CS-461

Foundation Models and Generative AI

Adaptation, Fine-Tuning, and Test-Time Training

Jonas Hübotter and Andreas Krause, Fall Semester 2025/26

Our aim for today

Typically two different regimes:

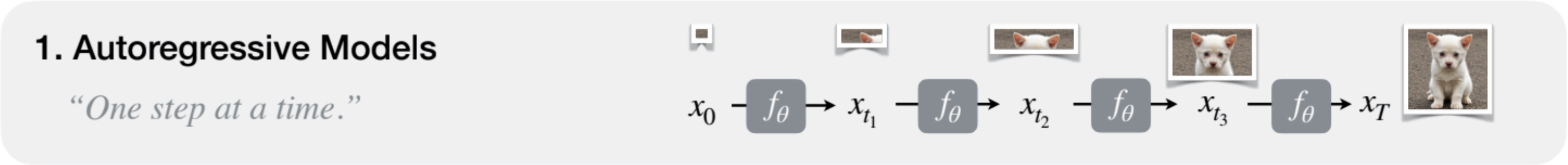
- **train-time**: foundation model is trained on a (wide) distribution of tasks
- **test-time**: foundation model is given a particular task to solve

we'll focus on task-specific learning

Task-specific learning today:

1. models are "manually" fine-tuned to a (narrow) distribution of downstream tasks (but then kept static)
2. models learn from context, but only over very short horizons

- We will focus on autoregressive models



- **Our aim**: "adding memory" to enable extensive learning at test-time

❶ Scaling attention to long sequences

Scaling attention to long sequences

Key concept: **Self-Attention**

- When autoregressively generating $p(x_t \mid x_{<t})$, self-attention "attends" to patterns **(values)** $V_{<t}$ in previous tokens by matching the current **query** q_t with previous **keys** $K_{<t}$

$$\text{Attention}(q_t; K_{<t}, V_{<t}) = \text{softmax}\left(\frac{q_t K_{<t}^{\top}}{\sqrt{d_K}}\right) V_{<t}$$

Legend

$k_t = \theta_K x_t$

$q_t = \theta_Q x_t$

$v_t = \theta_V x_t$

very slow!

- Naively computing $K_{<t}, V_{<t}$ at every step has $O(T^2)$ per-step latency

The solution: **KV Caching**

- Compute only the **new** q_t, k_t, v_t from x_t
- Append k_t to the cached $K_{<t}$ (in VRAM) to form the new $K_{\leq t}$

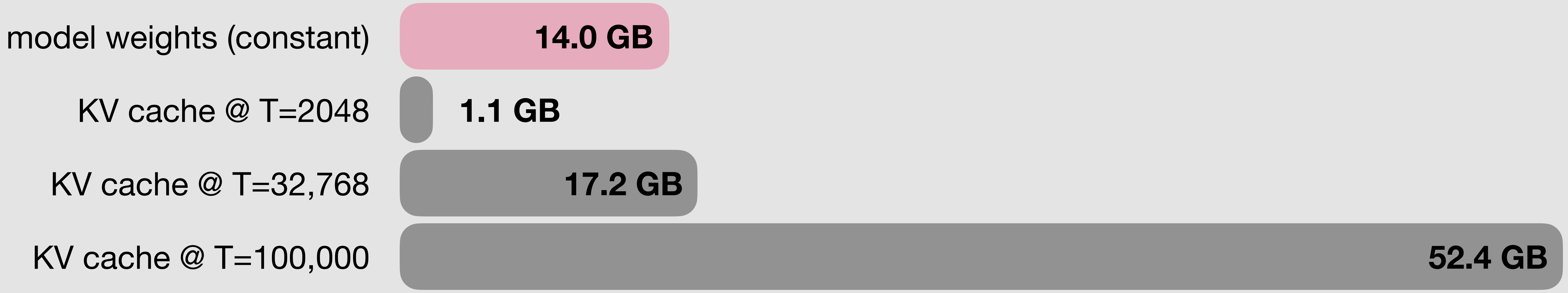
per-step latency is $O(1)$ with parallel key-lookup

The memory bottleneck

The problem: **KV cache size** grows linearly with sequence length T

- For long sequences, the memory required for the KV cache *quickly* exceeds the size of model!

Example (Llama 7B): 32 layers, hidden dimension 4096, 16-bit precision



At 32k context, the KV cache is already larger than model weights!

- At large sequences, the KV cache dominates VRAM → long-context is **memory-bound**!

The memory view of transformers

The KV cache in a transformer is a type of **memory**, but a memory that grows with time

- Let's look once more at attention…

$$\text{Attention}(q_t; K_{<t}, V_{<t}) = \text{softmax}\left(q_t K_{<t}^\top\right) V_{<t} = \sum_{s=1}^{t} w_s v_s, \quad w_s \propto e^{k_s^\top q_t} \qquad (d_K = 1)$$

- We can think of "attention" as a **memory** that can be learned from "dataset" $\{(k_s, v_s)\}_{s=1}^{t}$!

attention prescribes a particular way of estimating a memory!

Background: Kernel regression

A standard estimator from statistics is the Nadaraya-Watson estimator:

$$\hat{y}(x) = \sum_{i=1}^{n} w_i y_i, \quad w_i \propto k(x, x_i)$$

Examples:

- Nearest neighbor estimation: k is a one-hot encoding of neighborhoods
- Gaussian "RBF" kernel: $k(x, x_i) = e^{-\|x - x_i\|_2^2}$

→ self-attention is kernel regression with kernel $k(q_t, k_s) = e^{k_s^\top q_t}$

exercise: equivalent to rbf kernel for normalized queries & keys

Beyond the memory bottleneck

We can think of the attended past values $V_{<t}$ based on query q_t as a quantity to be learned:

$$\mathrm{Memory}(q_t; K_{<t}, V_{<t})$$

Two kinds of memories

uses all seen data for each prediction

Non-parametric estimates of $\mathrm{Memory}(q_t; \underbrace{K_{<t}, V_{<t}}_{\text{“dataset”}})$

- needs to store & access all data
- example: self-attention

Parametric models of $\mathrm{Memory}(q_t; \overbrace{W_t}^{\text{“weights”}})$ maintains a finite memory

- parameterizes memory as a learnable model of a finite size
- example: **linear attention**

Linear attention

Consider memory as a linear model: $\mathrm{Memory}(q_t; W) = W\, q_t$

- All previous values are compressed into the memory ("weights" / "state") W

Training: $\ell(W; x_t) = \frac{1}{2}\|\mathrm{Memory}(k_t; W) - v_t\|_2^2 = \frac{1}{2}\|W k_t - v_t\|_2^2$ self-supervised reconstruction loss

$$\nabla_W \ell(W; x_t) = (Wk_t - v_t)k_t^\top$$

$$\nabla_W \ell(W_0; x_t) = -\, v_t k_t^\top \quad (W_0 = 0)$$

batched gradient descent

$$W_t = W_0 - \eta \sum_{s=1}^{t} [-v_s k_s^\top] = \sum_{s=1}^{t} v_s k_s^\top \quad (\eta = 1)$$

"write"

note: state compresses keys & values

$$\mathrm{LinearAttention}(q_t; K_{<t}, V_{<t}) = V_{<t} K_{<t}^\top q_t = W_{t-1} q_t$$

"read"

why is this efficient?

- **Compute**: State updates in $O(1)$ $W_t = W_{t-1} + v_t k_t^\top$
- **Memory**: State consumes $O(1)$ since we only need to store W_t

Duality of linear attention

Linear attention can equivalently be derived as

- a **parametric** memory (compressing data into a fixed-size state)
- a **non-parametric** memory (keeping all data)

parametric

non-parametric

$$W_t q_t = \sum_{s=1}^{t} (k_s v_s^\top)\, q_t \quad \Longleftrightarrow \quad \sum_{s=1}^{t} (k_s^\top q_t)\, v_s$$

storing $d_K \times d_V$ matrix

computing $t \times t$ matrix

Fast & slow weights → test-time training

Sequence models learn at two frequencies:

- During inference a model learns a memory, either a growing cache or a parametric memory W
- During training a model learns parameters θ

→ Parameters θ learn *how to update the memory* along a sequence $x_{1:t}$

This is an example of **meta-learning**! "learning to learn"

- at test-time, an inner loop updates "fast weights" W
- at train-time, an outer loop learns "slow weights" θ that improve the inner loop

"inner" dataset $\{(k_s, v_s)\}_{s=1}^{t}$

"outer" dataset $\{(\mathbf{x}_i)\}_{i=1}^{n}$

Updating "fast weights" W in an inner loop with gradient descent is called **test-time training (TTT)**

Remember: meta-learning is about learning how to learn more efficiently

example: linear attention

Extensions of linear attention

Recall

$$\mathrm{LinearAttention}(q_t; K_{<t}, V_{<t}) = V_{<t}K_{<t}^\top q_t = \sum_{s=1}^{t} (k_s^\top q_s)\, v_s = W_t q_t$$

We can design alternative memory models!

Learning rule: Hebbian vs Delta

- Batched gradient descent (*linear attention*)

  $W_t = W_{t-1} + v_t k_t^\top$ "neurons that fire together, wire together"

  → can lead to "memory overflow"

- Online gradient descent

  $W_t = W_{t-1}(I - \eta k_t k_t^\top) + \eta v_t k_t^\top$

  "edit / overwrite" instead of just "add"

$$\nabla_W \ell(W; x_t) = (Wk_t - v_t)k_t^\top$$

$$W_t = W_{t-1} - \eta \nabla_W \ell(W_{t-1}; x_t)$$

$$W_t = W_{t-1} - \eta (W_{t-1}k_t - v_t)k_t^\top$$

$$W_t = W_{t-1}(I - \eta k_t k_t^\top) + \eta v_t k_t^\top$$

→ called the Delta rule

& used in DeltaNet / RWKV-7

Extensions of linear attention

Recall

$$\text{LinearAttention}(q_t; K_{<t}, V_{<t}) = V_{<t}K_{<t}^\top q_t = \sum_{s=1}^{t} (k_s^\top q_s)\, v_s = W_t q_t$$

We can design alternative memory models!

Learning rule: Hebbian vs Delta

- Batched gradient descent (*linear attention*)

  $$W_t = W_{t-1} + v_t k_t^\top$$

  "neurons that fire together, wire together"

  → can lead to "memory overflow"

- Online gradient descent (DeltaNet)

  $$W_t = W_{t-1}(I - \eta k_t k_t^\top) + \eta v_t k_t^\top$$

  "edit / overwrite" instead of just "add"

Forgetting: slowly forget "old" data

$$W_t = \underbrace{\text{diag}(\alpha_t)}_{\text{"weight decay"}} W_{t-1} - \eta \nabla \ell(W_{t-1}; x_t)$$

e.g., RWKV-7

Momentum:

$$W_t = W_{t-1} + S_t$$

$$S_t = \beta \underbrace{S_{t-1}}_{\text{"past surprise"}} - \eta \underbrace{\nabla \ell(W_{t-1}; x_t)}_{\text{"momentary surprise"}}$$

e.g., Titans

Summary of test-time training (so far)

Let's remember our goal: Efficient & sufficiently expressive memory to solve the test-time task

We have seen:

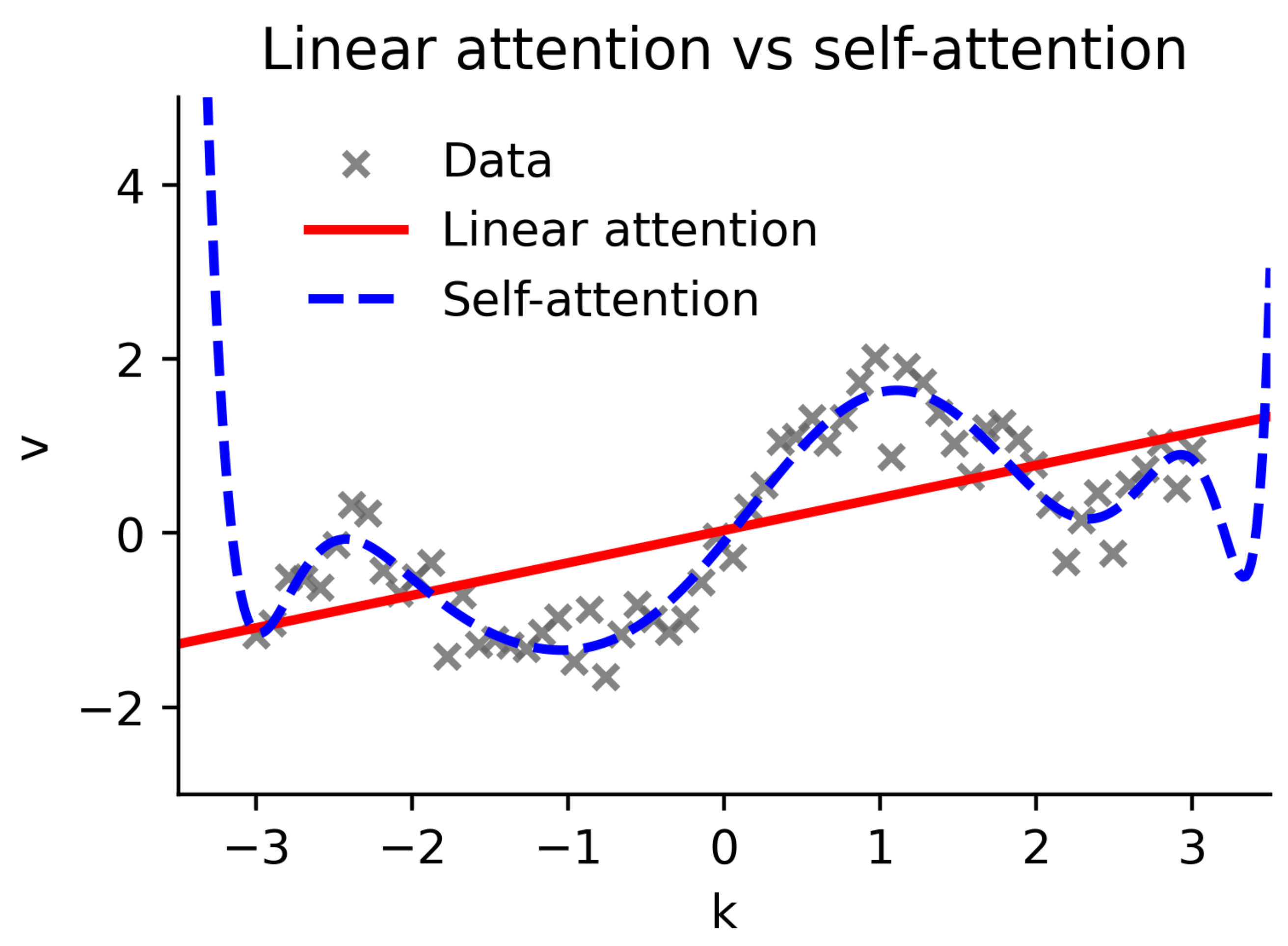
- Transformers / self-attention model memory as a **non-parametric** kernel regression
- Test-time training models memory as a **parametric** regression
  - simplest example: linear attention with a linear memory model

A 1d example

→ linear attention has limited expressivity

→ self-attention can struggle with generalization

→ self-attention is computationally inefficient

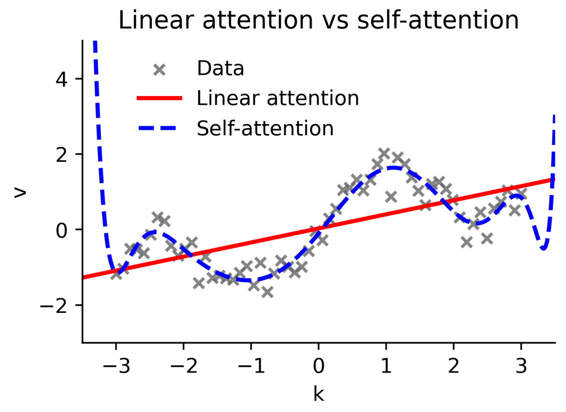


Summary of test-time training (so far)

Let's remember our goal: Efficient & sufficiently expressive memory to solve the test-time task

We have seen:

- Transformers / self-attention model memory as a **non-parametric** kernel regression
- Test-time training models memory as a **parametric** regression
  - simplest example: linear attention with a linear memory model



Over the past decade in ML, *deep* parametric models have efficiently learned complex patterns.

Non-parametric learning has not scaled beyond small datasets.

Key question: If we want to meta-learn models θ that learn to solve complex tasks at test-time, should their memory also be a *deep* parametric model?

Outlook: Open questions

Which "fast weights"?

- separate from θ (linear / deep)
- "fast weights" W = "slow weights" θ
  - or low-rank adapters of θ
- KV cache prefix 📝

$\underbrace{(k_1, v_1), \ldots, (k_t, v_t)}_{\text{KV cache}}, (k_{t+1}, v_{t+1})$

$\underbrace{(z_1, z_1'), \ldots, (z_d, z_d')}_{\text{trainable KV prefix of size } d}, (k_{t+1}, v_{t+1})$

Which loss?

- self-supervised reconstruction loss
  - at the current token: $\ell(W; x_i)$
  - across all previous tokens 📝
- other (self-)supervised losses (next part)
- context distillation 📝

behavior with kv cache is distilled into memory

Challenge: Parallel training

During inference TTT is efficient compared to self-attention

inference is inherently sequential!

BUT fast GPU training requires parallelization!

- Self-attention has no sequential dependency, but the **attention matrix** $QK^\top$ takes $O(T^2)$ space

  → fast training if attention matrices fit onto GPU

- TTT has a sequential dependency $W_1, W_2, \ldots, W_t$! How can we pre-train on a sequence in parallel?

Option 1: Parallel scan with linear attention

- Goal: $[x_1, x_2, x_3, x_4] \rightarrow [x_1, x_1 \oplus x_2, x_1 \ldots \oplus x_3, x_1 \ldots \oplus x_4]$
- Step 1: each element adds value from 1 pos to its left
  - $[x_1, x_2 \oplus x_1, x_3 \oplus x_2, x_4 \oplus x_3]$

  any associative operation

- Step 2: each element adds value from 2 pos to its left
  - $[x_1, x_1 \oplus x_2, x_1 \ldots \oplus x_3, x_1 \ldots \oplus x_4]$
- Completes in $\log_2(T)$ parallel steps for sequence length T

generalizes only to associative updates like the delta rule

Option 2: Large chunks of TTT 📝

- Keep memory "weights" W_t fixed across large chunk of sequence (like 4k tokens)

  "windowed" attention

- Within a chunk, use a KV cache restricted to the chunk
  - **low-level** KV cache + **high-level** memory
- Each chunk can be processed in parallel
  - can adjust chunk/memory size for maximum throughput

generalizes to arbitrary parametric memory!

Summary

- Self-attention (aka transformers) perform **non-parametric** learning at test-time
  → *memory bottleneck* when scaling to learning over long sequences!

- Test-time training (TTT) avoids the *memory bottleneck* by training a **parametric** model at test-time
  - Linear attention is the simplest example where the memory model is linear

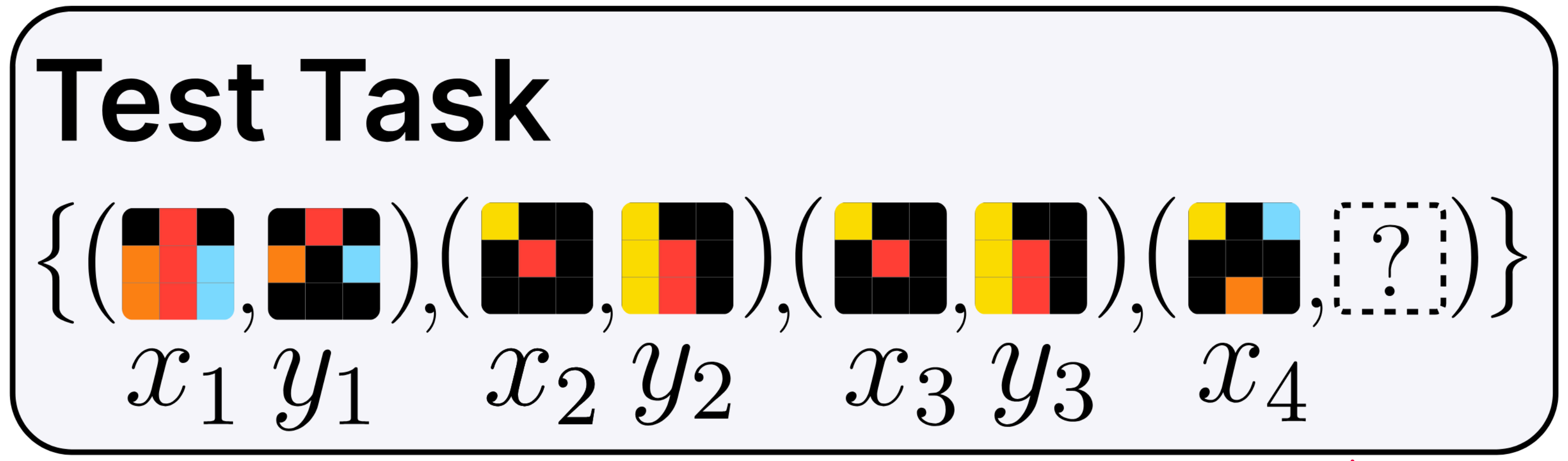
- While TTT avoids the memory bottleneck, training cannot generally be parallelized
  - → combination with self-attention through chunked TTT

Example: Few-shot learning

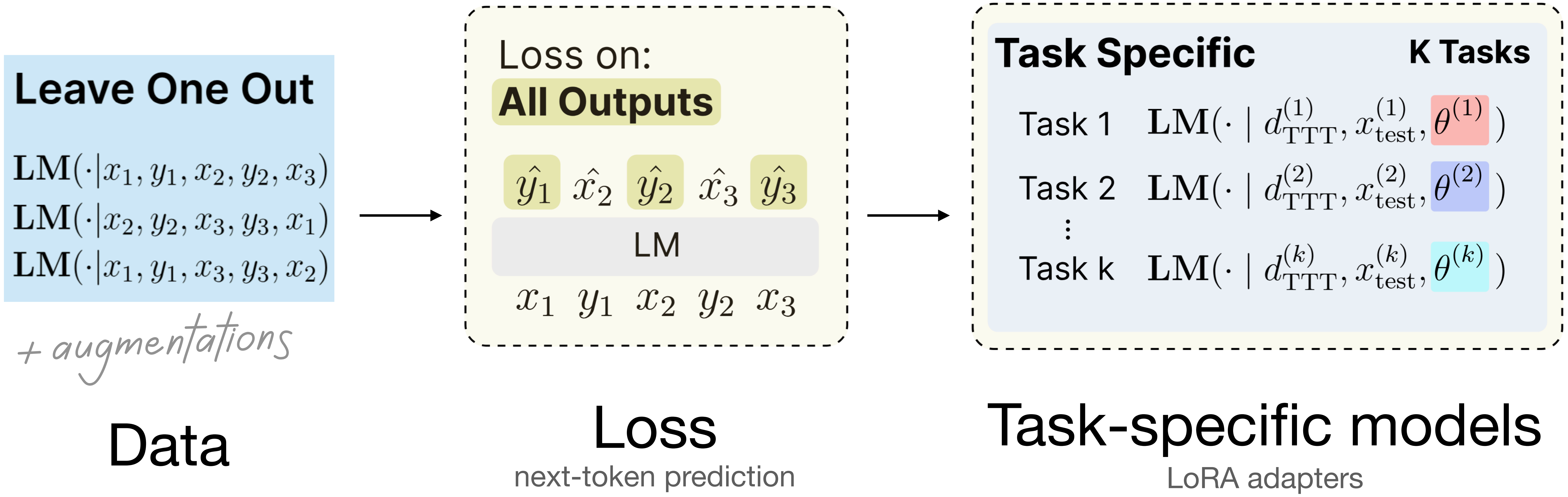
Sequences often have additional structure, for example:

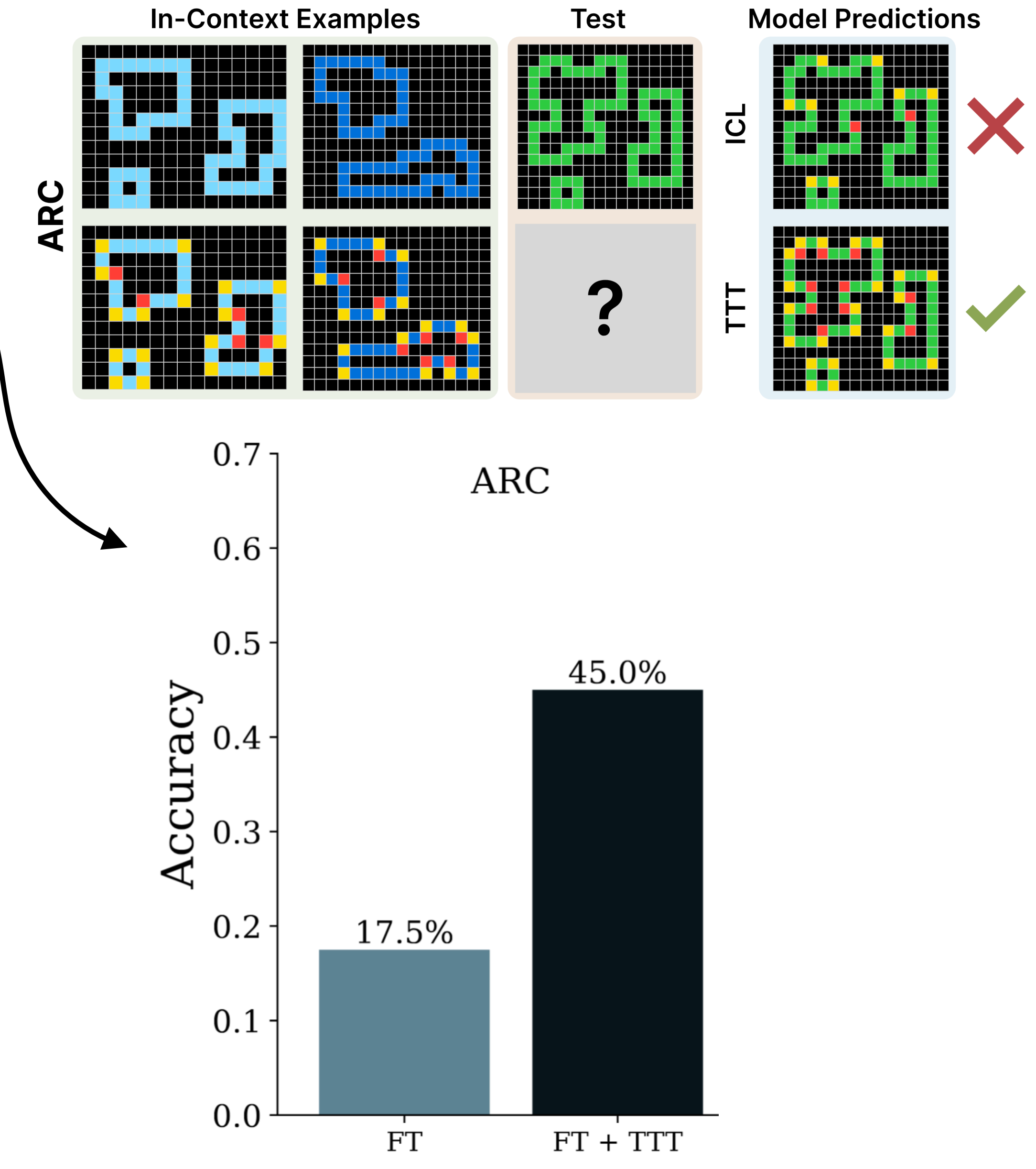
- prompt-response: $x^\star, y^\star$
- **few-shot demonstrations**: $x_1, y_1, x_2, y_2, \ldots, x_t, y_t, x^\star, y^\star$

A test-time training pipeline for few-shot learning:



task is only revealed by examples!





on top of Llama3 8B

Perspective: Meta-learning in a few-shot setting

Previously, we learned "slow weights" θ that lead to good "fast weights" θ' at test-time

We can do the same here!

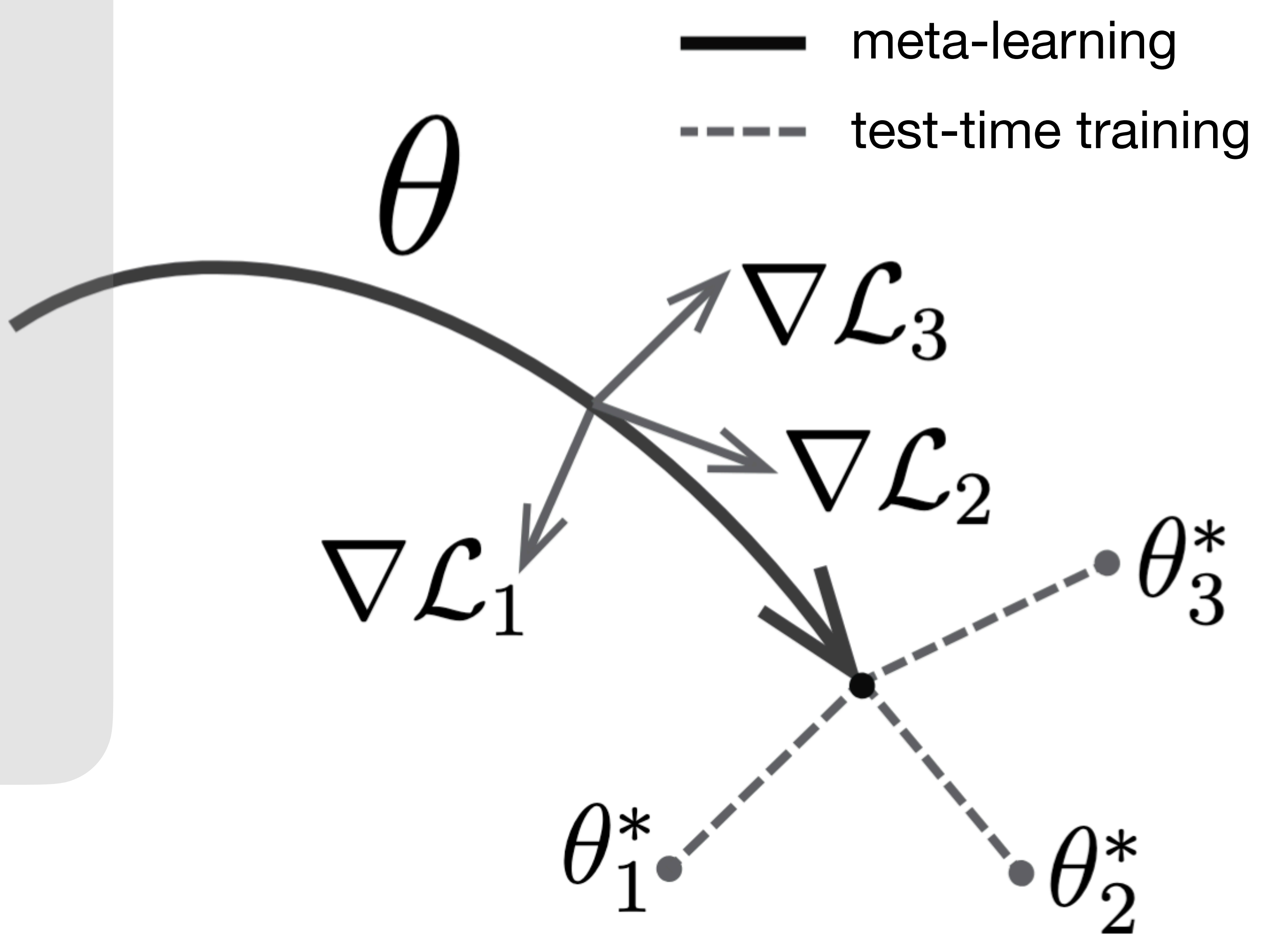
Canonical example: model-agnostic meta learning (**MAML**)

- The inner loop optimizes a loss on the few-shot examples

$$\theta'_{x^\star} = \theta - \alpha \nabla_\theta \sum_{i=1}^{k} \ell(\theta; x_i, y_i)$$

- The outer loop finds an initialization θ leading to small loss after the inner loop (on average across $x^\star$)

$$\theta \leftarrow \theta - \beta \nabla_\theta \sum_{x^\star} \ell(\theta'_{x^\star}; x^\star, y^\star)$$



2 What data to learn from at test-time?

What data to learn from at test-time?

Recap

- TTT enables learning across long sequences at test-time
- Few-shot learning is a setting where task-specific data is given at test-time: $x_1, y_1, \ldots, x_t, y_t, x^\star$

think: x_s = key, y_s = value

If task-specific data is not given to us: Can we acquire it from existing datasets?

- **Goal**: Find data $x_1, \ldots, x_t$ such that prediction error at $x^\star$ is minimized

Test instance $x^\star$

Training data $\{(x_s, y_s)\}_{s=1}^t$ ⟶ Learnt model $f : \mathcal{X} \to \mathcal{Y}$ ⟶ Prediction $f(x^\star)$

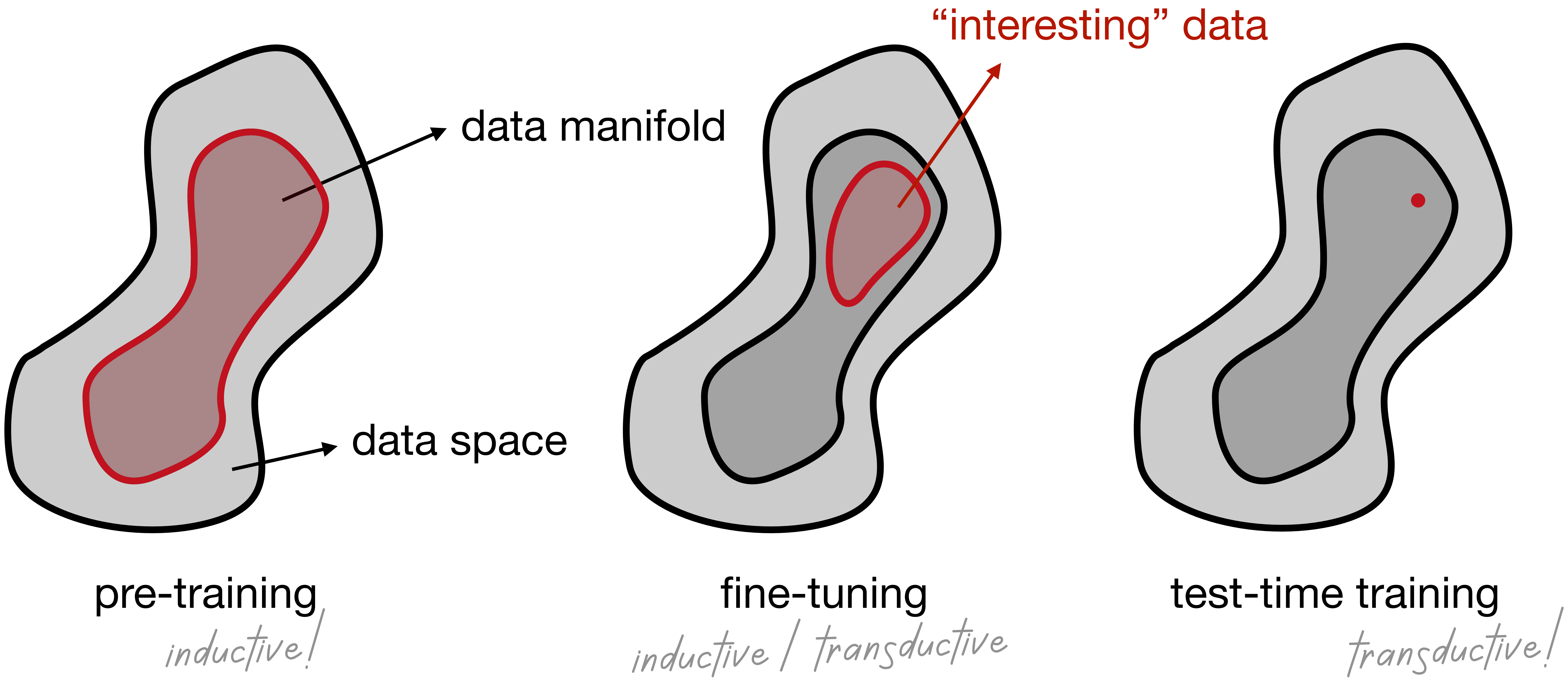
putting the model in the decision-making loop!

Perspective: Induction vs Transduction

Inductive learning: extract general rules from data

Transductive learning: learn only what you need

"When solving a problem of interest, do not solve a more general problem as an intermediate step. Try to get the answer that you really need but not a more general one." —*Vladimir Vapnik (80's)*



A simple probabilistic model

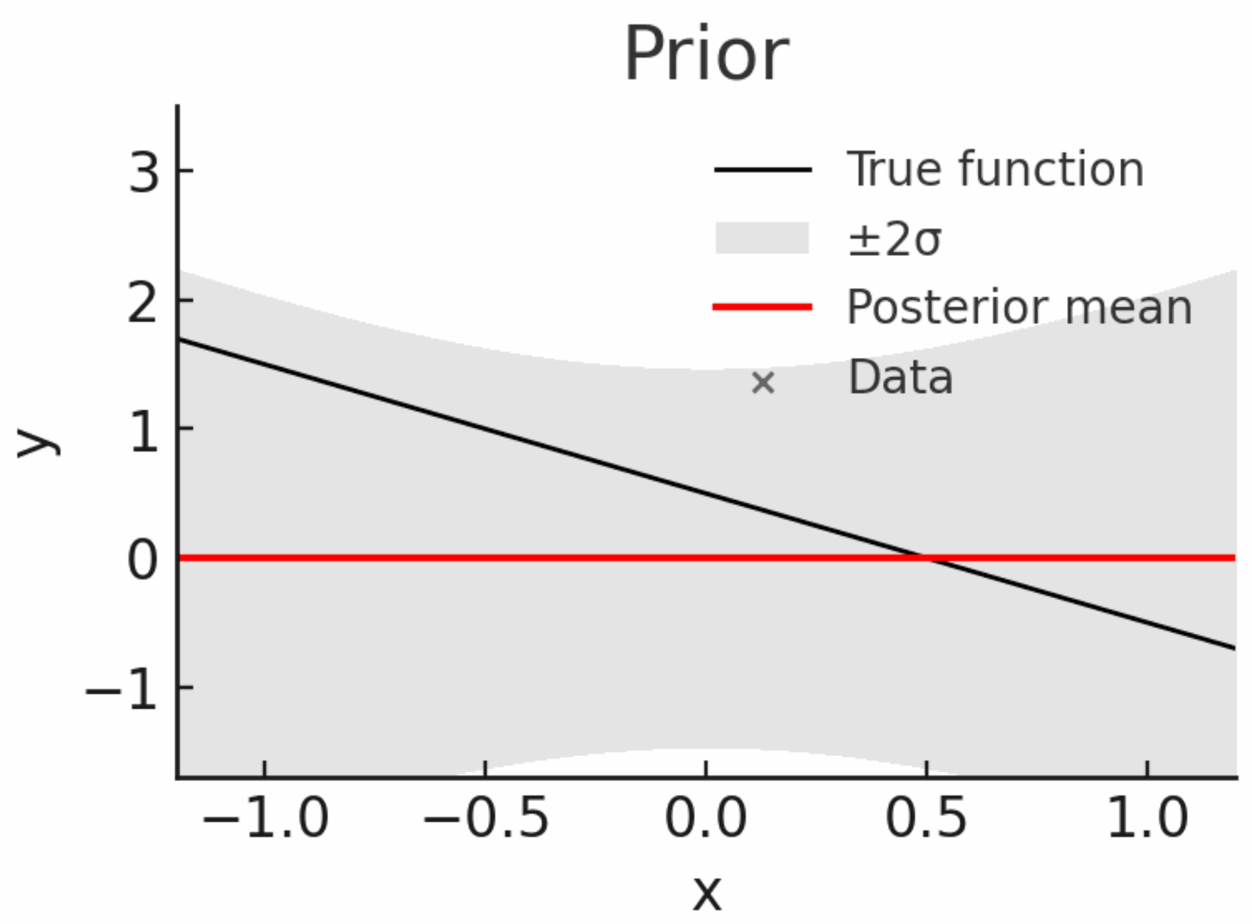
NeurIPS ‘24

Can we acquire task-specific examples from existing datasets?

Model

- **linear model**: assume $f(x) = \phi(x)^\top w$
- **Gaussian prior**: let $w \sim \mathcal{N}(0,I)$
- **Gaussian noise**: let $y_s = f(x_s) + \varepsilon_s$, with $\varepsilon_s \sim \mathcal{N}(0,\lambda)$

Bayesian linear regression



Goal: Retrieve $x_1, \ldots, x_t$ such that the predictive uncertainty $\mathrm{Var}(f(x^\star) \mid x_{1:t}, y_{1:t})$ is minimal

- Would be optimal to retrieve $x^\star$, but $f(x^\star)$ is typically not within the dataset

Corresponds to a regularized TTT

$$w_t = \mathrm{argmin}_w \sum_{s=1}^{t} (\phi(x_s)^\top w - y_s)^2 + \frac{\lambda}{2}\|w\|_2^2$$

$$\mathbb{E}[f(x^\star) \mid x_{1:t}, y_{1:t}] = w_t$$

Predictive uncertainty

Model

- **linear model**: assume $f(x) = \phi(x)^\top w$
- **Gaussian prior**: let $w \sim \mathcal{N}(0, I)$
- **Gaussian noise**: let $y_s = f(x_s) + \varepsilon_s$, with $\varepsilon_s \sim \mathcal{N}(0, \lambda)$

Goal: Retrieve $x_1, \ldots, x_t$ such that $\mathrm{Var}(f(x^\star) \mid x_{1:t}, y_{1:t})$ is minimal

$$\mathrm{Var}(f^\star \mid x^\star) = \phi(x^\star)^\top \mathrm{Var}(w) \phi(x^\star) = \phi(x^\star)^\top \phi(x^\star)$$

$$\Phi = \begin{bmatrix} \phi(x_1) \\ \vdots \\ \phi(x_t) \end{bmatrix} \qquad y_{1:t} = \Phi w + \varepsilon_{1:t}$$

$$\begin{bmatrix} y_{1:t} \\ f^\star \end{bmatrix} \mid x_{1:t}, x^\star \sim \mathcal{N}\left(0, \begin{bmatrix} \Phi\Phi^\top + \lambda I & \Phi\phi(x^\star) \\ (\Phi\phi(x^\star))^\top & \phi(x^\star)^\top \phi(x^\star) \end{bmatrix}\right)$$

$$\mathrm{Var}(f^\star \mid x_{1:t}, y_{1:t}, x^\star) = \phi(x^\star)^\top \phi(x^\star) - \phi(x^\star)^\top \Phi^\top (\Phi\Phi^\top + \lambda I)^{-1} \Phi\phi(x^\star)$$

Minimizing predictive uncertainty

Goal: Retrieve $x_1, \ldots, x_t$ such that $\mathrm{Var}(f(x^\star) \mid x_{1:t}, y_{1:t})$ is minimal

$$\mathrm{Var}(f^\star \mid x_{1:t}, y_{1:t}, x^\star) = \phi(x^\star)^\top \phi(x^\star) - \phi(x^\star)^\top \Phi^\top (\Phi\Phi^\top + \lambda I)^{-1} \Phi \phi(x^\star)$$

But: Combinatorial optimization over t variables!

Can minimize greedily (one-by-one):

$$x_1 = \mathrm{argmin}_x \, \mathrm{Var}(f^\star \mid x_1, y_1, x^\star) = \mathrm{argmax}_x \frac{(\phi(x^\star)^\top \phi(x))^2}{\|\phi(x)\|_2^2 + \lambda} \underset{\|\phi(x)\|_2^2 = 1}{=} \mathrm{argmax}_x \left(\measuredangle_\phi(x^\star, x)\right)^2$$

nearest neighbor retrieval!

$$x_2 = \mathrm{argmin}_x \, \mathrm{Var}(f^\star \mid x_{1,2}, y_{1,2}, x^\star) = \mathrm{argmax}_x \begin{bmatrix} \measuredangle_\phi(x^\star, x_1) \\ \measuredangle_\phi(x^\star, x) \end{bmatrix}^\top \begin{bmatrix} \measuredangle_\phi(x_1, x_1) + \lambda & \measuredangle_\phi(x, x_1) \\ \measuredangle_\phi(x_1, x) & \measuredangle_\phi(x, x) + \lambda \end{bmatrix}^{-1} \begin{bmatrix} \measuredangle_\phi(x^\star, x_1) \\ \measuredangle_\phi(x^\star, x) \end{bmatrix}$$

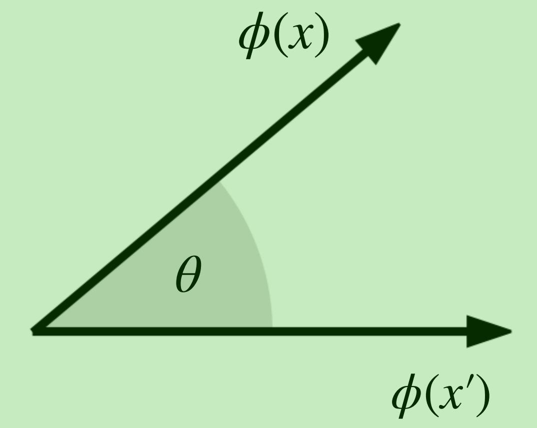
similarity to $x^\star$

diversity of $x_{1:t}$

$\lambda \to \infty$: pick nearest neighbor

$\lambda \to 0$: more diverse $x_{1:t}$

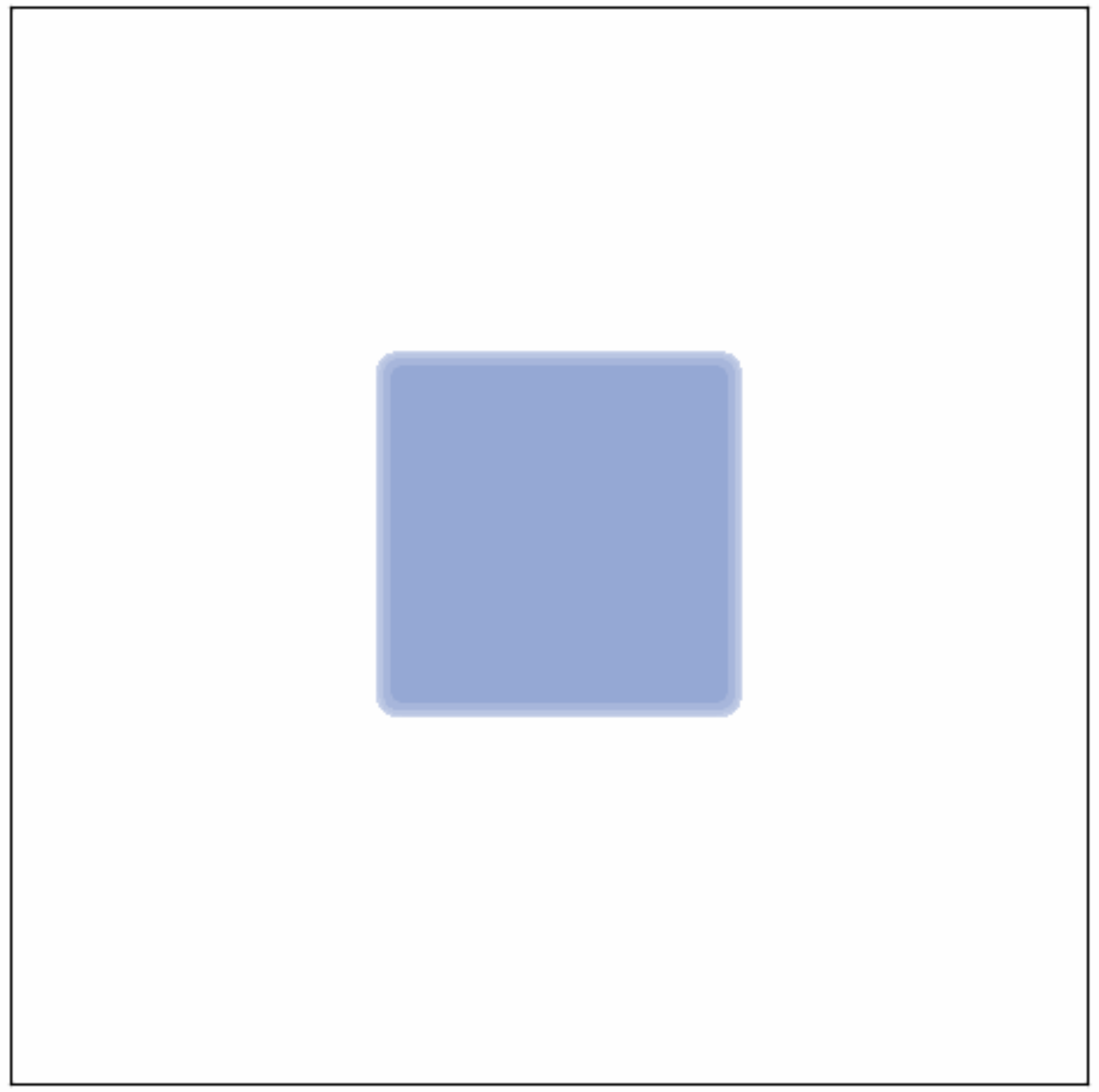
Cosine similarity



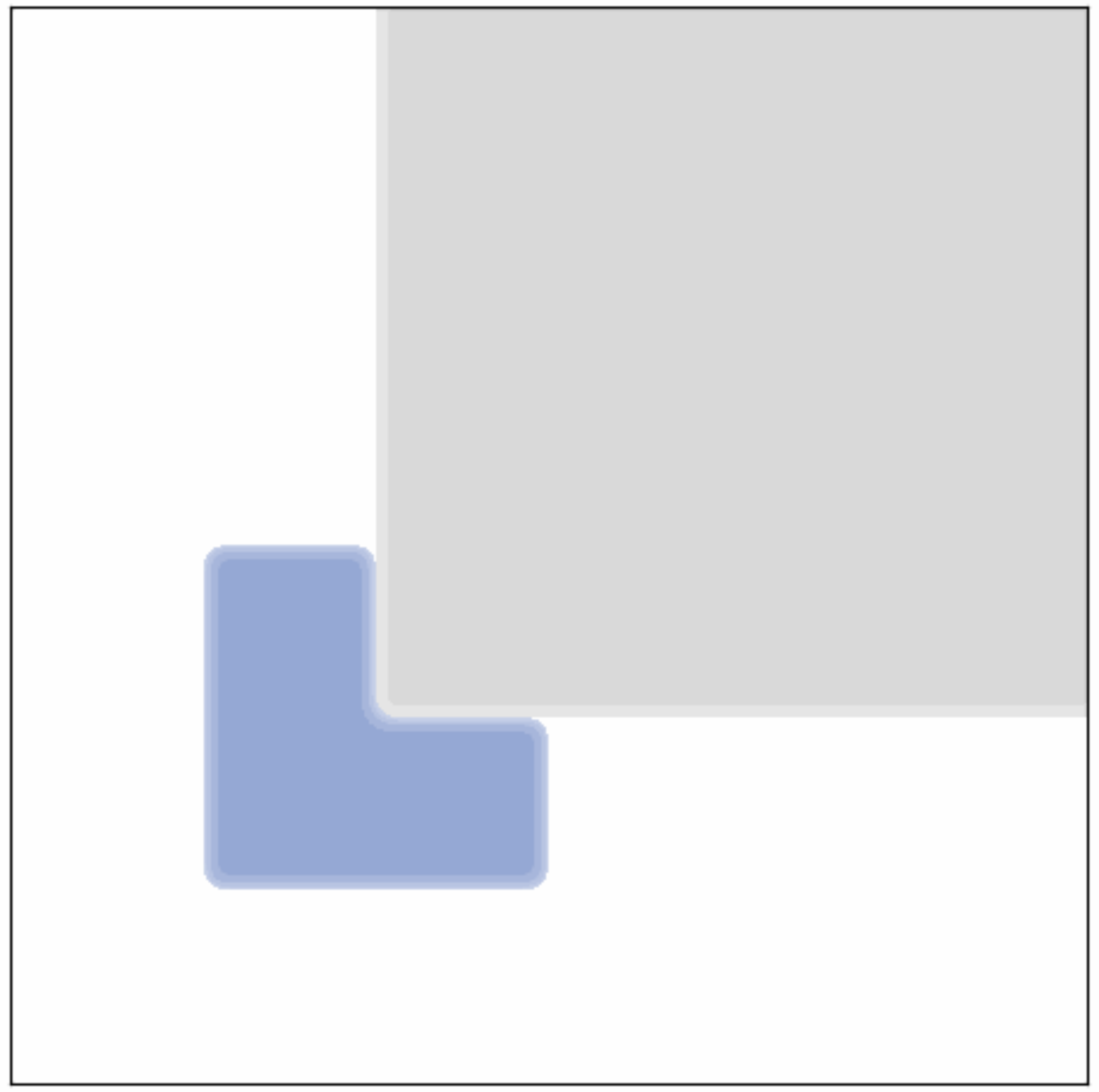
$$\measuredangle_\phi(x, x') = \frac{\phi(x)^\top \phi(x')}{\|\phi(x)\| \|\phi(x')\|} = \cos\theta$$

Visualization of transductive active learning

Example: Selecting data where features ϕ correspond to the RBF kernel (Euclidean similarity)



Blue: prediction targets



Blue: prediction targets
Gray: sample space

Summary: Learning from retrieved examples

We have seen: how to retrieve examples for learning with *linear* test-time training

$\text{Memory}(x; W_t) = W_t\, x$

x_1, y_1

x_2, y_2

$\vdots$

x_t, y_t

$x^\star$

Called **retrieval augmented generation (RAG)**

- Most common today: RAG + transformers / self-attention

doesn't scale to many retrieved examples!

- For RAG + test-time training, we can approximate deep test-time training (i.e., non-linear memory) as a linear function in frozen features $\phi(x)$ → next!

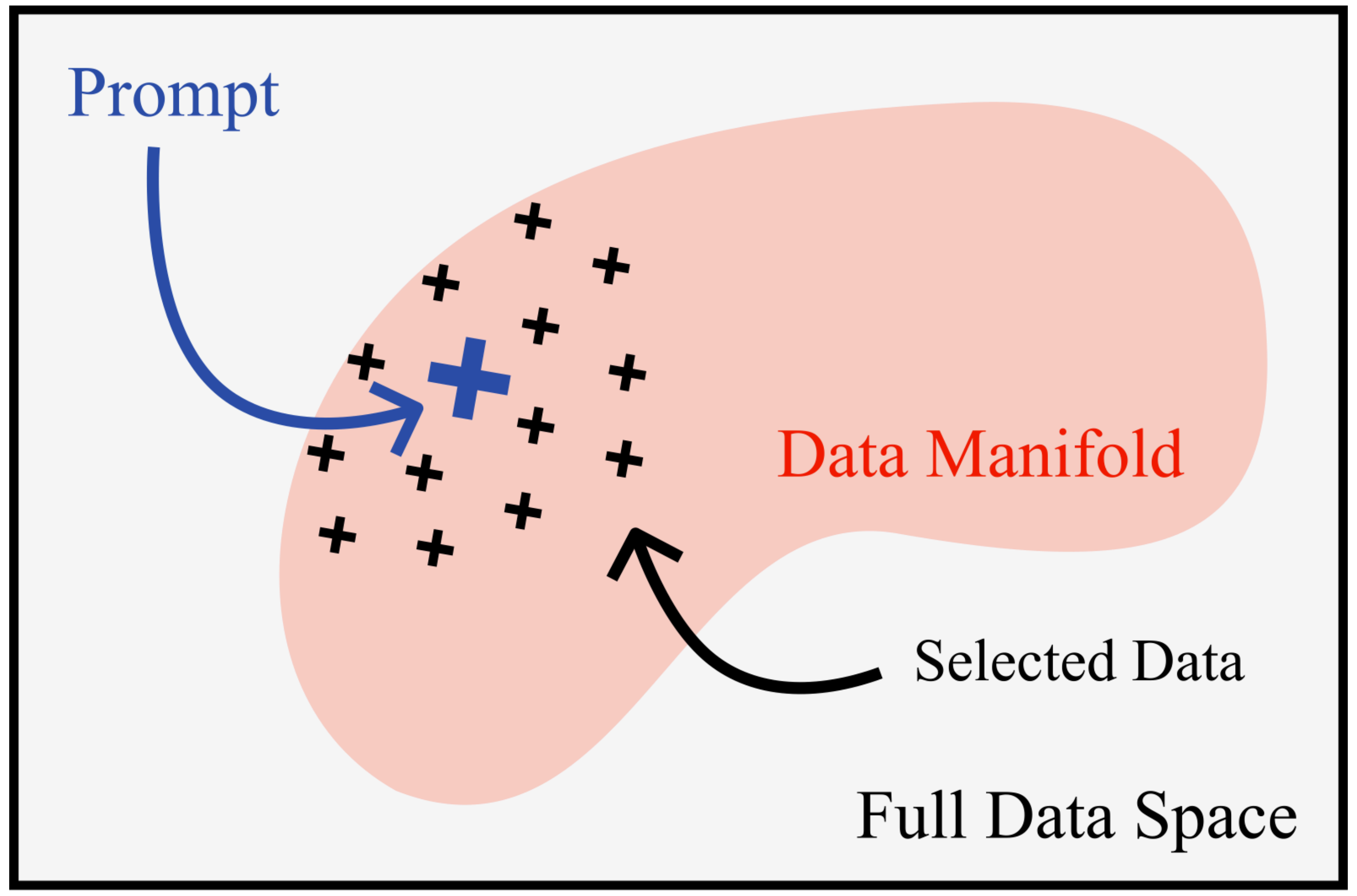
Example: Language modeling



ICLR ‘25

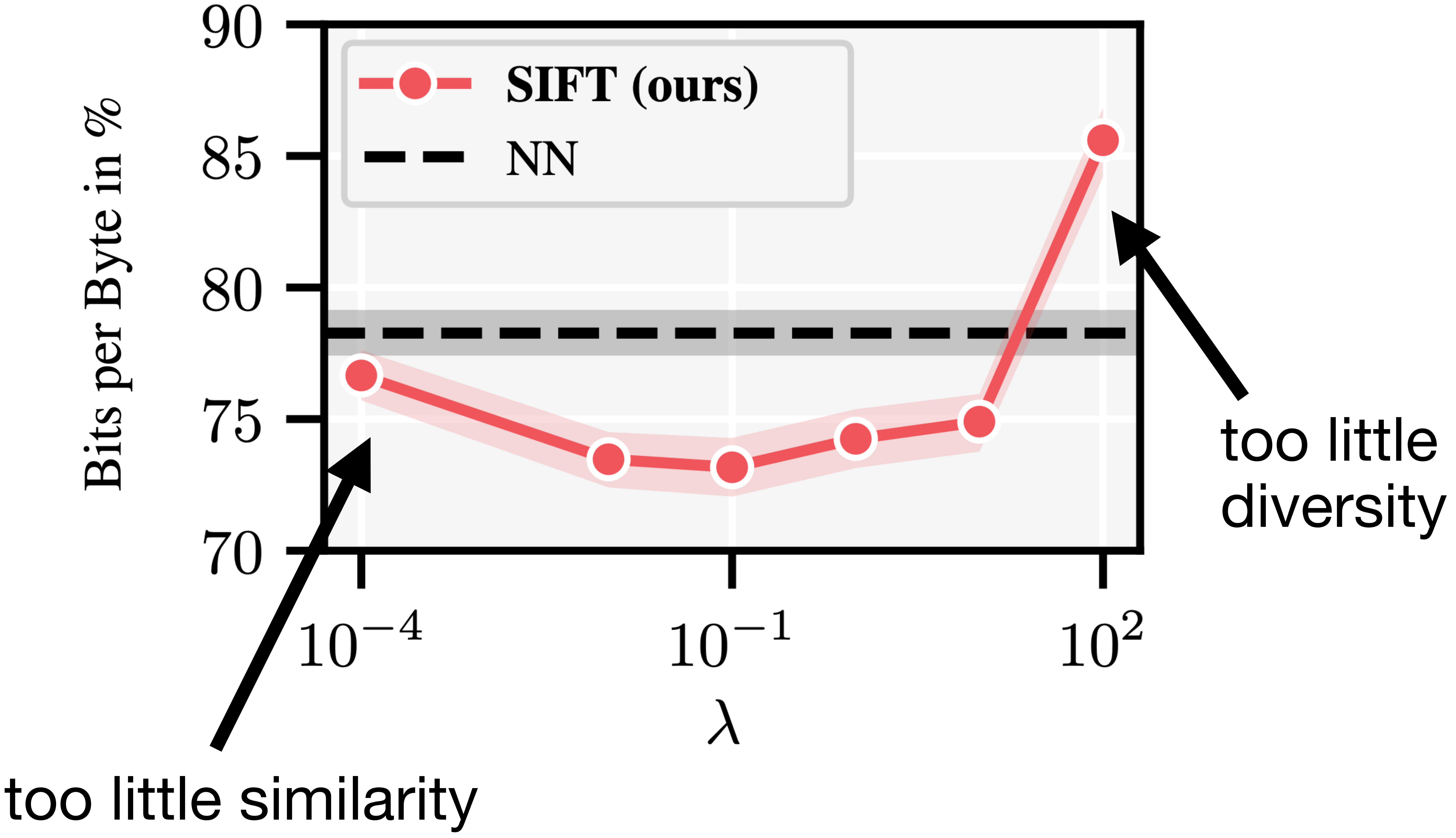
Selecting informative data for fine-tuning (SIFT):

Select data that maximally reduces “uncertainty” about how to solve the task



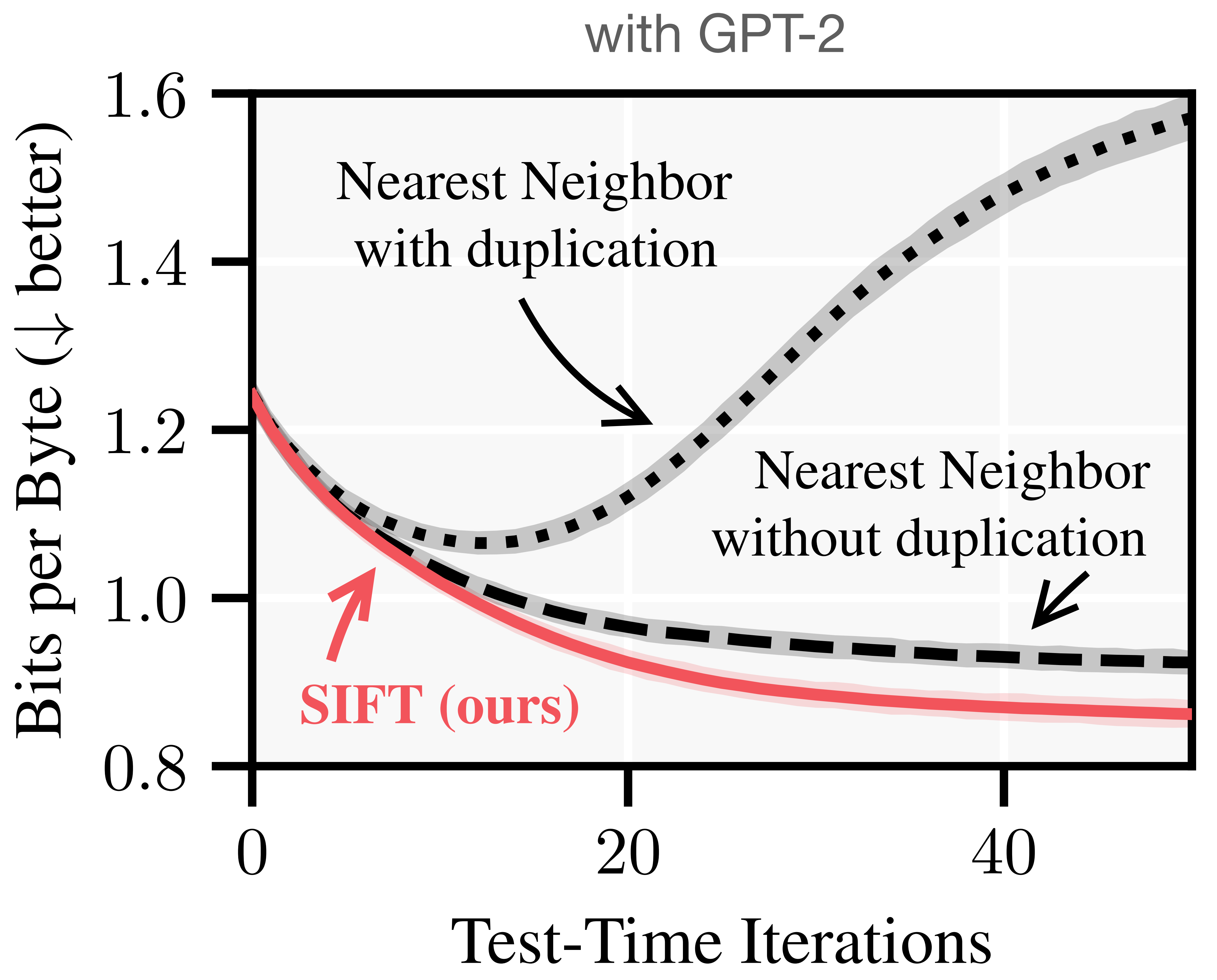
Simple TTT procedure:

1. given task x, find local data D_x (from dataset D)
2. fine-tune pre-trained model f on local data D_x to get **specialized model** f_x
3. predict $f_x(x)$


SIFT (ours)
NN
Bits per Byte in %
90
85
80
75
70
10^{-4}
10^{-1}
10^{2}
λ
too little diversity
too little similarity

Evaluation: language modeling on the Pile

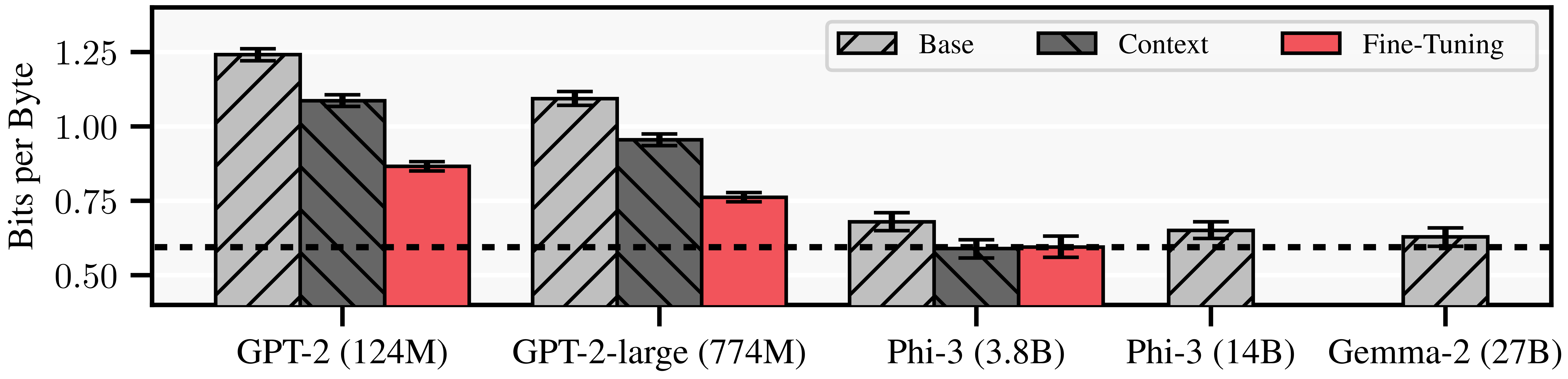
Pile dataset
NIH Grants
US Patents
GitHub
Enron Emails
Common Crawl
ArXiv
Wikipedia
PubMed Abstr.
Hacker News
Stack Exchange
PubMed Central
DeepMind Math
FreeLaw
All



	US	NN	NN-F	SIFT	Δ
NIH Grants	93.1 (1.1)	84.9 (2.1)	91.6 (16.7)	**53.8** (8.9)	↓31.1
US Patents	85.6 (1.5)	80.3 (1.9)	108.8 (6.6)	**62.9** (3.5)	↓17.4
GitHub	45.6 (2.2)	42.1 (2.0)	53.2 (4.0)	**30.0** (2.2)	↓12.1
Enron Emails	**68.6** (9.8)	**64.4** (10.1)	91.6 (20.6)	**53.1** (11.4)	↓11.3
Wikipedia	67.5 (1.9)	**66.3** (2.0)	121.2 (3.5)	**62.7** (2.1)	↓3.6
Common Crawl	92.6 (0.4)	90.4 (0.5)	148.8 (1.5)	**87.5** (0.7)	↓2.9
PubMed Abstr.	88.9 (0.3)	87.2 (0.4)	162.6 (1.3)	**84.4** (0.6)	↓2.8
ArXiv	85.4 (1.2)	**85.0** (1.6)	166.8 (6.4)	**82.5** (1.4)	↓2.5
PubMed Central	**81.7** (2.6)	**81.7** (2.6)	155.6 (5.1)	**79.5** (2.6)	↓2.2
Stack Exchange	78.6 (0.7)	78.2 (0.7)	141.9 (1.5)	**76.7** (0.7)	↓1.5
Hacker News	**80.4** (2.5)	**79.2** (2.8)	133.1 (6.3)	**78.4** (2.8)	↓0.8
FreeLaw	**63.9** (4.1)	**64.1** (4.0)	122.4 (7.1)	**64.0** (4.1)	↑0.1
DeepMind Math	**69.4** (2.1)	**69.6** (2.1)	121.8 (3.1)	**69.7** (2.1)	↑0.3
All	80.2 (0.5)	78.3 (0.5)	133.3 (1.2)	**73.5** (0.6)	↓4.8

Error relative to base model
(100 = base model, 0 = no error)

Test-time training vs Self-attention



	Context	Fine-Tuning	Δ
GitHub	74.6 (2.5)	**28.6** (2.2)	↓56.0
DeepMind Math	100.2 (0.1)	**70.1** (2.1)	↓30.1
US Patents	87.4 (2.5)	**62.2** (3.6)	↓25.2
FreeLaw	87.2 (3.6)	**65.5** (4.2)	↓21.7

GPT-2

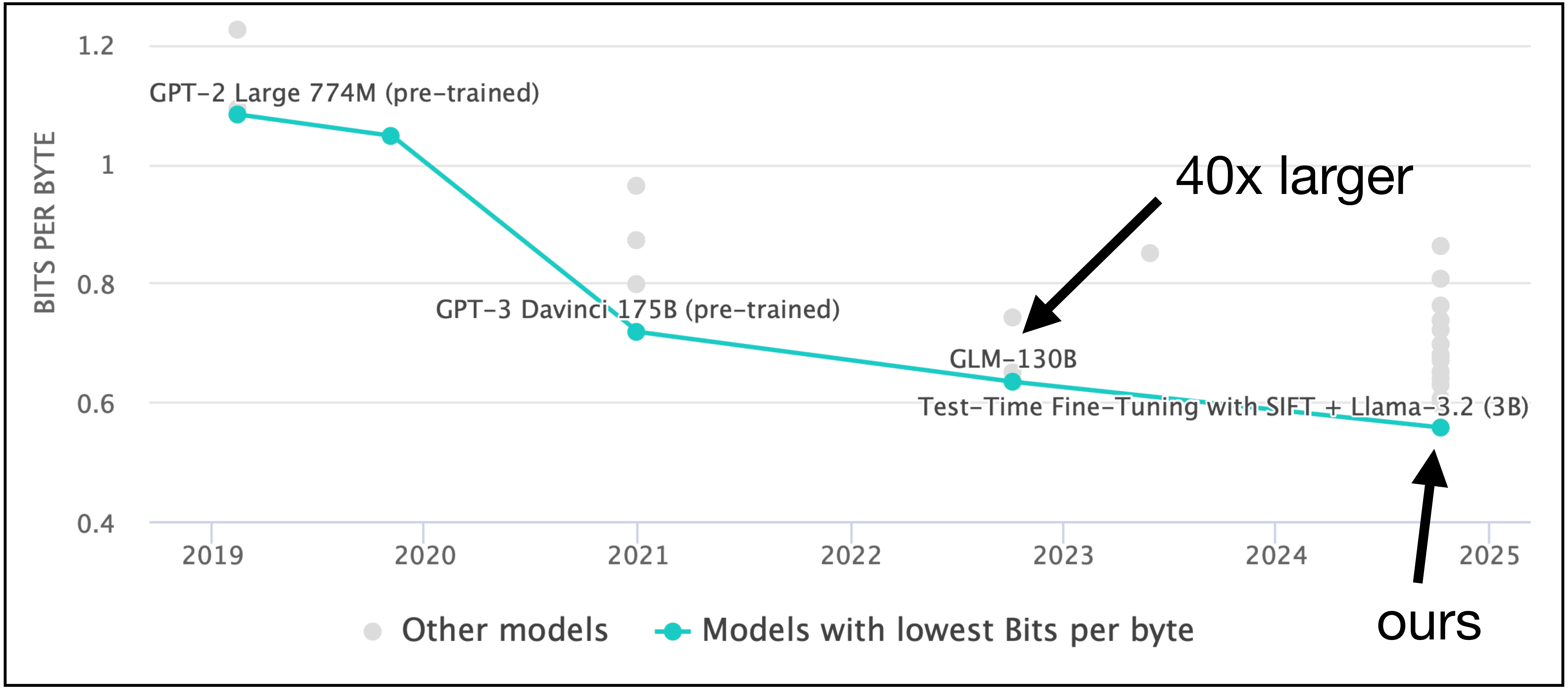
	Context	Fine-Tuning	Δ
GitHub	74.6 (2.5)	**31.0** (2.2)	↓43.6
DeepMind Math	100.2 (0.7)	**74.2** (2.3)	↓26.0
US Patents	87.4 (2.5)	**64.7** (3.8)	↓22.7
FreeLaw	87.2 (3.6)	**68.3** (4.2)	↓18.9

GPT-2-large

	Context	Fine-Tuning	Δ
DeepMind Math	100.8	75.3	↓25.5
GitHub	71.3	46.5	↓24.8
FreeLaw	78.2	67.2	↓11.0
ArXiv	101.0	94.3	↓6.4

Phi-3

SOTA on the Pile benchmark



https://paperswithcode.com/sota/language-modelling-on-the-pile

Learning through trial & error

What if do not have any labels / demonstrations to learn from?

- Can we generate examples ourselves through "practice" within an environment?

$x_1, \cancel{y_1}$

$x_2, \cancel{y_2}$

$\vdots$

$x_t, \cancel{y_t}$

$x^\star$

supervised learning

$\longrightarrow$

x_1, a_1, r_1

x_2, a_2, r_2

$\vdots$

x_t, a_t, r_t

$x^\star$

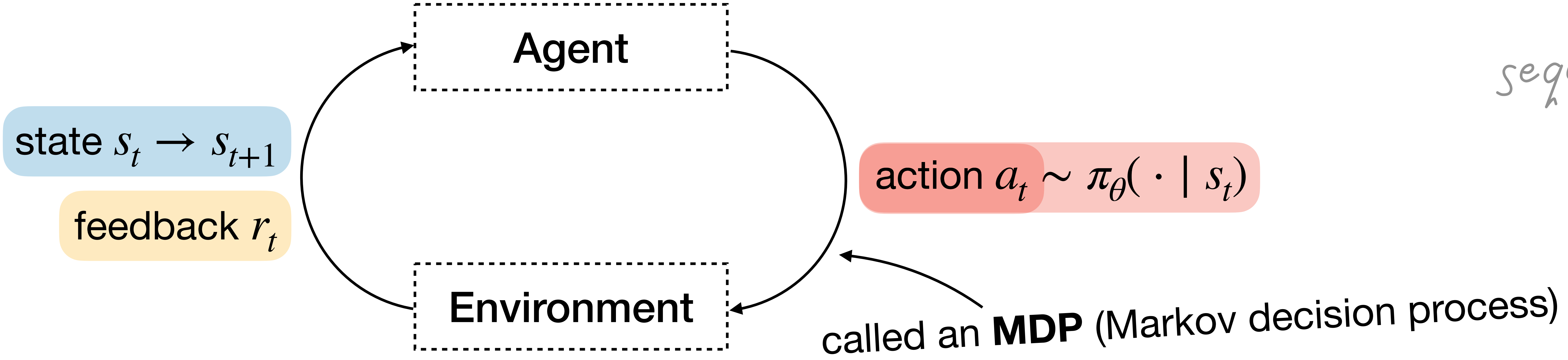
reinforcement learning

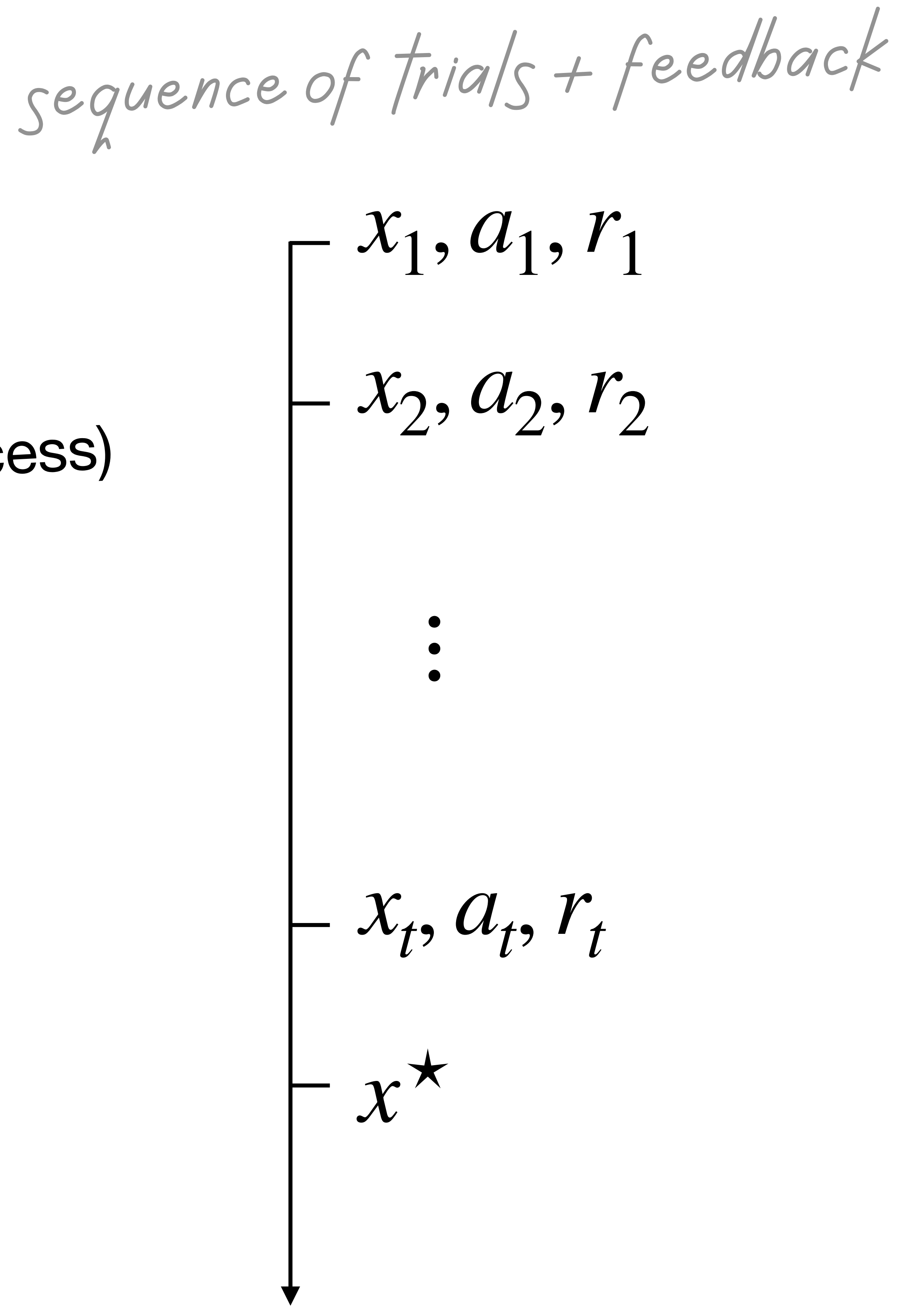
Attempt

Was attempt "good" or "bad"?

→ keep doing "good" attempts, stop doing "bad" attempts

Reinforcement learning (RL)





reinforcement learning

- **Action**: “attempt” by agent
  - example: coding a sorting algorithm
- **Feedback**: score indicating whether attempt was “good” or “bad”
  - example: does the code pass unit tests?
- **State**: “effect” of actions is conditionally independent of the past given the present state
  - example: updated file system (if an action modified the file system)
- **Objective**: maximize returns $\mathbb{E}_{\pi_\theta,\mathrm{MDP}}[\sum_{t=1}^{T} r_t]$

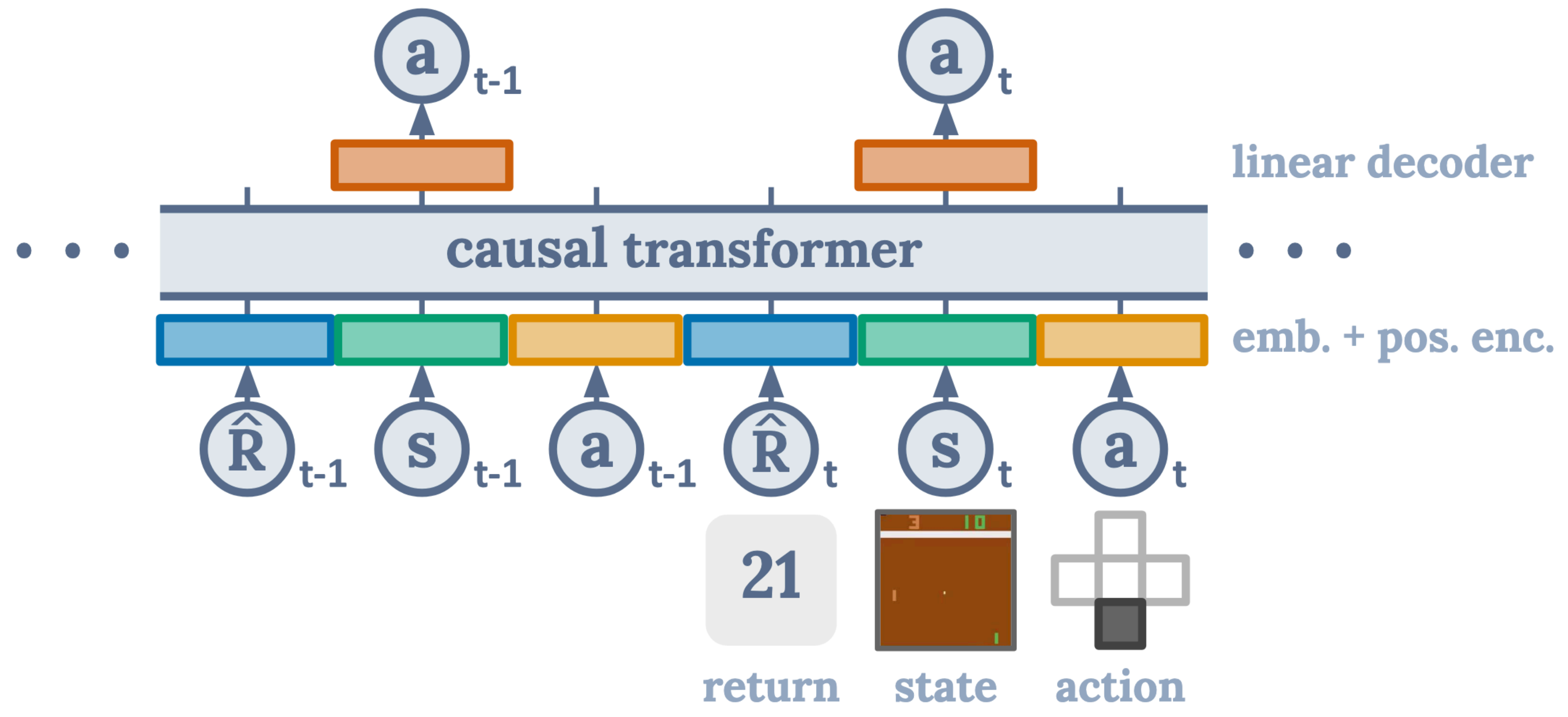
Perspective: Decision transformer

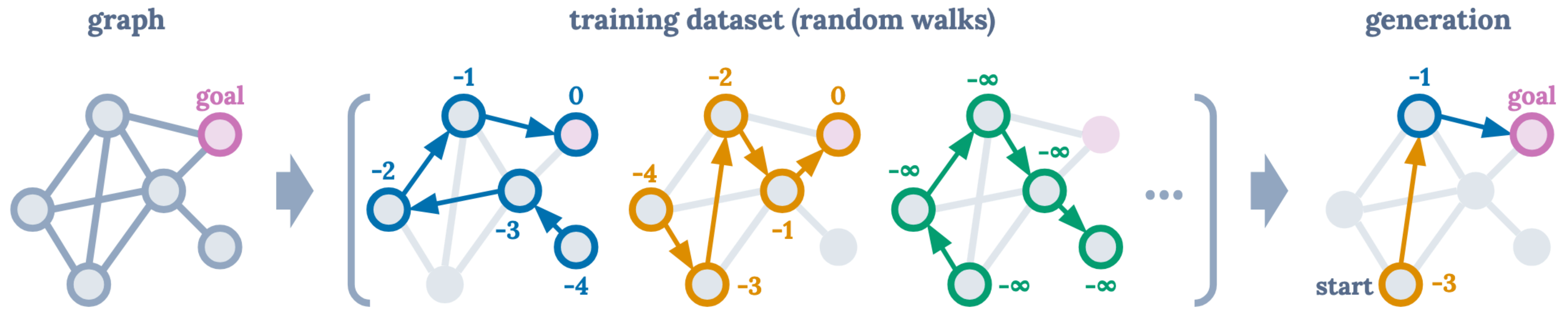
As seen multiple times already: The inner loop can learn

- with a non-parametric memory → self-attention / transformer doesn't scale to long sequences!
- with a parametric memory → training a policy with gradient descent (example: linear attention)

historically more common in rl

Example of RL with non-parametric memory: Decision transformer 📝





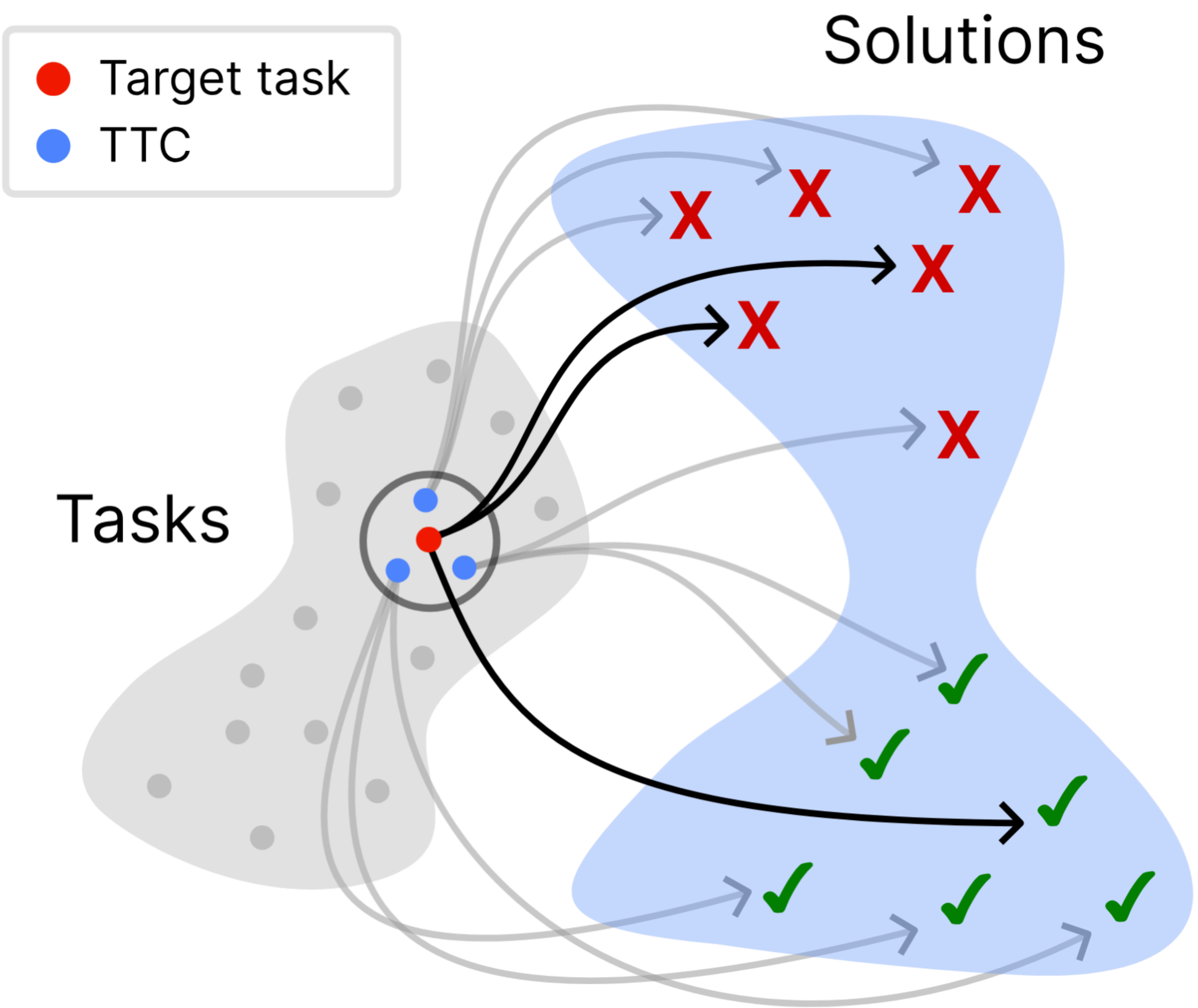
limited to relatively few examples

Today's models (like decision transformers) have seen *many* learning sequences during training & meta-learned how to learn from fewer examples at test-time → self-attention is feasible

Can TTT improve reasoning?

Preprint

- Can test-time training learn through trial & error?
- Given a test task, an LLM self-curates a **test-time curriculum (TTC)** of similar tasks for practicing
- The TTC is adaptively selected from a corpus (with SIFT) to balance similarity to the test task and diversity
- The LLM is trained on the TTC via RL



Test-time training for reasoning tasks

We treat each benchmark as a set of test tasks, and train on the TTC with RL

Model	AIME24	AIME25	MATH500	Codeforces	CodeElo	LCB^{v6}	GPQA-D

Test-time training for reasoning tasks

We treat each benchmark as a set of test tasks, and train on the TTC with RL

Model	AIME24	AIME25	MATH500	Codeforces	CodeElo	LCB^{v6}	GPQA-D
Qwen3-8B							
+ RL post-training							
+ TTC-RL							
Qwen3-4B-Instruct-2507							
+ RL post-training							
+ TTC-RL							
Qwen3-8B-Base							
+ RL post-training							
+ TTC-RL							

Pass@1 accuracy on reasoning tasks of
- base model
- model after global RL post-training
- TTC-RL

Test-time training for reasoning tasks

We treat each benchmark as a set of test tasks, and train on the TTC with RL

Model	AIME24	AIME25	MATH500	Codeforces	CodeElo	LCB^{v6}	GPQA-D
Qwen3-8B	21.67	23.33	69.55	20.85	13.73	20.61	49.11
+ RL post-training	41.67	38.33	82.50	27.83	22.67	25.95	56.47
+ TTC-RL	**50.83**$^{+29.2}$	**41.67**$^{+18.3}$	**85.10**$^{+15.6}$	**33.35**$^{+12.5}$	**29.34**$^{+15.6}$	**27.29**$^{+6.7}$	**58.38**$^{+9.3}$
Qwen3-4B-Instruct-2507	52.50	40.83	72.00	26.70	20.27	21.56	61.93
+ RL post-training	55.83	**47.50**	86.30	28.39	21.18	25.95	**62.82**
+ TTC-RL	**60.00**$^{+7.5}$	45.83$^{+5.0}$	**88.50**$^{+16.5}$	**34.99**$^{+8.3}$	**27.20**$^{+6.9}$	**26.91**$^{+5.4}$	61.93$^{+0.0}$
Qwen3-8B-Base	15.83	14.17	63.10	9.92	6.67	11.26	29.70
+ RL post-training	22.50	20.83	76.85	17.46	9.97	**18.51**	42.77
+ TTC-RL	**30.00**$^{+14.2}$	**21.67**$^{+7.5}$	**78.15**$^{+15.1}$	**17.84**$^{+7.9}$	**11.33**$^{+4.7}$	17.94$^{+6.7}$	**45.94**$^{+16.2}$

Pass@1 accuracy on reasoning tasks of
- base model
- model after global RL post-training
- TTC-RL

Takeaway: TTC-RL consistently achieves a higher pass@1 than general-purpose RL post-training on frontier open-weight models → learning how to use context (self-attention) for individual attempts.

Summary #2

Where we started: If task-specific data is not given to us, can we acquire it ourselves?

We saw:

- In supervised linear setting, we can compute "optimal" retrieval scheme in closed-form → SIFT
- This retrieval scheme also works empirically with non-linear TTT
- Can use reinforcement learning to learn through practice, without supervision / solutions

Can we *learn* how to acquire data instead of relying on simplifying assumptions?

in the spirit of machine learning

Outlook: Learning how to acquire data

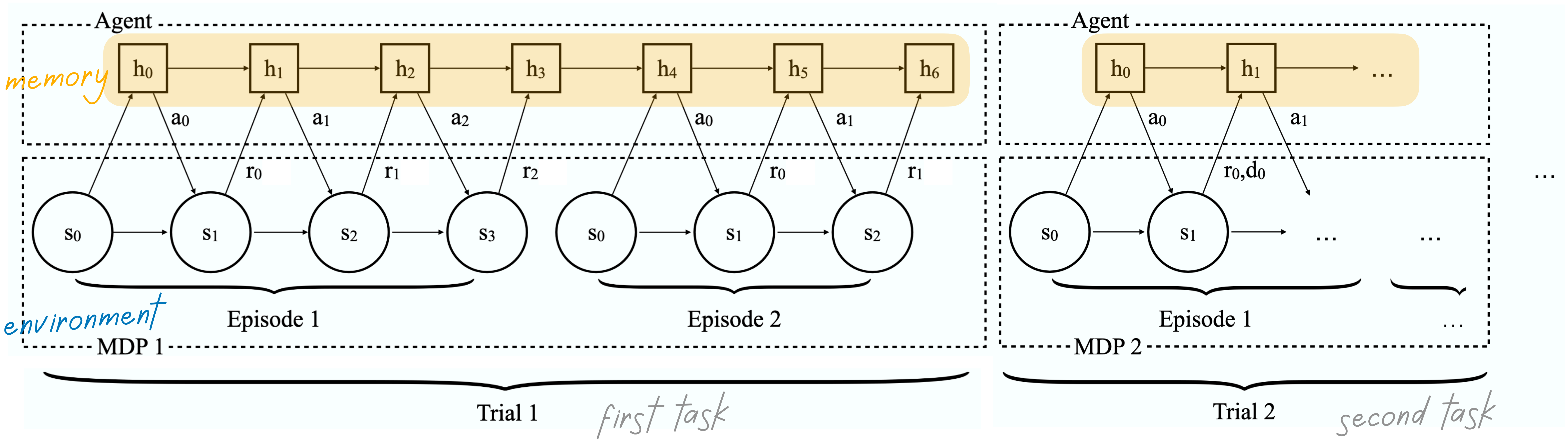
Goal: Can we *learn* how to acquire data instead of relying on simplifying assumptions?

- What is the optimal $x_1, \ldots, x_t$ for task $x^\star$ without any assumption on the underlying f?

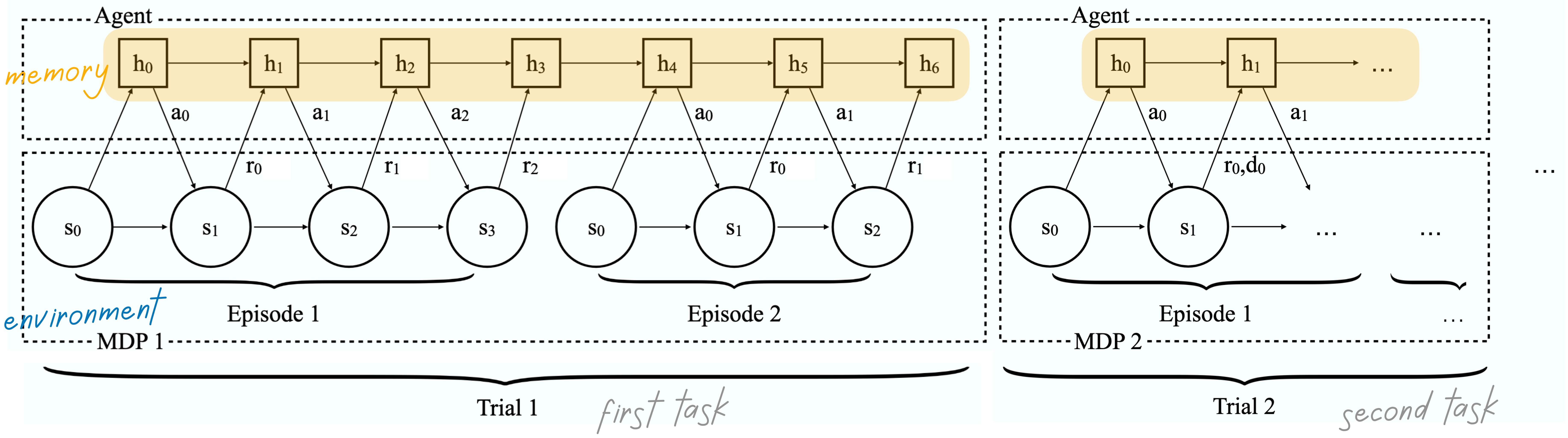
Recall meta-learning!

- Can learn what to store in memory (→ part 1)

Can also learn how to select data in the inner loop! → example: **RL²**

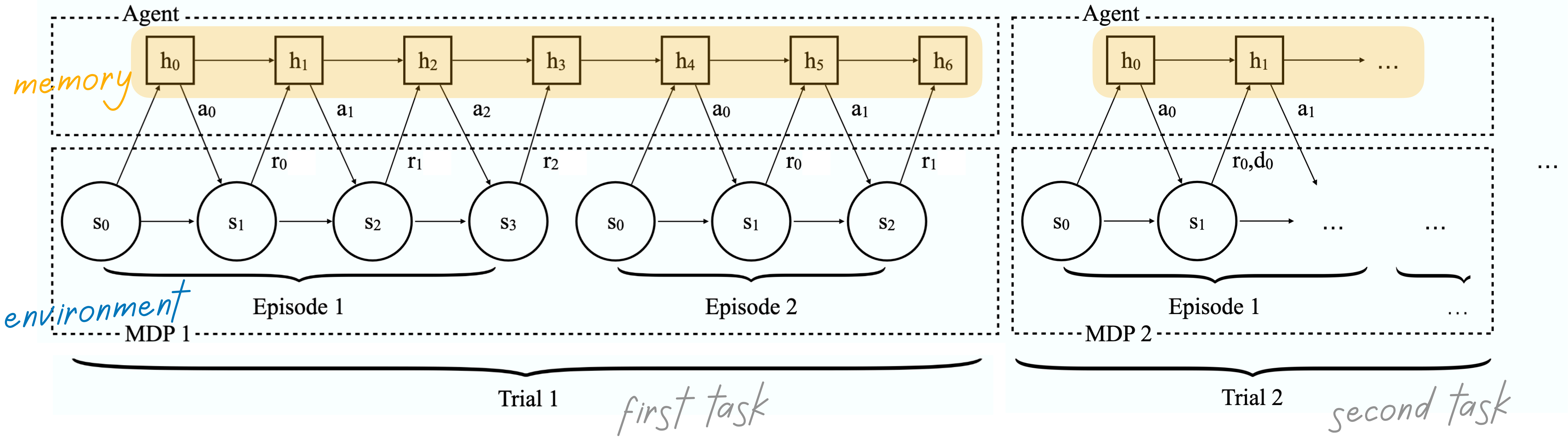


Outlook: RL²

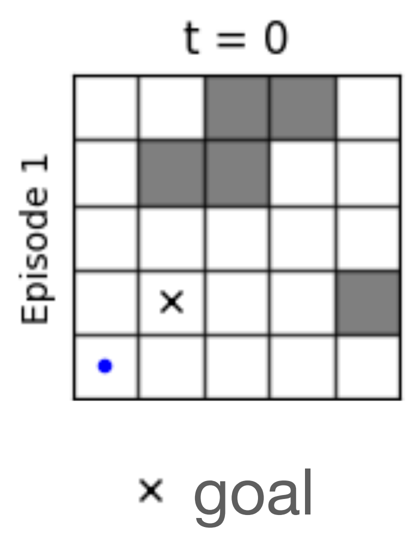


- **Test-time**: The inner loop aims to maximize returns in its environment (defined by task $x^\star$): $\mathbb{E}_{\pi_\theta,\text{env}(x^\star)}[\sum_{t=1}^{T} r_t]$
- **Train-time**: The outer loop finds an initialization leading to high returns of the inner loop (on average across $x^\star$) via RL: $\mathbb{E}_{x^\star}\mathbb{E}_{\pi_\theta,\text{env}(x^\star)}[\sum_{t=1}^{T} r_t]$

RL² example

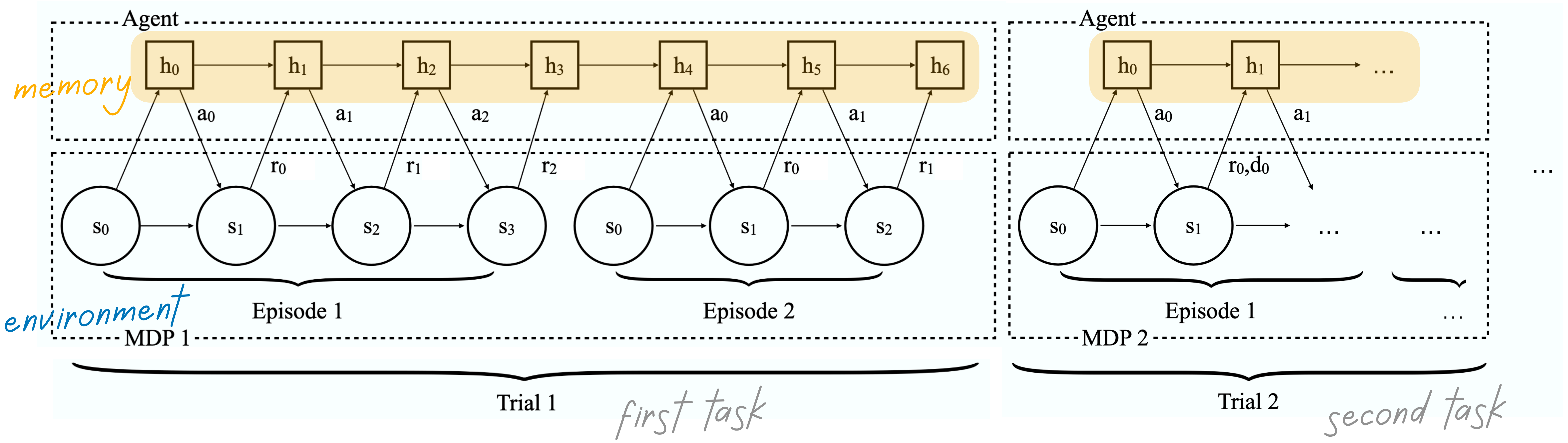


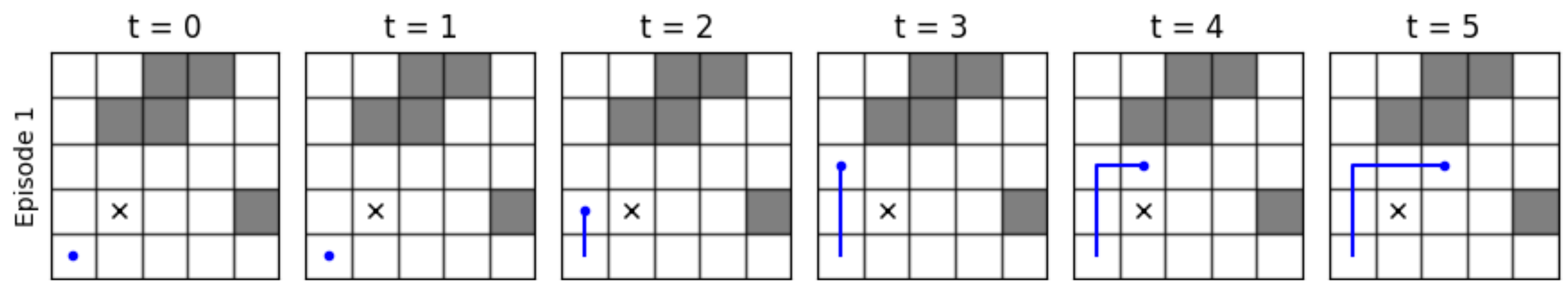
memory
environment
Agent
h_0 h_1 h_2 h_3 h_4 h_5 h_6
a_0 a_1 a_2 a_0 a_1
r_0 r_1 r_2 r_0 r_1
s_0 s_1 s_2 s_3 s_0 s_1 s_2
Episode 1
Episode 2
MDP 1
Trial 1
first task
Agent
h_0 h_1
a_0 a_1
r_0, d_0
s_0 s_1
Episode 1
MDP 2
Trial 2
second task



t = 0
Episode 1
× goal

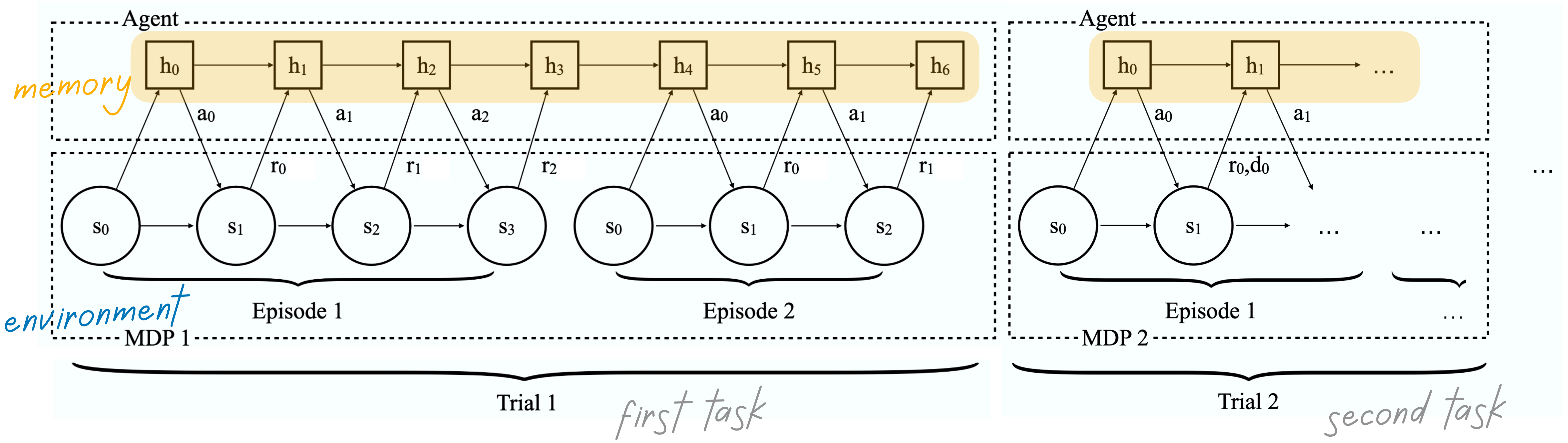
RL^2 example



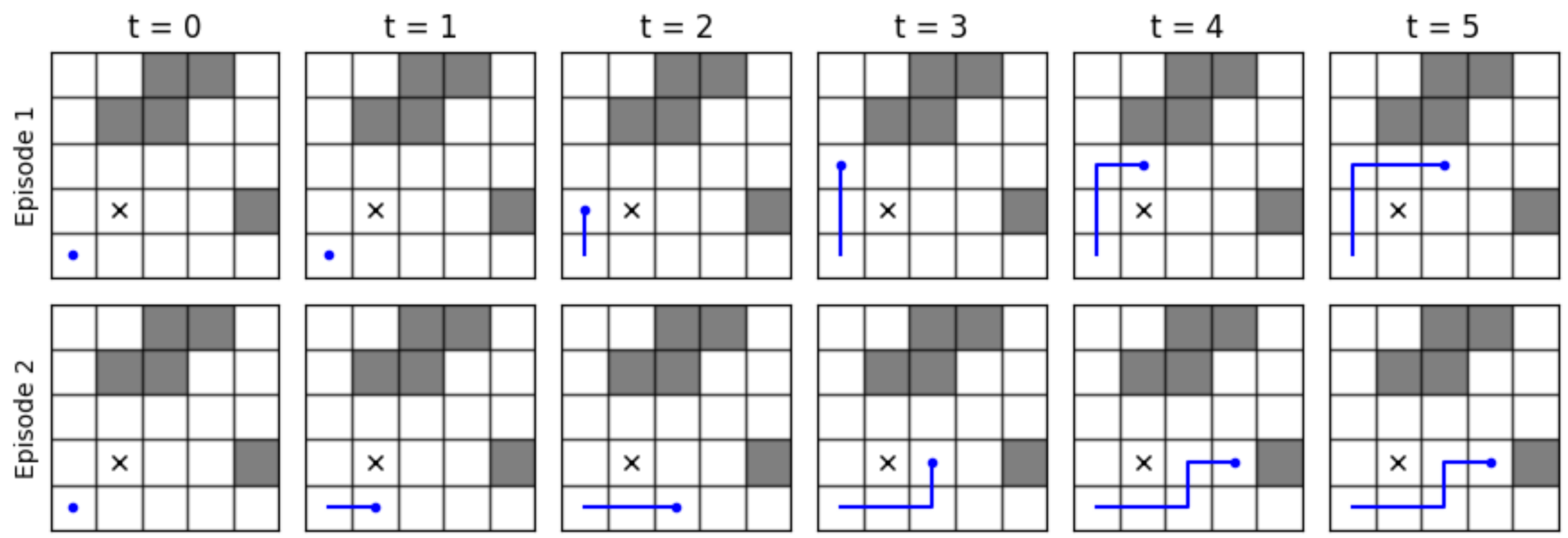


× goal

RL² example

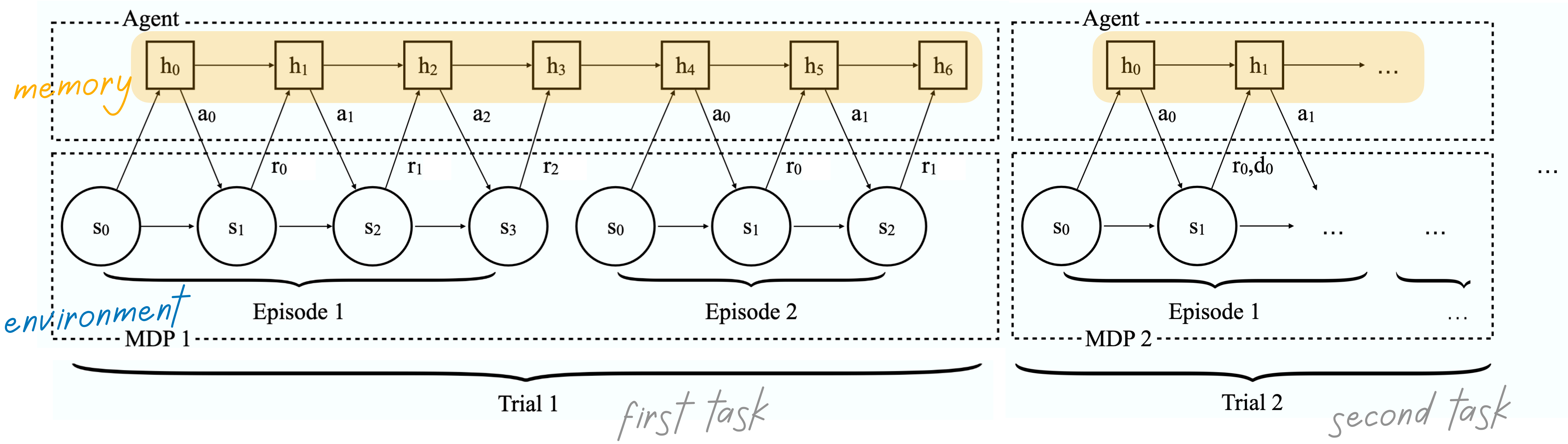


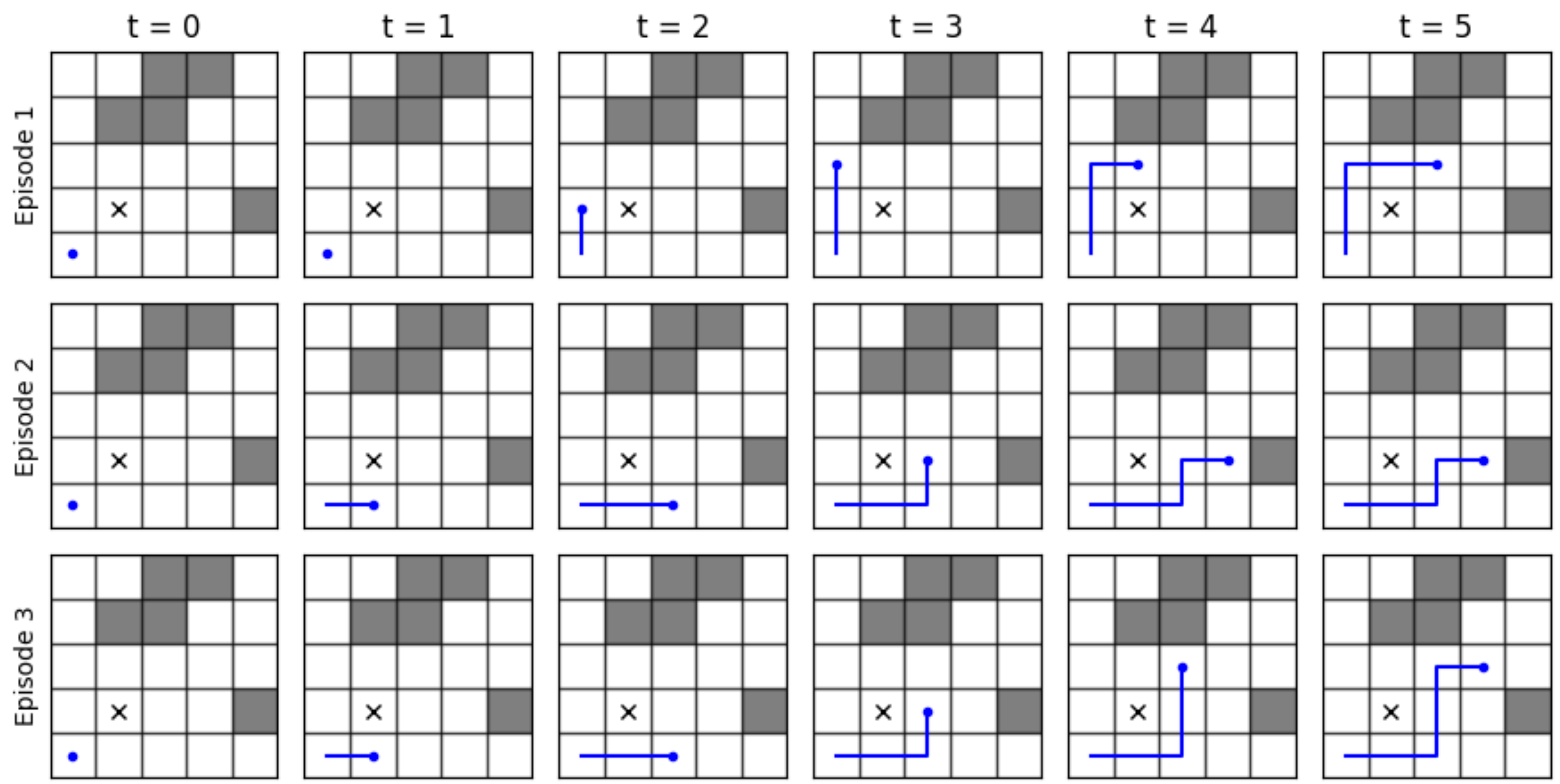
Agent
memory
h_0 h_1 h_2 h_3 h_4 h_5 h_6
a_0 a_1 a_2 a_0 a_1
r_0 r_1 r_2 r_0 r_1
s_0 s_1 s_2 s_3 s_0 s_1 s_2
Episode 1
Episode 2
environment
MDP 1
Trial 1
first task
Agent
h_0 h_1 ...
a_0 a_1
r_0,d_0
s_0 s_1 ...
Episode 1
MDP 2
Trial 2
second task



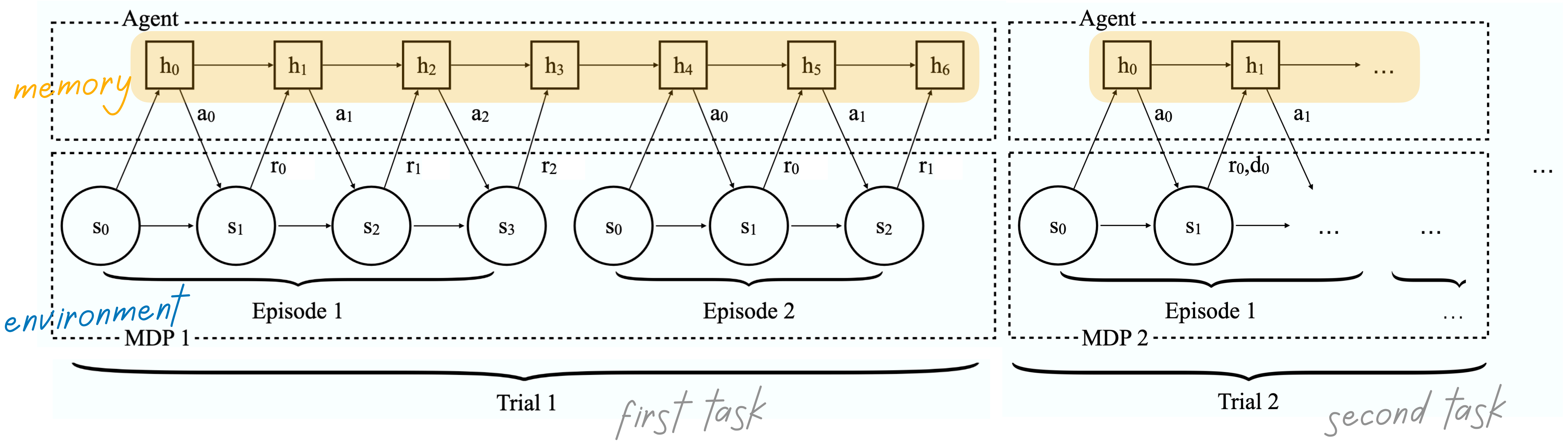
t = 0
t = 1
t = 2
t = 3
t = 4
t = 5
Episode 1
Episode 2

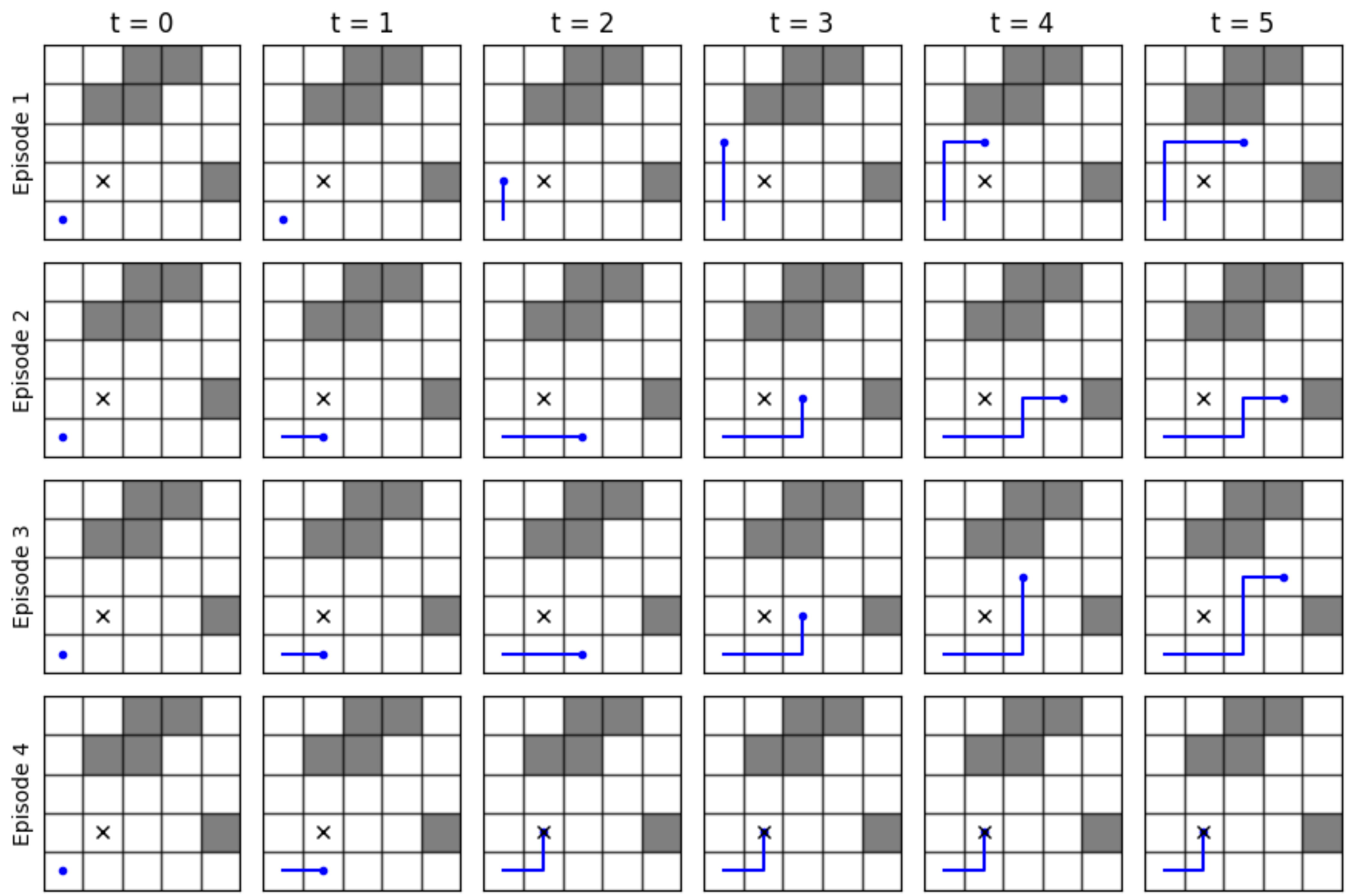
RL² example


Agent
memory
h_0 h_1 h_2 h_3 h_4 h_5 h_6
a_0 a_1 a_2 a_0 a_1
r_0 r_1 r_2 r_0 r_1
s_0 s_1 s_2 s_3 s_0 s_1 s_2
Episode 1
Episode 2
environment
MDP 1
Trial 1
first task
Agent
h_0 h_1 ...
a_0 a_1
r_0,d_0
s_0 s_1 ...
Episode 1
MDP 2
Trial 2
second task


t = 0
t = 1
t = 2
t = 3
t = 4
t = 5
Episode 1
Episode 2
Episode 3

RL² example





Key question: By solving many of these environments, can we *learn* an algorithm for efficient exploration of *novel* environments?

❶ TTT scales attention to long sequences

❷ Acquiring data to learn from at test-time

About us

- I'm a PhD student at LAS with Andreas Krause
- Our lab works on learning & adaptive systems that
  - actively acquire information
  - continually learn at test-time

Learning & Adaptive Systems

Talk to us if you're interested in doing a research visit at LAS!

- e.g., Master's thesis