No More Pesky Hyperparameters: Offline Hyperparameter Tuning For Reinforcement Learning

by

Archit Sakhadeo

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

Master of Science

Department of Computing Science
University of Alberta

Abstract

The performance of reinforcement learning (RL) agents is sensitive to the choice of hyperparameters. In real-world settings like robotics or industrial control systems, however, testing different hyperparameter configurations directly on the environment can be financially prohibitive, dangerous, or time consuming. We propose a new approach to tune hyperparameters from offline logs of data, to fully specify the hyperparameters for an RL agent that learns online in the real world. The approach is conceptually simple: we first learn a model of the environment from the offline data, which we call a calibration model, and then simulate learning in the calibration model using several hyperparameters. We evaluate the hyperparameters inside the calibration model based on some desirable performance criterion, and then identify promising hyperparameters for deployment. We identify several criteria to make this strategy effective, and develop an approach that satisfies these criteria. We empirically investigate the method in a variety of settings to identify when it is effective and when it fails. We demonstrate that tuning hyperparameters offline and deploying an RL agent with these hyperparameters is a more feasible problem to tackle than transferring a fixed policy learned from the offline data.

Preface

This thesis is based on a manuscript "No more pesky hyperparameters: Offline hyperparameter tuning for RL" by Han Wang*, Archit Sakhadeo*, Adam White, Xutong Zhao, James Bell, Puer (Paul) Liu, Nathan Taylor, Alona Fyshe, and Martha White. The paper is currently under review. Adam and Martha wrote and edited the main part of the paper. James set up the code infrastructure. Nathan implemented a visualization tool. Alona helped think through problems we were stuck on and gave regular feedback on the work. Han, Xutong, Paul, and I were responsible in different capacities for implementing the code, conducting the experiments, and writing the appendix in the paper.

This thesis was inspired by a real-world project on optimizing a water treatment plant using reinforcement learning (RL)†. The research direction of this thesis emerged from the need to tackle the problem of deployment of an RL agent to control this water treatment plant. Naturally, the first question that emerged regarding the problem of deployment was, "How do we select the hyperparameters of such an agent offline before deploying it on a real-world system?". This thesis‡ attempts to tackle the above question.

*Equal contribution.

†AI pilot project aims to make water treatment greener, more efficient and less expensive, https://www.ualberta.ca/folio/2020/10/ai-pilot-project-aims-to-make-water-treatment-greener-more-efficient-and-less-expensive.html

‡This thesis was written in three countries - Canada, India and Mexico.

To Aai and Baba,

for being a constant source of love and support.

Thank you for everything.

"What if your life was a sacred responsibility?

The body, a temple.

Every motive, pure.

Every glance, compassionate.

Every word, true.

Every act, right.

Every moment, holy."

- Naval Ravikant

"Any sufficiently advanced technology is indistinguishable from magic."

- Arthur C. Clarke

Acknowledgements

My journey as a graduate student has been influenced by several incredible people in both professional and personal ways. First and foremost, I would like to thank my advisors, Adam White and Alona Fyshe, for their mentorship and support throughout my graduate program. I am grateful to them for letting me explore my research interests and work on problems that excited me. Having worked closely with Adam, I now share his passion for carefully designing good experiments and a responsibility for doing good science. Through Adam's hands-on approach, I have realized the utility in thinking abstractly as well as getting in the weeds to truly understand the problem and the required solution. Though my interaction with Alona was limited, she always had a knack to provide insight and ask important questions whenever I was stuck on something. She was always a source of encouragement and was incredibly kind. I am extremely grateful to Alona for vouching for me during the graduate admissions. Second, I would like to thank Marlos Machado who was my external committee member. Marlos provided feedback and a stimulating discussion during the defense exam that helped improve the thesis. Third, I would like to thank Martha White who was almost like a third advisor to me. Martha's ability to find the "signal" in several stages of uncertainty during the course of the project was remarkable, and it has taught me to zoom out and seek clarity in similar situations. Fourth, I would like to thank Rupam Mahmood for taking a chance on me and giving me the opportunity to take a highly-selective advanced class in my first term even when I did not meet the prerequisites of having any experience in reinforcement learning (RL) or robotics. It helped me get hooked onto RL, which then strongly laid down

my research interests. Finally, I would like to thank Rich Sutton for teaching me to always separate the problem from the solution method, and this mental framework has helped me on several occasions. Rich instilled in me a new perspective of looking at artificial intelligence, and a sense that "ideas matter".

I am grateful to friends and colleagues at Amii and RLAI for providing a conducive research environment. In specific, I would like to thank Han Wang who I have pestered with hundreds of questions. Han has been incredibly helpful throughout, and I aspire to be as hardworking as Han someday. I often think, this thesis is as much hers as it is mine. I am grateful to Andrew Patterson and Han for carefully reviewing my thesis. I am also grateful to James Bell, Xutong Zhao, and Paul Liu for being awesome teammates. I am also grateful to Matthew Schlegel for spending time and discussing several RL concepts with me in the early stage of my program.

I am incredibly happy to have gotten to know Parash Rahman when we worked on a project together. Parash has been a great friend since then and we have enjoyed discussions about RL, philosophy, and life. I am also grateful to Shivam Garg and Soumyadeep Pal who were awesome roommates and friends during the two years and who helped make the isolation during the COVID-19 pandemic a lot bearable. I am going to cherish the long walks and banter with Shivam, and the conversations that happened after my return from India with Soumyadeep. I am grateful to have found a great friend in Aditya Jayaprakash who I instantly bonded with over several similar interests, coffee, food and good conversations. Aditya was the kind of person that I had been hoping to meet before I started my program. I am also grateful to have gotten to know Taivanbat (TK) Badamdorj as a great friend. TK is one of the funniest and likeable people I know, and we bonded over a very similar liking for humor.

Finally, I am incredibly fortunate to have Rathin (Nandu) Desai, Shadab Shaikh and Shubhankar Deshpande in my life. My life is significantly better because of them. The camaraderie that we share and the friendship that we have built over the years is truly special, and it has had a massive influence on who I am today as a person.

Table of Contents

List of Figures

Chapter 1

Introduction

Reinforcement learning (RL) agents are sensitive to the choice of hyperparameters that regulate speed of learning, exploration, degree of bootstrapping, amount of replay, representational capacity and so on. The vast majority of work in RL is focused on new algorithmic ideas and improving performance along some dimension: continued exploration, early learning speed, asymptotic performance, sample efficiency, or robust learning. These algorithmic and performance improvements in RL agents are almost always reported assuming near-optimal hyperparameters. The vast majority of empirical comparisons involve well-tuned implementations and reporting the best performance after a hyperparameter sweep. Reasonable progress has been made towards eliminating the need for tuning some hyperparameters by using adaptive methods [24, 28, 35, 38, 62, 66, 68]. However, recent surveys show that widely used agents employ dozens of hyperparameters and tuning is critical to their success in practice [22]. Recent experiments confirm that significant sensitivity to hyperparameters is exhibited on real robots as well [33].

The reason that domain specialization, hyperparameter sweeps, and fine-tuning are possible—and perhaps why our algorithms are so dependent on them—is because most empirical work in RL is conducted in simulation. Conducting experiments using simulators is fast and it scales with compute resources. If something goes wrong the experiments can be restarted without serious repercussions. Simulators are critical

for research because they facilitate rapid prototyping of ideas and extensive analysis. However, these features that make simulators desirable do not transfer to the real world. Running rigorous experiments in the real world can be *extremely time consuming* (slow action cycle time in the order of few seconds to few hours), *monetarily expensive* (high energy usage, expensive machinery, stock-trading bots), *might not always scale with compute resources* (difficult to deploy thousands of robots in parallel), and *might have safety related concerns* (self-driving cars). As a result, it is not possible to exhaustively sweep different hyperparameters in the real world. Often, it is not acceptable to test poor hyperparameters on a real system that could cause serious failures. In many cases, interaction with the real system is limited, or in more extreme cases only data collected from a human operator is available.

In simulators, hyperparameters are evaluated directly in the deployment environment. Whereas, in the real world, the deployment environment does not allow evaluating hyperparameters due to the above reasons. This begs the question, *"How do we select hyperparameters before deploying online RL agents in the real world?"*. To the best of our knowledge, this problem has not been tackled before. Unfortunately, simulators have allowed us to completely bypass this problem that researchers and engineers would have to face before deploying their RL agents in the wild.

It is not surprising that one of the major roadblocks to applied RL in the wild is extreme hyperparameter sensitivity [32, 33], and this is amplified by the absence of a clear strategy to select good hyperparameters.

1.1 Objective

There is an important constraint on selecting the agent hyperparameters before deploying them into the real world - the hyperparameters cannot be directly evaluated on the real system. Good hyperparameters are a function of the agent as well as the environment it is deployed on. Thus, if the agent cannot interact with the real system in any way, selecting the agent's hyperparameters is a difficult task. Fortunately, there

is an alternative avenue to evaluating algorithms without running on the real system: using previously logged data under an existing controller (human or otherwise). In applications like industrial control processes (eg: optimizing the process of water treatment) that have been historically controlled by some degree of automation and some degree of human controllers, logs of data are collected that include information like the sensor readings and the actions taken. The data are logged at the action cycle time of the controller which can be in the order of seconds to hours. These logs of data contain rich information that give a window into the underlying real system. The data logs can reveal the dynamics of the underlying environment while giving information about the state space and the action space of the agent. The data logs can thus be used as a proxy to the real system meant for deployment. Thus, these offline logs of data could instead be used to evaluate and select hyperparameters without directly interacting with the real world. Hyperparameters are general, and can even include a policy initialization that is adjusted online. We call this problem of selecting the agent hyperparameters for deployment purely from the offline data logs as the *Data2Online* problem.

This thesis seeks to answer the question:

> **Can we develop a strategy to solve the Data2Online problem - the problem of selecting the hyperparameters *offline* for the RL agent *to learn online in deployment*, when we only have access to offline data logs?**

Overall, our objective is to take a closer look at the Data2Online problem, to propose a solution strategy to this problem, and to highlight when the strategy works and when it does not.

1.2 Approach

We propose a novel strategy—to the best of our knowledge the first—to use offline data logs for selecting hyperparameters. The idea is simple: 1) we use the offline data to learn a *calibration model* (learned simulator/model of the underlying environment), 2) evaluate hyperparameters in the calibration model, and 3) report the best performing hyperparameter in the calibration model for deployment in the real world. The calibration model captures the transition dynamics of the environment, that is, it takes in the current state and action as input and outputs the next state, reward, and the termination signal. Learning online in the offline calibration model mimics learning in the environment, and so should identify or *calibrate* hyperparameters that are effective for online learning performance. The calibration model need not be a perfect simulator to be useful for identifying reasonable hyperparameters, whereas learning a transferable policy typically requires accurate models [69].

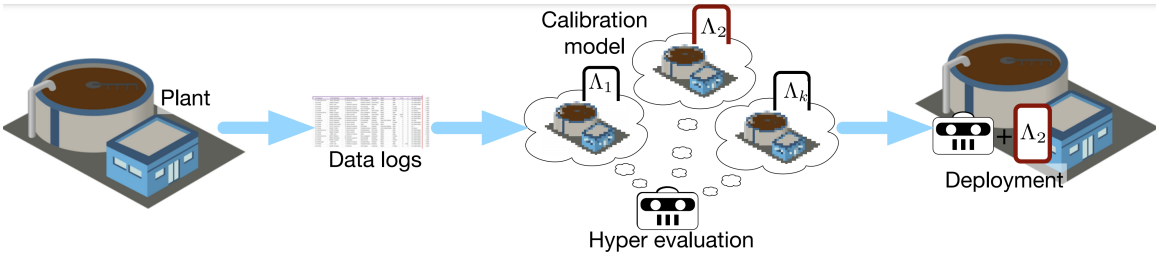

Figure 1.1: **Offline Hyperparameter Selection for Online RL.** This figure illustrates how hyperparameters can be selected offline for deploying an online RL agent. The water treatment plant resembles any real system. The controllers that operate the plant typically log the interaction data. These data logs can be used to construct a calibration model of the underlying plant. The calibration model need not resemble the plant perfectly. Hence, the calibration model is illustrated using a blurred image that approximately resembles the underlying plant. Each hyperparameter setting is denoted by Λ. Each hyperparameter setting can be evaluated in the calibration model based on some performance criterion - in this case it could be the ability to control the plant while being energy-efficient. The best performing hyperparameter is then selected for deployment in the underlying plant.

For example, consider designing a learning system for controlling a water treatment plant, given only a set of data logs visualized in Figure 1.1. We may want an agent to

control pump speeds, mixing tanks and chemical treatments to clean the water with minimal energy usage—but how do we set the learning rate and other hyperparameters of this agent? We can learn a calibration model offline from logs of data previously collected while human operators controlled the plant. From there the calibration model can be treated like any simulator to develop a learning system, including setting the hyperparameters for learning in deployment.

The calibration model allows us to still have the benefits of using a simulator but with the data actually being grounded in the real world.

1.3 Related Work

Somewhat surprisingly, the Data2Online problem has not been directly tackled before. Most work on selecting hyperparameters introduces meta-algorithms that learn hyperparameters, including work on meta-descent for stepsizes [24, 50, 66] and selecting the trace parameter [15, 35, 62]. These algorithms could be beneficial for offline hyperparameter selection, because they help reduce sensitivity to hyperparameters; but they are not a complete solution as they still have hyperparameters to tune. Other work has provided parameter-free methods that have theoretically defined formulas for hyperparameters [38]. Deriving such algorithms is important, but is typically algorithm-specific and requires time to extend to broader classes of algorithms, including new advances; it remains useful to consider how to tune hyperparameters for a problem. Finally, recent work has examined online hyperparameter selection, using off-policy learning to assess the utility of different hyperparameters in parallel [40, 55]. Otherwise, much of the work [10, 17, 25, 29, 39, 48, 49] has been focused on settings where it is feasible to obtain multiple runs under different hyperparameters—such as in simulation—with the goal to improve on simple grid search as in random search, Bayesian optimization, HyperBand, etc.

The solution to the Data2Online problem that we propose includes having offline logs of data, constructing a calibration model from these logs, and tuning the hyper-

parameters on the calibration model for the agent to learn online in the deployment environment with these hyperparameters. Following are some related topics in the literature that also tackle the problem of deployment, albeit through different approaches. Some of these assume access to offline logs of data while some assume access to a model of the environment. However, all of these approaches focus on transferring a fixed policy in the deployment environment.

1.3.1 Offline RL

The Data2Online problem setting differs from the standard offline or batch RL setting because the goal is to select *hyperparameters* offline for the agent to *learn online in deployment*, as opposed to learning a *policy* offline. Typically, in offline RL, a policy is learned on the batch of data using a method like Fitted Q Iteration (FQI) [16, 18, 45], and the resulting fixed policy is deployed. Our setting is less stringent, as the policy is learned and continually adapts during deployment. Intuitively, selecting just the hyperparameters for further online learning should not suffer from the same hardness problems as offline RL (see [58] for hardness results), because the agent has the opportunity to gather more data online and adjust its policy.

An inherent problem when using offline data in RL is that of *insufficient coverage*. If the data does not cover a part of the space, then the agent cannot learn about that part of the space. For hyperparameter selection, this is not as problematic since the agent is not learning a policy for re-use. Rather, the agent would simply select hyperparameters for a slightly different environment. The sufficient similarity in terms of the input types, reward magnitudes and transition dynamics means the agent may well still select reasonable hyperparameters for learning online.

Even in the offline batch RL setting, the hyperparameters of the learner must be set and most approaches either unrealistically use the real environment to do so [63] or use the action values learned from the batch to choose amongst settings [37] with mixed success. Offline RL involves learning from a dataset, but their hyperparameter

selection approaches are quite different due to the fact that they deploy fixed policies. Yang et al. (2020) [67] introduce the Offline Policy Selection problem and use an offline RL algorithm to learn and evaluate several different policies corresponding to different hyperparameter settings. The key difference is those hyperparameters will never be deployed in an online learning system, as we do in this thesis. Some offline RL work examines learning from data and adapting the policy online [3], including work that alternates between data collection and high confidence policy evaluation [12, 13]. Our problem is complementary to these, as a strategy is needed to select their hyperparameters.

1.3.2 Sim2Real

In Sim2Real the objective is to construct a high fidelity simulator of the deployment setting, then learn a policy in the simulator and transfer the fixed policy, and—in some cases—continue to fine tune in deployment. We focus on learning the calibration model from real collected data, whereas in Sim2Real the main activity is designing and iteratively improving the simulator itself [41] such that it closely resembles the underlying environment. Again, however, approaches for Data2Online can be seen as complementary and can highlight how to use the simulator developed in Sim2Real to pick hyperparameters for fine-tuning. It could suggest design decisions in the developed simulator for which hyperparameter selection for online learning is more effective. If the deployment environment is too complicated to directly model (e.g. unknown physical models), or if the simulator is expensive to sample, then our approach could be employed. As described in the above section, since in Data2Online we do not extract a policy, there are no strict requirements of having high coverage or equivalently an accurate model of the environment [69]. Thus, if the logged data are too limited to build an accurate model for Sim2Real, our approach could be used to select hyperparameters.

1.3.3 Domain Adaptation

Domain adaptation in RL involves learning on a set of source tasks to transfer to a target task. The most common goal has been to enable zero-shot transfer, where the learned policy is fixed and deployed in the target task [23, 65]. Our problem has some similarity to domain adaptation in that we can think of learning in the calibration model to identify hyperparameters as the source task, and then transferring the selected hyperparameters in the deployment environment as the target task. Domain adaptation, however, is importantly different than our Data2Online problem: (a) in our setting the training in the source task (tuning hyperparameters) uses a *learned* calibration model not a real environment and we need a mechanism to learn that model, (b) the relationship between our source and target is different than the typical relationship in domain adaptation, and (c) our goal is to select and transfer hyperparameters, not learn and transfer policies.

1.3.4 Learning From Demonstration and Imitation Learning

Learning from demonstration (LfD) and imitation learning involve attempting to mimic or extract a policy at least as good as a demonstrator. If the agent is learning to imitate online, then it is unrealistic to assume the demonstrator would generate enough training data required to facilitate hyperparameter sweeps. If the learner's objective is to imitate from a dataset, then this is exactly the problem studied in this thesis. Unfortunately, hyperparameter tuning in LfD is usually not addressed; instead it is common to use explicit sweeps [6, 8, 20, 36] or manual, task-specific tuning [19]. According to Ravichandar et al. (2020) [44], hyperparameter tuning is a major obstacle to the deployment of LfD methods.

1.4 Contributions

Thesis Statement: Offline data can be used to effectively select hyperparameters offline to be then transferred in the deployment environment for online learning, and transferring hyperparameters is a more feasible problem to tackle than transferring fixed policies.

This thesis contains three contributions that support the above statement.

1. **Propose a new strategy to tackle the Data2Online problem and outline conditions on the solution strategy. (Chapter 3)**

   We first introduce our offline hyperparameter strategy to solve the Data2Online problem using calibration models. We then outline conditions on the calibration model and learning agents for this strategy to be effective. We then propose a k-nearest neighbors (KNN) based model to obtain stable calibration models. We finally propose using a distance metric based on the transition dynamics by using the Laplacian representation [64] to more accurately find the nearest neighbors.

2. **Empirically demonstrate the effectiveness of our solution strategy under various scenarios, highlight a failure case, and demonstrate why transferring hyperparameters is a more feasible problem to tackle than transferring a fixed policy. (Section 5.1)**

   We then investigate the approach in three environments with different types of learning dynamics, using two different learning agents, under different offline data collection policies, under different deployment situations, and with ablations on the key components of our proposed calibration model. We demonstrate the effectiveness of our approach under many of these conditions and highlight one case where it fails to perform. We compare our approach against different

baseline strategies. In this, we evaluate the ability of our approach to *select* good hyperparameters among many other hyperparameters using grid search. We also make a case for transferring hyperparameters as against transferring a fixed policy from offline logs of data.

3. **Demonstrate that the calibration model works well with hyperparameter optimization techniques smarter than grid search, and the performance of the calibration model can further be improved by *fine-tuning* the hyperparameters instead of *selecting* among a set of hyperparameters. (Section 5.2)**

   We then develop a more efficient method to *fine-tune* hyperparameters in the calibration model, based on the cross-entropy method for optimization. We show that this addition can significantly improve over a hyperparameter grid search, as well as reducing the burden on the designers (engineers/researchers) to design sets of hyperparameters to sweep.

1.5 Thesis Layout

The thesis contains 6 chapters.

Chapter 2 covers background material needed to understand the remaining chapters. This chapter can be skipped by readers familiar with the reinforcement learning literature.

Chapter 3 covers the main description of the solution strategy that we propose to tackle the Data2Online problem. This chapter discusses in detail how we construct the calibration model, evaluate the hyperparameters on the calibration model, and highlights some conditions on the solution strategy.

Chapter 4 covers the details about the experimental setup. This chapter discusses the specifications of the environments, the learning agents that we use, the baseline agents we compare against, different data collection schemes we use in our experiments,

and how we evaluate the performance of our solution strategy.

Chapter 5 discusses the main experiments we perform to demonstrate the effectiveness of our approach under different variations of the calibration model, data collection schemes, deployment environments, and learning agents. This chapter also describes a situation where our strategy fails to select good hyperparameters. This chapter demonstrates how our strategy can *select* good hyperparameters offline from a set of several hyperparameters using grid search. This chapter further discusses how the calibration model can be used with smarter hyperparameter optimization techniques instead of grid search. It introduces a strategy based on the cross entropy method (CEM) to *fine-tune* the hyperparameters. We then demonstrate the benefits of the CEM based fine-tuning approach over grid search, and show how it can improve the performance of our strategy.

Finally, the thesis ends with chapter 6 discussing the conclusions from the thesis and future direction of work.

1.6 Summary

This chapter introduced the context the thesis is set in. This chapter motivated the problem we tackle. It described the objective of the thesis and a high-level approach we follow to tackle the problem. The chapter discussed how our work fits in the broader RL literature and how our work can be thought of as complementary to many other existing domains. The chapter then lists the contributions of this thesis. The chapter ends with the layout of the thesis.

Chapter 2

Background

We now discuss the background material needed to understand the concepts used later in the thesis. Readers who are knowledgeable in topics like the MDP framework, value functions, tile coding, online control algorithms like Expected Sarsa and Actor-Critic, and offline RL can safely skip this chapter.

2.1 Learning as Reward Maximization

Reinforcement learning (RL) is a problem formulation to learning from interaction [52]. An *agent* learns to make decisions through interaction with its *environment* to achieve a desired goal. Formulating goals as maximization of a special signal, called the *reward*, is one of the central ideas of reinforcement learning.

2.2 The MDP Framework

We formulate the reinforcement learning problem as a Markov Decision Process (MDP). An MDP is described by the tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$. $\mathcal{S}$ is the state space and $\mathcal{A}$ the action space. The transition probability $\mathcal{P} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ describes the probability of transitioning to the next state from a given state and action. $\mathcal{R} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ is the reward function that emits a scalar reward from a given state, action, and on transitioning to the next state. $\gamma : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ is the discount function [61].

Figure 2.1 shows the agent-environment interaction in an MDP. On each discrete

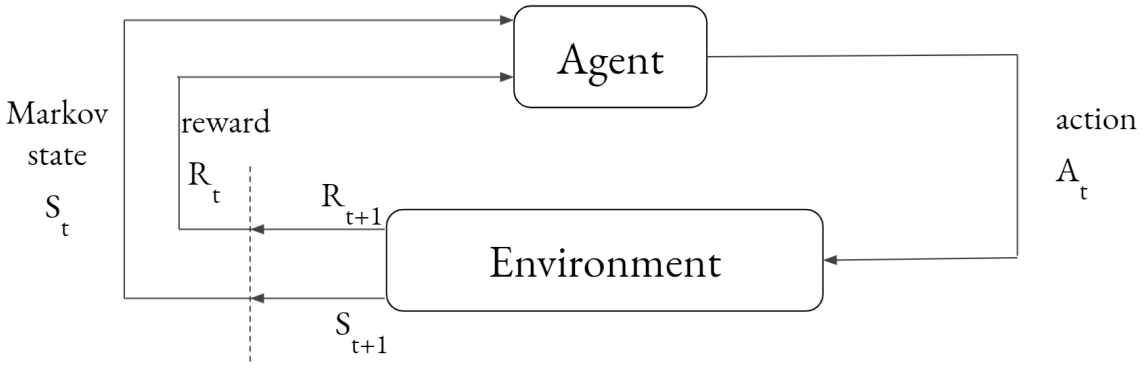

Figure 2.1: **Agent - environment interaction in a Markov Decision Process**

timestep t, the agent receives some representation of the environment's *state* $S_t \in \mathcal{S}$, and on that basis selects an *action* $A_t \in \mathcal{A}$. One timestep later, the agent is transitioned to a *new state* $S_{t+1} \in \mathcal{S}$ and receives a scalar *reward* $R_{t+1} \in \mathcal{R} \subset \mathbb{R}$. This interaction between the agent and the environment gives rise to a stream of experience: ... $S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1}, R_{t+2}, S_{t+2}$, ...

The agent selects actions at each timestep based on a policy $\pi : \mathcal{S} \times \mathcal{A} \to [0, 1]$. The policy is a function that maps the states to probabilities of selecting each possible action in the states. The policy describes the agent's behavior. The agent's objective is to find an optimal policy π_* that maximizes the sum of future rewards. We formally define π_* in Section 2.3.

An optimal policy π_* seeks to maximize the return G_t which is the sum of future rewards. In general, we seek to maximize the discounted return $G_t \doteq R_{t+1} + \gamma_{t+1} G_{t+1}$. The scalar $\gamma_{t+1} \in [0, 1]$ depends on the transition (S_t, A_t, S_{t+1}) [61]. For continuing problems, the discount may simply be a constant less than 1. For episodic problems the discount might be 1 during the episode, and become zero when (S_t, A_t) leads to termination. A constant discount rate γ signifies how far-sighted the agent is: $\gamma = 0$ indicates that the agent cares only about the immediate reward on the next step while $\gamma = 1$ indicates that the agent has an infinite horizon and cares about all future rewards equally. The effect of $0 < \gamma < 1$ is somewhere in between these extremes, with the importance of each reward geometrically decreasing further along the trajectory.

Common approaches to learn a policy that estimates π_* are Q-learning [59, 60] and Expected Sarsa [47, 52], which approximate the action-values—the expected return from a given state and action—and Actor-Critic methods [7, 27, 53] that learn a parameterized policy (also see [52]).

2.3 Value Functions

Another way of describing the return (with constant discount function γ) is as follows,

$$\begin{aligned} G_t &\doteq R_{t+1} + \gamma G_{t+1}, \\ G_t &= R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ..., \\ G_t &= \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}. \end{aligned}$$

The return at any given timestep is affected by the future rewards, that is, the return is affected by various future sources of stochasticity — the agent's policy, the transition dynamics function, the reward function. Hence, the expected return, averaged over all possible future sources of stochasticity, is a more useful measure than just the return. The value function encapsulates precisely this.

Value functions lie at the core of many RL solution methods. Value functions are a way to summarize the consequences of future behavior into a single number for a given state. The state-value function $v_\pi(s)$ is the value of a state s under a policy π. It tells how much expected return the agent can receive if it follows policy π from state s. It is described as follows:

$$\begin{aligned} v_\pi(s) &\doteq \mathbb{E}_\pi \left[G_t \,\middle|\, S_t = s \right], \\ &= \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\middle|\, S_t = s \right]. \end{aligned}$$

The action-value function $q_\pi(s, a)$ is the value of a state s and action a under a policy π. It tells how much expected return the agent can receive if it takes action a in state

s and then follows policy π. It is described as follows:

$$\begin{aligned} q_\pi(s,a) &\doteq \mathbb{E}_\pi \left[G_t \;\middle|\; S_t = s, A_t = a \right], \\ &= \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \;\middle|\; S_t = s, A_t = a \right]. \end{aligned}$$

There is a unique state-value function [2], called the *optimal* state-value function v_* that maximizes the value over all states:

$$v_*(s) \doteq \max_\pi v_\pi(s),$$

for all $s \in \mathcal{S}$. All policies that share the same optimal state-value function are optimal policies. An optimal policy is given as:

$$\pi_* \doteq \arg\max_\pi v_\pi(s),$$

for all $s \in \mathcal{S}$. Optimal policies also share the same optimal action-value function q_*, which is given as:

$$q_*(s,a) \doteq \max_\pi q_\pi(s,a),$$

for all $s \in \mathcal{S}$ and all $a \in \mathcal{A}$. Value-based RL methods typically estimate the optimal action-value function as a means to an optimal policy. A deterministic policy $\pi : \mathcal{S} \to \mathcal{A}$ that is greedy with respect to q_* in all states is an optimal policy:

$$\pi_*(s) \doteq \arg\max_a q_*(s,a),$$

for all $s \in \mathcal{S}$.

2.4 Tile Coding

Tile coding [4, 5] is a method for feature construction that takes in some state space and projects it into a high-dimensional space which allows for better generalization

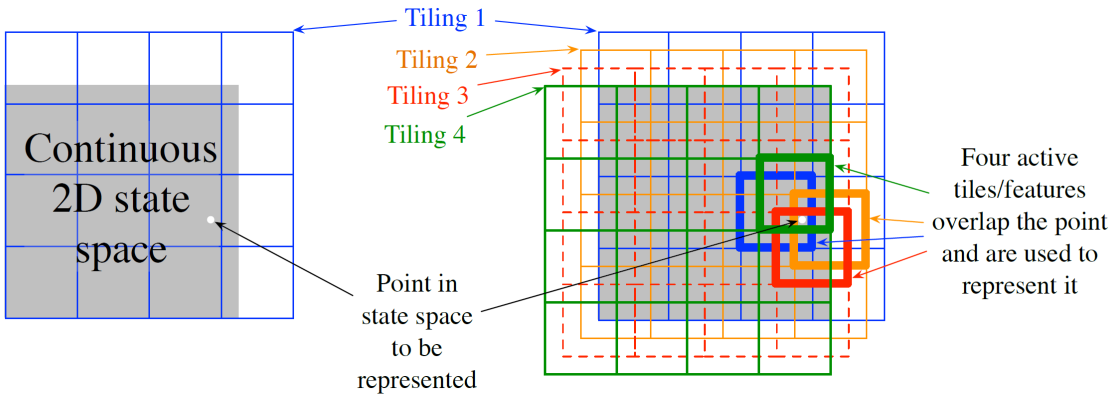

Figure 2.2: **Overlapping grid tilings on a 2-dimensional state space:** Figure re-used from Sutton & Barto (2018) [52].

between similar states and better discrimination between dissimilar states. It is a fixed representation method whose features are not learned as in neural networks. A specific point in the underlying state space will always have the exact same feature representation under tile coding even when learning progresses.

Tile coding gives us a method to deal with multi-dimensional continuous state spaces by discretizing them using grid-like data structures. For the sake of simplicity, let's assume a 2-dimensional continuous state space as in Figure 2.2. Tile coding partitions the state space using overlapping grids called *tilings*. Each tiling divides the state space into smaller squares called *tiles*. Let's assume a case where we have a single tiling with multiple tiles in it. All points inside any given tile have the same feature representation. This allows for state aggregation and generalization within that specific tile. It also helps to discriminate points that lie in two different tiles by assigning them different features. However, using one tiling cannot discriminate between points that lie inside the same tile. This limitation is overcome by having multiple overlapping tilings that are offset from each other by a small amount. Placing multiple overlapping tilings in this manner leads to tiles in each tiling to intersect with different tiles in other tilings. Thus, a point in the state space can now occupy different intersecting tiles across multiple tilings. This leads to finer discrimination among points that would have never been discriminated, as they would have existed

inside the same tile in the case of a single tiling. Larger tiles result in increased generalization, whereas having a large number of tilings on top of each other (offset by some amount) improves the discriminative ability of tile coding. The benefits of tile coding are two fold - it allows us to have better discrimination (more fine grained control, less bias) without loss of generalization.

Tile coding creates a representation for each point in space by concatenating the representation for each tiling. The representation for each tiling consists of a one hot vector that has a one for the tile that the point falls within and zero otherwise. Thus, it returns an n-hot representation where exactly one tile from every tiling is active. Tile coding thus creates a non-linear representation with binary features. The resulting feature vector can then be used with a parametric (e.g., linear or non-linear) function approximator to estimate the value function. From this point of view, the tile coded feature vector is a very high dimensional sparse representation of the low-dimensional dense state space. Sparse representations have been shown to somewhat reduce catastrophic forgetting [21, 30].

Tile coding can represent a wide range of functions, but its utility does not end there. Since grids are uniform, it is easy to compute which cell the current point (state) is in. The binary sparse feature representation only has a limited number of active tiles, which makes aggregating the results much faster than a dense representation. Due to its computational efficiency, tile coding can be used to quickly run preliminary experiments in low dimensional environments. However, as the number of dimensions grows, the number of required tiles grows exponentially. As a result, it can be necessary to tile input dimensions separately. Whether or not input dimensions can be treated independently depends on the specific problem. In this work, we tile the state dimensions separately and also pair-wise and then concatenate the resulting feature representation vectors.

A clear limitation of tile coding is that the features are non-adaptive. Thus, tile coding may not give the best feature representation for some given input space if the

task changes over time.

2.5 Expected Sarsa

Expected Sarsa is an online value-based RL algorithm which is used to estimate near-optimal action values and policies. We use the on-policy version of Expected Sarsa in this dissertation. In on-policy learning, the behavior policy (policy that is exploratory and used to generate behavior) and the target policy (policy that is learned about and that becomes the optimal policy) are same. We use the *Expected Sarsa(λ)* algorithm from Sutton & Barto (2018) [52]. We also tile code the state at each timestep to get a binary feature vector. For any incoming state S_t, tile coding the state gives a feature vector $\phi(S_t)$. We use these feature vectors, the action A_t, and a linear function approximator to estimate the action-values $\hat{q}$. The function approximator is parameterized by the vector $\mathbf{w} \in \mathbb{R}^d$. We update the action values using the Adam optimizer [28]. The update equations are given as follows:

$$
\begin{aligned}
\mathbf{g} &= -\delta_t \, \mathbf{z_t}, \\
\mathbf{m} &\leftarrow \beta_1 \, \mathbf{m} + (1 - \beta_1) \, \mathbf{g}, \\
\mathbf{v} &\leftarrow \beta_2 \, \mathbf{v} + (1 - \beta_2) \, (\mathbf{g} \odot \mathbf{g}), \\
\hat{\mathbf{m}} &= \frac{\mathbf{m}}{(1 - {\beta_1}^k)}, \\
\hat{\mathbf{v}} &= \frac{\mathbf{v}}{(1 - {\beta_2}^k)}, \\
\mathbf{w_{t+1}} &\leftarrow \mathbf{w_t} - \alpha \;\; \hat{\mathbf{m}} \odot \frac{1}{\sqrt{\hat{\mathbf{v}}} + e},
\end{aligned}
\tag{2.1}
$$

where $\odot$ is the operator for element-wise multiplication. Here α is the stepsize to update the parameters, $\beta_1 = 0.9$, $\beta_2 = 0.999$, $e = 10^{-8}$, $\mathbf{m} = \mathbf{0^d}$, $\mathbf{v} = \mathbf{0^d}$.

Here, $\mathbf{z}$ is defined as follows:

$$\mathbf{z_t} \doteq \gamma\, \lambda\, \mathbf{z_{t-1}} + \nabla \hat{q}\Big(\phi(S_t), A_t, \mathbf{w_t}\Big),$$
$$\mathbf{z_{-1}} \doteq \mathbf{0^d},$$

where λ is the eligibility trace parameter. Here, the TD error δ_t is given by:

$$\delta_t = R_{t+1} + \gamma \left[\sum_a \pi\Big(a \,\Big|\, \phi(S_{t+1}), \mathbf{w_t}\Big)\, .\, \hat{q}\Big(\phi(S_{t+1}), a, \mathbf{w_t}\Big)\right] - \hat{q}\Big(\phi(S_t), A_t, \mathbf{w_t}\Big),$$

Equation 2.1 is the semi-gradient Expected Sarsa update using linear function approximation, tile-coded features, and Adam optimizer.

To keep the policy stochastic and exploratory, we assume the policy π to have a softmax probability distribution over the action values. Thus, at timestep t,

$$\pi_t\Big(a \,\Big|\, \phi(S_t), \mathbf{w_t}\Big) = \frac{\exp\left(\dfrac{\hat{q}\Big(\phi(S_t), a, \mathbf{w_t}\Big)}{\tau}\right)}{\displaystyle\sum_b \exp\left(\frac{\hat{q}\Big(\phi(S_t), b, \mathbf{w_t}\Big)}{\tau}\right)},$$

$\forall\ b \in \mathcal{A}$, where τ is the temperature of the softmax policy.

2.6 Actor-Critic

Actor-Critic is an online policy gradient method that estimates π_*. Policy gradient methods directly learn a *parameterized policy* that can select actions without consulting the value function as in value-based methods. Actor-Critic separately learns a parameterized policy (*actor*) and a parameterized state-value function (*critic*). We use the One-step Actor-Critic (episodic) algorithm from Sutton & Barto (2018) [52]. The actor learns a policy π parameterized by $\boldsymbol{\theta} \in \mathbb{R}^d$ that is used for choosing an action A_t at each timestep, while the critic estimates the state-value function $\hat{v}$ parameterized by $\mathbf{w} \in \mathbb{R}^d$ to assess the policy's action selection online. For any incoming state S_t,

tile coding the state gives a feature vector $\phi(S_t)$. We update the actor and the critic using the following equations:

$$\mathbf{w_{t+1}} \leftarrow \mathbf{w_t} + \alpha^{\mathbf{w}} \delta_t \nabla \hat{v}\Big(\phi(S_t), \mathbf{w_t}\Big),$$
$$\boldsymbol{\theta_{t+1}} \leftarrow \boldsymbol{\theta_t} + \alpha^{\boldsymbol{\theta}} \gamma^t \delta_t \nabla \ln \pi\Big(A_t \;\Big|\; \phi(S_t), \boldsymbol{\theta_t}\Big),$$

where $\alpha^{\mathbf{w}}$ and $\alpha^{\boldsymbol{\theta}}$ are the stepsizes to update the critic and the actor respectively. Here, the TD error δ_t is given by:

$$\delta_t = R_{t+1} + \gamma\, \hat{v}\Big(\phi(S_{t+1}), \mathbf{w_t}\Big) - \hat{v}\Big(\phi(S_t), \mathbf{w_t}\Big),$$

We do the update using the SGD optimizer. To keep the policy stochastic and exploratory, we assume the policy π to have a softmax probability distribution over the action preferences. The action preferences are obtained using linear function approximation with tile-coded features. The action preferences $\hat{p}$ are parameterized by $\boldsymbol{\theta}$. Thus, at timestep t,

$$\pi_t\Big(a \;\Big|\; \phi(S_t), \boldsymbol{\theta_t}\Big) = \frac{\exp\Big(\hat{p}\Big(\phi(S_t), a, \boldsymbol{\theta_t}\Big)\Big)}{\sum_b \exp\Big(\hat{p}\Big(\phi(S_t), b, \boldsymbol{\theta_t}\Big)\Big)},$$

$\forall\ b \in \mathcal{A}$.

2.7 Fitted Q Iteration

Fitted Q Iteration (FQI) [16] is a classical batch RL algorithm. For a given log of data $\{(S_t, A_t, R_{t+1}, S_{t+1}, \gamma_{t+1})\}_{i=1}^{n_{\text{data}}}$, FQI tries to extract a Q function that estimates q_*. Greedifying with respect to this extracted Q function gives us a policy that estimates π_*. FQI learns the Q function in a supervised learning manner by learning on a randomly sampled mini-batch of data, thus breaking any temporal correlation in sequential experiences. We use Regularized Fitted Q Iteration (RFQI) [18] in this thesis. The detailed training algorithm is described in Algorithm 1.

Algorithm 1 Regularized Fitted Q Iteration

Input:

α: the learning rate

$\beta_1 = 0.9, \beta_2 = 0.999$: exponential decay rates for moment estimates

$e = 10^{-8}$: term added to the denominator to improve numerical stability

n_{batch}: mini-batch size

$\hat{q}_{\mathbf{w}}$: randomly initialized action-value function parametrized by $\mathbf{w}$

$\mathcal{D}$: offline dataset

K: number of training iterations

T: sync period

$\mathbf{m} = \mathbf{0}$

$\mathbf{v} = \mathbf{0}$

for iteration $k = 1, \ldots, K$ **do**

  if $(k \mod T) == 0$ **then**

    Sync target parameters: $\mathbf{w}' \leftarrow \mathbf{w}$

  Sample random mini-batch $\{(S_i, A_i, R_i, S_i', \gamma_i)\}_{i=1}^{n_{\text{batch}}}$ from $\mathcal{D}$

  (Adam optimizer update:)

    Compute the gradient $\mathbf{g}$ according to equation 2.2

    $\mathbf{m} \leftarrow \beta_1 \mathbf{m} + (1 - \beta_1)\mathbf{g}$

    $\mathbf{v} \leftarrow \beta_2 \mathbf{v} + (1 - \beta_2)(\mathbf{g} \odot \mathbf{g})$

    $\hat{\mathbf{m}} = \mathbf{m}/(1 - {\beta_1}^k)$

    $\hat{\mathbf{v}} = \mathbf{v}/(1 - {\beta_2}^k)$

    $\mathbf{w} \leftarrow \mathbf{w} - \alpha \hat{\mathbf{m}} \odot \dfrac{1}{\sqrt{\hat{\mathbf{v}}} + e}$

Return: $\hat{q}_{\mathbf{w}}$

At the k^{th} iteration, we randomly sample transitions $\{(S_i, A_i, R_i, S_i', \gamma_i)\}_{i=1}^{n_{\text{batch}}}$ from the data log. For any incoming state S, tile coding the state gives a feature vector $\phi(S)$. RFQI then minimizes the regularized mean squared temporal-difference error (MSTDE) on this mini-batch as given below:

$$L_k(\mathbf{w}) = \sum_{i}^{n_{\text{batch}}} \|y_i - \hat{q}_{\mathbf{w}}\Big(\phi(S_i), A_i\Big)\|^2 + \lambda_f \text{Pen}(\hat{q}_{\mathbf{w}}).$$

Here $\hat{q}_{\mathbf{w}}$ is the action value estimate parameterized by $\mathbf{w}$, $y_i = R_i + \gamma_i \max_a \hat{q}_{\mathbf{w}'}\Big(\phi(S_i'), a\Big)$ is the target for transition i, $\mathbf{w}'$ is the fixed target parameters, $\text{Pen}(\hat{q}_{\mathbf{w}})$ is a penalty term, and λ_f is the regularization coefficient. The gradient of the loss is obtained by differentiating with respect to the weight parameter $\mathbf{w}$:

$$\nabla_{\mathbf{w}} L_k(\mathbf{w}) = \sum_{i}^{n_{\text{batch}}} \left[\left(\hat{q}_{\mathbf{w}}\Big(\phi(S_i), A_i\Big) - y_i \right) \nabla_{\mathbf{w}}\, \hat{q}_{\mathbf{w}}\Big(\phi(S_i), A_i\Big) \right] + \lambda_f \nabla_{\mathbf{w}}\, \text{Pen}(\hat{q}_{\mathbf{w}}) \quad (2.2)$$

We set the penalty to squared L2 norm of the weights $\text{Pen}(\hat{q}_{\mathbf{w}}) = \|\mathbf{w}\|_2^2$. We update the parameters using the Adam optimizer. The action values are obtained by using a linear function approximator parameterized by $\mathbf{w}$ along with tile coded features and the action.

2.8 Offline Logs of Data

We assume that the agent has access to an offline log of data that it can use to initialize hyperparameters before learning online. This log consists of n_{data} tuples of experience $\mathcal{D} = \{(S_t, A_t, R_{t+1}, S_{t+1}, \gamma_{t+1})\}_{i=1}^{n_{\text{data}}}$. This log of data could have been historically generated by interaction in the environment using a human operator or engineered semi-optimal controller(s). If no historical logs of data are available, we can always deploy some controller to collect data logs before building our calibration model which we describe in the next chapter.

2.9 Summary

This chapter covered the background material needed to understand the concepts that come later on in this thesis. This chapter described the formulation of learning as reward maximization and the formulation of the reinforcement learning problem as a Markov Decision Process. The chapter gave a detailed background in tile coding which is a method for feature construction. The chapter further described online RL control algorithms like Expected Sarsa and Actor-Critic, while laying out their update equations. Another offline RL algorithm—Fitted Q Iteration— is described with its update equations. The chapter concluded with describing what the offline data logs consist of and how to collect them.

Chapter 3

Data2Online Using Calibration Models

As described in Section 1.2, calibration models are a key component of our solution strategy to the Data2Online problem. In this section, we introduce the idea of calibration models and how they can be used for hyperparameter selection. We first discuss how to use the calibration model to select hyperparameters before describing a way to learn the calibration model. We then discuss certain design criteria on the calibration model and the learning agent that make this strategy more appropriate. We finally propose a specific approach to obtain (or learn) a calibration model, from a batch of offline data, that respects the design criteria.

3.1 Using Calibration Models to Select Hyperparameters Offline

A calibration model is learned from an offline batch of data collected from some environment. It simulates the transition dynamics of the underlying data. Thus it is a learned simulator of the underlying environment. For a given state and action, it outputs the next state, reward, and termination signal. By doing this it can simulate one-step transitions, and a long rollout of such transitions gives a long trajectory of interaction. In Data2Online, the agent needs to reason about online learning behavior using only a static offline dataset. Such a long trajectory of interaction allows us to

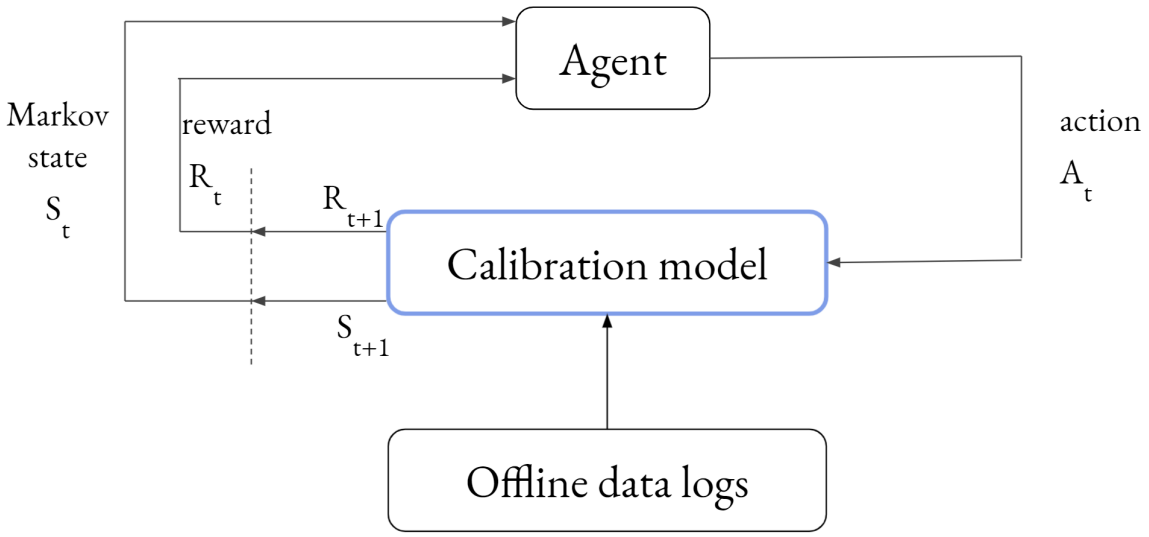

Figure 3.1: **Agent - Calibration Model interaction:** This figure is similar to Figure 2.1 with the difference being that the calibration model has replaced the environment. The blue thick border represents the calibration model. The internal details of the calibration model are described in the later sections.

simulate online learning in the calibration model. Figure 3.1 illustrates the interaction between the agent and the calibration model. It shows how the calibration model is constructed from the offline data logs. Once the calibration model is constructed, the agent can interact with it exactly how it interacts with the underlying environment as was illustrated in Figure 2.1. Thus by testing each hyperparameter setting in the calibration model, we can measure the online learning performance (eg: total reward obtained in the interaction) for each hyperparameter. Finally we can select the hyperparameter with the best online learning performance in the calibration model. We can then deploy this selected hyperparameter in the deployment environment without any remaining hyperparameters to tune. Thus, the calibration model can be used to specify (or calibrate) hyperparameters in the agent. Algorithm 2 describes this high level idea behind hyperparameter selection with calibration models.

Algorithm 3 gives a detailed description about the interaction between the agent and the calibration model, and gives the pseudocode for evaluating the online learning performance of the agent in its environment (here the calibration model). The interaction between the agent and the calibration model begins with the agent receiving

Algorithm 2 Hyperparameter Selection with Calibration Models

Input:
$\mathbf{\Lambda}$: hyperparameter set for learner *Agent*
$\mathcal{D}$: the offline log data
n_{steps}: number of interactions or steps
n_{runs}: number of runs

1: Train calibration model $\mathcal{C}$ with $\mathcal{D}$
2: **for** λ in $\mathbf{\Lambda}$ **do**
3: Perf[λ] = **AgentPerfInEnv**($\mathcal{C}$, $Agent(\lambda)$, n_{steps}, n_{runs})

Return: $\arg\max_{\lambda \in \mathbf{\Lambda}}$ Perf[λ]

a start state from the calibration model. The agent's policy then selects some action based on this state. The calibration model then transitions the agent to the next state and returns some scalar reward and a termination signal [1]. This interaction goes on until the termination signal is true, which happens if the agent succeeds (or fails) in its environment. The interaction can also be cut-off artificially. In our experiments, we make the decision to let the agent experience at least 30 episodes inside the calibration model in a given run for the agent to learn something useful. Each hyperparameter $\lambda \in \Lambda$ gets to learn online in the calibration model for a maximum of n_{steps} steps. Since the evaluation scheme guarantees at least 30 episodes, each episode can have a maximum of $\frac{n_{\text{steps}}}{30}$ steps. For each hyperparameter, we average the online learning performance across n_{runs} runs in the calibration model and report this number. In our case, we measure the average return observed in n_{steps} steps of interaction across n_{runs} runs as the online learning performance for a given hyperparameter.

Many components in this approach are modular and can be swapped with other choices. For example, instead of average return during learning (online performance), it might be more desirable to optimize the hyperparameters to find the best policy after a budget of steps. This would make sense if in deployment cumulative reward

[1] The exact details of the **EnvStep** function can be safely ignored for now as they become apparent in Section 3.3. At a high level, the function takes in the agent's action as input, and returns the next state, reward, and termination signal.

Algorithm 3 AgentPerfInEnv

Input: $\mathcal{C}$: the calibration model, *Agent*: the learner, n_{steps}: number of steps, n_{runs}: number of runs

$n_{\text{cutoff}} \leftarrow n_{\text{steps}}/30$ ▷ Ensure at least 30 episodes
ReturnsAcrossRuns = [ ]
for $i = 1 \dots n_{\text{runs}}$ **do**
  $t \leftarrow 0, Ret \leftarrow 0, Returns = [\ ]$
  $state \leftarrow$ **EnvStart**()
  for $j = 1 \dots n_{\text{steps}}$ **do**
    $action \leftarrow Agent(state)$
    $nextState, r, term =$ **EnvStep**$(action)$
    $Ret \leftarrow Ret + \gamma^t r$
    $state \leftarrow nextState$
    $t \leftarrow t + 1$
    if *term* is true or $t \geq n_{\text{cutoff}}$ **then**
      Append *Ret* to *Returns*
      $state \leftarrow$ **EnvStart**()
      $t \leftarrow 0, Ret \leftarrow 0$
  if last return $Ret > average(Returns)$ **then**
    append *Ret* to *Returns* ▷ Ensure last episode is not cut-off early
  $ReturnsAcrossRuns[i] \leftarrow average(Returns)$

Return: $average(ReturnsAcrossRuns)$

Algorithm 4 EnvStart (CalibrationModel)

Input: $\mathcal{D}$: the offline dataset

Search s from the set of start states in $\mathcal{D}$

Return: s

during learning was not important. We might also want a more robust agent, and instead of average return, we may use median return. Finally, grid search can be replaced with a more efficient hyperparameter selection method; we demonstrate this in Chapter 5.2. We can also make this hyperparameter search more robust to error in the calibration model by obtaining performance across an ensemble of calibration models. This involves using $n_{\text{ensembles}}$ random subsets of the log data, say by dropping at random 10% of transitions, and training $n_{\text{ensembles}}$ calibration models. The hyperparameter performance can either be averaged across these models, or a more risk-averse criterion could be used like worst-case performance. Using an average

Algorithm 5 EnvStep (CalibrationModel)

Input: a_t: the action chosen by the agent
$Trees$: KD-trees
k: number of nearest neighbors

1: $\phi \leftarrow \psi(s_t)$
2: $\{(\phi, s', r, T, d)\}_1^k \leftarrow$ **KDTreeSearch**$(\phi, a, Trees, k)$
3: $p[i] \leftarrow 1 - \frac{d_i}{\sum_{j \in [1,k]} d_j}, \forall i \in [1, k]$
4: Sample $(\phi, s', r, T, d)_c$ according to probability p
5: $s_t \leftarrow s'_c$

Return: s'_c, r_c, T_c

across models is like using a set of source environments to select hyperparameters—rather than a single source—and so could improve transfer to the real environment. In this work, we examine only the simplest form of this Data2Online strategy and use only one calibration model, and leave these natural additions to future work.

3.2 Criteria for Designing Calibration Models

The previous section described how the calibration models can be used to simulate online learning and how good hyperparameters can be selected based on their online learning performance. In this section, we highlight four key desirable criteria for designing the calibration model and selecting agents for which Data2Online should be effective. The criteria are described as follows:

1) Stability and self-correction under model iteration

A *stable* model is one where, starting from any initial state in a bounded state space, the model remains in that bounded space. Stability under model iteration is key in model-based RL. However, certain model-based RL algorithms use transitions from only short trajectories generated under the learned model, from real states observed in the environment. On the other hand, the calibration model is iterated for many steps because the agent interacts with the calibration model as if it were the true

environment—for an entire learning trajectory. Longer rollouts of transitions risk the accumulation of the prediction error. Hence, producing reasonable transitions under many steps of model iteration is key for the calibration model; much more so than in model-based RL.

Another scenario is that, even if the model is stable, it produces states that are not possible. In many cases, this could be benign and may help generalize to unseen states, but in other case this could be very misleading. For example, in a gridworld with walls, a learned model might produce outcome states that go through a wall because of generalization in the input space (x, y). Thus, model iteration can produce increasingly non-sensical states, as has been repeatedly shown in model-based RL [1, 14, 26, 54]. A *self-correcting* model is one that, even if it produces a few non-real states, it comes back to the space of real states.

It is key, therefore, that the calibration model be *stable* and *self-correcting*.

2) Calibration model that encodes pessimism and induces domain randomization

We would prefer to identify more conservative hyperparameters. We would rather set the hyperparameters cautiously even if this means the agent learns slower than it needed to in the real environment. This is even more important for our learning setup, where it is likely that the logged data does not have much coverage. Thus we would prefer the calibration model to encode pessimism in some form. A simple way to do this is to make the transitions in the calibration model stochastic. Adding stochasticity in the transitions also induces domain randomization which should help in searching more general hyperparameters. We discuss other approaches to encode such pessimism in the calibration model in Section 3.3.

3) Effective handling of actions with low data coverage

The model also needs to handle actions with no coverage or low coverage. It is

possible that the agent has access to several actions, but the offline data log includes only a few actions. The calibration model needs to deal with the scenario where the agent decides to take an action that is absent in the offline data log. In this case, the model cannot produce a reasonable transition with the *absent* action. A simple way to deal with this is to produce a transition from such states, back to a default state—like the start state—while sending a termination signal and the lowest return. Doing this should avoid learning further unreasonable transitions. This should also make the agent pessimistic about the actions not recorded in the offline data log.

A more likely scenario is that the agent decides to take an action in a given state, whereas the offline data log contains a different action for that state. There needs to be a strategy to deal with such scenarios. A simple way to deal with this is to undergo the most plausible transition that could be caused by a similar state and a similar action that are actually recorded in the offline data log. We describe more on this in Section 3.3.

4) Agents with hyperparameters that govern early learning only

The final criterion is a condition on the agent, rather than the model. Attempting to tune the hyperparameters for any long-running agent using only limited simulation with the calibration model can be problematic. Practically, we can only test each hyperparameter setting for a limited number of steps in the calibration model. So, the calibration model is only simulating early learning. It is hard to imagine that we could run the agent long enough to select hyperparameters for long run performance. This suggests that this approach will be most effective if we tune *initialization hyperparameters*: those that provide an initial value for a constant but wash away over time. Examples include an initial learning rate which is then adapted; magnitude of optimism for optimistic initialization; an initial architecture that is grown and pruned over time; and meta-parameters that play the role of adjusting the primary parameters online.

3.3 Using KNN Models to Obtain Stable Calibration Models

The calibration model acts like a model that simulates the transition dynamics of the underlying data. Thus for a given state and action, it outputs a next state, reward, and a termination signal. It may happen that the agent is in a particular state s in the calibration model, and the agent's policy selects some action a that is not recorded for state s in the offline data. Such a situation may arise if the action space is continuous, because the exact real-valued action selected by the policy may never be present in the dataset. A similar situation could also occur if the action space is discrete but the dataset had a different action in state s. We need an effective method to handle such situations while generating real next states and rewards.

We developed a non-parametric *k-nearest neighbor* (KNN) based calibration model that always generates real next states and rewards from the dataset. The idea is simple: we first use the entire offline data log to construct the model, with trajectories obtained by chaining together transitions, and by iteratively finding the most plausible transition—present in the offline data log—given the current state and action. It ensures the agent always produces real states observed in the data and remains in the true state distribution, and so is stable under many iterations. We construct it in such a way that it respects the design criteria specified in Section 3.2. We provide the pseudocode and summarize the approach in Algorithm 6, and highlight the key details in the next section.

3.3.1 Constructing a KNN Based Calibration Model

We use a k-d tree [9] to construct the KNN model. A k-d tree is a data structure that is typically used to quickly solve nearest neighbor queries. To initialize the calibration model (Algorithm 6 [2]), we format the transitions in the offline dataset as

[2]In Algorithm 6, we can ignore the **LaplaceRepTraining** function for now. We discuss this function in Section 3.3.2. Algorithm 6 would be valid by ignoring lines 2 and 3.

$(S_t, A_t, S_{t+1}, R_{t+1}, T_{t+1})$ for every timestep, where T refers to the termination signal which is $True$ if $\gamma = 0$, or if there is a failure or if the episode resets, else it is $False$. Every state-action pair (S_t, A_t) is used as the key in searching the k-d tree, and $(S_{t+1}, R_{t+1}, T_{t+1})$ is the value corresponding to the key.

It is important for the KNN-based calibration model to generate real next states and rewards in order to be stable under several model iterations. Hence, at the start of each episode (Algorithm 4), the calibration model randomly picks a start state from the set of real start states in the dataset. If the agent is in state S_t and decides to take action A_t based on its policy, the KNN calibration model finds a real state-action pair $(\tilde{S}, \tilde{A})$ that exists in the dataset and that is nearest to (S_t, A_t). The calibration model then internally undergoes a recorded real transition $(\tilde{S}, \tilde{A} \rightarrow \tilde{S'}, \tilde{R'}, \tilde{T})$ and returns the next state $\tilde{S'}$, reward $\tilde{R'}$, and termination signal $\tilde{T}$ to the agent. From the agent's perspective, it undergoes the transition $S_t, A_t \rightarrow \tilde{S'}, \tilde{R'}, \tilde{T}$. The intuition here is that, if (S_t, A_t) is close enough to $(\tilde{S}, \tilde{A})$, then $(S_{t+1}, R_{t+1}, T_{t+1})$ in the deployment environment would be similar to $(\tilde{S'}, \tilde{R'}, \tilde{T})$. The agent undergoes the above transition if a transition starting from (S_t, A_t) is not recorded in the dataset. However, if a transition from (S_t, A_t) is recorded in the dataset, then the KNN calibration model will undergo the recorded transition since the nearest existing neighbor to the (S_t, A_t) pair is itself. This meets the third criterion in Section 3.2 of effectively handling actions with low coverage.

The KNN-based approach allows us to find the most plausible transition given some current state and action. Most importantly, it returns a real combination of the next state, reward, and termination signal. An iterative rollout of such transitions would generate a trajectory as follows—a start state S_0 is randomly selected from the dataset of start states. The agent selects A_0, and the calibration model finds a real transition $(\tilde{S}, \tilde{A} \rightarrow \tilde{S'}, \tilde{R'}, \tilde{T})$ using the neighbor $(\tilde{S}, \tilde{A})$ nearest to (S_0, A_0). The agent then observes state $S_1 = \tilde{S'}$ and $R_1 = \tilde{R'}$. The agent then selects action A_1, and so on. This process can go on forever while generating only real states, thus enabling stability

under model iterations. Since the model is always in the space of real states, it does not need self-correction. Though simple, this KNN-based non-parametric approach perfectly satisfies the first criterion from Section 3.2. In contrast, learning stable parametric non-linear models can be quite complex as they need to jointly learn the transition dynamics function and a suitable Lyapunov function to ensure stability [34].

In the above explanation, we described searching for the nearest neighboring transition which is obtained by using k = 1 in the k-nearest neighbors model. However, it could happen that the agent encounters a particular state multiple times in the calibration model. If the agent's policy selects the same series of actions from that particular state, for k = 1, the agent would see the same trajectory, deterministically, when interacting with the calibration model and the agent could get stuck in a loop of transitions. This can be avoided by using k > 1, and thus searching the k nearest neighbors and selecting any one of them randomly to undergo a transition. The stochasticity in transitions can be increased by increasing the number of neighbors k. Each of the k closest neighbors would have a sampling probability proportional to $1 - \frac{d((s_j, a_j); (s_i, a_i))}{\sum_{z=1}^{k} d((s_z, a_z); (s_i, a_i))}, \forall j \in [1, k]$. Here, the function $d((s_j, a_j); (s_i, a_i))$ returns the distance between the state-action pairs (s_j, a_j) and (s_i, a_i). We discuss the distance metric in more detail later. Such a sampling strategy results in higher probability to sample the closest neighboring state-action pairs and lower probability to the neighboring state-action pairs that are further away. Thus, controlling for the number of k neighbors adds stochasticity in the transition dynamics function of the calibration model. This meets the second criterion from Section 3.2. This criterion induces domain randomization which has been shown to improve generalization of machine learning algorithms to new environments [57]. In our case this would lead to selecting more general hyperparameters that will be effective across different variations of the transition dynamics. The selected hyperparameter should be more pessimistic and more reliable than a highly optimistic hyperparameter selected with clean data. The

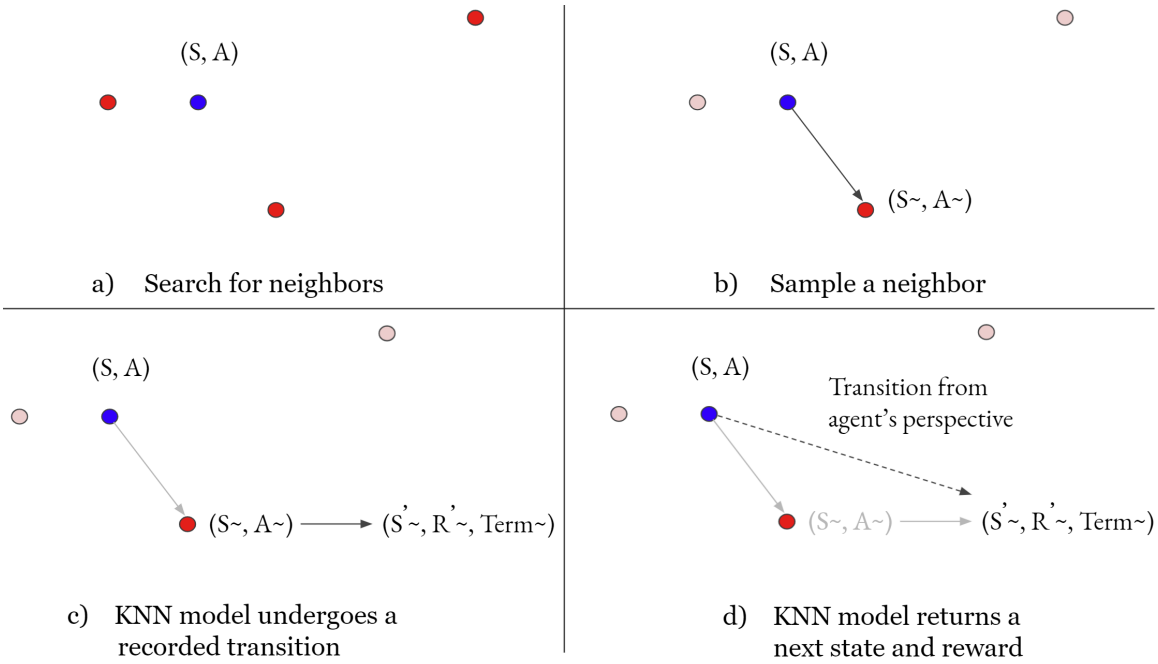

Figure 3.2: **One-step transition generated by the KNN calibration model**. Plot a) shows how the KNN model searches for k-nearest neighbors to the current state-action pair. Plot b) shows how the KNN model then samples one of these neighbors. Plot c) shows how the KNN model undergoes a one-step transition from the selected neighbor. This transition is also recorded in the offline data logs. Plot d) shows how the KNN model then returns the next state and reward observed from this one-step transition back to the agent. From the agent's perspective, it takes action A in state S, and observes next state $\tilde{S}'$ and reward $\tilde{R}'$.

latter could overfit the offline data log which could be problematic if the data log is limited or if the underlying environment changes. Figure 3.2 shows how the next transition is generated using the k-nearest neighbors method. Figure 3.3 illustrates the internal working of the KNN calibration model. A roll out of these one-step transitions can lead to a long trajectory of interaction.

Algorithm 7 describes how we construct the k-d tree and Algorithm 8 describes how we search the k-d tree. We group the transitions in the offline data log in different ways depending on the action space. If the environment has a discrete action space, the transitions are grouped into different k-d trees according to the actions. When searching for the nearest neighbors given some discrete action a that the agent takes, the calibration model searches only in the k-d tree corresponding to action

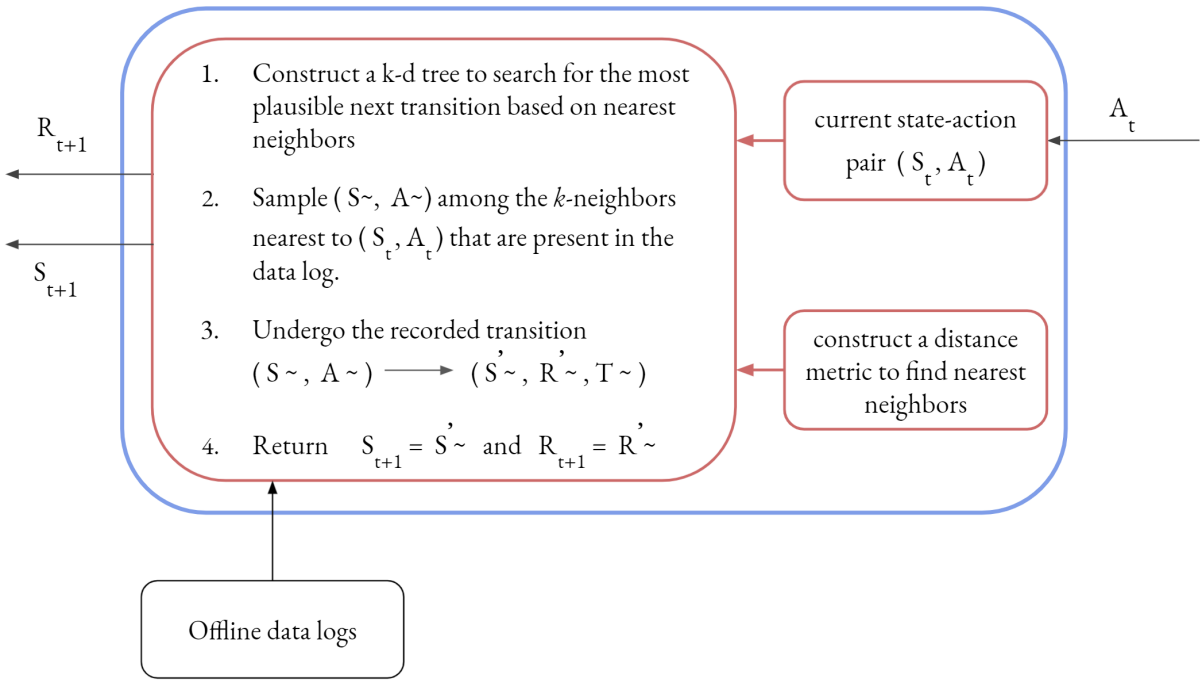

Figure 3.3: **Internal working of the KNN calibration model:** The figure describes how one-step transitions are generated inside the KNN calibration model. The KNN calibration model outputs only real next states and rewards that are actually recorded in the offline data log. Thus, long trajectories of interaction can be rolled out using this approach while enabling model stability. The blue thick border represents the KNN calibration model from Figure 3.1. The red border describes the internal details of the KNN calibration model.

a. The model uses the distance between the states to find the nearest neighbors — $d((s,a);(\tilde{s},a)) = d(s;\tilde{s})$. Whereas in continuous action space environments, only one tree is constructed. When searching for the nearest neighbors in this case, the calibration model uses the distance between state-action pairs to find the nearest neighbors — $d((s,a);(\tilde{s},\tilde{a}))$. We discuss more about the distance metric in Section 3.3.2.

Algorithm 6 CalibrationModelTraining

Input: $\mathcal{D}$: the offline dataset

1: Assume the dataset is a list of tuples of the form (S, A, S', R, T)
2: $\psi \leftarrow$ **LaplaceRepTraining**(S, A, S', T)
3: $\Phi \leftarrow \psi(S)$
4: $Trees \leftarrow$ **KDTreeConstruction**(Φ, A, S', R, T)
5: Fit $Trees$ into the calibration model $\mathcal{C}$

Return: $\mathcal{C}$

Algorithm 7 KDTreeConstruction

Input: Φ, A, S', R, T: transitions

1: **if** discrete action space **then**
2: Construct KD-trees $Trees$ for each action: use Φ as key and S', R, T as value.
3: **else**
4: Construct KD-trees $Trees$: use (Φ, A) as key and S', R, T as value.

Return: $Trees$

Algorithm 8 KDTreeSearch

Input: ϕ: representation
a: action
$Trees$: KD-trees
k: number of nearest neighbors

1: **if** discrete action space **then**
2: $\{(\phi, s', r, T, d)\}_1^k \leftarrow$ Search for the k nearest neighbors of ϕ in the tree corresponding to the action a ($Trees[a]$), where d refers to the distances.
3: **else**
4: $\{(\phi, s', r, T, d)\}_1^k \leftarrow$ Search for the k nearest neighbors of (ϕ, a) in a single tree $Trees$, where d refers to the distances.

Return: $\{(\phi, s', r, T, d)\}_1^k$

For a calibration model, all knowledge about the true environment comes from the offline dataset, including the state distribution, reward range, and the terminating signal. It could happen that the dataset does not have enough coverage of the underlying environment and thus may have missing data in some parts of the underlying MDP. Given a dataset, it is hard to imagine whether there are missing states or what the missing states are. To deal with missing data, KNN model assumes that any area that is not included in the dataset is dangerous to get into even in the true environment. It gives a pessimistic prediction in this case and lets the agent learn to avoid that area. In the discrete action space situation, if the agent picks an action that does not exist in the dataset or the distance between the current state and its nearest neighbor is above some threshold, the KNN model sets the termination signal to be true and returns the lowest return observed in the dataset, to prevent the agent to be too optimistic about this action. In continuous action space, it is unlikely for the dataset to include the same real valued action that the agent chooses in the calibration model. In this case, we set a threshold on the maximum allowed distance between the state-action pairs. Thus the calibration model terminates the current episode when the distance between the current state-action pair and its nearest neighbor is beyond this threshold value. This meets the other half of the third criterion from Section 3.2 where some actions are absent in the dataset.

Thus the KNN based model meets the design criteria as it is stable under model iterations, is stochastic to induce pessimism, can handle low action coverage scenarios, and does all of this while generating only real next states and rewards that are recorded in the data log. This makes it a suitable candidate to be used as a calibration model.

3.3.2 Distance Metric Based on the Laplacian Representation

The KNN model relies heavily on the distance metric to find the nearest neighbors, which are then used to generate a transition inside the model. Hence, using the right

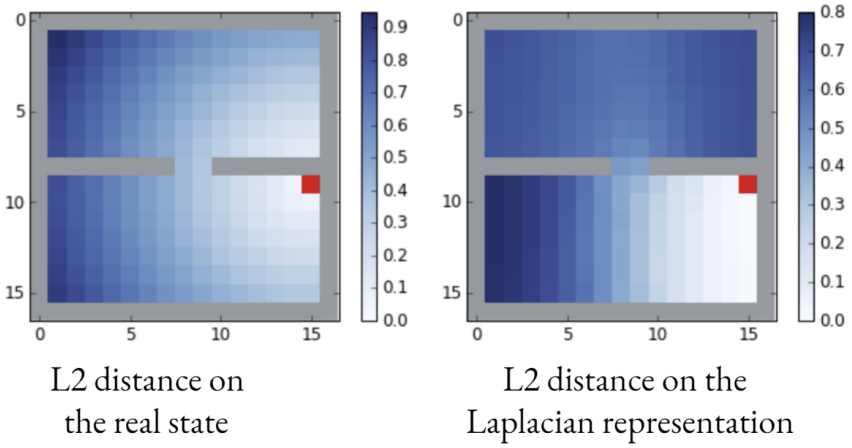

Figure 3.4: **L2 distance using (x, y) coordinate representation vs using Laplacian representation:** Figure re-used from Wu et al. (2019) [64].

distance metric is important for generating accurate transitions that better resemble the underlying environment. The default choice of Euclidean distance in the input space may not always be the appropriate choice for measuring similarity in the states. For example, if inputs correspond to (x, y), and the environment is a maze, two nearby points in Euclidean distance may actually be far apart in terms of transition dynamics. Images are another example where Euclidean distance is not appropriate.

Instead, we learn a distance metric based on the Laplacian representation [31, 64]. The Laplacian representation is trained by pushing the representations of random states to be far away from each other, while encouraging the representations of *nearby* states to be similar. Two states are nearby if it only takes a few steps for the agent to get to one state from the other. Euclidean distance in this new space is reflective of similarity in terms of transition dynamics.

Figure 3.4 shows the difference between the distance metrics when using L2 distance directly on the input state space which uses (x, y) coordinates and when using L2 distance on the Laplacian representation of the input state space. It shows how using L2 distance directly on the input state space does not care about the walls in the grid. It shows the states on the other side of the wall to be closer in distance to the red state despite the fact that it is not easy for the agent to transit to the other side of the wall.

Whereas, the learned Laplacian representation takes into consideration the transition dynamics of the grid world and correctly learns that the states on either sides of the walls are not closer to each other even when their coordinates can misleadingly tell that they might be similar. Thus, the Laplacian representation shows a better distance metric to measure similarity of states. Finding the distance between the Laplacian representations of the red state and some other state can be thought of as measuring a quantity that is approximately proportional to the total number of steps needed to reach the red state from the other state. Thus, measuring L2 distance based on the Laplacian representation instead of the raw states gives a more reliable nearest neighbor. This makes the transitions within the KNN calibration model better resemble the underlying environment.

The Laplacian representation ψ is trained in the same way as proposed by Wu et al. (2018) [64]. Algorithm 9 describes the pseudocode for training the Laplacian representation. We use a neural network with 2 hidden layers and 128 hidden units each to train the Laplacian representation on a batch of data. We use the Adam optimizer, a tanh activation function, and a mini-batch size of 128. The neural network takes in a state as input and the output vector is its Laplacian representation. In training the neural network, a state s from the trajectory $s, s_1, s_2, \cdots, s_n$ is chosen. Its nearby state s_u is sampled from the trajectory $s_1, s_2, \cdots, s_n$ with normalized probability $P_\kappa = [\kappa, \kappa^2, \cdots, \kappa^n]$. A random state s_v is also sampled from the data. The loss for training the Laplacian representation is calculated using the nearby state pair (s, s_u) and a random state pair (s, s_v) as follows:

$$Loss = \frac{1}{2}(\phi_s - \phi_{s_u})^2 + \beta((\phi_s^T \phi_{s_v})^2 - \delta\|\phi_s\|^2 - \delta\|\phi_{s_v}\|^2) \tag{3.1}$$

where β and δ can be considered as two regularization coefficients. Here, ϕ_s is equivalent to $\psi(s)$. In Equation 3.1, the first term encourages the difference between the representations of the nearby states to be as small as possible. The second term pushes the representations of the random states to be orthogonal (as different as

Algorithm 9 LaplaceRepTraining

Input: $D = \{(S, A, S', R, T)\}$: dataset
N: dataset size
β, δ: weights in loss function
α: learning rate
b: batch size

$Converge \leftarrow False$
Initialize representation function ψ
$i \leftarrow 0$
$LossList = [\ ]$
$AverageLossList = [\ ]$
while not $Converge$ and $i < 30000$ **do**
  $SampledS \leftarrow \{s_j\}, j \in [1, N]$
  $NearbyS \leftarrow \{s_{j+n}\}, n \sim P_\kappa$
  $RandomS \leftarrow \{s_l\}, l \in [1, N]$
  $\phi_s \leftarrow \psi(SampledS)$
  $\phi_c \leftarrow \psi(NearbyS)$
  $\phi_r \leftarrow \psi(RandomS)$
  Update Φ with $Loss = \frac{1}{2}(\phi_s - \phi_c)^2 + \beta((\phi_s^T \phi_r)^2 - \delta\|\phi_s\|^2 - \delta\|\phi_r\|^2)$
  $i \leftarrow i + 1$
  $LossList.append(Loss)$
  if $i \mod 1000 == 0$ **then**
    $AverageLoss = \sum_{m=0}^{1000}(LossList[m])$
    $AverageLossList.append(AverageLoss)$
    $LossList \leftarrow [\ ]$
    if $AverageLoss$ has increased 2 times in a row **then**
      $Converge \leftarrow True$

Return: ψ

possible). Thus the second term will be zero if the representations are orthogonal. The second term can also be zero if the representations themselves are of zero magnitude. The last two terms make sure that the representations of the randomly sampled states are not of zero magnitude. Every 1000 steps, we record the average loss over the last 1000 steps. If the recorded loss increases 2 times in a row, we stop training. The Laplacian representation is trained for a maximum of 30,000 steps if the early stopping criterion is not met.

The Laplacian representation has its own parameters to tune, namely κ (sampling probability constant), β and δ (regularization coefficients), α (learning rate of the

neural network), and the trajectory length n that is used to pick nearby states. We sweep over different values of these parameters and conduct the above training scheme for each combination. Finally we obtain several trained Laplacian representations corresponding to each parameter setting. We use 5-fold cross validation to pick the best parameters. The validation set is created by randomly picking 20% of the transitions in the dataset while the rest of the transitions were used as the training set. The trained Laplacian representations are then evaluated on a validation set by using the dynamic awareness score (Equation 3.2), which measures the quality of the learned representations. In Equation 3.2, N is the number of trajectories in the validation set, ϕ_i refers to the representation of the i_{th} state and ϕ'_i is the representation of the next state that comes after the i_{th} state in the trajectory. These are considered as nearby states. $\phi_{j\sim U(1,N)}$ is the representation of a randomly sampled state from the validation set. Since the Laplacian representation is supposed to capture the transition dynamics, the distance between nearby states' representations should be smaller than the distance between random states' representations. On average, when the L2 distance between the representation of nearby states ϕ_i and ϕ'_i is smaller than the L2 distance between the representations of random states ϕ_i and $\phi_{j\sim U(1,N)}$, the equation gives a higher score. Thus, a higher dynamic awareness describes that the learned Laplacian representations are of better quality. We then select the parameter that has the highest dynamic awareness score averaged across 5-fold cross validation for its learned Laplacian representation. The selected parameters for each experiment are specified in Chapter 5.1 and Chapter 5.2.

$$\text{Dynamics Awareness} = \frac{\sum_i^N ||\phi_i - \phi_{j\sim U(1,N)}||_2 - \sum_i^N ||\phi_i - \phi'_i||_2}{\sum_i^N ||\phi_i - \phi_{j\sim U(1,N)}||_2} \tag{3.2}$$

After a parameter setting has been selected, its corresponding Laplacian representation is trained using a training and validation set split (80:20) and with the

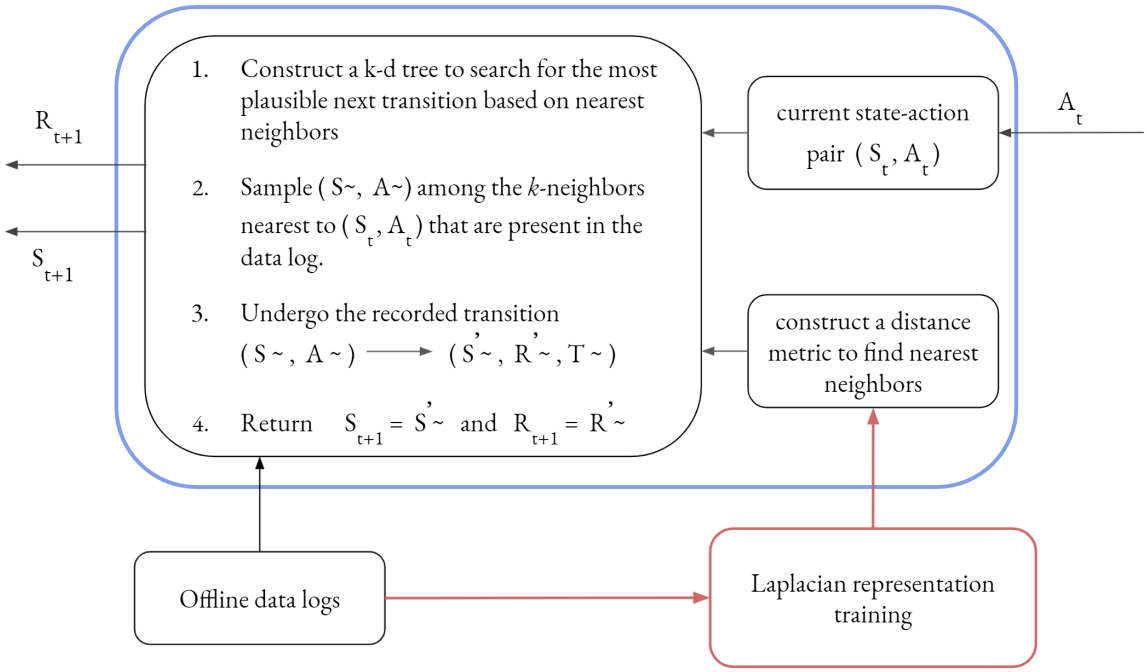

Figure 3.5: **Using the KNN calibration model with the Laplacian representation:** The Laplacian representation is trained using the same offline data log used to construct the calibration model. The distance metric uses this trained Laplacian representation to then find the nearest neighbors to generate transitions. The blue thick border represents the KNN calibration model. The red border represents the newly added component — Laplacian representation training — to the previous Figure 3.3.

early stopping criterion as above. Once the Laplacian representation of the best parameter setting is selected, we then need to use it find the distance between state-action pairs. For discrete actions, we define the distance metric only on state-action pairs with the same action. So the distance between (s, a) and $(\tilde{s}, a)$ is given by $d((s, a); (\tilde{s}, a)) = \|\psi(s) - \psi(\tilde{s})\|_2$. For continuous actions, the distance between (s, a) and $(\tilde{s}, \tilde{a})$ can be defined as $d((s, a); (\tilde{s}, \tilde{a})) = \left((\|a - \tilde{a}\|)^2 + (\|\psi(s) - \psi(\tilde{s})\|_2)^2\right)^{\frac{1}{2}}$.

Figure 3.5 illustrates how the Laplacian representation is used with the KNN calibration model. Thus, the whole process is to use the offline data log to train a KNN calibration model first. Then, we use the same offline data log to train the Laplacian representation for each state. The Laplacian representation then gives us a distance metric that can be used with the KNN calibration model to search for nearest neighbors to generate transitions. We can thus simulate online learning inside

the KNN calibration model forever while generating only real next states and rewards, and by using a distance metric that better resembles the transitions dynamics of the underlying environment.

3.4 Different Variants of the Calibration Model

So far we have looked at the KNN variant of the calibration model. However, we could also use other models for constructing the calibration model. A commonly used technique to learn a model is by training a neural network. Neural networks (NN) can be used to predict the next state, reward, and termination signal, given the current state and action. Note that the prediction given by the NN may not be a real state that can be seen in the dataset, while the KNN calibration model always returns a real state. The synthetic prediction of the NN calibration model brings the risk that the prediction error accumulates as the rollout length becomes longer, thus the NN calibration model could be less stable than KNN calibration model. Though an NN based calibration model does not necessarily meet the design criterion of being a stable model, we conduct experiments with it to demonstrate its performance just because NNs are a common choice for modelling the transition dynamics of the environment.

The NN model takes in the current state (or the Laplacian representation of the current state) and the action (one-hot encoding) as inputs, and it predicts the next state, reward, and termination signal. The NN model can thus directly generate a transition by prediction. During the training of the NN model in our experiments, the predictions of the next state, reward, and termination signal were done separately by independent neural networks with the same hidden layer structure. We use three 2-hidden layer neural networks with 128 hidden units each to train the NN model to predict its three outputs. We train the neural networks using the Adam optimizer (learning rate values specified in Chapter 5), a tanh activation function, and a mini-batch size of 16. The loss function for each neural network was the mean squared error. The validation set was created using 20% of the transitions in the dataset selected

randomly while the rest of transitions were the training set. When measuring the performance of the NN model, we used the mean squared error on the validation set. During training, we tested the performance every epoch, if the error increases 2 times in a row, we cut off training. Otherwise, the NN model is trained for at most 100 epochs.

In Section 5.1.1, we compare the hyperparameter selection results of the KNN calibration model and the NN calibration model. For both the KNN calibration model and the NN calibration model, we can either use the real state (raw observation vector) or the Laplacian representation of the real state as inputs. The Laplacian representation is specific to the offline data log. So for a given data log, both the KNN-based and NN-based calibration models use the same Laplacian representation trained using the data log. In the KNN calibration model with the raw observation vector, the distance metric to find the nearest neighbors is Euclidean distance between raw observations, whereas in the KNN calibration model with the Laplacian representation, the distance metric is Euclidean distance between the Laplacian representations. The NN based model takes either the raw observation vector or its Laplacian representation as the input. In both the cases, the NN based model predicts the next state, reward, and termination signal.

3.5 Summary

This chapter introduced the central idea of the thesis — developing a solution strategy to the Data2Online problem by using calibration models. This chapter first explained what calibration models are. The chapter further described how calibration models can be used to evaluate hyperparameters and then select promising hyperparameters offline. The chapter then highlighted key criteria to design calibration models and put some conditions on the types of agents for which Data2Online should be effective. The chapter then discussed how to use a k-nearest neighbor (KNN) based approach to construct the calibration model. The chapter further discussed how the KNN model

meets all the previously highlighted design criteria, making it a suitable choice for a stable calibration model. The KNN calibration model relies on a good distance metric to identify the nearest neighbors for a particular state-action pair. The chapter described the distance metric based on the Laplacian representation, highlighted its benefits, and described how to train it. Finally, the chapter discussed how we can build a calibration model using neural networks, though such a model will not necessarily be a stable model.

Chapter 4

Experimental Setup

This chapter describes the prerequisites to the experiments discussed in Chapter 5. This chapter then discusses the environments we use, the agents we compare, the data collection schemes, details about the hyperparameter sweep, and how we evaluate the agents in the calibration model and in the deployment environment. All experiments were conducted on a cluster and a powerful workstation using 8252 cpu hours.

4.1 Environments

We used three environments in our experiments - Puddle World, Acrobot, and Cartpole, as shown in Figure 4.1.

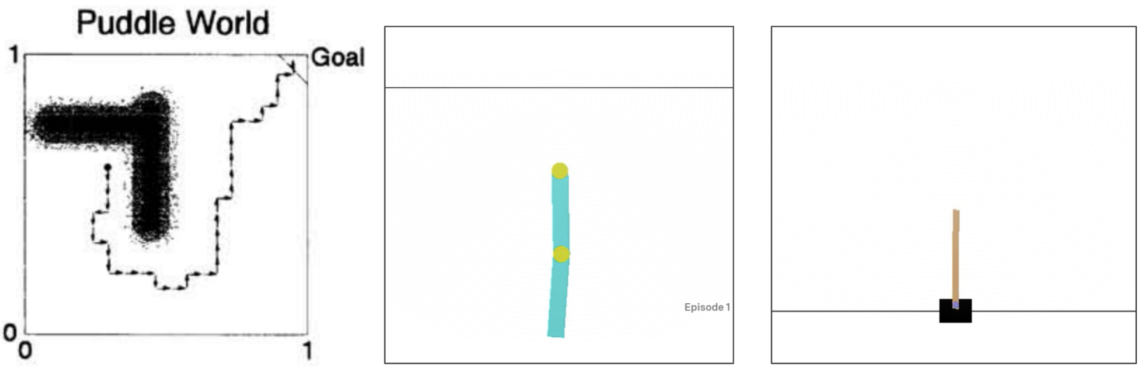

Figure 4.1: **Environments**

Puddle World

We used the Puddle World environment with the transition dynamics, reward function, state space, and actions exactly as described by Sutton (1996) [51]. Puddle World is an episodic task where the agent randomly starts in a non-goal region. The episode ends when the agent reaches the goal region in the upper right corner. The objective in Puddle World is to reach the goal region as fast as possible while avoiding the puddle which gives the agent negative rewards of high magnitude. The deeper the agent gets into the puddle, lower the reward it gets. Otherwise, the agent gets -1 per step until it gets to the goal. The discount factor $\gamma = 1.0$. The state-space is 2-dimensional, containing the x and y coordinates and the state space is $[0, 1]$ along both dimensions. The agent has 4 actions—left, right, up, down— which move 0.05 units in these directions unless the movement would cause the agent to leave the state space, in which case the movement is bounded by the state space. A random gaussian noise with mean = 0 and stddev = 0.01 is added to the motion along both dimensions.

Acrobot

We used the Acrobot environment with the transition dynamics, reward function, state space, and actions as described in OpenAI Gym [11]. Acrobot consists of 2 connected links with the top joint fixed and torque applied to the bottom joint. It is an episodic task where the links start in the rest position, pointing downwards. The episode ends when the tip of the bottom link reaches a specified height level marked by the horizontal bar in Figure 4.1. The objective is to raise the tip of the acrobot above the specified height level as fast as possible. The state space is 6-dimensional. The first 4 dimensions contain the cos and sin values of the angle between the first link and the vertical line, and the cos and sin values of angle between 2 links. The range for the first 4 dimensions is $[-1, 1]$ each. The last 2 dimensions are the angular velocities of the two joints. The range for the angular velocity of the top joint is $[-4\pi, 4\pi]s^{-1}$, while the range for the angular velocity of the bottom joint is $[-9\pi, 9\pi]s^{-1}$. At the

beginning of each episode, each dimension of the state is randomly set to be in range $[-0.1, 0.1]$. The reward on each step is -1. The agent is trained to cross the height level with the least number of steps. The discount factor $\gamma = 1.0$. The agent has 3 actions— +1, 0, -1 torque applied on the joint connecting the two links.

Cartpole

We used the Cartpole environment with the transition dynamics, state space, and actions exactly as described in OpenAI Gym [11]. However, we introduce some random gaussian noise with mean = 0 and stddev = 0.1 over the effect of actions by adding noise to the applied force at each timestep. Cartpole consists of a horizontally moving cart and a pole attached on top of it. The start state has the cart in the centre of the track, and the pole vertical. We use the continuing version of Cartpole as described by Sutton & Barto (2018) [52]. In this, the cartpole is *transitioned* to the start state when the pole falls below some angle ($\pm 12^\circ$) or when the cart goes off the track (± 2.4 units). Note that in this case, we do not reset the episode but it is instead considered as a transition back to the start state. The objective is to balance the pole vertically by moving the cart left or right. The agent gets a reward of -1 whenever the pole falls below some angle or when the cart goes off the track. Otherwise, it gets a reward of 0. The state space is 4-dimensional, containing cart position, cart velocity, pole angle, and pole angular velocity. At the beginning of the episode, each dimension of the state is randomly set to be in range $[-0.05, 0.05]$ units. The discount factor $\gamma = 0.9$. The agent has 2 actions - left and right, to move the cart.

4.2 Data Collection

Our goal from the experiments is to highlight the performance of our strategy to select hyperparameters offline from data logs in order to deploy online RL agents in the real system. Hence in our experiments, we assume access to offline data logs and not the underlying environment. The above section described the environments that we use in

our experiments. In this section, we highlight how we collect logs of data from these environments.

Data collection consists of two important components - a) the data collecting policy, and b) the amount of data collected. These two components affect the data coverage, and thus can affect the quality of the calibration model that uses these data logs. To get the policy for data collection, we train an Expected Sarsa agent in the environment until some level of performance is reached. We then freeze the policy and allow this fixed policy to interact with the same environment in order to log the transitions. We can control how much to train the agent. This gives us the flexibility to obtain policies with performance ranging between extremely poor to near-optimal. Policies with different levels of performance typically also have different coverage of the underlying environment. Once we obtain the data collecting policy, we can also control how long it gets to interact with the underlying environment and log transitions. Through this we can control the size of the offline dataset. Thus by varying the data collecting policy and the amount of collected data, we can test our strategy in different scenarios.

In real-world systems, we usually have only one dataset to choose the hyperparameters and one chance to deploy the agent. This case corresponds to having 1 random seed in our experiments. To evaluate the calibration model fairly and robustly, we test it with 30 random seeds and report the performance of the hyperparameters selected for all random seeds. Thus, we collect 30 offline datasets in total, each collected with a different seed. Each collected offline dataset can be then used to construct a calibration model and train a Laplacian representation. For every experiment described in Chapter 5.1 and Chapter 5.2, we mention the specific details of the policies we use to collect data and the total number of transitions of data we collect.

4.3 Agents

This section describes the agents we use in our experiments. This section also specifies the hyperparameters we sweep for each agent in each environment.

We use four agents: Expected Sarsa, Actor-Critic, Fitted Q Iteration (FQI), and a random hyperparameter selection baseline agent. We use Expected Sarsa and Actor-Critic as the two learning agents along with the calibration model to demonstrate how our strategy works. Our strategy uses the offline data logs to select promising hyperparameters for Expected Sarsa and Actor-Critic. We then evaluate the performance of these agents with the selected hyperparameters in the underlying environment to judge the quality of the selected hyperparameters and the performance of our strategy. To compare against our strategy, we use two baseline algorithms. The first baseline is FQI which is a classical offline RL algorithm. We test FQI alongside our strategy to demonstrate how transferring a fixed policy compares with transferring hyperparameters. The second baseline that we use is the random hyperparameter selection baseline agent (discussed further below). This baseline acts as a naive agent to compare against.

The hyperparameter values that we sweep for the agents were picked such that some combinations gave good performance, some gave moderate performance, and some gave poor performance. Designing the hyperparameter sets in such a way was important to see whether our strategy is able to select the best hyperparameters among a set of several hyperparameters with mixed performances.

1) Expected Sarsa

Here, we describe the experiment specific details of the Expected Sarsa agent and mention the hyperparameters that we sweep for the agent. The algorithmic details of Expected Sarsa are described in Chapter 2. The Expected Sarsa agent uses tile coded features as its input [52], with 16 tilings and 8 tiles. Each state dimension is tile-coded individually and pair-wise, and the resulting feature vectors are concatenated to get a single feature vector. We then use a linear function approximator along with the tile coded state and the action to estimate the action-values. The agent updates the action-values using Adam optimizer. In Adam optimizer, the second momentum

parameter β_2 was set as 0.999. The agent uses optimistic initialization of action-values for early exploration. We kept the policy stochastic by using a softmax function over the action-values and sampled the action from the induced probability distribution.

We sweep four key hyperparameters for which the performance of the Expected Sarsa agent is sensitive— the Adam stepsize α, the first momentum parameter β_1 of Adam optimizer, temperature of the softmax policy τ, and optimistic weight initialization w_0 of action-values. We chose these hyperparameters because their impact on performance is somewhat transient and can be overcome by continued learning; this reflects our desire for the agent to continually learn and adapt in deployment. We describe the hyperparameter sweep details for Expected Sarsa below.

In the Acrobot experiments in Chapter 5.1, we did grid search over the following hyperparameters resulting in 54 combinations:

1. Adam optimizer learning rate α: {0.003, 0.03, 0.3}
2. Adam optimizer momentum β_1: {0.0, 0.9}
3. Softmax temperature τ: {1.0, 10.0, 100.0}
4. Optimistic weight initialization w_0: {0.0, 4.0, 8.0}
5. Eligibility trace parameter $\lambda_e = 0.8$

In the Puddle World experiments in Chapter 5.1, we did grid search over the following hyperparameters resulting in 54 combinations:

1. Adam optimizer learning rate α: {0.01, 0.03, 0.1}
2. Adam optimizer β_1: {0.0, 0.9}
3. Softmax temperature τ: {1.0, 10.0, 100.0}
4. Optimistic weight initialization w_0: {0.0, 8.0, 16.0}
5. Eligibility trace parameter $\lambda_e = 0.1$

In the Cartpole experiments in Chapter 5.1, we did grid search over the following hyperparameters resulting in 54 combinations:

1. Adam optimizer learning rate α: {0.03, 0.1, 0.3}
2. Adam optimizer β_1: {0.0, 0.9}
3. Softmax temperature τ: {0.1, 1.0, 10.0}
4. Optimistic weight initialization w_0: {0.0, 6.0, 12.0}
5. Eligibility trace parameter $\lambda_e = 0.023$

2) Actor-Critic

Here, we describe the experiment specific details of the Actor-Critic agent and mention the hyperparameters that we sweep for the agent. The algorithmic details of Actor-Critic are described in Chapter 2. The Actor-Critic agent uses the same tile coding schema as the Expected Sarsa agent (16 tilings and 8 tiles). Each state dimension is tile-coded individually and pair-wise, and the resulting feature vectors are concatenated to get a single feature vector. The actor uses one linear function approximator for each action to obtain a list of scores (action preferences), and the scores of all the actions are converted to probabilities using a softmax function to get the agent's policy. The critic also uses a linear function approximator to predict the value of a given tile coded state. In our experiments, both the actor and the critic were zero-initialized and used SGD optimizer.

For Actor-Critic experiments, we vary two key hyperparameters for which the performance of the Actor-Critic agent is sensitive— the actor's learning rate and the critic's learning rate. To eliminate some hyperparameter combinations that are less meaningful and to keep compute tractable, we swept the critic's learning rate and the ratio between actor's learning rate and critic's learning rate. This is because of the prior knowledge that, in practice, the actor's learning rate is usually smaller than the critic's learning rate. We chose these hyperparameters because their impact on

performance is somewhat transient and can be overcome by continued learning; this reflects our desire for the agent to continually learn and adapt in deployment.

In both Acrobot and Puddle World experiments in Chapter 5.1, we did grid search over the following hyperparameters resulting in 36 combinations:

1. Critic's learning rate α: $\{0.001, 0.003, 0.01, 0.03, 0.1, 0.3\}$

2. Actor's learning rate: $\{0.001\alpha, 0.003\alpha, 0.01\alpha, 0.03\alpha, 0.1\alpha, 0.3\alpha\}$

Here, for example, if the critic's learning rate $\alpha = 0.3$, and the ratio between the actor's learning rate and critic's learning rate is 0.1, then the actor's learning rate $= 0.1\alpha = 0.1 \times 0.3 = 0.03$.

3) FQI

Here, we describe the experiment specific details of the FQI agent and mention the hyperparameters that we sweep for the agent. The algorithmic details of FQI are described in Chapter 2. We use a linear function approximator based on tile coding and use 16 tilings and 2 tiles. Each state dimension is tile-coded individually and pair-wise, and the resulting feature vectors are concatenated to get a single feature vector. We use Adam optimizer to perform gradient descent steps. We use a mini-batch size of 128, sync period of 512 steps, and 50,000 training iterations. We train regularized FQI offline with the same aforementioned 30 datasets that we use with our strategy (Section 4.2). For each training dataset, we choose one of the other offline datasets as its validation set and do a grid search on the hyperparameter set as described below. After offline training, for each dataset we choose the learned action-value function of the hyperparameter setting with the lowest final MSTDE on the validation dataset and deploy an ϵ-greedy policy (with $\epsilon = 0.1$) with respect to this action-value function to the true environment. Typically, FQI is deployed with a fixed extracted policy that is greedy ($\epsilon = 0.0$) with respect to its action-value function. However, in our experiments, the policy with $\epsilon = 0.0$ performed worse; so we added some exploration using a higher ϵ value. In many cases the greedy FQI agent got stuck in a small region

in the state space for a long time. In practice, ϵ-greedy FQI agent can jump out of these regions and perform better.

For FQI, we vary two key hyperparameters for which the performance of FQI is sensitive— Adam optimizer's learning rate and the regularization coefficient. In both Acrobot and Puddle World experiments, with other hyperparameters in the agent fixed, we did grid search over the following hyperparameters resulting in 15 combinations:

1. Adam optimizer learning rate α: $\{10^{-1}, 10^{-2}, 10^{-3}, 10^{-4}, 10^{-5}\}$

2. L2 regularization coefficient λ_f: $\{10^{-1}, 10^{-3}, 10^{-5}\}$

4) Random hyperparameter selection baseline

Finally, we compare our approach against a random hyperparameter selection baseline. This baseline simulates the case when we do not know the best hyperparameter setting so we randomly pick one from the list. We still use a base learning agent like Expected Sarsa (or Actor-Critic) to learn the policy. The baseline randomly selects one hyperparameter setting from the set of hyperparameters in each run, and we evaluate Expected Sarsa (or Actor-Critic) with these randomly selected hyperparameters and report performance. Since this baseline randomly picks a hyperparameter, it does not use the calibration model.

4.4 Evaluation of the Agents in the Calibration Model and the Deployment Environment

So far we have discussed how to collect the dataset, construct a calibration model from the data, test different hyperparameter settings inside the calibration model, and then select the best performing setting in the calibration model. However, we need some method to evaluate the efficacy of our strategy, that is, whether the selected hyperparameter setting indeed does perform well in the deployment environment. In this section, we discuss how we evaluate the hyperparameters in the calibration

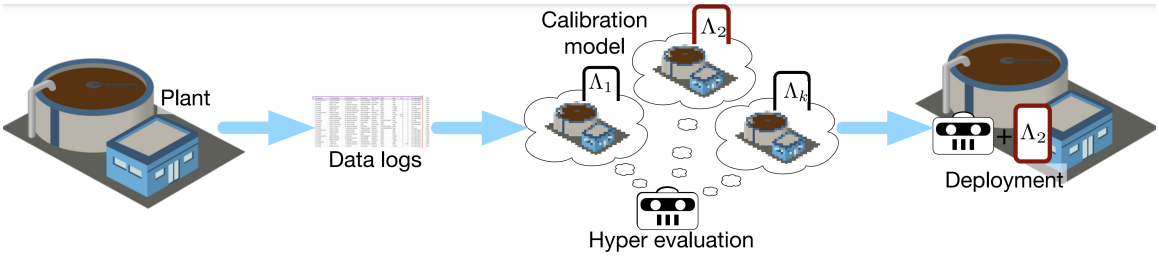

Figure 4.2: **Evaluation of hyperparameters in the calibration model.** This figure illustrates how hyperparameters are evaluated in the calibration model. The data logs can be used to construct a calibration model of the underlying environment. The calibration model need not resemble the underlying environment perfectly. Each hyperparameter setting is denoted by Λ. Each hyperparameter setting can be evaluated in the calibration model based on some performance criterion. The best performing hyperparameter is then selected for deployment in the underlying plant.

model specific to our experimental setup. We further discuss how we evaluate our strategy's performance, that is, the performance of the selected hyperparameters in the deployment environment. We also discuss how to evaluate the performance of the baselines.

Selecting hyperparameters in the calibration model:

For every dataset that is collected, we construct a unique calibration model and train the Laplacian representation. For every calibration model, we evaluate all hyperparameters 10 times over 10 random seeds. We then select the hyperparameter with the highest average online performance (total reward) across these 10 runs. Thus for every dataset (or calibration model), we select one hyperparameter setting to deploy. In any real world application, we typically will have one huge dataset to construct the calibration model and then finally select one hyperparameter to deploy. In our case, this corresponds to one run of the whole experiment— taking in a dataset, constructing the calibration model, evaluating the hyperparameters in the model and then selecting the best one. Figure 4.2 illustrates the evaluation of the hyperparameters in a single calibration model constructed from a single dataset. In our experiments, we do this whole process 30 times to understand how our strategy performs across multiple independent runs. Hence we collect 30 datasets to evaluate

the performance of our strategy across 30 different seeds. Thus, we select a total of 30 hyperparameters corresponding to these 30 datasets.

Evaluating the selected hyperparameters in the deployment environment:

Our strategy performs the above procedure, that is, to use the dataset to select a hyperparameter setting. However, we now need to evaluate whether the selected hyperparameters are good enough in the deployment environment, which is what we finally care about. To measure how well our strategy performs, we evaluate the online performance of the selected hyperparameters directly in the deployment environment. Each selected hyperparameter is evaluated for 30 runs for n_{steps} timesteps using 30 random seeds (n_{steps} depends on the experiment). The average of the 30 independent runs of each hyperparameter gives us the *true* performance of each hyperparameter in the deployment environment. This is used as the ground truth performance of that hyperparameter in the deployment environment. If the true performance of the selected hyperparameter is high, it means that our strategy has performed well by selecting a good hyperparameter. The true performance of the selected hyperparameter will always be lower than or equal to the true performance of the *best* hyperparameter in the sweep in the deployment enviroment. Hence, a smaller difference in the true best performance and the true performance of our selected hyperparameter suggests a more reliable calibration model. Ideally, we want to select the best hyperparameter in the sweep for every single run (or dataset). So the best case scenario would be to select the hyperparameter with the highest *true* performance each time.

To summarize the procedure - we take a dataset, construct a calibration model, evaluate each hyperparameter 10 times in the calibration model and select the hyperparameter with the highest average performance across these 10 runs, then evaluate the selected hyperparameter in the deployment environment for 30 times and report its true performance. We do this whole procedure 30 times for 30 datasets. In the end, we will be left with performance scores of the 30 selected hyperparameters. We follow

the above procedure of evaluating the hyperparameters in the calibration model and the deployment environment for Expected Sarsa and Actor-Critic.

Evaluating the baselines:

The random hyperparameter selection baseline simply selects one hyperparameter setting randomly for each dataset without constructing a calibration model. Thus we report the true performance of the 30 randomly selected hyperparameters for 30 datasets. Note that the random hyperparameter selection baseline still uses a base learning agent like Expected Sarsa or Actor-Critic. The FQI agent uses the offline datasets to extract a policy corresponding to the best performing hyperparameter. Thus, FQI extracts 30 policies from the 30 datasets. To evaluate the FQI agent, we run the extracted policy on the deployment environment for n_{steps} timesteps. Each extracted policy is evaluated 30 times with 30 random seeds in the deployment environment. Thus, all 30 extracted policies are evaluated 30 times. The average online performance of each deployed policy across the 30 runs is considered as the *true* performance of the deployed policy. Thus by following this procedure for our strategy and the baselines, we can get the true performance of our strategy and the baselines in the deployment environment. This will be useful to compare our strategy with the baselines.

Online performance metric:

The metric used to evaluate online performance of the hyperparameters is the same in the calibration model and the deployment environment. For example, an agent with a higher average episodic return in puddle world has better performance, so we use the area under the curve (AUC) of the average episodic return as a measure of performance in both the deployment environment and the calibration model. Additionally, we ensure the agent learns from at least 30 episodes in each run inside the calibration model. Thus, if the experiment is 15,000 timesteps long, we keep the timeout to be 500 so the agent sees at least 30 episodes.

4.5 Summary

This chapter described the setup for the experiments that follow in the next chapter. In specific, this chapter listed the environments that are used in the experiments, and gave a detailed description about their state space, action space, reward function, and the objective in each environment. The chapter further discussed how offline data logs are collected from these environments, such that the data logs can then be used to construct the calibration model and train the Laplacian representation. The chapter also listed the agents that are tested in our experiments, and gave a detailed description about the hyperparameters that we sweep for each agent in each environment. Finally, the chapter ended with describing how the agents are evaluated in the calibration model to select the hyperparameters, and then evaluated in the deployment environment in order to measure the performance of our strategy and the baselines.

Chapter 5

Experimental Results

In this chapter, we describe in detail the experiments that were conducted and discuss the results of these experiments. This chapter is divided into two sections. In Section 5.1, we conduct several experiments to investigate the scope of our strategy to select hyperparameters offline. We subject our solution strategy to several different scenarios of the Data2Online problem and highlight our findings. Section 5.1 focuses solely on our strategy's ability to *select* hyperparameters from a set of many hyperparameters using grid search. Section 5.2 of this chapter builds upon this naive idea of selecting hyperparameters from a set of hyperparameters. This section extends its further to provide a better way to use the calibration model to *fine-tune* the hyperparameters offline.

5.1 Offline Hyperparameter Selection Using Grid Search

In this section, we conducted a battery of experiments to provide a rounded assessment of when our approach can or cannot be expected to reliably select hyperparameters for online learning. We investigate varying the data collection policy and size of the data logs to mimic a variety of data collection scenarios ranging from a near-optimal operator to random data. We explore selecting hyperparameters of different types for different agents, and investigate a non-stationary setting where the environment

changes from data collection to deployment. In all experiments discussed in this section, our approach uses grid search to select good hyperparameters offline from a set of several possible hyperparameters.

5.1.1 Experiment 1: Comparison With Baselines and Using Different Calibration Models

We begin with the simplest first question: how does our approach compare to simple baselines and with different choices of calibration model type?

Experiment Description

First, we investigate the benefits of our approach with different choices of calibration model in two classic control domains and compare the performance of our strategy with the baselines. We further provide results with and without the Laplacian based distance metric. We then also conduct the same experiment with a different agent and report how our strategy performs in selecting hyperparameters for this different agent. Following are the experiment specifications:

Environments: We conducted experiments in two continuous state, discrete action, episodic deployment environments, Acrobot and Puddle World, as described in Chapter 4.

Data collection: As mentioned in Chapter 4, we used the Expected Sarsa agent for data collection. We train the agent to achieve some level of performance and then cut-off training. We then use the saved policy to collect data. We initialized the agent with a hyperparameter setting that is capable of achieving near-optimal performance. In this experiment, we used a near-optimal policy for each domain to collect data for building the calibration models. The near-optimal data collection policy for Acrobot can solve the task in 100 steps and the near-optimal data collection policy in Puddle World achieves an average episodic return of -25. We collected 5000 transitions for each of the 30 datasets with the near-optimal policy. In both cases of Acrobot and

Puddle World, the policy will provide the system with many examples of successful trajectories to the goal states in the 5000 transition data log.

Agents and calibration models: We used the FQI agent as a baseline agent to compare against our approach. FQI learns a policy from the calibration data and then deploys the learned policy fixed in the deployment environment representing a classical batch RL fixed-deployment baseline. We also used the random hyperparameter selection baseline agent as a naive baseline to compare against. Additionally, we used the linear Softmax-policy Expected Sarsa agent along with two calibration models - k-nearest neighbor (KNN) model and neural network (NN) model, both with the learned *Laplacian encoding* (see Section 3.3.2). The NN model is trained to predict the next state and reward given the current action and the Laplacian encoding of the current state as input to provide the network with a better transition-aware input representation. The FQI agent's training is described in Chapter 4. All agents are trained in the calibration model and evaluated in the deployment environment for 15,000 timesteps in Acrobot and for 30,000 timesteps in Puddle World.

Model construction and representation training: The KNN calibration model was trained as described in Section 3.3. We used k=3 in the k-nearest neighbors calibration model. The NN calibration model was trained as described in Section 3.4. For training the NN model in both the Acrobot and Puddle World experiments, we used Adam optimizer's learning rate of 0.0003. The Laplacian representation training details are described in Section 3.3.2. For the Laplacian representation training in Acrobot, we chose $\kappa = 0.8$, $\beta = 5$, $\delta = 0.5$, $\alpha = 0.00003$, and the length of trajectory n for picking nearby states was set to 20. In Puddle World, we used $\kappa = 0.8$, $\beta = 5$, $\delta = 0.05$, $\alpha = 0.0003$, and the length of trajectory n for picking nearby states was set to 10.

Hyperparameters used in the grid search: In this first experiment we selected the hyperparameters for the Expected Sarsa agent from data generated by a near-optimal policy that has good coverage of the underlying environments. The agent

used tile coding to map the continuous state variables to binary feature vectors. This on-policy Expected Sarsa agent learns quickly but is sensitive to several important hyperparameters. We investigated several dimensions of hyperparameters including the step-size and momentum parameters of the Adam optimizer, the temperature parameter of the policy, and the weight initialization for the action-value function. We chose these hyperparameters because their impact on performance is somewhat transient and can be overcome by continued learning; this reflects our desire for the agent to continually learn and adapt in deployment. For the grid search, we used the hyperparameter values specified in Section 4.3.

Evaluation pipeline: Our evaluation pipeline involves several steps. First we evaluate the *true performance* (steps per episode for Acrobot and return per episode in Puddle World) of each hyperparameter combination in the deployment environment running for 15,000 steps in Acrobot and 30,000 steps in Puddle World, averaging over 30 runs. We used the data collection policy to generate the data log, learn the calibration model, and train the Laplacian representation. We then evaluate the hyperparameters inside the calibration model and select the best performing one. We record the *true performance* of the selected hyperparameters to summarize the performance of the calibration model.

This whole process—running the data collection policy to generate a data log, learning the calibration model, and evaluating the hyperparameters—is repeated 30 times (giving 30 datasets with 30 corresponding hyperparameter selections). The statistic of interest is the median and the distribution of the *true performance* for the hyperparameters selected across runs. In the ideal case, we want the strategy to choose the best hyperparameter every time and the variance in *true performance* would then be zero. For the FQI baseline we simply plot the distribution of performance of each of the 30 extracted policies on the deployment environment. This whole evaluation pipeline will be the same for all the following experiments. Figure 5.1 summarizes the results.

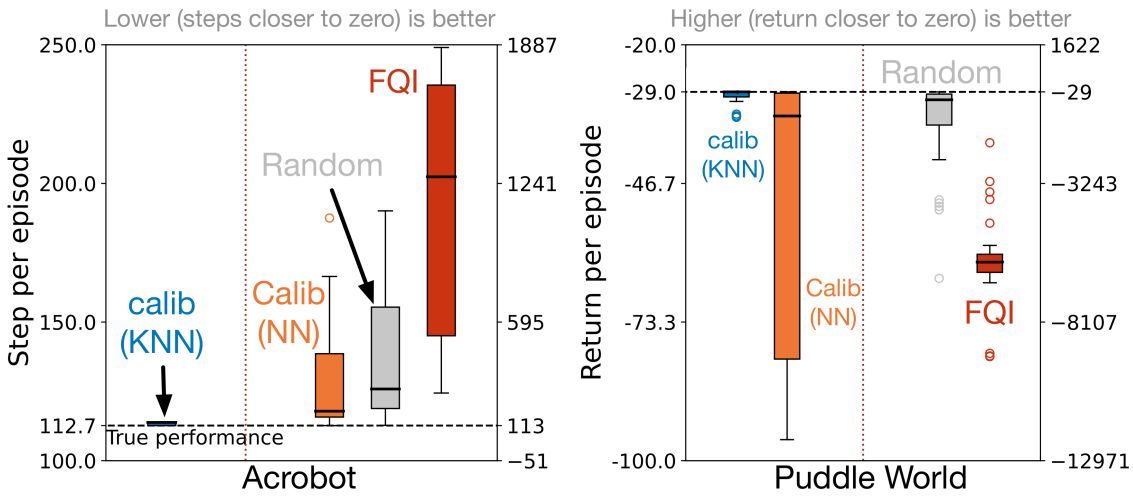

Figure 5.1: **Hyperparameter transfer with calibration model.** Each subplot shows the performance of two calibration models compared against FQI and the random hyperparameter selection baseline. The dashed horizontal line indicates the true performance of the Expected Sarsa agent with the best hyperparameter setting in the sweep in the deployment environment. Each box shows the distribution summarizing the true performance in deployment of the best hyperparameters selected in each run of the experiment. In Acrobot **lower is better**, and in Puddle World **higher is better**. If the centre of mass is close to the dashed horizontal line, then the system is choosing hyperparameters well. Low variance indicates that the system reliably chooses hyperparameters that perform similarly across runs. We include the performance of randomly selecting hyperparameters on each run as a baseline. In each box the bold line represents the median, the boxes represent the 25th and 75th percentiles, the whiskers (extending lines) represent the minimum and maximum performance, unless there are outliers which are represented by circles. The left side of each box uses the left y-axis and the right side (separated by the dotted vertical line) uses the right y-axis.

Results and Discussion

For both the Acrobot and Puddle World plots in Figure 5.1, there are two separate Y-axes scales. The vertical dotted line separates the box plots, such that the box plots to the left of the dotted line follow the left Y-axis and the ones to the right follow the right Y-axis. In Acrobot, since the Y-axis is average steps per episode, lower value on the Y-axis indicates better performance. Whereas in Puddle World, higher value on the Y-axis indicates better performance. Each box plot corresponds to the spread in the performance of the deployed hyperparameters across the 30 runs (30 datasets).

Each box plot is marked by a bold black line that indicates the median performance and the edges of the box indicate the quartiles. The whiskers indicate the minimum and maximum performance and the circles represent outliers. The horizontal dashed black line indicates the true performance of the Expected Sarsa agent with the best hyperparameter in the sweep. This line is the best possible attainable performance of our strategy. In the ideal case, our strategy always selects the best hyperparameter in the sweep for every run, and thus the box plot would collapse onto the horizontal dashed line and have zero variance.

Figure 5.1 shows that the KNN model's box plot has almost collapsed onto the horizontal dotted line in both Acrobot and Puddle World experiments. Thus, in both environments the KNN calibration model performed very well, selecting the same hyperparameters as would a sweep directly in the deployment environment. The NN calibration model performed poorly overall in comparison to the KNN model. The NN model performance can be unstable, choosing hyperparameters with good performance in some runs, but often choosing poor hyperparameters. The poor performance of the NN calibration model can be explained by model iterations that generate non-real next states and rewards, causing the prediction error at each transition step to accumulate. Both the KNN and NN calibration models use the Laplacian encoding. In both Acrobot and Puddle World, the KNN and NN calibration models outperform random hyperparameter selection baselie as well as the FQI baseline agent. In constrast, the FQI baseline performs much worse than even the random hyperparameter selection baseline. This suggests the data log is too limited to extract a good policy and deploy it without additional learning, but the same data appear useful for selecting hyperparameters with our approach.

Additional Results

In the above experiments, we demonstrated how both the KNN as well as the NN calibration models using Laplacian encoding outperform the two baselines - FQI

and the random hyperparameter selection baseline. It is also clear from the results how the KNN based calibration model performs much better than the NN based calibration model. The KNN calibration model almost always selects near-optimal hyperparameters in all the runs and thus has high performance and low variance. The NN calibration model, on the other hand, has quite high variance in its performance. Thus, for the exact same datasets the KNN model gives more stable results.

We conduct two additional experiments to give more insight into the strategy.

1) Calibration models using raw states (or observation vectors) as input, that is, without the Laplacian encoding

The above experiments use the calibration models with the Laplacian encoding. We now see what happens when we do not use the Laplacian encoding with the calibration model, that is, when we just use the raw states as the input. For training the NN calibration model with raw states in Acrobot we use the learning rate of 0.0003 in the Adam optimizer, while in Puddle World we use learning rate of 0.0001. All other training details of the KNN calibration model, the NN calibration model, and the Laplacian representation are exactly the same as in the above experiment. Figure 5.2 shows the results.

We observe mixed results when using the NN calibration model with and without the Laplacian encoding. In Puddle World, the NN model with the Laplacian encoding performs significantly better than the NN model without the Laplacian encoding. However in Acrobot, the model without the Laplacian encoding outperforms the model with the Laplacian encoding, however, the difference in the performance is smaller than what is seen in Puddle World.

We observe a minuscule difference in the performance of the KNN calibration model with and without the Laplacian encoding. In Puddle World, the two models perform nearly the same. Even in Acrobot, the two models perform quite similarly except in two outlier cases where the KNN model without the Laplacian encoding is not able to

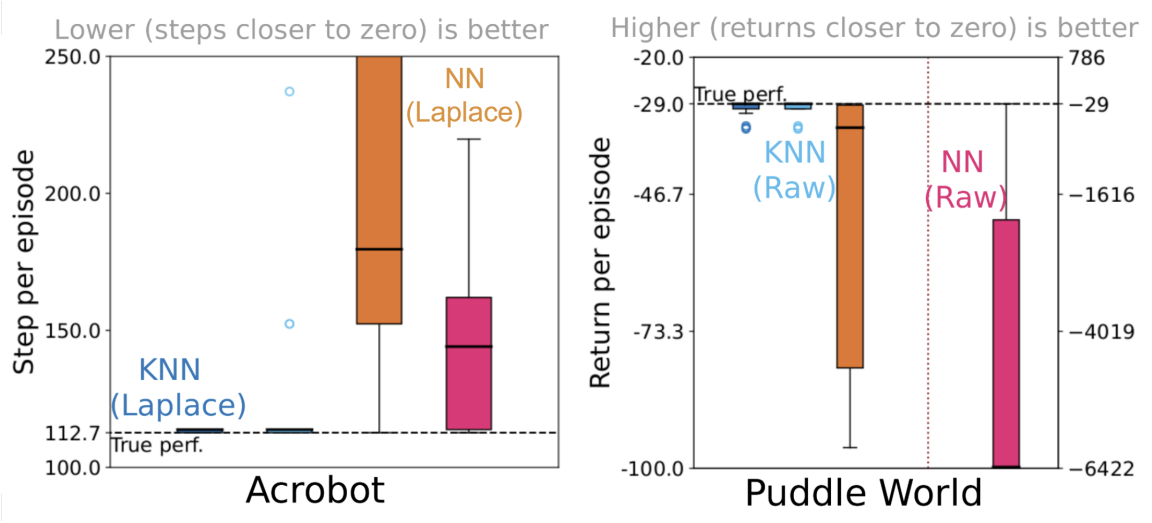

Figure 5.2: **Replacing Laplacian representation with raw state.** Each subplot shows the result of comparing using raw state and Laplacian representation in KNN based and NN based calibration model. In Acrobot (lhs) **lower is better** and in Puddle World (rhs) **higher is better**.

learn good hyperparameters.

So overall, the calibration models with the Laplacian encoding performed slightly better than the ones without the Laplacian encoding. The effects of using the Laplacian representation might be more pronounced in some different kinds of environments such as a grid world with walls in it. Using Euclidean distance on the Laplacian representations will have a significant positive effect on the performance in such environments where we need a transition-aware distance metric. However, throughout our experiments we note that the KNN model outperforms the NN model with or without the Laplacian representation.

2) Results when using a different agent

The results in the experiments in the above section use Expected Sarsa as the learning agent. But can we expect similar results when using a different agent? To answer this, we repeat the same experiment as above but now using Actor-Critic instead. For the grid search, we used the hyperparameter values specified in Section 4.3. We again compare our strategy with FQI and the random hyperparameter selection baseline. Figure 5.3 shows the results.

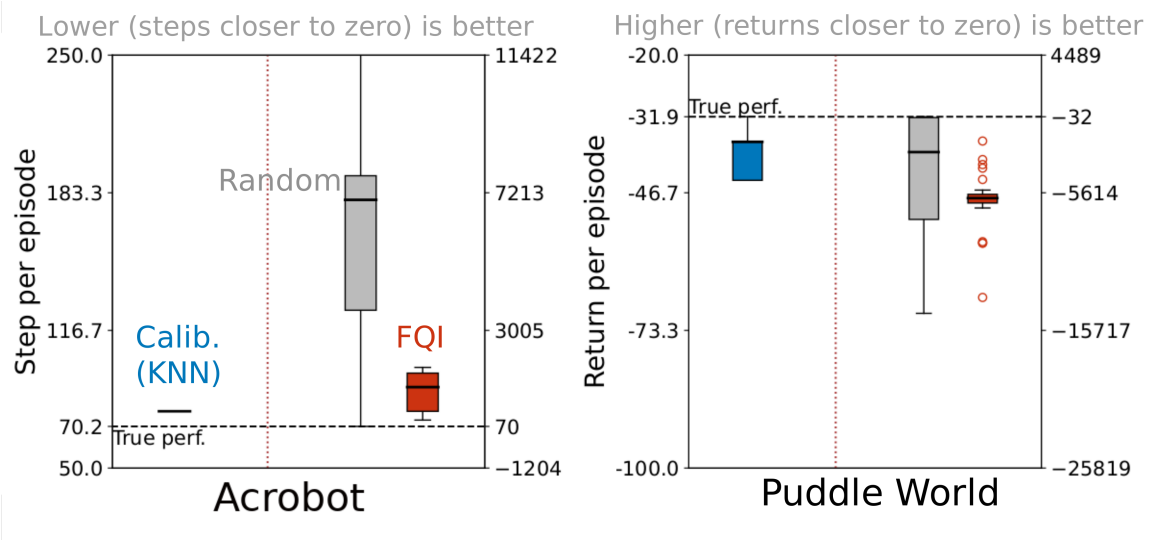

Figure 5.3: **Calibration model hyperparameter transfer with Actor-Critic agent.** Each subplot shows the calibration model performance with Actor-Critic agent, compared against FQI and a random hyperparameter selection baseline. The dashed horizontal line indicates the performance of Actor-Critic agent with the best hyperparameter setting in the sweep in the deployment environment. In Acrobot (lhs) **lower is better** and in Puddle World (rhs) **higher is better**.

Again, our strategy significantly outperforms both the baselines - FQI and random hyperparameter selection. The Actor-Critic results for our strategy are slightly worse than in the Expected Sarsa experiment, but nevertheless they are still good. This goes to show that the calibration model works quite well when selecting the hyperparameters of different types of agents.

5.1.2 Experiment 2: Effect of Varying Data Collection

Hyperparameter selection depends on the quality of the calibration model which, in turn, depends on the quality of the data logs. The data log quality depends on the coverage of the underlying environment. Data coverage is influenced by two major factors - a) the data collecting policy, and b) the total amount of transitions collected. So the data collecting policy and the dataset size are two important factors that can affect hyperparameter selection. In this experiment we evaluate the robustness of our approach to changes in the amount of offline data available, and the quality of the

policy used to collect the data.

Experiment Description

We conduct two different types of experiments: 1) vary the data collecting policy, keeping the amount of collected data fixed, and 2) vary the amount of data collected, keeping the data collecting policy fixed. The experiment specifications are:

Environment: We conducted the experiments in the Puddle World and Acrobot environments.

Data collection: Here we describe data collection in the two experiments:

1) Varying data collecting policy with fixed dataset size :

In this experiment, we fixed the dataset size to be 5000 transitions. We experimented with 3 different policies corresponding to *near-optimal*, *medium*, and *naive* performance for training our KNN Laplace calibration model. The near-optimal policy was identical to the one used in the previous experiment. The medium policy was designed to achieve roughly half the visits to goal states compared to the near-optimal policy in the same budget of 5000 timesteps. The naive policy was designed such that it achieved significantly fewer visits. In Acrobot, we require there are at least 50, between 20-30, and between 10-15 episodes in the 5000 transitions dataset, for near-optimal, medium, and naive policies respectively. In Puddle World, the number of episodes are more than 200, between 80-100, and between 20-50 respectively.

2) Varying dataset size with a fixed data collecting policy:

In this experiment, we fixed the data collecting policy to be the medium policy from above. We collected 3 different datasets with sizes of 5000, 1000, and 500 transitions. For Acrobot, we made sure the 5000, 1000, and 500 transitions datasets had 20-30, 4-6, and 2-3 episodes respectively. For Puddle World, we made sure they had 80-100, 16-20, and 8-10 episodes respectively.

Model construction and representation training: The KNN calibration model was trained as described in Section 3.3. We used k=3 in the k-nearest neighbors calibration model. The Laplacian representation training details are described in Section 3.3.2. For the Laplacian representation training in Acrobot, we chose $\kappa = 0.8$, $\beta = 5$, $\delta = 0.5$, $\alpha = 0.00001$, and the length of trajectory n for picking nearby states was set to 5. For Puddle World, we used $\kappa = 0.8$, $\beta = 5$, $\delta = 0.05$, $\alpha = 0.00001$, and the length of trajectory n for picking nearby states was set to 5.

Agents: We used the Expected Sarsa agent in both the experiments. In the Acrobot experiment, we trained and evaluated the agent for 15,000 timesteps in the calibration model as well as in the deployment environment. Whereas in the Puddle World experiment, we trained and evaluated the agent for 30,000 timesteps.

Hyperparameters used in the grid search: For the grid search, we used the hyperparameter values specified in Section 4.3.

Results and Discussion

Figure 5.4 describes the results of the experiment. It shows bar plots as against box plots shown in the previous experiment. The median performance here is indicated by the top of the bar plot and the quartiles are shown by the whiskers. Figure 5.4 a) and Figure 5.4 b) show that in both Acrobot and Puddle World, our strategy performed almost equally well, even while varying the dataset sizes. Even 500 transitions contains enough coverage of the state-space and successful terminations to produce a useful calibration model. This is in stark contrast to the FQI results in Experiment 1 in section 5.1.1, where a policy trained offline from the same size data log failed to solve either task despite containing 5000 transitions. Similarly, Figure 5.4 c) and Figure 5.4 d) show that in both Acrobot and Puddle World, our approach performs well despite significantly varying the data collecting policies. The near-optimal, medium, as well as the naive policy have different coverage of the underlying environment due to the nature of their *suboptimality*. However, all policies captured enough coverage for the

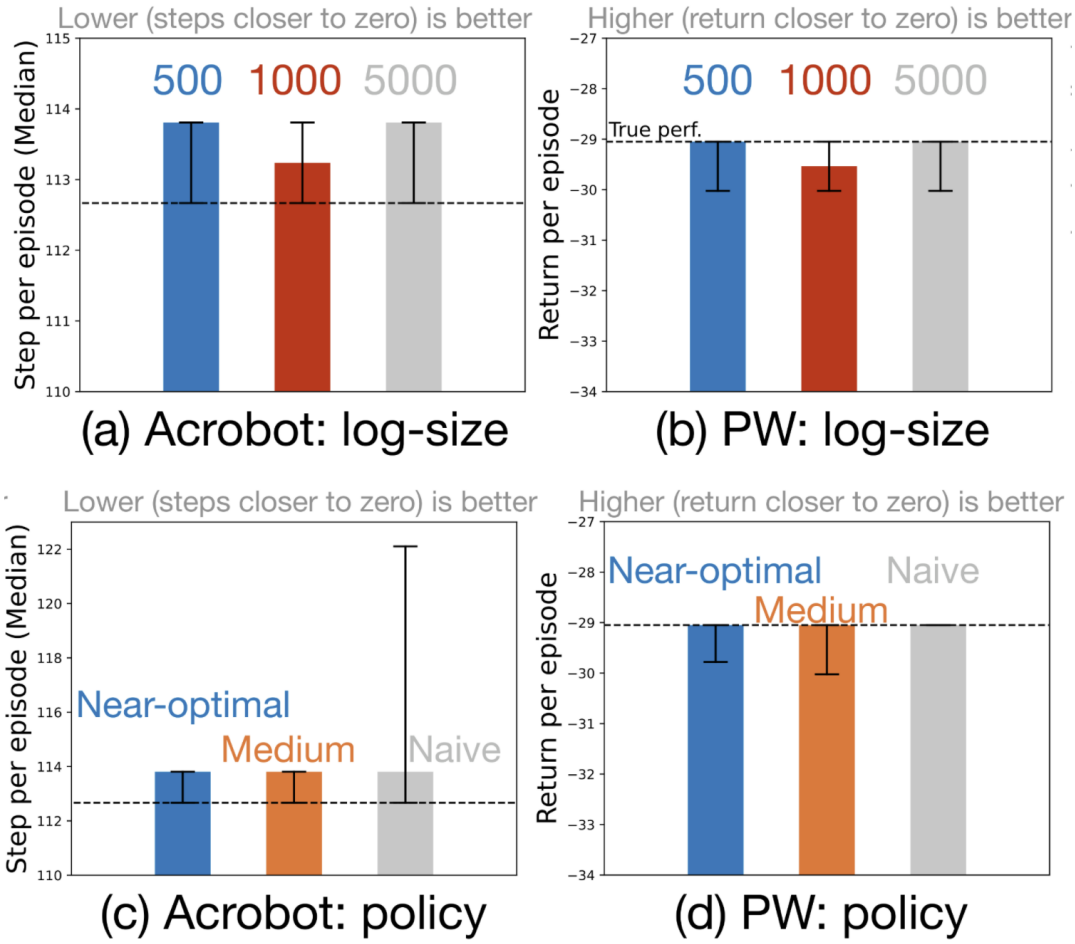

Figure 5.4: **The role of data logs.** Plots (a) and (b) show the median calibration model performance (with 25% and 75% quartiles) when changing the data log size. Plots (c) and (d) show the median performance when the data collecting policy changes. In Acrobot (lhs) **lower is better**, and in Puddle World (rhs) **higher is better**. Note, these are barplots: the median is shown by the top of the bar, and the quartiles are shown by the whiskers (extending lines). Overall our approach is largely insensitive to the data log size and policy in these two domains.

datasets to be useful. The results in Figure 5.4 show that our approach is largely insensitive to data log size and policy in these classic domains. Exploration in both these domains is not challenging; therefore, the success of the calibration model is not surprising.

5.1.3 Experiment 3: When the Deployment Environment Has Changed After Data Collection

Learning online is critical when we expect the environment to change. This can happen due to wear and tear on physical hardware, un-modelled seasonal changes, or the environment may appear non-stationary to the agent because the agent's state representation does not model all aspects of the true state of the MDP. In our problem setting, the deployment environment could change significantly between data collection and the online learning deployment phase. Intuitively we would expect batch approaches that simply deploy a fixed policy learned from data to do poorly in such settings. The following experiment simulates such a scenario.

Experiment Description

Following are the experiment specifications:

Environments: We conducted the experiment in the Acrobot environment. However, this experiment uses two variants of the Acrobot environment. Unlike before, we evaluate the selected hyperparameters and the FQI policy on a second, *changed* Acrobot environment wherein we double the length and mass of the first (top) link. This changes the dynamics but does not prevent learning reasonably good policies.

Data collection: Though we evaluate the selected hyperparameters on the second *changed* Acrobot environment, we use the original Acrobot environment for data collection. This change in the environment at the time of deployment mimics the situation when the environment has changed after data collection. We used a near-optimal policy to collect 5000 transitions from the original Acrobot environment. The

policy is considered as near-optimal when the averaged length of episode is less than 100. During policy learning, once the performance is above the given threshold, we cut-off learning and use the saved policy to collect data.

Model construction and representation training: The KNN calibration model was trained as described in Section 3.3. We used k=3 in the k-nearest neighbors calibration model. The Laplacian representation training details are described in Section 3.3.2. For the Laplacian representation training, we chose $\kappa = 0.8$, $\beta = 5$, $\delta = 0.5$, $\alpha = 0.00003$, and the length of trajectory n for picking nearby states was set to 20.

Agents: The four agents used in this experiment are:

1) As before, we used the Expected Sarsa agent and transferred the hyperparameters selected by our strategy using the KNN calibration model.

2) We also trained a policy with tile coded FQI and transferred the fixed policy.

3) Our third agent was obtained by transferring the best performing policy learned by Expected Sarsa in our calibration model to the second environment with no further learning. This agent differs from the first agent in that it transfers the learned policy in KNN calibration model as against transferring the hyperparameters.

4) Finally as a sanity check we obtain our fourth agent by training a policy using Expected Sarsa directly on the first environment and evaluating its performance on the second environment. We gave the agent enough time (50,000 timesteps) to train in the original environment for it to learn a good policy before transferring it to the changed environment. The learned policy is fixed and then transferred to the changed environment.

Here, only the first agent transferred hyperparameters using our strategy. The other three agents transferred fixed policies in the second *changed* environment. We train the first 3 agents for 15,000 timesteps using the calibration model (or offline dataset for FQI). We train the fourth agent for 50,000 in the original environment. After transferring these agents to the changed environment, we measure the performance

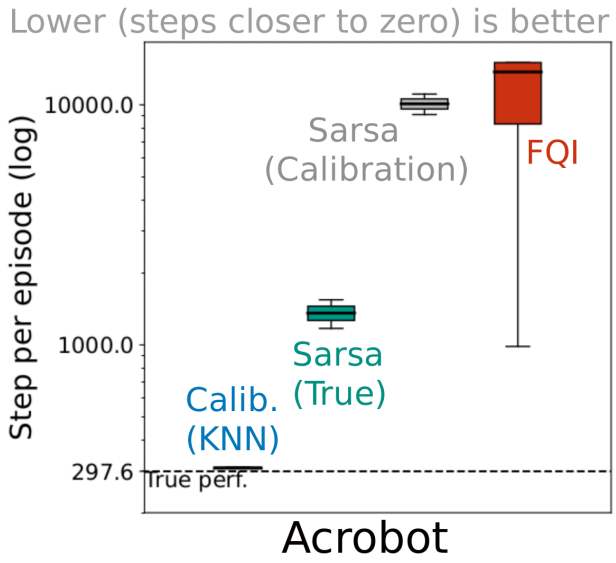

Figure 5.5: **When Acrobot changes.** Our approach is compared to FQI, a baseline transferring the policy learned in the original environment, and a baseline transferring the policy learned in the calibration model. This plot can be interpreted exactly as Figure 5.1: the median close to the horizontal dashed line represents good performance and small spread indicates that the system reliably chooses hyperparameters that perform similarity across runs.

of all the agents for 15,000 timesteps. This whole process was repeated 30 times (generating 30 datasets) to aggregate the results presented in Figure 5.5.

Hyperparameters used in the grid search: For the grid search, we used the hyperparameter values specified in Section 4.3.

Results and Discussion

Figure 5.5 shows the results of the four agents that were tested in this experiment. The Y-axis is plotted using a log scale. Since the Y-axis corresponds to the average number of steps per episode, lower number on Y-axis corresponds to better performance. The dashed horizontal line indicates the true performance of the Expected Sarsa agent using the best hyperparameter from the sweep in the second *changed* Acrobot environment. The blue box plot (first agent) indicates the performance of the Expected Sarsa agent whose hyperparameters are selected using the calibration model. The red box plot (second agent) is the performance of transferring a fixed policy learned using FQI on

the offline dataset. The grey box plot (third agent) is the performance of transferring a fixed policy learned using Expected Sarsa on the calibration model. Finally, the green box plot (fourth agent) is the performance of transferring a fixed policy learned using Expected Sarsa on the first Acrobot environment.

The results in Figure 5.5 show that the blue box plot (our strategy) has almost collapsed completely onto the dashed horizontal line. It indicates that our strategy is selecting near-best hyperparameters in all 30 runs for the *changed* environment despite only using data from the original environment. It highlights that our approach performs best and appears very robust (almost zero variance) despite the abrupt non-stationarity tested here. All the other three agents transferred fixed policies in the second changed environment. Figure 5.5 highlights that transferring static policies can be problematic when the environment changes. Clearly the difference between the two environments is significant; even transferring a policy directly learned on the first environment (green box plot) performs much worse than using our approach and learning a new policy from scratch. The transfer performance of the policy learned in the calibration model (grey box plot) is much worse. The only difference between the blue box plot and the grey box plot is that the former transferred hyperparameters, whereas the latter transferred a fixed policy while both used the exact same calibration model. This suggests that our approach of transferring hyperparameters (blue box plot) is somewhat robust by design to changes in the underlying environment between data collection and deployment. Even the FQI agent (red box plot) performs poorly in comparison to our approach. Thus, using the original environment data to select agent hyperparameters for a changed (but similar) environment is quite effective. This goes to show that hyperparameters are quite general in their nature and are fairly less sensitive to changes in the underlying environment than fixed policies. In summary, if the deployment environment is different than the environment used for data collection, transferring the agent hyperparameters and learning from scratch clearly seems to be much better than transferring fixed policies learned using several different techniques.

5.1.4 Experiment 4: A Failure Case - When the Coverage of the Collected Data Is Poor

Our approach is not robust to all environments and data collection schemes. In this section we investigate when it can fail. One obvious way our approach can fail is if the agent's performance in the calibration model is always the same: no matter what hyperparameter we try, the system thinks they all perform similarly. To illustrate this phenomenon, we conduct the following experiment.

Experiment Description

Following are the experiment specifications:

Environment: We conducted the experiment in the Cartpole environment. In Cartpole the agent must balance a pole in an unstable equilibrium as long as it can. If the cart reaches the end of the track or the pole reaches a critical angle, failure is inevitable. In Cartpole, near-optimal policies can balance the pole for hundreds of steps rarely experiencing failures or much of the state-space. A data log collected from the near-optimal policy would likely produce a calibration model where failures appear impossible and all hyperparameters appear excellent. Figure 5.6 plots the performance of the best hyperparameters selected according to the calibration model from three different policies, averaged over 30 runs.

Data collection: For data collection, we used a random policy, a near-optimal policy, a medium policy that was half as good as near-optimal. We collected 10,000 transitions using each policy for the datasets. Since the data logs with a near-optimal policy will hardly have any failures, we randomized the initial pole angles and cart position during data collection for all the three policies. Randomizing the initial pole angles and cart position and then collecting data also leads to better coverage of the underlying environment, in turn, giving more failures in the dataset to learn from. The datasets collected by the near-optimal, medium, and random policies had 40-50, 80-125, and 400-500 failures respectively.

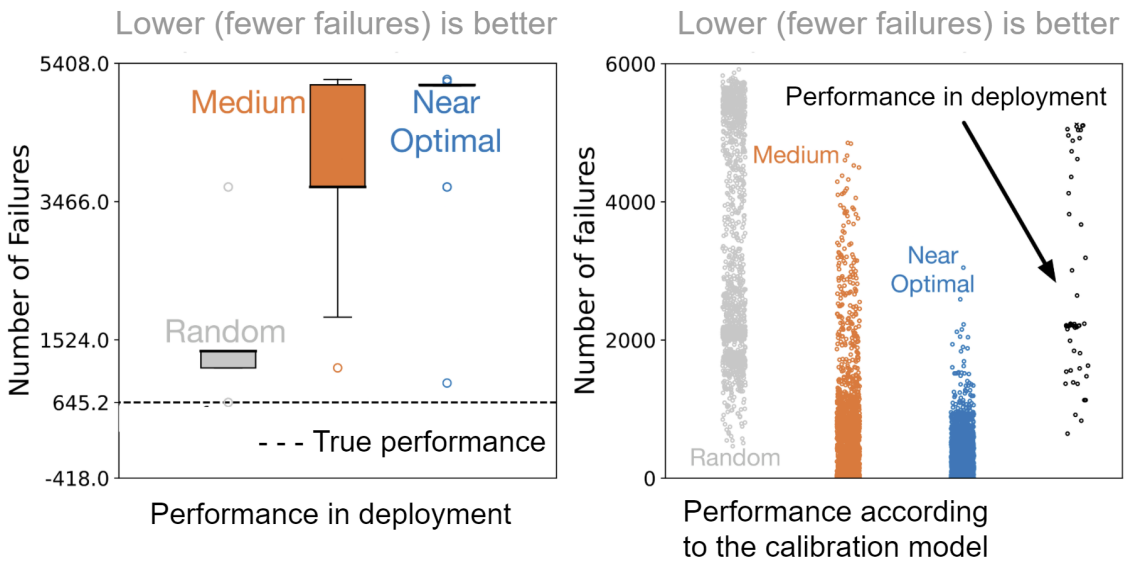

Figure 5.6: **Success and failure in Cartpole.** This plot shows performance of three different calibration models constructed from random, near-optimal and medium policies. **Left**: performance of the hyperparameters in deployment as selected by different calibration models—lower is better. **Right**: each model's evaluation of all hyperparameters across all 30 runs. Ideally the distribution of performance would match that of the hyperparameter performance in the deployment environment—black dots far right.

Model construction and representation training: The KNN calibration model was trained as described in Section 3.3. We used k=3 in the k-nearest neighbors calibration model. The Laplacian representation training details are described in Section 3.3.2. For the Laplacian representation training, we used $\kappa = 0.8$, $\beta = 5$, $\delta = 0.05$, $\alpha = 0.00003$, and the length of trajectory n for picking nearby states was set to 50 steps.

Agents: We used the Expected Sarsa agent in both the experiments. In both the experiments, we trained and evaluated the agent in the calibration model as well as in the deployment environment for 50,000 timesteps.

Hyperparameters used in the grid search: For the grid search, we used the hyperparameter values specified in Section 4.3.

Results and Discussion

In Figure 5.6, since the Y-axis corresponds to the average number of failures (pole drops or cart goes off the track), lower number on Y-axis corresponds to better performance. The left subfigure shows the results of the performance of our strategy for different data collecting policies. The grey box plot corresponds to the random policy, the orange box plot corresponds to the medium policy, and the blue box plot corresponds to the near-optimal policy. The right subfigure shows the performance of every hyperparameter combination in the grid search for every run inside the calibration model. The grey, orange, and blue scatter plots show how the hyperparameters perform inside the calibration model for the random, medium, and the near-optimal policies respectively. The black scatter plot shows the true performance of every hyperparameter combination in deployment. A good calibration model should have a somewhat similar pattern to the black scatter plot. However, it is clear how the calibration model formed using data from the near-optimal and medium policies causes the environment to appear easier than it actually is, in many cases leading to zero failures in the simulated online runs inside the calibration model.

There are two types of failures in Cartpole: one is when the pole drops (angular failure) and the other is the cart goes too far and out of the range (positional failure). The near-optimal policy has only positional failures. The medium policy tends to have both types of failures, though there are more positional failures than angular failures. On the other hand, the random policy predominantly has angular failures. The failure that the agent sees most often in the early learning stage is the angular failure. As a result of this, the calibration model formed using the data collected by the random policy has more useful information as it mimics what the agent is more likely to see in the true environment during early learning. This is followed by the medium policy and then by the near-optimal policy. As per the left subfigure, the random policy data gave the best performance, followed by the medium policy

and then the near-optimal policy. In the right subfigure, we see a similar pattern in the performance of the hyperparameters inside the respective calibration models. Performance in the calibration model under the random policy was most similar to the performance in deployment. This was followed by the medium policy and then the near-optimal policy.

It indeed shows that the dynamics of Cartpole combined with particular data collection policies can render the calibration model ineffective. Even with random starting states the calibration model for near-optimal policy rarely simulated dropping the pole. The random policy produced the best calibration model but it still could not identify the best hyperparameters. Unsurprisingly, the random policy drops the pole every few steps and thus the log contained many failures but high state coverage—explaining why the performance was better than near-optimal. One could certainly argue that many applications might not exhibit this behavior—especially since it is largely caused by a task with two modes of operation (failing or balancing). Additionally, using a random policy to achieve coverage is unrealistic in real-world applications like water treatment.

5.1.5 Summary

Section 5.1 described the experiments that highlight how our solution strategy performs in its ability to select good hyperparameters from a set of many hyperparameters. In specific, Section 5.1.1 described experiments that demonstrate superior performance of our strategy in comparison to the naive random hyperparameter selection baseline and the offline RL baseline (FQI). Section 5.1.1 also demonstrated how our solution strategy worked well across different environments and different learning agents. The experiments also highlighted how different calibration models compare with each other. Furthermore in Section 5.1.2, we demonstrated how our strategy is fairly insensitive to different data collecting policies and different dataset sizes, and performed well across different data regimes. Next, Section 5.1.3 demonstrated an experiment that

highlighted how transferring hyperparameters and learning from scratch in a changed deployment environment outperforms transferring a fixed policy learned from the offline data logs. Finally, Section 5.1.4 describes an experiment that highlights a scenario where our strategy fails to perform good. This failure case happens when the calibration model evaluates all hyperparameters—good or bad— with nearly equal performance.

5.2 Automated Offline Hyperparameter Fine-Tuning Using Cross Entropy Method for Optimization

The calibration model is an offline artifact that we can use as we like without impacting the deployment environment. In the experiments in Section 5.1, we discussed how to select good hyperparameters from a set of many hyperparameters using grid search. This involved evaluating each hyperparameter combination independently in the calibration model and finally selecting the best from a finite set of discrete hyperparameters. However, this is limiting because grid search scales very poorly as we increase the total number of dimensions to sweep or add more values to sweep along each dimension. Often, researchers end up doing several sequential cycles of grid search trying to find better hyperparameters. As a result, sweeping very fine values of hyperparameters becomes computationally expensive. Fortunately, we can do better than grid search. We can use the model in smarter ways to discover and evaluate good hyperparameters for deployment. We can have a hyperparameter optimization technique that fine-tunes the hyperparameters, is fully automated, scales well with more dimensions, and is more resource efficient than grid search.

In this section we discuss and evaluate how to use a simple black-box optimization strategy based on cross-entropy method for optimization (CEM) to search the model for high performing hyperparameters. Instead of evaluating a discrete set as in grid search, we can search a continuous space, exploiting smoothness of the performance surface, to improve performance in deployment.

CEM Algorithm:

CEM is a family of algorithms, where the general principle is to (1) maintain a distribution over plausibly maximal inputs, and (2) sample from that distribution and progressively concentrate around the maximal input. At a high-level, the CEM algorithm starts with some initial probability distribution over the hyperparameters. In each iteration, it samples some points (hyperparameters) using the distribution. We evaluate the online learning performance of these sampled hyperparameters in the calibration model. We then sort the hyperparameters based on their performance in the calibration model in a descending order, and select the top few as the *elite* hyperparameters. We then use these elite hyperparameters to nudge the probability distribution (that is used to sample hyperparameters) in the direction of the elite (high performing) hyperparameters. This cycle of sampling hyperparameters from the distribution, evaluating the sampled hyperparameters in the calibration model, creating the elite set of hyperparameters, and then re-constructing the probability distribution using the elite hyperparameters can go on forever. With each iteration of this cycle, the probability distribution moves closer towards the optimal hyperparameter (with the highest online learning performance in the calibration model). At convergence, the probability distribution will concentrate at the optimal hyperparameter. Figure 5.7 shows how the CEM algorithm tunes hyperparameters in the calibration model.

We need a blackbox optimizer, because our function—the performance of a hyperparameter λ in the calibration model—is not differentiable. Our setting has two nuances compared to the typical setting where CEM is used: 1) our function is expensive to query, and 2) we only get a stochastic sample. We provide a modified CEM algorithm, that still reflects the general framework for CEM, but using an incremental update—similar to stochastic gradient descent—to account for the stochasticity in our function query. We describe the detailed CEM algorithm in the Algorithm 10.

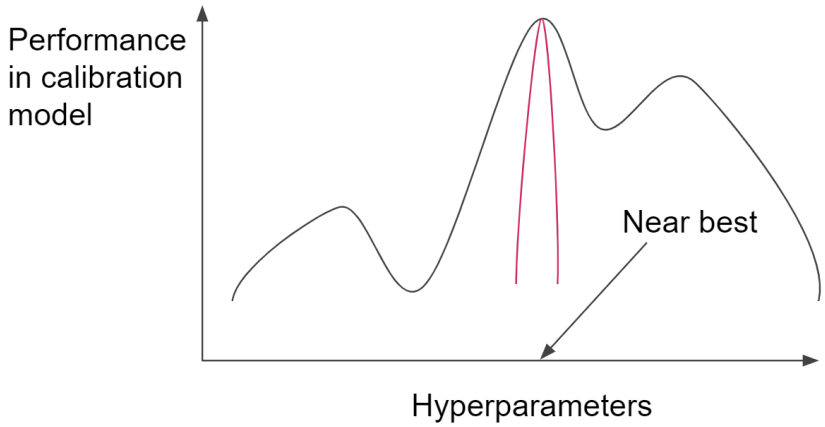

Figure 5.7: **Using CEM with the calibration model to tune hyperparameters**: The x-axis of the plot represents different hyperparameters and the y-axis represents the performance distribution of these hyperparameters in the calibration model. The performance distribution will have several local maxima. Here, the CEM algorithm maintains a pink colored truncated multivariate normal distribution (TMVN). With each iteration of the CEM algorithm, the pink TMVN concentrates towards the global maxima of the underlying performance distribution. At convergence, the mean of the TMVN is the optimal hyperparameter and the variance of the TMVN is zero.

The CEM algorithm naturally works well with continuous hyperparameters like the learning rate, eligibility trace parameter λ_e, ϵ in ϵ-greedy exploration, etc. With continuous hyperparameters, it can exploit the smoothness of the distribution and sample real-valued hyperparameters easily. However, discrete hyperparameters like the number of tiles/tilings, number of hidden units/layers, type of activation function, etc. need special treatment since the underlying distribution is not continuous. We assume that the discrete hyperparameters are of two types: ordered and unordered. For ordered hyperparameters, like the number of tilings, we can treat these like continuous hyperparameters. For example, the range for the number of tilings having values $\{1, 2, 4, 8\}$ could be converted to $[0, 4)$. When we sample h from a continuous interval $[0, 4)$, we can round down h to the nearest integer and map this continuous real valued h back to a discrete value, that is, the number of tilings by using 2^h. We then query the performance for that discrete hyperparameter. We can see this as the agent taking in the hyperparameter $h \in [0, 4)$ and itself doing the rounding and exponentiation

Algorithm 10 CEM Algorithm

Input:
lower and *upper* ranges for each hyperparameter for a truncated multivariate normal (TMVN) distribution.

Fixed parameters: *max_iterations* = 100 for running the cross entropy method
$\alpha = 0.1$ learning rate for mean and covariance
tol = 10^{-1} tolerance for convergence
N_samples = 32 number of samples to draw from the distribution in each iteration for evaluation
N_top = 5 keep the top 15% of the hyperparameters to update the mean

$\mu \leftarrow$ mean of ranges
$C \leftarrow$ diagonal matrix with range width on diagonal
$\mu_{\text{ave}} \leftarrow 0$, $\mu_{\text{old-ave}} \leftarrow 2$ tol
iteration = 1
while iteration $<$ *max_iterations* & $\|\mu_{\text{ave}} - \mu_{\text{old-ave}}\| >$ tol **do**
    Sample *N_samples* hyperparameters from the distribution TMVN(μ, C, *lower*, *upper*)
    Get a noisy performance estimate of these *N_samples* hyperparameters by running the agent in the calibration model
    Compute the mean μ_{top} and covariance C_{top} of the top *N_top* hyperparameters
    $\mu \leftarrow (1-\alpha)\mu + \alpha\mu_{\text{top}}$
    $C \leftarrow (1-\alpha)C + \alpha C_{\text{top}}$
    $\mu_{\text{old-ave}} = \mu_{\text{ave}}$
    $\mu_{\text{ave}} = \mu_{\text{ave}} + \frac{1}{\text{iteration}}(\mu - \mu_{\text{ave}})$
    iteration $\leftarrow$ iteration + 1
return μ

as part of the agent. This transformation allows us to use generalization to reason about if the agent prefers a smaller or bigger number of tilings. For unordered hyperparameters like the types of activation functions in a neural network (*tanh*, *ReLU*, *Sigmoid*), the hyperparameters should be evaluated separately as in grid search for each of the unordered discrete settings, with CEM picking amongst the other ordered hyperparameters. This is because CEM is faster than grid search when there is the ability to generalize between hyperparameters. Without order, we will not have generalization. This approach allows for the simplicity of using a truncated Gaussian distribution, specifically a truncated multivariate normal (TMVN) distribution. This

multivariate normal allows us to maintain a covariance, which means we can more quickly discard joint hyperparameter settings that are poor and reduce our search space.

One other nuance is that we do not get a perfect sample of the performance $f(h)$ of a hyperparameter. Rather, we get a noisy estimate $\hat{f}(h)$ because we do a limited number of runs per hyperparameter to save on computation since the function is expensive to query. Instead, the approach is to reason across CEM iterations about what region of hyperparameters is effective. To do so, we use an incremental update to the parameters of the TMVN, similar to a stochastic gradient descent update. For a sufficiently small learning rate, the distribution will concentrate and will converge to a set of hyperparameters corresponding to the global maxima. We use a learning rate of 0.1 since across experiments we found this to be effective.

Finally, we use Polyak averaging [42, 43, 46], as a way to overcome the noise in iterates in SGD, especially for a fixed learning rate. We maintain a running average of the SGD iterates. When this average stabilizes, we consider the algorithm to have converged. CEM should give us hyperparameters that are fine-tuned in the calibration model, and thus should select near-optimal hyperparameters in the calibration model.

5.2.1 Experiment 5: Using CEM Algorithm with Calibration Models

In this experiment, we demonstrate how using a smart hyperparameter optimization technique like CEM compares with grid search when used with the calibration model in our strategy. This experiment compares the performance of the hyperparameters obtained by *fine-tuning* with CEM with the performance of hyperparameters *selected* using a sweep with grid search.

Experiment Description

Environments: We conducted experiments in two environments - 1) Puddle World, and 2) Acrobot.

Data collection: We used the same medium policy from Experiment 2 to collect 500 transitions of data in both Puddle World and Acrobot. The medium policy completed 8-10 episodes in Puddle World and 2-3 episodes in Acrobot in each of the 30 datasets.

Model construction and representation training: The KNN calibration model was trained as described in Section 3.3. We used k=3 in the k-nearest neighbors calibration model. The Laplacian representation training details are described in Section 3.3.2. For the Laplacian representation training in Puddle World, we used $\kappa = 0.8$, $\beta = 5$, $\delta = 0.05$, $\alpha = 0.0003$, and the length of trajectory n for picking nearby states was set to 10. For the Acrobot experiment, we chose $\kappa = 0.8$, $\beta = 5$, $\delta = 0.5$, $\alpha = 0.00003$, and the length of trajectory n for picking nearby states was set to 20.

Agents: We used the Expected Sarsa agent in both the experiments. In both the experiments, we trained and evaluated the agent in the calibration model as well as in the deployment environment for 15,000 timesteps.

Hyperparameters used in the grid search and CEM: In the Puddle World experiment, we optimized the temperature τ of the softmax policy and the Adam stepsize α as continuous values in the ranges [0.0001, 10.0] and (0.0, 1.0] respectively using CEM. We ran the CEM experiment for 100 iterations, and each iteration sampling 32 hyperparameters which were all evaluated for 5 runs to get the performance measure of each hyperparameter within that iteration. We ran the CEM experiment once on each of the 30 datasets, to finally get 30 hyperparameters tuned by CEM. The CEM box plot (pink) for Puddle World in Figure 5.8 corresponds to the true performance of these 30 hyperparameters in the real environment.

To compare against the CEM results for Puddle World, we also report the grid

search results (blue box plot) in Figure 5.8. We did a grid search over the following 30 hyperparameter combinations:

1. Softmax temperature τ: {1, 2, 4, 6, 8, 10}

2. Adam optimizer learning rate α: {0.001, 0.003, 0.01, 0.03, 0.1}

We kept the following hyperparameters fixed

1. Adam optimizer momentum $\beta_1 = 0.0$

2. Optimistic weight initialization $w_0 = 0.0$

3. Eligibility trace parameter $\lambda_e = 0.1$

In the Acrobot experiment, we again optimized the temperature τ and stepsize α as continuous values but in the ranges [0.0001, 5.0] and (0.0, 0.1] respectively using CEM. The CEM box plot (pink) for Acrobot is shown in Figure 5.9.

To compare against the CEM results for Acrobot, we also report the grid search results (blue box plot) in Figure 5.9. We did a grid search over the following 30 hyperparameter combinations:

1. Softmax temperature τ: {0.0001, 1, 2, 3, 4, 5}

2. Adam optimizer learning rate α: {0.001, 0.003, 0.01, 0.03, 0.1}

We kept the following hyperparameters fixed

1. Adam optimizer momentum $\beta_1 = 0.0$

2. Optimistic weight initialization $w_0 = 0.0$

3. Eligibility trace parameter $\lambda_e = 0.8$

All other CEM algorithm details in the Acrobot experiment are the same as in the Puddle World experiment as described above.

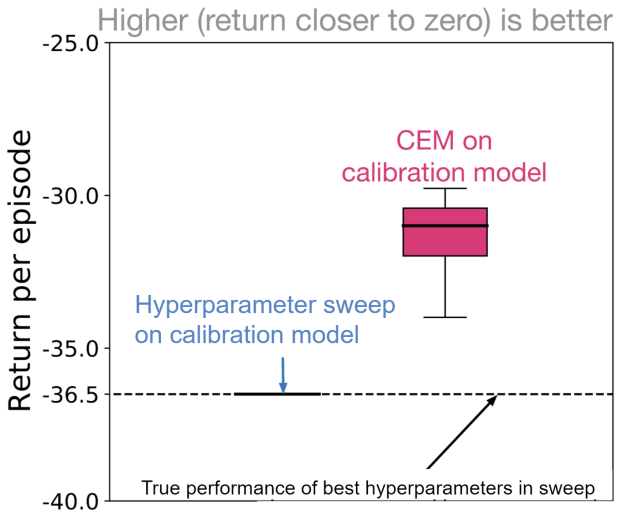

Figure 5.8: **Combining calibration model with CEM in Puddle World.** The performance of hyperparameter chosen by CEM in calibration model compared with hyperparameter sweeping in the calibration model. The y-axis is same as the one in Figure 5.1.

Results and Discussion

Figure 5.8 shows the Puddle World experiment results. The Y-axis corresponds to the average return per episode; higher numbers on Y-axis corresponds to better performance. The true performance of the best hyperparameter in the grid search is given by the dashed horizontal line. The blue box plot corresponds to the results when using the calibration model with grid search. In all the 30 runs of the experiment, our strategy always selects the best hyperparameter within the grid search. This is evident as the blue box plot has completely collapsed onto the dashed horizontal line, has zero variance, and thus consistently selects the best hyperparameter. However, these are the limits of using the calibration model with the grid search. The grid search strategy's performance is only as good as the quality of hyperparameters specified in the set of hyperparameters to sweep over.

On the other hand, the pink box plot corresponds to the results when using the calibration model with CEM. The performance improvements are stark when using CEM as against grid search. CEM can search this continuous space efficiently, because it uses a truncated multivariate normal that allows for generalization about the utility

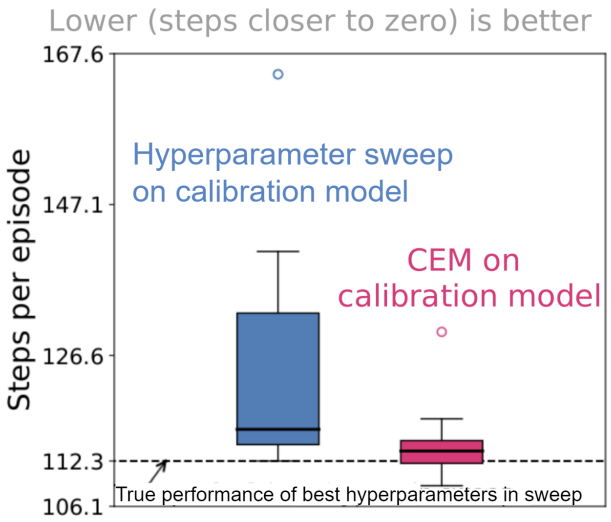

Figure 5.9: **Combining calibration model with CEM in Acrobot.** The performance of hyperparameter chosen by CEM in calibration model compared with hyperparameter sweeping in the calibration model. The y-axis is same as the one in Figure 5.1.

of hyperparameter values. Even when tuning only on the calibration model, the agent can significantly outperform the best hyperparameters found by a grid search on the true environment—this is why CEM is outperforming the best hyperparameters shown by the dashed line in Figure 5.8 for every single of the 30 runs.

Similar results are seen in the Acrobot experiment in Figure 5.9. Since the Y-axis corresponds to the average number of steps per episode, lower number on Y-axis corresponds to better performance. CEM outperforms grid search in this experiment as well.

The benefits of using CEM over a technique like grid search are very compelling. It is fully automated with no manual intervention as is usually needed when doing multiple sequential iterations of grid search. CEM gives better tuned hyperparameters as is demonstrated from the above experiments. Finally, CEM is more resource efficient than grid search and scales better than grid search when more dimensions are added.

5.2.2 Summary

Up until Section 5.1, we discussed how to select good hyperparameters from a set of hyperparameters using grid search. Section 5.2 builds upon that idea and extends it to fine-tuning the hyperparameters in the calibration model. Section 5.2 describes a hyperparameter optimization technique called CEM for fine-tuning the hyperparameters. It describes an experiment that demonstrates how CEM can further improve the performance of our strategy if we use it in place of grid search. This simply shows that smarter hyperparameter optimization techniques work well with the calibration model and can further help improve the performance of our strategy. We propose using any fine-tuning optimization technique like CEM, Bayesian optimization, etc. instead of grid search to take full advantage of the calibration model to identify near-optimal hyperparameters.

Chapter 6

Conclusion & Future Work

In this work, we introduced the Data2Online problem: selecting hyperparameters from a log of data, for deploying an online learning agent in a real environment. We then also introduced our solution strategy to tackle the Data2Online problem. The basic idea is to learn a calibration model from the data log, and then allow the agent to interact in the calibration model to identify good hyperparameters. Essentially, the calibration model is treated just like the real environment. We provided a simple approach, using k-nearest neighbors, to obtain a calibration model that is stable under many model iterations and only produces real states and rewards. We then conducted a battery of tests under different data regimes to demonstrate the effectiveness of our strategy.

As we highlight throughout, this problem should be more feasible than offline RL, which requires the entire policy to be identified from a log rather than just suitable hyperparameters for learning. Transferring a fixed learned policy also needs a very accurate model of the environment as in Sim2Real. Our strategy employing calibration models does not have the same constraint. Our own experiments highlighted that offline methods that attempted to learn and deploy a fixed policy performed poorly, whereas identifying reasonable hyperparameters was a much easier problem with consistently good performance across different deployment scenarios, many data collecting policies, and even small datasets. Nonetheless, we did identify one failure case, where the

data resulted in a calibration model that made the environment appear too easy and so most hyperparameters looked similar. When applying the calibration model in practice, we suggest paying attention to the coverage of the data log. The state and action space coverage in the data logs, in turn, affect how closely the calibration model resembles the underlying environment. However, as per our experiments, low coverage was less problematic for transferring hyperparameters than it was for transferring a fixed policy.

Our complete proposed strategy to solve the Data2Online problem is to create a stable KNN based calibration model from the offline logs of data, then train a distance metric based on the Laplacian representation to be used with the calibration model, and finally use a smarter hyperparameter optimization strategy like CEM for fine-tuning the hyperparameters in order to deploy them in the real world. However, all components of our approach —KNN calibration model, CEM algorithm, or the Laplacian representation based distance metric— can be independently swapped with better alternatives in the future, making the strategy modular and quite flexible. Improving our strategy along any of these dimensions should result in improved performance, that is, better hyperparameter selection. Through our experiments, we demonstrated improvements in performance through this modularity by replacing 1) the NN calibration model with the KNN calibration model, 2) distance metric based on raw input states with the one based on the Laplacian representation, and 3) grid search with the CEM algorithm.

In this thesis, in all our experiments, we treated the deployment environment as if our solution strategy did not have access to it, that is, treating it just like a real world system. However, the environments in our experiments did not capture the complexity that comes with real world systems, such as, having hundreds of sensors, several mixed actions (some discrete and some continuous) - all having different action cycle times, latency in perception and actuation, etc. So the next direction would be to test whether the strategy we developed works in a complex real world system.

This thesis research was inspired by a project on optimizing a water treatment plant using RL [56]. The research direction of this thesis emerged from the need to tackle the problem of deploying an RL agent to control this water treatment plant, which is a real world system. So as part of the future work, implementing the strategy we developed in this thesis to solve the deployment problem of the water treatment plant would be an important direction.

In this thesis we make the case that transferring hyperparameters and learning from scratch is a more feasible problem to solve than transferring a fixed policy when we only have access to offline logs of data. More work needs to be done to see how transferring hyperparameters compares with transferring a learned policy but now also allowing this policy to learn online instead of keeping it fixed.

In our proposed strategy, we transfer the hyperparameters in the deployment environment and learn from scratch. However, some sensitive deployment environments will not allow learning from scratch due to safety reasons. More work needs to be done to develop a strategy to deploy good hyperparameters along with a good policy initialization.

A minor direction would be to test using ensemble models in place of a single calibration model that we use in this thesis. By obtaining performance across an ensemble of calibration models, we can make the hyperparameter selection more robust to error in the calibration model. The hyperparameter performance can either be averaged across these models, or a more risk-averse criterion like worst-case performance could be used. We might also want to select hyperparameters more conservatively, and instead of using average return as the performance measure, we may want to test with median return. A theoretical direction that could be taken is to investigate why even with inaccuracies in the calibration model, the performance of our strategy is good as long as the relative ranking of hyperparameters in the calibration model is somewhat correct. Another direction could be to see how the coverage of the dataset affects the performance distribution of the hyperparameters in

the calibration model.

Naturally, as the first work explicitly tackling this problem, we have only scratched the surface of options. There is much more to understand about when our solution strategy will be effective, and when it might fail. Much more work can be done, theoretically and empirically, to understand the Data2Online problem.

Bibliography

[1] Z. Abbas, S. Sokota, E. Talvitie, and M. White, "Selective Dyna-style planning under limited model capacity," in *International Conference on Machine Learning*, 2020.

[2] A. Agarwal, N. Jiang, and S. Kakade, "Reinforcement learning: Theory and algorithms," 2019.

[3] A. Ajay, A. Kumar, P. Agrawal, S. Levine, and O. Nachum, "OPAL: Offline primitive discovery for accelerating offline reinforcement learning," *arXiv preprint arXiv:2010.13611*, 2020.

[4] J. Albus, "A theory of cerebellar function," *Mathematical Biosciences*, 1971.

[5] J. Albus, "Brains, behavior, and robotics," *BYTE Books, Peterborough, N.H.*, 1981.

[6] P. Barde, J. Roy, W. Jeon, J. Pineau, C. Pal, and D. Nowrouzezahrai, "Adversarial soft advantage fitting: Imitation learning without policy optimization," in *Advances in Neural Information Processing Systems*, 2020.

[7] A. G. Barto, R. S. Sutton, and C. W. Anderson, "Neuronlike adaptive elements that can solve difficult learning control problems," *IEEE Transactions on Systems, Man, and Cybernetics*, 1983.

[8] F. Behbahani, K. Shiarlis, X. Chen, V. Kurin, S. Kasewa, C. Stirbu, J. Gomes, S. Paul, F. A. Oliehoek, J. Messias, *et al.*, "Learning from demonstration in the wild," in *International Conference on Robotics and Automation*, 2019.

[9] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Communications of the ACM*, 1975.

[10] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization," *The Journal of Machine Learning Research*, 2012.

[11] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "OpenAI Gym," *arXiv preprint arXiv:1606.01540*, 2016.

[12] Y. Chandak, S. M. Jordan, G. Theocharous, M. White, and P. S. Thomas, "Towards safe policy improvement for non-stationary MDPs," *Advances in Neural Information Processing Systems*, 2020.

[13] Y. Chandak, G. Theocharous, S. Shanka, M. White, S. Mahadevan, and P. S. Thomas, "Optimizing for the future in non-stationary MDPs," *International Conference on Machine Learning*, 2020.

[14] V. Chelu, D. Precup, and H. P. van Hasselt, "Forethought and hindsight in credit assignment," in *Advances in Neural Information Processing Systems*, 2020.

[15] C. Downey and S. Sanner, "Temporal difference Bayesian model averaging: A Bayesian perspective on adapting Lambda," in *International Conference on Machine Learning*, 2010.

[16] D. Ernst, P. Geurts, and L. Wehenkel, "Tree-based batch mode reinforcement learning," *The Journal of Machine Learning Research*, 2005.

[17] S. Falkner, A. Klein, and F. Hutter, "BOHB: Robust and efficient hyperparameter optimization at scale," in *International Conference on Machine Learning*, 2018.

[18] A. M. Farahmand, M. Ghavamzadeh, C. Szepesvári, and S. Mannor, "Regularized Fitted Q-iteration for planning in continuous-space Markovian decision problems," in *Proceedings of the American Control Conference*, 2009.

[19] C. Finn, T. Yu, T. Zhang, P. Abbeel, and S. Levine, "One-shot visual imitation learning via meta-learning," in *Conference on Robot Learning*, 2017.

[20] S. K. S. Ghasemipour, R. Zemel, and S. Gu, "A divergence minimization perspective on imitation learning methods," in *Conference on Robot Learning*, 2020.

[21] S. Ghiassian, B. Rafiee, Y. L. Lo, and A. White, "Improving performance in reinforcement learning by breaking generalization in neural networks," in *Proceedings of the International Conference on Autonomous Agents and MultiAgent Systems*, 2020.

[22] P. Henderson, R. Islam, P. Bachman, J. Pineau, D. Precup, and D. Meger, "Deep reinforcement learning that matters," in *Proceedings of the AAAI Conference on Artificial Intelligence*, 2018.

[23] I. Higgins, A. Pal, A. Rusu, L. Matthey, C. Burgess, A. Pritzel, M. Botvinick, C. Blundell, and A. Lerchner, "DARLA: Improving zero-shot transfer in reinforcement learning," in *International Conference on Machine Learning*, 2017.

[24] A. Jacobsen, M. Schlegel, C. Linke, T. Degris, A. White, and M. White, "Meta-descent for online, continual prediction," *Proceedings of the AAAI Conference on Artificial Intelligence*, 2019.

[25] M. Jaderberg, V. Dalibard, S. Osindero, W. Czarnecki, J. Donahue, A. Razavi, O. Vinyals, T. Green, I. Dunning, K. Simonyan, C. Fernando, and K. Kavukcuoglu, "Population based training of neural networks," *arXiv preprint arXiv:1711.09846*, 2017.

[26] T. Jafferjee, E. Imani, E. Talvitie, M. White, and M. Bowling, "Hallucinating value: A pitfall of Dyna-style planning with imperfect environment models," *arXiv preprint arXiv:2006.04363*, 2020.

[27] H. Kimura and S. Kobayashi, "An analysis of Actor/Critic algorithms using eligibility traces: Reinforcement learning with imperfect value function," in *International Conference on Machine Learning*, 1998.

[28] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *International Conference on Learning Representations*, 2015.

[29] L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar, "Hyperband: A novel bandit-based approach to hyperparameter optimization," *The Journal of Machine Learning Research*, 2017.

[30] V. Liu, R. Kumaraswamy, L. Le, and M. White, "The utility of sparse representations for control in reinforcement learning," *Proceedings of the AAAI Conference on Artificial Intelligence*, 2019.

[31] S. Mahadevan and M. Maggioni, "Proto-value functions: A Laplacian framework for learning representation and control in Markov decision processes.," *The Journal of Machine Learning Research*, 2007.

[32] A. R. Mahmood, D. Korenkevych, B. Komer, and J. Bergstra, "Setting up a reinforcement learning task with a real-world robot," *International Conference on Intelligent Robots and Systems*, 2018.

[33] A. R. Mahmood, D. Korenkevych, G. Vasan, W. Ma, and J. Bergstra, "Benchmarking reinforcement learning algorithms on real-world robots," in *Conference on Robot Learning*, 2018.

[34] G. Manek and J. Z. Kolter, "Learning stable deep dynamics models," *Advances in Neural Information Processing Systems*, 2019.

[35] T. A. Mann, H. Penedones, S. Mannor, and T. Hester, "Adaptive Lambda least-squares temporal difference learning," *arXiv preprint arXiv:1612.09465*, 2016.

[36] J. Merel, Y. Tassa, D. TB, S. Srinivasan, J. Lemmon, Z. Wang, G. Wayne, and N. Heess, "Learning human behaviors from motion capture by adversarial imitation," *arXiv preprint arXiv:1707.02201*, 2017.

[37] T. L. Paine, C. Paduraru, A. Michi, C. Gulcehre, K. Zolna, A. Novikov, Z. Wang, and N. de Freitas, "Hyperparameter selection for offline reinforcement learning," *arXiv preprint arXiv:2007.09055*, 2020.

[38] M. Papini, M. Pirotta, and M. Restelli, "Smoothing policies and safe policy gradients," *arXiv preprint arXiv:1905.03231*, 2019.

[39] J. Parker-Holder, V. Nguyen, and S. Roberts, "Provably efficient online hyperparameter optimization with population-based bandits," *Advances in Neural Information Processing Systems*, 2020.

[40] S. Paul, V. Kurin, and S. Whiteson, "Fast efficient hyperparameter tuning for policy gradient methods," *Advances in Neural Information Processing Systems*, 2019.

[41] X. B. Peng, M. Andrychowicz, W. Zaremba, and P. Abbeel, "Sim-to-real transfer of robotic control with dynamics randomization," in *IEEE International Conference on Robotics and Automation*, 2018.

[42] B. Polyak, "New stochastic approximation type procedures," *Avtomatica i Telemekhanika*, 1990.

[43] B. Polyak and A. Juditsky, "Acceleration of stochastic approximation by averaging," *SIAM Journal on Control and Optimization*, 1992.

[44] H. Ravichandar, A. S. Polydoros, S. Chernova, and A. Billard, "Recent advances in robot learning from demonstration," *Annual Review of Control, Robotics, and Autonomous Systems*, 2020.

[45] M. Riedmiller, "Neural Fitted Q Iteration – first experiences with a data efficient neural reinforcement learning method," in *European Conference on Machine Learning*, 2005.

[46] D. Ruppert, "Efficient estimations from a slowly convergent Robbins-Monro process," *Technical report*, 1988.

[47] H. van Seijen, H. Van Hasselt, S. Whiteson, and M. Wiering, "A theoretical and empirical analysis of Expected Sarsa," in *IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning*, 2009.

[48] J. Snoek, H. Larochelle, and R. P. Adams, "Practical Bayesian optimization of machine learning algorithms," in *Advances in Neural Information Processing Systems*, 2012.

[49] N. Srinivas, A. Krause, S. Kakade, and M. Seeger, "Gaussian process optimization in the bandit setting: No regret and experimental design," in *International Conference on Machine Learning*, 2010.

[50] R. S. Sutton, "Adapting bias by gradient descent: An incremental version of delta-bar-delta," in *Proceedings of the AAAI Conference on Artificial Intelligence*, 1992.

[51] R. S. Sutton, "Generalization in reinforcement learning: Successful examples using sparse coarse coding," *Advances in Neural Information Processing Systems*, 1996.

[52] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, CC BY-NC-ND 2.0 Generic License, 2018.

[53] R. S. Sutton, "Temporal credit assignment in reinforcement learning," PhD thesis, 1984.

[54] E. Talvitie, "Self-correcting models for model-based reinforcement learning," in *Proceedings of the AAAI Conference on Artificial Intelligence*, 2017.

[55] Y. Tang and K. Choromanski, "Online hyper-parameter tuning in off-policy learning via evolutionary strategies," *arXiv preprint arXiv:2006.07554*, 2020.

[56] *AI pilot project aims to make water treatment greener, more efficient and less expensive*, https://www.ualberta.ca/folio/2020/10/ai-pilot-project-aims-to-make-water-treatment-greener-more-efficient-and-less-expensive.html.

[57] J. Tobin, R. Fong, A. Ray, J. Schneider, W. Zaremba, and P. Abbeel, "Domain randomization for transferring deep neural networks from simulation to the real world," 2017.

[58] R. Wang, D. P. Foster, and S. M. Kakade, "What are the statistical limits of offline RL with linear function approximation?" *International Conference on Learning Representations*, 2021.

[59] C. Watkins, "Learning from delayed rewards," PhD thesis, 1989.

[60] C. Watkins and P. Dayan, "Technical note: Q-learning," *Machine Learning*, 1992.

[61] M. White, "Unifying task specification in reinforcement learning," in *International Conference on Machine Learning*, 2017.

[62] M. White and A. White, "A greedy approach to adapting the trace parameter for temporal difference learning," in *Proceedings of the International Conference on Autonomous Agents & Multiagent Systems*, 2016.

[63] Y. Wu, G. Tucker, and O. Nachum, "Behavior regularized offline reinforcement learning," *arXiv preprint arXiv:1911.11361*, 2019.

[64] Y. Wu, G. Tucker, and O. Nachum, "The Laplacian in RL: Learning representations with efficient approximations," *International Conference on Learning Representations, arXiv:1810.04586, arXiv.org perpetual, non-exclusive license*, 2019.

[65] J. Xing, T. Nagata, K. Chen, X. Zou, E. Neftci, and J. L. Krichmar, "Domain adaptation in reinforcement learning via latent unified state representation," *Proceedings of the AAAI Conference on Artificial Intelligence*, 2021.

[66] Z. Xu, H. van Hasselt, and D. Silver, "Meta-gradient reinforcement learning," in *Advances in Neural Information Processing Systems*, 2018.

[67] M. Yang, B. Dai, O. Nachum, G. Tucker, and D. Schuurmans, "Offline policy selection under uncertainty," *arXiv preprint arXiv:2012.06919*, 2020.

[68] T. Zahavy, Z. Xu, V. Veeriah, M. Hessel, J. Oh, H. P. van Hasselt, D. Silver, and S. Singh, "A self-tuning Actor-Critic algorithm," *Advances in Neural Information Processing Systems*, 2020.

[69] W. Zhao, J. Peña Queralta, and T. Westerlund, "Sim-to-real transfer in deep reinforcement learning for robotics: A survey," *IEEE Symposium Series on Computational Intelligence*, 2020.