Inference-Time Compute and “Reasoning” Models

CS 288: Advanced Natural Language Processing

Slides adapted from Daniel Fried, Sean Welleck, Yejin Choi, Graham Neubig

How to Build an LLM?

1. Pretraining
    1. Collect training documents and preprocess them
    2. Optimize language modeling objective
    3. **Outcome:** really good *language* model, but:
        1. Doesn't have a useful (natural language) interface
        2. Doesn't necessarily exhibit desired behavior (alignment)

How to Build an LLM?

2. Posttraining
    1. Instruction-tuning to solve the interface problem
        1. Collect examples of natural language instructions paired with demonstrations
        2. Fine-tune base language model to generate response conditioned on instruction
    2. Reinforcement learning from human feedback to solve the alignment problem
        1. For some new instructions, sample candidate responses from the instruction-tuned model
        2. Ask human annotators to rank the set of candidates
        3. Train a reward model to, for a pair of candidates, assign a higher score to the candidate that the annotator ranked higher
        4. Fine-tune the instruction-tuned model via RL, using reward model
    3. **Outcome:** model that follows natural language instructions directly

How to Build an LLM?

- What's left?
- For complex tasks, model may not "know" the best way to solve it
- Model might be bad at some target task, for example:
  - Really challenging math problems
  - Very domain-specific problems, e.g., medical reasoning, new programming languages, etc.
- Running inference may be inefficient or impossible due to model size
- **Can we solve these problems without respending all of the compute we used to get our instruction-tuned model?**

Inference-Time Adaptation

- Too expensive to fine-tune a model?
- Too little (or no) data available for fine-tuning?
- No access to model weights?
- No access to output probabilities?
- No problem

Recall: In-Context Learning

Zero-shot prompting (base LM)

The cafeteria had 23 apples. If they used 20 to make lunch and bought 6 more, how many apples do they have?

The restaurant had 15 oranges. If they used 2 to make dinner and bought 3 more, how many oranges do they have?

Few-shot prompting

Q: Roger has 5 tennis balls. He buys 2 more cans of tennis balls. Each can has 3 tennis balls. How many tennis balls does he have now?

A: The answer is 11.

Q: The cafeteria had 23 apples. If they used 20 to make lunch and bought 6 more, how many apples do they have?

A: The answer is 27.

prompt, task input, model output

Wei et al. 2022

"Chain-of-Thought" (CoT) Prompting

Main idea: "prime" model to generate step-by-step solution to input problem

Q: Roger has 5 tennis balls. He buys 2 more cans of tennis balls. Each can has 3 tennis balls. How many tennis balls does he have now?

A: Roger started with 5 balls. 2 cans of 3 tennis balls each is 6 tennis balls. 5 + 6 = 11. The answer is 11.

Q: The cafeteria had 23 apples. If they used 20 to make lunch and bought 6 more, how many apples do they have?

A: The cafeteria had 23 apples originally. They used 20 to make lunch. So they had 23 - 20 = 3. They bought 6 more apples, so they have 3 + 6 = 9. The answer is 9.

prompt, task input, model output

Wei et al. 2022

Zero-Shot CoT

Main idea: format input to prompt model to generate a step-by-step solution

```
Q: The cafeteria had 23 apples. If they used 20 to make lunch
and bought 6 more, how many apples do they have?

A: Let's think step by step. The cafeteria had 23 apples
originally. They used 20 to make lunch. So they had 23 - 20 = 3.
They bought 6 more apples, so they have 3 + 6 = 9. The answer is
9.
```

Advantages:

- Extra tokens provide adaptive computation time
- If the CoT is faithful to the final answer, then it supports better verification of model output

prompt, task input, model output

Kojima et al. 2022

Zero-Shot CoT

Main idea: format input to prompt model to generate a step-by-step solution

No.	Category	Template	Accuracy
1	instructive	Let's think step by step.	**78.7**
2		First, (*1)	77.3
3		Let's think about this logically.	74.5
4		Let's solve this problem by splitting it into steps. (*2)	72.2
5		Let's be realistic and think step by step.	70.8
6		Let's think like a detective step by step.	70.3
7		Let's think	57.5
8		Before we dive into the answer,	55.7
9		The answer is after the proof.	45.7
10	misleading	Don't think. Just feel.	18.8
11		Let's think step by step but reach an incorrect answer.	18.7
12		Let's count the number of "a" in the question.	16.7
13		By using the fact that the earth is round,	9.3
14	irrelevant	By the way, I found a good restaurant nearby.	17.5
15		Abrakadabra!	15.5
16		It's a beautiful day.	13.1
-		(Zero-shot)	17.7

Kojima et al. 2022

“Meta-generation”

- Sampling from one LLM is a “generation”
- But we can improve performance if we perform and compose multiple (possibly independent) generations
  - (How? We’ll see a couple variations)

$$y = f_3 \circ f_2 \circ f_1$$

CoT, with latent trace *y*:

$$z \sim g(z|x), y \sim g(y|x, z)$$

Majority Voting

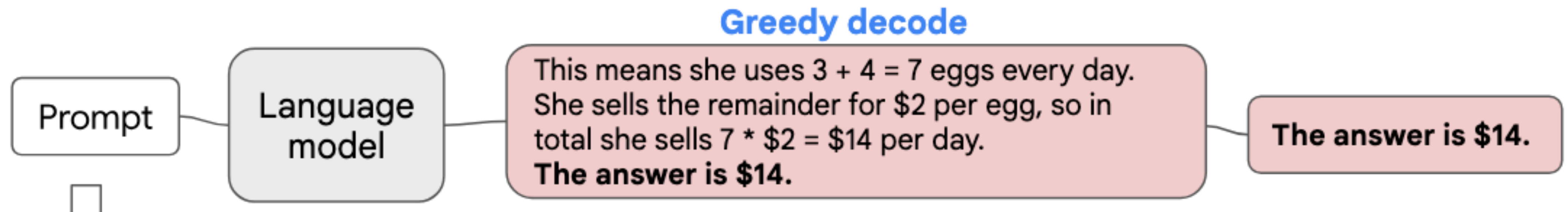
Sample *N* continuations independently

$$h(y^{(1)}, \ldots, y^{(N)}) \rightarrow (c(y^{(1)}), \ldots, c(y^{(N)}))$$

Choose the most frequent answer

$$\hat{a} = \arg\max_k \sum_{j=1}^{K} \sum_{i=1}^{N} \mathbb{I}(c(y^{(i)}) = j)$$

Majority Voting



Majority Voting

Chain-of-thought prompting

Prompt → Language model → Greedy decode: This means she uses 3 + 4 = 7 eggs every day. She sells the remainder for $2 per egg, so in total she sells 7 * $2 = $14 per day. **The answer is $14.** → **The answer is $14.**

Self-consistency

Q: *If there are 3 cars in the parking lot and 2 more cars arrive, how many cars are in the parking lot?*
A: There are 3 cars in the parking lot already. 2 more arrive. Now there are 3 + 2 = 5 cars. The answer is 5.
...
Q: Janet's ducks lay 16 eggs per day. She eats three for breakfast every morning and bakes muffins for her friends every day with four. She sells the remainder for $2 per egg. How much does she make every day?
A:

Language model

Sample a diverse set of reasoning paths

- She has 16 - 3 - 4 = 9 eggs left. So she makes $2 * 9 = $18 per day. — **The answer is $18.**
- This means she she sells the remainder for $2 * (16 - 4 - 3) = $26 per day. — **The answer is $26.**
- She eats 3 for breakfast, so she has 16 - 3 = 13 left. Then she bakes muffins, so she has 13 - 4 = 9 eggs left. So she has 9 eggs * $2 = $18. — **The answer is $18.**

Marginalize out reasoning paths to aggregate final answers

The answer is $18.

Main insight: by marginalizing over possible latent CoT traces, we get better performance

Wang et al. 2023

Rejection Sampling

If there are rejection criteria available, we can improve quality of sampled outputs by applying it

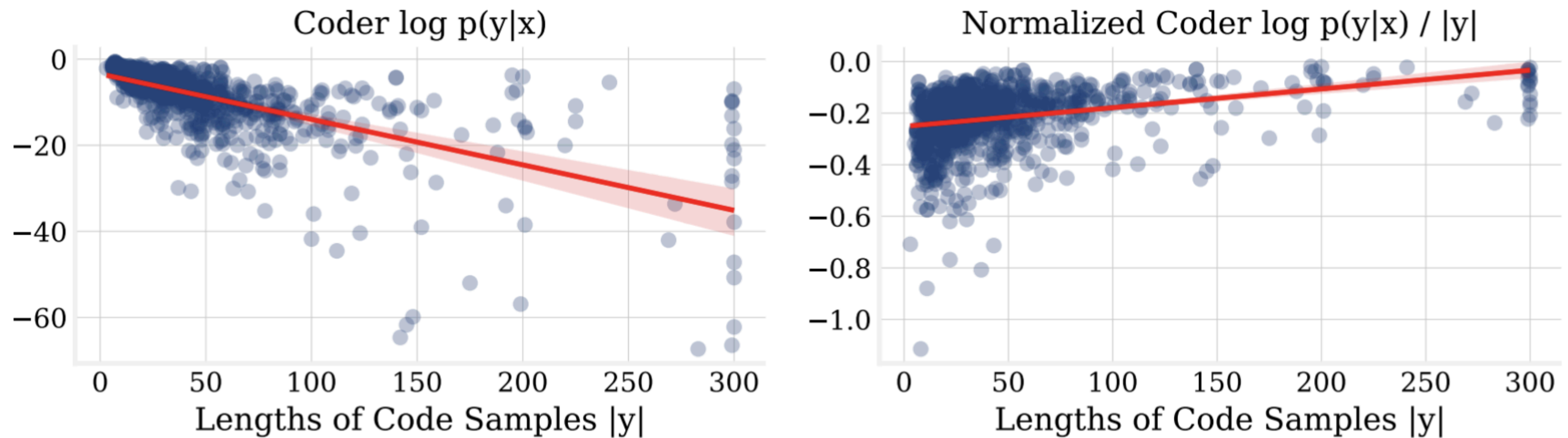
$$q_*(y) \propto \begin{cases} p_\theta(y) & y \text{ is valid JSON} \\ 0 & \text{Otherwise} \end{cases}$$

Reranking

$$f(x, g, v, N, \phi) = \underset{y^{(n)} | n \in \{1, \ldots, N\}}{\arg\max} \left\{ v(y^{(n)}) \mid y^{(n)} \sim g(\cdot|x), n \in \{1, 2, \ldots, N\} \right\}$$

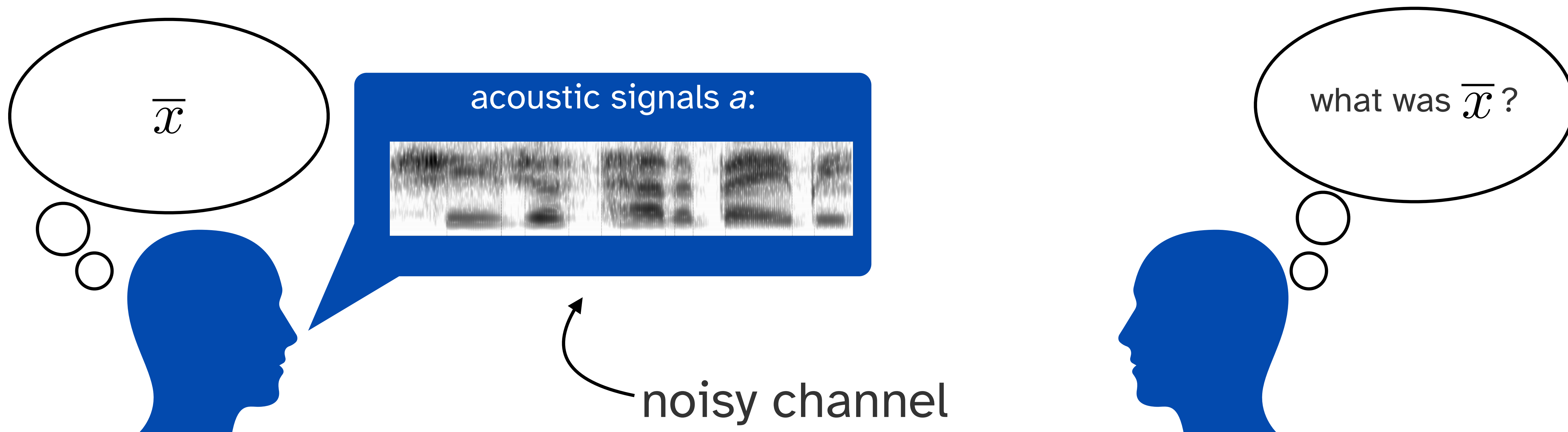
- Are there any other signals we could use to choose between a set of samples?
- One option: model probability
  - Sample many times, keeping track of model probability of samples, then choose the one with the highest probability
  - (What does this remind you of?)

Reranking by Model Probabilities



- Core issue: model probabilities correlate with output length, even when normalizing by length!
- What else could help us here?
  - (Hint: one of the foundational probabilistic models of NLP :)

Noisy Channel Model



Noisy Channel Model

Main idea: rerank outputs by an "inverse" model, which tells us how likely our observations were given some candidate solution

$$\begin{aligned} \overline{x}^* &= \arg\max_{\overline{x}} p(\overline{x} \mid a) \\ &= \arg\max_{\overline{x}} \frac{p(a \mid \overline{x})p(\overline{x})}{p(a)} \\ &= \arg\max_{\overline{x}} p(a \mid \overline{x})p(\overline{x}) \end{aligned}$$

Bayes' rule

acoustic model — $p(a \mid \overline{x})$

language model — $p(\overline{x})$

$\overline{x}$

acoustic signals *a*:

noisy channel

Reranking for Code Generation

Coder-Reviewer Reranking: sample programs and sort by p(y|x)p(x|y)

0-shot Python Function Completion

```
import math
def get_decimal(num: float):
  """ return the decimal part
  of the input number
  """
```

Coder: sample program via p(y|x)

```
frac, whole = math.modf(num)
return frac
```

Reviewer: check program via p(x|y)

```
import math
def f(num: float):
  frac, whole = math.modf(num)
  return frac
# write a docstring for the
# above function
def f(num: float):
  """ return the decimal part
  of the input number
  """
```

1. Sample outputs
2. Check probability of task description given sampled output
3. Rerank according to some mixture of original sampling probability, and inverse probability

$$\log p(x|y)p(y|x) = \log p(x|y) + \log p(y|x)$$

(Coder-Reviewer Reranking)

$$\text{argmax}_y \ \log \frac{p(y,x)}{p(x)p(y)^\alpha}$$

$$= \text{argmax}_y \ (1-\alpha)\log p(y|x) + \alpha \log p(x|y)$$

$$\frac{\log p(x|y)}{|x|} + \frac{\log p(y|x)}{|y|}$$

(Normalized Coder-Reviewer Reranking)

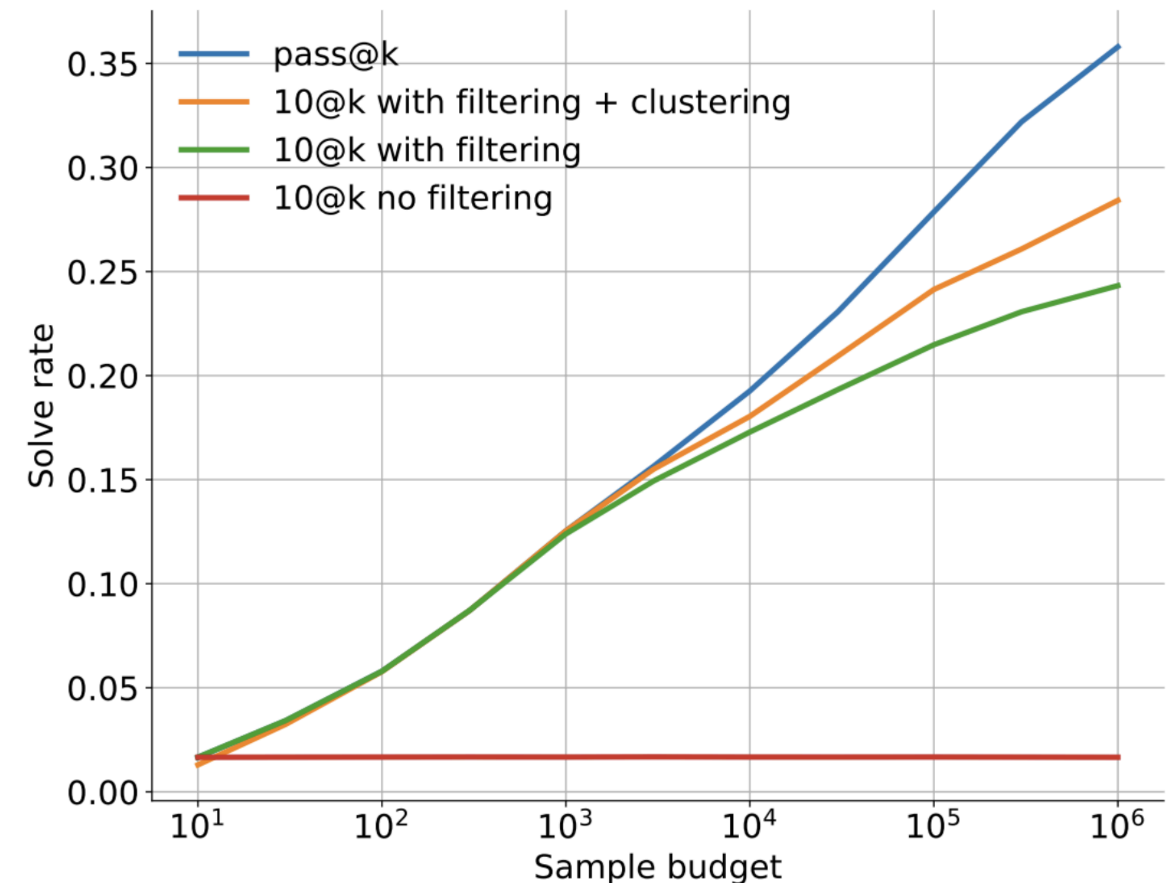
Generalization: Minimum Bayes Risk

- **Main idea:** assume we have access to some utility function that scores each hypothesis (candidate output)

- Then, we can:
  - Sample from the model *N* times (this gives us a hypothesis set, where sampling means the set should give us decent coverage of likely hypotheses)
  - Compute the utility of each hypothesis (this gives us an evidence set, after having evaluated each hypothesis)
  - Then, choose the item in the evidence set with the best utility

Example: AlphaCode

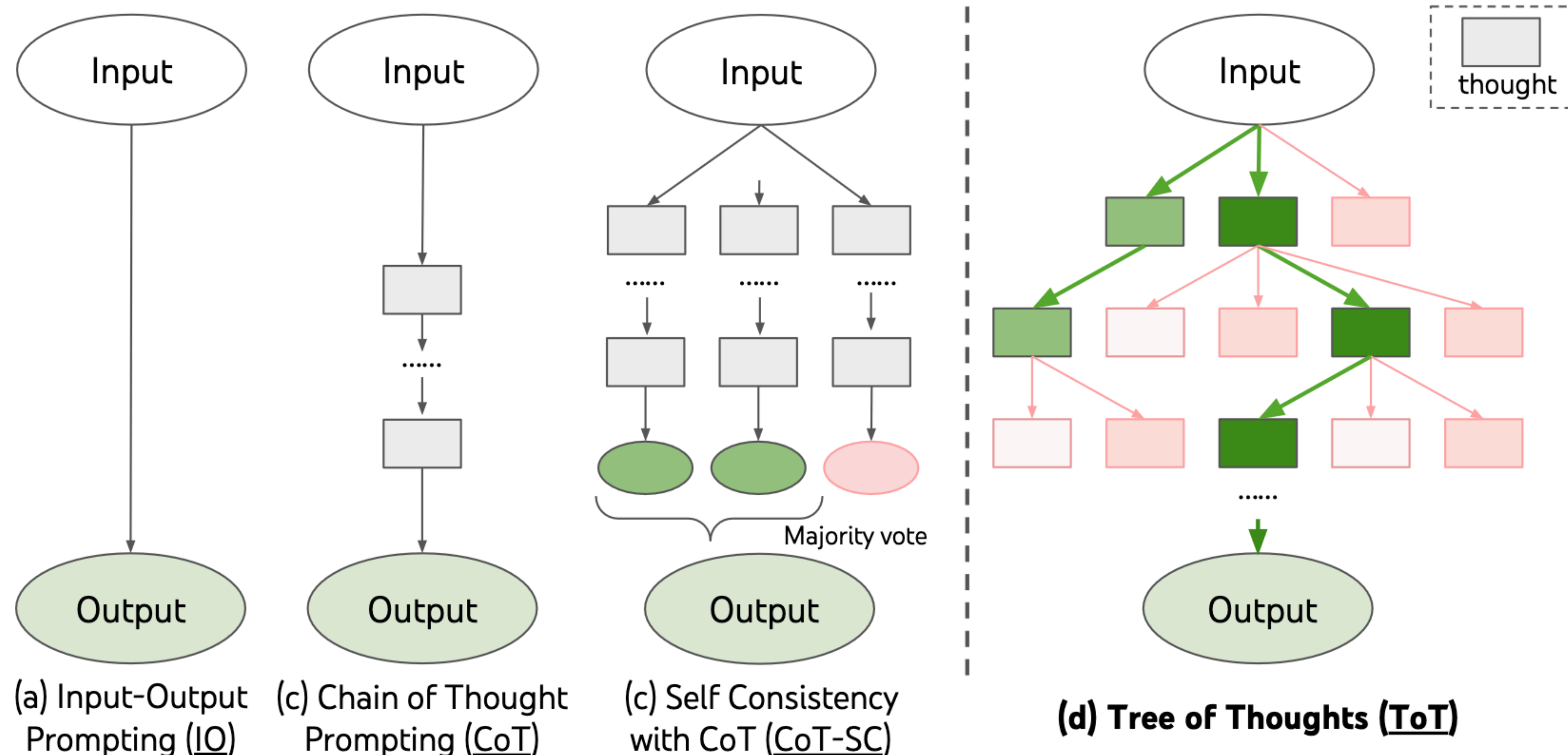
- Training
  - Pre-train encoder-decoder LMs (300-M — 41B parameters) on GitHub code
  - Fine-tune on 13K problems scraped from Codeforces
- Inference (on held-out code competition problems)
  - Sample a huge number (~1M) of candidate solutions for each problem
  - Filter candidates on public test cases
  - Organize remaining candidates into clusters, where all programs in each cluster give the same outputs for some model-generated inputs



https://alphacode.deepmind.com/

Structured Search

- Main idea: prompt language models to “consider different reasoning paths” and “self-evaluate choices to decide the next course of action”
- This lets us reject “chains” which are not promising early on



Yao et al. 2023

Limitations of CoT

- Found to perform best on tasks involving math and logic, and less well on open-ended reasoning tasks (e.g., general natural language reasoning)
- CoT traces aren't necessarily "faithful" to:
  - The answers produced
  - The model's "underlying reasoning"
- When instructed to explain an *incorrect* answer to a question, models often produce very plausible and confident-sounding explanations of why!
- Longer CoT generally lead to better model performance

Generation vs. Verification

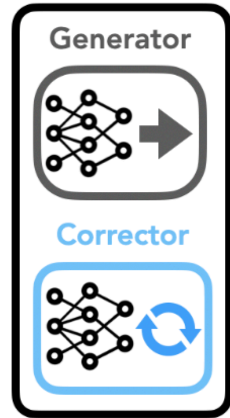
- **Generation:** map from some task description to an answer/ solution
  - Zero-shot
  - Chain-of-thought
  - Structured prompting
- **Verification:** map from some task description and an answer/ solution, to a judgment of the answer/solution
  - Pass/fail
  - Numerical score
  - Natural language critique
  - Edit

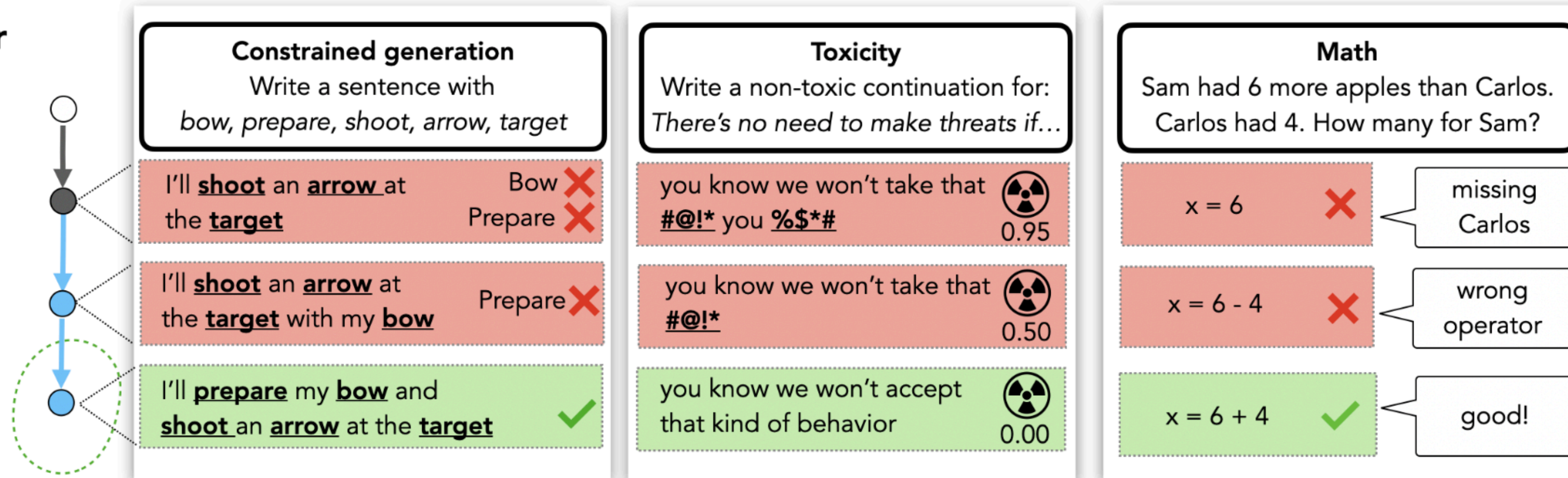
Iterative Refinement

- **Main idea:**
  - We have been using LLMs to do both *generation* and *verification* independently
  - Why not combine these into a single inference method to get better generation overall?
- In iterative refinement, we:
  - Generate: map from task description to answer/solution, possibly given existing critique of past answers/solutions
  - Verify: map from task description and a proposed answer/solution to some type of feedback
- Terminate when the verifier is “satisfied”, or until some maximum number of iterations

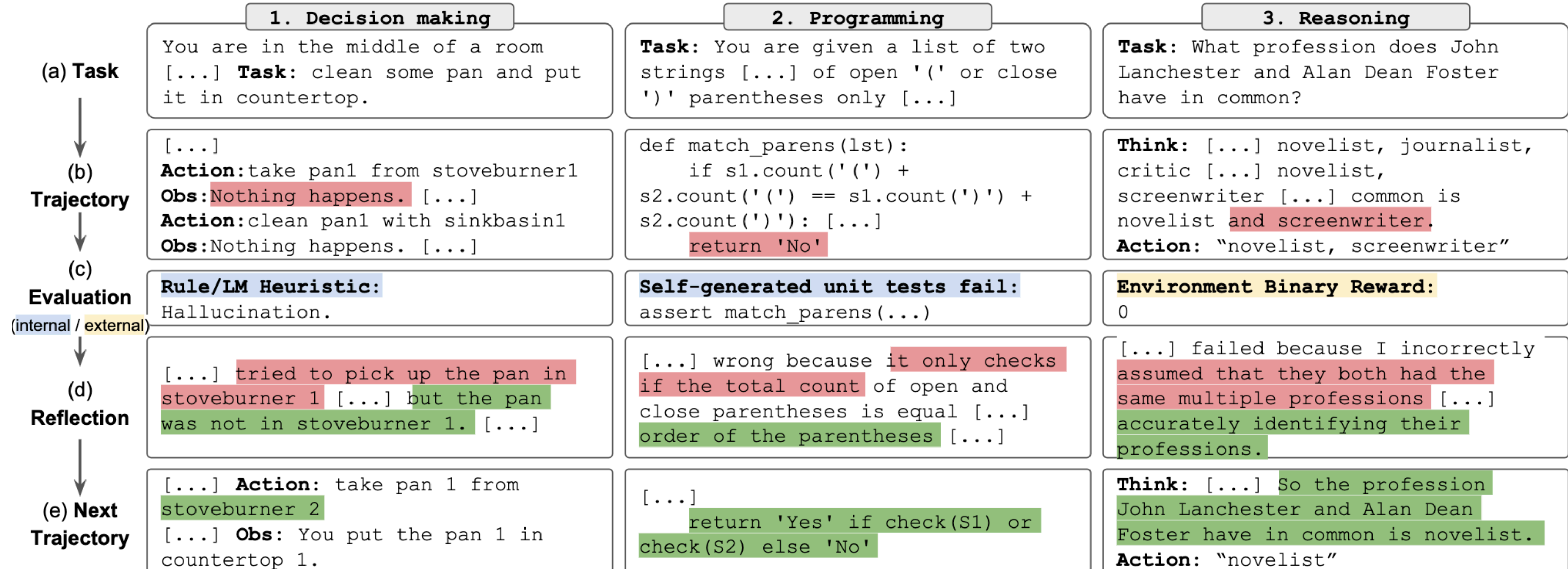
Iterative Refinement

Self-Corrector





Welleck et al. 2023



Shinn et al. 2023

Limitations of Iterative Refinement

- Intrinsic “self”-correction often fails without external feedback (e.g., from an environment) or oracle feedback
- Performance may actually degrade from iteration to iteration!
- Reasons for failures:
  - Models struggle to identify errors they’re likely to make
  - Models tend to judge initial answers/reasoning as correct (sycophancy?)
  - Models can’t correct what they don’t know
- Works well for surface-level errors (e.g., formatting) and with strong verifier models
- Works poorly when the task requires deep reasoning

Huang et al. 2024

Types of “Verifiers”

- What do they evaluate?
  - Outcome: is the solution correct?
  - Process: is the partial solution correct, or might it lead to a correct solution?
- How are they implemented?
  - “Self”-critique: just prompt the same model being used for generation
  - Trained reward models (neural networks)
  - Environment-specific feedback

Optimal Use of Test-Time Compute?

- Instead of scaling only training time compute, what if we scale test time compute?
  - Recall: how do we optimally scale training-time compute?
  - How might we scale test-time compute?
- Ways to scale
  - Generating more independent solutions, selecting the best with a verifier (best-of-N) (hyperparameter: sampling temperature)
  - Iteratively revise solutions with self-critique
  - Maintain a beam of different solutions, and prune based on "process" verifier rewards (hyperparameter: "diversity" of beam search)
- **Main question:** given a prompt, a generator, and available verifier(s), how should we allocate compute at test-time across different scaling methods?

Snell et al. 2024

Optimal Use of Test-Time Compute?

- **Observation:** optimal strategy depends on difficulty of the task
- **Approach:**
  - Decide on a way to evaluate a discrete "difficulty" of tasks you might encounter
  - Measure the difficulty of all of the training tasks
  - For each difficulty level, search across the possible strategies to identify which one generally works best
  - At inference time, estimate a new task's difficulty level, then apply its optimal strategy

Snell et al. 2024

Optimal Use of Test-Time Compute?

Findings from this study:

- A 14B parameter model using optimal test-time compute strategies can exceed the performance of models 4x larger
- Compared to outcome reward models (ORM) with beam search, using process reward models (PRM) enables 4-8x more efficient compute use
- We don't always need to use the largest models, as long as we have good inference strategies
- We should be focusing on <u>verification</u> more, especially PRMs

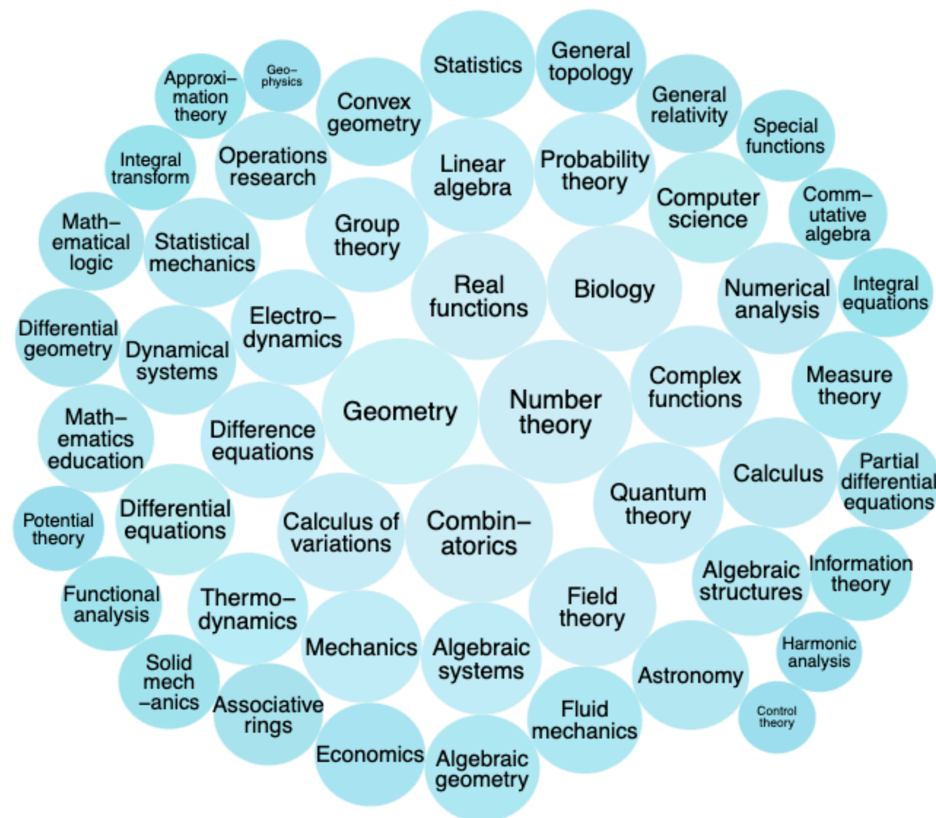
Snell et al. 2024

“Reasoning” Models

- Going all-in on serialized inference
- CoT is a powerful approach for getting better performance on any instruction-tuned model
- A “reasoning” model is explicitly trained to generate CoT

Supervised Fine-Tuning

- Main idea: curate a dataset of high-quality reasoning traces, then fine-tune models on these traces
- s1: 1,000 reasoning traces sampled from existing models
- Inference-time compute budget control



How many r in raspberry? **Question**

Let's break down the process of counting the letter 'r' in the word "raspberry" ...
* First letter: 'r' - This is an 'r', count = 1.
* Second letter: 'a' - Not an 'r', count remains 1 ...
* Sixth letter: 'e' - Not an 'r', count remains 1.
* Seventh letter: 'r' - This is an 'r', count = 2.
* Eighth letter: 'y' - Not an 'r', count remains 2 ...
The number of 'r's in "raspberry" is 2.
Wait, let's re-read the question carefully. It asks "How many r in raspberry?" ... * r - a - s - p - b - e - r - r - y ... * First 'r' ... * Second 'r' ... * Third 'r' ... Count = 3 ... **Reasoning trace**

My initial answer of 2 was incorrect due to a quick reading of the word. **Final Answer:** The final answer is $\boxed{3}$ **Response**

Muennighoff et al. 2025

Error Propagation in Sequential Decision Making

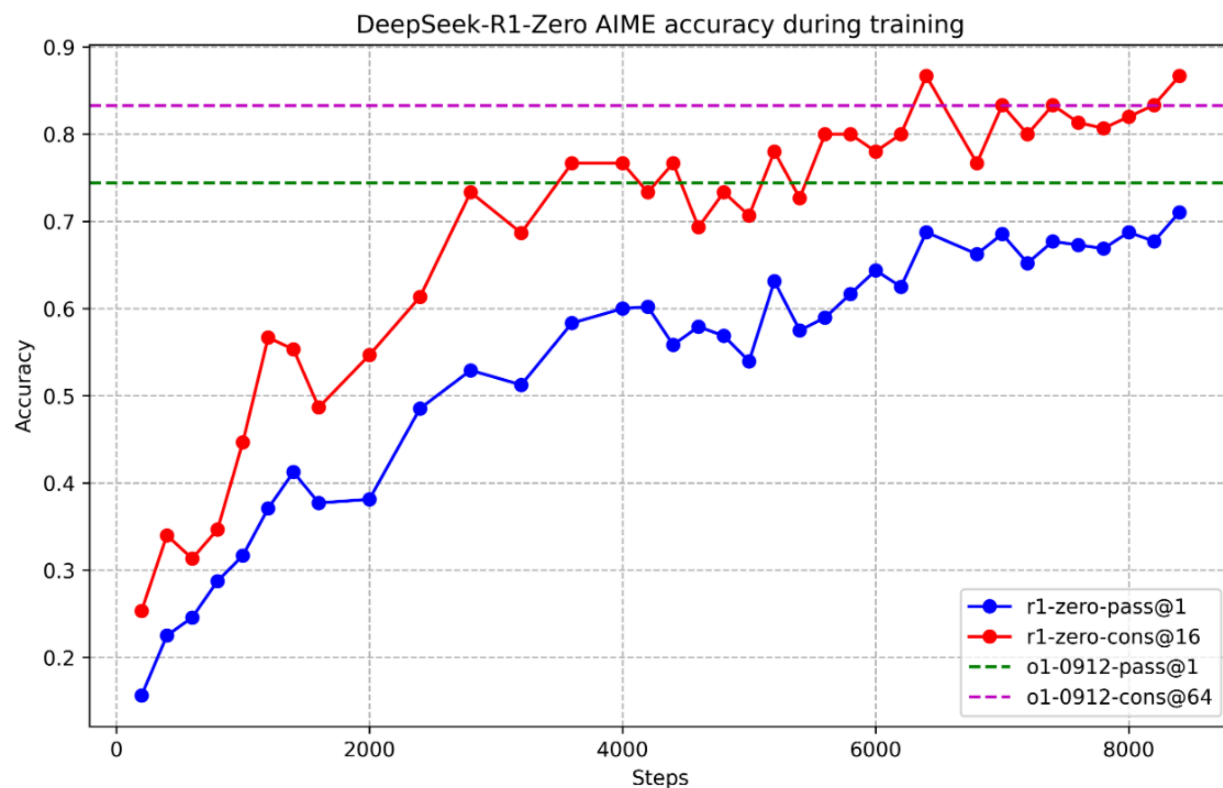
- Core challenge in dynamic systems: executing a policy may result in states for which it has poor estimates of what's best to do, e.g.:
  - Stuck in a corner
  - Repeated the same action over and over
  - Opened an app it was never trained on that looks similar to the correct app
- Problem: even SFT'd models haven't been trained on what to do, because experts rarely get into these situations

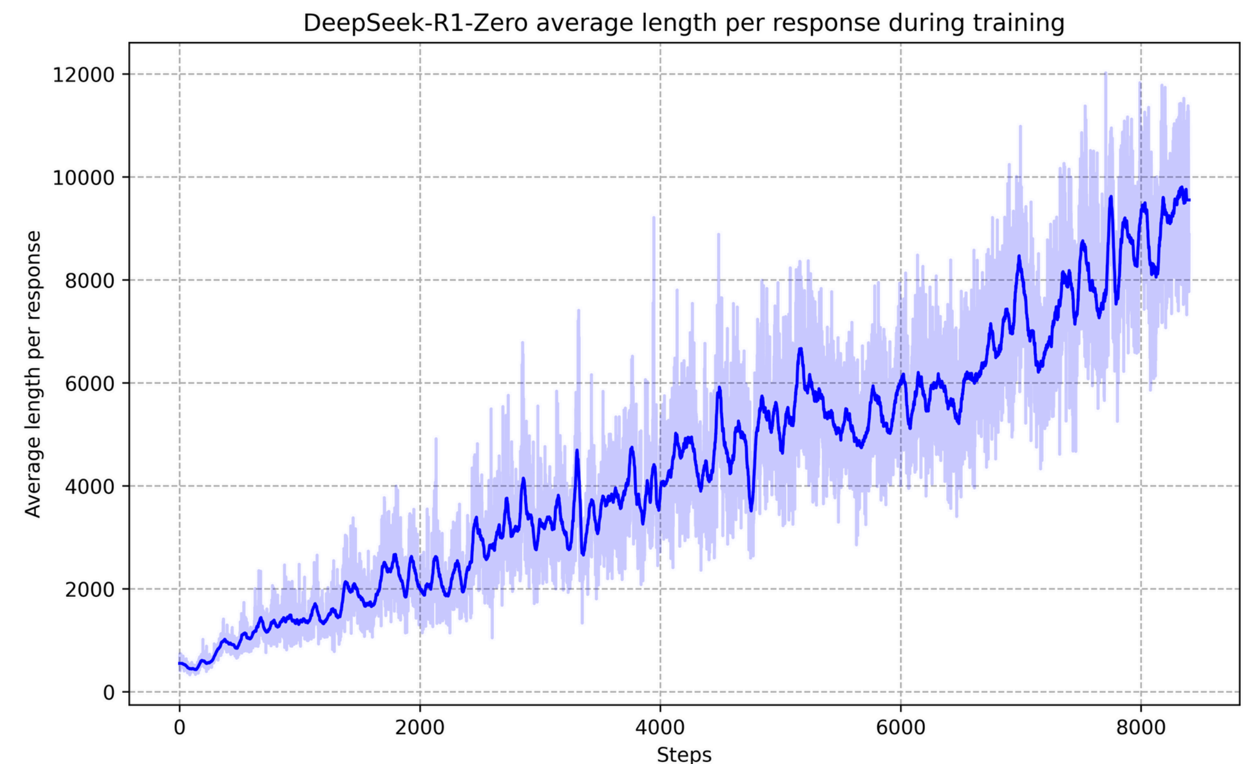
Learning from Exploration

- Instead of learning only from high-quality demonstrations, where local decisions are ~optimal, let's learn from what states the policy is likely to encounter at inference time
- **Really high-level overview:** learn by
  - Sampling from the current policy
  - Evaluating the quality of these samples
  - Optimizing the policy towards or away from the sample, depending on its quality

DeepSeek-R1

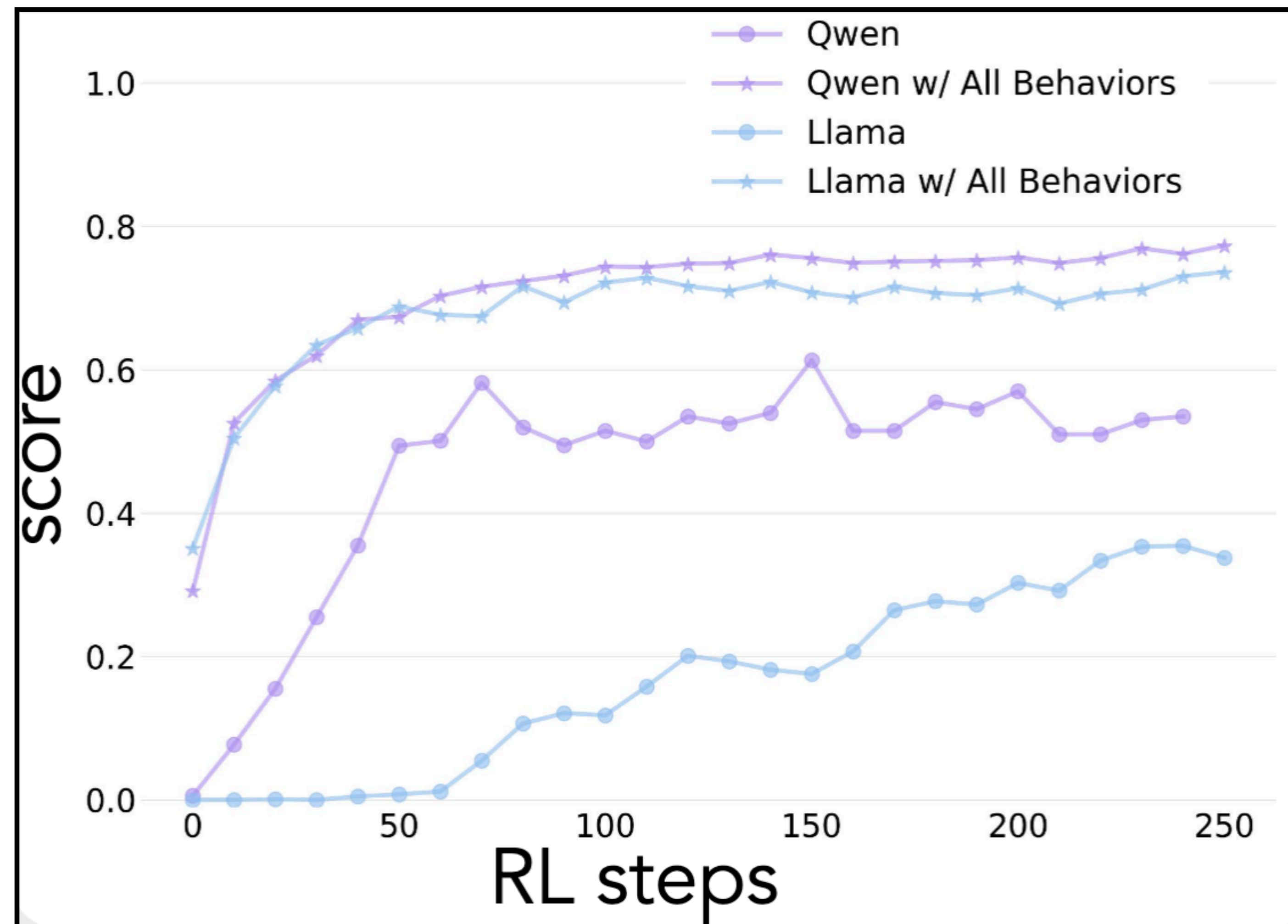
- During training, iteratively:
  - Map from questions to answers with CoT using existing policy, resulting in <question, CoT, answer> tuples
  - Assign a score to each tuple based on correctness
  - Optimize policy towards tuples with high score





Four Critical Reasoning Behaviors

- Self-verification
  - “Let me check my answer”
- Subgoal setting
  - “Let’s try to get a multiple of 10”
- Backtracking
  - “Let’s try a different approach, what if we...”
- Backward chaining
  - “Working backwards, 24 is 8 times 3”



But: faithfulness and interpretability problems remain

Some Remaining Questions in “Reasoning” Models

- How to train most effectively — online RL? pretraining data like s1?
- Under what conditions do different “reasoning strategies” “emerge”?
- How faithful are learned long CoT to actual reasoning processes? How interpretable are the CoT traces?
- How well does learning on one task generalize to other tasks?
- How can we more effectively take advantage of inference-time compute?