

Supervised Reinforcement Learning: From Expert Trajectories to Step-wise Reasoning

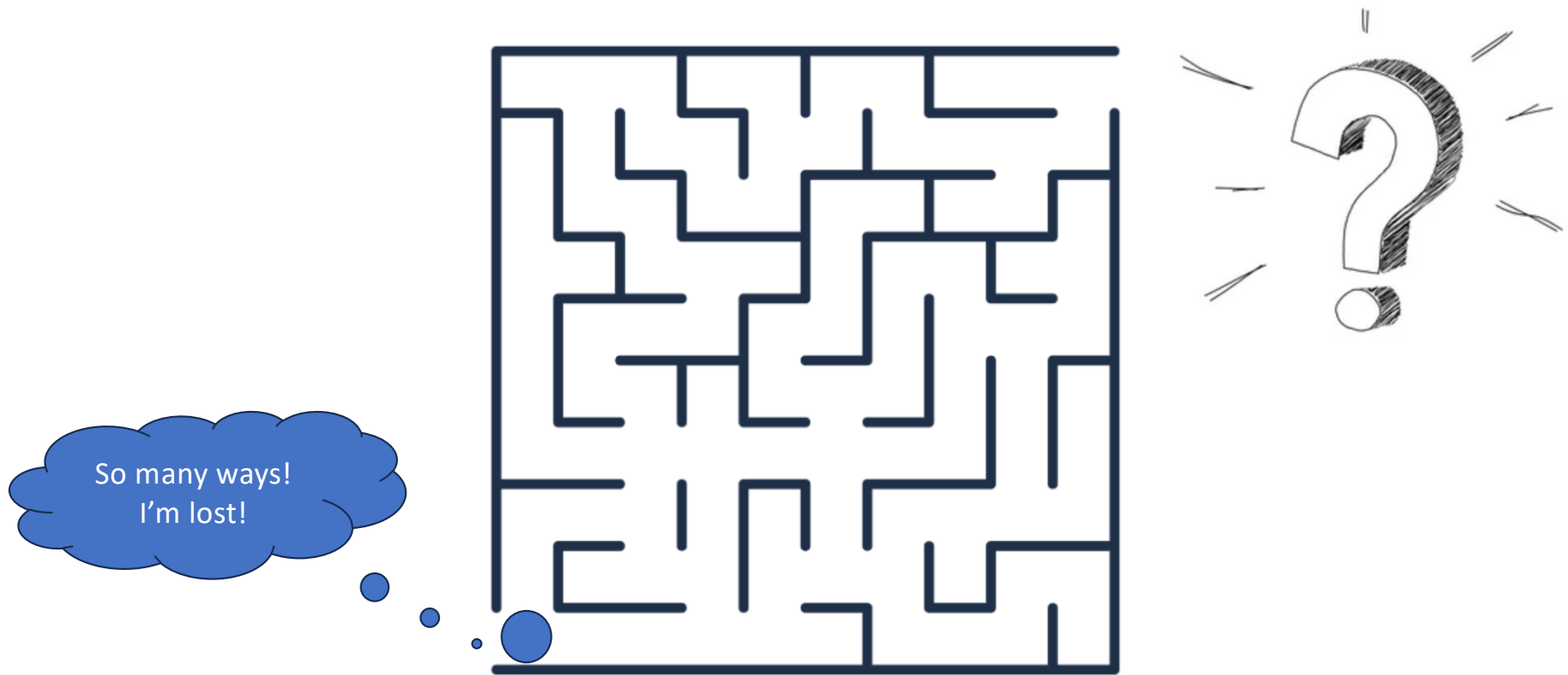
Yihe Deng^{2*†}, I-Hung Hsu^{1*}, Jun Yan¹, Zifeng Wang¹, Rujun Han¹, Gufeng Zhang³, Yanfei Chen¹, Wei Wang²,
Tomas Pfister¹ and Chen-Yu Lee¹

¹Google Cloud AI Research, ²UCLA, ³Google Cloud

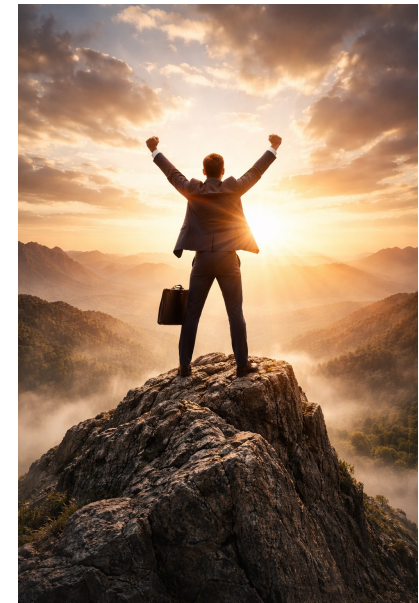
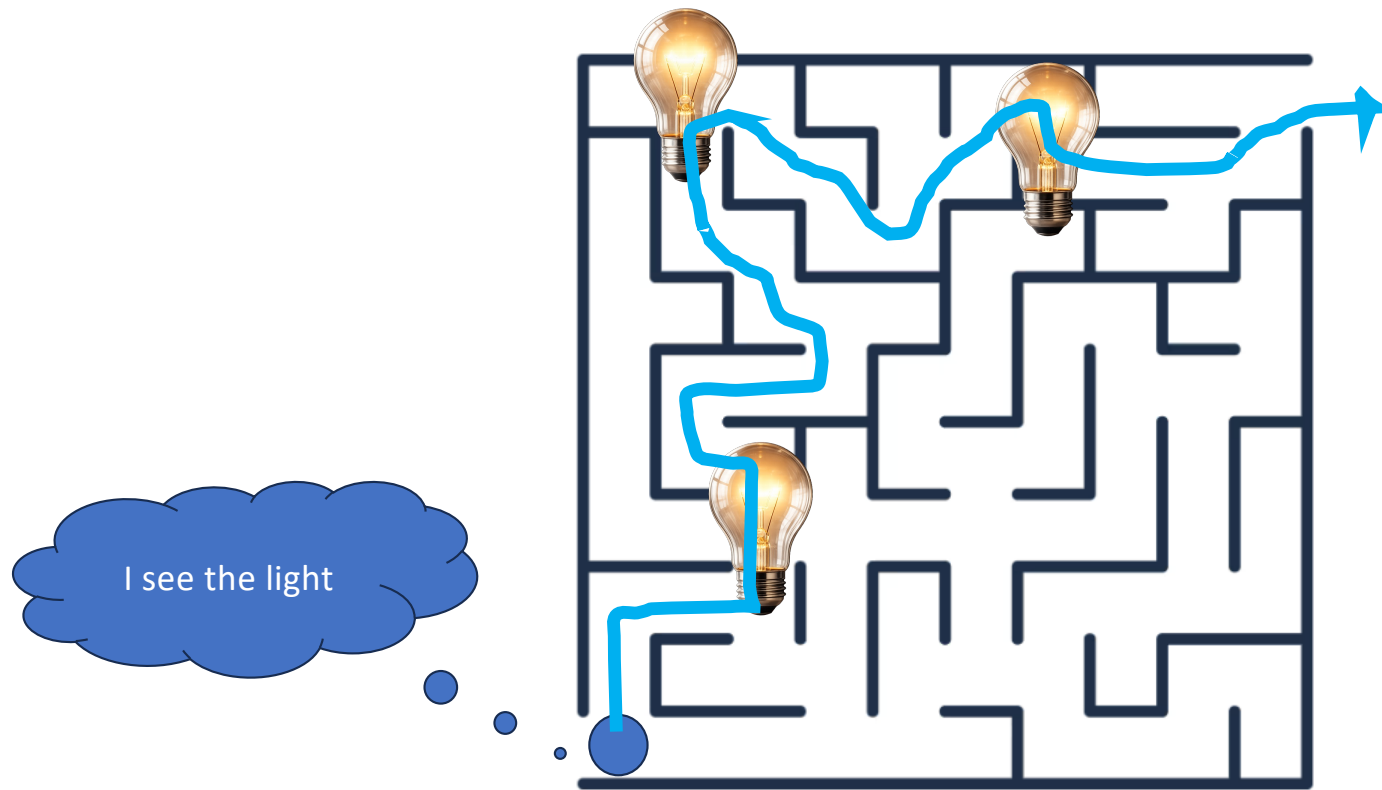
Plus some of my own suggestions for improvement

Presented by:
John Tan Chong Min

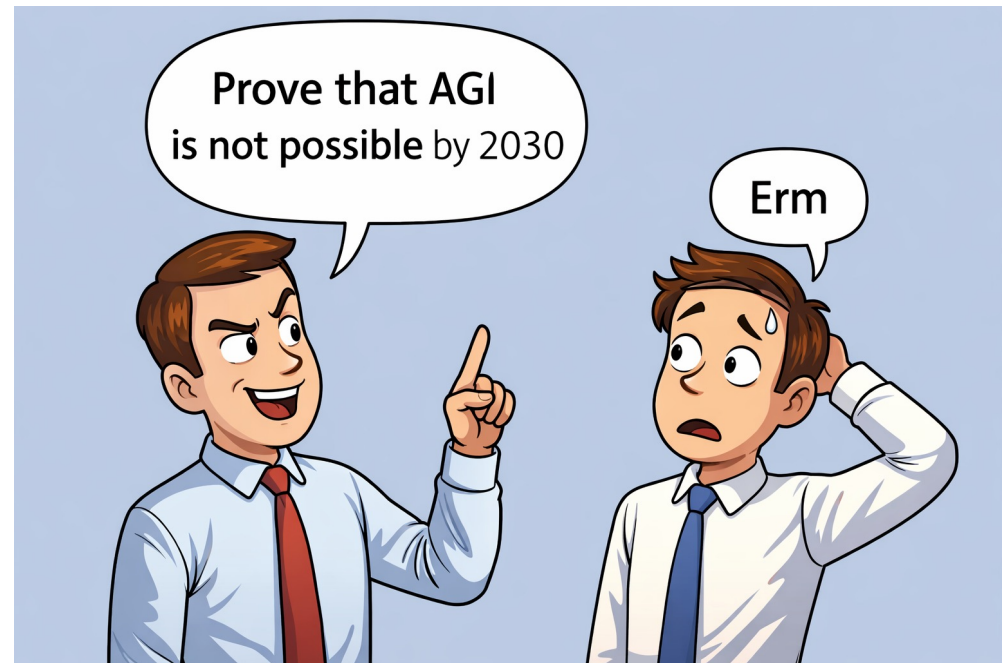
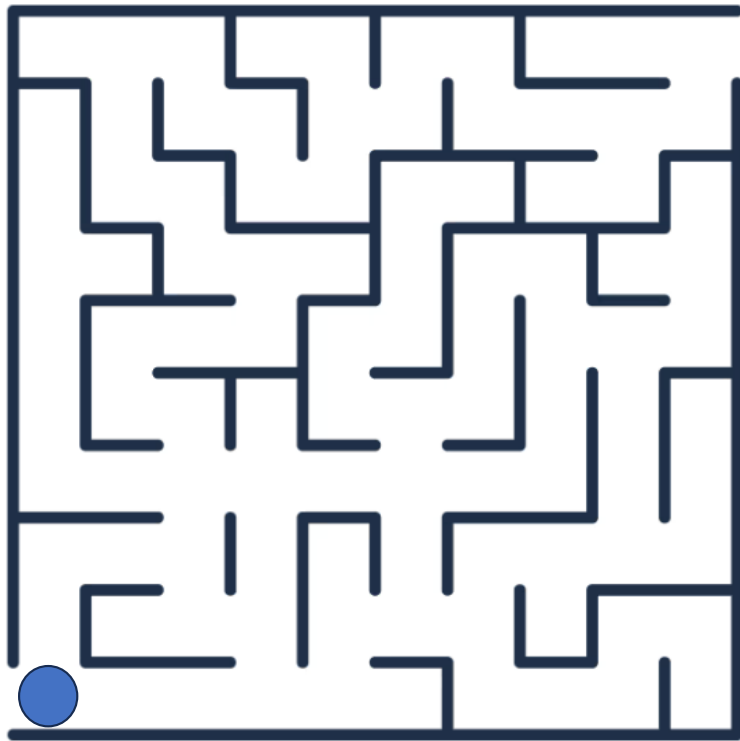
Imagine you are a ball trying to find the way out of a maze



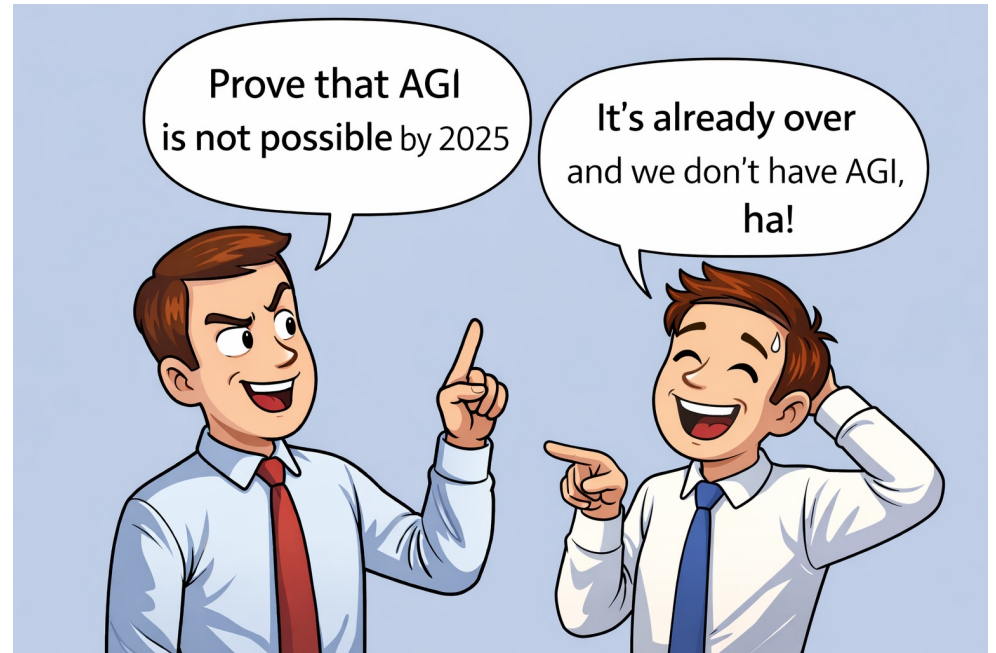
But now there are light bulbs guiding you along the way



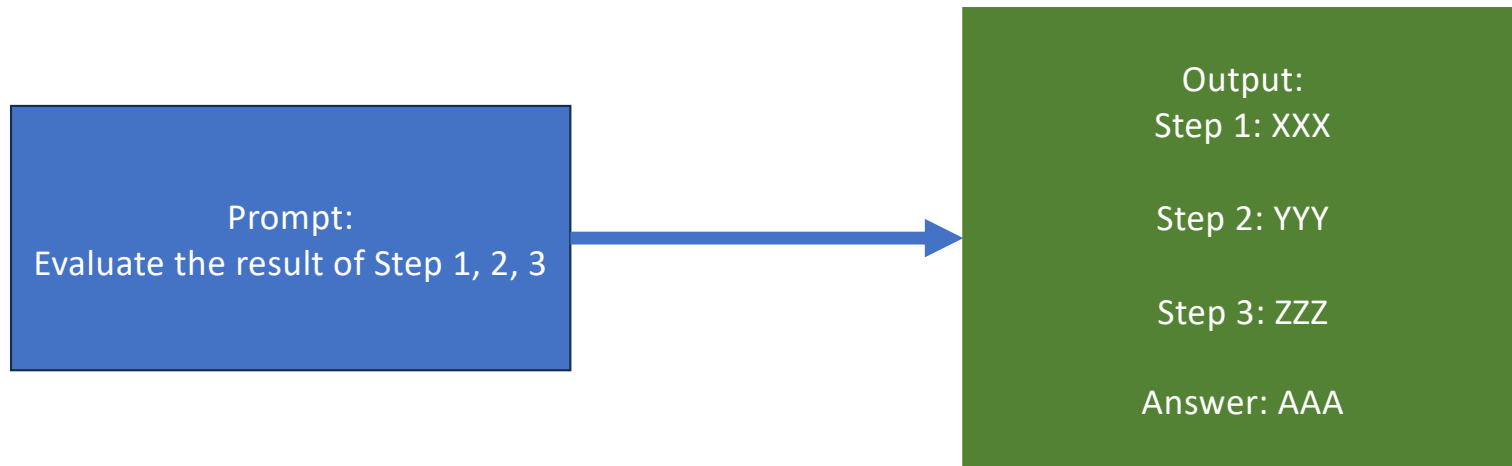
Trying to learn successful trajectories from a final output is like blindly solving a maze



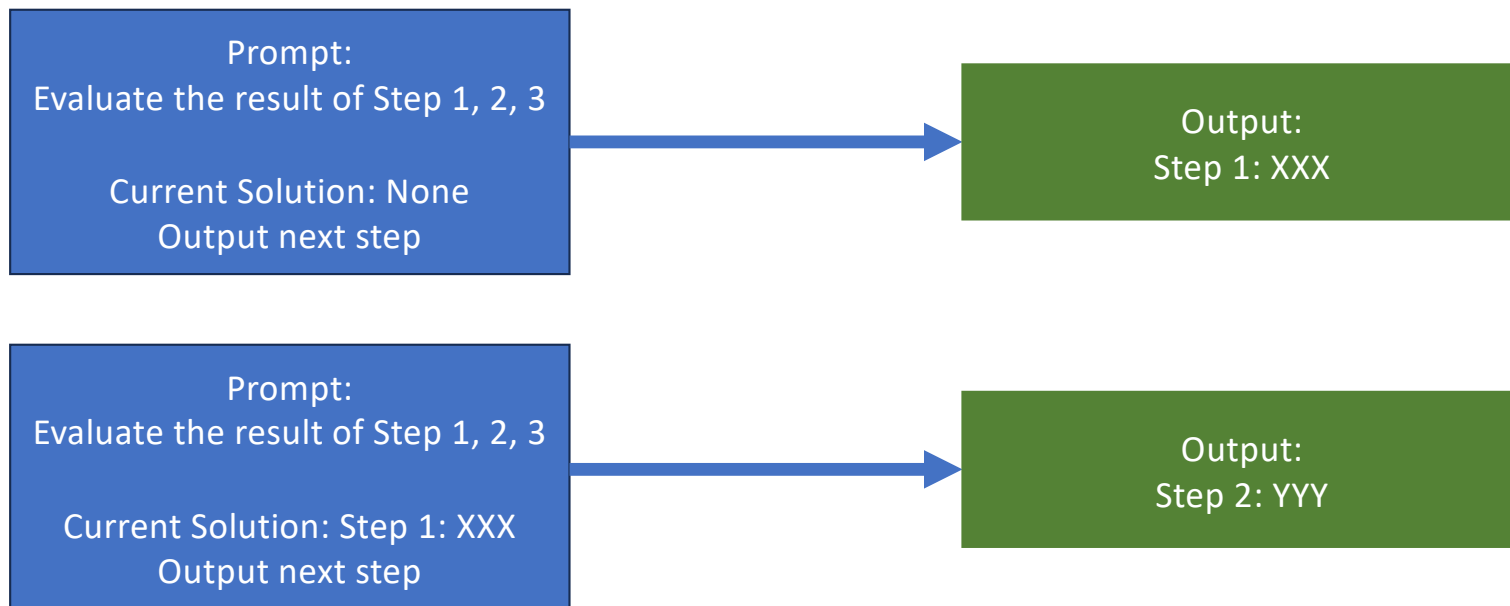
Learning from intermediate output is like having light bulbs in the maze



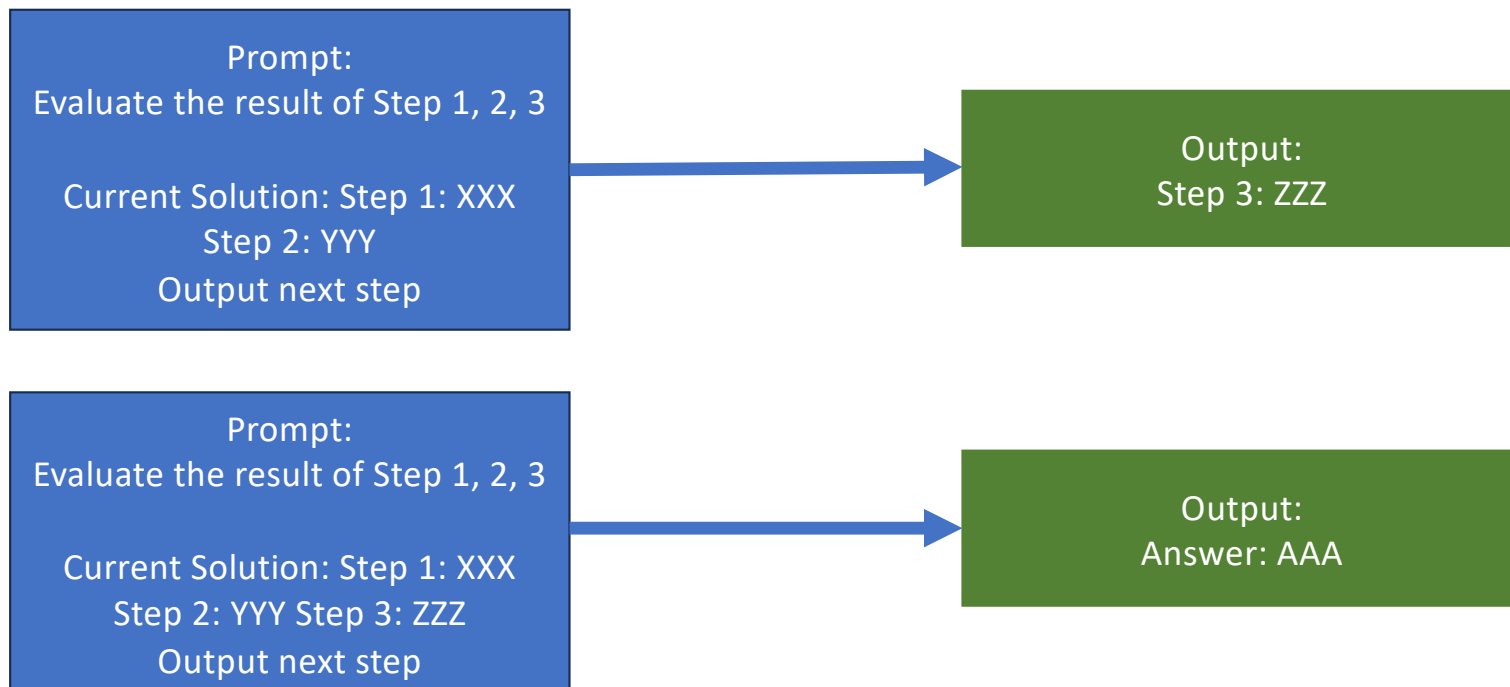
LLMs can struggle to learn from multiple steps at a go



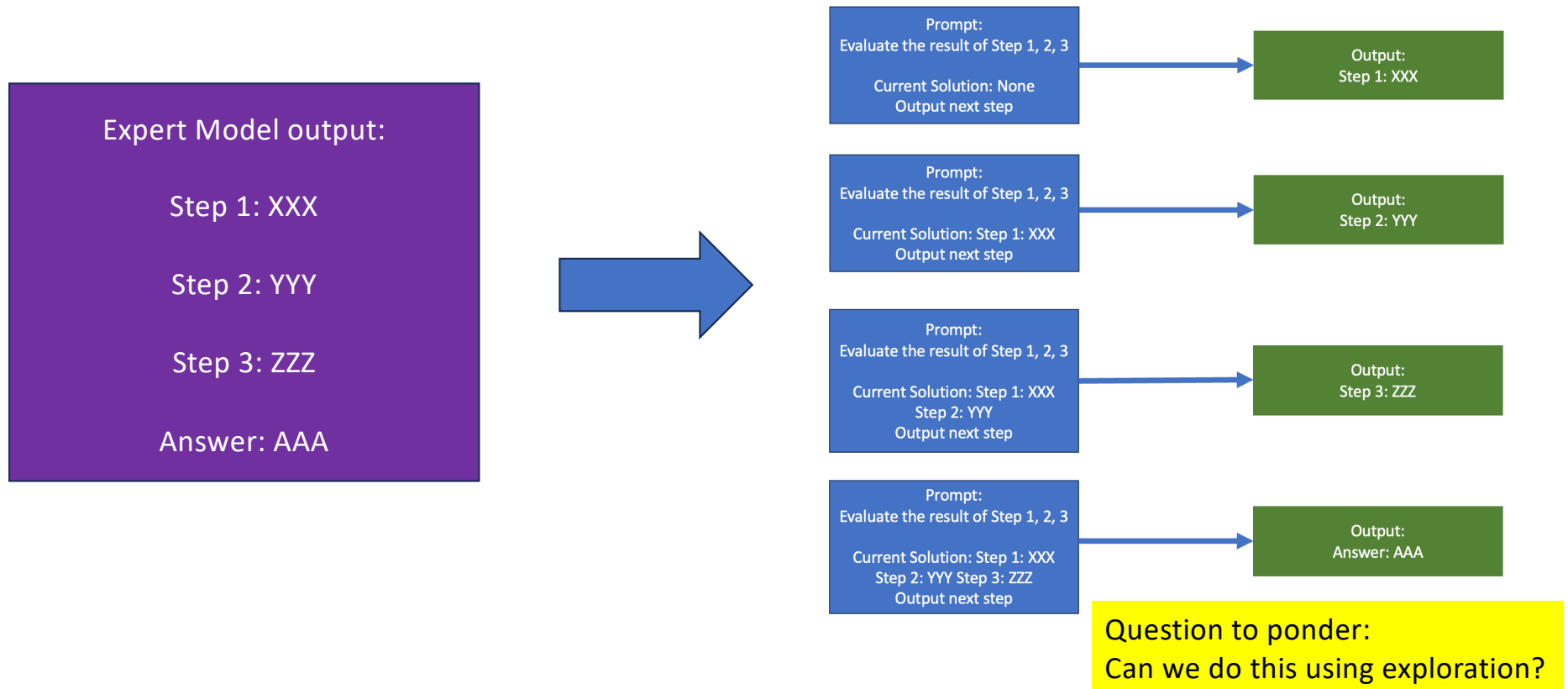
Why not make the problem easier (1/2)?



Why not make the problem easier (2/2)?



Derive the correct steps from expert model



SRL (red) helps to guide the model to achieve better results

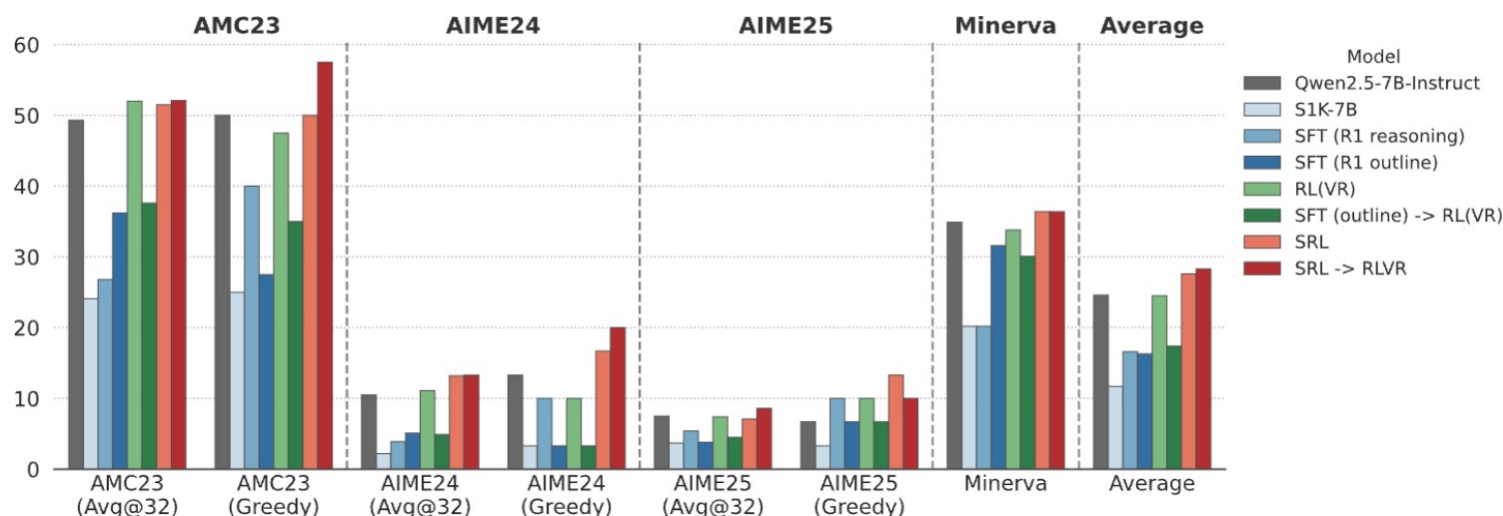


Figure 1 | Performance of our method (SRL) against baselines on math reasoning benchmarks, with all models trained on the challenging s1k dataset (Muennighoff et al., 2025). Our key observations are: (1) Directly applying SFT on this dataset leads to performance degradation compared to the base model. (2) While RLVR can improve generalization over SFT, the gains are marginal. (3) Our proposed SRL method substantially outperforms these baselines, and the SRL \rightarrow RLVR pipeline achieves the highest performance, overcoming the challenges of training on difficult data.

SRL provides more modular rewards

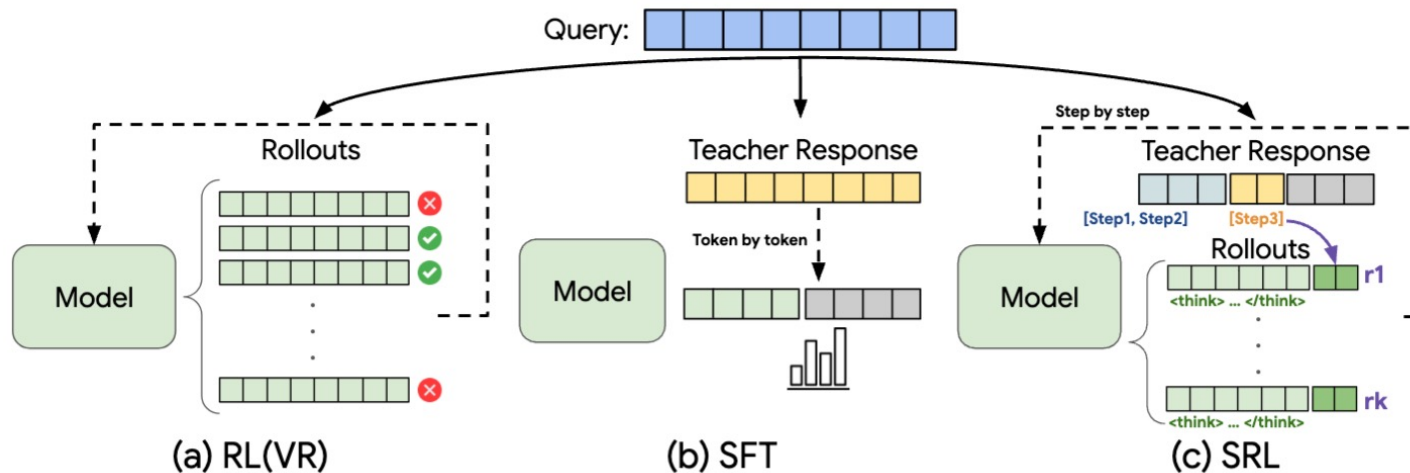


Figure 2 | Illustration of SRL as compared to RL(VR) and SFT. **(a) RL(VR)** takes a query as input and performs k rollouts. The final answer correctness is used as the reward. **(b) SFT** uses both a query \mathbf{x} and a complete teacher response \mathbf{y} as input, training with a per-token loss to maximize the probability $p(\mathbf{y}|\mathbf{x})$. **(c) SRL** also uses a query and a teacher response. It breaks the response into step actions and, at each step, uses the previous steps as context. The model generates a next step action along with its step-wise inner thoughts, and the reward r_k is based on the similarity between the model's and the teacher's action.

Training in Parts (technical stuff)

Action-based problem formulation. Given an expert solution trajectory \mathbf{y} that leads to a correct final answer, we decompose \mathbf{y} into a sequence of tuples: $\mathbf{y} = \{\mathbf{y}_{\text{step}_n}\}_{n=1}^N$. Each step represents a *logical action*: the concrete action to be operated. This formulation is domain-agnostic; for instance, an action in mathematical reasoning could be an algebraic manipulation, while for a software agent, it could be a command executed in a code repository.

Step-wise training data construction. To create training data for SRL, we leverage a powerful teacher model, θ_{expert} to generate solution trajectories. From a single complete solution with N steps, we construct $N - 1$ partial trajectories. For each step $k \in \{1, \dots, N - 1\}$, we create a new input prompt $\mathbf{x}_{\text{step}_k} = [\mathbf{x}, \mathbf{y}_{\text{step}_1}, \dots, \mathbf{y}_{\text{step}_{k-1}}]$, where the model's task is to predict the subsequent step, $\mathbf{y}_{\text{step}_k}$. This process transforms one expert solution into a rich set of training instances that teach the model to proceed correctly from various intermediate states.

Reward based on solution similarity for an action (does not factor in thinking tokens)

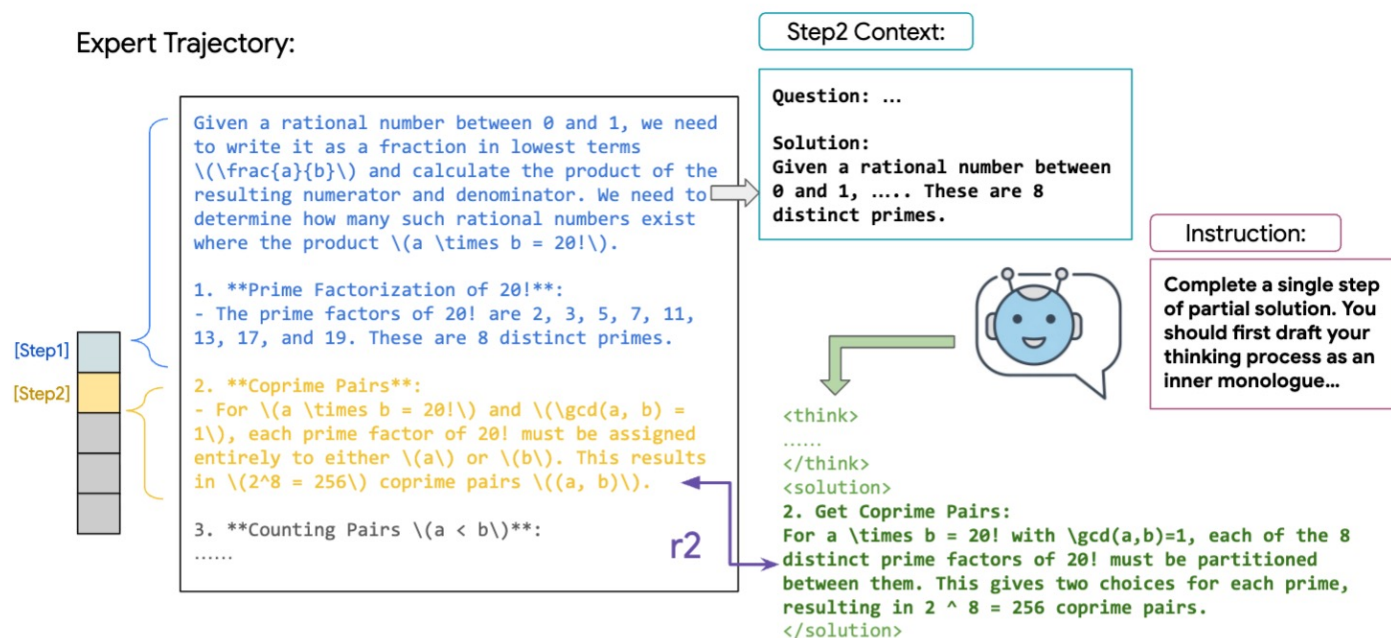


Figure 3 | Given a solution trajectory, we take each summarized step as an action to be learned and take the partial solution before the step as the context of our newly created data. The model is then prompted to generate its thinking process followed by the action for the current step. A reward (r_2 in the figure) is then calculated based on the similarity between the model's and the expert's action.

Reward is based on matching subsequence length

M (Matched elements): The total count of elements found in all non-overlapping matching blocks between the two sequences. The algorithm first finds the **longest contiguous matching subsequence** and then recursively searches for more matches in the segments to the left and right of that block. If we represent the set of all such matching blocks as a list of tuples (i, j, n) , where n is the length of the matching block, then M is the sum of all lengths n : $M = \sum_{(i,j,n) \in \text{MatchingBlocks}} n$.

Combining these definitions, we can calculate the similarity ratio $R \in [0, 1]$ as:

$$R = \frac{2 \sum_{(i,j,n) \in \text{MatchingBlocks}} n}{|S_1| + |S_2|}$$

Question to ponder: What if the solution is similar but does not match in segments? They could be semantically similar but use different terms!

SRL training causes model to have multiple thinking steps natively!

Example 1: Interleaved thinking and self-verification

Question: What is the least positive integer value of x for which the inequality $3x > 2x + 1$ is true?

<think>

To solve the inequality $3x > 2x + 1$, we can start by isolating x . Let's subtract $2x$ from both sides of the inequality.

</think>

Subtract $2x$ from both sides:

$$3x - 2x > 2x + 1 - 2x$$

This simplifies to:

$$x > 1$$

<think>

We can verify this by substituting $x = 2$ into the original inequality to ensure it holds true.

</think>

The inequality $x > 1$ tells us that x must be greater than 1. The smallest positive integer greater than 1 is 2. Therefore, the least positive integer value of x that satisfies the inequality is 2.

Substitute $x = 2$ into the original inequality:

$$3(2) > 2(2) + 1$$

This simplifies to:

$$6 > 5$$

This is true, so $x = 2$ is indeed a solution.

Therefore, the least positive integer value of x for which the inequality $3x > 2x + 1$ is true is 2.

Each step fits nicely as an action output in an agentic structure

I'll help you implement the necessary changes to fix the issue with the `OriginValidator` not properly handling wildcard (*) in allowed_origins. Let's follow the steps you outlined.

Step 1: Find and read relevant code

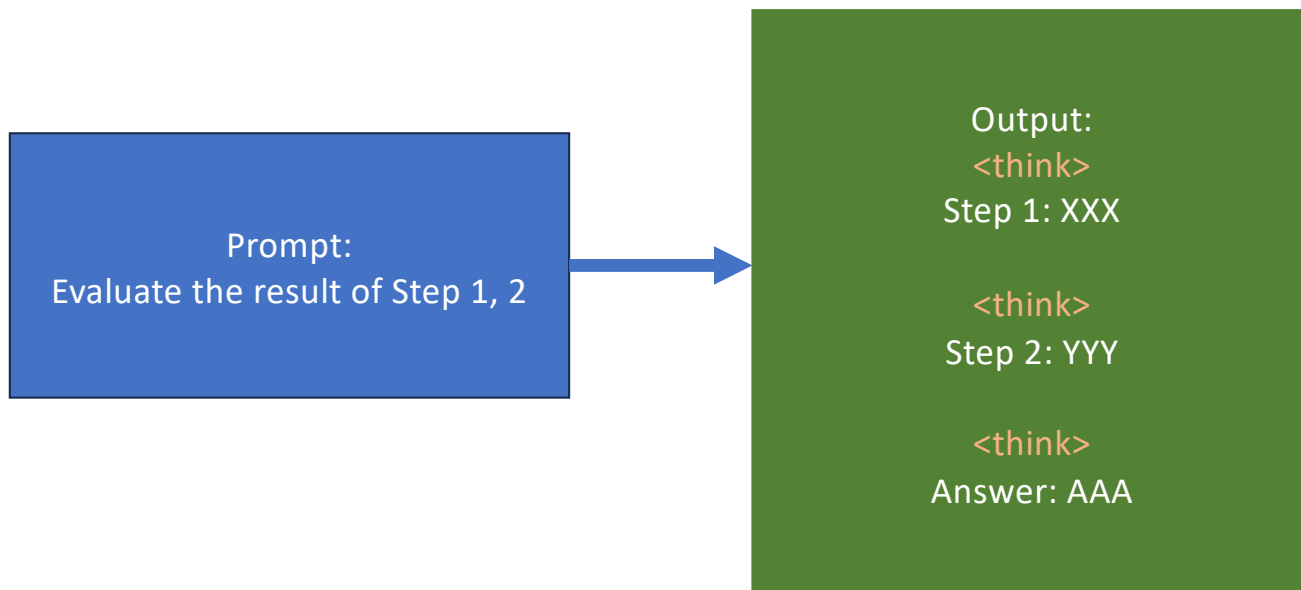
First, let's explore the repository structure to locate the `OriginValidator` class mentioned in the PR description. This is an extra long line added to demonstrate how the automatic line wrapping feature from the listings package works.

```
<function=bash>  
<parameter=command>find /testbed -type f -name "*.py" | grep -v "__pycache__" | sort</parameter>  
</function>
```


My thoughts

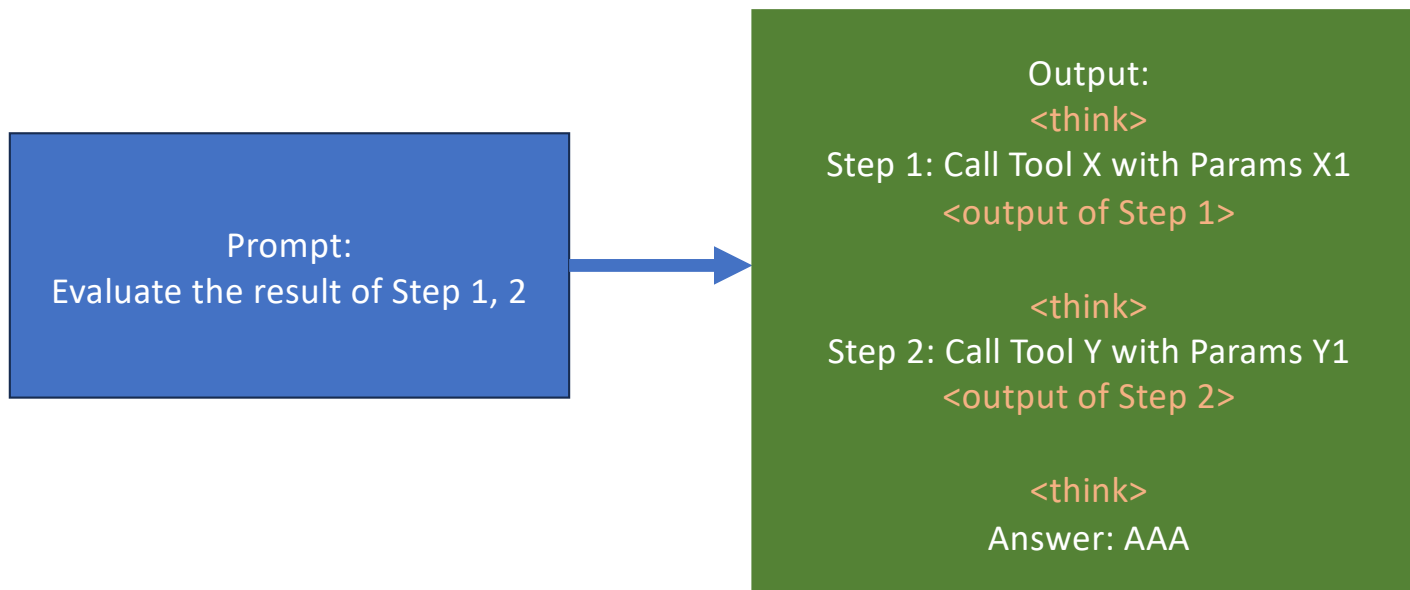
Not training on the thinking tokens is a good thing!

- Sometimes the model can get confused and think they have done a step in the future, hence omitting that step



We should also not train on the output of a tool call

- LLMs may hallucinate the result of a tool call (e.g. google search), which can lead to them skipping the tool call altogether



Other Thoughts

- Many real-world problems do not have an expert to guide the answer
 - Need some exploration techniques to try to get the answer, or have ways to reformulate the problem to a subproblem for easier solutions
- Many real-world problems also often have dynamically changing solutions
 - Instead of using RL-based reward of completion or no completion, reframing the problem as an output matching task for the action is a good idea
 - We can match the next action based on what has actually happened (see Goal-Directed Learning next slide)

Learning from Own Trajectory (Self-Supervised Action Learning)



Problem: Get from S1 to S3
Next Action: A1

Problem: Get from S1 to S2
Next Action: A1

Problem: Get from S2 to S3
Next Action: A2

Goal-Directed Learning

- At each time step, learns from:
 - **Previous states replay**
 - **Future states replay (only if lookahead trajectory found)**
- Intuition:
 - If we have a trajectory $A \rightarrow B \rightarrow C$
 - We know $A \rightarrow B$, $A \rightarrow C$, $B \rightarrow C$
 - Maximise learning from experience/lookahead

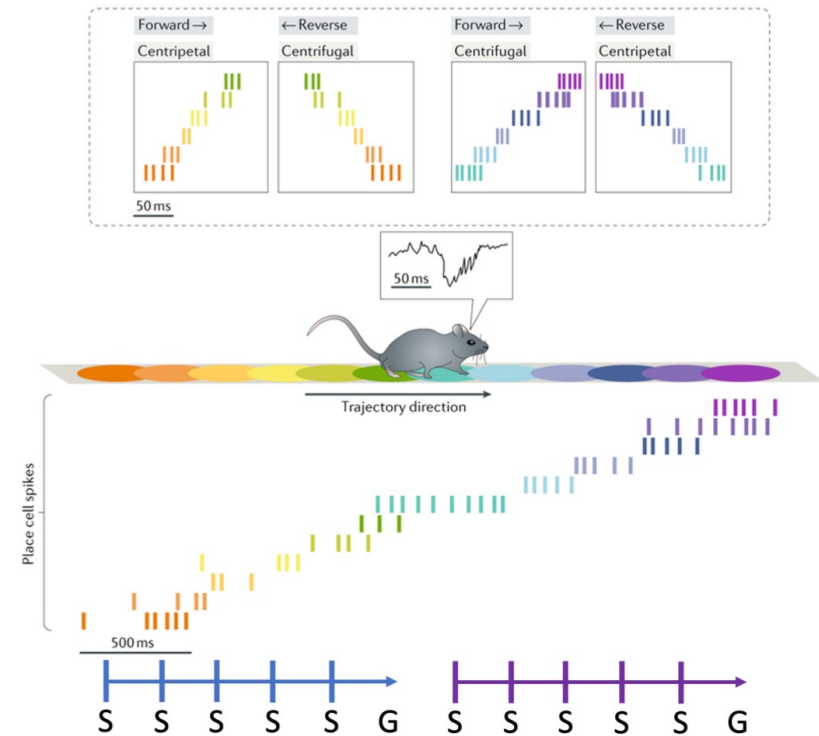
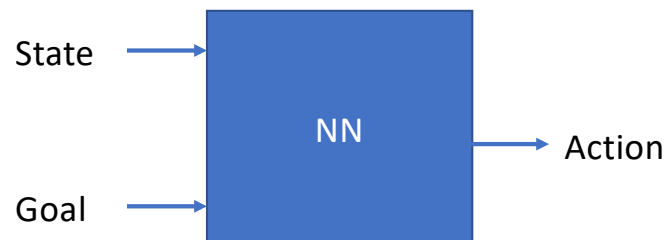


Figure extracted from Joo, H. R., & Frank, L. M. (2018). The hippocampal sharp wave-ripple in memory retrieval for immediate use and consolidation. *Nature reviews. Neuroscience*, 19(12), 744–757. <https://doi.org/10.1038/s41583-018-0077-1>

Question to Ponder

- Should the reward factor in the thinking tokens as well? How about the tool output tokens?
- Is the reward based on matching subsequence a little too restricted? Can it detect semantically similar answers with different wording?
- Should we train the LLM to do multiple steps of reasoning natively, or use a reasoning framework over the LLM?
- How can we integrate goal-directed learning and exploration into LLM training?