# Agent-State Construction with Auxiliary Inputs

by

Ruo Yu Tao

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science
University of Alberta

© Ruo Yu Tao, 2022

# Abstract

In most, if not every, realistic sequential decision-making tasks, the decision-making agent is not able to model the full complexity of the world. In reinforcement learning, the environment is often much larger and more complex than the agent, a setting also known as partial observability. In such settings, the agent must leverage more than just the current sensory inputs; it must construct an agent state that summarizes the agent's previous interactions with the world. Currently, the most common approach to tackle such a problem is to learn the agent-state function with a recurrent network. This is done with the agent's sensory stream as input, which is often augmented with transformations of the agent's observation. These augmentations are done in multiple ways, from simple approaches like concatenating observations to more complex ones such as uncertainty estimates or predictive representations. Nevertheless, although ubiquitous in the field, these additional inputs, which we term *auxiliary inputs*, are rarely emphasized, and it is not clear what their role or impact is. In this work we formalize agent-state construction with auxiliary inputs and present several examples of auxiliary inputs that incorporate information from the past, present, and/or future of the agent-environment interaction. We show that auxiliary inputs allow an agent to discriminate between observations that would otherwise be aliased, leading to more expressive features that smoothly interpolate between different states. We empirically evaluate this agent-state construction with different function approximators, using different instantiations of these auxiliary inputs across a variety of tasks. This approach is complementary to state-of-the-art methods such as recurrent neural networks, and acts as a heuristic that facilitates longer temporal credit assignment, re-

ducing the number of time steps needed when performing truncated backpropagation through time and leading to better performance.

*"Not knowing the chance of mutually exclusive events and knowing the chance to be equal are two quite different states of knowledge."*

—Ronald A. Fisher

*To my sister, Ruoshan.*

# Acknowledgements

I would like to begin by acknowledging and thanking my wonderful and supportive advisors, Marlos C. Machado and Adam White for their guidance, support, and trust during my time as a master's student. They have done a terrific job shaping me into a curious, principled, and careful scholar—most of what I know as a researcher has been from lessons they have taught and guidance they have given as their student. I am very excited to see all the interesting research we'll be collaborating on in the future! I would also like to thank Martha White for the help and advice she has given me in my projects and as a nascent academic.

I am incredibly thankful for my lab mates in the Reinforcement Learning and Artificial Intelligence Lab for the community, discussions, and guidance during my time as a graduate student at the University of Alberta. Particularly to this thesis, I would like to thank Matthew Schlegel for supporting and shaping my ideas and research direction throughout this project. I have learnt so much about research, neuroscience, predictions, and reinforcement learning in general from both you and our many talks and discussions. Besides Matt, I would like to thank Shibhansh Dohare, Prabhat Nagarajan, Khurram Javed, and Chunlok Lo for being both great friends and amazing lab mates.

I am also incredibly grateful for two particular lifelong friends, Liam Peet-Pare and Bradley Burega, for their friendship and support during my time in Edmonton. I look forward to the many adventures together after we all move on to the next part of our lives. Finally, I would like to thank the many wonderful people in the Alberta Machine Intelligence Institute for the community over the past two years.

# Table of Contents


**1 Introduction** **1**

1.1 Thesis Objectives and Contributions . . . . . . . . . . . . . . . . . . 3

**2 Background** **6**

2.1 Sequential Decision Making and Markov Decision Processes . . . . . . 7

2.2 Value Functions . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8

2.3 Bellman Equations and Value Iteration for Control . . . . . . . . . . 9

2.4 Temporal-Difference Learning and Control with Sarsa . . . . . . . . . 11

2.5 Value Function Approximation . . . . . . . . . . . . . . . . . . . . . 12

2.6 Deep Reinforcement Learning . . . . . . . . . . . . . . . . . . . . . 13

2.7 Partial Observability . . . . . . . . . . . . . . . . . . . . . . . . . . 14

2.8 Agent-State Functions . . . . . . . . . . . . . . . . . . . . . . . . . 15

2.9 Recurrent Neural Networks as Agent-State Functions . . . . . . . . . 16

**3 Auxiliary Inputs** **18**

3.1 Formalism and Examples . . . . . . . . . . . . . . . . . . . . . . . . 19

3.1.1 Frame Stacking . . . . . . . . . . . . . . . . . . . . . . . . . 20

3.1.2 Resolution and Depth . . . . . . . . . . . . . . . . . . . . . . 21

3.1.3 Incremental Functions . . . . . . . . . . . . . . . . . . . . . . 21

3.2 The Lobster Environment . . . . . . . . . . . . . . . . . . . . . . . 22

3.2.1 Environment Details . . . . . . . . . . . . . . . . . . . . . . . 22

3.3 Exponential Decaying Traces . . . . . . . . . . . . . . . . . . . . . . 25

3.3.1 Decaying Traces on the Lobster Environment . . . . . . . . . 25

3.3.2 Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27

3.3.3 Value Function Geometry of Trace Features . . . . . . . . . . 27

3.4 Approximate Belief State with Particle Filters . . . . . . . . . . . . . 29

3.4.1 Particle Filtering on the Lobster Environment . . . . . . . . . 31

3.4.2 Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32

3.5 Likelihoods for Incorporating Future Predictions and Past Information 33

3.5.1 Likelihoods as Predictions for the Lobster Environment . . . . 34
3.5.2 Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35

**4 Particle Filtering for Auxiliary Inputs 38**
4.1 Particle Filtering Environments . . . . . . . . . . . . . . . . . . . 38
4.1.1 Modified Compass World . . . . . . . . . . . . . . . . . . . . 38
4.1.2 RockSample . . . . . . . . . . . . . . . . . . . . . . . . . . . 40
4.2 Results and Discussion . . . . . . . . . . . . . . . . . . . . . . . . 41

**5 Scaling Up Auxiliary Inputs and Integration with RNNs 44**
5.1 The Fishing Environment . . . . . . . . . . . . . . . . . . . . . . . 46
5.2 Exponential Decaying Traces For Mapping . . . . . . . . . . . . . . 47
5.3 Results and Discussion . . . . . . . . . . . . . . . . . . . . . . . . 48

**6 Conclusion and Future Work 50**

**Appendix A: Lobster Environment Experimental Details 56**
A.1 Hyperparameters and Experimental Setup . . . . . . . . . . . . . . 56
A.2 Algorithmic details . . . . . . . . . . . . . . . . . . . . . . . . . . 57
A.2.1 Observations only . . . . . . . . . . . . . . . . . . . . . . . . 57
A.2.2 Value Iteration with Environment States . . . . . . . . . . . . 57
A.2.3 Exponential Decaying Trace . . . . . . . . . . . . . . . . . . . 57
A.2.4 Particle Filtering . . . . . . . . . . . . . . . . . . . . . . . . . 57
A.2.5 Likelihood Predictions . . . . . . . . . . . . . . . . . . . . . . 58

**Appendix B: Particle Filtering Experimental Details 59**
B.1 Environment Details . . . . . . . . . . . . . . . . . . . . . . . . . . 59
B.1.1 RockSample Environment Details . . . . . . . . . . . . . . . . 59
B.2 Environment-Specific Algorithmic Details and Hyperparameters . . . 61
B.2.1 Modified Compass World Experimental Setup and Hyperparameters . . . . . . . . . . . . . . . . . . . . . . . . . . . . 61
B.2.2 RockSample(7, 8) Experimental Setup and Hyperparameters . 62
B.3 Ablation Studies . . . . . . . . . . . . . . . . . . . . . . . . . . . . 64
B.3.1 RockSample(7, 8) Half Efficiency Distance Experiment . . . . 64
B.3.2 LSTM Action Concatenation Ablation . . . . . . . . . . . . . 65

**Appendix C: Fishing Experimental, Algorithmic and Environment Details 67**
C.1 Fishing Environment Details . . . . . . . . . . . . . . . . . . . . . 67

C.1.1 Mapping and Observations . . . . . . . . . . . . . . . . . . . . . 68
C.1.2 Stochasticity in the Environment . . . . . . . . . . . . . . . . . 68
C.2 Fishing-Specific Algorithmic Details and Hyperparameters . . . . . . 69
C.2.1 Convolutional Neural Network Architecture . . . . . . . . . . . 70
C.2.2 Convolutional Neural Network LSTM Architecture . . . . . . 71
C.2.3 Exponential Trace Implementation Details . . . . . . . . . . . 71
C.2.4 Recurrent Neural Network Implementation Details . . . . . . . 72

# List of Figures


3.1 The *Lobster* environment. . . . . . . . . . . . . . . . . . . . . . . . . . 23
3.2 The Lobster Environment MDP . . . . . . . . . . . . . . . . . . . . . . 23
3.3 Results for Exponential Decaying Traces in Agent States for the Lobster Environment . . . . . . . . . . . . . . . . . . . . . . . . . . 26
3.4 Results for Approximate Belief Distribution as Agent State in the Lobster Environment . . . . . . . . . . . . . . . . . . . . . . . . . . 32
3.5 Results for Likelihood as Agent State in the Lobster Environment . . 36
3.6 Combined Results in Lobster Environment . . . . . . . . . . . . . . . 37

4.1 Modified Compass World and RockSample(7, 8) . . . . . . . . . . . . 39
4.2 Results for Particle Filter Auxiliary Inputs in Modified Compass World and RockSample(7, 8) . . . . . . . . . . . . . . . . . . . . . . . . . 40

5.1 Fishing Environments and Results . . . . . . . . . . . . . . . . . . . . 45

B.1 Half Efficiency Distance Function Plots . . . . . . . . . . . . . . . . . 60
B.2 RockSample results for both $\delta_{hed} = 5, 20$ . . . . . . . . . . . . . . . . . 60
B.3 Action Conditioning Ablation for LSTM . . . . . . . . . . . . . . . . . 65

# Chapter 1

# Introduction

In reinforcement learning, an agent interacts with its environment in order to maximize a special numerical signal called the reward. The agent must make decisions based only on the information it observes from the environment, such as a first person camera view. This approach is quite general and has led to several high-profile success stories, such as agents capable of achieving impressive performance controlling fusion reactors [5], when playing curling [44], and when navigating balloons in the stratosphere [1]. An important feature of these problems is that the environment the agent is in—the real world—is much bigger than the agent itself. In this setting, the current observation from the data stream the agent experiences does not contain all relevant the information for the agent to act on, making the environment *partially observable*.

Partially observable problems pose a particular challenge in reinforcement learning because of the need for history summarization. All the information the agent could know about the state of its environment lies in the past interactions, or history, of the agent. This history grows as the agent continues to interact with the environment; as time progresses, this history becomes computationally unwieldy to process due to its growing size. To ameliorate this issue, the agent needs to select, at every time step, what information to retain that is useful to its task. Deciding on *what* to retain at every time step is at the crux of the history summarization problem, and is currently

an open area of research in reinforcement learning [37].

One general and scalable approach to history summarization is for the agent to construct an internal state of its previous interactions with the world. We refer to this internal state as the *agent state*. This has been key to the recent successes of reinforcement learning algorithms. The most famous example of this is most likely from the Deep Q-Network [21] algorithm, which achieved human-level performance in Atari 2600 games by explicitly expanding the input space by stacking 4 frames; which can be seen as a heuristic for capturing temporal information such as velocity and momentum of the video game sprites as an agent-state function. One way to *learn* agent-state functions is to leverage recurrent neural network architectures [9, 41, 5] to summarize an agent's history. These recurrent functions summarize history by using recurrent *latent* states (also called hidden states), and *learn* the function used to summarize history. Another approach that has been used in real-life use cases of reinforcement learning is to model predictive information about the agent's uncertainty over its effectiveness [44] or observations [1] to allow the agent to reason about what information the agent does *not* know to help resolve partial observability. Explicitly learning and leveraging predictions [28] is another approach that has been considered for history summarization. All these different examples for constructing agent-state functions have the same purpose: embedding necessary information from observations by expanding the feature space for a richer class of features with *auxiliary inputs* to ameliorate the issues of partial observability for better decision making.

Feature expansion has been considered in many different contexts, and has been widely used and investigated for neural networks over the years. Early adoptions of neural networks used random projections in the first layer of the neural network as "associator features" to map inputs to random binary features and expand the input space [4]. Expanding the input space to specifically tackle time-series data has been considered in the prediction context, where a convolution over the history of inputs [22] has been proposed as a formalism to memory for neural networks. In rein-

forcement learning, feature space construction and expansion was widely used before the advent of deep reinforcement learning. Techniques range from tile coding [37] to radial and fourier basis functions [37, 15]. More recently, function approximation with deep neural networks has allowed for more automated feature discovery with higher dimensional raw sensory inputs, such as pixels and depth maps, with minimal preprocessing.

In this work we investigate the idea of resolving partial observability through augmenting the inputs to a function approximator via joining feature expansion techniques with deep neural networks for reinforcement learning. We look to combine the simplicity and performance of simple feature expansion techniques with the natural adaptivity and flexibility of neural network function approximation.

## 1.1 Thesis Objectives and Contributions

In this thesis, we have three objectives: (1) to survey the feature engineering techniques used for resolving partial observability in successful real-world reinforcement learning applications, (2) to unify these approaches under a single formalization which we call *auxiliary inputs*, and (3) to demonstrate empirically, through a few illustrative examples, the efficacy of these approaches. In the context of agent-state construction for reinforcement learning in partially observable environments, auxiliary inputs are defined as *additional inputs, beyond environment observations, that incorporate or model information regarding the past, present and/or future of a reinforcement learning agent.*

Interesting to note is the fact that auxiliary inputs have been ubiquitous across recent, real-world applications of reinforcement learning. In robotics, uncertainty (or entropy) of the agent's location in the world [29] has been used as an auxiliary input feature to learn more accurate value functions. Recent work in stratospheric superpressure balloon navigation with deep reinforcement learning [1] has explored using not only the average magnitude and direction of the predicted wind columns

over time as input features, but also the variance of this wind column as an auxiliary input to the deep reinforcement learning agent for successfully navigating balloons in the stratosphere. In robotic curling, distance errors from previous throws and a trace of previous trajectories have been used as auxiliary inputs as features that help mitigate the partial observability induced by environment conditions such as changing ice sheets over time [44]. In biomedical applications, both a time-decayed trace of joint activity [24] and future predictions of prosthesis signals [23] were used as additional input features for controlling or aiding in the control of robotic prostheses.

In this thesis, we explore the idea that many approaches to partially observable problems can be viewed as a form of auxiliary input. The repeated success of agent-state construction with auxiliary inputs in complex domains motivates the guiding questions that we seek to answer throughout this thesis: (1) What shape can auxiliary inputs take that is general, effective, and easy to use to help in reinforcement learning? (2) Why do auxiliary inputs help? (3) How might auxiliary inputs affect and/or integrate with gradient-based agent-state construction methods? With these questions to guide us, we present the following contributions in this thesis:

1. **We formalize a general representation scheme for partially observable problems called *auxiliary inputs*.**

   We formalize auxiliary inputs as additional inputs to the agent-state function that incorporates/models information from the the past, present and/or future.

2. **We show examples of auxiliary inputs which add relevant information to input features that result in better learning.**

   We introduce and empirically test a few simple, fast, and general instantiations of auxiliary inputs (a decaying trace of observations, particle filters, and likelihoods as predictions). We show that these auxiliary inputs augment agent state by incorporating information relevant to the task at hand, and results in better learning in several problems by *expanding* the agent's state space to represent

more fine-grained control policies.

3. **We demonstrate that auxiliary inputs can improve recurrent neural network training.**

   Finally, we show that particular auxiliary inputs (specifically exponential decaying traces) can integrate well with recurrent neural networks trained with truncated backpropagation through time (T-BPTT), potentially allowing for a significant performance increase as compared to using only one or the other. We show this through experiments in two scaled-up visual domains.

To summarize, auxiliary inputs can be simple, performant, and should be the first approach most reinforcement learning practitioners take to tackling partial observability. In many cases, simple auxiliary inputs may be good enough if not better than more complex approaches such as recurrent function approximation, and can also be easily combined with these approaches for better performance in reinforcement learning.

# Chapter 2

# Background

In this chapter, we cover central concepts necessary for understanding this work. We also introduce the relevant terms, abbreviations, and notation in this section. We begin with introducing the formalism under which we study sequential decision making—Markov decision processes—in Section 2.1. Next, we define value functions in Section 2.2 as the function a decision-making agent is trying to optimize to achieve a task. To solve for these value functions, in Section 2.3 we consider the Bellman equations and the value iteration algorithm. Section 2.4 establishes temporal-difference methods for value function learning with experience, and introduces a control algorithm leveraging these methods—the Sarsa algorithm. To further extend these algorithms to larger state spaces, Section 2.5 introduces value function approximation to allow for generalization across states. The form of function approximation for value functions that we focus on in this thesis is deep neural networks, an approach which we describe in Section 2.6. In Section 2.7, we extend the Markov decision process formalism to the more general case of the partially observable setting. For decision making in this setting, we introduce agent-state functions in Section 2.8 to summarize the history of an agent. Finally, we describe recurrent neural networks and their use as agent-state functions in Section 2.9.

## 2.1 Sequential Decision Making and Markov Decision Processes

In this work, we consider the problem of decision making over time to achieve a goal. More specifically, we focus on the setting where decisions in previous time steps may affect future states of an environment that an agent interacts with. We use the reinforcement learning [37] problem setting and solution methods as the framework to formalize an agent learning from interactions with its environment to maximize a numerical signal called a *reward*, which characterizes a goal.

We model the agent's interaction with this environment as a Markov decision process (MDP). An MDP can be characterized by a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{R}, p, \gamma, S_0, S_T)$, where: $\mathcal{S}$ is the state space of the environment; $\mathcal{A}$ is the set of permissible actions; $\mathcal{R}$ is the set of rewards, which we specify as a subset of the real numbers in this work $\mathcal{R} \subseteq \mathbb{R}$. On each time step $t$, the agent takes an action $a_t \in \mathcal{A}$ in the environment. Partially in response to the agent's taken action, the environment transitions into a new state $s_{t+1} \in \mathcal{S}$ and receives a reward $r_{t+1} \in \mathcal{R}$, both according to the dynamics function $p : \mathcal{R} \times \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$. Periodically the environment enters a potentially random terminal state $S_T = \perp$ resetting the environment to a potentially random start state $S_0$. Note that random variables are denoted with capital letters in this work, so sampled states, actions and rewards are denoted as $S \in \mathcal{S}, A \in \mathcal{A}, R \in \mathcal{R}$. The agent's interaction is thus broken into a sequence of *episodes*.

States in this formalism have the *Markov property*, which specifies that the state includes all information pertaining to past agent-environment interactions that may affect future states. More formally, the Markov property states that transition probabilities are conditionally independent of all previous states and actions, given the current state and action.

The agent chooses actions to interact with the environment according to a policy $\pi : \mathcal{S} \times \mathcal{A} \to [0, 1]$, which maps a state to a probability distribution over actions. The

objective of a reinforcement learning agent is to maximize the future rewards of this episodic interaction with the environment by learning an optimal policy $\pi^*$.

## 2.2 Value Functions

The agent's primary goal is to learn a way of behaving that maximizes future reward. We specify these future rewards as the *value* of a state $s$ following a policy $\pi$, denoted as $v_\pi(s)$. This value of a state following a policy is the expected discounted sum of rewards following the policy $\pi$ starting at the state $s$:

$$v_\pi(s) \doteq \mathbb{E}_\pi \left[ \sum_{k=0}^{T-t-1} \gamma^k R_{t+k+1} \middle| S_t = s \right] \tag{2.1}$$

$$= \mathbb{E}_\pi \left[ G_t | S_t = s \right], \tag{2.2}$$

the return from time step $t$ is the discounted sum of rewards $G_t \doteq R_{t+1} + \gamma R_{t+2} + ... + \gamma^{T-1} R_T$, where $\gamma \in [0, 1)$ is the discount and $T$ is the (stochastic) time of termination. $G_t$ is also called the *return*. The expectation $\mathbb{E}_\pi$ is subscripted with the policy $\pi$ to indicate that the expectation is with respect to the policy $\pi$ (as well as the transition dynamics $p$, which we omit for clarity). We can also define the *action-value* function $q$, which is the expected discounted sum of rewards if the agent starts at state $s$ and takes action $a$, and follows the policy $\pi$ thereafter:

$$q_\pi(s, a) = \mathbb{E}_\pi \left[ G_t | S_t = s, A_t = a \right]. \tag{2.3}$$

Value functions define a partial ordering over policies. There are potentially many, but at least one, policy that is better than or equal to all other policies in their value functions, which we call an *optimal* policy:

$$\pi_* \doteq \arg\max_\pi v_\pi(s) \tag{2.4}$$

for all $s \in \mathcal{S}$. The value function of all optimal policies are called *optimal value*

*functions*, and *optimal action-value functions* if we also condition on taking an action:

$$\forall s \in \mathcal{S},\ v_*(s) \doteq \max_\pi v_\pi(s) \tag{2.5}$$

$$\forall s, a \in (\mathcal{S}, \mathcal{A}),\ q_*(s, a) \doteq \max_\pi q_\pi(s, a) \tag{2.6}$$

All policies which share the optimal value function (or action-value functions) are optimal policies. Hence the goal of an agent is to find an optimal policy $\pi_*$ that maximizes the return in expectation across start states $\mathbb{E}_\pi[G_0 \mid S_0]$. In the next section we introduce methods to learn optimal policies.

## 2.3 Bellman Equations and Value Iteration for Control

Now that we have defined the functions we are interested in for sequential decision making, we consider *how* to calculate these value functions. To do so, we introduce the Bellman equations, which are equations that define the value of a state (or state-action pair for action-values) in terms of the value of successor states:

$$v_\pi(s) = \sum_a \pi(a \mid s) \sum_{s', r} p(s', r \mid s, a) \left[ r + \gamma v_\pi(s') \right]. \tag{2.7}$$

If the function $p$ is completely known (e.g. as an $|\mathcal{S}| \times |\mathcal{S}|$ matrix for state transition probabilities), the equation becomes a set of $|\mathcal{S}|$ linear equations with $|\mathcal{S}|$ unknowns for the value function—a system of equations we can solve iteratively for what we call a *policy evaluation* algorithm. This class of algorithms which use full knowledge of the dynamics function are called *dynamic programming* (DP) methods. To do this, consider a sequence of approximate value functions $v_0, v_1, \ldots$. The algorithm begins with initial values $v_0$ (e.g. initialized with 0) for all states, and at every step performs the updates:

$$v_{k+1}(s) \leftarrow \sum_a \pi(a \mid s) \sum_{s', r} p(s', r \mid s, a) \left[ r + \gamma v_k(s') \right] \tag{2.8}$$

until convergence, where $v_k = v_{k+1}$, in which case $v_k = v_\pi$. Similarly for action-values, we have the Bellman equation:

$$q_\pi(s,a) = \sum_{s',r} p(s', r \mid s, a) \left[ r + \gamma \sum_{a'} \pi(a' \mid s') q_\pi(s', a') \right] \tag{2.9}$$

and corresponding update equation

$$q_{k+1}(s,a) \leftarrow \sum_{s',r} p(s', r \mid s, a) \left[ r + \gamma \sum_{a'} \pi(a' \mid s') q_k(s', a') \right]. \tag{2.10}$$

where updates are made over all state-action pairs for the action-value version of the policy evaluation algorithm. These update equations suggest a technique that is prevalent throughout reinforcement learning methods: using *bootstrapped* estimates as update targets. Bootstrapping in this context refers to the updating of one estimate with another estimate. For example, for value estimation in Equation 2.8, we update our state value estimates with the rewards and discounted next state value estimates. For the optimal value and action-value functions, we have the following Bellman optimality equations:

$$v_*(s) = \max_a \sum_{s',r} p(s', r \mid s, a) \left[ r + \gamma v_*(s') \right] \tag{2.11}$$

$$q_*(s,a) = \sum_{s',r} p(s', r \mid s, a) \left[ r + \gamma \max_{a'} q_*(s', a') \right]. \tag{2.12}$$

Equation 2.11 results in the following update rule for the *value iteration* algorithm, for estimating the optimal value function (and hence optimal policy for control):

$$v_{k+1}(s) \leftarrow \max_a \sum_{s',r} p(s', r \mid s, a) \left[ r + \gamma v_k(s') \right] \tag{2.13}$$

where you perform updates until $|v_{k+1} - v_k| < \Delta$, for some small $\Delta$. The value iteration algorithm is a combination of two phases characteristic of control algorithms based on value functions: policy evaluation and *policy improvement*. Policy evaluation takes a step towards a more accurate value function, whereas policy improvement greedifies the policy (in this case, with the max operator in Equation 2.11) to improve the policy towards the optimal policy.

These approaches all require a model of the environment to calculate these value functions and updates. We now consider how to learn an optimal policy with a *stream of experience* instead—without knowledge of the dynamics function.

## 2.4 Temporal-Difference Learning and Control with Sarsa

We leverage a technique central to reinforcement learning solution methods called temporal-difference (TD) learning [36] to learn from a stream of experience for control. TD methods directly learn from sampled raw experience, as opposed to DP methods which require full knowledge of the dynamics function.

We consider a TD method for learning action values for control—the Sarsa [31] algorithm. In this algorithm, an agent follows its policy $\pi$ to gather experience, with $\pi$ defined by the incrementally estimated action-value function at timestep $t$, $q_t$. In this work, our policy $\pi$ is defined by two cases: either we choose a greedy action with respect to $q_t$, or we choose a random action with probability $0 \leq \epsilon \leq 1$ to ensure sufficient exploration of the state space. This form of policy is called an *$\epsilon$-greedy* policy. After sampling and taking an action $A_t \sim \pi(\cdot \mid S_t)$, we receive the next reward and next state $R_{t+1}, S_{t+1} \sim p(\cdot, \cdot \mid S_t, A_t)$ and pick the next action $A_{t+1} \sim \pi(\cdot \mid S_{t+1})$. This gives us a tuple of sampled experience for time step $t$, $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$, which we use for our sample update. Learning an action-value function with TD learning is a sample update version of Equation 2.10—the Sarsa algorithm performs the following update on the action-value function at the current time step, which we denote as $q_{t+1}$ here:

$$q_{t+1}(S_t, A_t) \leftarrow q_t(S_t, A_t) + \alpha(R_{t+1} + \gamma q_t(S_{t+1}, A_{t+1}) - q_t(S_t, A_t)). \tag{2.14}$$

Since data sampled by the policy $\pi$ is being used to improve $\pi$ itself, we call this update an *on-policy* method. The alternative is an *off-policy* method, which improves a policy from data generated by a different policy to itself. The Sarsa algorithm collects a tuple

of experience every step based on an $\epsilon$-greedy policy defined above, and performs this online update. The policy improvement step is done through picking actions from our $\epsilon$-greedy policy, which results in an $\epsilon$-*soft* optimal policy.

## 2.5 Value Function Approximation

As state spaces become large, maintaining a value for every possible state in $\mathcal{S}$ becomes computationally intractable—methods that *generalize* between states are needed. All the methods defined in the previous section are *tabular* solution methods, where a value is associated with each individual state. To generalize across states, a value function needs to be able to assign similar values to states that are similar. To do so, we extend the above reinforcement learning methods to the *function approximation* setting, where we learn approximate solutions to our value functions and policies.

To approximate a value function, we can use a vector of weights $\boldsymbol{\theta} \in \mathbb{R}^d$ to parameterize a differentiable function of these weights. Note that vectors are denoted with bolded symbols in this manuscript. This function approximates the action-value function of some policy $\pi$: $\hat{q}_\pi(s, a, \boldsymbol{\theta}) \approx q_\pi(s, a)$. To approximate these action-value functions for control, one can use the sampled (with the same $\epsilon$-greedy policy as described in Section 2.4) tuples $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$ for the semi-gradient Sarsa update at timestep $t + 1$:

$$\boldsymbol{\theta}_{t+1} \leftarrow \boldsymbol{\theta}_t + \alpha \left[R_{t+1} + \gamma \hat{q}_\pi(S_{t+1}, A_{t+1}, \boldsymbol{\theta}_t) - \hat{q}_\pi(S_t, A_t, \boldsymbol{\theta}_t)\right] \nabla_{\boldsymbol{\theta}_t} \hat{q}_\pi(S_t, A_t, \boldsymbol{\theta}_t) \quad (2.15)$$

where $\alpha > 0$ is the step size, and $\nabla_{\boldsymbol{\theta}_t}$ is the gradient of the function $\hat{q}$ with respect to parameters $\boldsymbol{\theta}_t$.

A simple instantiation for the function $\hat{q}$ is to use a *linear* function. In this case, we assume our state-action pairs can be represented by a *feature vector*, which is represented by the function $\boldsymbol{x} : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}^d$, with $\boldsymbol{x}_t \doteq \boldsymbol{x}(S_t, A_t)$. A linear value function is thus defined as $\hat{q}(s, a, \boldsymbol{\theta}) \doteq \boldsymbol{\theta}^\intercal \boldsymbol{x}(s, a)$. The gradient of the linear function $\hat{q}$, $\nabla_{\boldsymbol{\theta}} \hat{q}$, is

simply the feature vector $\boldsymbol{x}(s, a)$. While this method is simple and well understood, the space of representable functions are limited to only linear functions of the feature vector. In the following section, we consider a class of function approximation that can potentially be much more expressive—*non-linear* function approximation—as well as techniques used for reinforcement learning with deep neural networks.

## 2.6 Deep Reinforcement Learning

Deep neural networks are a highly expressive class of function approximation that have shown impressive results as a value function approximator and for reinforcement learning as a whole. Its success in a large, pixel-based domain was first demonstrated in the Deep Q-Network algorithm [21], which trains a deep neural network for value function approximation with a similar semi-gradient TD method as to Equation 2.15, with gradients of the action-value approximator calculated through the back-propagation algorithm [30]. The algorithm also utilizes a few design choices to ensure stability throughout the training process. Two algorithmic choices considered in the work is the use of an *experience replay buffer* and a momentum-based optimizer.

Experience replay buffers [17] store previous transitions experienced by the agent to allow for more stable batch updates. Contrasting the online updates as described in Section 2.4, experience replay randomly samples batches of transitions from a buffer for an update that is somewhere in the spectrum of online and offline updates. Updating with a batch of randomly sampled transitions allows for updates that more closely resemble the canonic supervised learning setting—an *expected* update over a batch of data which are temporally decorrelated. Additionally, an expected update decreases the variance of the gradients, allowing for smoother gradients when updating parameters. Finally, experience replay also allows for better temporal credit assignment in environments with sparse rewards, as good transitions (e.g. transitions with larger TD-error) are repeatedly reused for updates, as compared to online updates, where these transitions are used once then “thrown away”.

Momentum-based optimizers have been widely used in deep reinforcement learning, and have been shown to improve performance in a wide range of settings. The Adam optimizer [13] is one momentum based optimizer that has seen wide adoption in deep reinforcement learning. Empirical results on a range deep reinforcement learning settings [8] have shown the benefits of the Adam optimizer, including better overall performance, a faster rate of learning, and less hyperparameter sensitivity.

## 2.7 Partial Observability

In most real-world decision-making scenarios, the environment the agent acts in is much larger than the agent itself. In these settings, it is the norm that the underlying state of the environment is unavailable to the agent. Instead, the agent receives *observations* to reason about. Extending the formalism introduced in Section 2.1, the agent instead interfaces with a partially observable Markov decision process (POMDP) [11], defined by the extended tuple $(\mathcal{S}, \mathcal{A}, \mathcal{R}, p, \gamma, S_0, S_T, o, \mathcal{O})$, where $\mathcal{O}$ is the space of observations and $o : \mathcal{S} \to \mathcal{O}$ is the observation function. In a more specified form, our observation space can be a vector space $\mathcal{O} \subseteq \mathbb{R}^n$, with the observation function producing a vector $\boldsymbol{o}(S_t) \doteq \boldsymbol{o}_t$. Note that in general, $|\mathcal{O}| << |\mathcal{S}|$. In this setting, the agent does not observe the state, only the current observation vector $\boldsymbol{o}_t$. Unlike environment states in MDPs, these observations no longer have the Markov property, which implies that to perform optimally under partial observability, the agent has to utilize its history of interactions in order to be performant.

In the POMDP setting, the policy is defined over the history of interactions, as the agent cannot observe the underlying state. Let $h_t \doteq \{\boldsymbol{o}_0, a_0, \boldsymbol{o}_1, ..., \boldsymbol{o}_t\} \in \mathcal{T}$ where $\mathcal{T}$ denotes the space of all possible trajectories of observations and actions of all possible lengths. The new goal is to find a policy conditioned on history, $\pi : \mathcal{T} \times \mathcal{A} \to [0, 1]$, that maximizes the return in expectation across start states $\mathbb{E}_\pi[G_0 \mid S_0]$.

## 2.8 Agent-State Functions

In many problems, learning policies over full histories is not tractable and the agent must make use of agent-state functions to summarize its history. The *agent-state* function maps a given $h_t$ to an agent state vector $\boldsymbol{x}_t \in \mathcal{X}$.

The agent-state is an attempt to generate a Markovian state of the environment, and it can be seen as a form of compression of the history of the environment [19]. The history of an agent represents all the possible information the agent could possibly know about the current state of the environment, with some potential amount of irreducible uncertainty with respect to the Markovian state of the environment. For each time step in an episode, while it might be possible to maintain a complete history of the episode and use this as input to our agent-state function, this approach is not practical since this history would grow linearly with time steps taken.

In the past, the traditional way to summarize history was through constructing a *belief* state [33]—a distribution over potential states given the current history. While this method has shown successes in the past and recent history [16], the underlying assumptions that define the approach lack scalability. Firstly, it requires some model of the world to update belief states from one time step to another. This model includes both (potentially approximations of) the transition dynamics and the probability of emitting an observation given a state. Secondly, most approaches require knowledge of the state space $\mathcal{S}$ in order to form a belief state over it.

The agent state can be more scalable than reasoning about the belief state, at the cost of approximating the Markov property. In most recent approaches to agent-state construction, including recurrent neural networks and general value function networks [32], the agent-state function has a recursive form:

$$\boldsymbol{x}_{t+1} \doteq u_\phi(\boldsymbol{x}_t, a_t, \boldsymbol{o}_{t+1}) \in \mathbb{R}^k, \tag{2.16}$$

where $\boldsymbol{\phi} \in \mathbb{R}^b$ are the parameters of the parametric agent-state function $u_{\boldsymbol{\phi}}$. At every time step, the agent takes an action, receives an observation, and incorporates this

observation into the current agent state, reducing the memory and time requirements for maintaining this history to a constant factor per time step. An agent-state function begins with some initial agent state $x_0$, defined depending on the agent-state function.

We can easily extend Sarsa to approximate the value function from agent state, by simply extending the definition of the feature vector function $\boldsymbol{x}$ from Section 2.5 to be the agent-state function in Equation 2.16. The action-value approximation then becomes: $\hat{q}(\boldsymbol{x}_t, a_t, \boldsymbol{\theta}_t) \doteq \boldsymbol{\theta}^T \boldsymbol{x}_t \approx q_\pi(s_t, a_t)$. Thus, one can view the estimated value as a linear function of the agent-state, which itself is a recursive, potentially non-linear, function of $\boldsymbol{x}_t, a_t$, and $\boldsymbol{o}_{t+1}$. To learn a control policy with an agent state, semi-gradient Sarsa simply adapts both $\boldsymbol{\theta}$ and $\boldsymbol{\phi}$ from the agent's interaction with the environment in order to improve reward maximization of the agent's policy, as before. The aim of this work is to investigate how including auxiliary inputs in agent-state construction (as input to $u$) can improve value-based reinforcement learning agents. In the next section, we consider one popular approach to learning this parametric agent-state function—recurrent neural networks.

## 2.9 Recurrent Neural Networks as Agent-State Functions

One popular instantiation of the agent-state function are *recurrent neural networks* [30] (RNNs). RNNs are parameterized recursive functions (which we denote as $u$ in this section) that maintain an internal *hidden state*, $\boldsymbol{x} \in \mathcal{X} \subseteq \mathbb{R}^k$, which is updated at every time step with some input; in reinforcement learning, these inputs are the observation and action. This hidden state update is the main mechanism in which RNNs process variable length sequences. At time step $t+1$ with parameters $\boldsymbol{\phi}$, we have the following hidden state update:

$$\boldsymbol{x}_{t+1} \doteq u_{\boldsymbol{\phi}}(\boldsymbol{x}_t, a_t, \boldsymbol{o}_{t+1}). \tag{2.17}$$

In typical RNNs, $u$ is non-linear and differentiable with respect to its parameters $\boldsymbol{\phi}$. For example, in a standard one-layer RNN [7], $u$ is defined by a linear combination of the weight matrices $\boldsymbol{\phi}$ with $\boldsymbol{x}_t$ and its inputs $\boldsymbol{o}_{t+1}$ and $a_t$, followed by a sigmoid activation to produce the next hidden state. Long short-term memory networks (LSTMs) [10] are another RNN architecture which leverage gating mechanisms and an additional cell state, which, put together, are meant to help with longer-term dependencies across sequences.

Comparing the update rule for RNNs in Equation 2.17 to the agent-state update in Equation 2.16, we see that RNNs are simply a *learnable* agent-state function. We now consider how to train and update the RNN parameters $\boldsymbol{\phi}$ for this agent-state function. To approximate value functions over agent state, we can define a vector of weights $\boldsymbol{\theta}$ as in Section 2.8, and estimate action-values $\hat{q}(\boldsymbol{x}_t, a_t, \boldsymbol{\theta}_t) \doteq \boldsymbol{\theta}^T \boldsymbol{x}_t \approx q_\pi(s_t, a_t)$, where $\boldsymbol{x}_t$ is the RNN hidden state/learnt agent state. At every update step, we update both $\boldsymbol{\phi}$ and $\boldsymbol{\theta}$ with one of the semi-gradient TD update rules defined previously in this chapter. To calculate gradients for the TD update, RNNs require us to calculate gradients over *time* with the backpropagation through time (BPTT) or the truncated backpropagation through time (T-BPTT) [42] algorithms. Since RNNs are recursive functions of the same parameters, the BPTT algorithm calculates gradients for the entire trajectory by "unrolling" the gradient computation backwards for each step that the function is applied.

One limitation to BPTT is that the computational cost of calculating these gradients scales linearly with the number of time steps in the trajectory. To alleviate this issue, T-BPTT truncates this backwards gradient calculation to a fixed number of steps; while this alleviates the computational costs of BPTT, it does so with a few trade-offs. The biggest trade-off being the fixed depth in which credit can be assigned, since gradients are only calculated and applied a fixed length back in time. Another trade-off with truncation is that it is an additional hyperparameter that has to be tuned depending on the nature of the problem setting.

# Chapter 3

# Auxiliary Inputs

In our work, we investigate the use of auxiliary inputs as an additional input into the agent-state function. In this chapter, we first formalize and define these auxiliary inputs as a convolution over the past and future trajectory of the agent. Next, we introduce the *Lobster* environment—a simple partially observable environment we use as a motivating example for auxiliary inputs. The environment represents a continuing foraging task, with both stochastically regenerating rewards and stochastic transitions. Afterwards, we introduce the three instantiations of auxiliary inputs that we consider in this work: exponential decaying traces of observations as a simple form of memory, approximate belief state with particle filtering, and likelihoods as a prediction of potential future observations.

Finally, we evaluate these three approaches as auxiliary inputs to our agent-state function on the Lobster environment. We first show that these three methods are all able to outperform an agent only using current observations at each step; we also show that each approach exhibits a similar average rate of return in this environment. Next, we visualize the learnt policy of the agent to show *how* these auxiliary inputs are helping with decision making in this partially observable environment—we show that these auxiliary inputs allow an agent to smoothly interpolate between the values of the ground-truth states, and allows the agent to discriminate between states that would otherwise be aliased.

## 3.1 Formalism and Examples

To formalize these auxiliary inputs, we adapt and generalize the formalism of defining memory for neural network inputs as a convolution over the history of inputs [22]. As opposed to only summarizing the past history of experiences, we define auxiliary inputs to incorporate information from/model the past, present and/or future. To do this, auxiliary inputs must summarize *trajectories* for decision making, rather than only the agent's history. We denote the trajectory of an agent at time $t$ as $\mathrm{T}_t \doteq \{\boldsymbol{o}_0, a_0, \boldsymbol{o}_1, ..., \boldsymbol{o}_t, a_t \ldots, \boldsymbol{O}_T\} \in \mathcal{T}$, where $T$ denotes the terminal time step of the trajectory. At the time step $t$, an agent will have only a partially realized trajectory, $\{\boldsymbol{o}_0, a_0, ..., \boldsymbol{o}_t, a_t, \boldsymbol{O}_{t+1}, A_{t+1} \ldots, \boldsymbol{O}_T\}$, where observations and actions including and before $t$ are actualized variables (denoted with lower case letters), whereas all future actions and observations from $t+1$ to $T$ are still random variables.

Let $\boldsymbol{m} : \mathcal{T} \to \mathbb{R}^m$ denote an auxiliary input as a function of the trajectory of the agent, which maps the trajectory $\mathrm{T}_t$ to a vector. In the more general form, we can have multiple functions $\boldsymbol{m}^i, i \in \{1, \ldots, N\}$ which correspond to each of our auxiliary inputs over our history. For the $i$th auxiliary input at time $t$, we denote this as $\boldsymbol{m}^i(\mathrm{T}_t) \doteq \boldsymbol{m}^i_t$. The set of $N$ auxiliary inputs at time $t$ is then written as the tuple $\boldsymbol{M}_t \doteq (\boldsymbol{m}^1_t, \ldots, \boldsymbol{m}^N_t)$. With these additional auxiliary inputs, we re-define our agent-state function to include these auxiliary inputs:

$$\boldsymbol{x}_{t+1} \doteq u_\phi(\boldsymbol{x}_t, a_t, \boldsymbol{o}_{t+1}, \boldsymbol{M}_{t+1}) \in \mathbb{R}^k. \tag{3.1}$$

We further specify the function $\boldsymbol{m}$ to allow us to formalize the different approaches to auxiliary inputs in our work. Without loss of generality, we modify our definition of trajectory to be a sequence of $(\boldsymbol{O}, A) \in \mathcal{O} \times \mathcal{A}$ observation-action tuples (we can simply append the empty set $A_T \doteq \emptyset$ to the final observation): $\mathrm{T}_t \doteq \{(\boldsymbol{o}_0, a_0), (\boldsymbol{o}_1, a_1), ..., (\boldsymbol{o}_t, a_t), ..., (\boldsymbol{O}_T, A_T)\} \in \mathcal{T}$. With this, our $i$th auxiliary input at time $t$, $\boldsymbol{m}^i_t$ can be seen as the function $a$ of a convolution over the trajectory

of processed observation-action pairs:

$$\boldsymbol{m}_t^i \doteq f\left(\sum_{\tau=0}^{T} k_i(\tau) g_i(\boldsymbol{O}_\tau, A_\tau)\right) \tag{3.2}$$

where $k_i$ is the $i$th kernel function $k : \mathbb{N} \to \mathbb{R}$ of the convolution, and $g$ is the preprocessing function applied to the observation-action pair before convolving with the rest of the trajectory. One function $f$ that particularly focus on in this work is the *expectation* function $\mathbb{E}$ over a given probability distribution:

$$\boldsymbol{m}_t^i \doteq \mathbb{E}\left[\sum_{\tau=0}^{T} k_i(\tau) g_i(\boldsymbol{O}_\tau, A_\tau)\right] \tag{3.3}$$

$$= \sum_{\tau=0}^{t} k_i(\tau) g_i(\boldsymbol{o}_\tau, a_\tau) + \mathbb{E}\left[\sum_{\tau=t+1}^{T} k_i(\tau) g_i(\boldsymbol{O}_\tau, A_\tau)\right]. \tag{3.4}$$

We explicitly consider expectations here due to the unobserved random variables (future observations, actions, and terminal time step) past the current time step. This expectation will be with respect to the sampling distribution of these trajectories. Note that taking an expectation over the actualized variables of past observations simply returns the actualized variables themselves. Equation (3.4) makes this distinction between the actualized, observed time steps, and the random variables of the trajectory clear: from 0 to $t$ our auxiliary input function maps over observed, actualized variables, and from $t+1$ to $T$ we take the expectation with respect to the sampling distribution over this convolution over all potential future observations and actions.

Many auxiliary inputs can be defined by the function $\boldsymbol{m}$. To clarify how this formulation might be used, we show how the frame stacking [21] technique, widely used in the Arcade Learning Environment [2, 20], fits into this formalism.

### 3.1.1 Frame Stacking

As an auxiliary input, the number of previous frames to stack corresponds to the number of auxiliary inputs ($N = 3$, as the current frame is accounted for). Our

preprocessing function $g$ for frame stacking is simply the function that just returns the observation: $g(\boldsymbol{o}_t, a_t) \doteq \boldsymbol{o}_t$. The kernel function for $i \in \{1, \dots, 3\}$ is defined as

$$k_i(\tau) \doteq \mathbb{1}_{[\tau = t-i]}$$

where $\mathbb{1}_{[cond]}$ is the indicator function, which is 1 if *cond* is true, 0 otherwise. With these definitions in place, Equation 3.3 defines 3 auxiliary inputs at time $t$, $\boldsymbol{M}_t \doteq (\boldsymbol{m}_t^1, \dots, \boldsymbol{m}_t^3)$. Furthermore, based on the convolution of this time-indicator function, each of these inputs correspond to the observation $\boldsymbol{o}_{t-i}$—or the previous 3 observations seen by the agent. $\boldsymbol{M}_t$ defines the stack of the last 3 observations, which (in addition to the current observation) is frame stacking.

### 3.1.2 Resolution and Depth

Viewing frame stacking as a form of auxiliary inputs elucidates an interesting property of our formalization: auxiliary inputs defined by Equation 3.3 represent differing *depths* and *resolutions* of the information you retain with regards to your trajectory. We define depth to be at what temporal length these auxiliary inputs retain information with regards to the agent's trajectory, and resolution to be the extent that information regarding individual observation-action pairs are preserved by the auxiliary inputs. Frame stacking is a form of auxiliary inputs that is low in depth (as we only see 3 time steps before the current time step), but high in resolution (since we retain all the relevant information of these 3 observations).

### 3.1.3 Incremental Functions

One important factor to note is the space and time complexity of calculating these auxiliary inputs. While we define these auxiliary inputs to be a function of history, we focus on algorithms that are incremental update functions for producing auxiliary inputs: $\boldsymbol{M}_t \doteq h(\boldsymbol{M}_{t-1}, \boldsymbol{o}_t, a_t)$.

With auxiliary inputs formalized, we now consider a simple partially observable

environment to help explain *why* these auxiliary inputs might help with reinforcement learning.

## 3.2 The Lobster Environment

We introduce a small motivating example, the *Lobster* environment, which we use to provide an intuition on how auxiliary inputs lead to a more expressive agent state. In this chapter, we aim to show that these auxiliary inputs for agent state allows the agent to discriminate between observations that would otherwise be aliased, and allows for the smooth interpolation in the value function between different states in the environment, implicitly modelling their underlying state. With this environment, we aim to show that these different forms of auxiliary inputs all help resolve partial observability for better decision making. We begin by detailing the partially observable environment we test our approaches on.

### 3.2.1 Environment Details

In the Lobster environment, shown in Figure 3.1, a fishing boat has to travel between 3 locations—represented as nodes in the graph—to collect lobsters from lobster pots. Only locations L1 and L2 have lobster pots, which refill randomly over time after being collected. The environment starts with both pots filled. The boat/agent can only see whether or not a pot is filled if the agent is in the corresponding location. Notice that, because of the partial observability regarding the visibility of the lobster pots, the Lobster environment is not a 3-state Markov decision process, and the nodes in Figure 3.1 do not directly represent states in the environment. The full MDP with corresponding states is pictured in Figure 3.2. An agent in this environment has 3 actions: $\mathcal{A} \doteq \{\texttt{left}, \texttt{right}, \texttt{collect}\}$.

As this environment is partially observable, we now detail the observation vector $\boldsymbol{o}_t \in \{0,1\}^9$ the agent receives at every time step. We list out 9 ordered true or false questions which correspond to the elements (either 0 or 1 respectively) in the

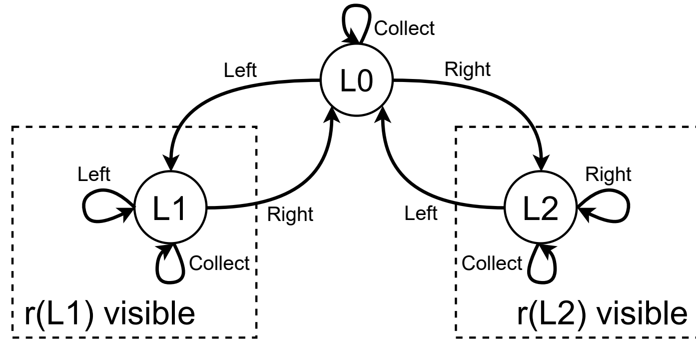



Figure 3.1: **The *Lobster* environment.**

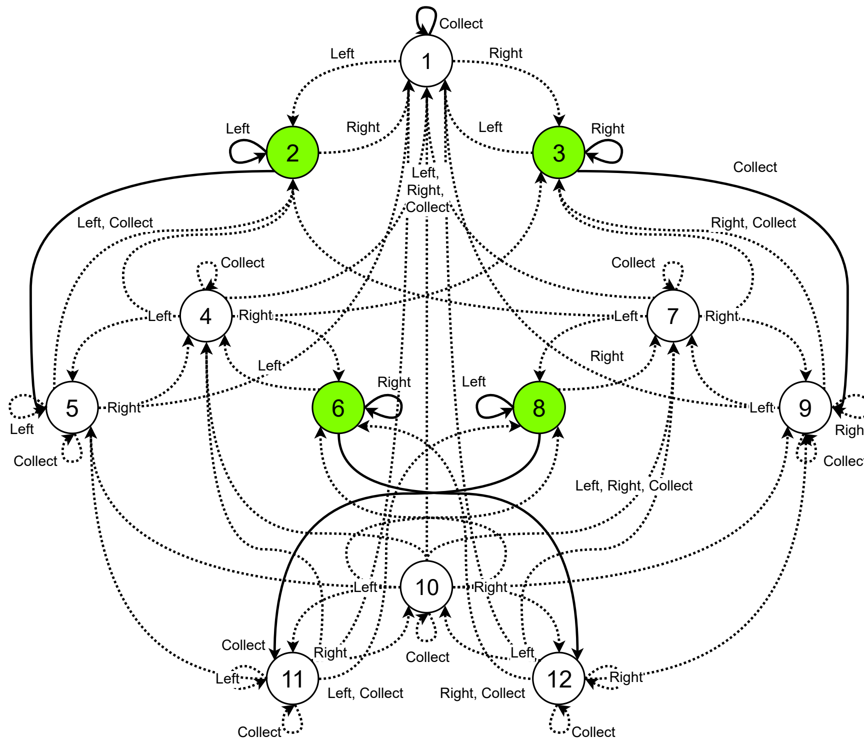



Figure 3.2: **The Lobster Environment MDP**
"Slippery" transitions at certain states are not pictured. Green nodes represent states where performing the `collect` action will yield a reward. Particular to only this figure, solid lines are deterministic transitions, whereas dotted lines are stochastic transitions.

observation vector:

$$
\boldsymbol{o}_t \doteq \begin{bmatrix} \text{0. Is the agent in location 0?} \\ \text{1. Is the agent in location 1?} \\ \text{2. Is the agent in location 2?} \\ \text{3. Is the reward in location 1 observable and missing?} \\ \text{4. Is the reward in location 1 observable and present?} \\ \text{5. Is the reward in location 1 unobservable?} \\ \text{6. Is the reward in location 2 observable and missing?} \\ \text{7. Is the reward in location 2 observable and present?} \\ \text{8. Is the reward in location 2 unobservable?} \end{bmatrix} \tag{3.5}
$$

We now detail the sources of stochasticity in the Lobster environment. Actions that try to transition between locations in the Lobster environment succeed with probability $p_{slip} = 0.6$; if the transition fails, the agent "slips" and stays in the same location. At every time step, if a reward is not present, it regenerates according to its own Poisson processes, with an expected number of steps for regeneration of $\Lambda \doteq 10$ for each reward. Further details of this environment are elucidated in Appendix A.

With this formalism and environment in place, we describe three auxiliary input techniques with the formalism introduced in Section 3.1 that incorporate or model information from the past, present and/or the future. We consider these three techniques throughout our work. To demonstrate that these auxiliary inputs help with the task in the Lobster environment, we compare the performance of each form of auxiliary input to two agents: one using only the observations described in Equation 3.5, another using the fully observable environment state. We use these two agents as baselines as a gauge to see how much our auxiliary inputs are able to help resolve partial observability. Both of these baselines are fully specified in Appendix A.2

All experimental results show (Figs. 3.3a, 3.4a and 3.5a) the mean (solid line) and standard error to the mean (shaded region) over 30 seeds, with hyperparameters swept and chosen. We defer a detailed description of the hyperparameters swept and

experimental setup to Appendix A.

## 3.3 Exponential Decaying Traces

To incorporate information from the past of the agent, we consider exponential decaying traces of history as auxiliary inputs to agent-state. Decaying traces simply keep an exponentially decaying weighted sum of our observations and actions:

$$\boldsymbol{m}_t^i \doteq \sum_{\tau=0}^{t} \lambda^{t-\tau} g_i(\boldsymbol{o}_\tau, a_\tau) \tag{3.6}$$

with $\lambda < 1$. The kernel function $k_i(\tau)$ in this case is a simple exponential function over past time steps:

$$k_i(\tau) \doteq \begin{cases} \lambda^{t-\tau} & \text{if } \tau \leq t, \\ 0 & \text{otherwise.} \end{cases} \tag{3.7}$$

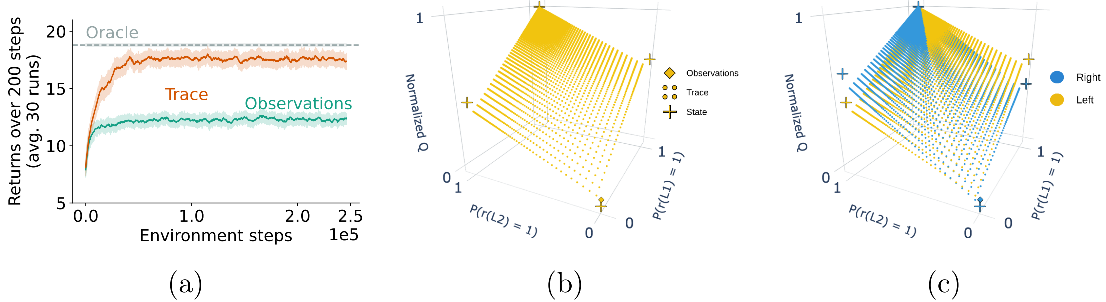
Our decaying trace auxiliary inputs are simply convolutions over time with this kernel function. Written in an incremental form, we have $\boldsymbol{m}_t^i \doteq \lambda \boldsymbol{m}_{t-1}^i + g_i(\boldsymbol{o}_\tau, a_\tau)$.

This form of auxiliary input acts as a model of the past by acting as an exponential timer for events in the observation. When used as an auxiliary input to the agent-state function, an exponential decaying trace of observations allows the agent to take into consideration the time in which events occur in the observation vector $\boldsymbol{o}$ or action $a$. This particular form of history summarization is high in depth, but low in resolution, since $g_i(\boldsymbol{o}_\tau, a_\tau)$ is aggregated together across time steps.

### 3.3.1 Decaying Traces on the Lobster Environment

To use decaying traces as an auxiliary input for the Lobster environment, we take decaying traces of the elements in the observation vector that indicates whether or not each reward is collected. With the indexing and observations from Equation 3.5, our decaying trace for the lobster environment is defined as:

$$\boldsymbol{m}_{t+1}^i \doteq \begin{cases} \lambda \boldsymbol{m}_{t-1}^i & \text{if reward } i \text{ is unobservable} \\ \boldsymbol{o}_t[3i] & \text{otherwise} \end{cases} \tag{3.8}$$




(a) (b) (c)

Figure 3.3: **Results for Exponential Decaying Traces in Agent States for the Lobster Environment**
(a): Online returns in the Lobster environment for our trace agent, in comparison to the policy given ground-truth state (dotted line). See text for details. (b): Normalized action-values for an agent with decaying trace auxiliary inputs in location 0 for the `left` action. The axis denote the probability of rewards being present in locations L1 and L2. The combination of which represents a possible input for the exponential decaying trace agent state. The values for the ground-truth state (crosses) and for using only the observation (diamond) are overlaid for comparison. (c): Action-values for both `left` (yellow) and `right` (blue) actions. Note: Since our exponential decaying traces represents time *since* the reward was observed as missing, we plot $1 - \boldsymbol{m}^i$ since $1 - \boldsymbol{m}^i \propto P(r(Li) = 1)$. Note that in this plot, the observation-only action values for both the left and right actions overlap, where only the `right` action value is visible.

.

for each auxiliary input $i \in \{1, 2\}$ that corresponds to rewards in locations 1 and 2, respectively. Note that each auxiliary input here is a vector of size $\mathbb{R}^1$. As a reminder, $\boldsymbol{o}_t[3i]$ corresponds to the boolean that answers the question "Is the reward in location $i$ observable and missing?" Within our Lobster environment experiments, we use a decay rate of $\lambda = 0.9$.

Put together, our auxiliary inputs are defined as $\boldsymbol{M}_t \doteq (\boldsymbol{m}_t^1, \boldsymbol{m}_t^2)$, which we use as part of our agent-state function. Our agent state for trace decay auxiliary inputs in the Lobster environment is the concatenation of the observation and the two auxiliary inputs in $\boldsymbol{M}_t$: $\boldsymbol{x}_t \doteq [\boldsymbol{o}_t, \boldsymbol{M}_t] \in \mathbb{R}^{11}$. The dimension of this agent state is the number of dimensions of the observation, plus two dimensions from the two auxiliary inputs $9 + 2 = 11$.

### 3.3.2 Results

We now consider the results from using exponential decaying trace auxiliary inputs in the Lobster environment. For the agents described in this section, we use the Sarsa [31] algorithm to learn a control policy (with the exception of the ground-truth state agent, which uses value iteration to learn the optimal policy). We also use linear function approximation for all agents. For this trace decay agent, a step size of $\alpha \doteq 10^{-3}$ was selected from a hyperparameter sweep, with an epsilon of $\epsilon \doteq 0.1$ for the epsilon-greedy policy. All hyperparameters swept and algorithmic details are fully described in Appendix A.

The performance of the agent with exponential decaying trace as auxiliary inputs is summarized in Figure 3.3a. The policy learned with exponential decaying converges to a higher return than the agent using only observations. We can see this in the learned policies of our trace agents versus the observations only agent: when using only observations, the agent dithers between location L0 and location L1 *or* L2, but not both. On the other hand, the agent that leverages an additional trace as auxiliary input collects the reward from one of the rewarding locations, and then traverses to the other rewarding location depending on the value of the traces.

### 3.3.3 Value Function Geometry of Trace Features

To get a better idea of *how* exponential decaying traces augments our policy, we consider the value function learnt over these auxiliary inputs. Specifically, since our value function is approximated with linear function approximation, we visualize and compare how the auxiliary inputs of exponential decaying traces affect the value function geometry of our learnt policy in Figs. 3.3b and 3.3c.

We compare action-values when the agent is at location L0 between the three algorithms plotted in Figures 3.3b and 3.3c: the agent using observations only (the single diamond-shaped point), the agent using ground-truth environment states (the crosses), and finally the agent with exponential decaying traces as auxiliary inputs

(the small circular points). The $x - y$ axes (bottom two axes) captures two input features that represent the likelihood of each reward being present. For the ground-truth environment states, these features simply represent whether or not the rewards are present or not. Since we have two rewards in locations L1 and L2, this corresponds to four possible environment states at location L0—one for each possible state of the two rewards (since they can be either present or not present). For the observation-only agent, both of these corresponding features can only take on a single value due to partial observability: both features are 0 since at location L0, this agent will only ever see 0 elements for the features which correspond to whether or not the rewards in each location are there. We now describe how we plot the exponential decaying trace features.

From Equation 3.8, our exponential decaying traces *decrease* the more time elapsed since last observing each reward was missing. This means that our trace features for each reward should be inversely proportional to the likelihood of each reward being present. Given this, we plot the complement $(1 - \boldsymbol{m}^i)$ of each exponential decaying trace input in Figures 3.3b and 3.3c.

We first consider Fig. 3.3b; in this plot we calculate and visualize all possible input features mentioned above for the three algorithms over the normalized action-values of the `left` action, given the agent is at location 0. We can see that augmenting the agent state with auxiliary inputs *expands* the state space of the agent over two dimensions of time: one dimension for each reward observation that we have our exponential decaying trace over. The four vertices of this expanded state space represent and coincide with the four ground-truth states when the agent is at location 0.

Learning a value function with these exponential decaying traces allow the agent to smoothly interpolate between the values of the actual ground-truth states. The agent does this through resolving (checking out a location) or accumulating (waiting in another location) the decaying trace observations. Learning a value function with such agent states essentially allows the agent to disentangle and discriminate states

that would otherwise be mapped to the same observation. This expansion in the state space allows for more expressivity in the value function.

This additional expressivity is also reflected in the policy. This is depicted in Figure 3.3c; in addition to the action-value function for the `left` action, we also visualize the action-value function for the `right` action of the agent with exponential decaying traces. By overlaying the action-values of both actions we can see how the decaying trace agent, as well as the agent using the ground-truth states, learn action-values that in some corners are greater for the `right` action and in other corners are greater for the `left` action, actually leading to a sensible greedy policy. Alternatively, the agent that uses only the environment observations has no choice but to collapse the action-values of both actions into the same value.

## 3.4 Approximate Belief State with Particle Filters

We now consider a classic approach to auxiliary inputs for resolving partial observability: constructing approximate belief states. A common use of auxiliary inputs in reinforcement learning is to leverage *uncertainty* information with regards to the underlying state of the environment. In deep reinforcement learning for stratospheric superpressure balloon navigation [2], uncertainty over the current wind column is used as auxiliary inputs to take into account the non-stationarity of wind directions and magnitudes. Prediction error as auxiliary inputs are used in robotic curling [44] to account for the shifting ice over a game of curling. In this section, we investigate the use of uncertainty as auxiliary inputs—approximating a distribution over all states at every time step. In order to get this measure of uncertainty of the environment state, we leverage a few assumptions with regards to available transition dynamics and known state structure for the particle filtering approach for approximating a distribution over possible states, or a belief state [11].

To construct these belief states, we consider a Monte Carlo based approach with particle filtering [14] to approximate a distribution over state [39, 25] as auxiliary

inputs for agent state. With this approach, we maintain an approximate distribution over possible states by incorporating the current observation and action through updating weights and particles with the emission probabilities and dynamics function respectively. We begin with $m$ particles, which we denote by a vector of particle states $\hat{\boldsymbol{s}}_0 \in \{1, \ldots, |\mathcal{S}|\}^m$, initialized according to the start state distribution of the environment, and instantiate a vector of weights $\boldsymbol{w}_0 \in \mathbb{R}^m$. At every step $t+1$, and for every particle $\forall j \in \{1, \ldots, m\}$, we update the particles and weights by first propagating all particles forward with the action taken with the dynamics function:

$$\hat{\boldsymbol{s}}_{t+1}[j] \sim p(\cdot \mid \hat{\boldsymbol{s}}_t[j], A_t), \tag{3.9}$$

and updating each particle's weight according to the probability of emitting the observation (emission probability) received:

$$\overline{\boldsymbol{w}}_{t+1}[j] \doteq P\{\boldsymbol{O}_t = \boldsymbol{o}_t \mid S_t = \hat{\boldsymbol{s}}_t[j]\} \cdot \boldsymbol{w}_t[j]. \tag{3.10}$$

This produces the unnormalized weights of all particles. We get our new set of weights by simply normalizing: $\boldsymbol{w}_{t+1} \doteq \frac{\overline{\boldsymbol{w}}_{t+1}}{\sum_{i=1}^{m} \overline{\boldsymbol{w}}_{t+1}}$. These weights are essentially the mechanism in which we summarize our past trajectory.

With this mechanism to update particles and weights, we form our auxiliary inputs based on these weights. In this approach, we only have a single auxiliary input vector where $N = 1$, which we simply denote as $\boldsymbol{m}_t$.

These auxiliary inputs calculate our approximate distribution of states at time step $t+1$, $\boldsymbol{m}_{t+1}$, by summing over weights for each particle for a given state:

$$\boldsymbol{m}_{t+1} \doteq \sum_{\tau=t}^{t} \sum_{j=0}^{m} \boldsymbol{w}_{t+1}[j] \odot \mathbb{1}_{[\hat{s}_t[j]]} \tag{3.11}$$

Where the bolded $\mathbb{1}_{[s]}$ corresponds to the one-hot encoding of length $|\mathcal{S}|$, with a 1 at state $s$. In this case, our auxiliary inputs $\boldsymbol{m}_{t+1}$ are our agent-state function, and we have $\boldsymbol{x}_{t+1} \doteq \boldsymbol{m}_{t+1}$.

At every step, the auxiliary input function $\boldsymbol{m}$ is defined by Equation 3.11. Our current action and observation are incorporated into our auxiliary input by both the

propagation of particles forward given an action and the re-weighting of our particles through the emission probability of the current observation given the particle. Our kernel function is defined as:

$$k(\tau) \doteq \begin{cases} 1 & \text{if } \tau = t, \\ 0 & \text{otherwise.} \end{cases} \tag{3.12}$$

The preprocessing function $g$ is defined by $g(\boldsymbol{o}_t, a_t) \doteq \sum_{j=0}^{m} \boldsymbol{w}_{t+1}[j] \odot \mathbb{1}_{[\hat{s}_t[j]]}$. The actions $a_t$ are incorporated into the particle updates for $\hat{s}_t[j]$ and observations $\boldsymbol{o}_t$ are incorporated into the weight updates for $\boldsymbol{w}_{t+1}$. Put together, $\boldsymbol{m}_t$ resolves partial observability through the counter-factual updates of the particles and their corresponding weights at the present time step with the current observation and action.

This particle filtering for approximate belief state is an approach to auxiliary inputs that is low in resolution and can be high in depth, depending on if the number of particles is adequate. This approach has low resolution because individual observations are incorporated into the approximate belief state through its emission probabilities. Approximate belief states summarize all observations through its re-weighting of particle weights, and is unable to discern between individual observations its seen in its trajectory. On the other hand, if there are an adequate number of particles for an environment, this approach can have high depth, seeing as belief states summarize all the relevant information with regards to environment state and transition dynamics [12].

### 3.4.1 Particle Filtering on the Lobster Environment

For this particle filtering approach in the Lobster environment, we approximate a distribution over the 12 underlying ground-truth states (as per Figure 3.2), based on the approximated distribution over state over $n_{particles} = 100$ particles. At every step, we follow the steps in Section 3.4 to get our approximate belief state as our auxiliary input. In this case, since we only have a single auxiliary input vector, we have that $\boldsymbol{M}_t \doteq \boldsymbol{m}_t$. In addition to this, our auxiliary inputs in this case define the entire agent

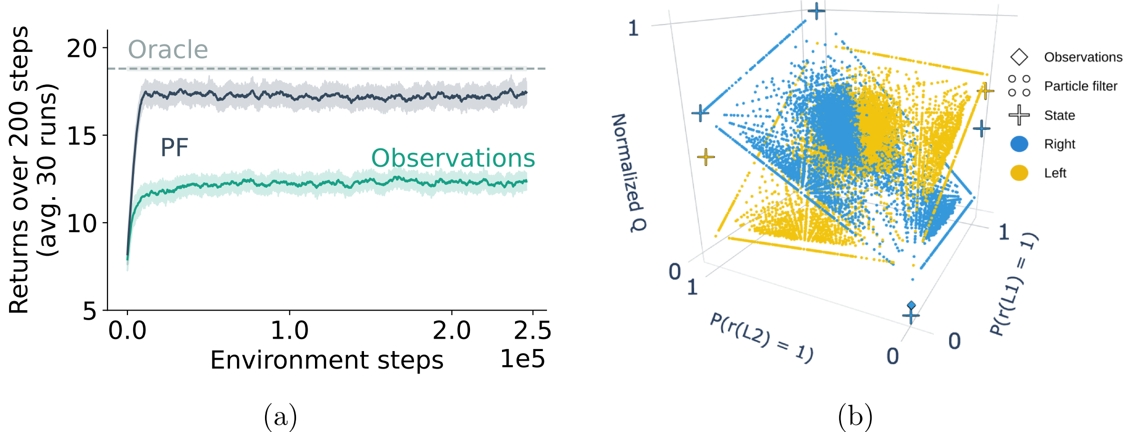



Figure 3.4: **Results for Approximate Belief Distribution as Agent State in the Lobster Environment**
(a): Online returns in the Lobster environment for our particle filtering agent, in comparison to the optimal policy given ground-truth state (dotted line). (b): The action-values for both `left` (yellow) and `right` (blue) actions for the value function learnt over particle filtering auxiliary inputs. Again, the observation-only agent is represented by overlapping diamonds in this plot.

.

state, since we already incorporate both observations and actions at each step in the particle filter update: $\boldsymbol{x}_t \doteq \boldsymbol{M}_t$.

### 3.4.2 Results

For this particle filtering agent, once again we leverage the Sarsa control algorithm as well as a selected step size of $\alpha \doteq 10^{-3}$ (from a hyperparameter sweep) and an epsilon of $\epsilon \doteq 0.1$. Similar to the results from the exponential decaying traces in Section 3.3.2, the particle filtering agent also converges to a higher rate of return as compared to the agent using observations only, and performs worse than if the agent was given full state information.

We also show similar value function geometry plots to Fig. 3.3c, except for the belief distribution features in Fig. 3.4b. In this case, the bottom two $x - y$ axes correspond to, for each reward $i \in \{1, 2\}$:

$$P(r(Li) = 1) \doteq \sum_{s \in \mathcal{S}} \boldsymbol{m} \odot \mathbb{1}_{[s \text{ where reward } r(Li) \; = \; 1 \; \& \; s \text{ where the agent is in location } 0]}$$

which is the sum of the probabilities over environment states where the agent is in location 0 and the reward in location $i$ is present. In this particular visualization, we visualize all the features collected from multiple rollouts of the policy learnt by the particle filtering agent. Given these features, we see a similar policy at location 0 to the agent utilizing exponential decaying traces as auxiliary inputs, where the greedy policy will go left if the likelihood of $r(L1)$ being present is higher than the likelihood of $r(L2)$ being present, and right if the converse is true. Approximate belief states with particle filtering are also able to resolve partial observability with a similar performance to exponential decaying traces. Finally, we consider our last form of auxiliary inputs—using likelihoods as future predictions.

## 3.5 Likelihoods for Incorporating Future Predictions and Past Information

We now consider how to incorporate future predictions together with past information for auxiliary inputs. In the reinforcement learning setting, one popular choice of predictions are general value functions [35]. These predictions take on the form of a discounted sum over *cumulants* (which can also be a function of observations and actions like our preprocessing function $g_i$), with some separate cumulant termination function (which could be represented with our kernel function $k_i$). While general value functions fit into our formalism, learning and leveraging these predictions as auxiliary inputs in partially observable settings has historically proven to be challenging. The difficulty arises in the fact that, as an auxiliary input, the prediction at time step $t$ depends on the previous prediction at $t-1$ as well as the current observation and the previous action. While there have been (even real-world) examples of general value functions as auxiliary inputs for control [43], learning predictions from previous predictions, while learning a value function for control over these features is a difficult task; it is also not clear that this approach is helpful for settings beyond a small set of partially observable environments paired with specific general value functions [32].

In this section, we consider simpler predictions as a proof-of-concept for using future predictions together with information from the past as auxiliary inputs—ground-truth likelihoods.

### 3.5.1 Likelihoods as Predictions for the Lobster Environment

We describe auxiliary inputs that incorporate both future predictions and past information for the Lobster environment. Let $i \in \{1, 2\}$ be the indices for both our two auxiliary inputs and our rewarding locations. These auxiliary inputs predict whether or not a reward at a given location L$i$ will be present if the agent takes the *expected* number of steps to that location. These auxiliary inputs essentially amount to answering the question: "Given that I saw r(L$i$) missing some steps ago, if I take the mean number of steps to reach L$i$, what is the likelihood that r(L$i$) will be regenerated?"

We now describe how we model this future prediction as a likelihood. We do this through calculating, in closed form, the likelihood that r(L$i$) is present, given that some number of steps have elapsed with r(L$i$) as unobservable. In this auxiliary input scheme, those number of steps will depend on the number of steps since the agent has seen r(L$i$) missing *and* the average number of steps needed to reach L$i$. In this approach, we assume we have the privileged information of the rate $r$ at which either rewards are regenerated. Let $E^i_\tau$ be the event that the reward at location L$i$ is regenerated within $\tau$ steps, and $E^{i'}_\tau$ be the complementary event, where the reward does not regenerate after $\tau$ steps. Since this is a Poisson process, this means that the likelihood of a reward at location L$i$ regenerating after $\tau$ steps is:

$$P(E^i_\tau) = 1 - P(E^{i'}_\tau) = 1 - \exp\{\tau \cdot r\} \tag{3.13}$$

where (3.13) is simply the probability that at least one Poisson process occurs after $\tau$ steps. To calculate this prediction for our auxiliary inputs, we need the total number of steps in our trajectory between last observing r(L$i$) as missing, and the average number of steps to reach L$i$ from the current location. We find this total number of

steps by summing these two number of steps, and finding the corresponding likelihood. Our auxiliary input is then defined as:

$$\boldsymbol{m}_t^i \doteq 1 - \exp\left\{\sum_{\tau=0}^{t} \mathbb{1}[(\boldsymbol{o}_\tau \text{ has r(L}i\text{) missing}) \ \& \ (\tau > \text{last observed r(L}i\text{) missing})] + \mathbb{E}_{\pi_i}\left[\sum_{\tau=t+1}^{\infty} \mathbb{1}[\boldsymbol{o}_\tau[3i+2] = 1]\right]\right\} \tag{3.14}$$

where $\pi_i$ corresponds to the policy of going to location L$i$ (for L1 that is simply the policy that always goes left, for L2 this policy always goes right). $\boldsymbol{o}_\tau[3i+2]$ is the observation feature that answers the question "Is the reward at L$i$ unobservable?". Again, like in Section 3.3.1, each auxiliary input here is a vector in $\mathbb{R}^1$, with $\boldsymbol{M}_t \doteq (\boldsymbol{m}_t^1, \boldsymbol{m}_t^2)$. Each auxiliary input defined here corresponds to a future prediction about the reward. In this case, our kernel function is simply a function which filters time steps based on if the step was in the past/present or future. Our preprocessing function is then defined by the function that returns the two element vector with the two predicates defined in Equation 3.14. Put together we get our likelihoods (one for each reward) which we use as auxiliary inputs for our agent.

Similar to particle filtering auxiliary inputs, since this approach summarizes its history and future predictions with likelihood functions, it is low in resolution and high in depth. It is low in resolution because these likelihoods summarize its history into a probability distribution, and would be hard to recover individual observations. It is high in depth as exponential decaying traces are also high in depth: the likelihood probability distribution is also an exponential function of time steps, and so will have depth depending on the Poisson process rate.

### 3.5.2 Results

Similar to our previous two auxiliary inputs, this approach also uses the Sarsa algorithm for control, with a step size of $\alpha \doteq 0.001$ selected based on a hyperparameter sweep. The epsilon used here was also $\epsilon \doteq 0.1$.

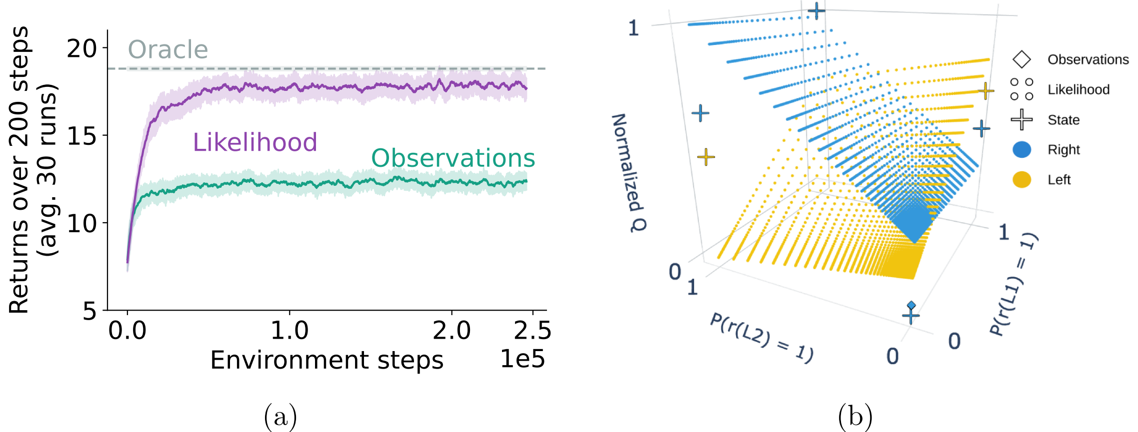



Figure 3.5: **Results for Likelihood as Agent State in the Lobster Environment**
(a): Online returns in the Lobster environment for our agent leveraging likelihood auxiliary inputs, in comparison to the optimal policy given ground-truth state (dotted line). (b): The action-values for both `left` (yellow) and `right` (blue) actions for the value function learnt over likelihood auxiliary inputs.

With these likelihoods as auxiliary inputs, we also outperform the agent using observations only, and are even closer to the optimal policy in terms of returns as shown in Fig. 3.5a. We also visualize the same normalized action-values of the two likelihood action-values in Fig. 3.5b, and see a similar greedy policy as the previous two auxiliary inputs in Sections 3.3 and 3.4.

Comparing all three auxiliary input performances in Figure 3.6, we see the similarities in the performances across the three algorithms. The likelihood approach very slightly outperforms the other two auxiliary inputs—this is likely due to the additional privileged information given to the agent in the form of closed-form probabilities of the reward being present. From all the visualizations of the action-value functions seen for the three auxiliary input approaches, we can see similarities in the learnt value functions between three auxiliary inputs that incorporate or model different kinds of information. Learning a value function over these features result in similar policies. This implies that these auxiliary inputs all help resolve partial observability in one way or another.

The results from these auxiliary inputs in this Lobster environment gives an ex-

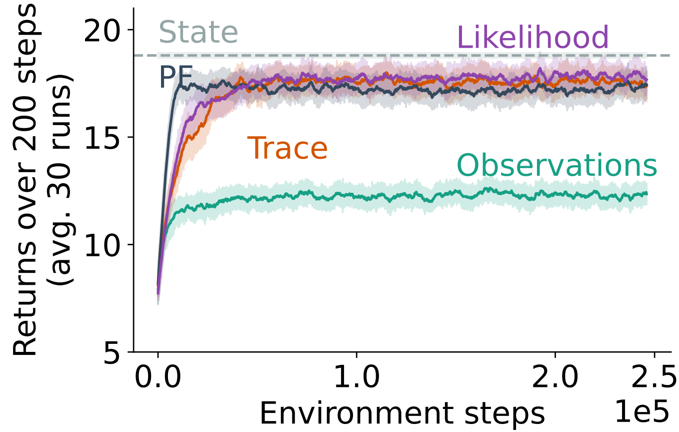



Figure 3.6: **Combined Results in Lobster Environment**
All learning curves from the three algorithms in one plot for comparison.

ample of when auxiliary inputs are useful for reinforcement learning—*For auxiliary inputs to be useful, they must add information into the agent state that is relevant to the task at hand.* All the auxiliary inputs presented in this section incorporates or models information from the past, present and/or future with regards to the two partially observed rewards. We introduce these three forms of auxiliary inputs as demonstrations and to investigate popular auxiliary inputs used in real-world reinforcement learning. Given this, we note that there are many forms of auxiliary inputs we have not considered in this manuscript; two further examples being entropy over a belief state [29] or a running average accumulator [40] as auxiliary inputs.

In the rest of this work, we investigate why and how two of these forms of auxiliary inputs are useful for reinforcement learning—particle filtering and exponential decaying traces as auxiliary inputs. We first consider the particle filtering approach.

# Chapter 4

# Particle Filtering for Auxiliary Inputs

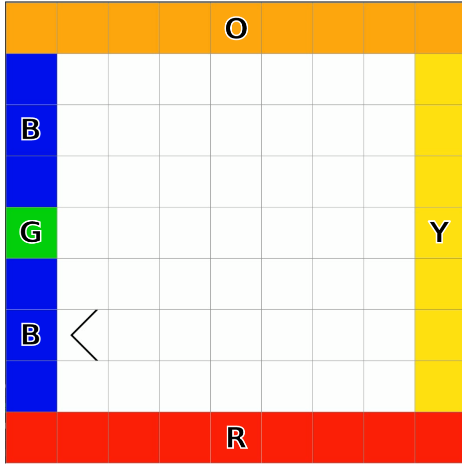
In this chapter, we further investigate and evaluate the auxiliary input approach of particle filtering for an approximate belief state, as described in Section 3.4. We look to answer the question: with the additional assumptions that this approach brings, how do these forms of auxiliary inputs fare in partially observable environments? We answer this question by evaluating this class of auxiliary inputs on two classic partially observable environments: A modified version of the *Compass World* [27] environment, and the *RockSample* [34] environment. We compare this approach to agents using leveraging gradient-based agent-state functions (LSTMs) for agent-state construction, and show that these auxiliary inputs consistently outperform LSTM-based agent-state functions.

## 4.1 Particle Filtering Environments

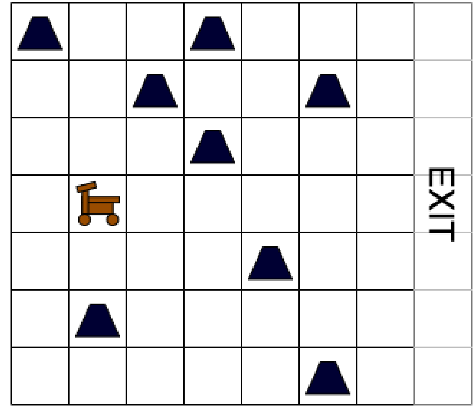
We first describe the details of the two partially observable environments that we use to evaluate our approximate belief state approach to auxiliary inputs. We start with the modified version of Compass World.

### 4.1.1 Modified Compass World

In this partially observable $9 \times 9$ grid world (as shown in Fig. 4.1a), the state of the agent is defined by both its position and its pose (or the direction its facing). The agent can be in one of the possible positions within the four walls of the grid




(a) *Modified Compass World*

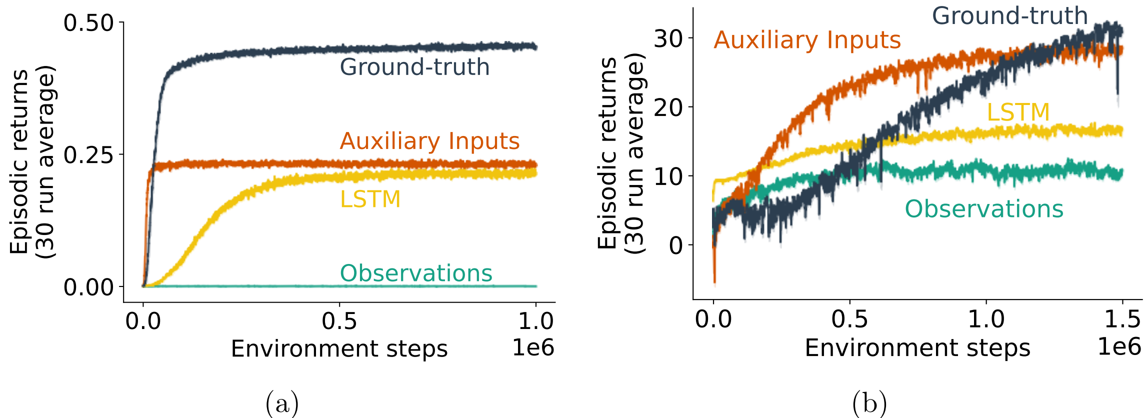



(b) *RockSample(7, 8)*

Figure 4.1: **Modified Compass World and RockSample(7, 8)**
Environments for evaluating approximate belief states as auxiliary inputs.

world, and can face one of the four cardinal directions. As for its observations, the agent can only see the color of the square directly in front of it. The observation in this environment is simply a one-hot vector $\boldsymbol{o} \in \mathcal{O} \subset \{0, 1\}^5$. The non-zero index represents the color of the square directly in front of the agent, and the zero vector represents no color in front of the agent. The agent has 3 actions: $\mathcal{A} \doteq \{$`move forward`, `turn left`, `turn right`$\}$.

The goal of the agent is to traverse the grid world to face the green wall, after which the agent receives a reward of +1 and the environment reaches a terminal state. Rewards at all other time steps are 0. The agent is initialized in a random position and pose (that is not the goal) as the initial start state. Due to both the limiting observations and random start position and pose, the agent has to resolve both its $x$ and $y$ coordinates in order to reach the goal. This environment is modified from the original Compass World environment in that the goal location is in the *middle* of the west blue wall, as opposed to the top of the west blue wall. This is to add difficulty to this environment; in the original environment, the agent would only need to traverse up to the north facing the orange wall (resolve its $y$-position) and head west, until it reached the goal. In this modified version, the agent needs to




(a) (b)

Figure 4.2: **Results for Particle Filter Auxiliary Inputs in Modified Compass World and RockSample(7, 8)**
Online discounted returns over environment steps for agents in both the Modified Compass World (4.2a) and RockSample(7, 8) (4.2b) environments. Colors in both plots correspond to the same class of agents.

resolve both $x$ and $y$ coordinates to reach the goal.

### 4.1.2 RockSample

Rocksample(7, 8) (as shown in Fig. 4.1b) is partially observable environment with a $7 \times 7$ grid world with 8 rocks randomly scattered throughout the environment. At the start of an episode, each rock is randomly assigned to be either good or bad. Initially, the agent is unaware of whether or not rocks are good or bad, but has individual actions to check the goodness of each rock. This check action is done with an imperfect sensor that gets noisier the farther away the agent is to each rock. Besides these check actions (one for each rock), the agent can also move in the four cardinal directions, with a total of $8 + 4 = 12$ actions.

The goal of the agent is to collect as many good rocks as possible before exiting to the eastern border of the grid world. Collecting a good rock gives a positive reward of $+10$, and exiting to the right also gives a positive reward of $+10$. Collecting a bad rock results in a negative reward of $-10$. After a good rock is collected, it turns into a bad rock, and further collecting of this rock results in a negative reward of $-10$. Further details of this environment are listed in Appendix B.1.1

## 4.2 Results and Discussion

We compare our particle-filter-based auxiliary inputs to other agent-state functions for Modified Compass World and RockSample in Figures 4.2a and 4.2b respectively. The particle filtering-based agent (orange, labelled as "Auxiliary Inputs") is compared to three baseline agents: An agent using only observations (teal, labelled as "Observations"), an agent which uses LSTM as its agent-state function (yellow, labelled as "LSTM"), and an agent that uses the ground-truth environment state (blue, labelled as "Ground-truth").

In these figures, we plot the online returns during training over environment steps. All experimental results shown report the mean (solid lines) and standard error to the mean (shaded region) over 30 runs. Hyperparameters for each algorithm were selected based on a sweep. In these experiments, all agents utilize similar neural network architectures as its function approximator. Further details of the experimental setup and hyperparameters swept can be found in Appendix B. Further details of the particle filter and this agent-state function for each environment in this section can be found in Appendix B.2. In these results, we plot online discounted returns over environment steps.

As a point of clarification, the LSTM-based agent conditions on both the observation and action [32] at every time step. We describe the specifics of this LSTM action conditioning technique, as well as ablations for the LSTM agent with and without action conditioning in Appendix B.

The particle filter agent outperforms the baselines with the exception of the agent acting on the ground-truth state in both environments. These discrepancies in the discounted returns that each agent converges to reflect the ability for this auxiliary input to preserve useful information with regards to its trajectory for decision making, given the additional assumptions that the approximate belief state entails.

In Modified Compass World, the ground-truth agent converges to a much higher

return than the other agents. This is because it is fully observable and has more information available to it in its data stream than the other agents—namely the position and orientation of the agent. The learnt policy for this agent traverses directly to the green goal state, without having to localize first. For RockSample, all the actual moralities (goodness or badness) of the rock are available to the ground-truth agent at the start of an episode. The learnt policy for this agent traverses directly to the good rocks, collects them, then exits to the right.

On the other hand, the agents acting on only observations must first resolve its partial observability. We first consider the learnt policies from both the LSTM and particle filter agent states in Modified Compass World. The agents first move forward until they see a wall color. Seeing this wall color resolves one of their position coordinates, and allows the agent to navigate towards the west wall. The agent then traverses either up or down, periodically checking the color of the west wall until they can resolve the other position coordinate, and get to the goal. This resolving of coordinates is *explicitly* represented as elements of the distribution over state going to 0. In RockSample, using an approximate belief state as auxiliary inputs is particularly useful in this environment because these auxiliary inputs allow the agent to learn a policy that takes into account the uncertainty of the rocks in the current time step. The agent will traverse closer to a rock before checking whether the rock is good or bad since accuracy of the check decreases with distance. From the results, the plots also suggest that the additional information that these particle filter auxiliary inputs provide also helps in exploration and the rate of learning of the agent in both these environments, something that we briefly discuss as future work in Chapter 6.

The particle filter auxiliary input approach (with the additional assumptions that approximate belief states afford) also consistently outperforms the agent using an LSTM agent-state function. Not only do these auxiliary inputs converge faster, but also converges to a higher average return. This implies that these auxiliary inputs are able to represent and add information into the agent state that may be hard to

represent in recurrent neural network approaches, and are also beneficial for faster value function learning.

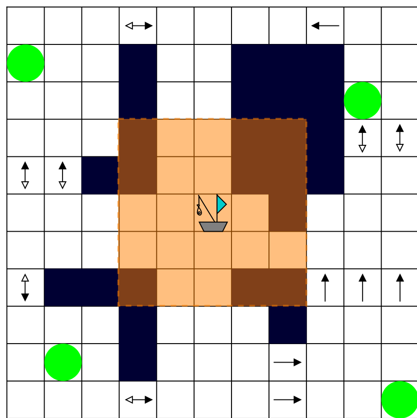
Incorporating trajectory information with particle filtering leveraged knowledge of the dynamics of the environment in order to form auxiliary inputs that were highly relevant for decision making, based on a distribution over ground-truth state. While the particle filtering approximate belief state assumptions may be strong assumptions to make, it is the case in many real-world use cases that these assumptions (or approximations thereof) are not completely unreasonable. Now that we have shown that auxiliary inputs are able to add complex, relevant information to the agent state given a model of the environment, we now consider the case where we do not have these assumptions available and investigate modelling the past for auxiliary inputs.

# Chapter 5

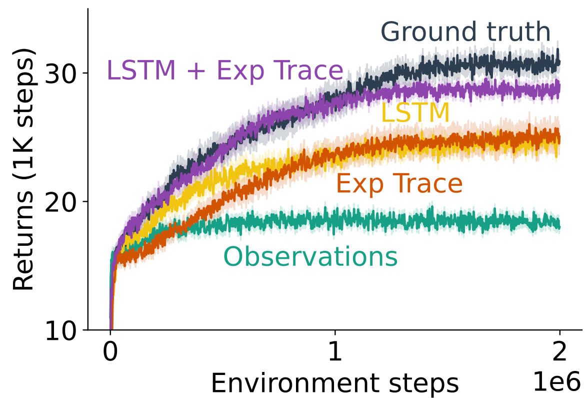
# Scaling Up Auxiliary Inputs and Integration with RNNs

With the efficacy of auxiliary inputs demonstrated in smaller environments, in this section we now consider how to scale up approaches to auxiliary inputs, as well as the role of auxiliary inputs in gradient-based agent-state functions. Specifically, we investigate ways in which to scale up exponential decaying traces to larger, pixel-based environments, and how these traces integrate with recurrent neural networks as an agent-state function, as described in Section 2.9. To do this, we first introduce two scaled-up versions of the Lobster environment—the Fishing environments. These new environments are two continuing, partially observable environments that require the agent to resolve its partial observability and collect regenerating rewards in a 2 dimensional stochastic grid world.

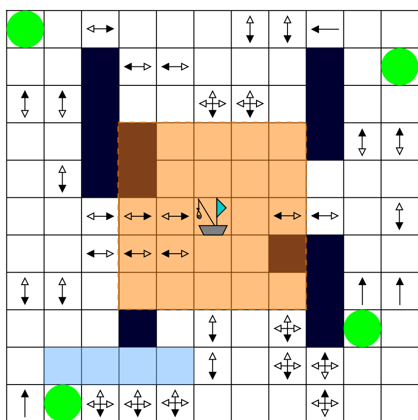
Using these two Fishing environments as testbeds, we show how we use exponential decaying traces as auxiliary inputs for this partially observable pixel-based environment, and compare and integrate these trace features with recurrent neural network agent-state functions. We show that in these environments, exponential decaying traces either match or outperform an LSTM-based agent. Furthermore, we show that combining LSTMs together with exponential decaying traces produces even better performance, as compared to using either only LSTMs or only exponential decaying traces.



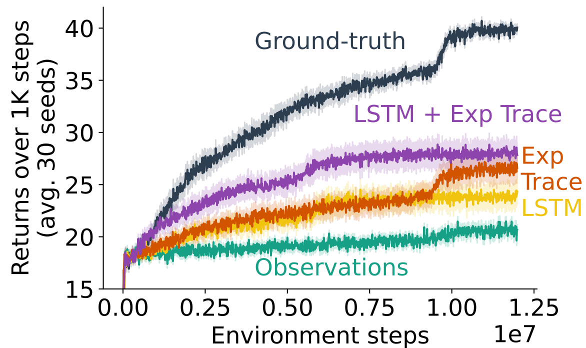
(a) Fishing 1.




(b) Fishing 1 results.



(c) Fishing 2.




(d) Fishing 2 results.

Figure 5.1: **Fishing Environments and Results**
(a, c) Fishing Boat environments. At every time step the agent receives a map with the $5 \times 5$ area around itself updated. This map includes obstacles, currents and rewards. Currents labelled with multiple directions represent stochastic currents. Dark blue tiles represent obstacles, whereas light blue tiles represent obstacles the agent is able to see through. (b, d) Results for the first and second Fishing Boat environments respectively, averaged over 30 runs. Standard errors are shaded for each curve.

## 5.1 The Fishing Environment

We first introduce the *Fishing* environments - two pixel-based stochastic and partially observable environments. This environment is reminiscent of a scaled-up version of the Lobster environment, in that it was designed to test an agent's ability to reason about partial observability and time, except in a scaled-up setting. In both of these foraging environments, the goal of the agent (a fishing boat) is to continually navigate around the stochastic currents and obstacles to collect fish from fishing locations (denoted by the green circles). Currents push the agent one step in the direction it is facing. After collecting the fish from a net, the net is re-cast and fills up over a random amount of time.

While full details of both environments are elucidated in Appendix C.1, we give a summary of the environment dynamics in this section. We depict Fishing 1 and Fishing 2 in Figures 5.1a and 5.1c, respectively. There are 4 actions in both environments, each corresponding to moving in one of the cardinal directions. Both environments are represented by an $11 \times 11$ image with 4 rewards, as denoted by the green points. Like in the Lobster environment (c.f. Section 3.2), in these environments the four rewards regenerate stochastically after being collected. Once collected, the rewards stay and do not disappear without being collected, and the agent receives a reward of $+1$. Currents in these environments also change stochastically over time. The multi-directional arrows represent these shifting currents for each given position, and the directions in which they might be in. In addition to these sources of stochasticity, the agent also has a chance of "slipping" at every time step and remaining in the same position. Beyond this, there are also walls throughout the environment, denoted by the dark blue tiles, that block the agent from traversing to or through. Bumping into a wall results in a no-op. Finally, Fishing 2 in Figure 5.1c also has a glass wall denoted by the blue tiles, which the agent can see through, but cannot traverse through. We now describe the observations that the agent sees at every step.

Partial observability in the domain comes from a few sources. The first source of partial observability is the environment map which the agent receives as observations. At every step, the position of the agent is given and an agent-centric map of the environment is accumulated, much like in robot navigation and mapping [e.g., 6, 38]. In a given step, the map is only updated with a potentially occluded $5 \times 5$ area around the agent's current position. Walls throughout the environment occlude parts of the $5 \times 5$ observable map. This $5 \times 5$ area contains information on the relative positions and the direction of the currents, walls and rewards at the current time step. As the agent traverses about the environment, previous features in the accumulated map (such as reward availability and current direction) begin to "stale" since currents and rewards change stochastically with time, and only get updated again after the agent traverses to the relevant area and observes the $5 \times 5$ image of the map which the feature resides in. Details of this mapping is described in Appendix C.1.1.

Due to the stochasticity of the currents and rewards, the degree of partial observability of each environment is dictated by the number of stochastic elements throughout the map, and the rate at which these random variables change. In Fishing 1 (Figure 5.1a), we have an environment with low levels of partial observability and stochasticity. There is a sparse number of currents throughout the map, with most currents acting as a gateway to the rewarding areas. Rewards and currents in this environment also have a relatively slow rate of change. Fishing 2 (Figure 5.1c) is an environment that is much more partially observable, with several fast-changing currents throughout the environment. Full details of the stochasticity of both environments are specified in Appendix C.1.2.

## 5.2 Exponential Decaying Traces For Mapping

To encode information with regards to the past history of the agent as an auxiliary input for these environments, we use exponential decaying traces (as per Section 3.3) over the past observable regions in the environment map. In this case we have auxil-

iary inputs that are all updated at once as a *matrix*, which we define as $\boldsymbol{M}_t \in \mathbb{R}^{m \times m}$, where $m$ is the width and height of the map, with $N = m$. Our auxiliary input is then:

$$\boldsymbol{M}_t \doteq \sum_{\tau=0}^{t} \lambda^{t-\tau} \mathbb{1}(\boldsymbol{o}_\tau) \tag{5.1}$$

where our kernel function is the same as defined in Section 3.3, and our preprocessing function is defined as $g(\boldsymbol{o}, a) \doteq \mathbb{1}(\boldsymbol{o}_\tau) \in \{0,1\}^{m \times m}$. $\mathbb{1}(\boldsymbol{o}_\tau)$ is an $m \times m$ binary map which indicates which areas of the global map are observable from $\boldsymbol{o}_\tau$ at time $\tau$. At each step, all the locations that are not currently observable are decayed by a factor of $\lambda < 1$. This auxiliary input encodes the time since the agent has observed a particular location as an exponentially decaying timer. The incremental version of this auxiliary input is simply:

$$\boldsymbol{M}_t \doteq \max(\lambda \boldsymbol{M}_{t-1} + \mathbb{1}(\boldsymbol{o}_t), \mathbf{1}) \tag{5.2}$$

where $\mathbf{1}$ is an $m \times m$ matrix of ones.

## 5.3 Results and Discussion

We show our results for both environments in Figures 5.1b and 5.1d. Experimental details including hyperparameters swept, algorithmic details, and environment details are included in Appendix C. Results shown here are offline evaluation returns over environment steps, where we evaluate our agent after every fixed number of steps. We compare our exponential decaying trace auxiliary inputs (orange) to a few baselines: an LSTM-based agent with action concatenation (yellow), and an agent with only the observation map as described before as input (teal), and a combination of both the trace auxiliary inputs combined with an LSTM agent-state function (purple). In both environments, exponential decaying traces as auxiliary inputs are comparable to, or performs slightly better than the LSTM-based agent. In the simpler Fishing 1, using exponential decaying traces matches the performance of the LSTM agent, as shown in Figure 5.1b. While the LSTM agent performs temporal credit assignment by

using additional compute with T-BPTT and TD error propagation, our exponential decaying trace can be seen as a simple way of performing temporal credit assignment by only propagating TD error *through the input features*, reminiscent of eligibility traces [37]. Doing so requires much less computation per time step as compared to T-BPTT. In Fishing 2, the additional stochasticity seems to harm the performance of the LSTM agent as compared to the agent using exponential decaying traces as auxiliary inputs, with the decaying trace agent slightly outperforming the LSTM agent.

Exponential decaying traces are also able to integrate *with* LSTMs. In both environments, combining decaying trace auxiliary inputs with LSTM function approximation increased the performance of the agent by quite a large margin. As pointed out by [26], adding an exponential decaying trace as input to an LSTM learning through T-BPTT seems to add robustness to the truncation window length. In our case, for control, it seems to both increase the rate of learning, and also increase the average returns of the learnt policy. This example suggests that auxiliary inputs can integrate well with gradient-based agent-state construction.

# Chapter 6

# Conclusion and Future Work

To conclude, auxiliary inputs are helpful tools for reinforcement learning practitioners to resolve partial observability which also have the potential to integrate well with existing, gradient-based agent-state functions.

In this work we advocate for the general principle of auxiliary inputs as an addition to agent-state construction, and evaluate different instantiations of auxiliary inputs for reinforcement learning. We first formalize auxiliary inputs, as well as consider three different instantiations of these inputs in Chapter 3. Using the Lobster environment introduced in Section 3.2, we demonstrate the efficacy of these auxiliary inputs in resolving partial observability, as well as how these auxiliary inputs allow us to *expand* the input feature space of the agent to allow us to interpolate between ground-truth states, and for a more fine-grained policy. With this formalism in place, we investigate the performance of the particle filtering approach to auxiliary inputs on a few classic partially observable environments in Chapter 4. We show the efficacy of approximate belief states as auxiliary inputs in these hard, partially observable environments. Finally, in Chapter 5, we investigate the use of simple exponential decaying traces of observation features as auxiliary inputs on the scaled-up Fishing environment. Besides showing matching or better performance of these trace features in this environment as compared to LSTMs, we also show how this auxiliary input can *integrate* with recurrent neural network agent-state functions, and improve performance.

As for future work, the most immediate extension of our investigation would be to use more complex predictions as auxiliary inputs. In terms of using future predictions to resolve partial observability, general value functions [35] and predictive state representations [18] are two promising approaches for using predictions for resolving partial observability as an auxiliary input. While a promising area of research, future predictions have their limitations: in the context of using future predictions for next-step predictions as well as for control, general value functions have been shown to be effective in only a very limited scope of environments and predictions [32]. Another very promising direction for future work would be to integrate and combine different forms of auxiliary inputs, which leverage information from all parts of the trajectory, from both past and future. Another promising direction for future work is to leverage auxiliary inputs for better exploration. While techniques to estimate the uncertainty of an agent for exploration is near ubiquitous throughout reinforcement learning, an interesting avenue for exploration is to alter the *inputs* of an agent for more robust exploration.

# Bibliography


[1] Bellemare, M. G., Candido, S., Castro, P. S., Gong, J., Machado, M. C., Moitra, S., Ponda, S. S., and Wang, Z. (2020). Autonomous Navigation of Stratospheric Balloons using Reinforcement Learning. *Nature*, 588 7836:77–82.

[2] Bellemare, M. G., Naddaf, Y., Veness, J., and Bowling, M. (2013). The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279.

[3] Bellman, R. (1957). A markovian decision process. *Journal of Mathematics and Mechanics*, 6(5):679–684.

[4] Block, H. D. (1962). The perceptron: A model for brain functioning. i. *Reviews of Modern Physics*, 34:123–135.

[5] Degrave, J., Felici, F., Buchli, J., Neunert, M., Tracey, B., Carpanese, F., Ewalds, T., Hafner, R., Abdolmaleki, A., de las Casas, D., Donner, C., Fritz, L., Galperti, C., Huber, A., Keeling, J., Tsimpoukelli, M., Kay, J., Merle, A., Moret, J.-M., Noury, S., Pesamosca, F., Pfau, D., Sauter, O., Sommariva, C., Coda, S., Duval, B., Fasoli, A., Kohli, P., Kavukcuoglu, K., Hassabis, D., and Riedmiller, M. (2022). Magnetic Control of Tokamak Plasmas through Deep Reinforcement Learning. *Nature*, 602(7897):414–419.

[6] Elfes, A. (1987). Sonar-based real-world mapping and navigation. *The Institute of Electrical and Electronics Engineers Journal on Robotics and Automation*, 3(3):249–265.

[7] Elman, J. L. (1990). Finding structure in time. *Cognitive Science*, 14(2):179–211.

[8] Ghiassian, S., Rafiee, B., Lo, Y. L., and White, A. (2020). Improving performance in reinforcement learning by breaking generalization in neural networks. In *Proceedings of the 19th International Conference on Autonomous Agents and MultiAgent Systems*, AAMAS '20, page 438–446, Richland, SC. International Foundation for Autonomous Agents and Multiagent Systems.

[9] Hausknecht, M. J. and Stone, P. (2015). Deep recurrent q-learning for partially observable mdps. In *2015 Association for the Advancement of Artificial Intelligence Fall Symposia, Arlington, Virginia, USA, November 12-14, 2015*, pages 29–37. AAAI Press.

[10] Hochreiter, S. and Schmidhuber, J. (1997). Long Short-term Memory. *Neural Computation*, 9:1735–1780.

[11] Kaelbling, L. P., Littman, M. L., and Cassandra, A. R. (1998a). Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101:99–134.

[12] Kaelbling, L. P., Littman, M. L., and Cassandra, A. R. (1998b). Planning and Acting in Partially Observable Stochastic Domains. *Artificial Intelligence*, 101(1-2):99–134.

[13] Kingma, D. P. and Ba, J. (2015). Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations Conference Track Proceedings*.

[14] Kitagawa, G. (1996). Monte Carlo Filter and Smoother for Non-Gaussian Nonlinear State Space Models. *Journal of Computational and Graphical Statistics*, 5(1):1–25.

[15] Konidaris, G., Osentoski, S., and Thomas, P. (2011). Value function approximation in reinforcement learning using the fourier basis. In *Proceedings of the Twenty-Fifth Association for the Advancement of Artificial Intelligence Conference*, page 380–385. AAAI Press.

[16] Kurniawati, H. (2022). Partially observable markov decision processes and robotics. *Annual Review of Control, Robotics, and Autonomous Systems*, 5(1):253–277.

[17] Lin, L.-J. (1992). Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine Learning*, 8(3):293–321.

[18] Littman, M. L., Sutton, R. S., and Singh, S. (2001). Predictive Representations of State. In *Advances in Neural Information Processing Systems*, pages 1555–1561.

[19] Lu, X., Roy, B. V., Dwaracherla, V. R., Ibrahimi, M., Osband, I., and Wen, Z. (2021). Reinforcement Learning, Bit by Bit. *ArXiv*, abs/2103.04047.

[20] Machado, M. C., Bellemare, M. G., Talvitie, E., Veness, J., Hausknecht, M. J., and Bowling, M. (2018). Revisiting the Arcade Learning Environment: Evaluation Protocols and Open Problems for General Agents. *Journal of Artificial Intelligence Research*, 61:523–562.

[21] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M. A., Fidjeland, A., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., and Hassabis, D. (2015). Human-level Control through Deep Reinforcement Learning. *Nature*, 518:529–533.

[22] Mozer, M. C. (1996). *Neural Net Architectures for Temporal Sequence Processing*, volume 265, page 243–264. Addison-Wesley.

[23] Pilarski, P., Dick, T., and Sutton, R. (2013). Real-time prediction learning for the simultaneous actuation of multiple prosthetic joints. In *IEEE International Conference on Rehabilitation Robotics*, volume 2013, pages 1–8.

[24] Pilarski, P. M., Dawson, M. R., Degris, T., Carey, J. P., and Sutton, R. S. (2012). Dynamic switching and real-time machine learning for improved human control of assistive biomedical robots. In *2012 4th IEEE RAS & EMBS International Conference on Biomedical Robotics and Biomechatronics*, pages 296–302.

[25] Pineau, J. and Gordon, G. J. (2007). POMDP Planning for Robust Robot Control. In *Robotics Research*, pages 69–82.

[26] Rafiee, B., Abbas, Z., Ghiassian, S., Kumaraswamy, R., Sutton, R. S., Ludvig, E. A., and White, A. (2022). From eye-blinks to state construction: Diagnostic benchmarks for online representation learning. *Adaptive Behavior*.

[27] Rafols, E., Koop, A., and Sutton, R. S. (2005a). Temporal Abstraction in Temporal-difference Networks. In *Advances in Neural Information Processing Systems*, pages 1313–1320.

[28] Rafols, E. J., Ring, M. B., Sutton, R. S., and Tanner, B. (2005b). Using predictive representations to improve generalization in reinforcement learning. In *International Joint Conferences on Artificial Intelligence*, pages 835–840.

[29] Roy, N. and Thrun, S. (1999). Coastal Navigation with Mobile Robots. In *Advances in Neural Information Processing Systems*, pages 1043–1049.

[30] Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, 323(6088):533–536.

[31] Rummery, G. A. and Niranjan, M. (1994). On-line Q-learning using connectionist systems. Technical Report TR 166, Cambridge University Engineering Department, Cambridge, England.

[32] Schlegel, M., Jacobsen, A., Abbas, Z., Patterson, A., White, A., and White, M. (2021). General Value Function Networks. *Journal of Artificial Intelligence Research*, 70:497—-543.

[33] Smallwood, R. D. and Sondik, E. J. (1973). The optimal control of partially observable markov processes over a finite horizon. *Operations Research*, 21(5):1071–1088.

[34] Smith, T. and Simmons, R. (2004). Heuristic Search Value Iteration for POMDPs. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 520—-527.

[35] Sutton, R., Modayil, J., Delp, M., Degris, T., Pilarski, P., White, A., and Precup, D. (2011). Horde : A scalable real-time architecture for learning knowledge from unsupervised sensorimotor interaction categories and subject descriptors. volume 2, pages 761–768.

[36] Sutton, R. S. (1988). Learning to predict by the methods of temporal differences. *Machine Learning*, 3(1):9–44.

[37] Sutton, R. S. and Barto, A. G. (2018). *Reinforcement Learning: An Introduction*. The MIT Press, Cambridge, MA, USA.

[38] Thrun, S. (1998). Learning metric-topological maps for indoor mobile robot navigation. *Artificial Intelligence*, 99(1):21–71.

[39] Thrun, S. (1999). Monte Carlo POMDPs. In *Advances in Neural Information Processing Systems*.

[40] Vasan, G. and Pilarski, P. (2017). Learning from demonstration: Teaching a myoelectric prosthesis with an intact limb via reinforcement learning. *IEEE ... International Conference on Rehabilitation Robotics : [proceedings]*, 2017:1457–1464.

[41] Vinyals, O., Babuschkin, I., Czarnecki, W. M., Mathieu, M., Dudzik, A., Chung, J., Choi, D. H., Powell, R., Ewalds, T., Georgiev, P., Oh, J., Horgan, D., Kroiss, M., Danihelka, I., Huang, A., Sifre, L., Cai, T., Agapiou, J. P., Jaderberg, M., Vezhnevets, A. S., Leblond, R., Pohlen, T., Dalibard, V., Budden, D., Sulsky, Y., Molloy, J., Paine, T. L., Gulcehre, C., Wang, Z., Pfaff, T., Wu, Y., Ring, R., Yogatama, D., Wünsch, D., McKinney, K., Smith, O., Schaul, T., Lillicrap, T., Kavukcuoglu, K., Hassabis, D., Apps, C., and Silver, D. (2019). Grandmaster Level in StarCraft II Using Multi-Agent Reinforcement Learning. *Nature*, 575:350–354.

[42] Werbos, P. J. (1990). Backpropagation through time: What it does and how to do it. *Proceedings of the Institute of Electrical and Electronics Engineers*, 78:1550–1560.

[43] White, A. (2015). *Developing a Predictive Approach to Knowledge*. PhD thesis, University of Alberta.

[44] Won, D.-O., Müller, K.-R., and Lee, S.-W. (2020). An adaptive deep reinforcement learning framework enables curling robots with human-like performance in real-world conditions. *Science Robotics*, 5(46):eabb9764.

# Appendix A: Lobster Environment Experimental Details

## A.1 Hyperparameters and Experimental Setup

We now detail the experimental setup and hyperparameters swept for all agents mentioned in Chapter 3. We first detail all the shared settings and swept hyperparameters between all algorithms:

- Learning algorithm: $Sarsa(0)$
- Function approximator: *Linear*
- Optimizer: *Adam*
- Discount rate: $\gamma = 0.9$
- Environment train steps: 250K
- Max episode steps: 200
- Step sizes: $[10^{-2}, 10^{-3}, 10^{-4}, 10^{-5}]$

Because this environment is a continuing task, we evaluate our agents based on the undiscounted returns over 200 time steps, with no terminal time steps. All hyperparameter sweeps were done over 30 seeds, with the best hyperparameters decided for each algorithm based on mean undiscounted returns over these 200 time steps over these $250K$ steps and 30 seeds. The results reported are over 30 different, additional seeds, run on the selected hyperparameters. We now briefly describe the two baselines that we employ as comparisons to the auxiliary input techniques we introduce, as well as minor details of the auxiliary input algorithms used in this section.

## A.2 Algorithmic details

### A.2.1 Observations only

As a baseline for performance, we consider the observations only agent. This agent simply uses the Sarsa(0) algorithm and the configurations listed in Section A.1 to learn a policy over the observations described in Equation 3.5. This agent is labelled in teal as "Obs only".

### A.2.2 Value Iteration with Environment States

We also consider an optimal agent acting on the fully-observable version of this task, with full knowledge of the transition dynamics. We use this agent to see how close to optimal our partially observable agents can perform. We use the transition probabilities over the 12 possible states to perform value iteration [3] to calculate the optimal value functions for control. We iterate through all states until our maximum change in value function over all states $\Delta$ is less than the threshold value $\theta = 10^{-10}$, $\Delta < \theta$. We use this optimal value function over the 12 states together with the transition probabilities to get the optimal policy. We evaluate this optimal policy over 200 steps and collect 1000 runs to get both the mean and standard error to the mean as the shaded dotted line in all learning rate plots: Figures 3.3a, 3.4a, 3.5a and 3.6.

### A.2.3 Exponential Decaying Trace

For our trace decaying auxiliary input, we use a $\lambda = 0.9$ as the decay rate for the exponential decaying traces for each reward.

### A.2.4 Particle Filtering

With particle filtering, our approximate belief state becomes more accurate with more particles we initialize with. In this environment, we instantiate the particle filter with 100 particles to begin with. In the rare case of particle depletion, where there are no

particles in the current environment state, we reset all particle weights to be uniform and re-weight them from that time step onwards.

### A.2.5 Likelihood Predictions

To predict the ground-truth reward regeneration likelihoods, we assume we know the ground-truth rates of our Poisson processes for both rewards, which in this case is $\frac{1}{\Lambda} \doteq \frac{1}{10}$ for both rewards (on average, 1 reward regeneration every 10 steps). With this rate of reward, we calculate the likelihood of a reward being present, given that we get to the respective rewarding location as per Section 3.5.1.

# Appendix B: Particle Filtering Experimental Details

## B.1 Environment Details

Below we consider all the details of the RockSample environment. We do not provide further details of the Modified Compass World environment here because the description in Section 4.1.1 fully specifies the environment.

### B.1.1 RockSample Environment Details

We now consider the environment and implementation details of the RockSample environment not considered in Section 4.1.2.

The position of each rock is determined in this environment based on a uniform sampling (without replacement) of 8 positions out of all possible $7 \times 7$ positions in the grid. This position is deterministically defined by the seed which the experiment is run on, and these positions do not change for each agent trained on a particular seed. The initial goodness and badness of rocks is determined based on a uniform Bernoulli distribution for every rock at every environment reset.

As mentioned when introducing this environment, the sensor available to the agent for checking the goodness and badness of rocks has noise proportional to the $L_2$ distance between the agent and the rock in question, which we denote as $\delta$. This noise is essentially based on another Bernoulli distribution, where with probability $p(\delta)$ the sensor returns the correct sensor reading of the rock, and with probability $1 - p(\delta)$ the agent receives an incorrect reading. The probability function $p$ is defined

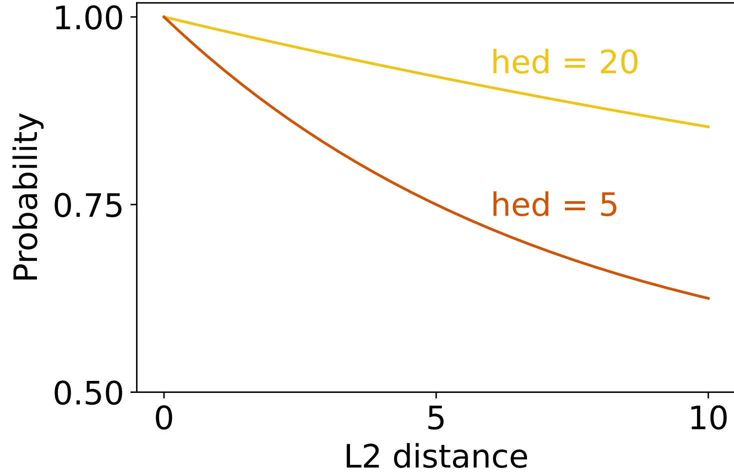
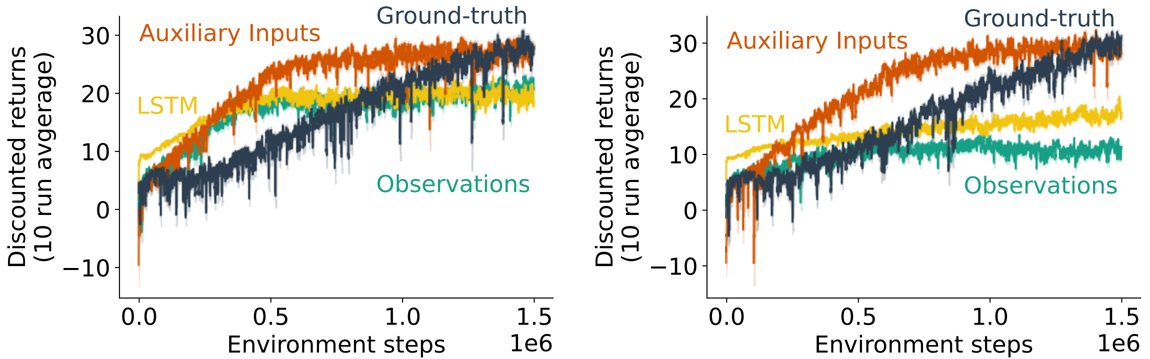



Figure B.1: **Half Efficiency Distance Function Plots**
Probability of a correct sensor reading as a function of distances for different half efficiency distances ($\delta_{hed} = 5, 20$).




(a) Half efficiency distance $\delta_{hed} = 20$ results. (b) Half efficiency distance $\delta_{hed} = 5$ results.

Figure B.2: **RockSample results for both** $\delta_{hed} = 5, 20$
In (a) and (b) mean and standard error to the mean are shown over 10 seeds. We use $\delta_{hed} \doteq 5$ for the results in our work.

by the *half efficiency distance* $\delta_{hed}$ of the sensor. Overall, this probability function is defined by the function:

$$p(\delta) \doteq 0.5 \times (1 + 2^{-\frac{\delta}{\delta_{hed}}}). \tag{B.1}$$

We plot this function in Figure B.1. We also discuss results for different agents introduced in Chapter 4 in RockSample for different half efficiency distances in Section B.3. In our results in Section 4.1.2, we use a half efficiency distance of 5.

## B.2 Environment-Specific Algorithmic Details and Hyperparameters

We now detail the environment-specific algorithmic details and hyperparameters swept for all agents pertaining to Modified Compass World and RockSample.

### B.2.1 Modified Compass World Experimental Setup and Hyperparameters

For our Modified Compass World experiments, we use and sweep the following hyperparameters:

- Learning algorithm: $Sarsa(0)$
- Function approximator: Neural Network
- Layers: 1
- Hidden units: 100
- Optimizer: *Adam*
- Discount rate: $\gamma = 0.9$
- Environment train steps: 1M
- Max episode steps: 1000
- Step sizes: $[10^{-3}, 10^{-4}, 10^{-5}]$
- Number of particles: 1 for each possible start state ($9 \times 9 \times 4 - 1 = 323$)

All hyperparameter sweeps were done over 10 seeds, with the best hyperparameters decided for each algorithm based on mean discounted returns over these 200 time steps and 10 seeds. We then use these selected hyperparameters and run experiments for 30 different seeds to obtain the results presented in Figure 4.2a. For the LSTM agents in this environment, the agents all use one-hot action concatenation [32] with its input features to the LSTM cell. An ablation study for this action concatenation is done in Appendix B.3.2.

#### Observations Only

With the observations only baseline, our observation vector is defined by a vector of size 5, where each feature corresponds to whether or not the color directly in front

of the agent is being observed. An all zero vector represents no color being shown in front of the agent.

**Particle Filtering**

We use particle filtering to approximate a belief state over the possible pose and position of the agent. Our feature vector for this approach is of size $7 \times 7 \times 4 + 5$, where the first $7 \times 7 \times 4$ features represent all possible combinations of positions and poses of the agent, and the last 5 features are the same observation vector as described in Appendix B.2.1. This position and pose belief state is approximated through the particle filtering approach described in Section 3.4, where emission probabilities are simply binary variables representing whether or not each position and pose combination can emit the given color. For this environment, we simply use one particle for each possible starting position and pose combination, $7 \times 7 \times 4-$, since the environment dynamics are deterministic, outside of the initial start state.

**Recurrent Neural Network**

Finally, our RNN-based approach uses the same observations as described in Appendix B.2.1, except with an LSTM as the function approximator. In addition to using a recurrent neural network for function approximation, we also use *action conditioning* [32]. In our setting, actioning conditioning simply consists of concatenating a one-hot encoding of the previous time step's action to the observation vector fed into the RNN.

### B.2.2 RockSample(7, 8) Experimental Setup and Hyperparameters

In our RockSample(7, 8) experiments, we leverage a replay buffer [17] for all of our experience.

For the Rocksample(7, 8) experiments, we use and/or sweep the following hyperparameters:

- Learning algorithm: $Sarsa(0)$
- Function approximator: Neural Network
- Layers: 1
- Hidden units: 100
- Optimizer: *Adam*
- Discount rate: $\gamma = 0.99$
- Environment train steps: 1.5M
- Max episode steps: 1000
- Step sizes: $[10^{-3}, 10^{-4}, 10^{-5}]$
- Buffer size: $[10K, 100K]$
- Number of particles: 100

All hyperparameter sweeps were done over 10 seeds, with the best hyperparameters decided for each algorithm based on mean discounted returns over these 200 time steps. We then use these selected hyperparameters and run experiments for 30 different seeds to obtain the results presented in Figure 4.2b. For the LSTM agents in this environment, the agents also use one-hot action concatenation (c.f. Appendix B.3.2).

.

**Observations Only**

With the observations only baseline, our observation vector is defined by a vector of size $7+7+8 = 22$, where the first $7+7$ features represent one-hot encodings of the $x$ and $y$ coordinates of the agent respectively, and the final 8 features represent the most recently observed rock moralities (goodness or badness). In our implementation of RockSample, these observed rock moralities are initialized at 0.5, and take on values depending on the most recent check of each rock. So if rock number 1 was checked and seen as good 5 steps ago, and this was the most recent check of rock 1, then the feature representing this rock would be a 1 feature, representing a good rock.

**Particle Filtering**

To approximate a belief distribution over state, we leverage particle filtering as mentioned in Section 3.4. In this approach, our input feature vector is also defined as a vector of length $7 + 7 + 8$. The first $7 + 7$ features are once again a one-hot encoding of the $x$ and $y$ coordinates for the first and second 7 features respectively. The final

8 features are an approximate belief state of the current state of the rock *moralities*, instead of the underlying state of the environment. This is to reduce the dimensionality of the input features. These last 8 features are simply the normalized sum over the weights over all particles. For this particle filtering algorithm, we start with 100 particles as well.

#### Recurrent Neural Network

Finally, our RNN-based approach uses the same observations as Appendix B.2.2, except with an LSTM as the function approximator. Similarly to the Modified Compass World LSTM agent, we also use action conditioning here.

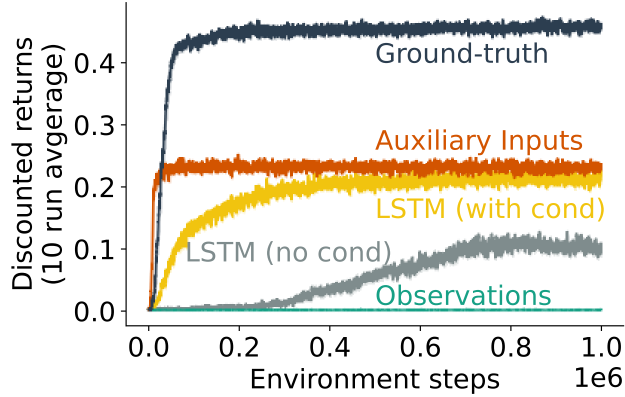
Since we use a replay buffer with LSTMs for this algorithm, to fix a truncation length for T-BPTT, we sample trajectories from our replay buffer of length *truncation length*, and roll our trajectories out and propagate gradients backwards across the sampled trajectory. This requires us to store and sample hidden states from our LSTM in our replay buffer.

## B.3 Ablation Studies

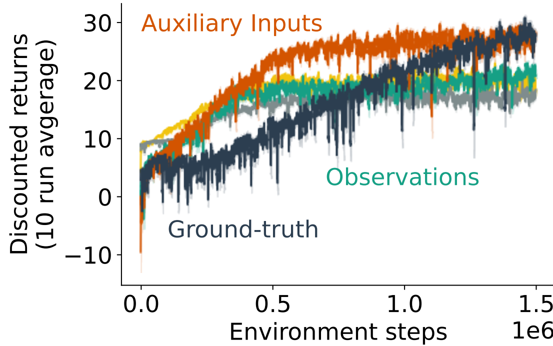
Here we list the ablation studies we perform over both our environments and select algorithms.

### B.3.1 RockSample(7, 8) Half Efficiency Distance Experiment

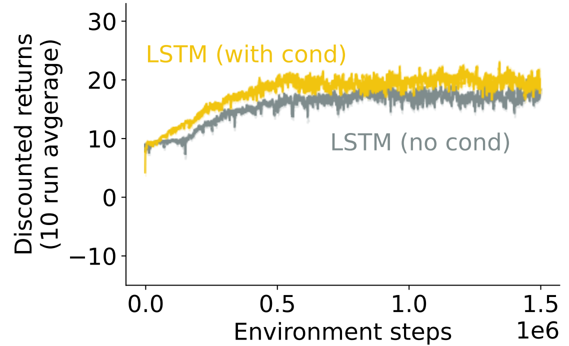
We perform a small experiment to see the effect of the half efficiency distance $\delta_{hed}$ on the performance of our algorithms in Figures B.2a and B.2b. From these learning curves, we can conclude that a lower $\delta_{hed}$ (or a less accurate sensor over distance to rocks) doesn't significantly affect the performance of the particle filtering auxiliary input, nor does it affect the performance of the ground-truth agent. This parameter seems to affect the LSTM-based agent and observations-only-based agent the most, decreasing performance for both. The results presented in the main body of this work




(a) Modified Compass World LSTM action conditioning ablation.




(b) RockSample(7, 8) LSTM action conditioning ablation.




(c) Fig. (b), but comparing only action conditioning vs no action conditioning.

Figure B.3: **Action Conditioning Ablation for LSTM**
Results are over 10 seeds for both Modified Compass World and RockSample(7, 8).

use $\delta_{hed} \doteq 5$.

### B.3.2 LSTM Action Concatenation Ablation

We conduct another small ablation study on action concatenation with the LSTM agent on both Modified Compass World and RockSample(7, 8). Results are shown in Figure B.3. Action conditioning seems to generally help the LSTM agent in both environments—this is because we are conditioning on and providing more information than the LSTM agent not conditioned on actions. We note that the degree in which this action conditioning helps varies from environment to environment. In Modified Compass World, action conditioning is vital to the performance of the LSTM

agent, whereas in RockSample(7, 8) we observe only potential marginal performance increases (potential since we have overlapping standard error bars). This reveals a further point with regards to action conditioning for RNNs: actioning conditioning will help depending on how much knowledge of the action resolves partial observability. In Modified Compass World, where the state is extremely partially observable, knowing the previous action resolves a big portion of the partial observability of the environment, whereas in RockSample, the previous action does not reveal too much about the environment state.

# Appendix C: Fishing Experimental, Algorithmic and Environment Details

## C.1 Fishing Environment Details

In this section, we describe the specifics for both Fishing environments, as well as the environment parameters for both Fishing 1 and 2, including the rates of change for currents and the rates of regeneration for rewards.

We first describe the implementation specifics of currents. For all currents in both Fishing 1 and 2, currents push you one tile towards the direction that the current is facing. In both environments, the agent is first moved in the direction which the action it takes corresponds. After moving in that direction, depending on the current that the agent lands on, it will be moved in the direction which the current is facing. For example, if the agent moves into a current facing in the same direction it just moved in, say east, then the current will bring the agent one more tile in the east direction. In this one transition, the agent will move two tiles in the eastward direction, because of the move action and the current.

As for rewards in the environment, the agent receives a reward of $+1$ for collecting a reward. If the agent is pushed by the current into a wall, it receives a reward of $-0.1$; otherwise, the agent receives a reward of 0.

As for starting position, the agent starts in the $(x, y)$ position $(5, 5)$ in both environments.

### C.1.1 Mapping and Observations

One part of the partial observability of this environment is the limited observation an agent receives at every step. The agent receives a $5 \times 5$ agent-centric observation vector at every step. All agents view an accumulated (over time) agent-centric *map* of these observation vectors, which we call the agent map. For our $11 \times 11$ grid world, this amounts to a square observation of length $11 + 11 - 1 = 21$—the dimensions beyond the length 11 of the grid world account for the agent-centric view when the agent is at the edges of the grid world. At every step, the $5 \times 5$ observable area is updated on the agent map with the given coordinates of the agent. Hence, with every additional step, since rewards and currents change stochastically over time, previously observed (but currently unobserved) missing rewards and currents on the agent map are less and less accurate as time progresses.

The agent's view is also *obstructed* by the walls in the environment. If an agent is next to a wall, then everything behind the wall is obstructed, even if the area was meant to be observable. This is true for all walls except for glass obstacles, denoted in blue in Fishing 2. These glass obstacles act as walls, except that the agent is able to see through them, unlike normal walls. Glass walls are placed here so that the agent is able to see the direction that the currents are facing within the tunnel to the reward.

The agent observes a tensor of shape $21 \times 21 \times 6$, where the last dimension indicates the channels for different aspects of the environment/observation. The agent is able to see obstacles (walls and glass walls, 1st channel), currents and their direction (next 4 channels), and finally the locations of rewards if they are present (last channel).

### C.1.2 Stochasticity in the Environment

Both rewards and currents in both environments are defined by Poisson processes, with potentially different rates corresponding to each reward and each current. We now describe these rates for both environments. All currents depicted with a single-

direction arrow denote a current that is static and does not change direction. Note for all currents, when a current is sampled to change, we sample uniformly at random from the remaining current directions, excluding the original current direction.

Besides the stochasticity from the Poisson processes, the agent also has a 0.1 probability of "slipping" for a move, where the agent takes an action, and with probability 0.1 the agent simply stays in place.

#### Fishing 1 Poisson Processes

In Fishing 1, all our stochastic processes have equal rates of regeneration or current flipping. This rate is 60—or on average, these Poisson processes will activate in expectation over after 60 steps. This larger rate (c.f. Section 3.2) is such that the agent is more incentivized to go collect other rewards, rather than staying at one particular reward and waiting for regeneration.

#### Fishing 2 Poisson Processes

In Fishing 2, rates of reward generation are all set to 50, except for $(y, x)$ coordinates $(8, 9)$, which has a rate of regeneration of 100. Note, from here onwards we will list positions as tuples of coordinates of $(x, y)$. As for our currents, we group our currents based on their reward regeneration rates (in order from left to right, top to bottom in the grid world):

10: (0, 6), (0, 7), (2, 5), (2, 6), (4, 1), (5, 7), (5, 8), (5, 10), (8, 7), (9, 7).

20: (1, 3), (1, 4), (7, 0), (7, 1), (8, 5), (9, 5), (10, 2), (10, 3), (10, 4).

30: (5, 2), (5, 3), (5, 4).

40: (0, 2), (2, 0), (2, 1), (3, 9), (3, 10), (6, 2), (6, 3), (6, 4), (9, 8), (10, 8).

## C.2 Fishing-Specific Algorithmic Details and Hyperparameters

We now detail the algorithmic details and hyperparameters swept for all agents on the Fishing environments. In both our Fishing experiments, we leverage a replay buffer

[21] for training. We list the hyperparameters swept for all our algorithms below:

For the Fishing experiments, we use a convolutional neural network to parse our agent map tensor. We use and/or sweep the following hyperparameters:

- Learning algorithm: $Sarsa(0)$
- Function approximation: Convolutional Neural Network
- Layers: 1
- Hidden units: 64
- Batch size: 64
- Optimizer: *Adam*
- Discount rate: $\gamma = 0.99$
- Environment train steps: 2M for Fishing 1, 12M for Fishing 2
- Max episode steps: 1000
- Step sizes: $[10^{-4}, 10^{-5}, 10^{-6}, 10^{-7}]$
- Buffer size: $100K$
- Evaluation frequency: 2K for Fishing 1, 10K for Fishing 2

All hyperparameter sweeps were done over 5 seeds, with the best hyperparameters decided for each algorithm based on mean discounted returns over the last 100 evaluation steps. Results in this section use *offline evaluation* returns over environment steps. Offline evaluations are conducted every *evaluation frequency* steps (as listed above). We run 5 test episodes per offline evaluation, and also average over these test episodes as well as seeds for a final average return for a given evaluation step at a certain training step. We then use these selected hyperparameters and this offline evaluation to run experiments for 30 different seeds to obtain the results presented in Figures 5.1b and 5.1d.

### C.2.1 Convolutional Neural Network Architecture

We now detail the architecture for our convolutional neural network. All our convolutional layers use a stride of 1 and no padding:

- Conv2D(output channels = 32, kernel size = 10)
- Relu activation

- Conv2D(output channels = *hidden size*, kernel size = 7)
- Relu activation
- Conv2D(output channels = *hidden size*, kernel size = 1)
- Linear layer with output $n_{actions}$

### C.2.2 Convolutional Neural Network LSTM Architecture

Our LSTM implementation is a convolutional neural network with an LSTM layer after the convolutional layers:

- Conv2D(output channels = 32, kernel size = 10)
- Relu activation
- Conv2D(output channels = *hidden size*, kernel size = 7)
- Relu activation
- Conv2D(output channels = *hidden size*, kernel size = 1)
- Relu activation
- Linear layer with output *hidden size*
- LSTM(hidden state size = *hidden size*)
- Linear layer with output $n_{actions}$

### C.2.3 Exponential Trace Implementation Details

While we describe the approach to using exponential decaying traces for the Fishing environment in Section 5.2, we go into detail here with regards to implementation details and hyperparameters swept.

Our exponential decaying traces are simply another channel in our agent map tensor, with the same size of $15 \times 15$. It is a tensor of elements in the range of $[0, 1]$, with each element denoting how long it has been since observing that particular position (where 1 denotes the agent is currently observing this area, and 0 denoting it has never observed this position).

As for the decay rates, we swept the following rates: $[1, 0.95, 0.85, 0.65]$.

### C.2.4 Recurrent Neural Network Implementation Details

With our RNN implementation, we simply use the same technique of training an LSTM with a replay buffer as in Appendix B.2.2, where we sample trajectories of length *truncation length* for T-BPTT. We swept the following truncation lengths for both Fishing environment hyperparameter sweeps: $[1, 5, 10]$.