Fast **TRAC**

A Parameter-free Optimizer for Lifelong Reinforcement Learning

Aneesh Muppidi, Zhiyu Zhang, Heng Yang

Harvard John A. Paulson
School of Engineering
and Applied Sciences

COMPUTATIONAL ROBOTICS

Lifelong Reinforcement Learning

In the lifelong setting, an agent is always adapting to new tasks or distribution shifts

Task: walk

…

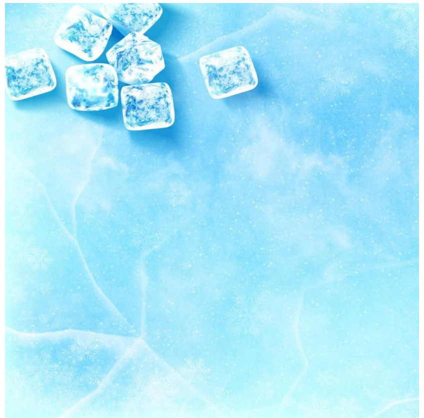
shift 1

…

shift 2

…

shift 3



Procgen

L1

L2

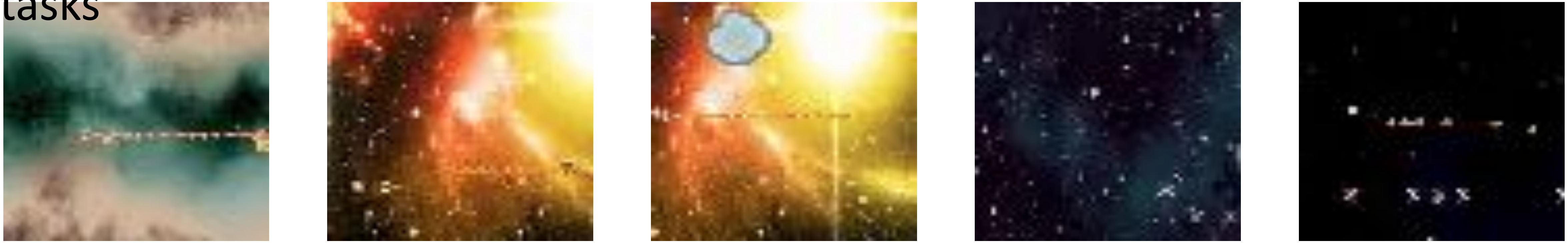
L3

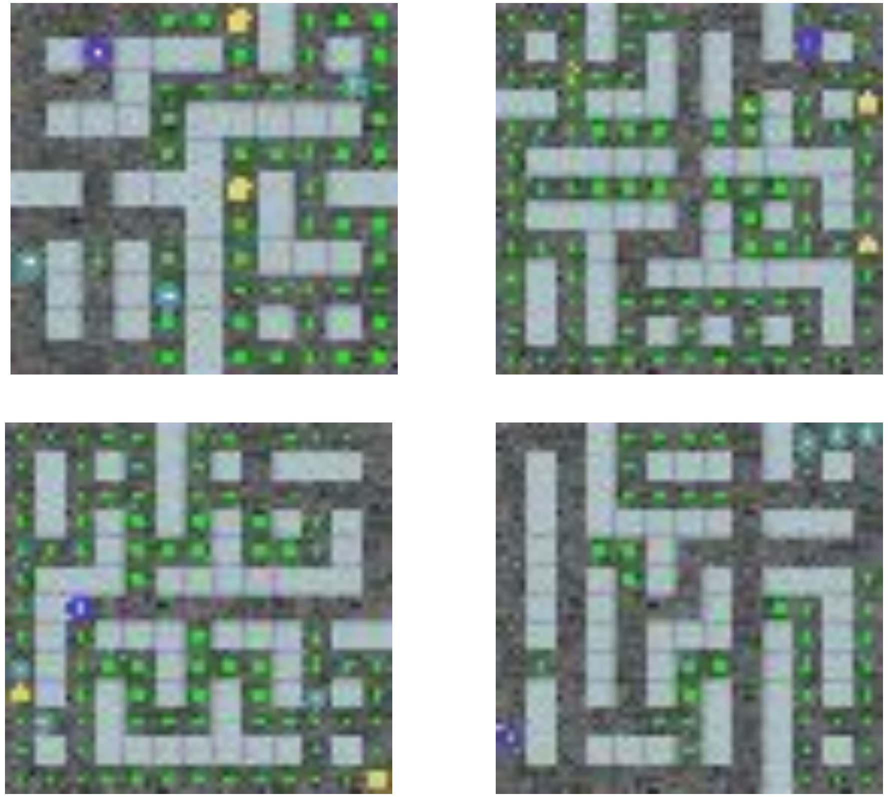
L4

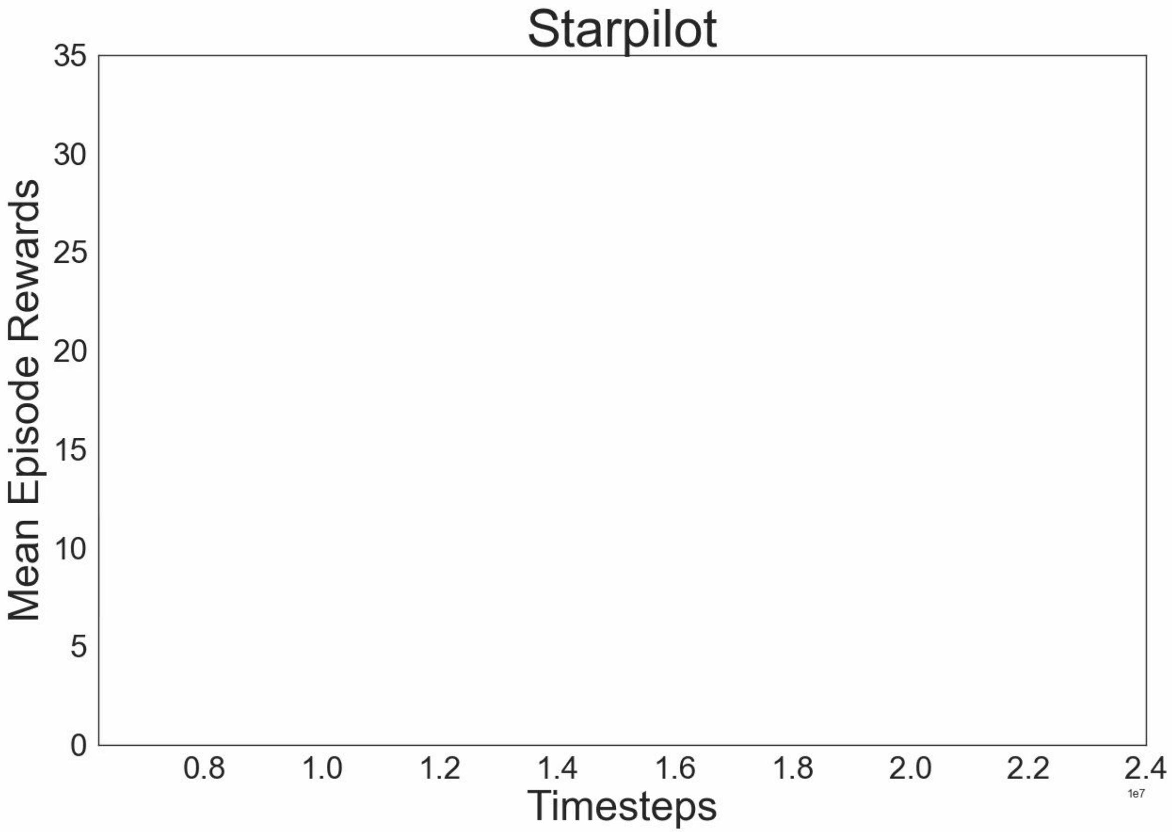
…LN

- similar, but different reward functions, transitions, obstacles, dynamics…

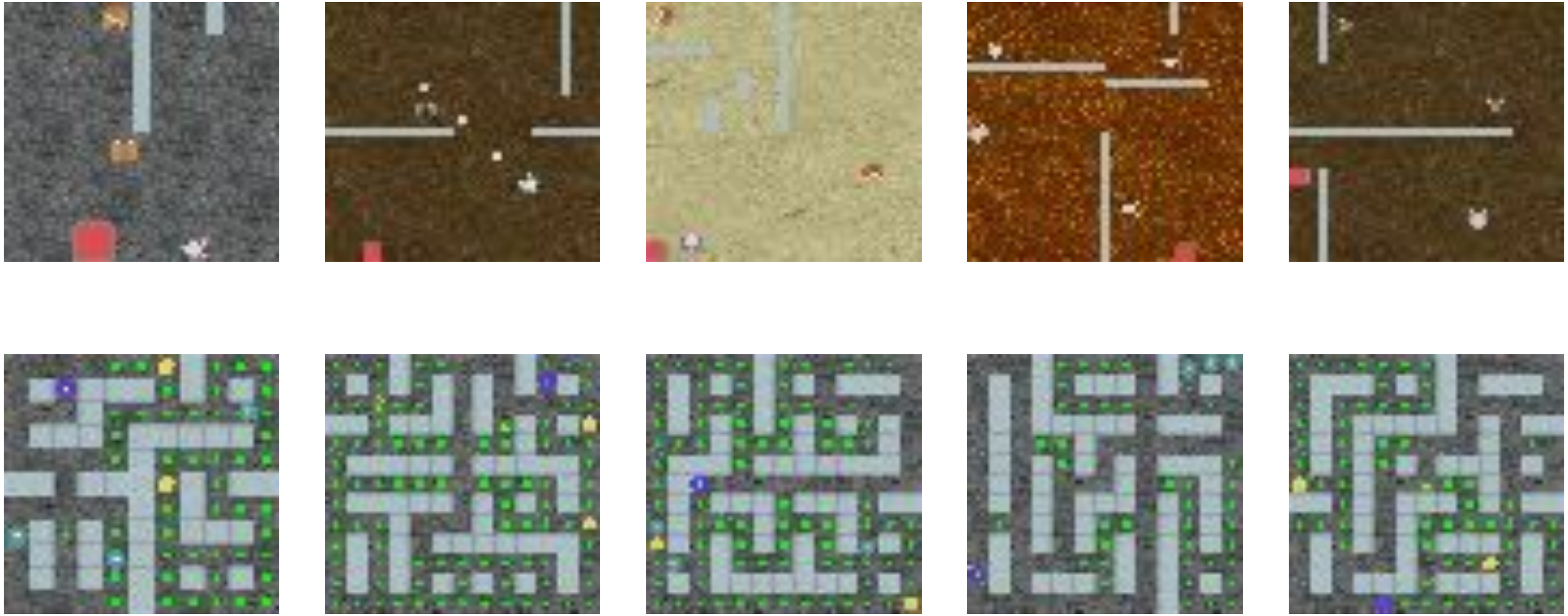
Loss of plasticity arises from adapting to a **strictly ordered** sequence of tasks



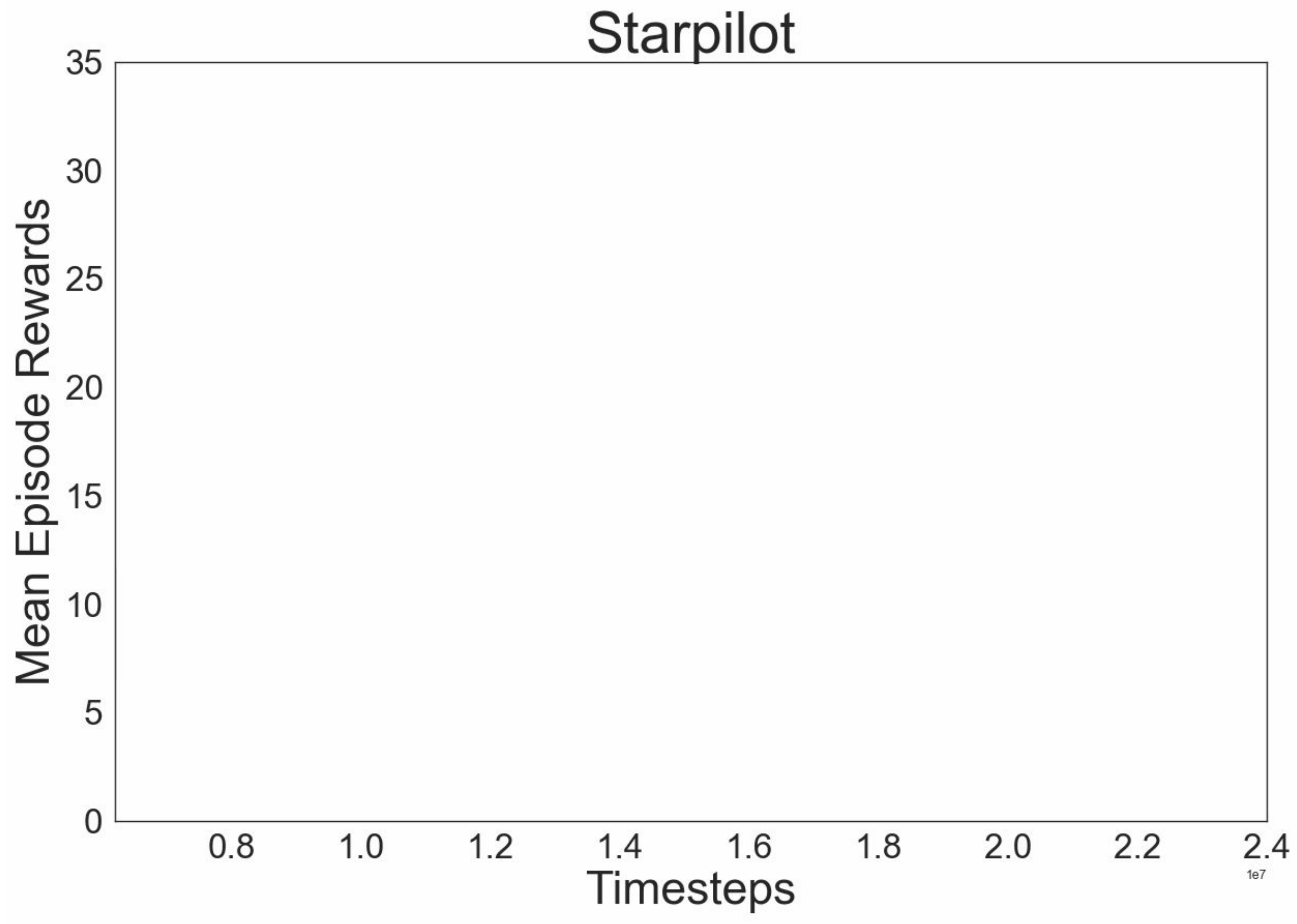




Procgen

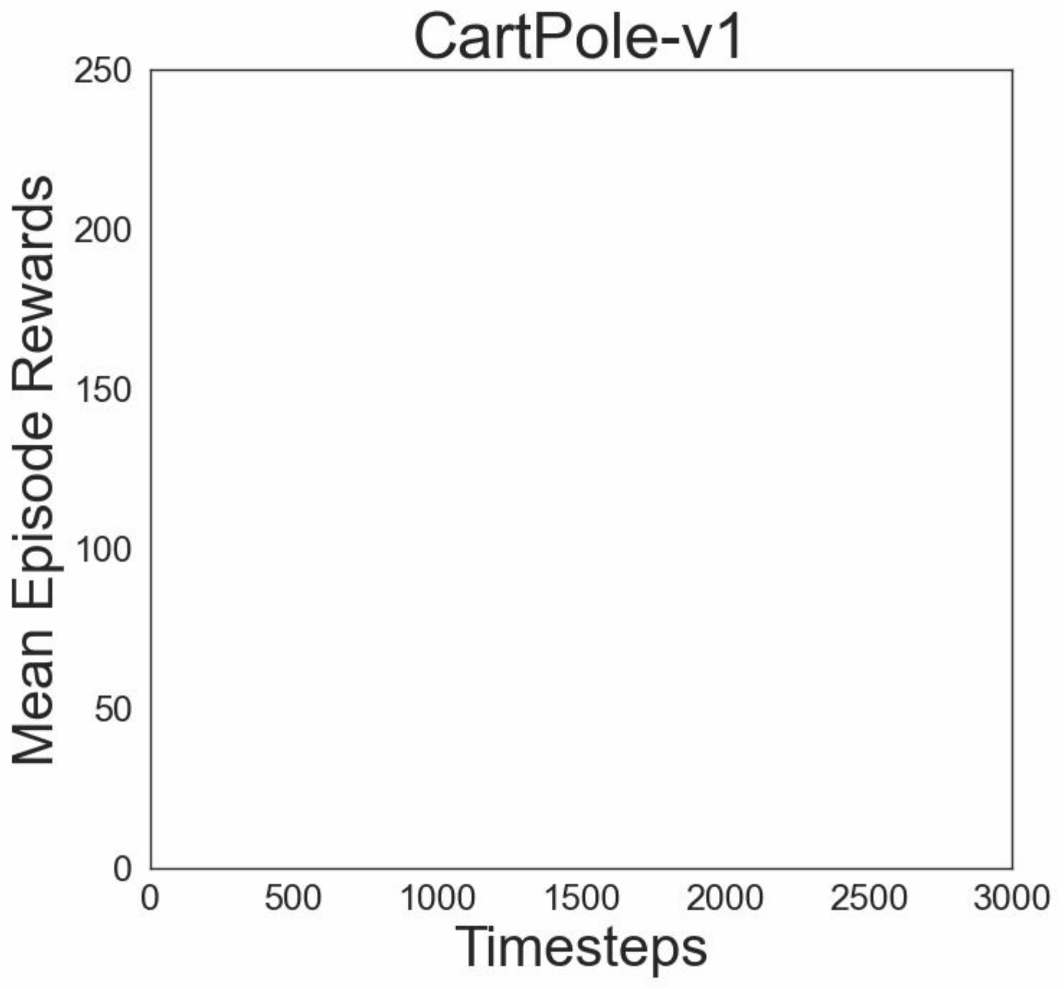


Lifelong RL Suffers from *Loss of Plasticity*

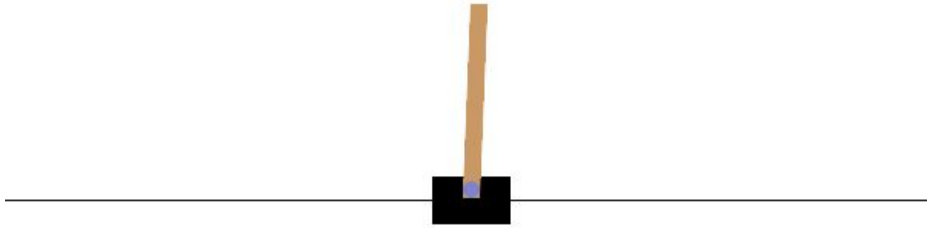


- At every new distribution shift (level), our ability to learn is less (less reward obtainable)
- Eventually, we are **not** able to adapt at all
- AKA: *negative transfer, primacy bias, capacity loss*

Lifelong RL Suffers from *Loss of Plasticity*



- Policy collapse is possible.



Why does *Loss of Plasticity* occur?

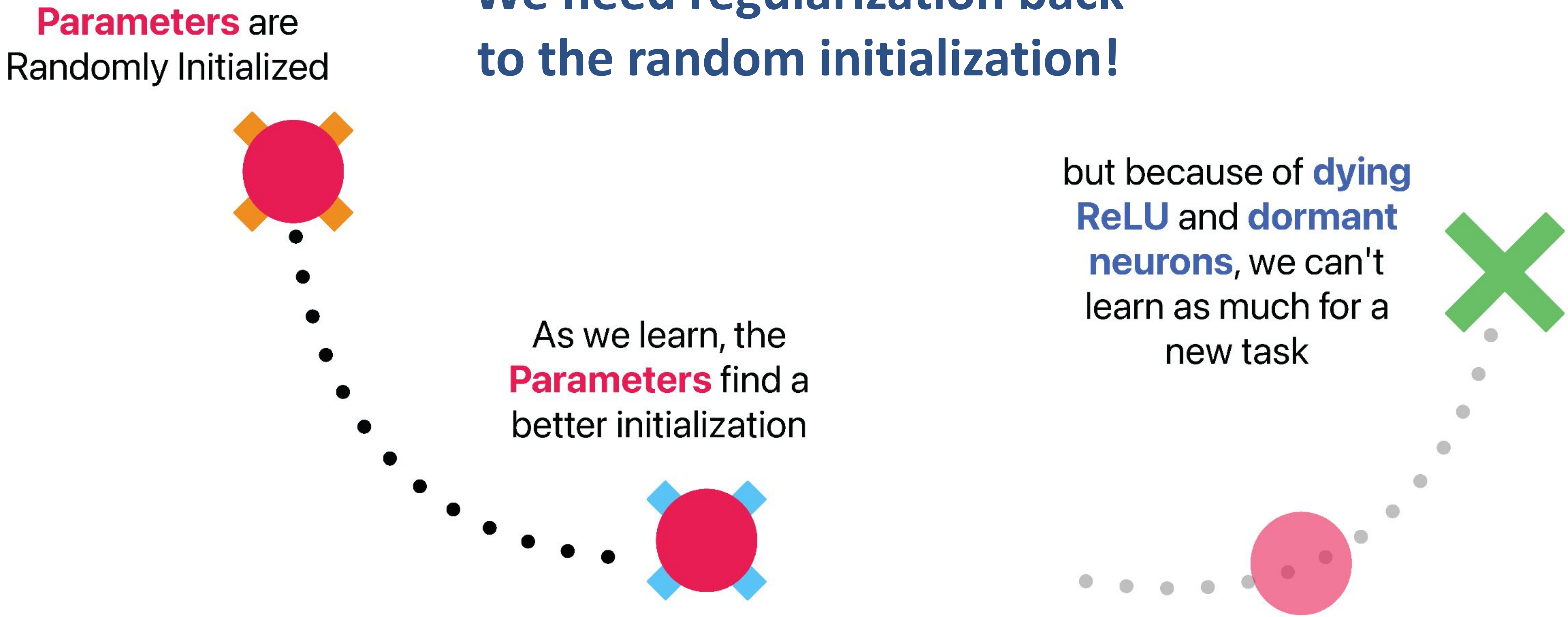
Parameter norm growth: Large weight magnitudes can cause optimization issues.

Saturated activations: Dead or inactive units lead to less expressive networks.

Ill-conditioned loss landscapes: regions where the gradients either explode (large gradients) or vanish (small gradients), making it difficult for the optimizer to find a good path to minimize the loss.

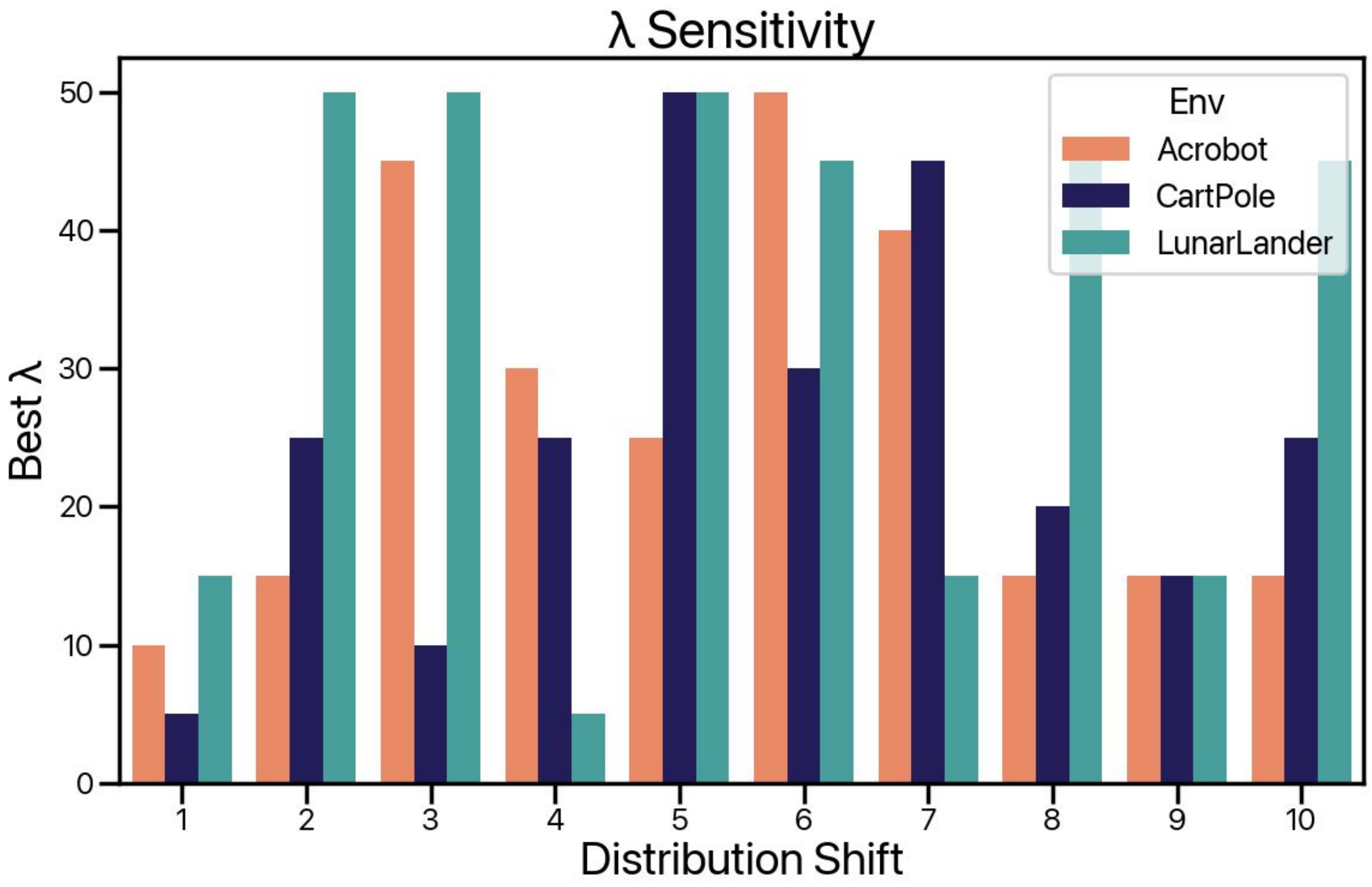
Lyle et al., 2023

We need regularization back to the random initialization!



Kumar et al., 2024

L2 is too sensitive and violates the lifelong setting



- L2 regularization towards the initial random parameters helps, **but** requires a regularization strength
- The regularization strength is sensitive to different tasks and environments
- *So how do we set it before we run the agent?*

Some other solutions

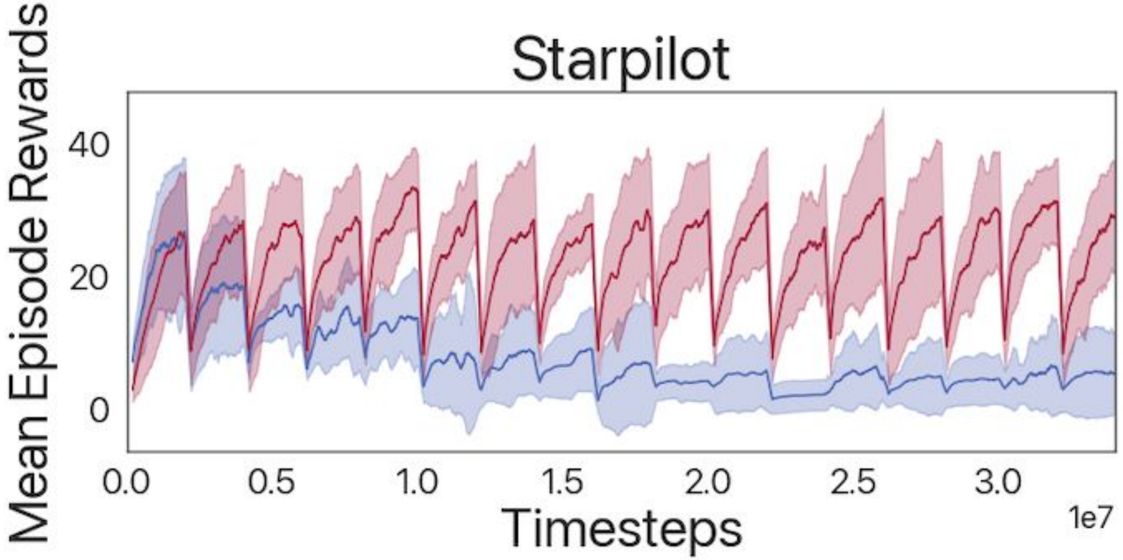
- Reset the network
- Reset some layers in the network
- Reset problematic neurons in the network
- Reset all the parameters, but not all the way
- Regularize the network parameters or features to avoid divergence
- *But again, how do we know when to reset before we run the agent?*

Harvard John A. Paulson
School of Engineering
and Applied Sciences

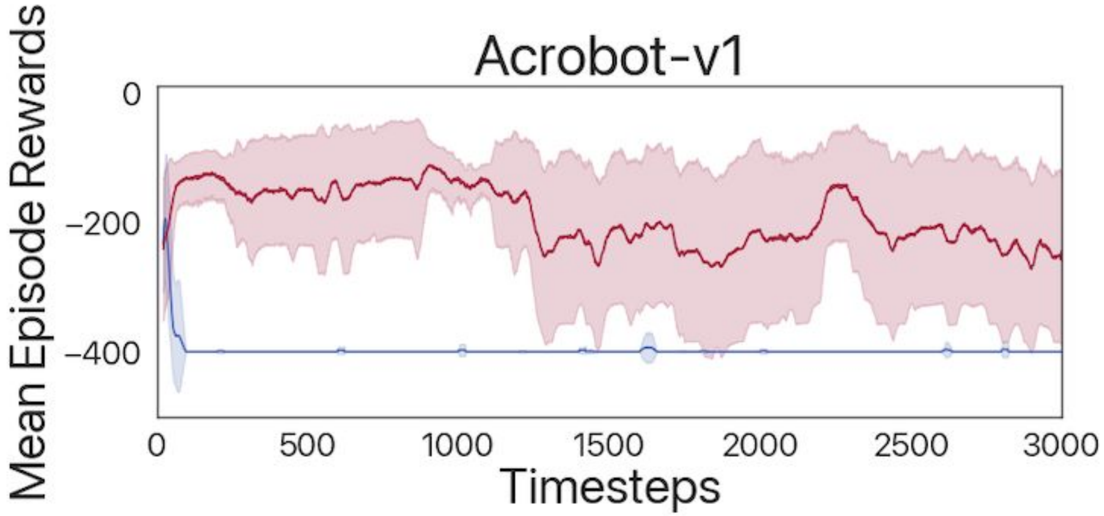
Sokar et al., 2023; Nikishin et al., 2022; Ash and Adams 2020.

What I will demonstrate

Mitigate Loss of plasticity



Accelerate forward transfer



How to implement in your DL/RL experiments with only one line change!

and surprisingly…

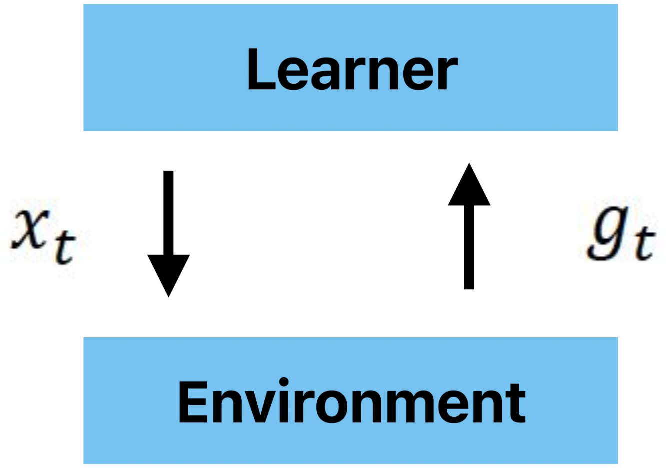
In this **non-convex**, **non-stationary** optimization problem, we can look to online **CONVEX** optimization for help.

OCO Background

Online Convex Optimization is a two-player repeated game.

In each round:

- we pick a decision x_t in a closed convex set X, and reveal it to the environment
- the environment picks a convex loss function $l_t : \mathcal{X} \to \mathbb{R}$
- we suffer the loss $l_t(x_t)$,, and observe a subgradient $g_t \in \partial l_t(x_t)$
- the environment determines if the game should stop – let T be the total number of rounds.



The goal is to minimize its total loss over all rounds, despite not knowing the environment’s loss function in advance

Notes directly from “A modern introduction to online learning
by Francesco Orabona

OCO Background

Definition. With an alternative fixed decision $u \in \mathcal{X}$ called a comparator,

$$\mathrm{Regret}_T(Env, u) := \sum_{t=1}^{T} l_t(x_t) - \sum_{t=1}^{T} l_t(u).$$

Goal. Without knowing the time horizon T, the environment Env and the comparator u beforehand, our goal is to guarantee an upper bound of $\mathrm{Regret}_T(Env, u)$, sublinear in T.

Online Gradient Descent

OGD uses the projected gradient step $x_{t+1} = \Pi_{\mathcal{X}}(x_t - \eta g_t)$.

However, OCO algorithms also require a scaling factor, which gives us the following regret bound

$$\mathrm{Regret}_T(\mathrm{Env}, u) \leq O\left(\frac{\|u - \mathbf{x}_1\|^2}{\sigma\eta} + \sigma\eta\sum_{t=1}^{T}\|\mathbf{g}_t\|^2\right)$$

Online Gradient Descent

$$\text{Regret}_T(\text{Env}, u) \leq O\left(\frac{\|u - \mathbf{x}_1\|^2}{\sigma\eta} + \sigma\eta\sum_{t=1}^{T}\|\mathbf{g}_t\|^2\right)$$

The optimal scaling value is:

$$\sigma = \frac{\|u - \mathbf{x}_1\|}{\eta\sqrt{\sum_{t=1}^{T}\|\mathbf{g}_t\|^2}}$$

Which would give us:

$$\text{Regret}_T(\text{Env}, u) \leq O\left(\|u - \mathbf{x}_1\|\sqrt{\sum_{t=1}^{T}\|\mathbf{g}_t\|^2}\right)$$

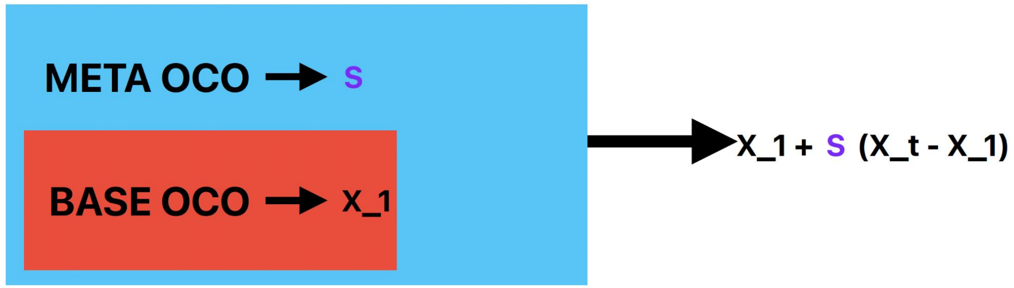
Meta OCO algorithms

Imagine a meta OCO algorithm tuner (to calculate the scaling factor), and a base OCO algorithm.

how can we calculate an unknown scaling factor on the fly?

We have a “tuner” algorithm take $\langle \mathbf{g}_t, \mathbf{x}_t^{\text{base}} - \mathbf{x}_1 \rangle$ as input, and then calculate new scaling value (based on a history of gradients):

$$\mathbf{x}_t^{\text{scaled}} = \mathbf{x}_1 + \sigma_t \cdot (\mathbf{x}_t^{\text{base}} - \mathbf{x}_1)$$



Meta OCO algorithms

Meta OCO reduces through many reductions to Coin betting framework, which relies on calculating a wealth function

$$\sum_{t=1}^{T} c_t x_t \geq \phi\left(\sum_{t=1}^{T} c_t\right)$$

1. we place a bet $x_t \in \mathbb{R}$;
2. *Env* picks a coin c_t (possibly depending on our bet, and historical bets);
3. we observe c_t and win $c_t x_t$ amount of money.

Our goal is to guarantee a wealth function ϕ, evaluated at a quantity that characterizes the complexity of the coin / market instance.

Meta OCO algorithms

Meta OCO reduces through many reductions to Coin betting framework, which relies on calculating a wealth (potential) function – this solved by solving the **backwards heat equation**

1. we place a bet $x_t \in \mathbb{R}$;
2. *Env* picks a coin c_t (possibly depending on our bet, and historical bets);
3. we observe c_t and win $c_t x_t$ amount of money.

Our goal is to guarantee a wealth function ϕ, evaluated at a quantity that characterizes the complexity of the coin / market instance.

Meta OCO algorithms

Two tuners based on two potential functions arise through this type of framework:

- AdaNormalHedge [Luo and Schapire 2015] – suboptimal regret
- **Erfi potential function** [Harvey et al., 2020; Zhang et al., 2024] – optimal regret

$$\text{Regret}_T^{\text{tuner}}(\sigma) \leq \tilde{O}\left(|\sigma|\sqrt{\sum_{t=1}^{T}\langle \mathbf{g}_t, \mathbf{x}_t^{\text{base}} - \mathbf{x}_1\rangle^2}\right)$$

Calculate scaling values even without: $\|u - \mathbf{x}_1\|$

Harvard John A. Paulson School of Engineering and Applied Sciences

Connecting OCO and Lifelong RL

- **Policy definition:** A policy refers to the distribution of an agent's actions, parameterized by a weight vector $\theta_t \in \mathbb{R}^d$, updated over time based on historical observations.
- **Loss function:** After selecting an action and receiving feedback from the environment, the agent defines a loss function $J_t(\theta)$, which characterizes the hypothetical performance of each parameter.
- **Policy gradient:** The agent computes the policy gradient $g_t = \nabla J_t(\theta_t)$ which represents the direction to update the current policy to improve performance.
- **Optimization update:** Using a first-order optimization algorithm OPT, the agent updates the weights as $\theta_{t+1} = \mathrm{OPT}(\theta_t, g_t)$

We introduce a meta-optimizer called TRAC

Algorithm 1 TRAC: Parameter-free Adaption for Continual Environments.

Input: A policy gradient oracle $\mathcal{G}$; a first order optimization algorithm BASE; a reference point $\theta_{\text{ref}} \in \mathbb{R}^d$; n discount factors $\beta_1, \ldots, \beta_n \in (0, 1]$ (default: $0.9, 0.99, \ldots, 0.999999$).

Initialize: Create n copies of Algorithm 2, denoted as $\mathcal{A}_1, \ldots, \mathcal{A}_n$. For each $j \in [1 : n]$, $\mathcal{A}_j$ uses the discount factor β_j. Initialize the algorithm BASE at θ_{ref}. Let $\theta_1 = \theta_{\text{ref}}$.

for $t = 1, 2, \ldots$ **do**

  Obtain the t-th policy gradient $g_t = \mathcal{G}(t, \theta_t) \in \mathbb{R}^d$.

  Send g_t to BASE as its t-th input, and get its output $\theta_{t+1}^{\text{Base}} \in \mathbb{R}^d$.

  For all $j \in [1 : n]$, send $\langle g_t, \theta_t - \theta_{\text{ref}} \rangle$ to $\mathcal{A}_j$ as its t-th input, and get its output $s_{t+1,j} \in \mathbb{R}$.

  Define the scaling parameter $S_{t+1} = \sum_{j=1}^{n} s_{t+1,j}$.

  Update the weight of the policy,

$$\theta_{t+1} = \theta_{\text{ref}} + \left(\theta_{t+1}^{\text{Base}} - \theta_{\text{ref}}\right) S_{t+1}.$$

end for

We introduce a meta-optimizer called TRAC

Algorithm 2 1D Discounted Tuner of TRAC.

Input: Discount factor $\beta \in (0, 1]$; small value $\varepsilon > 0$ (default: 10^{-8}).
Initialize: The running variance $v_0 = 0$; the running (negative) sum $\sigma_0 = 0$.
for $t = 1, 2, \ldots$ **do**
    Obtain the t-th input h_t.
    Let $v_t = \beta^2 v_{t-1} + h_t^2$, and $\sigma_t = \beta\sigma_{t-1} - h_t$.
    Select the t-th output

$$s_{t+1} = \frac{\varepsilon}{\text{erfi}(1/\sqrt{2})} \text{erfi}\left(\frac{\sigma_t}{\sqrt{2v_t} + \varepsilon}\right),$$

    where erfi is the *imaginary error function* queried from standard software packages.
end for

TRAC: Algorithm

- TRAC operates on top of a Base Optimizer (i.e Adam/SGD)
- It selects a scaling factor ***S*** to scale the update of the Base optimizer
- TRAC uses the erfi function in a data-dependent way to select ***S***
- With ***S*** , we make an update to the parameters the regularizes towards **theta ref**, in our case this is the random parameter initialization
- **TRAC is insensitive to the step size**

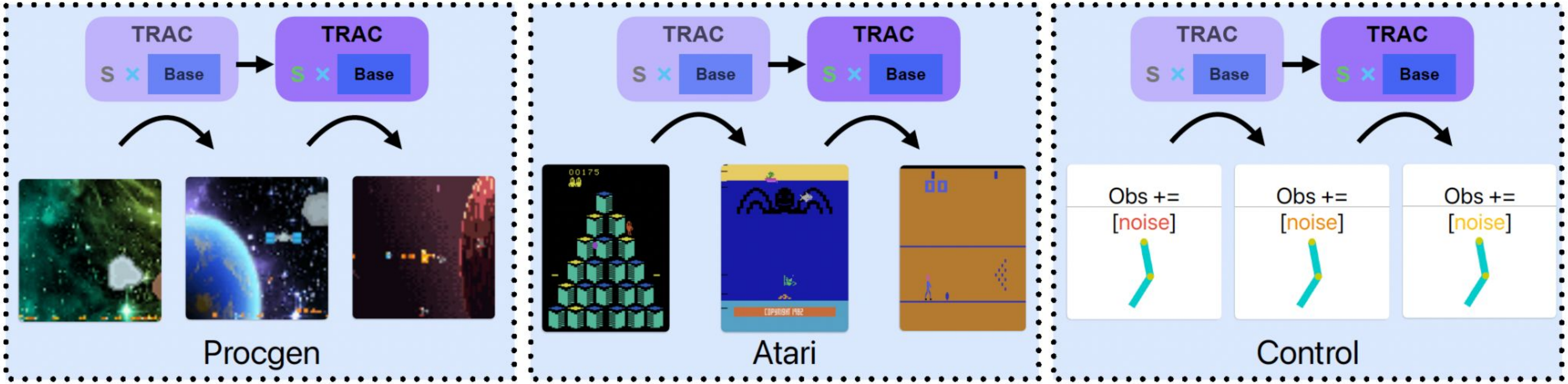
$$\theta_{t+1} = \theta_{\text{ref}} + \left(\theta_{t+1}^{\text{Base}} - \theta_{\text{ref}}\right) S_{t+1}.$$

Harvard John A. Paulson
School of Engineering
and Applied Sciences

Experiments

High-dimensional, vision-based

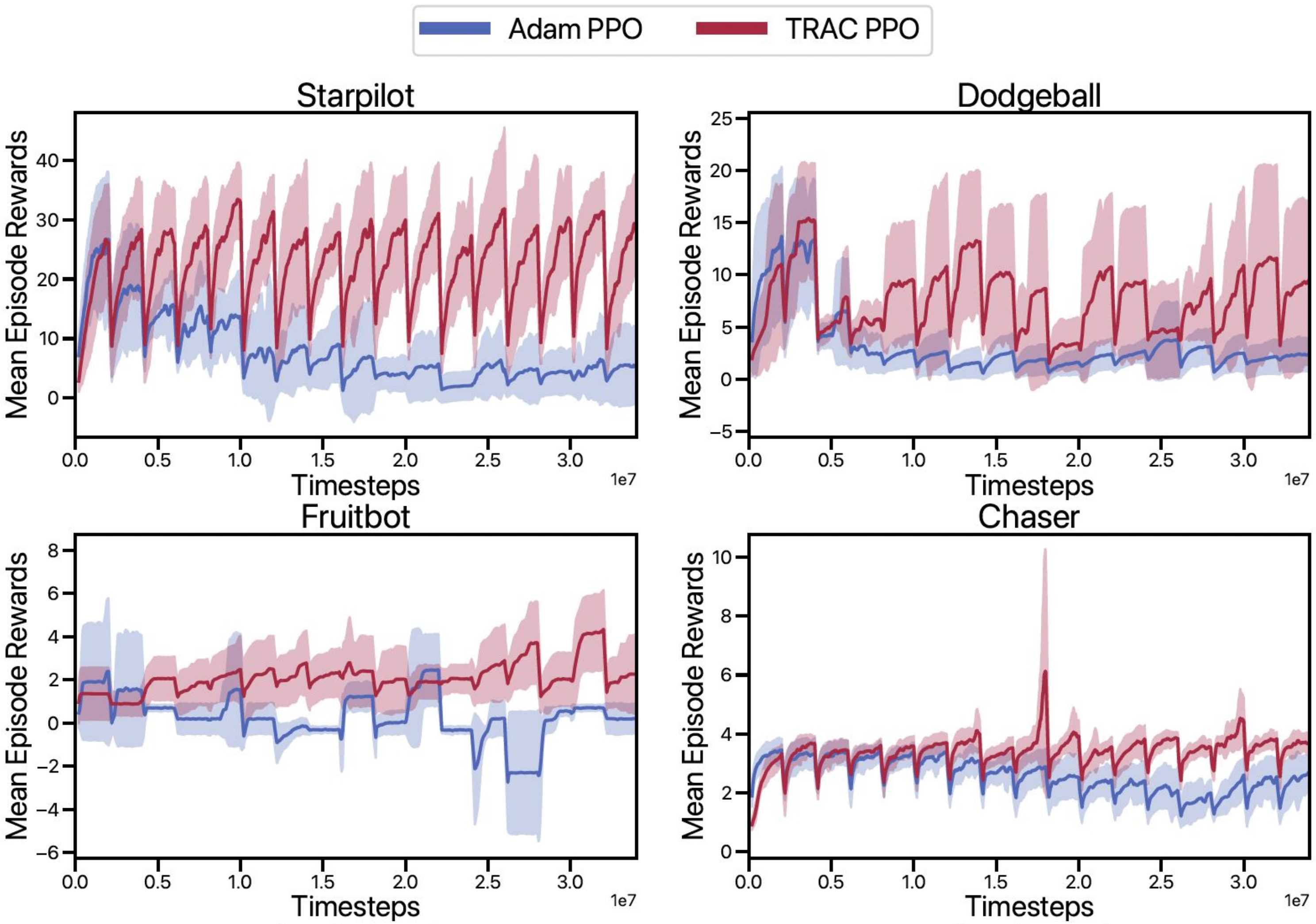
Low-dimensional, control



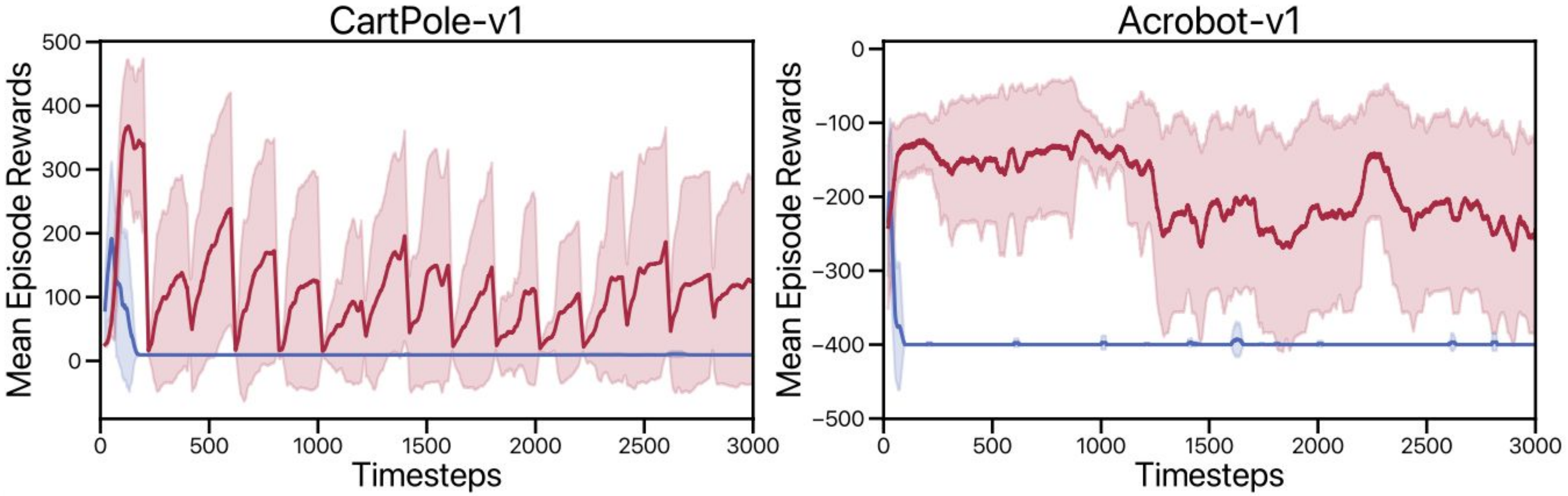
Here we change the level/game

Here we perturb the observation states

We avoid plasticity loss

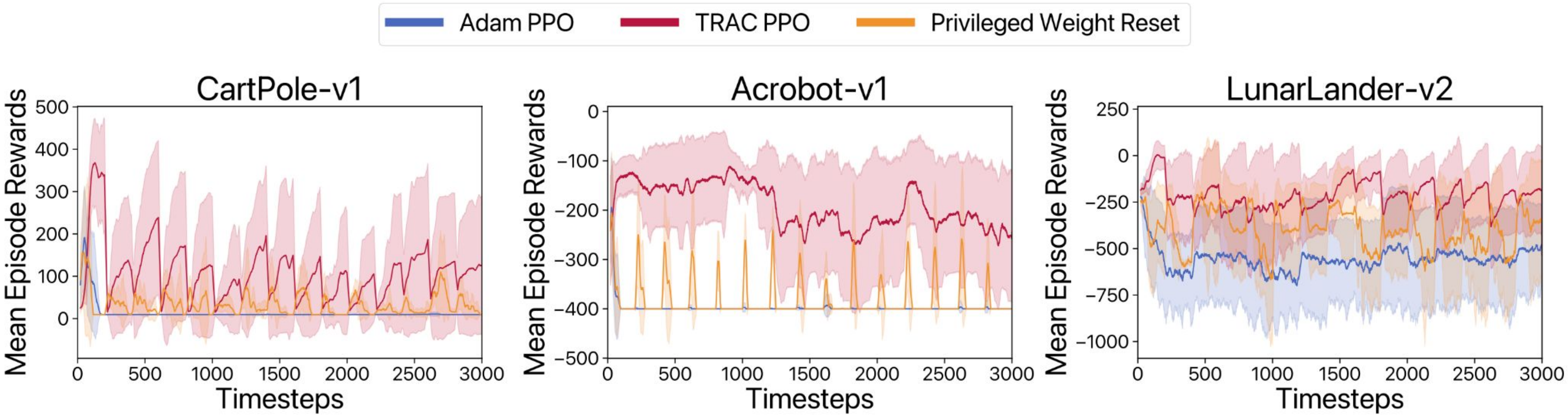


We avoid policy collapse



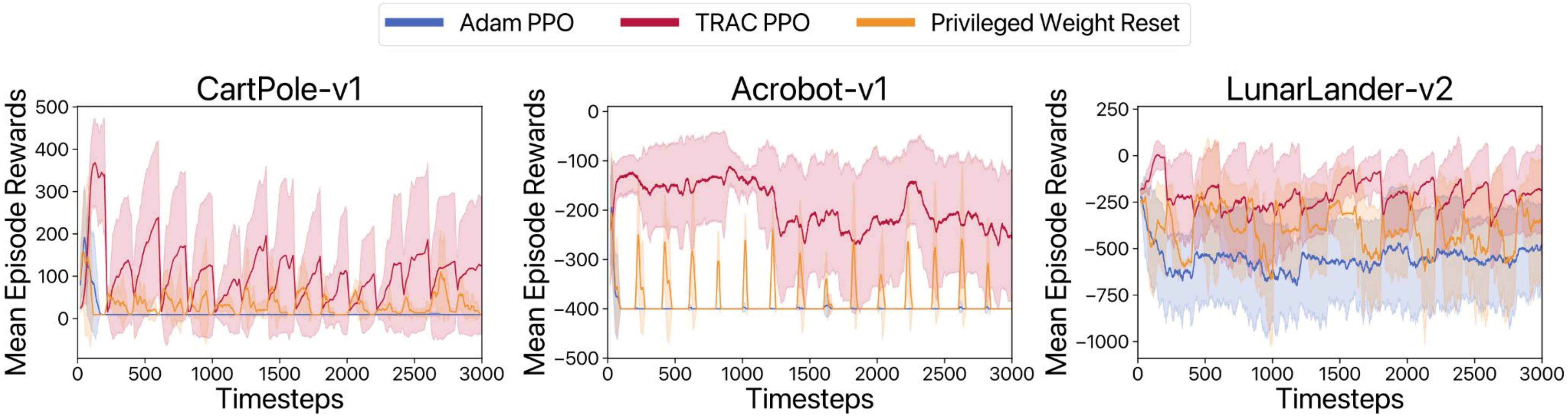
Harvard John A. Paulson
School of Engineering
and Applied Sciences

We also encourage positive transfer (rapid adaptation)

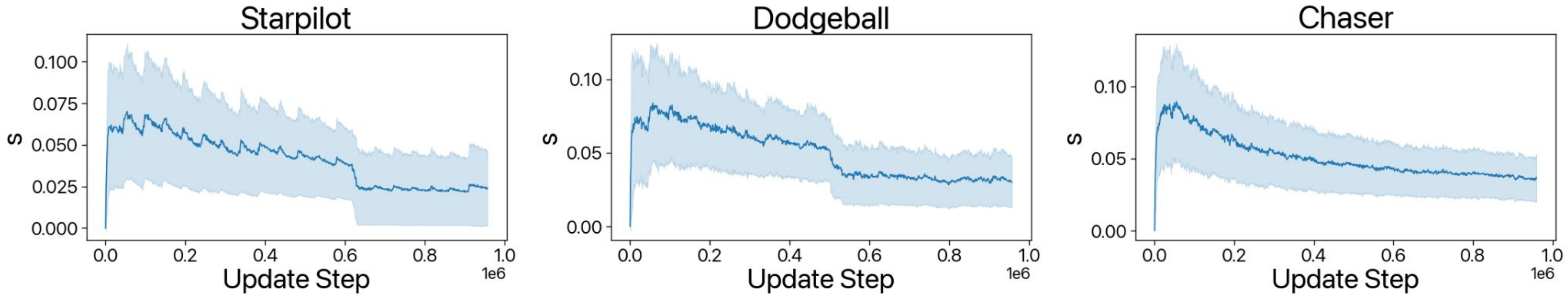


Harvard John A. Paulson
School of Engineering
and Applied Sciences

We also encourage positive transfer (rapid adaptation)

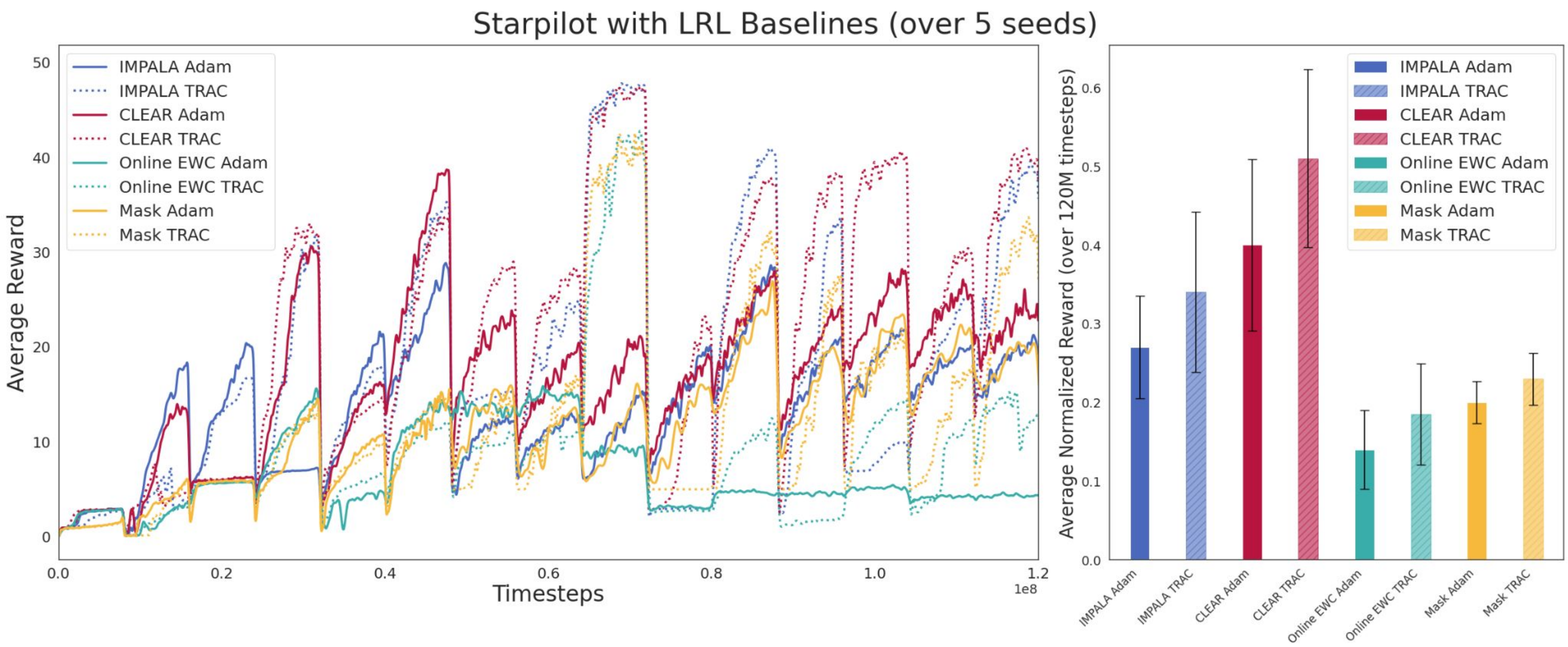


Scaling values proposed by TRAC



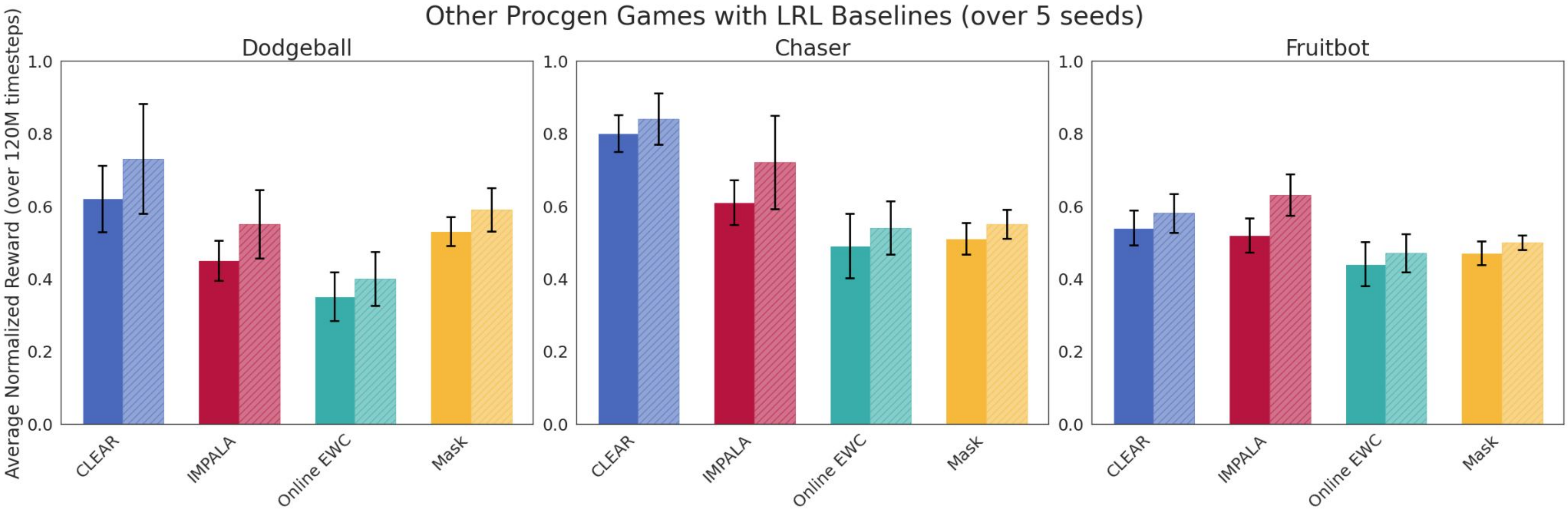
Harvard John A. Paulson
School of Engineering
and Applied Sciences

PPO is not alone in plasticity loss; TRAC works in other LRL algorithms



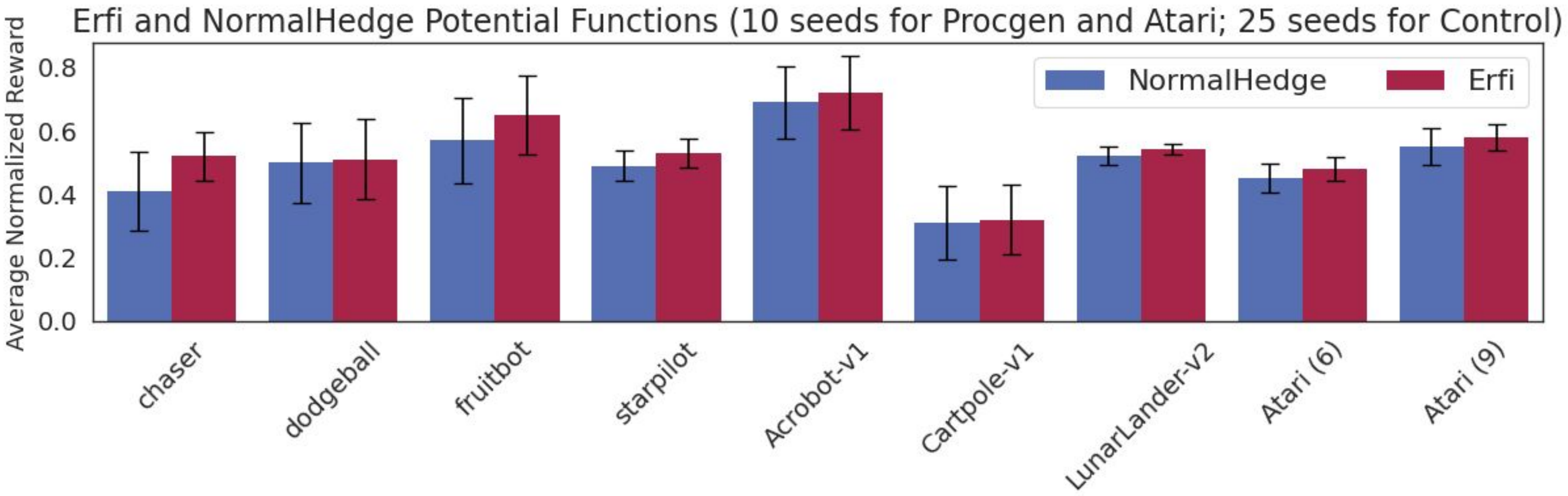
Harvard John A. Paulson
School of Engineering
and Applied Sciences

PPO is not alone in plasticity loss; TRAC works in other LRL algorithms



Harvard John A. Paulson
School of Engineering
and Applied Sciences

Other Meta OCO Tuners also work!!



Stronger Analysis Questions

- Analyze saturated activations with TRAC vs Adam
- Look at relationship between **S** and Parameter norm (looks like inversely correlated)

TRAC is easy to implement

Can be implemented in your RL or lifelong experiments, with only one line change!

```
from trac_optimizer import start_trac
# with TRAC
optimizer = start_trac(log_file='logs/trac.text', Adam)(model.parameters(), lr=0.01)
# using your optimizer methods exactly as you did before (feel free to use others as well)
optimizer.zero_grad()
optimizer.step()
```

Fast **TRAC** 🏎️

Scan 👉

pip install — PyPI

Code — GitHub

Paper — arXiv

Examples — colab

Harvard John A. Paulson School of Engineering and Applied Sciences

COMPUTATIONAL ROBOTICS