Developing RL Agents that Learn Many Subtasks

Martha White
Associate Professor
University of Alberta

Goals for the Talk

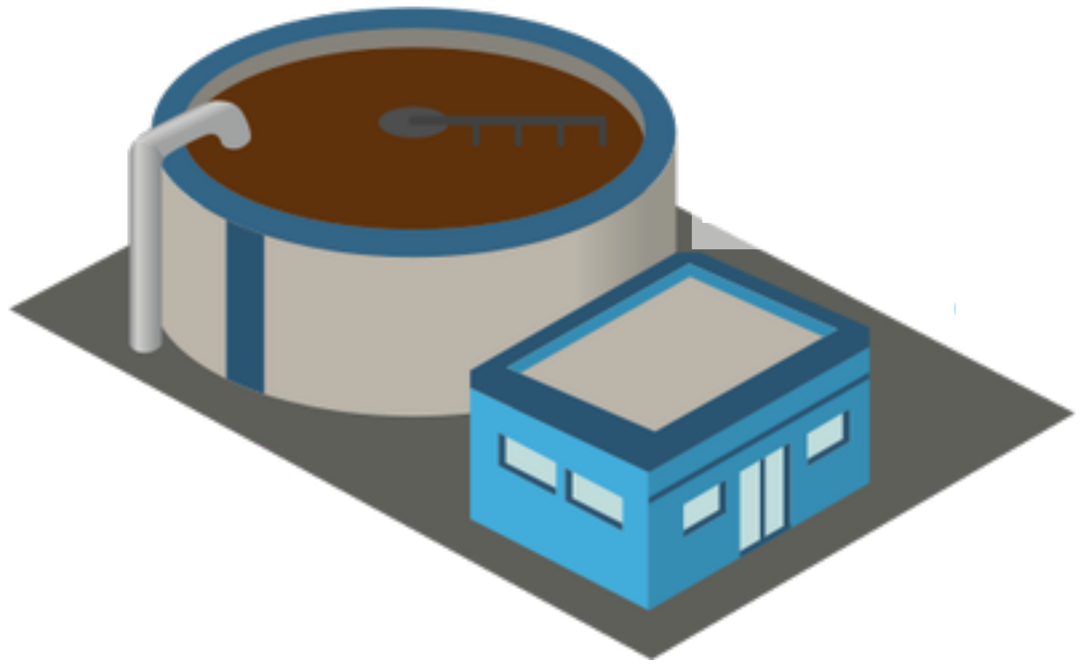
- Motivate that general purpose agents need to learn many subtasks in parallel
- Introduce the Continual Subtask Learning setting
  - which allows us to focus on developing such agents
- Point out exciting open research questions in this area
  - as well as some progress we have made

Let’s start with a brief background in RL

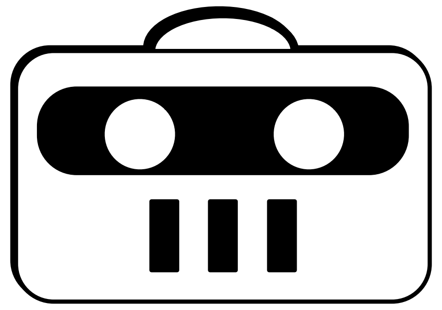
Problem Setting: Reinforcement Learning

- An agent interacts with the environment, to maximize reward

s', r



s



a

Agent learns policy $\pi(a \mid s)$
to maximize expected return

$$\mathbb{E}_\pi[G_t \mid S_t = s]$$

with $G_t = R_{t+1} + \gamma R_{t+2} + \dots$

$s_0, a_0, r_1, s_1, a_1, r_2, s_2, a_2, \dots$

Most Learning Approaches use Value Estimation

- A value function v_π tells us the expected return from a state s, under policy π
  - $v_\pi(s) = \mathbb{E}_\pi[G_t \mid S_t = s] = \mathbb{E}_\pi[R + \gamma v_\pi(S') \mid S = s]$
- Action-value q_π allows us to improve the policy, by taking greedy actions
  - $q_\pi(s, a) = \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a]$
  - $\pi'(s) = \arg\max_{a \in \mathcal{A}} q_\pi(s, a)$ obtains as good or higher return in each state

An Example of a Learning Agent: Sarsa

- Sarsa Agent learns policy through trial-and-error interaction
- Learns action-values $\hat{q}$ and uses softmax (Boltzmann) policy on $\hat{q}$ to select actions proportionally to their value: $\pi(a \mid s) \propto \exp(\hat{q}(s, a))$
- In state s_t, the agent takes action $a_t \sim \pi(\cdot \mid s_t)$, transitions to s_{t+1} and receives reward r_{t+1} and preemptively samples $a_{t+1} \sim \pi(\cdot \mid s_{t+1})$
- It updates its value estimate $\hat{q}$ with parameters $\mathbf{w}$ using

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha_t(\hat{G}_t - \hat{q}(s_t, a_t)) \nabla_w \hat{q}(s_t, a_t)$$

- where $\hat{G}_t \doteq r_{t+1} + \gamma \hat{q}(s_{t+1}, a_{t+1})$ approximates the true return from s_t under π

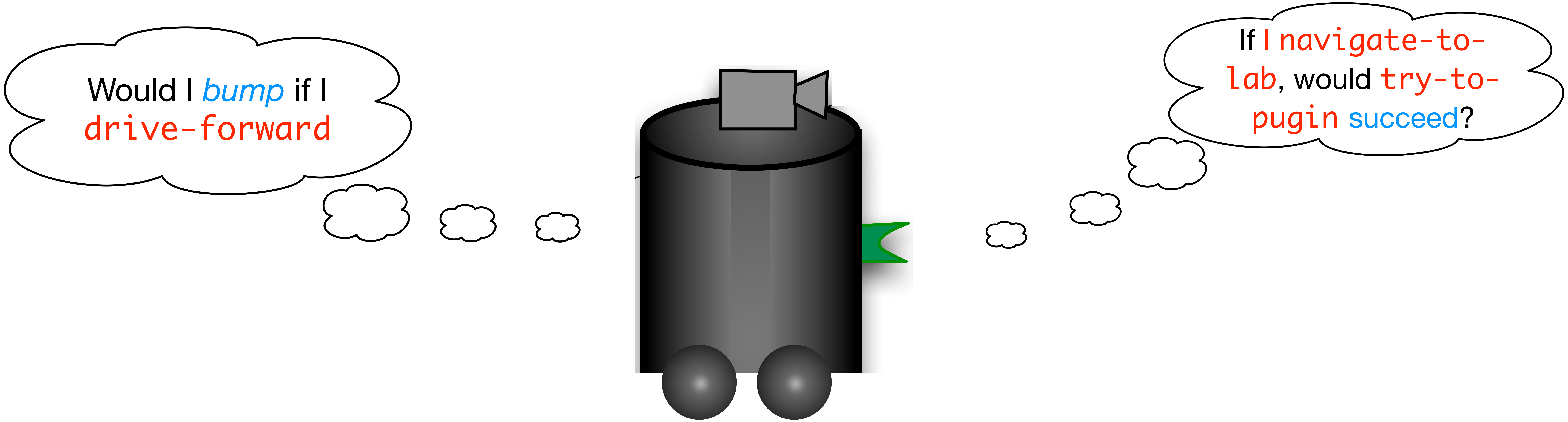
But how far can we get with this simple (model-free, trial-and-error) update?

We Need More for the Lifelong Learning Setting

- Many steps of interaction
- Potentially vast environments
- Consider examples such as
  - AssistantBot interacting with people
  - CourierBot navigating a city
  - EcoAgent controlling energy usage for an (expanding) network of buildings

Example: CourierBot

- The RL agent needs to make many predictions about the world
  - What will happen if I pick up this object?
  - How many steps until I get to the door?
  - How much longer can I drive before I need to recharge?



Lifelong Learning is a Practical Paradigm

- Real-world environments
  - are complex and potentially vast
  - require the agent to run for a long time
- Lifelong learning is not grandiose nor is it only about AGI
- We will need to tackle this setting to obtain agents for complex environments

Claim: Under a Long Sequence of Interaction…

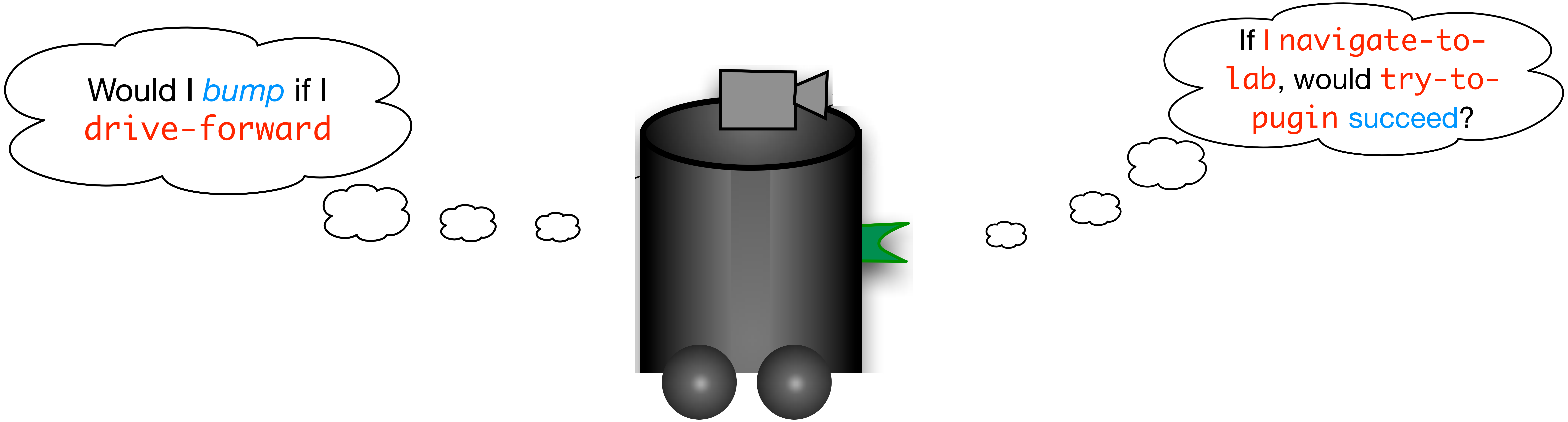
- the agent should accumulate knowledge about its environment
- that knowledge can be used to learn/adapt faster in
  - new situations
  - under nonstationarity, which can arise even just from limited function approximation in a large, complex world

Knowledge as Subtasks

- Subtasks are modular pieces of knowledge about the world that can be re-used
- We use two types of subtask specifications
  - Options/Skills - Control Subtasks
  - General Value Functions (GVF) - Prediction Subtasks

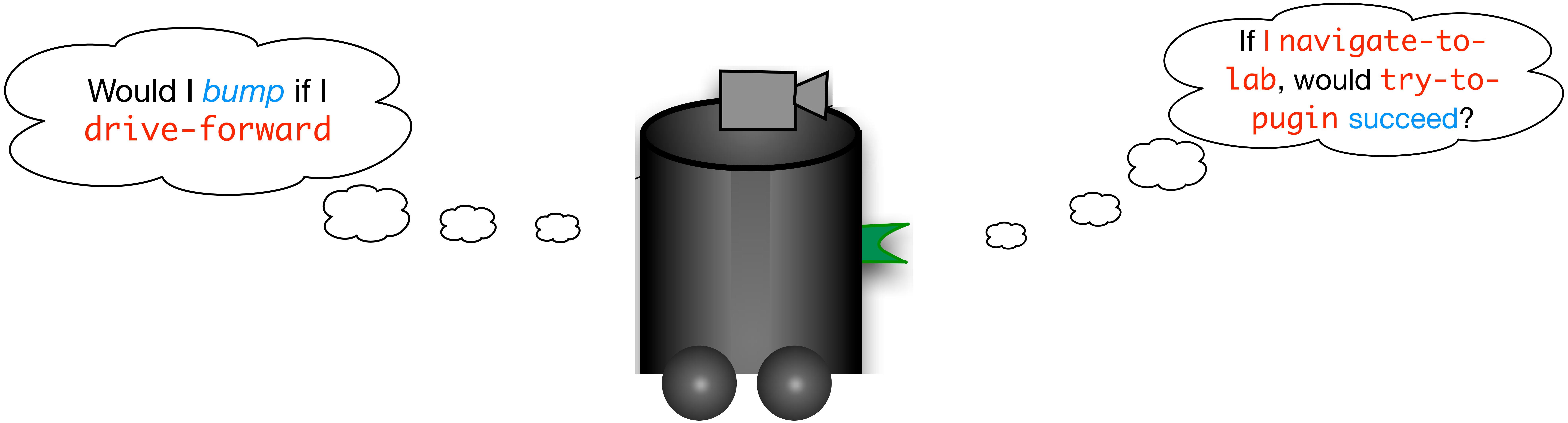
Example of an Option (Control Subtask)

- Learn an **option** policy π that navigates to the lab (e.g., with Sarsa)
  - The primary task is to maximize reward in the environment
  - This subtask (navigate to lab) helps the agent with the primary task



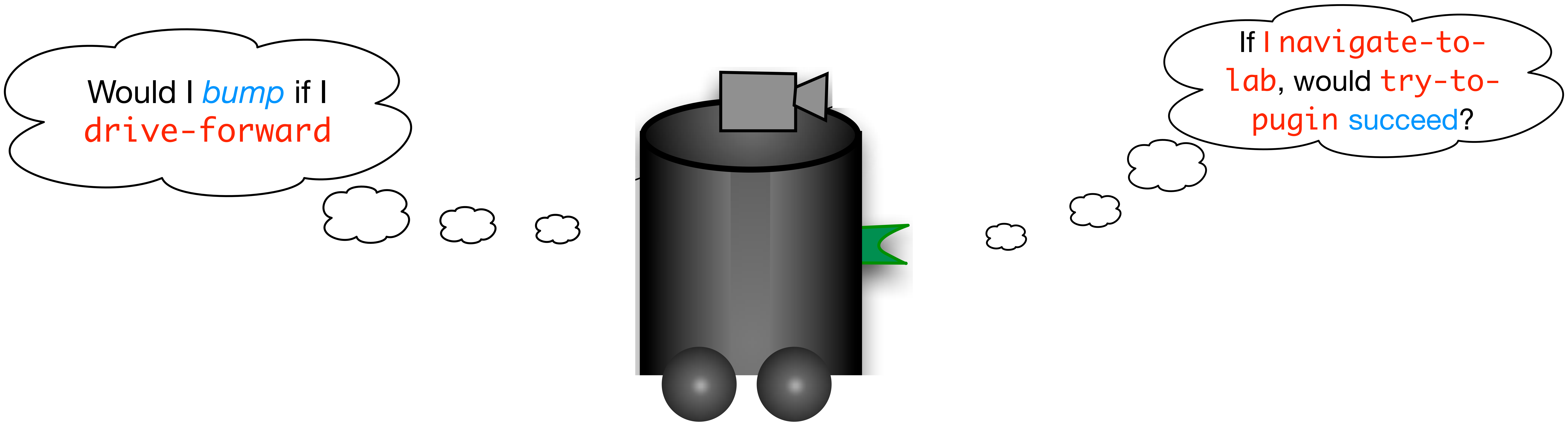
Example of an Option (Control Subtask)

- Learn an **option** policy π that navigates to the lab
- This option can be repeatedly used by the agent
  - as a macro-action: executing a known skill or behavior
  - for planning: it can reason about the utility of going to the lab



Examples of GVFs (Prediction Subtask)

- What is the probability I will successfully plug-in, if I run the navigate-to-lab option policy π?
- How many packages will I receive today?



General Value Functions allow us to encode these types of predictions

They are a simple generalization of value functions, to reason about any signal or cumulant (instead of only the reward)

One Example of a GVF Subtask

- **Question**: What is the probability I will successfully plug-in, if I run the navigate-to-lab option policy π_{lab}?
- **Answer**: Learn value function with **cumulant** in-place of reward
- $c(s, a, s') = \begin{cases} 1 & \text{if s' = plugged-in} \\ 0 & \text{else} \end{cases}$
- $Q^{\pi_{lab}}(s, a) = \mathbb{E}_{\pi_{lab}}[C_{t+1} + C_{t+2} + \ldots \,|\, S_t = s, A_t = a]$

One Example of a GVF Subtask

- **Question**: What is the probability I will successfully plug-in, if I run the navigate-to-lab option policy π_{lab}?
- **Answer**: Learn value function with **cumulant** in-place of reward
- $c(s, a, s') = \begin{cases} 1 & \text{if s' = plugged-in} \\ 0 & \text{else} \end{cases}$
- $\hat{q}(s, a) \approx Q^{\pi_{lab}}(s, a) = \mathbb{E}_{\pi_{lab}}[C_{t+1} + C_{t+2} + \ldots \,|\, S_t = s, A_t = a]$

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha_t \left(c_{t+1} + \gamma \underbrace{\sum_{a' \in \mathcal{A}} \hat{q}(s_{t+1}, a')}_{\mathbb{E}_{\pi_{lab}}[\hat{q}(s_{t+1}, A')]} - \hat{q}(s_t, a_t) \right) \nabla_w \hat{q}(s_t, a_t)$$

Another Example of a GVF Subtask

- **Question**: How many packages will I receive today?
  - How many packages in total will I receive until the end of the day, under my typical behavior π?
- **Answer**: Learn value function with **cumulant** in-place of reward
- $c(s, a, s') = \begin{cases} 1 & \text{if s' = received package} \\ 0 & \text{else} \end{cases}$
- $Q^\pi(s, a) = \mathbb{E}_\pi[C_{t+1} + C_{t+2} + \ldots \mid S_t = s, A_t = a]$ with termination of return when s' indicates it is the end of the day
- The return accumulates 1s each time there is a package, until day end

GVF Subtasks Can Encode Simpler One-Step Models

- The cumulant C_{t+1} could correspond to a sensor value on the next step
    - $Q^{\pi}(s, a) = \mathbb{E}_{\pi}[C_{t+1} \,|\, S_t = s, A_t = a]$ is the expected sensor value
    - set termination to occur immediately
- More generally GVFs consider longer horizon predictions about the future
    - But cannot perfectly represent n-horizon predictions, such as those used in time series prediction

We nonetheless focus on GVFs and Options to specify subtasks:

- allow us to use the same value function algorithms throughout the system
- still provide a sufficiently rich language to specify subtasks

What is the Alternative to Learning Subtasks?

- Many RL systems use end-to-end learning of policies, such as with Sarsa
  - No models
  - No options
  - No GVFs
- For **smaller environments** (which can be covered in some reasonable time), that are stationary, there is not much need to learn secondary objects
  - So it is sensible to just use **Sarsa**
- For more **complex environments**, it is not too controversial that these secondary components (**subtasks**) are needed to obtain effective agents

Learning Multiple Subtasks is an Old Idea in AI

- Early formalisms in lifelong learning looked at learning subtasks **sequentially**
  - The experimenter designed the sequence of tasks for the agent
- We want to learn subtasks **in parallel**, from a single stream of experience
  - The agent decides for itself what subtasks to focus on and where to go in the environment to better learn the subtasks

Learning Multiple Subtasks is an Old Idea in AI

- Early formalisms in lifelong learning looked at learning subtasks **sequentially**
  - The experimenter designed the sequence of tasks for the agent
  - Naturally **on-policy**
- We want to learn subtasks **in parallel,** from a single stream of experience
  - The agent decides for itself what subtasks to focus on and where to go in the environment to better learn the subtasks
  - Naturally **off-policy**, agent needs to reason counterfactually

On-policy vs Off-policy

- CourierBot is dropping off a package near the lab (its charging station)
- It finds a new shortcut through a building
- It is currently executing policy π to drop off the package, but can use this new data to update its navigate-to-lab option policy π_{lab}
    - It updates π_{lab} **off-policy**, since it **counterfactually** reasons about how to improve π_{lab} when following π
- **On-policy** updating requires that we **execute** π_{lab} to learn about its value
- RL algorithms have **divergence** and **variance** issues under off-policy updates

Learning Multiple Subtasks is an Old Idea in AI

- Early formalisms in lifelong learning looked at learning subtasks **sequentially**
  - The experimenter designed the sequence of tasks for the agent
  - Naturally **on-policy**
- We want to learn subtasks **in parallel**, from a single stream of experience
  - The agent decides for itself what subtasks to focus on and where to go in the environment to better learn the subtasks
  - Naturally **off-policy**, agent needs to reason counterfactually
- We finally have **the tools** to explore this problem setting
  - significant improvements in off-policy algorithms within even just a few years

Committing to this Inductive Bias

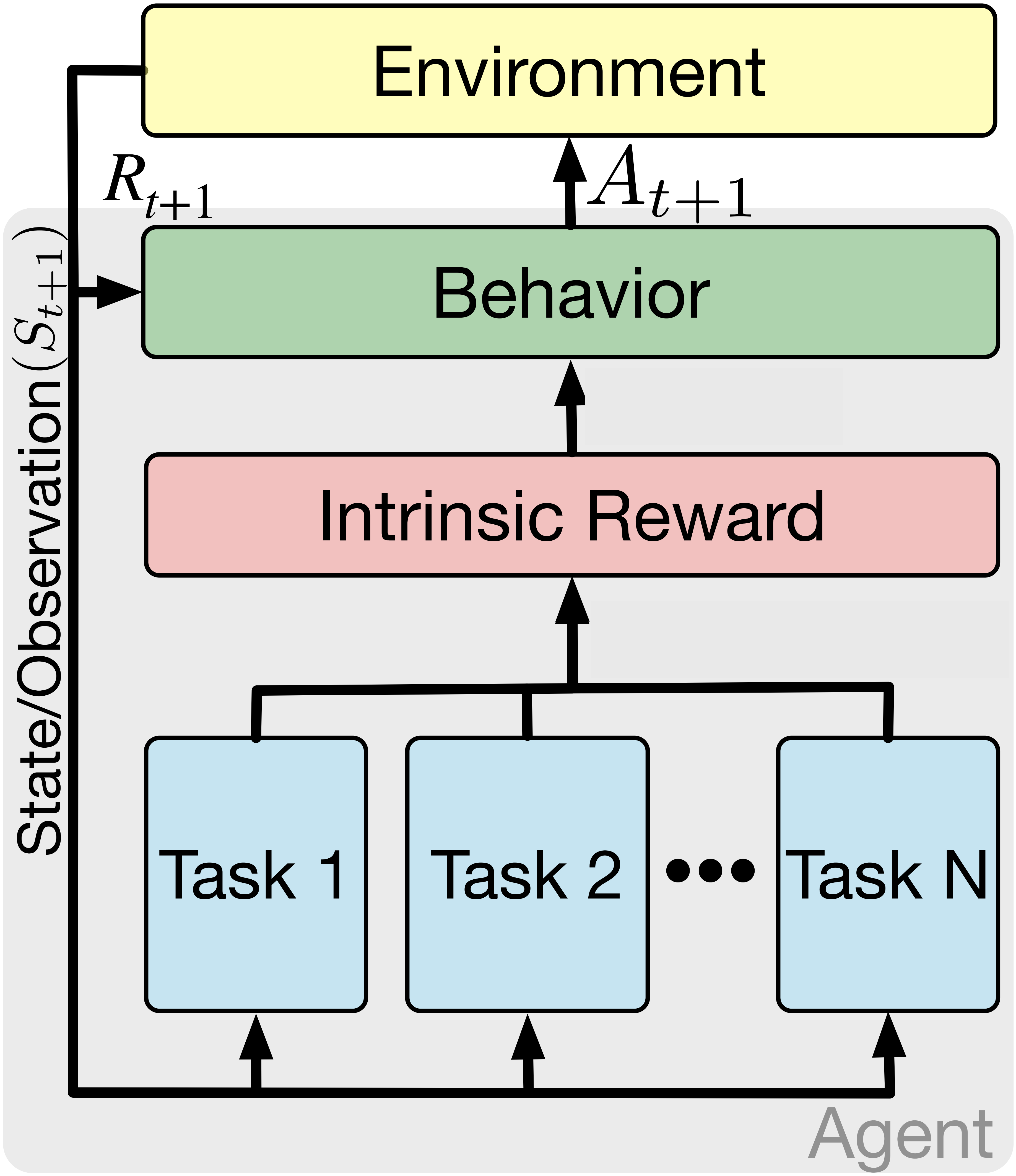
- Assumption: The agent can learn more effectively in a complex world by learning and **re-using modular components** (subtasks)
- Under this assumption, we can ask:
  - how can the agent discover which subtasks are useful?
  - how can the agent learn these subtasks efficiently?

Committing to this Inductive Bias

- Assumption: The agent can learn more effectively in a complex world by learning and re-using modular components
- Under this assumption, we can ask:
  - how can the agent discover which subtasks are useful?
  - **how can the agent learn these subtasks efficiently?**

A Lifelong Learning RL Agent

- The RL agent needs to **adapt behavior** to
  - maximize reward
  - learn about the subtasks, that help maximize reward
- **Intrinsic reward** reflects information gain for subtasks
- Agent maximizes both external reward and intrinsic reward

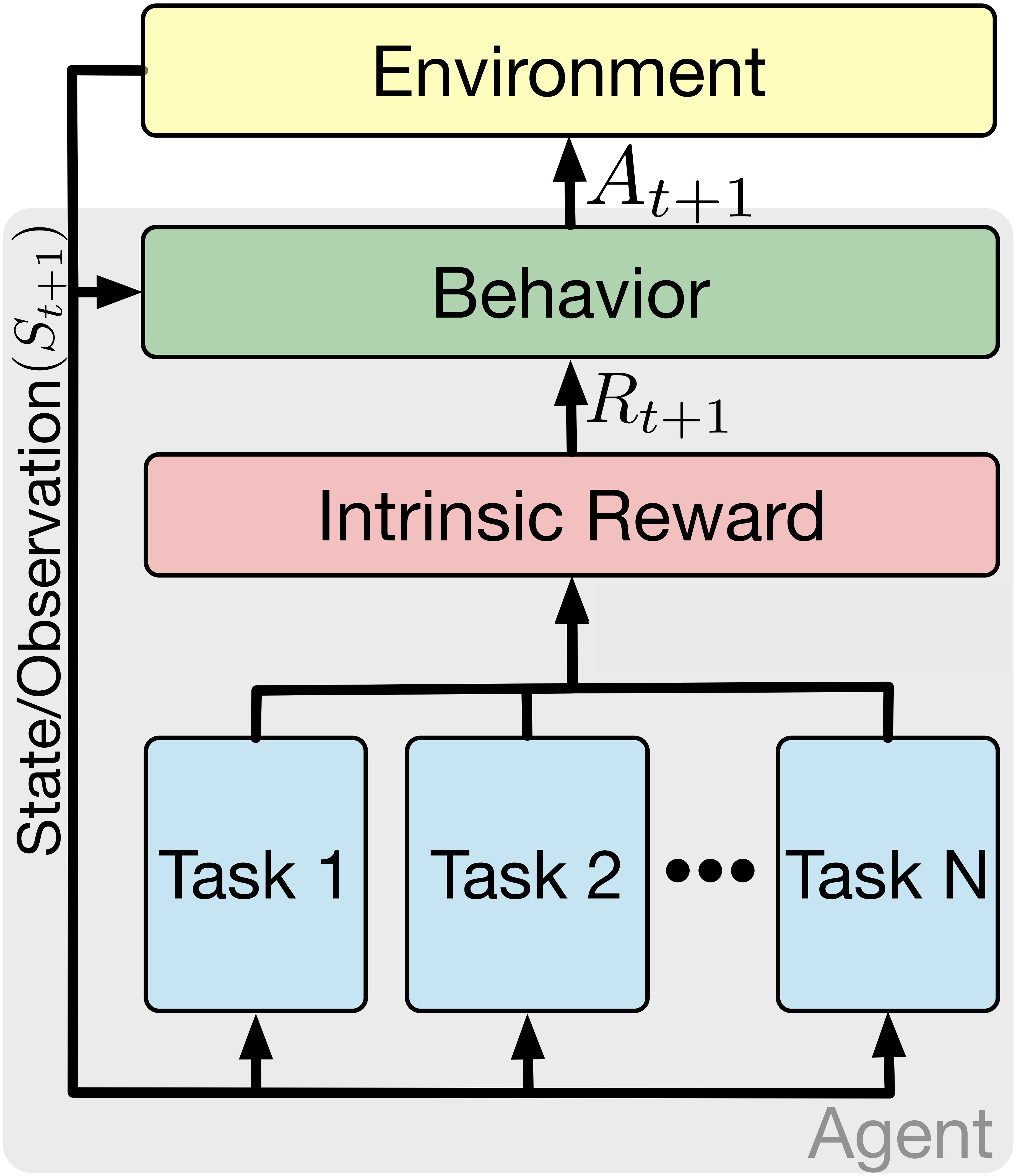


To make traction on this difficult problem, we focus first on understanding how to develop algorithms for maximizing intrinsic reward (subtask learning)

Continual Subtask Learning

No reward from the environment

Focus is on efficiently learning the subtasks

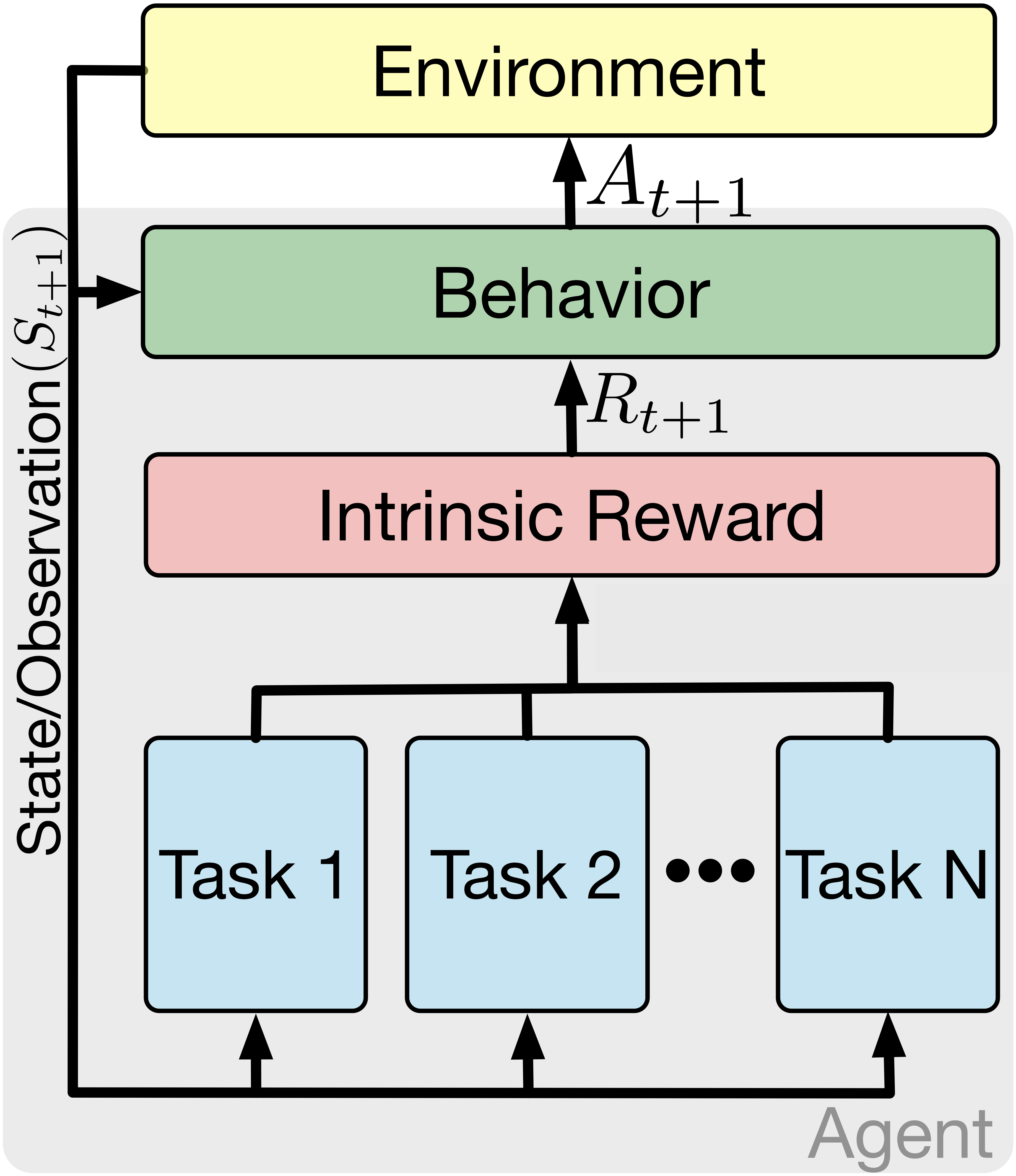


$$R_{t+1} = \sum_{j=1}^{N} R_{t+1}^{j}$$

Continual Subtask Learning

No reward from the environment

Focus is on efficiently learning the subtasks



Technical challenges:

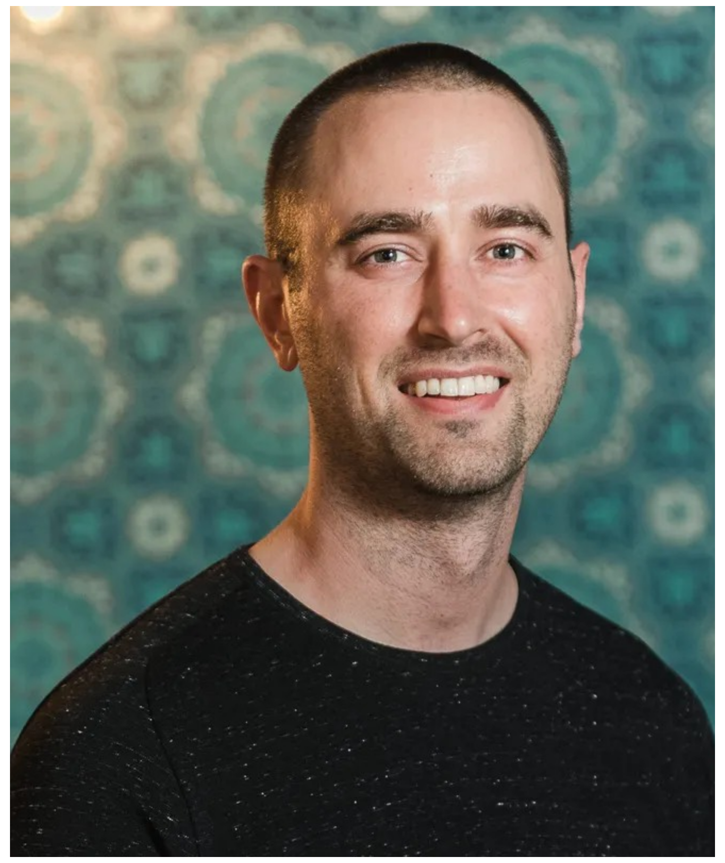
1. Algorithms for the subtasks
2. Intrinsic reward specification
3. Algorithm for the behavior

Let's consider a small experiment in a bandit setting

Let’s consider a small experiment in a bandit setting

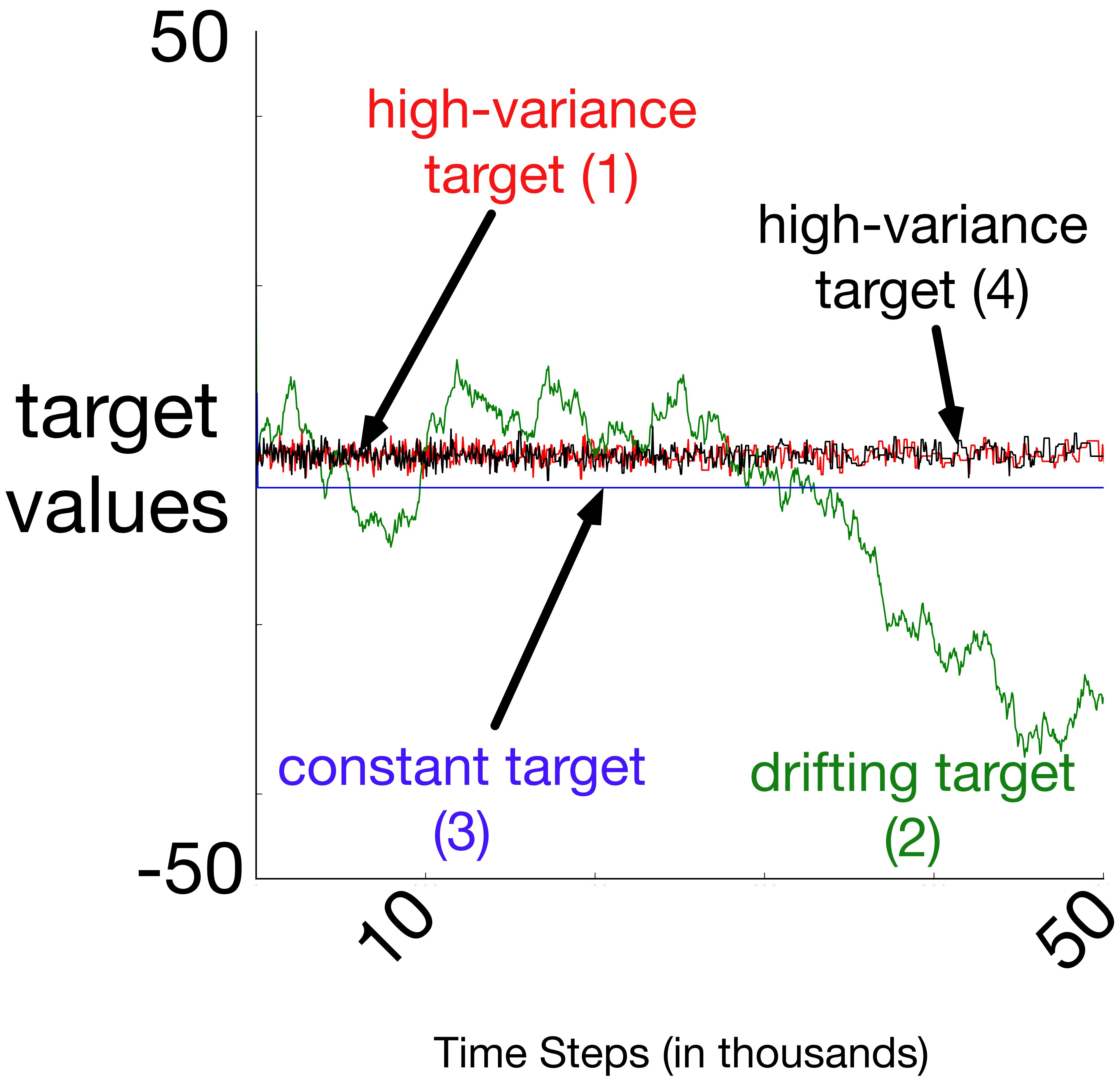
from a larger journal paper in JAIR on understanding intrinsic rewards:
Adapting Behaviour via Intrinsic Reward: A Survey and Empirical Study

primarily with Cam Linke and Adam White



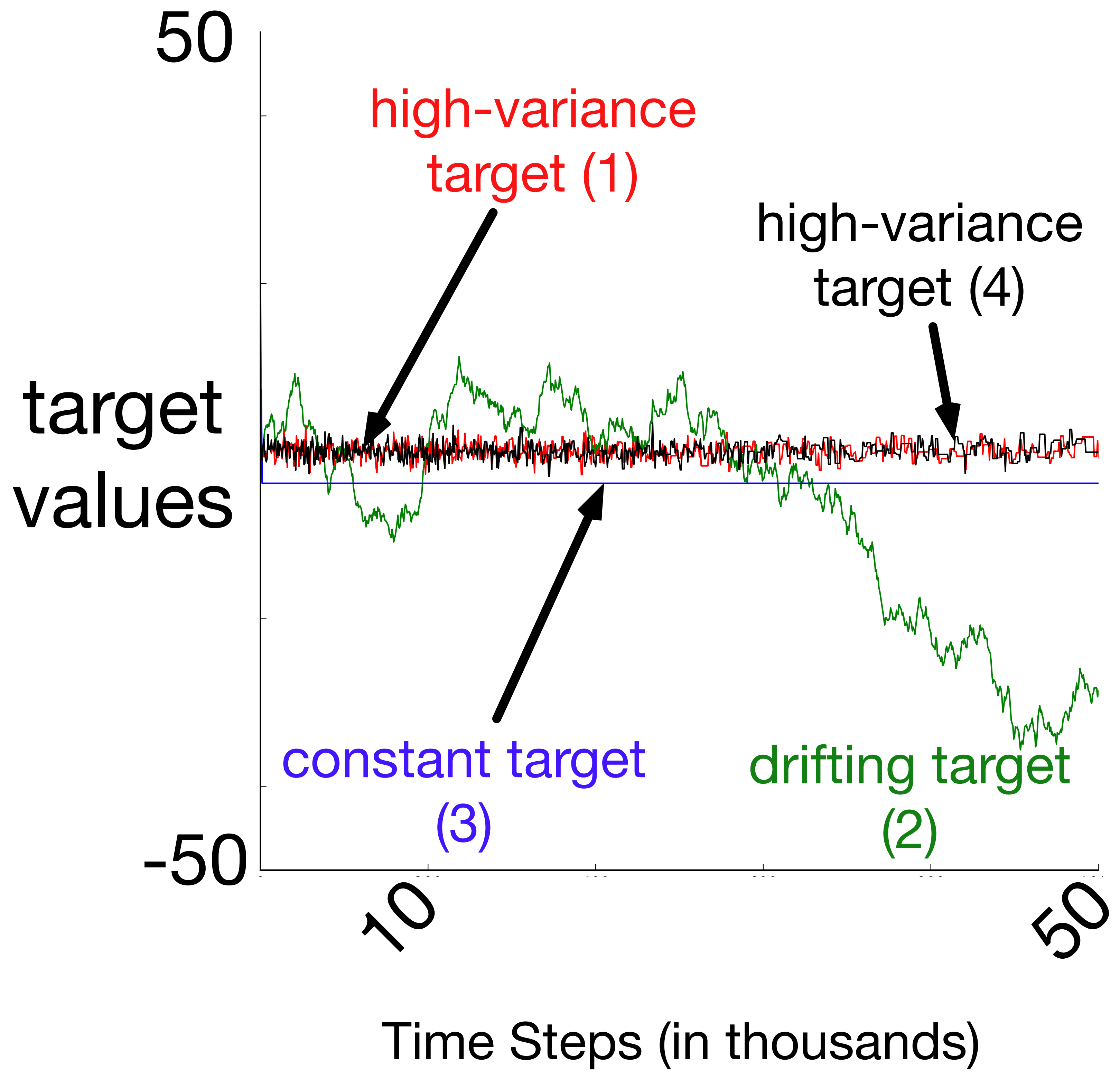
Small Bandit Experiment

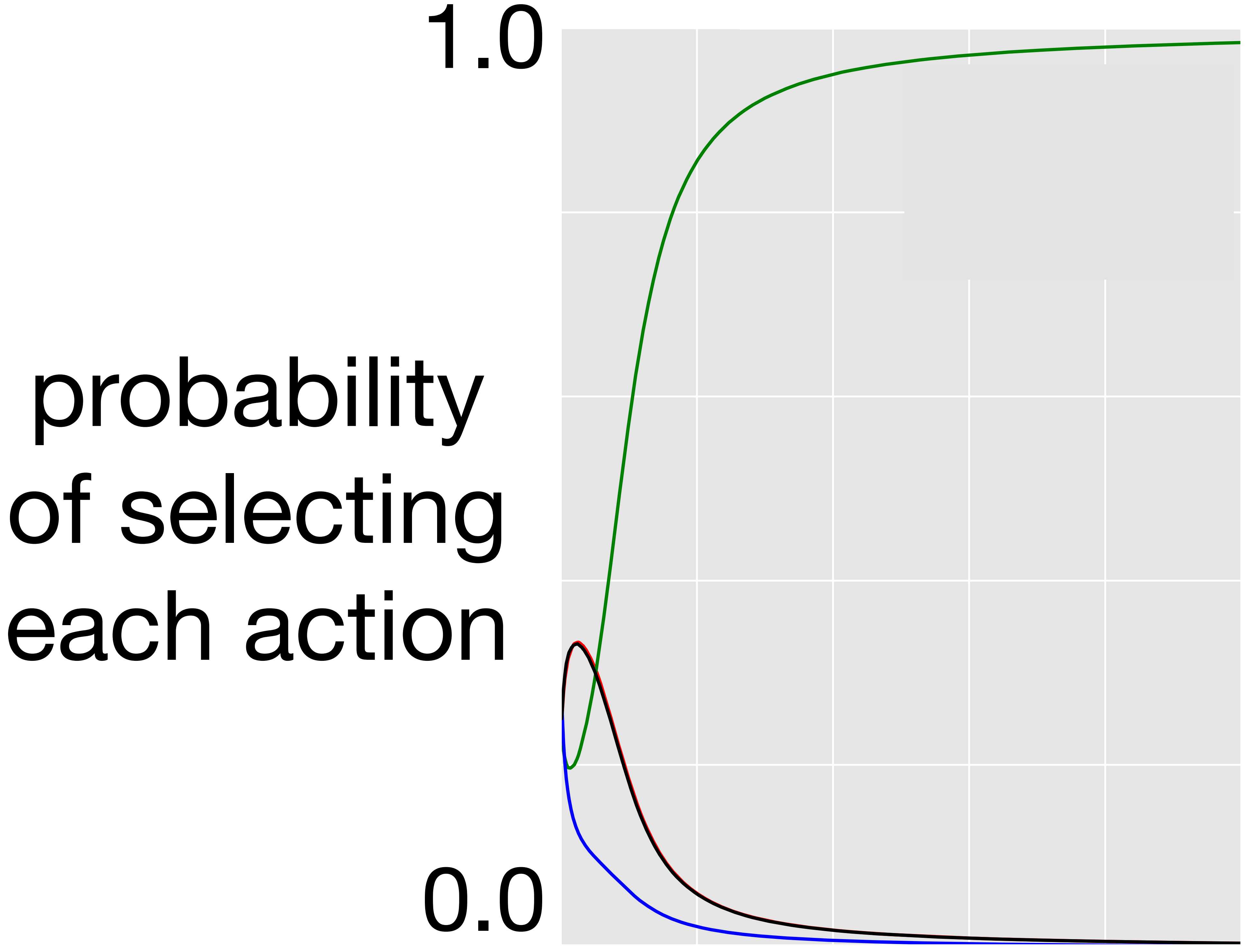
- There is **no context** or state
- Each subtask learner is estimating the **mean** of a **different target**
    - N independent learners (4)
    - $w_{t+1}^j \leftarrow w_t^j - \alpha_t^j (w_t^j - y_t^j)$
- Each **action** only generates **data** for **one** subtask learner
    - there are N actions



Behavior has to learn to balance the needs of all these subtask learners

Example of Good Behavior





Constant can be learned fast, behavior should stop selecting relatively early on
Distractors take longer (due to stochasticity), but also should stop being selected
Drifting needs to be selected forever (non-stationary)

Let’s examine the effect of using two different intrinsic rewards and two different subtask learners and the interactions between these choices

Two Intrinsic Rewards

- **Squared Prediction Error:** $(y_t^j - w_t^j)^2$

$$\mathbb{E}[(Y_t^j - w_t^j)^2] = \mathbb{E}\Big[\underbrace{(Y_t^j - E[Y_t^j])^2}_{\text{stochasticity}} + \underbrace{(E[Y_t^j] - w_t^j)^2}_{\text{amount of learning}}\Big]$$

- **Weight Change:** $|\, w_t^j - w_{t-1}^j \,|$
  - Reflects amount of learning: how much subtask learner adjusted its weights

Two Subtask Learners

- LMS with a **Fixed Stepsize**
  - $w^j_{t+1} \leftarrow w^j_t - \alpha^j (w^j_t - y^j_t)$
- LMS with an **Adaptive Stepsize**
  - $w^j_{t+1} \leftarrow w^j_t - \alpha^j_t (w^j_t - y^j_t)$

Two Subtask Learners

- LMS with a **Fixed Stepsize**
  - $w^j_{t+1} \leftarrow w^j_t - \alpha^j (w^j_t - y^j_t)$
  - cannot modulate learning down, so is not robust to noise
- LMS with an **Adaptive Stepsize**
  - $w^j_{t+1} \leftarrow w^j_t - \alpha^j_t (w^j_t - y^j_t)$
  - can slowly decrease stepsize to average out noise and stop learning

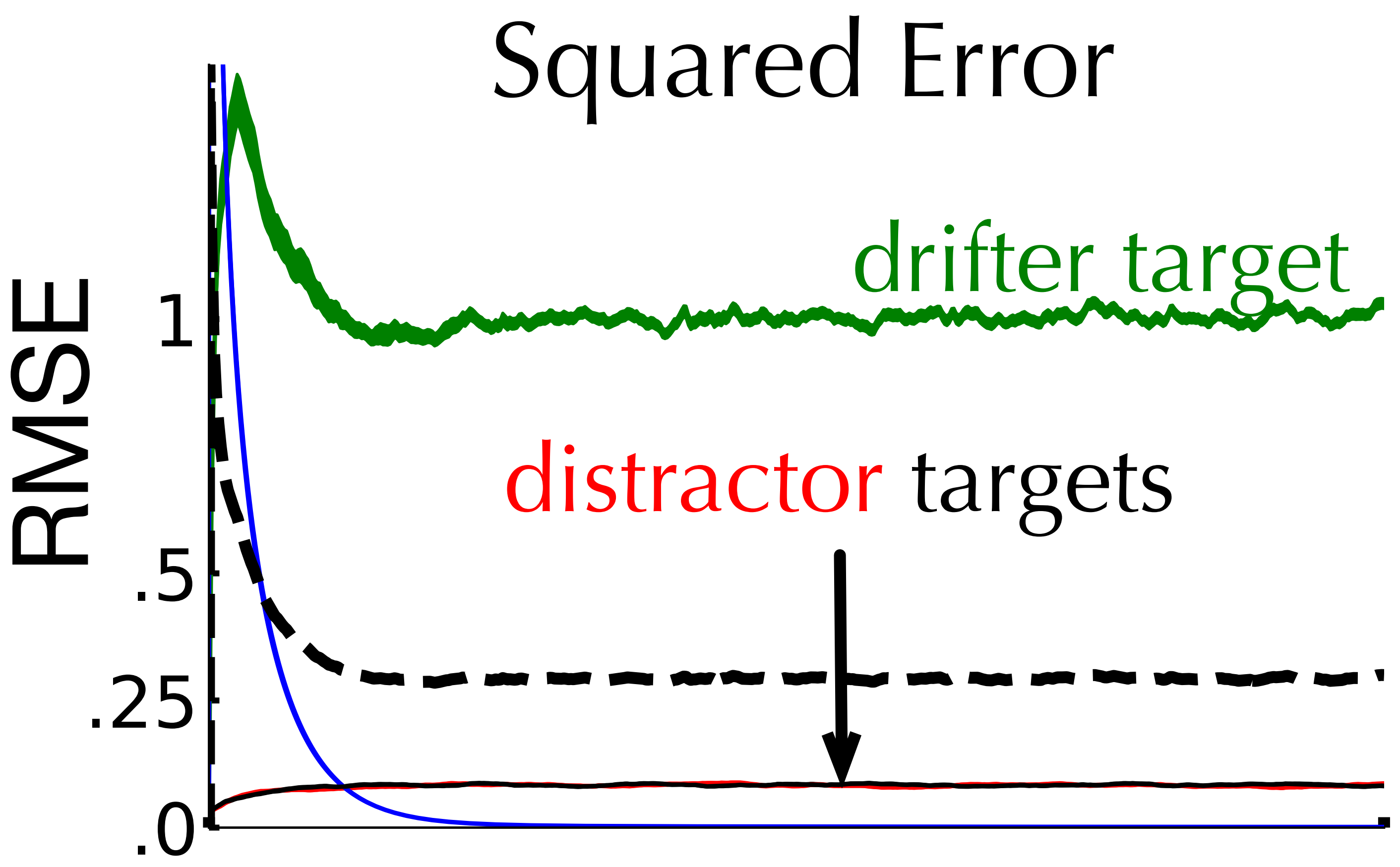
Only Auto+Weight Change Effective

Fixed Stepsize

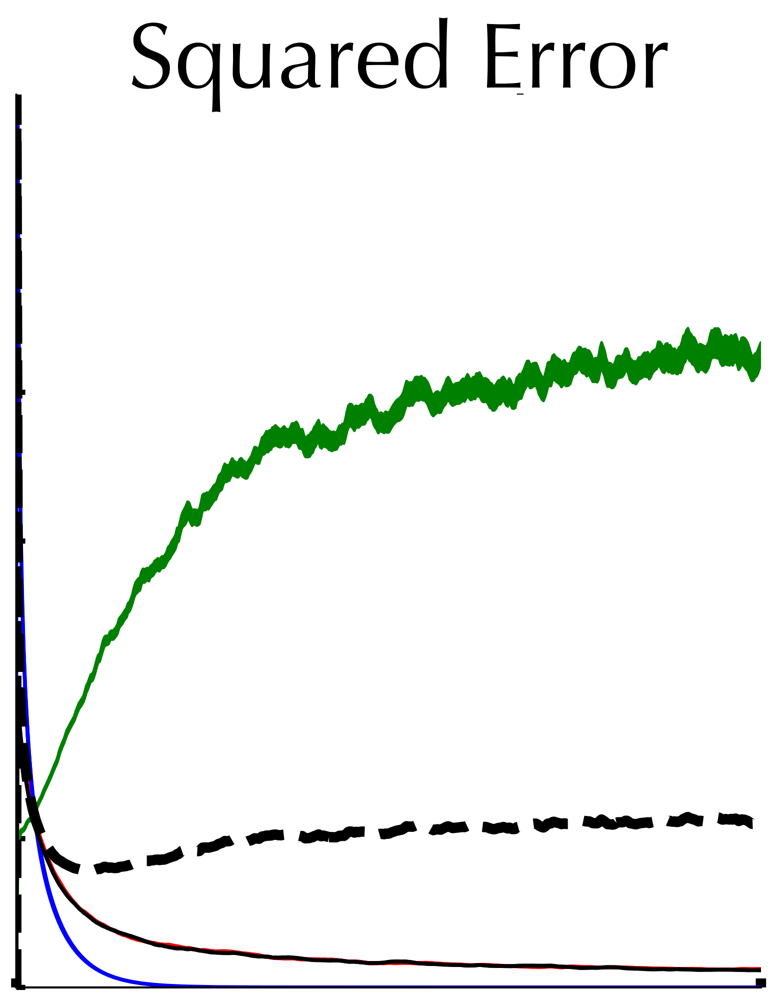
Auto (adaptive stepsize)

Squared Prediction Error

= stochasticity + amount of learning



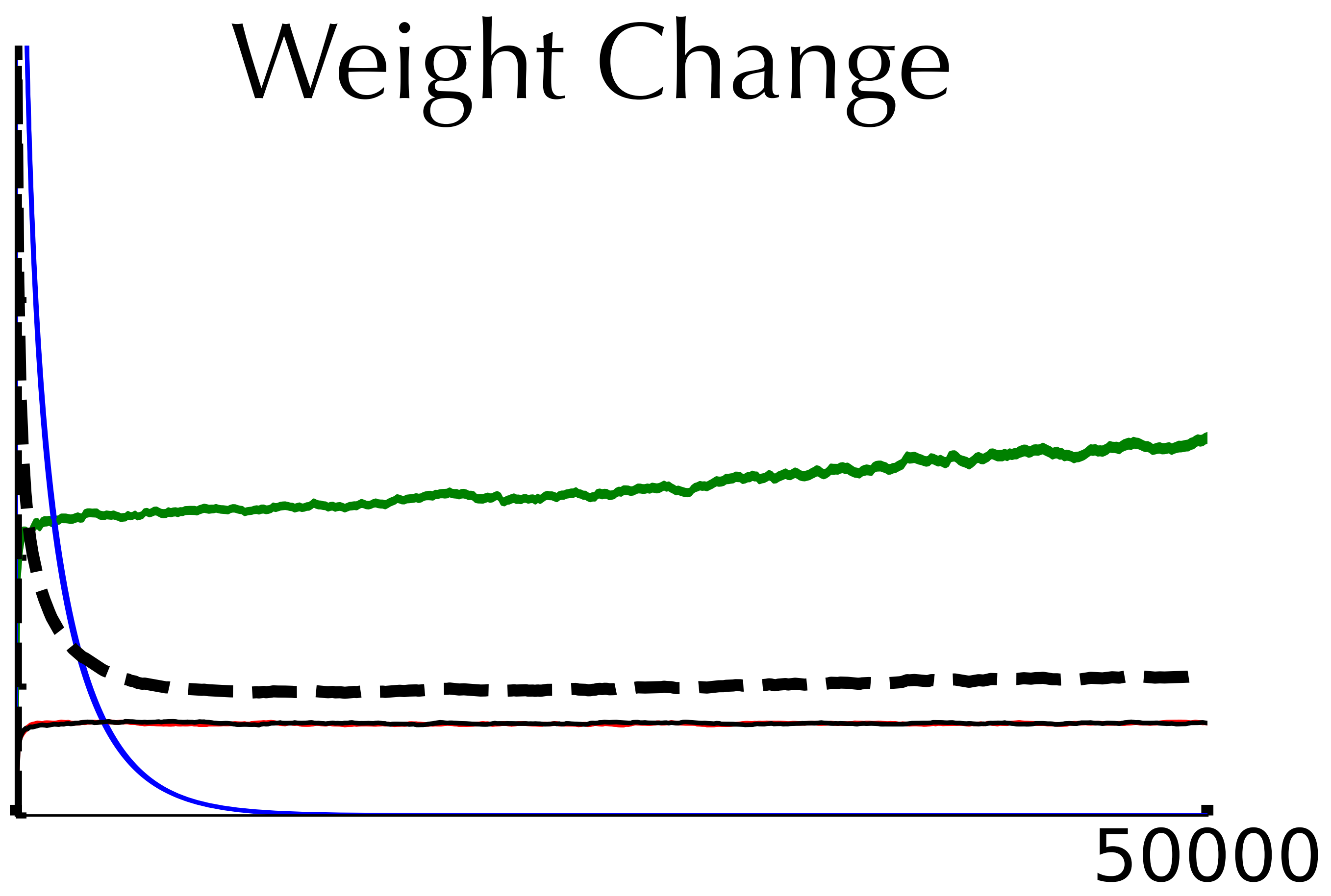
☹️



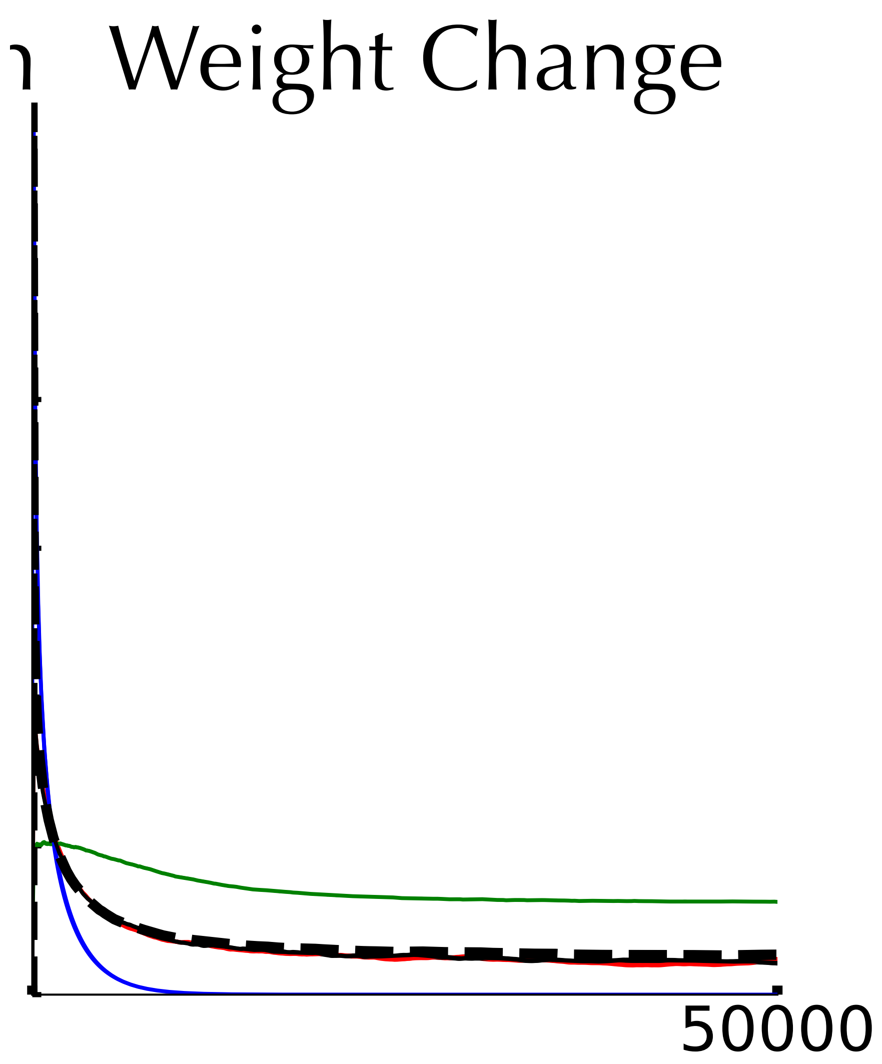
☹️

Weight Change

= amount of learning



☹️



🙂

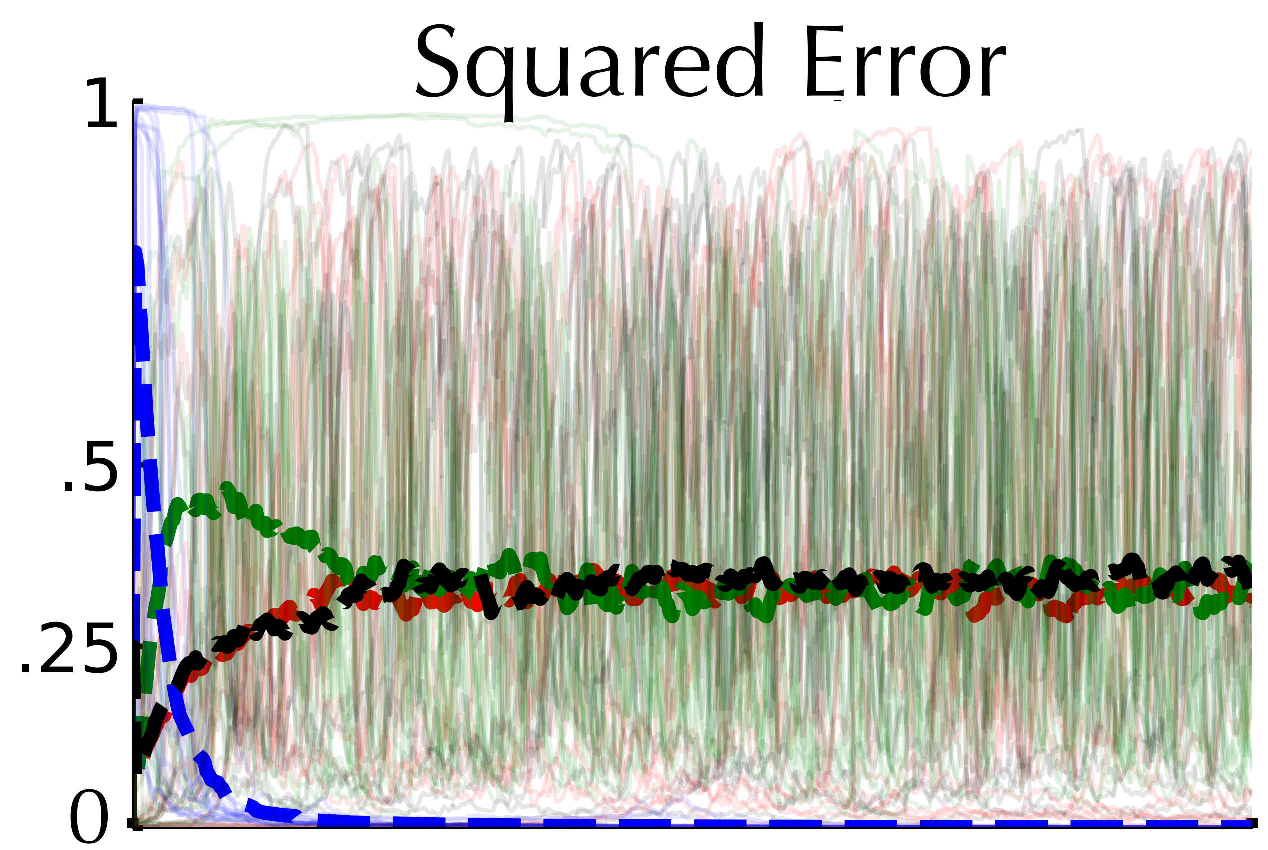
Action Selection Probabilities

Fixed Stepsize

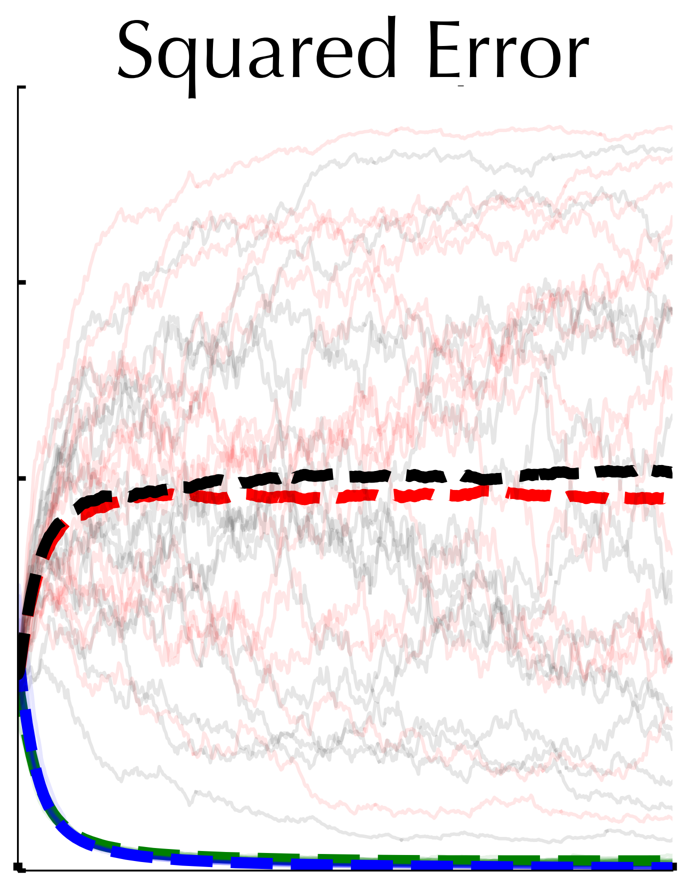
Auto (adaptive stepsize)

Squared Prediction Error

= stochasticity + amount of learning



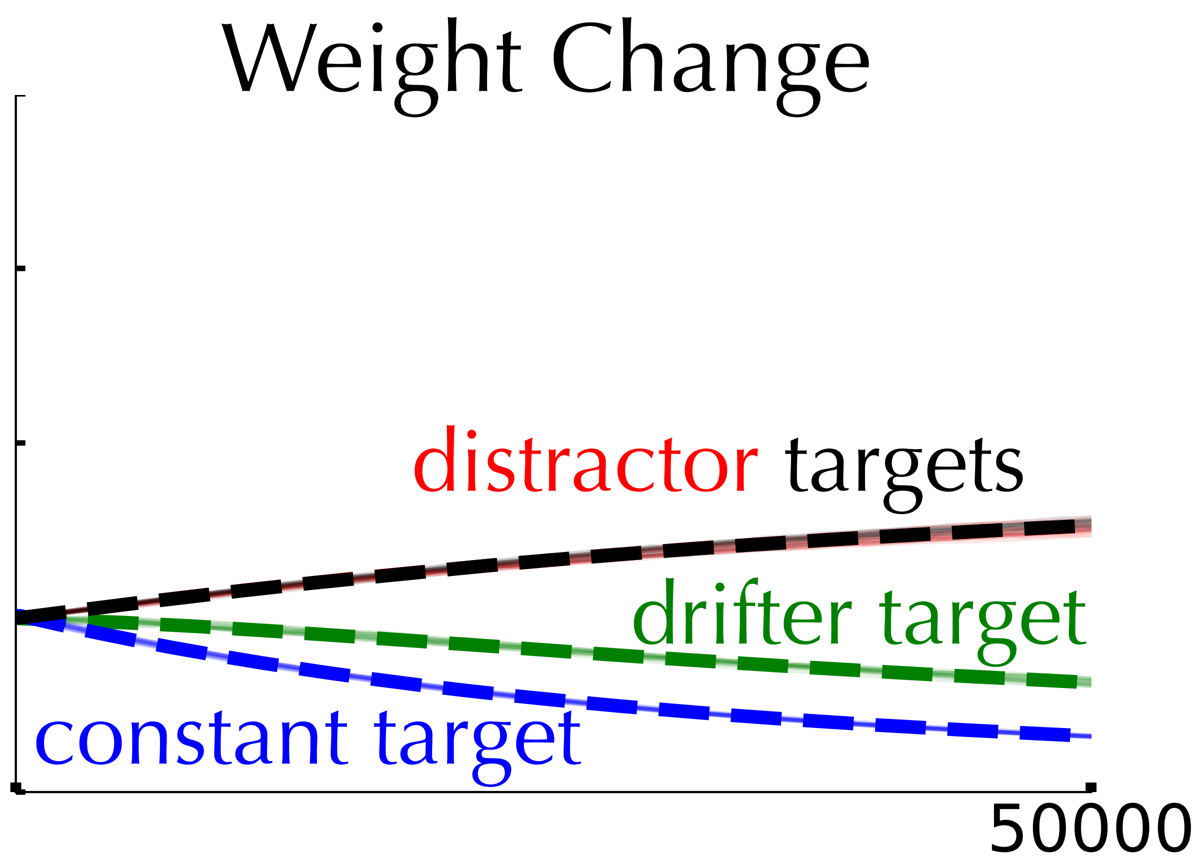
☹️



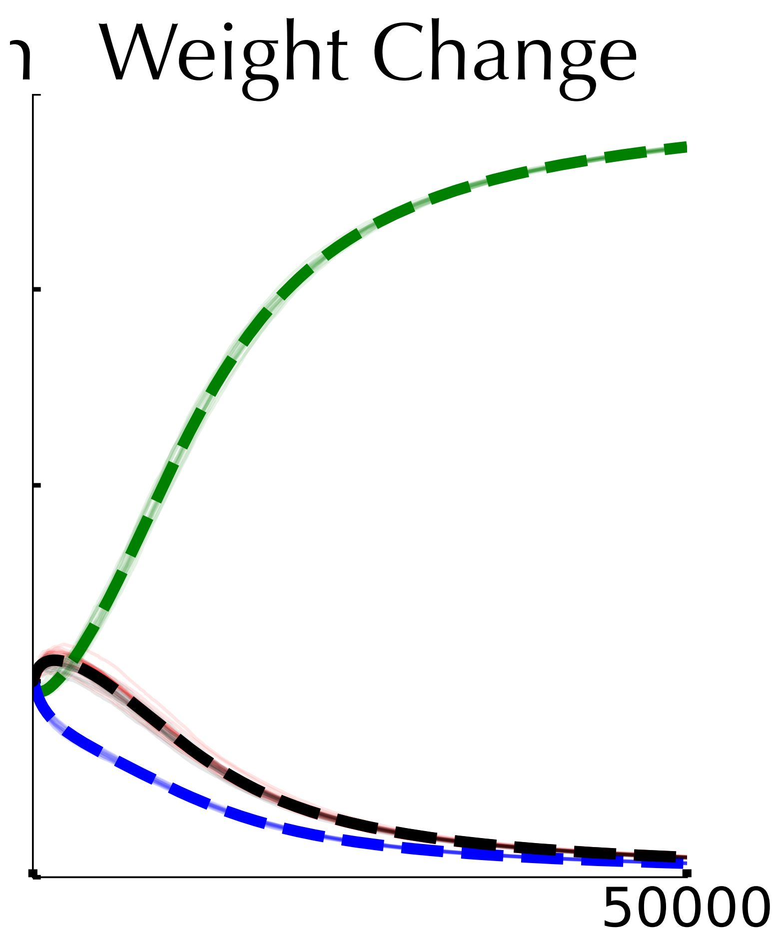
☹️

Weight Change

= amount of learning



☹️



🙂

Key Takeaway 1

- Designing effective Continual Subtask Learning (CSL) systems requires considering interactions between learning components
- For CSL we need:
  - Subtask Learners that modulate their own learning
  - Intrinsic Rewards based on Amount of Learning (not error)

Key Takeaway 1

- Designing effective CSL systems requires considering interactions between learning components
- For CSL we need:
  - Subtask Learners that modulate their own learning
  - Intrinsic Rewards based on Amount of Learning (not error)
- **Open Challenge**: Characterizing ideal subtask learners and intrinsic rewards
  - We show some connection between Weight Change with MAP subtask learners and Information Gain with Bayesian subtask learners

Now coming back to the RL setting

Key Technical Challenge

- **Identify intrinsic rewards** that lead to efficient learning

Key Technical Challenges

- **Identify intrinsic rewards** that lead to efficient learning
  - Current simple strategy is to use Weight Change
  - With sample efficient RL algorithms that use adaptive stepsizes
- Design subtask learners that learn efficiently from **off-policy data**
- The intrinsic rewards are **non-stationary**
  - RL algorithms are designed for stationary rewards

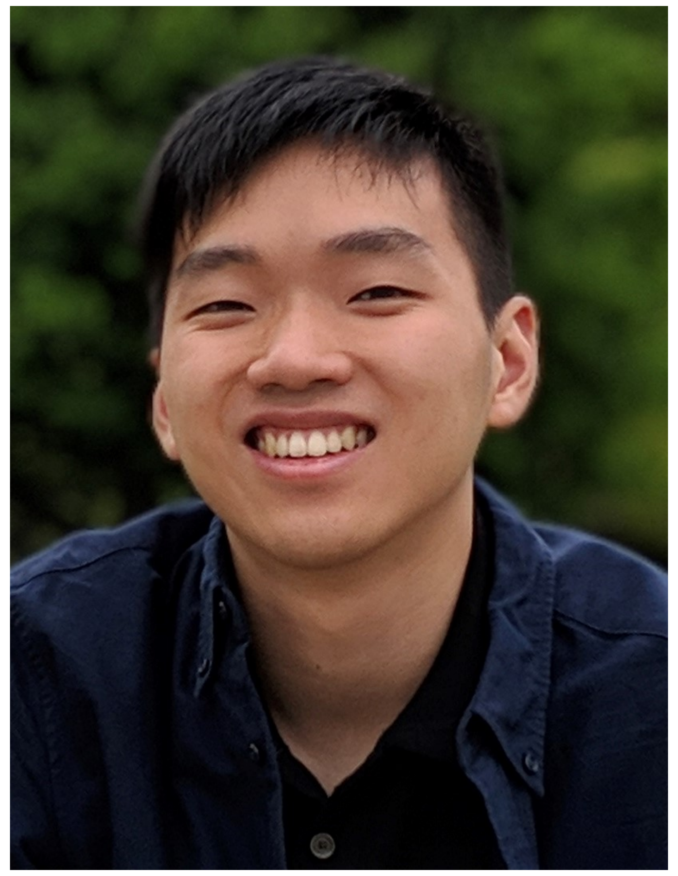
Off-policy Algorithms

- My lab has focused **a lot** on designing effective off-policy algorithms
- See recent journal submission summarizing much of this work
- **"A Generalized Projected Bellman Error for Off-policy Value Estimation in Reinforcement Learning"**, with my PhD student, Andrew Patterson
- **Key Takeaway 2**: We have made a lot of progress on understanding how to make stable off-policy algorithms
    - **Open Challenge**: improving sample efficiency and convergence rates

Let’s focus on the technical challenge of nonstationary rewards

Recent Paper: Continual Auxiliary Task Learning

- **Focus**: Handling non-stationarity in the rewards
- **Key idea**: Use **Successor Features** to learn stationary feature information and only track changing rewards



Chunlok Lo

Matt Schlegel

Raksha Kumaraswamy

Adam White

Defining Successor Features

- Let $\mathbf{x}(s, a)$ be the features for state-action pair (s, a)
- The **successor features** ψ are the cumulative, discounted sum of the features when following policy π
- $$\begin{aligned}\psi(s, a) &= \mathbb{E}_\pi[\mathbf{x}(S_t, A_t) + \gamma \mathbf{x}(S_{t+1}, A_{t+1}) + \gamma^2 \mathbf{x}(S_{t+2}, A_{t+2}) + \ldots \mid S_t = s, A_t = a] \\ &= \mathbb{E}_\pi[\mathbf{x}(S_t, A_t) + \gamma \psi(S_{t+1}, A_{t+1}) \mid S_t = s, A_t = a]\end{aligned}$$
- This recursive form looks **just** like a value function (simply vector-valued)
  - ψ can be learned using any value function learning approach

Why Are Successor Features Useful?

- If the rewards are linear in the features, $r(s, a) = \mathbf{x}(s, a)^\top \mathbf{w}*$
- Then the action-values for a policy π can be obtained immediately with the SF

$$Q^\pi(s, a) = \psi(s, a)^\top \mathbf{w}*$$

To See Why…

$$r(s,a) = \mathbf{x}(s,a)^\top \mathbf{w}^*$$

$$\begin{aligned}\psi(s,a)^\top \mathbf{w}^* &= \mathbb{E}_\pi[\mathbf{x}(S_t, A_t)^\top \mathbf{w}^* + \gamma \mathbf{x}(S_{t+1}, A_{t+1})^\top \mathbf{w}^* + \ldots \,|\, S_t = s, A_t = a] \\ &= \mathbb{E}_\pi[r(S_t, A_t) + \gamma r(S_{t+1}, A_{t+1}) + \ldots \,|\, S_t = s, A_t = a] \\ &= Q^\pi(s,a)\end{aligned}$$

Why Are Successor Features Useful?

- If the rewards are linear in the features, $r(s, a) = \mathbf{x}(s, a)^\top \mathbf{w}*$
- Then the action-values for a policy π can be immediately with the SF using

$$Q^\pi(s, a) = \psi(s, a)^\top \mathbf{w}*$$

- To estimate $Q^\pi(s, a)$, we only need to solve a regression problem and learn weights $\mathbf{w}$ such that $r(s, a) \approx \mathbf{x}(s, a)^\top \mathbf{w}$, to get

$$\hat{q}(s, a) = \psi(s, a)^\top \mathbf{w}$$

Wait, This Seems Worse

- If the rewards are linear in the features, $r(s,a) = \mathbf{x}(s,a)^\top \mathbf{w}^*$
- To estimate $Q^\pi(s,a)$, we only need to solve a regression problem and learn weights $\mathbf{w}$ such that $r(s,a) \approx \mathbf{x}(s,a)^\top \mathbf{w}$
- We've **exchanged** the easier problem of directly estimating $Q^\pi(s,a)$ with estimating $\psi(s,a)$ which outputs a vector of the same size as $\mathbf{x}(s,a)$

When Are Successor Features Useful?

- If the rewards are linear in the features, $r(s, a) = \mathbf{x}(s, a)^\top \mathbf{w}*$
- To estimate $Q^\pi(s, a)$, we only need to solve a regression problem and learn weights $\mathbf{w}$ such that $r(s, a) \approx \mathbf{x}(s, a)^\top \mathbf{w}$
- But now we've **exchanged** the easier problem of directly estimating $Q^\pi(s, a)$ with estimating $\psi(s, a)$
- This **effort** is only worth it if we get to **re-use** $\psi(s, a)$

SF Is Useful When Rewards Are Nonstationary

- Tracking (slowly) changing rewards is fundamentally simpler than tracking the resulting changing value function: if new $\tilde{r}(s,a) = \mathbf{x}(s,a)^\top(\mathbf{w}^* + \epsilon)$

$$\psi(s,a)^\top(\mathbf{w}^* + \epsilon) = Q^\pi(s,a) + \mathbb{E}_\pi[\epsilon(S_t, A_t) + \gamma + \epsilon(S_{t+1}, A_{t+1}) + \ldots \,|\, S_t = s, A_t = a]$$

- where $\epsilon(s,a) = \mathbf{x}(s,a)^\top \epsilon$

SF Is Useful When Rewards Are Nonstationary

- Tracking (slowly) changing rewards fundamentally simpler than tracking the resulting changing value function: if new $\tilde{r}(s, a) = \mathbf{x}(s, a)^\top(\mathbf{w}^* + \epsilon)$

$$\psi(s, a)^\top(\mathbf{w}^* + \epsilon) = Q^\pi(s, a) + \mathbb{E}_\pi[\epsilon(S_t, A_t) + \gamma + \epsilon(S_{t+1}, A_{t+1}) + \ldots \mid S_t = s, A_t = a]$$

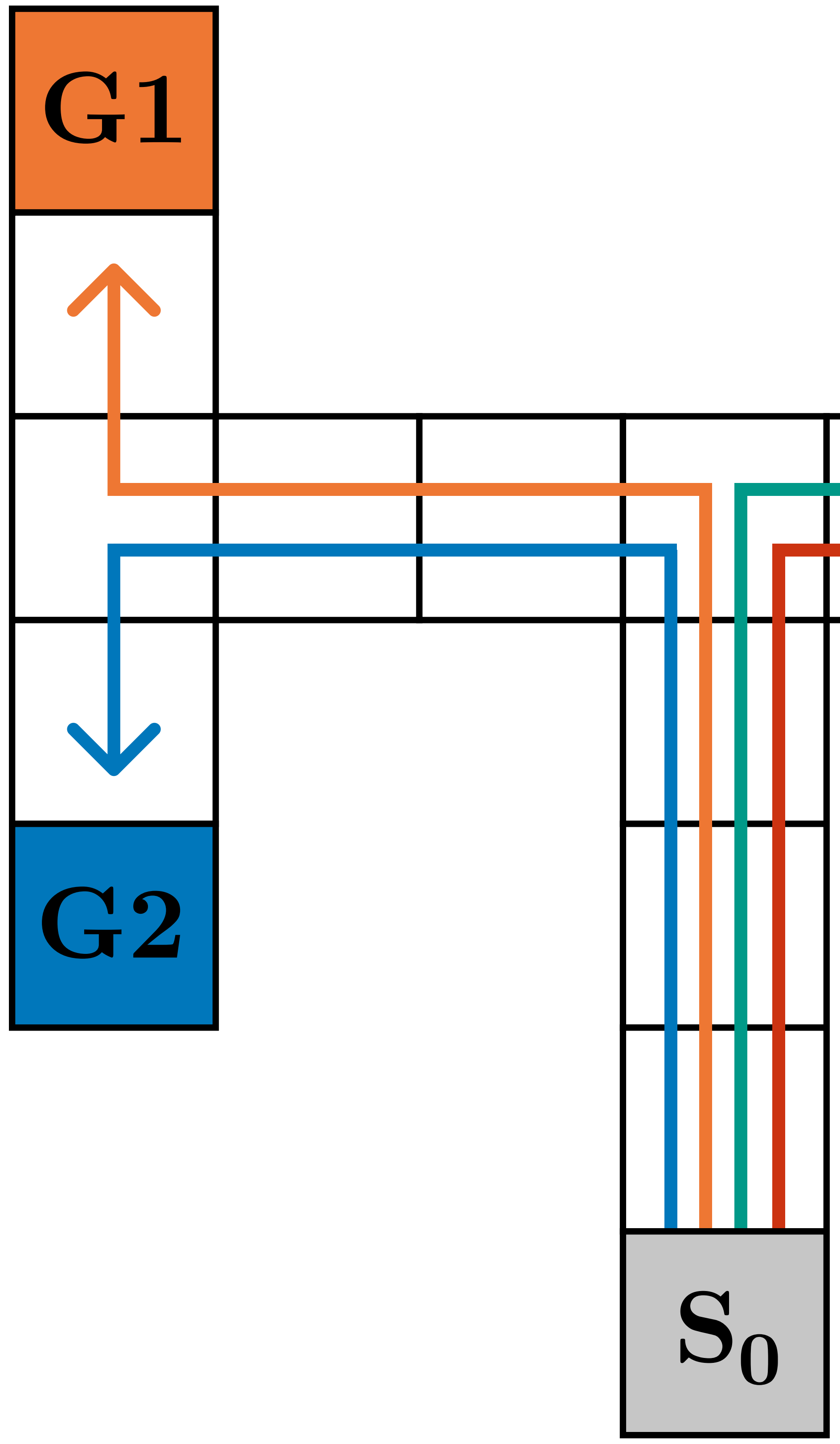
- where $\epsilon(s, a) = \mathbf{x}(s, a)^\top \epsilon$

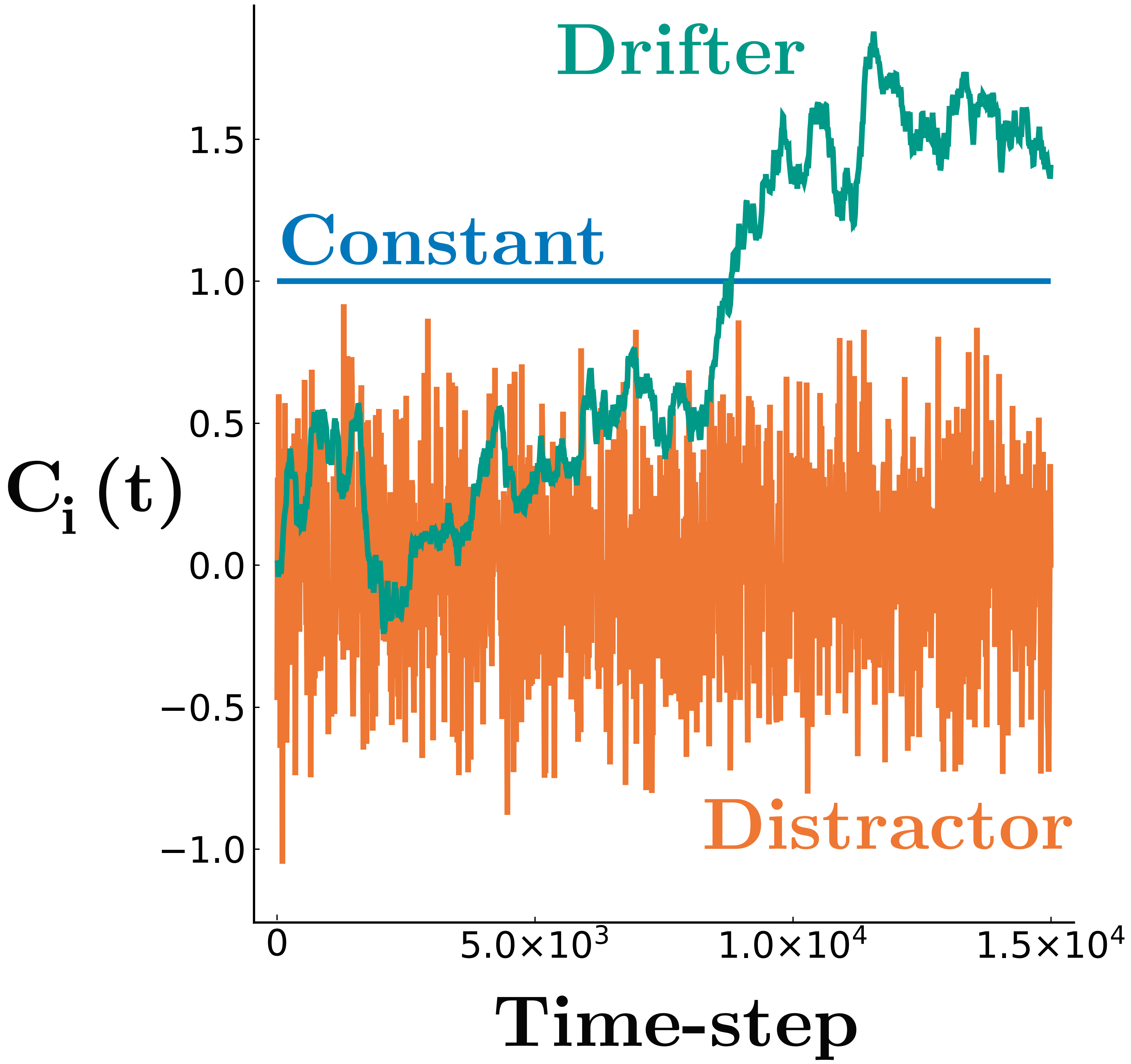
- **Result in paper formalizing this intuition**: convergence rate for value estimation with SF when estimating $\mathbf{w}^*$ is better than known convergence rate for TD-based value estimation algorithms

Let's test out this idea

Both subtask learners and behavior learn value functions
Both can leverage SF for non-stationary signals

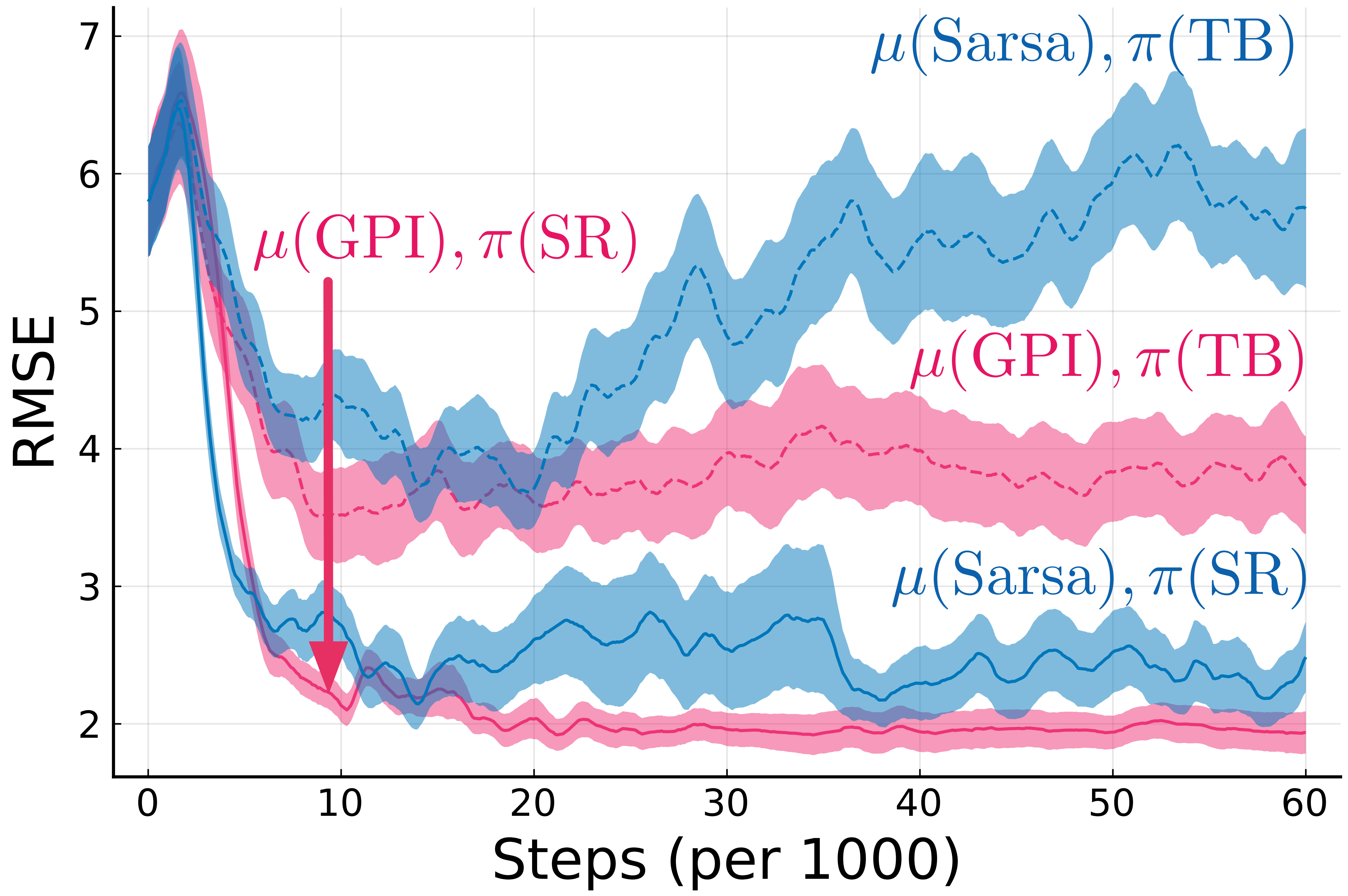
An Experiment in the T-Maze





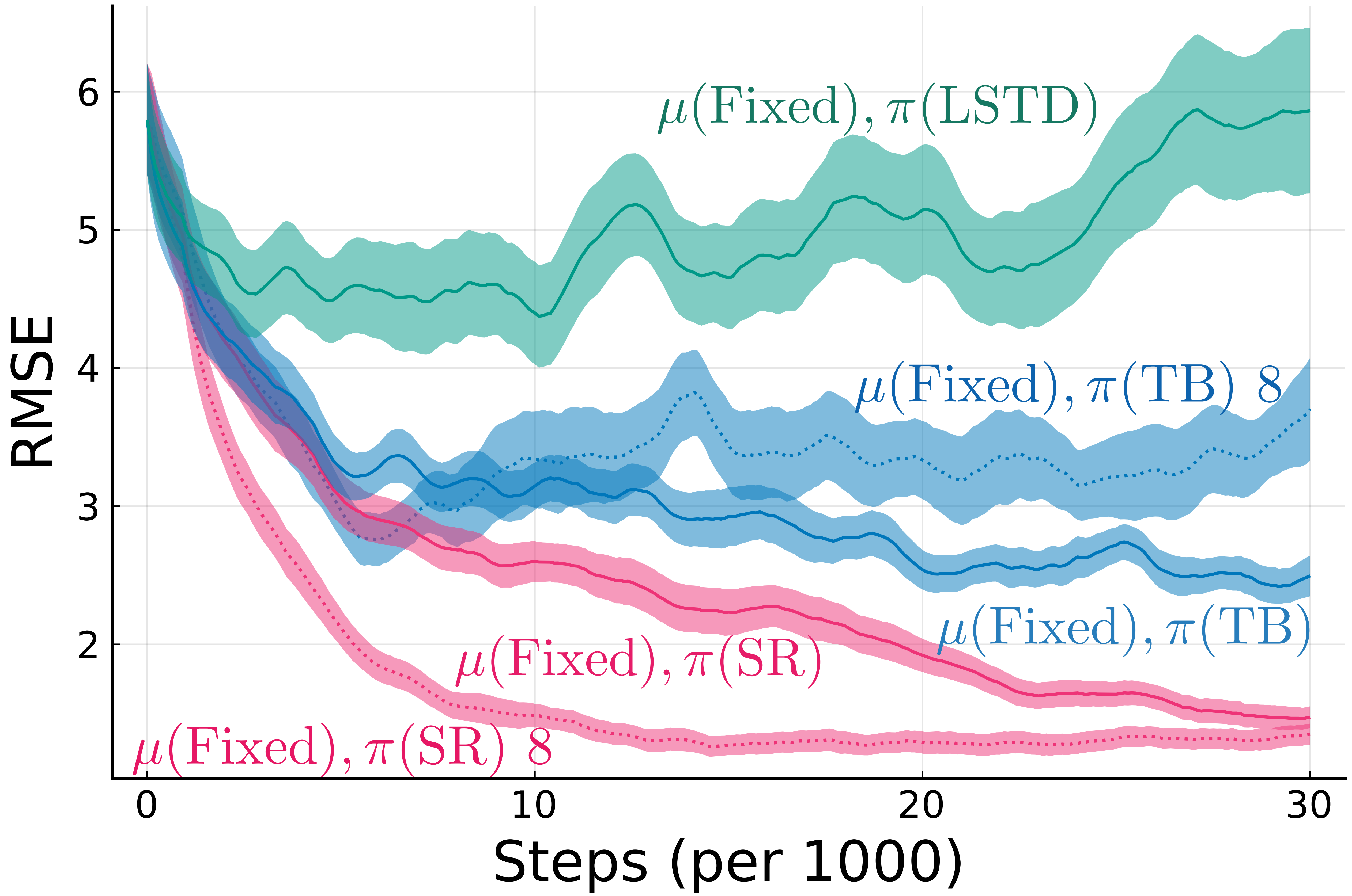
One subtask per cumulant (constant at G2 and G4, drifter at G3, distractor at G1)

SF Improves Performance for both the Behavior and Subtask Learners



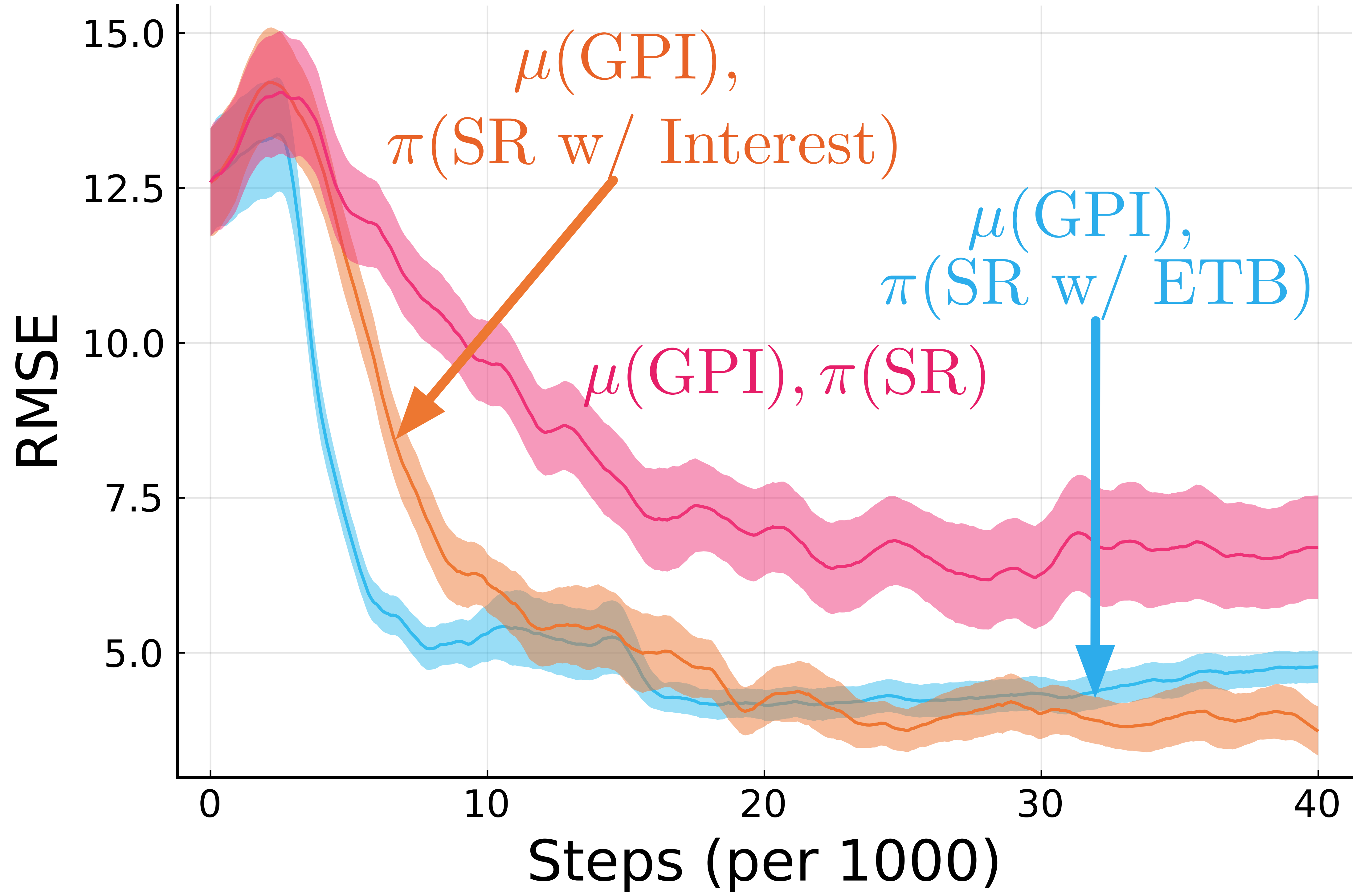
- μ(GPI) = behavior uses SF
- π(SR) = subtask uses SF
- μ(Sarsa) = behavior uses Sarsa
- π(TB) = subtask uses Tree-Backup, a sample efficient off-policy algorithm designed for stationary rewards

Improving Sample Efficiency of Subtask Learners with Replay

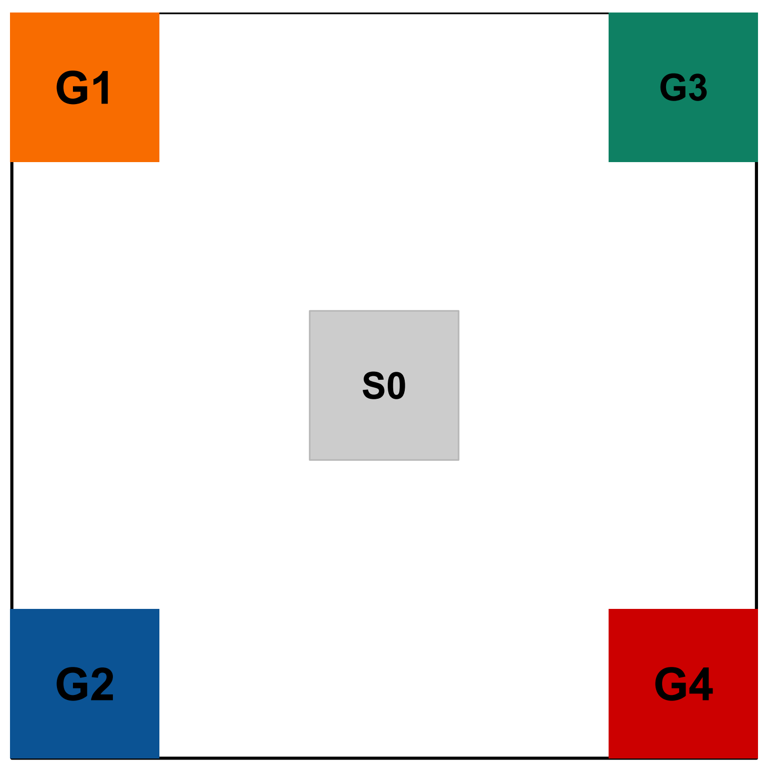


- Labels with "8" means we use 8 replay steps (8x more updates)
- Replay does not interface well with non-stationary data
- Rewards are stale in the buffer
- π(SR) uses replay only for stationary SF part, updates reward model online

Incorporating Off-Policy Ideas Also Helps



- Result in an Open 2-D World
- Add Emphatic Weightings (corrects bias in distributions)
- Add interest (focuses function approximation on a subset of states, counterfactual reasoning everywhere not feasible)



The Biggest Limitation of the Approach

- Choice of reward features $\mathbf{x}(s, a)$ critical
- More compact features are much more computationally efficient
    - $\psi(s, a)$ is a function approximator that input (s,a) and outputs a vector of the same size as $\mathbf{x}(s, a)$
- If reward features generalize too much, then this skews the value estimate
    - SF was mostly useful for the nonstationary cumulant in the subtask learners, where it was easy to hand design good $\mathbf{x}(s, a)$

The Biggest Limitation of the Approach

- Choice of reward features $\mathbf{x}(s, a)$ critical
- More compact features are much more computationally efficient
  - $\psi(s, a)$ is a function approximator that input (s,a) and outputs a vector of the same size as $\mathbf{x}(s, a)$
- If reward features generalize too much, then this skews the value estimate
  - SF was mostly useful for the nonstationary cumulant in the subtask learners, where it was easy to hand design good $\mathbf{x}(s, a)$
- **Open challenge**: learn reward features for SF, taking into consideration impacts on the value estimate accuracy

Summary of the Talk

- **Point 1:** General purpose agents (including for applications) require the system to be built with subtask learning in mind
- **Point 2:** The Continual Subtask Learning (CSL) problem formalizes the problem of efficiently learning many subtasks in parallel, off-policy
- **Point 3:** Key points to consider when designing CSL agents:
  - critical to have sample efficient subtask learners that can modulate learning
  - rewards are always non-stationary (since they reflect learning); the behavior algorithm should be designed to handle this non-stationarity

Key Algorithmic Insights

- Off-policy algorithms are mature enough to help us move forward in CSL
    - but improving them further can have significant impacts on improving these systems due to complex interactions
- Successor Features facilitate handling non-stationary cumulants and rewards
- Weight Change is a simple, but effective Intrinsic Reward

Key Algorithmic Insights

- Off-policy algorithms are mature enough to help us move forward in CSL
  - but improving them further can have significant impacts on improving these systems due to complex interactions
- Successor Features facilitate handling non-stationary cumulants and rewards
- Weight Change is a simple, but effective Intrinsic Reward
- …And there is much more to do!
  - (1) theoretical connection between maximizing intrinsic reward and optimal learning of subtasks, (2) better subtask and behavior learners,
    (3) incorporating environment rewards, (4) utility in applications,
    (5) discovery of subtasks, …

Thank you!