Let it Cook: Learning to Wait in Sequential Decision Making

Christopher Watson, Arjun Krishna, Dinesh Jayaraman, Rajeev Alur

Keywords: Hierarchical RL, Action Repeat, Multi-objective RL, Waiting

Summary

In sequential decision making, an agent typically observes its environment and acts at every timestep. However, such active participation may not always be necessary; tasks such as brewing coffee include periods that are served equally well by letting the environment evolve without constant monitoring and control. During such periods, the agent could simply wait to conserve its resources, or redirect its attention to another task. We capitalize on these opportunities by training a "waiting policy" that decides *where and how long to wait*. This involves forgoing sensing to commit to a *wait* action, representing a deliberate pause for a set number of timesteps. We formalize "learning to wait" as minimizing the frequency of sensing and decision making without sacrificing task performance (e.g., the total amount of time to complete a task). To train a waiting policy, we propose an approach that employs reinforcement learning with lexicographically ordered objectives. In experiments across 4 discrete-state household tasks and 3 continuous-state environments, we show that our approach successfully learns waiting behaviors, and can adapt pre-trained policies to wait where appropriate. While different tasks permit different amounts of waiting without sacrificing task performance, our approach consistently finds solutions with significant waiting, sometimes waiting for over 50% of the task duration.

Contribution(s)

1. We formalize learning to *wait*—committing to take a distinguished *wait* action that represents not actively pursuing task progress for a chosen duration while forgoing sensing—as a multi-objective optimization problem with lexicographically-ordered objectives in which the agent must wait as much as possible without sacrificing expected task performance.
   Context: Prior work has studied the related problem of learning to commit to repeat arbitrary actions, but use scalar objectives that implicitly define a tradeoff between task performance and action repetition.
2. We propose to use a variant of Q-Learning with lexicographically-ordered objectives (*LQ-Learning*) to learn *waiting policies* that sense and make a decision at as few timesteps as possible without sacrificing expected cumulative task reward. Additionally, this can be effectively used to train a *wrapper waiting policy* that adapts a pre-trained policy to wait where appropriate.
   Context: We extend an existing algorithm for lexicographic RL to handle extended-duration macro-actions.
3. We show empirically that our approach can train "waiting policies" that (1) achieve task performance on par with that of non-waiting policies trained with vanilla RL and (2) perform extended-duration waiting where appropriate.
   Context: Our experiments are performed in simulated environments.

Let it Cook: Learning to Wait in Sequential Decision Making

Christopher Watson, Arjun Krishna, Dinesh Jayaraman, Rajeev Alur
{ccwatson,arjk,dineshj,alur}@seas.upenn.edu

Department of Computer and Information Science, University of Pennsylvania, USA

Abstract

In sequential decision making, an agent typically observes its environment and acts at every timestep. However, such active participation may not always be necessary; tasks such as brewing coffee include periods that are served equally well by letting the environment evolve without constant monitoring and control. During such periods, the agent could simply wait to conserve its resources, or redirect its attention to another task. We capitalize on these opportunities by training a "waiting policy" that decides *where and how long to wait*. This involves forgoing sensing to commit to a *wait* action, representing a deliberate pause for a set number of timesteps. We formalize "learning to wait" as minimizing the frequency of sensing and decision making without sacrificing task performance (e.g., the total amount of time to complete a task). To train a waiting policy, we propose an approach that employs reinforcement learning with lexicographically ordered objectives. In experiments across 4 discrete-state household tasks and 3 continuous-state environments, we show that our approach successfully learns waiting behaviors, and can adapt pre-trained policies to wait where appropriate. While different tasks permit different amounts of waiting without sacrificing task performance, our approach consistently finds solutions with significant waiting, sometimes waiting for over 50% of the task duration.

1 Introduction

The prevailing paradigm in sequential decision making treats agents as active, closed-loop systems that continually monitor and react to their environment. This approach assumes that making meaningful progress towards a goal requires constant engagement from the agent—resulting in active utilization of its sensory, computational, and motor resources. However, many real-world tasks inherently possess passive dynamics that can be exploited to achieve goals more efficiently (Todorov, 2009). Consider the mundane but complex choreography of a busy kitchen: a kettle takes time to boil, a soup takes time to cook, and a coffee machine operates independently once set to brew. In each of these scenarios, the most effective action for a human, or a robot, is to simply wait. An agent that identifies where and how long to wait can switch to a timed "standby mode" to conserve cognitive and motor resources, or alternatively redirect these resources to an auxiliary task. Given the benefits of strategically waiting, the key questions that arise are: How would one (1) formalize the objective of waiting? and (2) develop learning approaches that train an agent to optimize this waiting objective?

For each environment, we define a distinguished *wait* action, e.g., standing still while coffee brews, or holding the paddle still in the video game of *Pong*. Then "waiting" is simply the act of committing to perform this *wait* action for multiple consecutive timesteps. We aim to discover and exploit opportunities where the agent can wait for an extended period. While our approach is technically agnostic to the effect the *wait* action has on the environment, we limit our experiments to settings where "waiting" corresponds to letting the environment evolve according to an intuitive notion of passive dynamics, as in the aforementioned examples.

Learning to wait balances two potentially competing objectives: maximizing task performance and minimizing the number of times the agent has to sense and respond to the environment (which corresponds to maximizing the amount of time the agent spends waiting). We consider agents that strictly prioritize task performance (measured by cumulative reward from a Markovian environment) and only seek to maximize waiting when doing so does not degrade task performance. Formally, this preference transforms the problem into a multi-objective optimization problem with a lexicographic ordering: the agent seeks to find $\max[J^0, J^1]$, where the *cumulative task reward* J^0 is represents expected task performance and *cumulative waiting reward* $J^1 = -D$ decreases with the number of times D the agent is called upon to sense and make a decision (to either wait or act). For us, waiting periods involve no sensing, no computation, and no task-oriented behavior. Minimizing D thus combines two desiderata: (1) reducing sensing and decision computation, and (2) reducing the need for the agent's active interventions. In a multi-tasking setting, waiting may correspond to periods during which the agent could redirect its resources towards another task. More generally, waiting periods permit switching to a timed "standby mode" that conserves sensory, computational, and motor resources.

To exploit our objective's structure, we apply a *lexicographic* Multi-Objective Reinforcement Learning (MORL) algorithm across several discrete- and continuous-state environments. The amount of waiting possible depends on the environment; in some environments, our method yields policies that wait (in some environments, for over 50% of the task duration) without sacrificing task performance compared to a vanilla RL, non-waiting baseline. We further compare our method against a standard MORL technique that uses scalarized rewards to explore the Pareto frontier between task performance and increased waiting, mediated by a reward-weighting scalarization coefficent. Under a strict preference for task performance, our lexicographic MORL approach achieves near-identical task and waiting performance to the scalarized approach, without needing a hyperparameter sweep to discover the best reward-weighting coefficient for each environment. We also show that our approach can be used to *wrap* a pretrained policy to add waiting behavior without sacrificing task performance. Finally, we demonstrate how learned waiting behavior may enable policy interleaving in a multi-task scenario.

2 Waiting in Sequential Decision Making

We define a *waiting Markov Decision Process (WMDP)* to be a tuple $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathit{wait}, \mathcal{W}, \mathcal{P}, R, \mu)$ where $\mathcal{S}$ is the (discrete or continuous) set of states, $\mathcal{A}$ is the (discrete) set of primitive actions, $\mathit{wait} \in \mathcal{A}$ is the distinguished *wait* action, $\mathcal{W} \subseteq \mathbb{N}^+$ is the (finite) set of waiting durations, $\mathcal{P}(s_{t+1} \mid s_t, a_t)$ is the probability of transitioning from s_t to s_{t+1} upon taking primitive action a_t, $R : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ is the reward function, and μ is the initial state distribution. What differentiates a WMDP from a standard Markov decision process is the distinguished wait action and the set $\mathcal{W}$ of waiting durations. Each natural number $w \in \mathcal{W}$ characterizes a *waiting macro-action* that corresponds to waiting for w consecutive timesteps—that is, forgoing sensing and executing the primitive wait action for w consecutive timesteps.[1] We will denote the duration-w waiting macro-action by its natural number w. We do not make any special requirements of the dynamics $\mathcal{P}$ under the *wait* action, however in applications we will choose the *wait* action in a way that corresponds to a natural notion of non-intervention (e.g. standing idly while soup boils on the stove, or not moving the paddle in a game of *Pong*). We aim to learn policies that can commit to "wait" for an extended period, as illustrated in Figure 1.

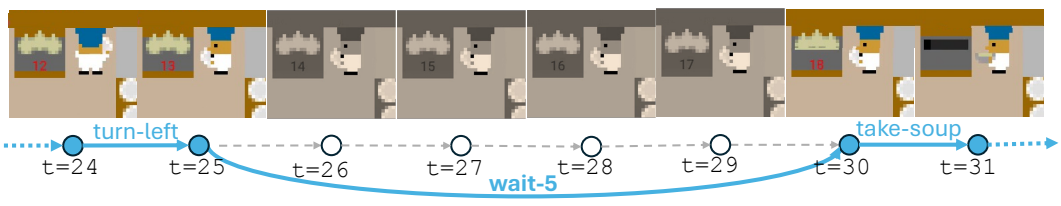

Figure 1: In our `Cook` environment, the agent commits to a duration-5 waiting macro-action near the end of the soup cooking process. No sensing, computation, or active motion occurs while waiting.

[1] A *waiting* macro-action is a special kind of *semi-Markov option* in the terminology of Sutton et al. (1999).

The $wait$ action and set of waiting durations $\mathcal{W}$ let us define the notion of a *waiting policy* $\pi : \mathcal{S} \to \Delta(\mathcal{A} \cup \mathcal{W})$ that interacts with the WMDP to receive both *task rewards* and *waiting rewards*.[2] When queried at a state s, the waiting policy yields either a primitive action $a \in \mathcal{A}$, which has its usual effect according to the dynamics $\mathcal{P}$, or a waiting macro-action $w \in \mathcal{W}$. A WMDP $\mathcal{M} = (\mathcal{S}, \mathcal{A}, wait, \mathcal{W}, \mathcal{P}, R, \mu)$, a waiting policy $\pi : \mathcal{S} \to \Delta(\mathcal{A} \cup \mathcal{W})$, and a finite time horizon $H \in \mathbb{N}$ induce a distribution over length-H trajectories $s_0 a_0 r_1^0 r_1^1 s_1 \ldots a_{H-1} r_H^0 r_H^1 s_H$ generated by drawing an initial state $s_0 \sim \mu$ and querying the policy π at timestep 0 to obtain either a waiting macro-action or a primitive action. At each timestep t, if the policy is queried and yields a waiting macro-action w then the next w actions $a_t, a_{t+1}, \ldots, a_{t+w-1}$ are each the waiting action $wait$ and the policy is not queried again until timestep $t+w$. If instead the the queried policy yields a primitive action $a \in \mathcal{A}$ then $a_t = a$ and the policy is queried at the next timestep $t+1$. For each timestep $1 \leq t \leq H$, the state s_t is sampled as $s_t \sim \mathcal{P}(\cdot \mid s_{t-1}, a_{t-1})$ and the *task reward* r_t^0 is defined as $r_t^0 = R(s_{t-1}, a_{t-1}, s_t)$, which matches the usual notion of reward in an MDP. The *waiting reward* r_t^1 is -1 if the policy is queried at timestep t and 0 otherwise. When the maximum episode horizon H is reached, the episode is abruptly terminated, regardless of whether the agent is in the middle of a waiting macro-action.

Given a trajectory $\tau = s_0 a_0 r_1^0 r_1^1 s_1 \ldots a_{H-1} r_H^0 r_H^1 s_H$ we define the (undiscounted) *cumulative task reward* to be $J^0(\tau) = \sum_{1 \leq i \leq H} r_i^0$. We define the (undiscounted) *cumulative waiting reward* to be $J^1(\tau) = \sum_{1 \leq i \leq H} r_i^1$, i.e., -1 times the number of times the policy π is queried during the trajectory.

Lexicographic waiting objective. Intuitively, a good waiting policy is one that obtains good task performance (as measured by cumulative task reward) while also waiting as much as possible. Since we consider trajectories with an *a priori* fixed horizon H in which the only extended-duration actions are the *waiting macro-actions*, querying the waiting policy fewer times corresponds to more waiting. Given a WMDP $\mathcal{M}$ and a finite time horizon H, we define the vector-valued objective of a waiting policy π as $J(\pi) \in \mathbb{R}^2$ as:

$$J(\pi) = \mathbb{E}_{\tau \sim \mathcal{M}, \pi}[J^0(\tau), J^1(\tau)] \tag{1}$$

We adopt the standard lexicographic ordering $\leq$ over objective values defined such that $[J^0, J^1] \leq [J^{0\prime}, J^{1\prime}] \Leftrightarrow J^0 < J^{0\prime}$ OR $(J^0 = J^{0\prime}$ AND $J^1 \leq J^{1\prime})$. In other words, a policy is "better" than another policy if and only if it either (1) achieves higher cumulative task reward J^0, or (2) it achieves equal J^0 and higher cumulative waiting reward J^1. In the following section we will describe our RL-based approach that seeks to learn a policy π^* that achieves maximal $J(\pi^*)$.

Our choice of waiting objective—to minimize expected number of times the waiting policy is queried—is a natural fit for settings where active decision making incurs a computational or sensory cost. This idea was explored by Zhou et al. (2024), which uses scalar rewards that include an *interaction cost* to encourage action repetition, and Harb et al. (2018), which uses the analogous *deliberation cost* to reduce the number of times a high-level policy is queried in hierarchical reinforcement learning. In the aforementioned works, the magnitude of the (interaction or deliberation) cost implicitly defines a tradeoff between task performance and policy query frequency. In contrast, our lexicographic objective (1) renders the relative magnitudes of the task rewards and the waiting rewards inconsequential. In other words, if we were to define WMDP trajectories such that each r_t^1 were scaled by an arbitrary positive factor, the objective in (1) would induce the same ordering over optimal policies and our LQ-Learning learning algorithm (Algorithm 1, which we introduce in Section 3.1) would behave similarly. Practically, this alleviates the need to perform a hyperparameter sweep to find an appropriate weighting of task reward vs. waiting reward.

Another important distinguishing characteristic of our formalization is that the only way to avoid querying the policy is by selecting the temporally extended waiting macro-actions. Thus, our objective naturally encourages the agent to pick those actions. Further, since "wait" actions in our settings cede control to the environment's passive dynamics, more waiting translates to conserving not only sensing and computation, but also motor resources. Penalizing all decisions, including the choice to commit to waiting, encourages the agent to take *longer duration* waiting actions when possible. Preferring long contiguous durations of waiting can enable downstream multitasking applications where a long waiting period provides more time for the agent to "fill in" the waiting period by executing an auxiliary policy to make progress towards another task; we explore a simple instantiation of this idea in the context of the `Coffee` task in Section 4.

[2] For a measurable space $\mathcal{X}$, we write $\Delta(\mathcal{X})$ to denote the set of probability measures over $\mathcal{X}$.

3 Learning to Wait

3.1 Lexicographic Q-Learning

Our goal, to maximize the vector-valued objective (1), lends itself naturally to a *lexicographic* MORL approach (Gábor et al., 1998; Skalse et al., 2022) that seeks a policy that maximizes a vector-valued reward signal. We adapt the lexicographic version of Q-Learning described by Skalse et al. (2022) to handle waiting macro-actions. Our resulting LQ-Learning algorithm, which we detail in Algorithm 1, maintains two Q-estimates: the task Q-estimate Q^0 and waiting Q-estimate Q^1. Each Q-estimate considers the "full action set" $\mathcal{A} \cup \mathcal{W}$, which contains both primitive actions and waiting macro-actions. Greedy policy inference selects an action that is σ-close to optimal with respect to Q^0's future cumulative task reward estimate, breaking ties according to Q^1's future cumulative waiting reward estimate, where $\sigma \in \mathbb{R}^+$ is a small tolerance parameter that softens the lexicographic comparison. The tolerance parameter prevents a slightly inaccurate empirical Q^0 estimate from erroneously ruling out the lexicographically optimal action during the Q^1 update in Line 15 of Algorithm 1. We only update the Q estimates in response to the output (an element of $\mathcal{A} \cup \mathcal{W}$) made by the policy when the policy is queried. We do not perform updates based on the primitive *wait* actions applied in the middle of a waiting macro-action, because these do not directly correspond to decisions made by the policy. This is analogous to how Sutton et al. (1999) defines *SMDP Q-Learning*.

Algorithm 1: LQ-Learning

Input: WMDP $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathit{wait}, \mathcal{W}, \mathcal{P}, R, \mu)$, episode horizon H, learning rate α, lexicographic tolerance σ, initial Q tables $Q^0, Q^1 \in \mathbb{R}^{|\mathcal{S}| \times |\mathcal{A} \cup \mathcal{W}|}$.

1 **for each** *episode* **do**
2 $\quad t \leftarrow 0, s \leftarrow \mathcal{M}$.`reset()`
3 $\quad$ **while** $t < H$ **do**
4 $\quad\quad u \leftarrow \begin{cases} \texttt{Lex-Argmax}(s, Q^0, Q^1, \sigma) & \text{w.p. } 1-\epsilon \\ \text{Uniform}(\mathcal{A} \cup \mathcal{W}) & \text{w.p. } \epsilon \end{cases}$ `epsilon-greedy exploration`
5 $\quad\quad$ **if** $u \in \mathcal{W}$ **then** `Waiting macro-action`
6 $\quad\quad\quad G^0 \leftarrow 0$ `Task reward during macro-action`
7 $\quad\quad\quad$ **for** u *times* **do** `Take u primitive wait steps`
8 $\quad\quad\quad\quad s', r^0 \leftarrow \mathcal{M}$.`step`$(\mathit{wait})$
9 $\quad\quad\quad\quad G^0 \leftarrow G^0 + r^0$
10 $\quad\quad$ **else** `Primitive action`
11 $\quad\quad\quad s', r^0 \leftarrow \mathcal{M}$.`step`$(u)$
12 $\quad\quad\quad G^0 \leftarrow r^0$
13 $\quad\quad G^1 \leftarrow -1$ `Waiting reward: -1 per policy query`
$\quad\quad$ `Update Q estimates`
14 $\quad\quad Q^0[s,u] \leftarrow (1-\alpha) Q^0[s,u] + \alpha \left(G^0 + \max_{v \in \mathcal{A} \cup \mathcal{W}} Q^0[s', v] \right)$
15 $\quad\quad Q^1[s,u] \leftarrow (1-\alpha) Q^1[s,u] + \alpha \left(G^1 + Q^1[s', \texttt{Lex-Argmax}(s', Q^0, Q^1, \sigma)] \right)$
16 $\quad\quad s \leftarrow s', t \leftarrow t+1$
17 **Function** `Lex-Argmax`(s, Q^0, Q^1, σ)**:**
18 $\quad U \leftarrow \left\{ u \in \mathcal{A} \cup \mathcal{W} \;\middle|\; Q^0[s,u] \geq \max_{u' \in \mathcal{A} \cup \mathcal{W}} Q^0[s,u'] - \sigma \right\}$
19 $\quad$ **return** $\operatorname{argmax}_{u \in U} Q^1[s,u]$

3.2 Learning A Waiting Wrapper For a Pretrained Policy

Thus far, we have focused on training a policy *from scratch*. In some settings, there may already exist a policy $\pi_{base} : \mathcal{S} \to \Delta(\mathcal{A})$ that achieves satisfactory cumulative (task) reward in a non-waiting MDP environment. Such a policy may be handwritten, or trained via e.g. imitation learning or RL. Direct

deployment of π_{base} in a WMDP[3] would garner expected cumulative task reward $J^0(\pi_{base})$ equal to expected cumulative reward in a non-waiting MDP, but extremely low cumulative waiting reward. We will now describe how to train a waiting policy that serves as a wrapper around π_{base}. This approach enjoys efficient exploration because the wrapper policy has a small set of possible actions: either execute π_{base} to evolve the system under its controlled dynamics, or wait and let it evolve passively.

Given a WMDP $\mathcal{M} = (\mathcal{S}, \mathcal{A}, wait, \mathcal{P}, R, \mu)$ and a base policy $\pi_{base} : \mathcal{S} \to \Delta(\mathcal{A})$ we define the induced two-action WMDP $\mathcal{M}' = (\mathcal{S}, \{a_{base}, wait\}, \mathcal{P}', R, \mu)$ where $a_{base} \notin \mathcal{A}$ is a new primitive action that "simulates" π_{base}. The dynamics are defined such that $\forall s, s' \in \mathcal{S}$

$$\mathcal{P}'(s' \mid s, wait) = \mathcal{P}(s' \mid s, wait) \qquad \mathcal{P}'(s' \mid s, a_{base}) = \sum_{a \in \mathcal{A}} (\mathcal{P}(s' \mid s, a) \pi_{base}(a \mid s))$$

A waiting policy $\pi : \mathcal{S} \to \Delta(\{a_{base}, wait\} \cup \mathcal{W})$ can choose to simulate π_{base} by taking action a_{base} or wait. Learning a waiting wrapper around a pre-trained policy π_{base} may be useful in several settings. When training waiting policies from scratch requires extensive environmental interaction then taking advantage of an existing base policy π_{base} (trained without waiting considerations) may significantly reduce the amount of environmental interaction needed to obtain a policy that achieves satisfactory value of Equation (1). For example, in domains where large pre-trained policies exist, such as in robotic manipulation, (e.g. Octo Model Team et al. (2024); Brohan et al. (2023); Black et al. (2026); NVIDIA et al. (2025)) it may be unnecessary to train a policy from scratch. Using the method described in this section, we may exploit the pre-trained base policy's task performance while also exploring its waiting affordances for improved sensory, computational, and motor resource efficiency during task execution. Of course, it is possible that there exists no wrapper waiting policy π with expected objective value that exceeds $J(s \mapsto \delta_{a_{base}})$. In our experimental results, we observed non-trivial waiting behaviors even when the base policy π_{base} was trained through standard RL (with no additional care to make it robust in states that may be reached via extended-duration waiting).

4 Experiments

We compare the effectiveness of our LQ-Learning approach (Section 3.1) and a scalarized reward baseline (which we will describe in Section 4.2), both in the context of learning a waiting policy from scratch and wrapping a pre-existing policy to wait as much as possible without sacrificing expected cumulative task reward. Finally, we show a case study of how learning to wait can be used as a first step toward interleaving policies to complete multiple tasks efficiently.

4.1 Environments

Each of our environments (with the exception of `CartPole`) is a goal-reaching environment: we consider the task reward to be -1 per timestep, which encourages the agent to reach the goal as fast as possible. We also include *early episode termination* if the agent reaches the goal before the *a priori* episode horizon is reached. While our choice of task reward allows us to visualize cumulative task reward and the number of timesteps in which the policy is queried (i.e., zero minus the cumulative waiting reward) on a single set of axes, we note that our approach also applies to settings where cumulative task reward does not enjoy any special correlation to a number of elapsed timesteps. Our environments are described below, with additional details in Appendix A. In all of our environments the agent observes the complete state of the environment; in the cooking and coffee preparation tasks this includes the amount of timesteps remaining for a soup to finish cooking, the coffee machine to finish heating, or the coffee to finish brewing.

Cook. We study three variations of a cooking task in the `overcooked-ai` (Carroll et al., 2019), which simulates the popular video game *Overcooked* (Ghost Town Games, 2016) in which the agent must cook and serve dishes. In our `Cook` environment the agent must gather and place 3 onions in the pot, which starts an 18 timestep countdown until the soup is cooked and ready. The agent must also pick up a dish, use the dish to pick up the soup, and then deliver the soup to the goal location. In `CookLonger` the

[3] Deploying $\pi_{base} : \mathcal{S} \to \Delta(\mathcal{A})$ in a WMDP with actions $\mathcal{A}$ and wait durations $\mathcal{W}$ assumes the tacit lift to $\pi_{base} : \mathcal{S} \to \Delta(\mathcal{A} \cup \mathcal{W})$.

soup takes 36 timesteps to cook, and in `CookTwice`, the agent must cook and deliver two soups, each of which takes 18 timesteps to cook. The *wait* action makes the agent stand still and we consider $\mathcal{W}=\{5,15\}$.

Coffee. We implement a `Coffee` brewing task in a deterministic 2D MiniGrid gridworld (Chevalier-Boisvert et al., 2023). The agent must go to the coffee machine and *toggle* it to start it heating. After an 8 timestep countdown the agent must toggle the machine again to start coffee brewing, which triggers an 18 timestep countdown, after which the agent must toggle the coffee machine again to collect the coffee. In addition to the multi-phase interaction with the coffee machine, the agent must go to and *toggle* the cream and sugar to pick them up. The episode terminates (success) when the coffee, cream, and sugar are collected (in any order). The *wait* action makes the agent stand still and we consider $\mathcal{W}=\{5,15\}$.

Pong. Our `Pong` environment is a modified version of the Atari game Pong, rewritten with inspiration from the Gymnax (Lange, 2022) implementation. The agent plays against a scripted opponent, and its objective is to win one point as quickly as possible. If the ego agent loses a point, the episode does not terminate but the ball restarts in the middle of the arena. The agent observes a continuous-valued low-dimensional vector observation comprising each paddle's position, the ball's current and last timestep positions, and the ball's current velocity. The *wait* action does not move the paddle and we consider $\mathcal{W}=\{2,4,8,16,32,64,128\}$.

MountainCar. We adopt Gymnasium's (Towers et al., 2024) `MountainCar`, in which a car must climb up from a valley to reach the top of a mountain as fast as possible. This is only possible by gradually building up momentum over multiple back-and-forth passes. Unlike our other environments, even small amounts of waiting hinder task performance. The agent observes the car's position and velocity. The *wait* action applies zero force to the car (which may still roll freely) and we consider $\mathcal{W}=\{2,4,6,8,10,12,14,16,18,20\}$.

Cartpole. We adapt Gymnasium's (Towers et al., 2024) `Cartpole` environment. We add an additional primitive action *wait* to standard CartPole's 2-action set ("left" and "right"). Unlike our other environments, the goal is to balance the pole as long as possible, to the maximum horizon $H=200$; we use a task reward of $+1$ per timestep. The agent observes the cart's position and velocity, and the pole's angle and angular velocity. We consider $\mathcal{W}=\{2,4,6,8,10,12,14,16,18,20\}$.

4.2 Scalar reward baseline

A simpler alternative to LQ-Learning uses a coefficient λ to scalarize each task reward r^0 and waiting reward r^1 into a combined reward $r^0+\lambda r^1$. To train a policy that optimizes Equation (1), one must perform a hyperparameter sweep over values of λ in each new environment. Our LQ-Learning approach does not require such hyperparameter tuning, however lexicographic MORL approaches (such as LQ-Learning) require specific policy architectures and algorithms (e.g. those presented in Skalse et al. (2022)) that are less mature than scalar-reward RL approaches. For a scalarization coefficient $\lambda \in \mathbb{R}^{\geq 0}$ we define the λ-scalarized expected cumulative reward of a policy (for a fixed waiting MDP and horizon) to be $J_\lambda(\pi)=J^0(\pi)+\lambda J^1(\pi)$ In order to train a policy that optimizes J_λ we apply the non-lexicographic version of the SMDP Q-Learning algorithm (Sutton et al., 1999) defined similarly to our LQ-Learning (Section 3.1) except there is only one Q-estimate and the (scalarized) reward at timestep t is $r_t=r_t^0+\lambda r_t^1$. When $\lambda=0$ we recover the original task reward's objective that does not incentivize waiting. When λ is extremely large, there will always be an optimal policy that waits as much as possible, regardless of the task rewards.

4.3 Implementation details

We apply LQ-Learning and Q-Learning with scalar rewards to the discrete `Cook`, `CookTwice`, `CookLonger` and `Coffee` tasks. We apply their deep RL analogs LDQN (details in Appendix B.1) and DQN (Mnih et al., 2015) to the continuous-state environments. We adapt the Q-Learning and DQN implementations found in SKRL (Serrano-Muñoz et al., 2023) to LQ-Learning and LDQN following Skalse et al. (2022). Our DQN-based experiments use Q-networks that are MLPs with ReLU activations and 2 hidden 256 neurons per layer. Further hyperparameter details can be found in Appendix B.

For the scalar reward baseline, we sweep over the λ values 0,0.01,0.1,0.5,1,5,10,15. We report the result of the best λ value with respect to Equation (1), additional results are found in Appendix C.1.

4.4 Results

To what extent can an agent wait? We first study the extent to which an agent can wait without sacrificing task performance across our suite of tasks. We apply our LQ-Learning/LDQN approach (Section 3.1). Our comparison in Figure 3 shows that LQ-Learning and LDQN consistently yield policies that achieve high *cumulative task reward* (close to that obtained by vanilla RL that maximizes cumulative task reward as its sole objective without the option to wait), so a quick summary of the waiting affordances of the task exploited by our approach is possible through Figure 2, which plots the proportion of episode timesteps spent in different duration waiting macro-action by our LQ-Learning and LDQN policies.

Intuitively, `Cook` and `Coffee` (visualized in Figure 3) would present opportunities for the agent to wait (e.g. during the time that soup takes to cook or coffee takes to brew) without sacrificing task performance (here, time to goal). Indeed, in `Cook` the learned policy chooses the duration 5 wait action twice during the 33 timesteps taken to fetch ingredients, cook, and serve the soup. At first it seems counterintuitive that the agent does not take the duration 15 waiting action given that the soup takes 18 timesteps to cook, however our LQ-Learning-learned policy is in fact optimal with respect to Equation (1). Close inspection of the learned policy's behavior reveals that our LQ-Learning approach discovers a policy that fetches the dish during the time the soup is cooking: this permits optimal cumulative task reward, and an optimal value of Equation (1), yet precludes the opportunity to wait for 15 uninterrupted timesteps.[4] Similarly, in `Coffee`, LQ-Learning learns a clever policy that achieves a higher value of Equation (1) than possible by naively waiting: the learned policy fetches sugar as the coffee brews.

Training a waiting wrapper policy around a pre-existing policy (Section 3.2) can expose different opportunities to wait without sacrificing task reward than when learning a waiting policy from scratch. We see this when we train a waiting wrapper policy to wrap a (suboptimal) handwritten policy in our tabular environments: Our naive handwritten `Cook` base policy stands idly by as the soup cooks; our wrapper waiting policy trained via LQ-Learning successfully learns to take the duration-15 wait during this idle period. Similarly, our wrapper waiting policy for `Coffee` performs a duration-5 and a duration-15 wait as the machine heats and brews, respectively. In our continuous-state environments, we wrap a policy trained using vanilla DQN. Surprisingly, even these natural RL-trained afford the ability to exercise waiting behaviors. For example, our learned waiting wrapper policies for `Pong` and `Cartpole` preserve the DQN-trained base policy's perfect cumulative task reward while spending well over half of the timesteps committed to waiting. This suggests that learning to wait, conveniently applied as a wrapper *over* pre-trained policies, could query the base policy more selectively without reducing task performance.

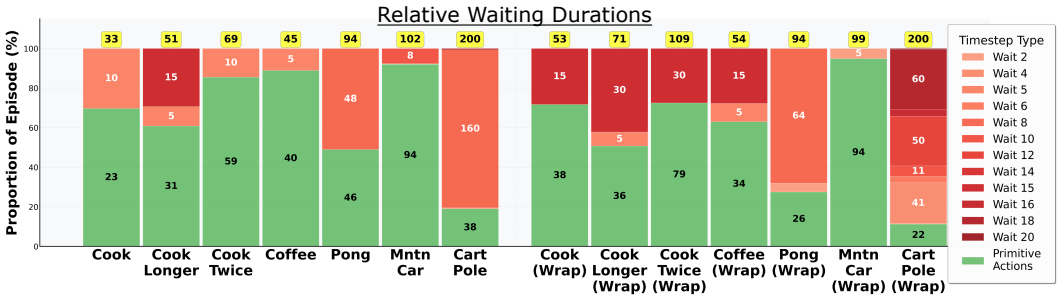

Figure 2: Empirical performance averaged over 1000 trajectories from the best trained policy of 10 random training seeds using LQ-Learning or LDQN. The bar segments show the proportion of timesteps spent waiting vs. actively engaging with the environment. Bar heights are normalized; the numbers within each segment denote the absolute number of timesteps and the number at the top denotes absolute episode length.

[4] Any policy that performs a duration-15 waiting macro action would take at least 37 timesteps to complete `Cook`.

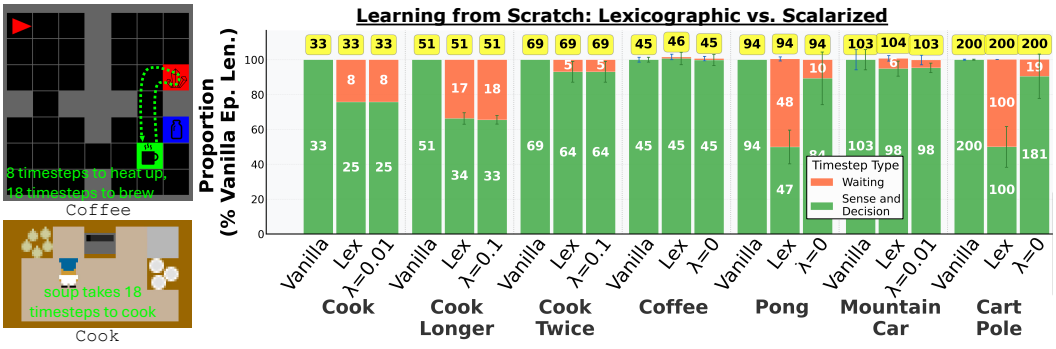

Figure 3: (**Left**) Environment visualizations. In our `Coffee` interleaving case study, the duration-15 wait taken while coffee brews permits us to interleave a "fetch sugar" policy that follows the dashed green path. (**Right**) Empirical mean episode length (height of bar) and number of decisions made (height of green bar segment) of our approach ("Lex" denotes LQ-Learning for discrete environments, LDQN for others) and the best scalarized approach from our sweep over λ values. Episode length is negative cumulative task reward for most environments; for `CartPole` episode length is cumulative task reward. We also include results from vanilla RL (Q-Learning for discrete environments, DQN for others) that is trained on task reward without the option to wait. Within each environment, heights are normalized with respect to the vanilla RL baseline. Labels denote absolute numbers of timesteps. Error bars represents $\pm$ 1 standard deviation, computed over 10 random initial policy training seeds.

LQ-Learning is a good way to learn waiting (compared to the scalar reward baseline). We now investigate the extent to which our LQ-Learning/LDQN approach accurately finds a policy that toes the fine line between increasing waiting and sacrificing task performance. Figure 3 visualizes episode duration (which directly reflects cumulative task reward) and number of decisions made (zero minus cumulative waiting reward) for three approaches for each task: (1.) Vanilla RL that maximizes task performance with no access to waiting options, (2.) Our LQ-Learning/LDQN approach, and (3.) The scalarized objective approach with the best (as determined by Equation (1)) weighting coefficient found in our sweep over λ values. A policy that maximizes Equation (1) should yield episode length approximately equal to that achieved by Vanilla RL, and should wait as much or more than the scalarized approach. Our LQ-Learning and LDQN policies in Figure 3 indeed approximate this expected behavior.

Learning a waiting policy wrapper is sample efficient. We find it often takes fewer training steps to saturate the performance of a wrapper waiting policy than a waiting policy trained from scratch. For example, in `Pong` training a waiting wrapper policy via LDQN converges after $\sim$250k training environment steps, while training a waiting policy from scratch with LDQN typically takes more than 600k steps (Figure 4). This gain in sample efficiency may be due to (1) fewer actions being exposed to the wrapper policy and (2) a stark dichotomy between states from which the base policy can perform well vs. poorly, which means the waiting wrapper consistently receives low (relative to not waiting) future cumulative task reward when it waits inappropriately. Such a gain in sample efficiency need not exist in general, however learning curves in Appendix C show efficiency gains in each environment except `MountainCar`.

Waiting enables downstream interleaving. We now describe a simple form of *policy interleaving* as a downstream application of learning to wait. Recall from Figure 2 that our approach adapts a handwritten base policy that completes the `Coffee` task (by first turning on the coffee machine, then brewing coffee, then collecting sugar and cream) in 54 timesteps into a waiting policy that completes the task in the same amount of timesteps, but which commits to waiting. We now assume access to two additional handwritten policies, one for fetching sugar and another for fetching cream. Each of these auxiliary policies, when started from a location on the grid, brings the agent to collect its respective item, and returns to the grid position from which the auxiliary policy was invoked. We also assume access to accurate predictions of how long the auxiliary task will take to execute.

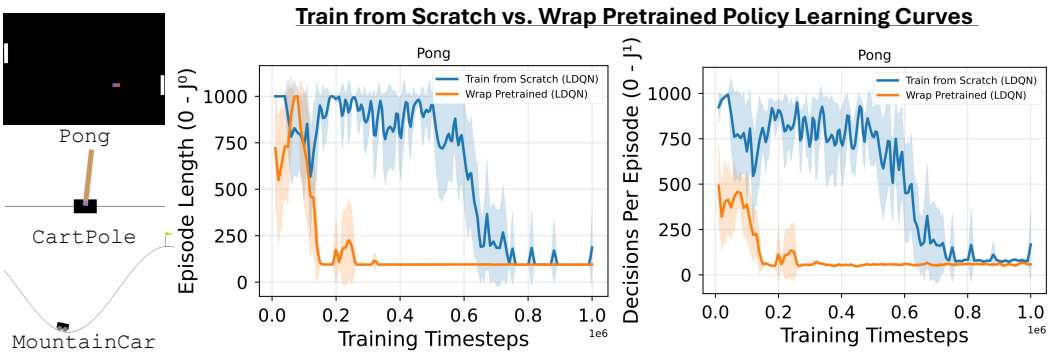

Figure 4: **(Left)** Environment visualizations. **(Right)** Learning curve comparison for our LDQN applied to train a waiting policy for `Pong` from scratch vs. as a wrapper around a frozen pretrained policy. Each curve is the mean of 10 random policy training seeds, shaded regions denote 95% confidence interval.

We then interleave as follows: if the waiting policy selects a duration-N wait, we check the time-to-completion estimates for each not-yet-completed auxiliary task and execute the longest-duration auxiliary task that can be completed in N timesteps. Our waiting wrapper policy's duration-15 waiting gap is long enough to fetch the sugar; performing this interleaved behavior results allows the agent to complete `Coffee` in a total of 45 timesteps, matching the amount of time taken to complete the task by vanilla Q-Learning applied to learning how to maximize task performance from scratch (Figure 3).

5 Related Work

Cost of Agent Intervention. Prior research (Hansen et al., 1996; Heemels et al., 2012; Nagahara et al., 2016; Zhou et al., 2024; Krishna et al., 2025) demonstrates how imposing costs on sensing and intervention compel the agent to adopt open-loop strategies in which the agent commits to a sequence of actions without intermediate feedback, typically by relying on internal model of the environment until uncertainty necessitates sensing. Closest in spirit to our work is Zhou et al. (2024), in which the agent selects both an action and a number of times to repeat the action, although they do not explore how adjusting the magnitude of their scalar *interaction cost* trades off between task performance and policy query frequency. While we also encourage the agent to sense and deliberate infrequently, we focus on a stricter notion of non-intervention formalized by the distinguished *wait* action. Committing to taking the wait action, i.e., surrendering control to the system's passive dynamics, is qualitatively different than committing to take a predetermined sequence of arbitrary actions because (1.) waiting may conserve motor resources (e.g. the energy needed to walk through a kitchen) and (2.) periods of waiting can be "filled in" with a policy that pursues an auxiliary goal, under appropriate conditions on the dynamics under the waiting action and the auxiliary goal policy.

Temporal Abstractions. RL has long utilized temporal abstractions to improve efficiency in complex domains (Sutton et al., 1999; Pateria et al., 2021; Hutsebaut-Buysse et al., 2022). Grounded in bounded rationality (Simon, 1957), Harb et al. (2018) argue that agents should identify temporally extended options that effectively minimize the *deliberation cost* associated with option selection. This is similar in spirit to Equation (1). In particular, Harb et al. (2018) seek to limit the number of times a high-level policy over options may be queried and propose a scalarized objective based on the Lagrangian formulation (analogous to our scalarized RL baseline Section 4.2) of this constrained optimization problem. However, a drawback compared to our proposed approach based on lexicographic RL is the need for extensive hyperparameter sweeps to identify appropriate weighting of deliberation cost (which corresponds to the λ value of our scalar approach) relative to the cumulative task reward. *Hierarchical RL with Timed Subgoals* (Gürtler et al., 2021) introduces fine-grained temporal abstraction by letting a high-level policy call *specify when each call to a low-level policy should terminate*. This is similar in spirit to dynamically-determined action repeat (Lakshminarayanan et al., 2017; Sharma et al., 2017; Biedenkapp et al., 2021) where, in addition to choosing a primitive action, the agent chooses a number timesteps to repeat that primitive

action. While these approaches reduce decision frequency in practice, their focus is reducing the amount of environmental interaction (sample complexity) needed to train a performant policy. This contrasts with our objective, which explicitly seeks to minimize the number of decisions the agent makes during deployment and privileges extended *waiting* as opposed to extended application of arbitrary primitive actions.

Multi-objective RL. Multi-objective MDPs (MOMDPs) are often employed when a single scalar reward cannot capture the trade-offs between competing desiderata (Roijers et al., 2013; Hayes et al., 2022). While optimizing a scalarized objective can make use of standard RL techniques, the objective itself requires precise tuning of weights to navigate the Pareto front. In contrast, lexicographic approaches (Gábor et al., 1998; Skalse et al., 2022) allow for a strict prioritization of objectives, which is well-suited for our setting in which we wish to wait as much as possible without sacrificing expected cumulative task reward. This is closely related to the motivation of constrained policy optimization in safe RL (García & Fernández, 2015; Gu et al., 2024); however, rather than choose an *a priori* constraint on the minimal acceptable task performance the lexicographic approach allows the agent to adaptively discover what the best possible task performance and optimize waiting subject to this implicit constraint.

6 Discussion

Summary. In this paper we formulated learning to wait as maximizing Equation (1), which encourages the agent to spend long durations committed to apply the *wait* action without sensing, to the extent possible without sacrificing expected cumulative task reward. Lexicographic MORL allows the agent to discover the optimal expected cumulative task reward, alleviating the need for a human to perform hyperparameter sweeps over a reward weighting coefficient. We can thus train a policy, or a wrapper around an existing policy, that waits as much as possible without sacrificing task performance. Waiting inherently saves sensory, computational, and motor resources and may be a useful first step towards multi-task policy interleaving.

Future work. Currently, our approach does not leverage the inherent *structure* of extended duration waiting. Future work could improve sample efficiency during training by e.g. fitting a single-step dynamics model to multi-timestep waiting experiences, taking inspiration from Zhou et al. (2024). Secondly, we only expect our techniques to work well in *fully observed* environments. If, for example, the agent's observations did not differentiate when the soup in the `Cook` task was just starting to cook vs. when it was almost ready, the agent would not be able to learn appropriate waiting behavior. An exciting direction of future work would be to learn *memoryful* waiting policies. In many practical settings (e.g. cooking) the task-relevant latent dynamics are highly correlated with the passage of time; which may permit accurate deterministic belief updates. Finally, we presented only an initial proof-of-concept case study of policy interleaving. To truly expose opportunities to perform auxiliary tasks would require a stringent definition of what actions cause the aspects of the environment's state that are relevant to the primary task to evolve as they would under the waiting action. Without such restrictions, the progress made towards the auxiliary task could impede the completion of the primary task.

Acknowledgments

This project was funded by NSF SLES 2331783, NSF CAREER 2239301, ONR N00014-22-1-2677, DARPA TIAMAT HR00112490421, and a gift from Amazon AWS to the ASSET center at Penn.

References

Andrew G. Barto, Richard S. Sutton, and Charles W. Anderson. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-13(5):834–846, 1983. DOI: 10.1109/TSMC.1983.6313077.

André Biedenkapp, Raghu Rajan, Frank Hutter, and Marius Lindauer. Temporl: Learning when to act. In Marina Meila and Tong Zhang (eds.), *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pp. 914–924. PMLR, 18–24 Jul 2021.

Kevin Black, Noah Brown, Danny Driess, Adnan Esmail, Michael Equi, Chelsea Finn, Niccolo Fusai, Lachy Groom, Karol Hausman, Brian Ichter, Szymon Jakubczak, Tim Jones, Liyiming Ke, Sergey Levine, Adrian Li-Bell, Mohith Mothukuri, Suraj Nair, Karl Pertsch, Lucy Xiaoyang Shi, James Tanner, Quan Vuong, Anna Walling, Haohuan Wang, and Ury Zhilinsky. π_0: A vision-language-action flow model for general robot control, 2026.

Anthony Brohan, Noah Brown, Justice Carbajal, Yevgen Chebotar, Xi Chen, Krzysztof Choromanski, Tianli Ding, Danny Driess, Avinava Dubey, Chelsea Finn, Pete Florence, Chuyuan Fu, Montse Gonzalez Arenas, Keerthana Gopalakrishnan, Kehang Han, Karol Hausman, Alexander Herzog, Jasmine Hsu, Brian Ichter, Alex Irpan, Nikhil Joshi, Ryan Julian, Dmitry Kalashnikov, Yuheng Kuang, Isabel Leal, Lisa Lee, Tsang-Wei Edward Lee, Sergey Levine, Yao Lu, Henryk Michalewski, Igor Mordatch, Karl Pertsch, Kanishka Rao, Krista Reymann, Michael Ryoo, Grecia Salazar, Pannag Sanketi, Pierre Sermanet, Jaspiar Singh, Anikait Singh, Radu Soricut, Huong Tran, Vincent Vanhoucke, Quan Vuong, Ayzaan Wahid, Stefan Welker, Paul Wohlhart, Jialin Wu, Fei Xia, Ted Xiao, Peng Xu, Sichun Xu, Tianhe Yu, and Brianna Zitkovich. Rt-2: Vision-language-action models transfer web knowledge to robotic control, 2023.

Micah Carroll, Rohin Shah, Mark K Ho, Tom Griffiths, Sanjit Seshia, Pieter Abbeel, and Anca Dragan. On the utility of learning about humans for human-ai coordination. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (eds.), *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

Maxime Chevalier-Boisvert, Bolun Dai, Mark Towers, Rodrigo de Lazcano, Lucas Willems, Salem Lahlou, Suman Pal, Pablo Samuel Castro, and Jordan Terry. Minigrid & miniworld: Modular & customizable reinforcement learning environments for goal-oriented tasks. *CoRR*, abs/2306.13831, 2023.

Zoltán Gábor, Zsolt Kalmár, and Csaba Szepesvári. Multi-criteria reinforcement learning. In *Proceedings of the Fifteenth International Conference on Machine Learning*, ICML '98, pp. 197–205, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc. ISBN 1558605568.

Javier García and Fernando Fernández. A comprehensive survey on safe reinforcement learning. *Journal of Machine Learning Research*, 16(42):1437–1480, 2015.

Ghost Town Games. Overcooked, 2016.

Shangding Gu, Long Yang, Yali Du, Guang Chen, Florian Walter, Jun Wang, and Alois Knoll. A review of safe reinforcement learning: Methods, theories, and applications. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 46(12):11216–11235, 2024. DOI: 10.1109/TPAMI.2024.3457538.

Nico Gürtler, Dieter Büchler, and Georg Martius. Hierarchical reinforcement learning with timed subgoals. In A. Beygelzimer, Y. Dauphin, P. Liang, and J. Wortman Vaughan (eds.), *Advances in Neural Information Processing Systems*, 2021.

Eric Hansen, Andrew Barto, and Shlomo Zilberstein. Reinforcement learning for mixed open-loop and closed-loop control. In M.C. Mozer, M. Jordan, and T. Petsche (eds.), *Advances in Neural Information Processing Systems*, volume 9. MIT Press, 1996.

Jean Harb, Pierre-Luc Bacon, Martin Klissarov, and Doina Precup. When waiting is not an option: Learning options with a deliberation cost. *Proceedings of the AAAI Conference on Artificial Intelligence*, 32(1), Apr. 2018. DOI: 10.1609/aaai.v32i1.11831.

Conor F. Hayes, Roxana Rădulescu, Eugenio Bargiacchi, Johan Källström, Matthew Macfarlane, Mathieu Reymond, Timothy Verstraeten, Luisa M. Zintgraf, Richard Dazeley, Fredrik Heintz, Enda Howley, Athirai A. Irissappane, Patrick Mannion, Ann Nowé, Gabriel Ramos, Marcello Restelli, Peter Vamplew, and Diederik M. Roijers. A practical guide to multi-objective reinforcement learning and planning. *Autonomous Agents and Multi-Agent Systems*, 36(1), April 2022. ISSN 1387-2532. DOI: 10.1007/s10458-022-09552-y.

W.P.M.H. Heemels, K.H. Johansson, and P. Tabuada. An introduction to event-triggered and self-triggered control. In *2012 IEEE 51st IEEE Conference on Decision and Control (CDC)*, pp. 3270–3285, 2012. DOI: 10.1109/CDC.2012.6425820.

Matthias Hutsebaut-Buysse, Kevin Mets, and Steven Latré. Hierarchical reinforcement learning: A survey and open research challenges. *Machine Learning and Knowledge Extraction*, 4(1):172–221, 2022. ISSN 2504-4990. DOI: 10.3390/make4010009.

Arjun Krishna, Edward S. Hu, and Dinesh Jayaraman. The value of sensory information to a robot. *ICLR*, 2025.

Aravind S. Lakshminarayanan, Sahil Sharma, and Balaraman Ravindran. Dynamic action repetition for deep reinforcement learning. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, AAAI'17, pp. 2133–2139. AAAI Press, 2017.

Robert Tjarko Lange. gymnax: A JAX-based reinforcement learning environment library, 2022.

Marlos C. Machado, Marc G. Bellemare, Erik Talvitie, Joel Veness, Matthew Hausknecht, and Michael Bowling. Revisiting the arcade learning environment: Evaluation protocols and open problems for general agents (extended abstract). In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*, pp. 5573–5577. International Joint Conferences on Artificial Intelligence Organization, 7 2018. DOI: 10.24963/ijcai.2018/787.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin A. Riedmiller, Andreas Kirkeby Fidjeland, Georg Ostrovski, Stig Petersen, Charlie Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518:529–533, 2015.

Andrew William Moore. Efficient memory-based learning for robot control. Technical report, University of Cambridge, 1990.

Masaaki Nagahara, Daniel E. Quevedo, and Dragan Nešić. Maximum hands-off control: A paradigm of control effort minimization. *IEEE Transactions on Automatic Control*, 61(3):735–747, 2016. DOI: 10.1109/TAC.2015.2452831.

NVIDIA, Johan Bjorck, Nikita Cherniadev Fernando Castañeda, Xingye Da, Runyu Ding, Linxi "Jim" Fan, Yu Fang, Dieter Fox, Fengyuan Hu, Spencer Huang, Joel Jang, Zhenyu Jiang, Jan Kautz, Kaushil Kundalia, Lawrence Lao, Zhiqi Li, Zongyu Lin, Kevin Lin, Guilin Liu, Edith Llontop, Loic Magne, Ajay Mandlekar, Avnish Narayan, Soroush Nasiriany, Scott Reed, You Liang Tan, Guanzhi Wang, Zu Wang, Jing Wang, Qi Wang, Jiannan Xiang, Yuqi Xie, Yinzhen Xu, Zhenjia Xu, Seonghyeon Ye, Zhiding Yu, Ao Zhang, Hao Zhang, Yizhou Zhao, Ruijie Zheng, and Yuke Zhu. GR00T N1: An open foundation model for generalist humanoid robots. In *ArXiv Preprint*, March 2025.

Octo Model Team, Dibya Ghosh, Homer Walke, Karl Pertsch, Kevin Black, Oier Mees, Sudeep Dasari, Joey Hejna, Charles Xu, Jianlan Luo, Tobias Kreiman, You Liang Tan, Lawrence Yunliang Chen, Pannag Sanketi, Quan Vuong, Ted Xiao, Dorsa Sadigh, Chelsea Finn, and Sergey Levine. Octo: An open-source generalist robot policy. In *Proceedings of Robotics: Science and Systems*, Delft, Netherlands, 2024.

Shubham Pateria, Budhitama Subagdja, Ah-hwee Tan, and Chai Quek. Hierarchical reinforcement learning: A comprehensive survey. *ACM Comput. Surv.*, 54(5), jun 2021. ISSN 0360-0300. DOI: 10.1145/3453160.

Antonin Raffin. Rl baselines3 zoo. https://github.com/DLR-RM/rl-baselines3-zoo, 2020.

Diederik M. Roijers, Peter Vamplew, Shimon Whiteson, and Richard Dazeley. A survey of multi-objective sequential decision-making. *J. Artif. Int. Res.*, 48(1):67–113, October 2013. ISSN 1076-9757.

Antonio Serrano-Muñoz, Dimitrios Chrysostomou, Simon Bøgh, and Nestor Arana-Arexolaleiba. skrl: Modular and flexible library for reinforcement learning. *Journal of Machine Learning Research*, 24 (254):1–9, 2023.

Sahil Sharma, Aravind S. Lakshminarayanan, and Balaraman Ravindran. Learning to repeat: Fine grained action repetition for deep reinforcement learning. In *International Conference on Learning Representations*, 2017.

Herbert Simon. A behavioral model of rational choice. *Models of man, social and rational: Mathematical essays on rational human behavior in a social setting*, 6(1):241–260, 1957.

Joar Skalse, Lewis Hammond, Charlie Griffin, and Alessandro Abate. Lexicographic multi-objective reinforcement learning. In Lud De Raedt (ed.), *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI-22*, pp. 3430–3436. International Joint Conferences on Artificial Intelligence Organization, 7 2022. DOI: 10.24963/ijcai.2022/476. Main Track.

Richard S. Sutton, Doina Precup, and Satinder Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112(1):181–211, 1999. ISSN 0004-3702.

Emanuel Todorov. Efficient computation of optimal actions. *Proceedings of the national academy of sciences*, 106(28):11478–11483, 2009.

Mark Towers, Ariel Kwiatkowski, Jordan Terry, John U Balis, Gianluca De Cola, Tristan Deleu, Manuel Goulão, Andreas Kallinteris, Markus Krimmel, Arjun KG, et al. Gymnasium: A standard interface for reinforcement learning environments. *arXiv preprint arXiv:2407.17032*, 2024.

Helen Zhou, Audrey Huang, Kamyar Azizzadenesheli, David Childers, and Zachary Lipton. Timing as an action: Learning when to observe and act. In *International Conference on Artificial Intelligence and Statistics*, pp. 3979–3987. PMLR, 2024.

Supplementary Materials

The following content was not necessarily subject to peer review.

A Additional Environment Details

Cook. The `Cook`, `CookLonger`, and `CookTwice`, environments are implemented using Carroll et al. (2019). All three variants use the same kitchen layout shown in Figure 3. The environment has deterministic 2D grid dynamics: the agent has 6 actions available: go left, go right, go up, go down, stay=*wait*, or interact. Interact will pick up items (onion, bowl, finished soup) or put down the object the agent is holding. In Figure 1 we write "turn-left" to mean go left, and "take-soup" to mean interact. The agent's discrete observation space includes the agent's 2D position, orientation, whether the agent is holding an {onion, dish, ready soup, uncooked soup}, the pot phase ("empty", "has some but not all ingredients", "is cooking", or "is done"), how many ingredients are in the pot, and the time remaining for a currently cooking soup to be done. The maximum episode horizon H is 200 for each variant.

Coffee. The `Coffee` environment is a 2D gridworld implemented using Minigrid (Chevalier-Boisvert et al., 2023). The discrete state is defined by the agent's position, orientation (north, south, east, or west), whether the sugar has been collected or not, whether the cream has been collected or not, the progress of the coffee machine towards heating up (not toggled yet, the number of timesteps remaining to heat, or that it is fully heated), the progress of the coffee machine towards brewing (not toggled yet, the number of timesteps remaining to brew, or that the coffee is brewed and ready to collect), and whether the coffee has been collected or not. The agent observes the complete state at each timestep. We use the standard set of 7 discrete actions available to a Minigrid agent but define custom behavior (indicated in parentheses where appropriate): "turn left", "turn right", "go forward", "pickup" (no-op), "drop" (no-op), "toggle", and "done"=*wait* (we redefine "done" to make the agent stay in place with no special effect on episode termination). The "toggle" action is used for all interactions with the world (starting coffee machine to warm up, starting coffee machine to brew, collecting coffee, collecting cream, collecting sugar). The maximum episode horizon is $H = 200$. A truly optimal policy completes `Coffee` in 41 timesteps, but none of our learning based approaches learned the optimal behavior during our allotted training duration (10m timesteps); our best policies need 45 timesteps.

In our interleaving case study, the "fetch sugar" and "fetch cream" sub-policies are handwritten optimal policies that, when called for a particular agent position and orientation, make the agent go to the sugar (resp. cream) location, "toggle" to pick up the sugar (resp. cream), and return the agent to the position and orientation from which the subpolicy was called. It might seem counterintuitive that our simple scheduler does not choose to execute the "fetch cream" policy during the duration-5 wait that the wrapper waiting policy (wrapped around our handwritten policy for the entire `Coffee` task) executes while waiting for the coffee machine to heat up. This is because it would take more than 5 timesteps to collect the cream *and return to the position and orientation from the start of the duration-5 wait.* Overall task completion would be faster if the agent were able to fetch the cream while the coffee machine heats up; future work could explore more sophisticated forms of policy interleaving to enable such behavior.

Pong. We adapt Gymnax's (Lange, 2022) implementation of the well-known Pong environment. The agent's primitive action space $\mathcal{A}$ comprises four actions: `move_paddle_up` `move_paddle_down`, *wait* (which leaves the paddle in place), and `hit` which leaves the paddle in place and returns the ball if contact would be made. We include the nonstandard `hit` action to make the connection between waiting and not actively affecting the environment thematically clear; our techniques would work equally well with standard pong dynamics. The goal is to score one point as fast as possible against a scripted opponent; the maximum episode horizon is $H = 1000$. Our `Pong` environment includes the standard "sticky actions" introduced by Machado et al. (2018) to introduce a small amount of stochasticity into transitions. This makes the environment very slightly non-Markovian with respect to the observations supplied to the agent.

CartPole. We adapt the standard `CartPole-v0` (Barto et al., 1983) environment from Gymnasium (Towers et al., 2024) to have an additional *wait* action that applies no force to the cart. Thus the action set is {"left", "right", *wait*}. The maximum episode horizon is $H = 200$ and the task reward is $+1$ for each step. We retain the typical termination logic from the original implementation.

MountainCar. We use the standard `MountainCar-v0` (Moore, 1990) environment implementation from Gymnasium. There are three primitive actions: "accelerate to the left", "don't accelerate"=*wait*, and "accelerate to the right." The task reward is -1 per timestep and the maximum episode horizon is $H = 200$.

B Training Details

During training, we terminate training episodes once the *a priori* episode horizon is reached or (in the case of goal-reaching tasks) when the goal is reached. For `CartPole` we inherit the usual termination behavior when the pole's angle becomes excessively low.

For each of `Cook`, `CookLonger`, `CookTwice`, and `Coffee` we train each policy for 10 million environment steps. For each of our continuous state tasks we train each policy for 1 million timesteps.

When selecting the best model checkpoint of a training run, we use the environment's cumulative reward (for LQ-Learning and LDQN we use the lexicographic ordering over the task and waiting rewards, for the other approaches we use the usual ordering over the (scalarized or vanilla) reward). We evaluate an intermediate checkpoint every 10k training environment steps, and evaluate 20 eval episodes (using greedy, not epsilon-greedy policy inference) to obtain an empirical estimate of the expected cumulative reward.

We report the hyperparameters used for all experiments. Lexicographic Q-learning parameters are shown in Table 1, and lexicographic DQN parameters are shown in Table 2. DQN hyperparameters were adapted from the RL Baselines3 Zoo (Raffin, 2020) and manually tuned for each environment.

B.1 Lexicographic RL Implementation

We adapt the Q-Learning and DQN implementations from SKRL (Serrano-Muñoz et al., 2023) into the LQ-Learning and LDQN algorithms following Skalse et al. (2022). We also add support for durative macro-actions as described for the *SMDP Q-Learning* algorithm in Sutton et al. (1999). Durative macro-actions were not part of the standard MOMDP environments treated by Skalse et al. (2022) or the base implementation of SKRL.

Our Q-Learning and LQ-Learning implementations do not make use of a replay buffer. Our DQN and LDQN implementations do use a replay buffer: we insert durative-action transitions into the replay buffer in the straightforward way, that is, we include the state from which the action was taken, the (durative) action, the state reached after finishing (or reaching episode termination) the durative action, and the cumulative reward accrued during the low-level transitions within the durative action.

Table 1: LQ-Learning hyperparameters.

Parameter	Symbol	Value
Discount factor	γ	1.0
Learning rate	α	0.1
Exploration rate	ε	0.05
Lexicographic slack	σ	0.001

C Learning Curves

We now present learning curves for the training runs reported in Figure 2 and Figure 3. Note that Figure 3 presents just the best (in terms Equation (1) for LQ-Learning and LDQN and in terms of scalarized

Table 2: Lexicographic DQN hyperparameters by environment.

Parameter	**CartPole**	**MountainCar**	**Pong**
Network Architecture			
Hidden layers	[256,256]	[256,256]	[256,256]
Activation	ReLU	ReLU	ReLU
Core Hyperparameters			
Discount factor (γ)	1.0	1.0	1.0
Learning rate (α)	2×10^{-3}	4×10^{-3}	4×10^{-4}
Batch size	64	128	32
Replay Buffer			
Buffer size	100,000	10,000	100,000
Training Schedule			
Learning starts	1,000	1,000	1,000
Gradient steps	128	8	1
Update interval	256	16	4
Target Network			
Update interval	10	600	1,000
Polyak (τ)	1.0	1.0	1.0
Exploration (ε-greedy)			
Initial ε	1.0	1.0	1.0
Final ε	0.04	0.07	0.01
Decay timesteps	8,000	24,000	100,000

return for scalarized approaches) policy checkpoint, which may appear before the max number of training environment steps is reached. In the learning curves, the bold line shows the average of ten training seeds. The shaded region shows 1 standard deviation. For some environments, notably `CookTwice`, policy optimization is unstable, meaning that averaging performance across ten random seeds does not clearly reflect the performance of each individual seed's policy.

For all approaches, we provide learning curves for episode length (negative J^0 for most environments; positive J^0 for `CartPole`). This characterizes the cumulative task reward for our lexicographic MORL and the cumulative reward for vanilla RL. For all approaches except vanilla RL, we also include a curve for number of decisions taken (negative J^1). This characterizes the cumulative waiting reward for LQ-Learning and LDQN approaches. For the scalarized approaches we also include a learning curve for the scalarized return, which is always a linear combination of J^0 and J^1 weighted by the coefficient λ as described in Section 4.2.

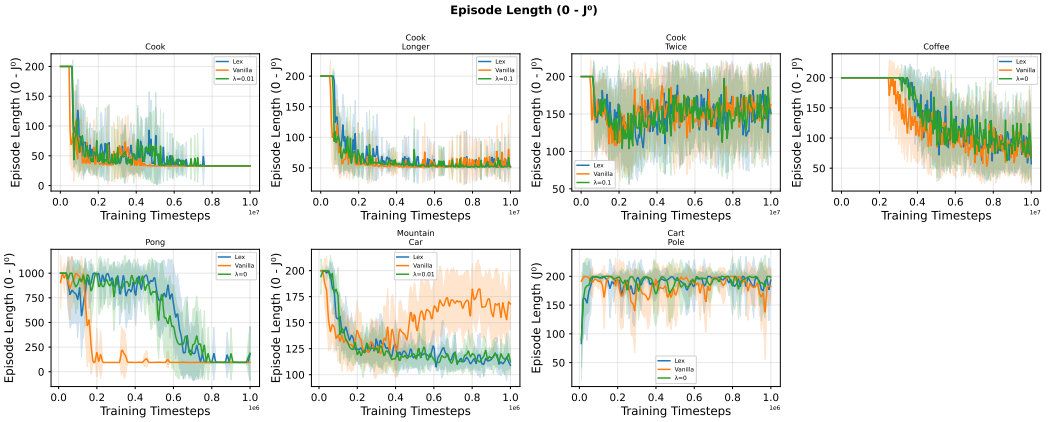

Figure 5: Episode length vs. training timesteps for learning from scratch experiments (see Figure 3). Episode length equals negative cumulative task return for most environments and equals positive task return for `CartPole`. Each curve is the mean of 10 random policy training seeds; shaded regions show $\pm$ 1 standard deviation. Each point along the curve is the empirical mean over 20 evaluation trajectories (with greedy policy inference); evaluation occurs once every 10,000 environment steps during training.

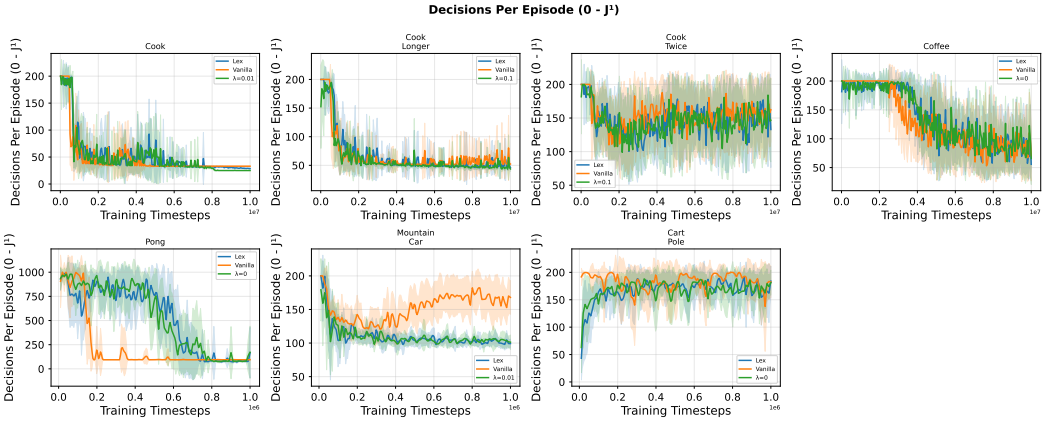

Figure 6: Decisions per episode vs. training timesteps for learning from scratch experiments (see Figure 3). Decisions per episode is negative cumulative waiting return for all environments. Each curve is the mean of 10 random policy training seeds; shaded regions show $\pm$ 1 standard deviation. Each point along the curve is the empirical mean over 20 evaluation trajectories (with greedy policy inference); evaluation occurs once every 10,000 environment steps during training.

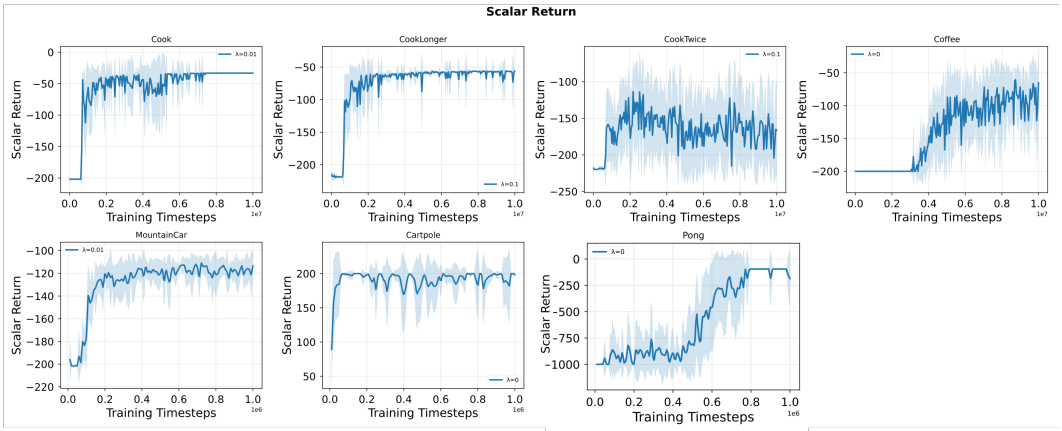

Figure 7: Per-episode cumulative scalar reward vs. training timesteps for learning from scratch experiments (see Figure 3). Each curve is the mean of 10 random policy training seeds; shaded regions represents $\pm$ 1 standard deviation. Each point along the curve is the empirical mean over 20 evaluation trajectories (with greedy policy inference); evaluation occurs once every 10,000 environment steps during training. We only report curves for the best scalarization coefficient λ, as described in Section 4.

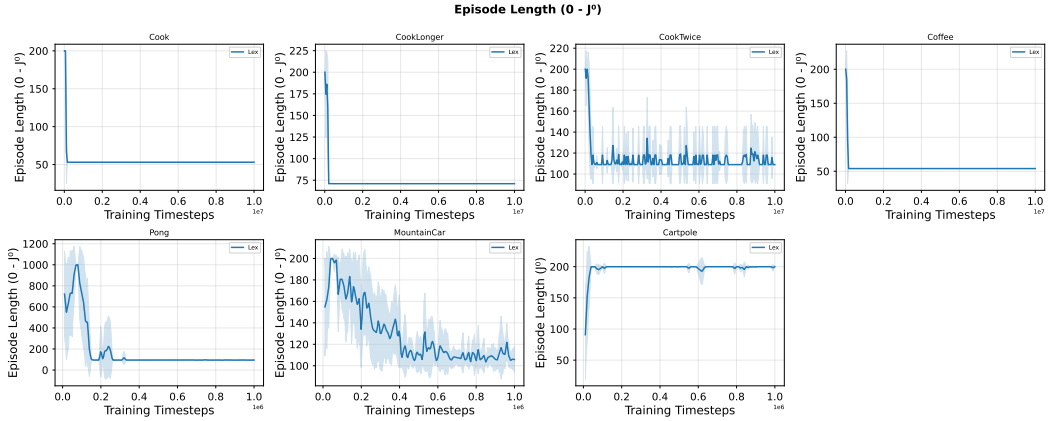

Figure 8: Episode length vs. training timesteps for learning from wrapper waiting policy experiments (see Figure 2). Episode length equals negative task return for most environments and equals positive task return for `CartPole`. Each curve is the mean of 10 random policy training seeds; shaded regions represents $\pm$ 1 standard deviation. Each point along the curve is the empirical mean over 20 evaluation trajectories (with greedy policy inference); evaluation occurs once every 10,000 environment steps during training.

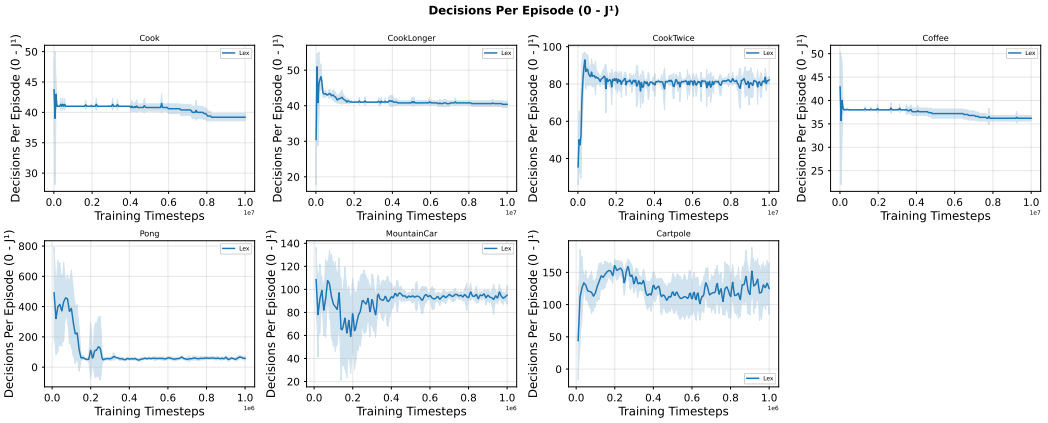

Figure 9: Decisions per episode vs. training timesteps for learning from wrapper waiting policy experiments (see Figure 2). Decisions per episode is negative waiting return for all environments. Each curve is the mean of 10 random policy training seeds; shaded regions represents $\pm$ 1 standard deviation. Each point along the curve is the empirical mean over 20 evaluation trajectories (with greedy policy inference); evaluation occurs once every 10,000 environment steps during training.

C.1 Extended Tabular results

In the following results tables, we report the mean and standard deviation for all experiments performed. This includes scalarized training values of λ that do not appear in Figure 3. For some environments, we extended our sweep to additional values of λ. Unless otherwise specified, we report mean and standard deviation over 10 training runs. Our results for learning a waiting wrapper policy using scalarized rewards are reported over 5 training runs.

Method	Episode Length $(0-J^0)$	Policy Queries per Episode $(0-J^1)$
Scratch LQ-Learning	33.0 ± 0.0	25.0 ± 0.0
Scratch scalar 0	33.0 ± 0.0	32.2 ± 1.687
Scratch scalar 0.01	33.0 ± 0.0	25.0 ± 0.0
Scratch scalar 0.1	33.0 ± 0.0	25.0 ± 0.0
Scratch scalar 0.5	33.0 ± 0.0	25.0 ± 0.0
Scratch scalar 1	33.0 ± 0.0	25.0 ± 0.0
Scratch scalar 5	37.0 ± 0.0	23.0 ± 0.0
Scratch scalar 10	37.0 ± 0.0	23.0 ± 0.0
Scratch scalar 15	37.0 ± 0.0	23.0 ± 0.0
Scratch scalar 20	102.2 ± 84.173	19.4 ± 4.648
Scratch vanilla Q-Learning	33.0 ± 0.0	33.0 ± 0.0
Wrap LQ-Learning	53.0 ± 0.0	39.0 ± 0.0
Wrap scalar 0	53.0 ± 0.0	53.0 ± 0.0
Wrap scalar 0.01	53.0 ± 0.0	39.0 ± 0.0
Wrap scalar 0.1	53.0 ± 0.0	39.0 ± 0.0
Wrap scalar 0.5	53.0 ± 0.0	39.0 ± 0.0
Wrap scalar 1	53.0 ± 0.0	39.0 ± 0.0
Wrap scalar 5	56.0 ± 0.0	38.0 ± 0.0
Wrap scalar 10	200.0 ± 0.0	14.2 ± 0.447
Wrap scalar 15	56.0 ± 0.0	38.0 ± 0.0
Wrap scalar 20	200.0 ± 0.0	14.2 ± 0.447

Table 3: `Cook` results. "Wrap" refers to wrapping a base policy that is handwritten and which stands idly by as soup cooks.

Method	Episode Length $(0-J^0)$	Policy Queries per Episode $(0-J^1)$
Scratch LQ-Learning	51.0 ± 0.0	33.8 ± 1.687
Scratch scalar 0	51.0 ± 0.0	48.2 ± 1.932
Scratch scalar 0.01	51.0 ± 0.0	34.4 ± 1.897
Scratch scalar 0.1	51.0 ± 0.0	33.4 ± 1.265
Scratch scalar 0.5	51.4 ± 0.966	31.2 ± 2.53
Scratch scalar 1	51.4 ± 0.966	29.8 ± 1.751
Scratch scalar 5	52.0 ± 0.0	24.0 ± 0.0
Scratch scalar 10	52.0 ± 0.0	24.0 ± 0.0
Scratch scalar 15	111.2 ± 76.427	20.0 ± 5.164
Scratch vanilla Q-Learning	51.0 ± 0.0	51.0 ± 0.0
Wrap LQ-Learning	71.0 ± 0.0	40.2 ± 1.033
Wrap scalar 0	71.0 ± 0.0	71.0 ± 0.0
Wrap scalar 0.01	71.0 ± 0.0	39.0 ± 0.0
Wrap scalar 0.1	71.0 ± 0.0	39.0 ± 0.0
Wrap scalar 0.5	71.0 ± 0.0	39.0 ± 0.0
Wrap scalar 1	71.0 ± 0.0	39.0 ± 0.0
Wrap scalar 5	71.0 ± 0.0	39.0 ± 0.0
Wrap scalar 10	200.0 ± 0.0	14.2 ± 0.447
Wrap scalar 15	200.0 ± 0.0	14.0 ± 0.0

Table 4: `CookLonger` results. "Wrap" refers to wrapping a base policy that is handwritten and which stands idly by as soup cooks.

Method	Episode Length $(0-J^0)$	Policy Queries per Episode $(0-J^1)$
Scratch LQ-Learning	69.0 ± 0.0	64.2 ± 4.131
Scratch scalar 0	69.2 ± 0.632	68.4 ± 2.675
Scratch scalar 0.01	69.0 ± 0.0	65.0 ± 4.216
Scratch scalar 0.1	69.0 ± 0.0	64.2 ± 4.131
Scratch scalar 0.5	69.8 ± 1.033	61.0 ± 2.309
Scratch scalar 1	71.8 ± 3.676	57.4 ± 4.502
Scratch scalar 5	200.0 ± 0.0	15.1 ± 1.287
Scratch scalar 10	200.0 ± 0.0	15.7 ± 2.946
Scratch scalar 15	200.0 ± 0.0	14.5 ± 0.527
Scratch scalar 20	200.0 ± 0.0	14.5 ± 0.527
Scratch vanilla Q-Learning	69.0 ± 0.0	69.0 ± 0.0
Wrap LQ-Learning	109.0 ± 0.0	81.0 ± 0.0
Wrap scalar 0	109.0 ± 0.0	81.0 ± 0.0
Wrap scalar 0.01	109.0 ± 0.0	81.0 ± 0.0
Wrap scalar 0.1	109.0 ± 0.0	81.0 ± 0.0
Wrap scalar 0.5	109.0 ± 0.0	81.0 ± 0.0
Wrap scalar 1	200.0 ± 0.0	14.0 ± 0.0
Wrap scalar 5	200.0 ± 0.0	14.0 ± 0.0
Wrap scalar 10	200.0 ± 0.0	14.0 ± 0.0
Wrap scalar 15	200.0 ± 0.0	14.0 ± 0.0
Wrap scalar 20	200.0 ± 0.0	14.0 ± 0.0

Table 5: `CookTwice` results. "Wrap" refers to wrapping a base policy that is handwritten and which stands idly by as soup cooks.

Method	Episode Length ($0-J^0$)	Policy Queries per Episode ($0-J^1$)
Scratch LQ-Learning	45.5 ± 0.707	45.1 ± 1.595
Scratch scalar 0	45.1 ± 0.568	44.7 ± 1.418
Scratch scalar 0.01	45.3 ± 0.823	44.5 ± 1.179
Scratch scalar 0.1	45.5 ± 0.972	45.1 ± 1.101
Scratch scalar 0.5	45.4 ± 0.843	41.0 ± 2.494
Scratch scalar 1	45.75 ± 1.165	40.5 ± 3.295
Scratch scalar 2	47.6 ± 2.797	38.0 ± 2.211
Scratch scalar 5	49.0 ± 3.162	36.429 ± 1.272
Scratch scalar 10	156.1 ± 70.699	21.1 ± 9.231
Scratch scalar 15	200.0 ± 0.0	14.4 ± 0.699
Scratch scalar 20	200.0 ± 0.0	14.2 ± 0.422
Scratch scalar 30	200.0 ± 0.0	14.8 ± 1.229
Scratch vanilla Q-Learning	44.8 ± 0.632	44.8 ± 0.632
Wrap LQ-Learning	54.0 ± 0.0	36.0 ± 0.0
Wrap scalar 0	54.0 ± 0.0	54.0 ± 0.0
Wrap scalar 0.01	54.0 ± 0.0	36.0 ± 0.0
Wrap scalar 0.1	54.0 ± 0.0	36.0 ± 0.0
Wrap scalar 0.5	54.0 ± 0.0	36.0 ± 0.0
Wrap scalar 1	54.0 ± 0.0	36.0 ± 0.0
Wrap scalar 2	54.0 ± 0.0	36.0 ± 0.0
Wrap scalar 5	57.0 ± 0.0	35.0 ± 0.0
Wrap scalar 10	200.0 ± 0.0	14.0 ± 0.0
Wrap scalar 15	200.0 ± 0.0	14.0 ± 0.0
Wrap scalar 20	200.0 ± 0.0	14.4 ± 0.894
Wrap scalar 30	200.0 ± 0.0	14.2 ± 0.447

Table 6: `Coffee` Results. "Wrap" refers to wrapping a base policy that is handwritten and which stands idly by as the coffe machine heats and as coffee brews.

Method	Episode Length ($0-J^0$)	Policy Queries per Episode ($0-J^1$)
Scratch LDQN	94.414 ± 1.091	46.898 ± 9.105
Scratch scalar 0	94.0 ± 0.0	83.929 ± 14.225
Scratch scalar 0.01	94.09 ± 0.242	45.146 ± 10.493
Scratch scalar 0.05	94.227 ± 0.254	45.606 ± 8.269
Scratch scalar 0.1	94.664 ± 1.523	44.793 ± 7.438
Scratch scalar 0.2	94.1 ± 0.217	46.874 ± 3.954
Scratch scalar 0.5	94.083 ± 0.224	33.915 ± 4.048
Scratch scalar 1	94.208 ± 0.368	26.525 ± 2.934
Scratch scalar 2	94.449 ± 0.508	22.517 ± 2.687
Scratch scalar 5	95.203 ± 1.67	22.515 ± 4.199
Scratch scalar 10	94.694 ± 1.309	25.176 ± 4.199
Scratch scalar 15	118.437 ± 30.631	25.326 ± 4.026
Scratch scalar 20	94.025 ± 0.06	24.42 ± 2.995
Scratch scalar 30	102.392 ± 22.485	27.073 ± 3.162
Scratch scalar 50	564.16 ± 459.882	15.403 ± 8.02
Scratch scalar 100	1000.0 ± 0.0	8.2 ± 0.632
Scratch vanilla DQN	94.0 ± 0.0	94.0 ± 0.0
Wrap LDQN	94.439 ± 0.415	30.499 ± 3.497
Wrap scalar 0	94.399 ± 0.231	91.586 ± 6.197
Wrap scalar 0.01	94.401 ± 0.286	32.373 ± 2.655
Wrap scalar 0.05	94.409 ± 0.344	29.72 ± 1.54
Wrap scalar 0.1	94.332 ± 0.477	29.733 ± 2.613
Wrap scalar 0.2	94.363 ± 0.329	27.433 ± 2.794
Wrap scalar 0.5	94.455 ± 0.286	23.748 ± 1.966
Wrap scalar 1	94.744 ± 0.271	21.842 ± 0.864
Wrap scalar 10	96.362 ± 2.502	20.559 ± 0.544
Wrap scalar 2	95.199 ± 1.779	20.469 ± 0.727
Wrap scalar 5	96.778 ± 1.526	20.264 ± 0.748
Wrap scalar 15	281.496 ± 401.745	30.734 ± 15.112
Wrap scalar 20	287.182 ± 399.216	21.642 ± 8.706
Wrap scalar 30	1000.0 ± 0.0	8.0 ± 0.0
Wrap scalar 50	1000.0 ± 0.0	8.4 ± 0.548
Wrap scalar 100	1000.0 ± 0.0	8.0 ± 0.0

Table 7: `Pong` results. "Wrap" refers to wrapping a base policy that is trained using vanilla DQN.

Method	Episode Length (J^0)	Policy Queries per Episode ($0-J^1$)
Scratch LDQN	199.986 $\pm$ 0.042	99.76 $\pm$ 23.421
Scratch scalar 0	200.0 $\pm$ 0.0	180.72 $\pm$ 25.435
Scratch scalar 0.01	199.952 $\pm$ 0.106	83.298 $\pm$ 28.076
Scratch scalar 0.1	199.846 $\pm$ 0.487	81.455 $\pm$ 26.969
Scratch scalar 0.5	199.647 $\pm$ 1.116	66.388 $\pm$ 19.676
Scratch scalar 1	199.843 $\pm$ 0.345	64.69 $\pm$ 24.025
Scratch scalar 2	162.941 $\pm$ 71.322	49.522 $\pm$ 29.661
Scratch scalar 5	128.33 $\pm$ 66.645	23.113 $\pm$ 18.95
Scratch scalar 10	145.52 $\pm$ 55.1	21.197 $\pm$ 15.645
Scratch scalar 15	41.68 $\pm$ 1.042	2.535 $\pm$ 0.067
Scratch scalar 20	41.135 $\pm$ 0.37	2.498 $\pm$ 0.019
Scratch vanilla DQN	199.727 $\pm$ 0.542	199.727 $\pm$ 0.542
Wrap LDQN	199.973 $\pm$ 0.084	55.359 $\pm$ 11.093
Wrap scalar 0	200.0 $\pm$ 0.0	134.08 $\pm$ 36.439
Wrap scalar 0.01	199.824 $\pm$ 0.368	52.035 $\pm$ 7.038
Wrap scalar 0.1	200.0 $\pm$ 0.0	56.029 $\pm$ 7.818
Wrap scalar 0.5	199.959 $\pm$ 0.093	52.433 $\pm$ 11.197
Wrap scalar 1	200.0 $\pm$ 0.0	55.852 $\pm$ 7.065
Wrap scalar 2	197.424 $\pm$ 5.76	52.249 $\pm$ 5.758
Wrap scalar 5	199.087 $\pm$ 1.577	26.8 $\pm$ 4.069
Wrap scalar 10	174.452 $\pm$ 9.828	15.105 $\pm$ 1.088
Wrap scalar 15	41.347 $\pm$ 0.308	2.512 $\pm$ 0.018
Wrap scalar 20	41.409 $\pm$ 0.27	2.518 $\pm$ 0.03

Table 8: `Cartpole` Results. "Wrap" refers to wrapping a base policy that was trained using vanilla DQN.

Method	Episode Length ($0-J^0$)	Policy Queries per Episode ($0-J^1$)
Scratch LDQN	103.577 ± 1.631	97.532 ± 4.436
Scalar 0	103.626 ± 3.505	98.212 ± 3.246
Scalar 0.01	102.601 ± 2.841	97.978 ± 2.771
Scalar 0.1	105.242 ± 3.447	96.311 ± 3.548
Scalar 0.5	107.898 ± 4.655	88.284 ± 4.549
Scalar 1	125.611 ± 27.028	78.364 ± 24.184
Scalar 2	200.0 ± 0.0	10.613 ± 1.627
Scalar 5	200.0 ± 0.0	10.101 ± 0.318
Scalar 10	200.0 ± 0.0	10.232 ± 0.518
Scalar 15	200.0 ± 0.0	10.054 ± 0.131
Scalar 20	200.0 ± 0.0	10.117 ± 0.343
Scratch vanilla DQN	102.791 ± 5.985	102.791 ± 5.985
Wrap LDQN	101.512 ± 2.908	95.693 ± 3.791
Wrap scalar 0	100.284 ± 0.314	95.194 ± 3.462
Wrap scalar 0.01	100.559 ± 0.782	94.772 ± 3.535
Wrap scalar 0.1	101.703 ± 1.28	93.395 ± 3.979
Wrap scalar 0.5	104.484 ± 4.043	90.104 ± 7.571
Wrap scalar 1	114.675 ± 21.038	85.823 ± 19.539
Wrap scalar 2	200.0 ± 0.0	10.395 ± 0.86
Wrap scalar 5	200.0 ± 0.0	10.033 ± 0.052
Wrap scalar 10	200.0 ± 0.0	10.011 ± 0.025
Wrap scalar 15	200.0 ± 0.0	10.044 ± 0.062
Wrap scalar 20	200.0 ± 0.0	10.013 ± 0.029
Wrap scalar 50	200.0 ± 0.0	10.0 ± 0.0

Table 9: Results for `MountainCar` environment. "Wrap" refers to wrapping a base policy that is trained using vanilla DQN.