
</tr>
<tr>
<td>Jeff Maggert</td>
<td>Delaware</td>
<td>53.0</td>
</tr>
<tr>
<td>Franklin Langham</td>
<td>Connecticut</td>
<td>67.0</td>
</tr>
<tr>
<td>Billy Mayfair</td>
<td>California</td>
<td>69.0</td>
</tr>
<tr>
<td>K. J. Choi</td>
<td>Alabama</td>
<td>69.0</td>
</tr>
</tbody>
</table>
```

**Output:** 

no
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
At test time:

- Generate \( \{y_1, \ldots, y_K\} \sim p_\theta(\cdot|x) \)
- Select the program \( y_k \) with the highest score \( R(x, y_k) \).
  \[
  r(x, y_k) = p_\theta(y_k|x) \cdot p_\phi(v = 1|x, y_k, E(y_k)) \\
  R(x, y_k) = \sum_{y_{k'}} \text{ with same exec result as } y_k r(x, y_{k'})
  \]
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.
Figure 2: Comparison of LEVER and baselines with Codex-Davinci. LEVER and its ablation results are in solid bars.

**Figure 10:** LEVER improves performance. Using execution info is important.
Figure 11: Scaling the number of samples
Tradeoffs

Reward modeling:

- Does not require updating generator $p_\theta$
- Simple learning objective: standard maximum likelihood
- Strong performance
- Bounded by the generator’s capabilities
- Expensive at generation time
- Reward model is imperfect
• Reinforcement learning
• Reward modeling
• Expert iteration
Alternate between **search** and **learning**:

- **Search**: Use an ‘expert model’ to find good outputs
- **Learning**: Fine-tune on the discovered outputs
- Repeat
Thinking Fast and Slow with Deep Learning and Tree Search

Thomas Anthony¹, Zheng Tian¹, and David Barber¹,²

¹University College London
²Alan 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 MGHex 1.0, the most recent Olympiad Champion player to be publicly released.

Figure 12: Anthony et al 2017
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)
Builds on recent ideas, e.g. for reasoning [13, 12], generation [4], preference alignment [3].
Algorithm 1: ReST (Expectation-Maximization). Given a initial policy (e.g., pre-trained LM), ReST\textsuperscript{EM} iteratively applies Generate and Improve steps to update the policy.

**Input:** \( D \): Training dataset, \( D_{val} \): Validation dataset, \( L(x, y; \theta) \): loss, \( r(x, y) \): 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 \( D_i \) by sampling: \( D_i = \{ (x^j, y^j) \}_{j=1}^{N} \) s.t. \( x^j \sim D, \ y^j \sim p_\theta(y|x^j) \)
  
  Annotate \( D_i \) with the reward \( r(x, y) \).

  // **Improve** (M-step)

  **while** reward improves on \( D_{val} \) **do**
  
  | Optimise \( \theta \) to maximize objective: \( J(\theta) = \mathbb{E}_{(x,y) \sim D_i} [r(x, y) \log p_\theta(y|x)] \)
  
  **end**

**end**

**Output:** Policy \( p_\theta \)
Self-training with execution

Figure 13: On the MATH dataset, improves for multiple iterations
Self-training with execution

Figure 14: On a subset of APPS: initially improves, then overfits.
Self-training with execution

Figure 15: On a subset of APPS: improves pass@k
Connection with reinforcement learning

$$\mathcal{L}_{RL}(\theta) = \mathbb{E}_{x \sim D, y \sim p_\theta(y|x)} [R(x, y)]$$
Connection with reinforcement learning

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

Policy gradient methods: interleave updates and generation

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

See the Rest-EM paper [9] for more details on the connection.
Connection with reinforcement learning

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

Policy gradient methods: interleave updates and generation

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

Self-training: generate a large dataset, then update

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

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

Policy gradient methods: interleave updates and generation

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

Self-training: generate a large dataset, then update

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

See the Rest-EM paper [9] for more details on the connection.
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!
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
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.*


**Training verifiers to solve math word problems.**


**Lever: Learning to verify language-to-code generation with execution.**


**Execution-based code generation using deep reinforcement learning.**

**Beyond human data: Scaling self-training for problem-solving with language models.**

*ArXiv, abs/2312.06585, 2023.*


**STar: Bootstrapping reasoning with reasoning.**  

**Fine-tuning language models from human preferences.**  