Calibration Models for Real-World Deployment of Reinforcement Learning Agents

by

Jordan Coblin

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 sensitivity of reinforcement learning algorithm performance to hyperparameter choices poses a significant hurdle to the deployment of these algorithms in the real-world, where sampling can be limited by speed, safety, or other system constraints. To mitigate this, one approach is to learn a *calibration model* from offline data logs, and use this model to simulate trajectories for the purpose of hyperparameter tuning. While there has been preliminary success applying calibration models to simple simulated problems, more work is needed to understand the desirable properties of such models and to test their feasibility in a real-world setting.

In this work, we take the first steps toward characterizing desirable properties of calibration models and provide the first application of a calibration model towards a real-world industrial prediction task. We investigate several measures that can be used to understand model quality and evaluate calibration model implementations according to these measures. The calibration models are then tested on a prediction task for sensors in a water treatment plant (WTP) located in Alberta, Canada. We find that various types of calibration models can be used to simulate simple environments, while generalizing models tend to collapse due to compounding prediction error in the more complex real-world setting. We show how a non-parametric k-nearest neighbors calibration model with a Laplacian distance metric is able to produce realistic rollouts over long-horizons in the WTP setting, and can be used successfully for hyperparameter tuning. Finally, we aim to bridge the gap towards real-world deployment and demonstrate how this model can be scaled to a year's worth of data.

"All models are wrong, but some are useful."

-George E.P. Box

To my parents and my grandparents, who made this life possible and have enriched it in so many ways.

Acknowledgements

First and foremost, I'd like to thank my supervisor, Adam White, for being an invaluable mentor and constant supporter throughout my masters. Adam's dedication to conducting good science and his commitment to helping his students succeed contributed greatly to my growth as a researcher.

I'd also like to thank Martha White, for all of her support and expertise on the water treatment project, where her ideas helped to form the basis for many of our methods and experiments. I never would have guessed when I was first introduced to reinforcement learning through the RL course on Coursera, that I would one day have the opportunity to work with both of the course's wonderful instructors. I also owe many thanks to my primary collaborator on this project, Han Wang, for her continuous guidance and extreme generosity with her time and attention.

Next up, I'd like to thank my brilliant friends and colleagues in the RLAI lab at the University of Alberta, as well as that peculiar breed of people who find themselves studying RL Theory. I've learned so much from each of them in both research and in life.

I thank my fellow misfits and grapplers at Bairro Jiu Jitsu, for all the shared blood, sweat, tears, and armbars. Despite the objective absurdity of the activity we choose to spend so much of our free time doing, I am rarely happier than after a couple of hours hard-spent on the mats with these folks.

Lastly, I'd like to thank you, dear reader, for taking the time to glance through my humble thesis. Amidst a seemingly infinite sea of information, you have somehow found your eyes dancing across these words. Enjoy!

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Reinforcement learning (RL) has been central to numerous breakthroughs in artificial intelligence (AI) in the past decade, where artificial agents trained using RL algorithms have surpassed human-level performance in games such as Atari (Mnih, Kavukcuoglu, Silver, Rusu, *et al.* 2015), Go (Silver *et al.* 2016), and StarCraft (Vinyals *et al.* 2019). However, the application of RL algorithms in real-world systems remains a complex endeavor, with significant challenges related to sampling limitations, partial observability, explainability, safety constraints, and so forth (Dulac-Arnold *et al.* 2021). One especially challenging aspect of real-world RL is the selection of hyperparameters, which are the parameters that govern the learning process of an RL algorithm. Hyperparameters are typically set by a designer before training begins, and can have a significant impact on the performance of an algorithm (Henderson *et al.* 2018). The process of selecting hyperparameters is often done through a combination of intuition and extensive sweeps over configurations, where a single run could require tens of millions of environment samples (Bergstra and Bengio 2012). This process can be especially challenging in real-world settings, where the lack of a high-fidelity simulator can make hyperparameter tuning prohibitively slow and expensive. Further, the deployment of agents with poorly tuned hyperparameters can be risky, as it may lead to unnecessary financial cost or damage to the physical system, such as in the case of a water treatment plant dosing large amounts of chemicals into the system or crashes in helicopter control.

There are a variety of possible approaches for mitigating the hyperparameter tuning bottleneck, such as developing algorithms with robust default hyperparameters (Hafner, Pasukonis, *et al.* 2023), few or no hyperparameters at all (Jacobsen *et al.* 2019; Kingma and Ba 2014; M. White and A. White 2016; Zahavy *et al.* 2020), performing hyperparameter tuning online (Paul *et al.* 2019; Tang and Choromanski 2020), or using offline logs to evaluate an online agent (Mandel *et al.* 2016). In

Chapter 3, we argue that each of these approaches are either not mature enough or ill-suited to the task of evaluating an online learning algorithm over long horizons.

Alternatively, one could learn a model of the environment through offline logs, and use this model to simulate environment interactions for the purpose of hyperparameter tuning (H. Wang, Sakhadeo, *et al.* 2022). We will refer to such a model as a *calibration model*. Because many environment interactions are typically required in order to evaluate an RL algorithm, a calibration model is designed to be stable over long-horizon rollouts, where a rollout is a sequence of state-action pairs that are generated through agent-model interactions. Stability here is defined in the context of model-generated rollouts, where predicted states stay within a region close to the previous state and do not diverge to areas of the state space that are unreachable in the true environment (Talvitie 2017; H. Wang, Sakhadeo, *et al.* 2022). The calibration model is then used to evaluate hyperparameter configurations, offering a safe and efficient means of performing hyperparameter tuning. In this thesis, we investigate the feasibility of calibration models in both simulated and real-world settings, and explore different model implementations and evaluation metrics for the calibration model.

1.1 Contributions

The central goals of this thesis are to develop a better understanding of calibration models, and to test the calibration model approach in a real-world setting. We begin by asking what the important properties are of good calibration models, and whether we can design evaluation measures that roughly capture these properties. A recurring theme we will encounter is related to a model's stability under long-horizon rollouts. While planning in model-based RL is typically performed with rollouts of 10-100 steps (Chua *et al.* 2018; Hafner, Lillicrap, Norouzi, *et al.* 2020; Holland *et al.* 2019), evaluating a hyperparameter setting is typically on the order of thousands or even millions of timesteps. As a result, we posit that a model should be able to produce a *reasonable* trajectory over long-horizons, and offer several perspectives on how to characterize a reasonable trajectory. We investigate how different model implementations behave across environments, testing both discrete and continuous-action cases, and seek to validate assumptions around how different types of model inaccuracies affect model efficacy. Finally, we present a study on the application of calibration models towards a real-world task, where we perform hyperparameter selection for a prediction agent operating in a water treatment plant located in Alberta, Canada.

The main contributions of this thesis can be summarized as follows:

- We evaluate several measures for evaluating calibration models, including multi-step MSE, state-space distribution, and invalid transition count.
- We investigate several calibration model implementations: a feedforward neural network, a recurrent neural network, a KNN model with Euclidean distance metric, and a KNN model with Laplacian distance metric. These models are evaluated against the proposed metrics in two simple environments.
- We provide the first experiments for the continuous-action KNN calibration model proposed in H. Wang, Sakhadeo, *et al.* (2022).
- We demonstrate the first application of calibration models to a real-world task and show how the KNN calibration model can be scaled up to a full year's worth of data.

Chapter 2

Background

In this chapter, we provide relevant background on several topics central to this thesis. This includes an overview of fundamental components of the reinforcement learning framework, as well as brief introductions to transition models, Laplacian representations, and dynamic time warping.

2.1 Reinforcement Learning

Reinforcement learning refers to the problem of learning how to act in an environment in order to maximize a cumulative scalar reward. The agent learns to interact with the environment through a series of actions, receiving feedback in the form of rewards and observations. The agent's goal is to learn a policy that maps states to actions in order to maximize the expected sum of rewards over time. RL has been successfully applied to a wide range of problems, including games (Mnih, Kavukcuoglu, Silver, Rusu, *et al.* 2015; Silver *et al.* 2016), robotics (Levine *et al.* 2016a), and recommendation systems (Afsar *et al.* 2022). In this section, we provide an overview of the key components of the RL framework, including Markov Decision Processes (MDPs), policies, value functions, function approximation, and general value functions.

2.1.1 Markov Decision Processes

An RL problem can be framed as a finite Markov Decision Process (MDP), where an agent interacts with an environment over a series of discrete timesteps. At each timestep t, the agent receives a representation of the environment's state $S_t \in \mathcal{S}$ and given this information, selects an action $A_t \in \mathcal{A}$ which is sent back to the environment. The environment then transitions to a new state $S_{t+1} \in \mathcal{S}$ and returns this state, along with a reward $R_{t+1} \in \mathcal{R}$ to the agent. For *episodic* tasks, this process repeats until the agent reaches a terminal state, at which point the episode ends and a new episode

begins. Alternatively, we also consider the setting where this interaction may continue indefinitely and does not break up into distinct episodes - we call these *continuing* tasks. More formally, an MDP can be defined as the 5-tuple $(\mathcal{S}, \mathcal{A}, p, r, \mu)$ where:

- $\mathcal{S}$ is the state space
- $\mathcal{A}$ is the action space
- $p : \mathcal{S} \times \mathcal{R} \times \mathcal{S} \times \mathcal{A} \to [0, 1]$ denotes the state transition probability function, which specifies the probability of transitioning to state s' given the previous state s and action a:

$$p(s', r|s, a) \doteq Pr\{S_t = s', R_t = r \mid S_{t-1} = s, A_{t-1} = a\}.$$

- $r : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ is the immediate reward function $r(s, a, s')$, providing the reward received after transitioning to state s' from state s with action a, and
- μ represents the distribution over all possible start states $s_0 \in \mathcal{S}$

An important property of an MDP is that it satisfies the *Markov property*, which requires that the dynamics of the environment depend only on the previous state S_{t-1} and action A_{t-1}, and not any previous states or actions (i.e. its history). That is,

$$Pr\{S_t \mid S_{t-1}, A_{t-1}\} = Pr\{S_t \mid S_0, A_0, ..., S_{t-1}, A_{t-1}\}.$$

In other words, the previous state is modeled as containing all the relevant information about the history of the agent-environment interaction needed for making decisions.

2.1.2 Policies and Value Functions

The agent's goal is to learn a policy $\pi : \mathcal{S} \to \mathcal{P}(\mathcal{A})$ that maximizes the expected sum of discounted reward over some time horizon $t = 0..T$:

$$G_t \doteq \sum_{k=0}^{T} \gamma^k R_{t+k+1}, \tag{2.1}$$

where $\gamma^t \in [0, 1]$ is a discount factor that controls how much influence future rewards have on the return, and can be used to keep the return from becoming infinite in the case that $T = \infty$. The policy that maximizes the expected return is known as the *optimal policy* π_*.

A key concept in RL is the *value function* $v_\pi : \mathcal{S} \to \mathbb{R}$, which reflects the expected return from state s, following policy π. Similarly, the *action-value function* q_π :

$\mathcal{S} \times \mathcal{A} \to \mathbb{R}$ reflects the expected return from taking action a in state s, following policy π. The value function of a state under a policy π can be expressed as:

$$v_\pi(s) \doteq \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s\right], \text{ for all } s \in \mathcal{S}. \tag{2.2}$$

Whereas the value function of a state-action pair (s, a) under a policy π can be expressed as:

$$\begin{aligned} q_\pi(s, a) &\doteq \mathbb{E}_\pi[G_t|S_t = s, A_t = a] \\ &= \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s, A_t = a\right], \text{ for all } s \in \mathcal{S} \text{ and } a \in \mathcal{A}. \end{aligned} \tag{2.3}$$

The optimal value functions $v_*(s)$ and $q_*(s, a)$ are defined as the value functions of the optimal policy π_*, where:

$$v_*(s) \doteq \max_\pi v_\pi(s) \quad \text{and} \quad q_*(s, a) \doteq \max_\pi q_\pi(s, a). \tag{2.4}$$

The optimal policy can be generated from the optimal value functions by selecting the action that maximizes the value function at the next state (i.e. *greedy action selection*).

A key feature of returns and value functions is that they can be expressed recursively:

$$\begin{aligned} v_\pi(s) &\doteq \mathbb{E}_\pi[G_t|S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1}|s_t = s]. \end{aligned} \tag{2.5}$$

Using this recursive definition and expanding out the expectation, we can express the value $v_\pi(s)$ of a state in terms of the values of its successor states $v_\pi(s')$, the agent's policy $\pi(a, s)$, and the transition function $p(s', r|s, a)$, through what is known as a Bellman equation:

$$v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r|s, a) \left[r + \gamma v_\pi(s')\right], \text{ for all } s \in \mathcal{S}. \tag{2.6}$$

The Bellman equations can be solved to find the value function of a policy, and can also be used to derive iterative algorithms for finding the optimal policy.

2.1.3 Temporal-Difference Learning

One important class of methods that do not require a model of the environment is known as *temporal-difference* (TD) learning. When the transition function is known,

that is, that the agent has access to a perfect model of the environment, the optimal policy can be found using dynamic programming to solve the Bellman equation. However, in many cases the true transition function is unknown, and the agent must try to learn the optimal policy through trial-and-error. TD methods use a combination of sampling and bootstrapping to learn the value function of a policy in the absence of an environment model. Sampling refers to the process of collecting experience by interacting with the environment, while bootstrapping refers to the process of updating value function estimates of a given state based on estimates of the value function of successor states.

One of the simplest TD algorithms is known as TD(0), which uses a single-step bootstrapping approach for policy evaluation (also referred to as the *prediction problem*). The TD(0) update rule is given by:

$$V(S_t) \leftarrow V(S_t) + \alpha \left[R_{t+1} + \gamma V(S_{t+1}) - V(S_t) \right]. \tag{2.7}$$

Here we use V to denote a tabular estimate of the value function v_π, and $\alpha \in \mathbb{R}^+$ is a learning rate parameter that controls the size of the update. This update is performed after every step of agent-environment interaction, and can be shown to converge to the true value function (Sutton 1988).

The TD(0) prediction algorithm can be extended to the control setting by using the estimated value function to select actions. Of particular interest is the off-policy Q-learning algorithm, which uses the estimated value function to select actions greedily, and updates the value function according to the following update rule:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_{a \in \mathcal{A}} Q(S_{t+1}, a) - Q(S_t, A_t) \right]. \tag{2.8}$$

Here Q directly approximates the optimal action-value function q_*, independent of the agent's policy. A convergence guarantee for Q-learning exists as long as the general requirement for finding optimal policies is met, namely that all state-action pairs continue to be visited and updated.

2.1.4 Function Approximation

In order to extend RL methods to larger state spaces, we will move towards approximate solution methods, also known as *function approximation*. In theory, we can store the learned value approximations of every state in a single table, such that an exact representation of the value function is maintained at all times. This is known as a *tabular* representation of the value function. However, in many cases, the state

space $\mathcal{S}$ is too large to store or update the value function for all states in a tabular fashion. In these cases, we can use function approximation to learn a parameterized value function $\hat{v}(s, \mathbf{w}) \approx v_\pi(s)$, where $\mathbf{w} \in \mathbb{R}^d$ is a weight vector of parameters. This allows us to generalize the value function to states that have not been visited before, and to update the value function in a more efficient manner. For example, we can use a neural network to approximate the value function, and update the weights of the network using stochastic gradient descent (SGD). A general SGD update for state-value prediction can be given by:

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha \left[U_t - \hat{v}(S_t, \mathbf{w}_t) \right] \nabla \hat{v}(S_t, \mathbf{w}_t), \tag{2.9}$$

where U_t is an estimate of $v_\pi(S_t)$, $\alpha \in \mathbb{R}^+$ is a learning rate parameter, $\mathbf{w}_t$ are the function approximator parameters at timestep t, and $\hat{v}(S_t, \mathbf{w}_t)$ is the approximate value of state s given parameters $\mathbf{w}_t$. When U_t is an unbiased estimate of $v_\pi(S_t)$, this update rule is guaranteed to converge to a local optimum approximation of the true value function (Sutton and Barto 2018). Bootstrapped methods such as TD(0) and Q-learning can also be used with function approximation, by replacing the target U_t with the appropriate bootstrapped estimate of the value function, e.g. $U_t = R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}_t)$ for TD(0). However, these methods involve a biased estimate of $v_\pi(S_t)$, given that U_t is dependent on the weights $\mathbf{w}_t$. Such methods are referred to as *semi-gradient* methods, and are known to be more susceptible to instability and divergence compared with true gradient methods. Still, TD methods are often used in tandem with function approximation due to their computational efficiency and ability to be used in an online manner. An important instance of semi-gradient methods is the Deep Q-Network (DQN) algorithm (Mnih, Kavukcuoglu, Silver, Graves, *et al.* 2013), which uses a neural network to approximate the action-value function $q_\pi(s, a)$, and introduces techniques such as target networks and experience replay to aid with learning stability. We will use the DQN algorithm throughout the experiments in this thesis for learning policies.

2.1.5 General Value Functions

The value functions we've seen up to this point have been dependent solely on the agent's policy π. However, the value function can be further generalized in two other ways. First, we can generalize discounting by changing the scalar discount factor to a termination function $\gamma : \mathcal{S} \to [0, 1]$, where the discount rate now becomes a function of the state. Second, we can generalize the notion of a reward to encompass any arbitrary signal of interest, for example, individual features of the agent's state representation,

or predictions of these features. We refer to this signal $C_t \in \mathbb{R}$ as the *cumulant*. A *general value function* (GVF) (Sutton, Modayil, *et al.* 2011) encapsulates these modifications, resulting in the updated expression:

$$v_{\pi,\gamma,C}(s) \doteq \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} (\prod_{i=t+1}^{k} \gamma(S_i)) C_{t+k+1} \middle| \; S_t = s \right]. \tag{2.10}$$

An important feature of GVFs is that they allow us to express arbitrary prediction problems in a form that can be solved using standard RL techniques, such as TD-learning. Prediction of cumulant signals has been pursued as a type of *auxiliary task* for an agent, where the goal is to acquire knowledge of the world through tasks that are not limited to solely maximizing reward. This type of knowledge acquisition was explored in the Horde architecture (Sutton, Modayil, *et al.* 2011), and for multi-timescale predictions (i.e. *nexting*) in Modayil *et al.* (2012), where GVFs were used to formulate large collections of multi-step predictions of robot sensors. Some works even argue that world knowledge in general can be reduced to a rich set of these kinds of low-level predictions (Sutton 2009). In Chapter 6, we leverage the GVF framework to define a real-world prediction task, which acts as a testbed for our experiments.

2.2 Transition Models

A transition model (also sometimes referred to as a dynamics model, or world model) $\hat{p}(s', r|s, a)$ attempts to approximate the environment's true transition function $p(s', r|s, a)$. This approximate function is typically learned given observations from the true environment, $\mathcal{D} = \{(s_i, a_i, r_i, s'_i)_i\}$, where i is the index of the transition, collected by a (potentially unknown) behavior policy π_β. A model can be "rolled out" by seeding $\hat{p}$ with an initial state and action (s_0, a_0), and then using subsequent model predicted states $\hat{s}_t$ as input for later predictions, in an autoregressive fashion,

$$\hat{s}_{t+1} \sim \begin{cases} \hat{p}(s_t, a_t) & \text{if } t = 0, \\ \hat{p}(\hat{s}_t, a_t) & \text{if } t > 0. \end{cases} \tag{2.11}$$

While transition models are typically learned through a one-step prediction objective, where only the next state s' and reward r are prediction targets, some works also explore the use of multi-step prediction objectives to encourage better long-horizon predictions (Oh *et al.* 2015; Talvitie 2014).

It is a well-known problem that dynamics models can produce trajectories which diverge from the true environment over long horizons (Chua *et al.* 2018; Hafner, Lillicrap, Norouzi, *et al.* 2020). This is because even small errors in the model can

compound over time, leading to large errors in the predicted trajectory (Lambert *et al.* 2022; Talvitie 2014). This poses a challenge for planning-based RL algorithms, where longer horizon rollouts are useful for policy improvement (Holland *et al.* 2019). In order to mitigate this problem, model-based RL algorithms often limit the rollout horizon, though other approaches are sometimes used, such as leveraging an ensemble of models to reduce trajectory uncertainty (Chua *et al.* 2018), or executing rollouts in a learned latent space (Hafner, Lillicrap, Ba, *et al.* 2020).

In this thesis, we will be primarily focused on transition models that are stable for long-horizon rollouts. For a more in-depth discussion on long-horizon rollout models and several possible implementations, please refer to Chapter 4.

2.3 Laplacian Representations

The matter of how to represent state comprises one of the key decisions when designing RL agents. Whether using hand-designed features such as tile coding or state aggregation, leveraging inductive biases as in convolutional neural networks, or relying on the flexibility of a standard fully-connected neural network, each of these designs aims to tackle the problem of transforming raw observations into a representation that is most useful for the agent to solve the task at hand. One method of representing state that has garnered interest involves the usage of the *graph Laplacian matrix* L. The graph Laplacian is an important operator in the field of spectral graph theory and can be defined as follows: given an undirected graph $G = (V, E)$ where $V = \{v_1, ..., v_n\}$ is the set of vertices in the graph and $E = \{e_{ij}\}$ is the set of edges connecting v_i and v_j, with adjacency matrix $A \in \{0, 1\}^{n \times n}$,

$$A \doteq \begin{cases} \mathrm{A}_{ij} = 1 & \text{if } e_{ij} \in E, \\ \mathrm{A}_{ij} = 0 & \text{otherwise,} \end{cases}$$

and degree matrix $D \in \mathbb{Z}^{n \times n}$,

$$D \doteq \begin{cases} \mathrm{D}_{ij} = \text{degree}(v_i) & \text{if } i = j, \\ \mathrm{D}_{ij} = 0 & \text{otherwise,} \end{cases}$$

the Laplacian matrix is defined as simply

$$L \doteq D - A. \tag{2.12}$$

Alternatively, the Laplacian can be viewed as an operator on the space of functions on a graph $f : V \to \mathbb{R}$,

$$Lf(v_i) \doteq \sum_{v_j \sim v_i} (f(v_i) - f(v_j)), \tag{2.13}$$

where $v_i \sim v_j$ denotes adjacency between v_i and v_j. Similarly, for a weighted graph, where each edge e_{ij} has an associated weight w_{ij}, the Laplacian operator can be defined as:

$$Lf(v_i) \doteq \sum_{v_j \sim v_i} w_{ij}(f(v_i) - f(v_j)). \tag{2.14}$$

The Laplacian can be thought of as a diffusion operator that describes the flow of information through a graph, and the eigenvectors of the Laplacian capture different temporal properties of diffusion within the graph (Mahadevan and Maggioni 2007). Specifically, we can capture the largest timescale properties of the graph by taking the eigenvectors $[\boldsymbol{u}_1, ..., \boldsymbol{u}_d] \in \mathbb{R}^{|V|}$ associated with the d smallest eigenvalues of the Laplacian.

The eigenvectors of the Laplacian have found various applications in RL, and were originally introduced to RL as a means of constructing a useful representation for value function approximation (Mahadevan and Maggioni 2007). This representation uses the Laplacian eigenvectors as a set of basis functions, also known as proto-value functions (PVFs), and was designed to be learned in conjunction with policy iteration in order to speed up learning. In addition to value function approximation, PVFs have been used for other purposes in RL, such as option discovery (Machado *et al.* 2017) and reward shaping (Wu *et al.* 2018). In each case, PVFs can be used to construct $\boldsymbol{\psi} : \mathcal{S} \to \mathbb{R}^d$, a state representation mapping where each feature is the i'th eigenvector evaluated at state s:

$$\boldsymbol{\psi}(s) = [\boldsymbol{u}_1[s], ..., \boldsymbol{u}_d[s]]. \tag{2.15}$$

We will refer to this PVF mapping as the *Laplacian representation* for the remainder of this thesis. As the Laplacian representation encodes information about the geometry of an MDP, in Chapter 4 we will see how the space of Laplacian representations can be useful toward computing similarity between states, for the purpose of constructing an approximate transition model. Analytically computing the eigenvectors of the Laplacian becomes difficult when the state space is large (or continuous) or when a model of the environment is unknown, as is the case in model-free RL. Hence, in Chapter 4 we will also introduce one method of approximating the eigenvectors, following work in Wu *et al.* (2018), and discuss modifications we found useful for learning the approximation.

2.4 Dynamic Time Warping

Dynamic time warping (DTW) is a technique used to compare two time series sequences that may vary in speed or timing, and has been used for applications such as speech recognition, information retrieval, and time series analysis (Kruskal and Liberman 1983). It is particularly useful when comparing sequences that have similar shapes but are out of phase, or when there are missing or noisy data points. DTW works by finding the optimal alignment between two sequences by warping the time axis to minimize the distance between corresponding points. Given two time series sequences, a query sequence $X = (x_1, ..., x_{T_x})$, and reference sequence $Y = (y_1, ..., y_{T_y})$, a dissimilarity function f is defined on pairs of elements across sequences:

$$d(i, j) \doteq f(x_i, y_j) \geq 0, \tag{2.16}$$

where $i = 1...T_x$ is used to index elements in X, and $j = 1...T_y$ is used to index elements in Y. A common choice for f is Euclidean distance, but other functions are also possible. Warping functions ϕ_x and ϕ_y, align indices of the two sequences, i and j, to a common time axis k:

$$\begin{aligned} i &= \phi_x(k), & k &= 1, 2, ..., T \\ j &= \phi_y(k), & k &= 1, 2, ..., T. \end{aligned}$$

Given a choice of $\phi = (\phi_x, \phi_y)$, a global dissimilarity measure can be defined over the full sequences:

$$d_\phi(i, j) \doteq \frac{1}{M_\phi} \sum_{k=1}^{T} d(\phi_x(k), \phi_y(k)) m_\phi(k), \tag{2.17}$$

where $m_\phi(k) > 0$ is a per-step weighting coefficient, and M_ϕ is a normalization factor that allows comparisons across different paths.

A selection of constraints are typically imposed on ϕ to ensure good alignment. The monotonicity constraint ensures temporal order is maintained after warping:

$$\begin{aligned} \phi_x(k+1) &\geq \phi_x(k) \\ \phi_y(k+1) &\geq \phi_y(k). \end{aligned}$$

Local continuity constraints determine the allowable paths that the warping function can take to a given point (i, j) and can take many forms. One example is the constraint introduced in Sakoe and Chiba (1978):

$$\phi_x(k+1) - \phi_x(k) \leq 1$$
$$\phi_y(k+1) - \phi_y(k) \leq 1,$$

though many other choices are possible. Local continuity constraints are often best expressed through geometric diagrams, as in Rabiner and Juang (1993). Local slope constraints impose further limits on admissible paths, by putting a limit on the magnitude of change in a specific direction from $\phi(k)$ to $\phi(k+1)$. Finally, global path constraints aim to exclude certain regions of the (i, j) plane from the set of admissible paths, for example by limiting the temporal deviation between two matched elements.

Specific choices of constraints can be characterized by a *step pattern*, which determines the set of allowed transitions from a given matched pair $(\phi_x(k), \phi_y(k))$. The choice of step pattern can have a significant impact on the quality of the alignment, and the best choice of pattern is often problem-dependent.

Chapter 3

Hyperparameter Tuning

Despite the many successes of RL and deep learning, the selection of hyperparameters remains a dark art, with many practitioners relying on intuition or costly search processes to find good hyperparameters. This is especially true in the case of Deep RL (i.e. RL with deep neural network function approximators), where the hyperparameter space is often large and complex, and where the performance of an algorithm is typically highly sensitive to small changes in hyperparameters. In this section, we will discuss the role of hyperparameters in RL, the challenges of tuning them for real-world tasks, and give an overview of typical methods used for hyperparameter tuning.

3.1 Hyperparameters in RL

Reinforcement learning algorithms typically have a number of hyperparameters that need to be tuned in order to achieve good performance, and can be highly sensitive to small changes in or interactions between these hyperparameters (Andrychowicz *et al.* 2020; Engstrom *et al.* 2020; Henderson *et al.* 2018). Hyperparameter choices are also often environment-specific, meaning that an algorithm may have a different set of optimal hyperparameters across different environments (Henderson *et al.* 2018). Examples of hyperparameters typically tuned for RL methods include the discount factor γ^k, the optimizer learning rate α, the replay buffer size, and the neural network architecture (i.e. when using neural network function approximation). Studies like (Henderson *et al.* 2018) also consider various components of the algorithm as hyperparameters, such as value clipping, reward scaling, learning rate annealing, observation normalization methods, and network initialization schemes, finding that these choices have a greater impact on performance than the general training algorithm.

In general, hyperparameters are not learned during training, but rather are set

by a designer before training begins. This means that either the designer must have some prior knowledge about the problem and the algorithm in order to select appropriate hyperparameters, or they must employ some process to search over different hyperparameter settings (i.e. *hyperparameter tuning*). This process can be very time-consuming, especially when the algorithm is slow to train or there are many hyperparameters to search over. While progress has been made towards reducing the need for hyperparameter tuning through the use of adaptive methods (Jacobsen *et al.* 2019; Kingma and Ba 2014; M. White and A. White 2016; Zahavy *et al.* 2020), hyperparameter tuning remains a major bottleneck in the usage of RL algorithms.

3.2 Tuning for Real-World Tasks

The process of hyperparameter tuning is typically done in a simulation setting, where either the target environment itself is a simulation (e.g. Atari, Mujoco, etc.), or where a simulator is used to approximate the target environment (e.g. a physics-based simulator for a real-world system). In these cases, collecting samples from the environment can be cheap and fast, and despite being a bottleneck in the training process, hyperparameter tuning can usually be achieved without excessive overhead. However, this bottleneck can become extreme when applying RL to real-world tasks, where constructing a simulator is often infeasible (i.e. due to cost or complexity) and thus data collection can become extremely slow or expensive. For example, in Luo *et al.* (2022), only 300 samples per day can be collected from a commercial HVAC system where no simulator is available, and in Janjua *et al.* (2023), 86k samples can be collected per day from a water treatment plant. For reference, a single training run of a PPO agent in Mujoco lasts for 1M timesteps in Schulman *et al.* (2017), while a single run of a Rainbow agent in Atari consists of 200M timesteps in Hessel *et al.* (2017). This means that it could take weeks or months to perform even a small grid search.

In addition to time and computational cost, agents deployed in the real-world may have financial or safety constraints that limit the ability to run large sweeps over poorly performing hyperparameters. For example, in the case of a data center cooling system, it may not be acceptable to run a policy that causes the system to overheat, as this could damage the hardware. Or, in a pilot water treatment plant, it may be costly to dose large amounts of chemicals into the system. This suggests that hyperparameter tuning for real-world tasks is ideally achieved in an offline manner, where sweeps are performed either using historical data or in a simulation setting (i.e. when a high-fidelity simulator is available). Throughout this thesis, we will be primarily

interested in the setting where no simulator is available and instead where we have access to offline logs alongside limited access to collecting online samples - we will refer to this as the *NoSim* setting. Note that both the NoSim and simulator settings pose a new challenge: the distribution shift between the tuning and deployment environments. Much work has been done to study and address this issue, also known as transfer learning, or sim-to-real (Bellemare *et al.* 2020; Kaufmann *et al.* 2023; Matas *et al.* 2018; Rusu *et al.* 2018), but these works typically focus on transferring a policy, not the hyperparameters.

3.3 Tuning Approaches

There are many approaches to hyperparameter tuning in RL, each with its own trade-offs in terms of time, computational cost, and effectiveness. The most common approaches include grid search, random search, Bayesian optimization, and population-based algorithms. In grid search, the hyperparameter space is discretized and an agent is trained for each combination of hyperparameters, usually across several random seeds. This approach is simple and easy to parallelize, but can be very slow and inefficient, especially when the hyperparameter space is large. Random search is similar to grid search, but instead of uniformly discretizing the hyperparameter space, random samples are drawn from the space and models are trained with these hyperparameters. This approach is more efficient than grid search (Bergstra and Bengio 2012), but can still be inefficient given a large hyperparameter space. Bayesian optimization is an adaptive approach that uses information from past runs to guide the search process in a probabilistic manner, and has been shown to be more efficient than grid and random search in many cases (Snoek *et al.* 2012). Lastly, population-based training (PBT) (Jaderberg *et al.* 2017), aims to jointly optimize both an agent and its hyperparameters by using a population of concurrently running agents to select and propagate hyperparameters in an online fashion.

While these approaches are successfully used across many RL applications, they are primarily designed for the online setting, where an agent has the ability to sample large amounts of data from either the target environment or a simulator. However, in the NoSim setting, we have no such luxury - instead, the agent must rely on offline data to select hyperparameters. While there has been much work in offline RL to develop algorithms that can learn from offline data (Agarwal *et al.* 2020; Fujimoto *et al.* 2019; Kumar *et al.* 2020; Yu *et al.* 2020), these are primarily geared towards learning a fixed policy, as opposed to evaluating hyperparameter settings for an online agent. Highlighting this difference is the fact that offline RL algorithms themselves

have hyperparameters, which are either tuned using the real environment (Wu *et al.* 2019) or from the Q-values learned during offline training (Paine *et al.* 2020), with limited success. The offline RL methods mentioned above typically involve evaluation of a fixed policy, however we are interested in evaluating an online learning algorithm where the policy is in flux. There is one previous work that considers how to use offline data to evaluate an online agent (Mandel *et al.* 2016), but it is not effective for longer horizons as is typically required for hyperparameter tuning.

To tackle this problem of evaluating an online agent, (H. Wang, Sakhadeo, *et al.* 2022) introduce the idea of a *calibration model*, which is a transition model learned from offline data that can be used to evaluate hyperparameters. Given such a model, any of the previously covered hyperparameter tuning approaches can be used in a typical online fashion. The authors propose that an imperfect simulator might still be useful for identifying good hyperparameters, in contrast with typical model-based RL or sim-to-real methods, where an accurate model or simulator is typically required for policy transfer. The remainder of this thesis will be dedicated to exploring the idea of a calibration model along with several of its possible implementations.

Chapter 4

Calibration Models

The notion of a *calibration model* is one possible solution to the problem of hyperparameter tuning for real-world RL agents (H. Wang, Sakhadeo, *et al.* 2022). This model is essentially an environment simulator learned from an offline dataset, with potentially lower fidelity requirements than simulators used for policy transfer. In order to use a calibration model for hyperparameter tuning, we require the model to be stable under long rollouts, as evaluation rollouts are typically run for thousands, if not millions of timesteps. In this chapter, we will discuss perspectives on evaluating model quality, and cover several candidate transition models for producing stable trajectories over many timesteps. We discuss two popular choices of parametric models (feedforward and recurrent neural networks) before introducing a non-parametric method based on the k-nearest neighbors algorithm. Finally, we cover the difficulties of evaluating a calibration model, and propose several metrics that can be used for evaluation.

4.1 Preliminaries

Let us consider a setting similar to that of offline reinforcement learning, where there exists access to a static dataset of state transitions $\mathcal{D} = \{(s_i, a_i, r_i, s'_i)_i\}$, where i is the index of the transition, collected by a (potentially unknown) behavior policy π_β from the true environment. The calibration is a learned approximation to the transition function, $\hat{p}(s', r|s, a)$, that is able to simulate the true environment to an extent that is useful for selecting hyperparameters. More specifically, we would expect the model to produce realistic transitions over many timesteps, and to be able to produce a variety of transitions that are similar to those observed in the true environment. The exact meaning of the term realistic will vary depending on the environment and the task, but we will discuss several properties of trajectories that we believe are important for

evaluation. First, let us introduce a characterization of calibration models by their ability to generalize to unseen state-action pairs and explain the importance of this distinction.

4.2 Generalizing and Non-Generalizing Models

Standard transition models in model-based RL are parametric models that are trained to generalize to unseen state-action pairs by attempting to approximate the true transition function $p(s', r|s, a)$. This is useful for planning, as it allows the model to produce transitions for any state-action pair, rather than just those observed in the training data. However, this generalization capability can also lead to compounding prediction error when the model is rolled out for long horizons, as the model may produce transitions that are not realistic or valid. Note that this compounding prediction error is highly related to the difficulty in training models for multi-step prediction (Talvitie 2014; Venkatraman *et al.* 2015).

In contrast, we might consider a transition model that is designed to minimize compounding prediction error, for example by limiting transitions to those observed in the training data. Such a model might allow trajectories to remain stable over long horizons, as it would never transition to parts of the state space that were not seen in the dataset, and thus prevent large errors or collapse of predictions. However, this long-horizon stability comes at the cost of generalization, as the model would only be able to produce transitions for state-action pairs that are similar to those observed in the training data. We will refer to these models as *non-generalizing* models, and we will refer to the models that are able to generalize to unseen state-action pairs as *generalizing* models. Note that the exact threshold we use to define a long-horizon rollout will vary depending on the environment, but we will typically be interested in horizons of at least 50-100 timesteps (and often much longer), as this is the length of trajectories typically needed to evaluate a given hyperparameter setting.

In the following sections, we will discuss several implementations of both generalizing and non-generalizing models, and investigate their performance and stability under selected conditions.

4.3 Data Coverage

A learned model is inherently limited by the quality of the offline data that is used for training. A dataset can be characterized by its trajectory quality and state-action coverage (Schweighofer *et al.* 2021); for example, trajectory quality can be measured

by average dataset return and state-action coverage can be measured by number of unique state-action pairs. Another classification of datasets that is often used is based on the policy used for data collection. For example, a dataset can be generated by random policy, medium policy, expert policy, or some combination of policies (Fu *et al.* 2021). In our experiments in Chapter 5, we will primarily focus on the setting where the model has good coverage over the state-action space and some examples of near-optimal trajectories, using mixed policies to achieve such coverage. We explore the usage of large real-world datasets in Chapter 7, where we expect more state-action coverage to produce an improved model.

4.4 Neural Network Models

Neural network models are one popular choice for transition models in model-based RL, as they are able to learn complex functions from raw data, The flexibility of neural networks to learn such functions has led to breakthroughs in a variety of fields, including computer vision (Krizhevsky *et al.* 2012), natural language processing (Devlin *et al.* 2018), and reinforcement learning (Mnih, Kavukcuoglu, Silver, Rusu, *et al.* 2015; Silver *et al.* 2016). The art of training these networks has also advanced significantly, making them robust and reliable tools for a variety of tasks. Thus, it is no surprise that neural networks have been used as dynamics models in a number of model-based RL algorithms (Chua *et al.* 2018; Deisenroth and Rasmussen 2011; Hafner, Lillicrap, Norouzi, *et al.* 2020). When applied to transition models, neural networks are generally used to estimate the dynamics of the environment $p(s', r|s, a)$, which makes them *generalizing* models, as previously discussed. In this section, we will discuss two popular choices of neural network models for transition models: feedforward neural networks and recurrent neural networks.

4.4.1 Feedforward Neural Network

The most standard form of neural network is a multi-layer, fully-connected neural network, also known as a feedforward neural network (FNN, but we will also refer to this as NN). FNNs are parameterized by a set of weights and biases, and are able to learn complex functions from raw data. For such parametric models, we typically frame the learning objective as finding a set of parameters θ that minimizes the one-step prediction loss over transitions in the dataset:

$$\theta^* \doteq \arg\min_{\theta} \frac{1}{|\mathcal{D}|} \sum_{(s_t, a_t, s_{t+1}) \in \mathcal{D}} \|s_{t+1} - \hat{p}_{\text{FNN}}(s_{t+1}|s_t, a_t; \theta)\|_2^2, \tag{4.1}$$

where $\mathcal{D}$ is the dataset of transitions collected from the true environment and $\hat{p}_{\text{fnn}}$ is the parameterized model. Notice that $\hat{p}_{\text{FNN}}$ is only conditioned on the current state and action, and does not take into account any previous states or actions in the trajectory.

This optimization problem can be solved with the aid of optimizers such as SGD or Adam (Kingma and Ba 2014), along with other useful techniques such as hold-out validation and input normalization to improve generalization and speed up learning.

Examples of model-based RL algorithms that use FNNs for their transition models include model-ensemble trust-region policy optimization (ME-TRPO) (Kurutach *et al.* 2018), where an ensemble of deep FNN models are used to stabilize learning, and Nagabandi *et al.* 2018, where an FNN is used to learn the dynamics of the environment for a model-predictive control algorithm.

4.4.2 Recurrent Neural Network

If the transition dynamics of an environment are not Markovian, then a standard feedforward neural network may be insufficient to capture the true one-step dynamics. In other words, the next state $s\prime$ may depend on more than just the previous state and action (s_t, a_t). This characteristic is also often referred to as *partial observability*. In this case, it is common to use a recurrent neural network (RNN) to capture the temporal dependencies in the data. The RNN model can be defined as $\hat{p}_{\text{RNN}}(s_{t+1}|s_t, a_t, h_t; \theta)$ Here, $h_t \in \mathbb{R}^{n_h}$ is the hidden state of the RNN at time t that serves as memory for the previous states and actions, where n_h is the number of neurons in the hidden state. The hidden state is updated at each timestep according to:

$$h_{t+1} = g(\theta_h h_t + \theta_x x_t(s_t, a_t)), \tag{4.2}$$

where $g : \mathbb{R}^{n_h} \to \mathbb{R}^{n_h}$ is a non-linear activation function, $\theta_h \in \mathbb{R}^{n_h \times n_h}$ and $\theta_x \in \mathbb{R}^{n_h \times n_x}$ are the parameters of the RNN, $\theta = \theta_h \cup \theta_x$, and $x_t \in \mathbb{R}^{n_x}$ is the feature vector timestep t, which is typically a concatenation of the current state and action, i.e. $x_t = [s_t, a_t]$.

The learning objective is similar to that of the FNN, but the model is trained using backpropagation through time (BPTT) (Werbos 1990). Vanilla RNNs are rarely used due to their difficulty in learning long-term dependencies, and are often replaced with more sophisticated architectures such as long short-term memory (LSTM) (Hochreiter and Schmidhuber 1997) or gated recurrent units (GRU) (Cho *et al.* 2014). In this thesis, we will use the GRU architecture for our RNN experiments, which are similar in performance to LSTMs, but are typically faster to run and train.

RNNs are a popular choice for model-based RL algorithms in partially observable environments, and have been the focus of several recent works (Chiappa *et al.* 2017; Hafner, Lillicrap, Norouzi, *et al.* 2020; Kaiser *et al.* 2019). For example, the DreamerV2 algorithm (Hafner, Lillicrap, Norouzi, *et al.* 2020) uses an LSTM-based model to learn the transition dynamics of the environment, and is able to produce realistic rollouts over medium horizons ($H = 15$).

4.5 K-Nearest Neighbors Models

In this section, we will discuss a *non-generalizing* k-nearest neighbors (KNN) model that aims to avoid compounding prediction error by limiting model transitions to the space of real transitions observed from the true environment (H. Wang, Sakhadeo, *et al.* 2022). This model is non-parametric and relies on a notion of distance within a representation space to select transitions. We posit that while a KNN model may sacrifice generalization capabilities, its ability to remain stable over long horizons make it a good candidate for a use case such as hyperparameter selection.

4.5.1 Algorithm Overview

Like previous calibration models discussed, the KNN model aims to provide an approximation of $p(s', r|s, a)$ from which we can draw samples. That is, for a given state-action pair (s, a), the KNN model must produce a next state s' and reward r. In order to produce novel trajectories that differ from those observed in the offline dataset, we can consider a transition tuple $(s_i, a_i, r_i, s'_i) \in \mathcal{D}$ to be a *neighbor* of a query state-action pair (s, a) if the state-action pair is close to the query pair in some representation space. At each timestep in a rollout, the model will first find the k most similar transitions to the query pair in the dataset, using an efficient nearest neighbor search algorithm such as a k-d tree (Bentley 1975). Then, the model will stochastically sample from these k transitions, in proportion to their distance d from the query pair. The probability of sampling a transition with index i from the query pair (s, a) is given by:

$$Pr\{S_{t+1} = s'_i, R_t = r_i \mid S_t = s, A_t = a\} \doteq \frac{1}{Z(s,a)} \left(1 - \frac{d_i(s,a)}{\sum_{j=1}^{k} d_j(s,a)}\right), \tag{4.3}$$

where $d_i(s, a)$ is the distance between (s, a) and the i'th neighbor (s_i, a_i, r_i, s'_i), and $Z(s, a)$ is the normalization constant used to map values to the interval $[0, 1]$:

$$Z(s,a) \doteq \sum_{l=1}^{k} \left(1 - \frac{d_l(s,a)}{\sum_{j=1}^{k} d_j(s,a)}\right). \tag{4.4}$$

Hence, transitions that are closer to (s,a) in the representation space are more likely to be sampled, and transitions with larger distance are less likely to be sampled. Note that the softmax function over $-d_i(s,a)$ is another reasonable candidate sampling distribution:

$$Pr\{S_{t+1} = s'_i, R_t = r_i \mid S_t = s, A_t = a\} \doteq \frac{e^{\frac{-d_i(s,a)}{\tau}}}{\sum_{j=1}^{k} e^{\frac{-d_j(s,a)}{\tau}}}, \tag{4.5}$$

with temperature parameter τ to allow for tuning the degree of randomness in the sampling process. While this tunable softmax provides a more flexible sampling distribution, we found the simple inverse-proportional distribution from Equation 4.3 to be sufficient for our experiments.

Up to this point we have introduced the notion of a distance function $d(s,a)$ without specifying how this distance is computed. The simplest approach is to use the Euclidean distance between the state-action pairs within the raw state space, $d_i(s,a) = \|(s,a) - (s_i,a_i)\|_2^2$. However, this approach is not always suitable, as it does not take into account the structure of the underlying MDP and may not provide an accurate measure of similarity between state-action pairs. This can become problematic, for example in a 2D gridworld, where states that are close together in terms of (x,y) coordinates have a wall between them; in this case we would not want to output a small distance between the two states, despite a small Euclidean distance. Instead, we may want to learn a function $\psi(s,a)$ that maps state-action pairs to a representation space where the distance between pairs is more indicative of their similarity. In the next section we will discuss a learned mapping function and distance metric based on the graph Laplacian.

Before moving onto the learned distance metric however, our KNN algorithm must also consider how to handle the case where the query pair (s,a) is not close to any of the neighbors in the dataset. This might occur when there is poor dataset coverage over certain regions of the state-action space. While there are different ways we might handle this, we follow the approach taken in H. Wang, Sakhadeo, *et al.* (2022) and take a *pessimistic* view of the action. In practice, this means that if there are no neighbours within a certain distance threshold of the query pair, we return a minimum reward computed from the dataset and terminate the episode. This pessimistic approach helps to prevent the model from producing unrealistic transitions, while simultaneously deterring the agent from entering low-coverage areas of the state-action

space.

4.5.2 Laplacian Distance Metric

The Laplacian representation, as introduced in Chapter 2, provides a natural way to compute state similarity in an MDP, due to its ability to capture the geometric structure of the MDP. States that are close in Euclidean distance may be far apart in the MDP, and the Laplacian representation has been shown to offer an improved notion of distance in such cases (Wu *et al.* 2018). In this new representation space, state similarity can be expressed as the L^2 distance between state representations, $d_i(s) = \|\boldsymbol{\psi}(s) - \boldsymbol{\psi}(s_i)\|_2^2$.

While analytically computing the eigenvectors of the Laplacian is straightforward in small, discrete state spaces where a model of the environment is known, it becomes significantly more challenging when these criteria are not met. As a result, various works have focused on the problem of accurately approximating the Laplacian from samples (K. Wang, Zhou, Feng, *et al.* 2022; K. Wang, Zhou, Zhang, *et al.* 2021; Wu *et al.* 2018).

We follow the approach taken in Wu *et al.* (2018) and H. Wang, Sakhadeo, *et al.* (2022), which leverages spectral graph drawing to stochastically approximate the eigenfunctions of the Laplacian via sampling. The Laplacian is considered for a fixed behavior policy, such that the transition distribution forms a Markov reward process. For a dataset $\mathcal{D}$ of transitions, the graph drawing objective can be expressed as

$$\sum_{s_t \sim \mathcal{D}} \left\|\psi_\theta(s_t) - \psi_\theta(s_{t+1})\right\|_2^2 + \sum_{s_i, s_j \sim \mathcal{D}} \left((\psi_\theta(s_i)^T \psi_\theta(s_j))^2 - \left\|\psi_\theta(s_i)\right\|_2^2 - \left\|\psi_\theta(s_j)\right\|_2^2\right), \quad (4.6)$$

where $\psi_\theta : \mathcal{S} \to \mathbb{R}^d$ is the representation learned using a neural network function approximator, with representation dimension d, and parameters θ. Minimizing this objective function through, for example, stochastic gradient descent yields an approximation to the Laplacian eigenfunctions. Intuitively, we can view this objective as being comprised of an *attractive term* and a *repulsive term*. The first term is attractive insofar as it encourages ψ_θ to map states s_t and their successors s_{t+1} closely in the representation space - this roughly captures temporal distance within an MDP. Conversely, the second term encourages independently sampled state pairs from the dataset to have orthogonal representations.

Several hyperparameters are introduced to allow more flexibility in learning. The β hyperparameter controls the weighting of the repulsive term relative to the attrac-

tive term, ζ regularizes representations away from zero, and $\kappa \in [0, 1)$ extends the objective to multi-step transitions, yielding the updated expression:

$$\sum_{s_t \sim \mathcal{D}, u \sim P_\kappa} \left\| \psi_\theta(s_t) - \psi_\theta(s_{t+u}) \right\|_2^2 + \sum_{s_i, s_j \sim \mathcal{D}} \left(\beta (\psi_\theta(s_i)^T \psi_\theta(s_j))^2 - \zeta \left\| \psi_\theta(s_i) \right\|_2^2 - \zeta \left\| \psi_\theta(s_j) \right\|_2^2 \right). \tag{4.7}$$

By extending the objective to handle multi-step transitions, we allow the representation to directly learn longer temporal relationships between states. To accomplish this in practice, we store trajectories of length z in the replay buffer - when a state trajectory $[s_0, s_1, ..., s_z]$ is selected, we need to choose two states to use for our update. The first state is naturally s_0 and the second state is selected with sampling probability $P_\kappa = [\kappa, \kappa^2, ..., \kappa^z]$, such that κ determines how often nearer vs. further states are selected for updates. See Appendix A.1 for values of these hyperparameters that show good empirical performance.

In order to validate learned Laplacian representations, we use a separate metric called *dynamics awareness* (H. Wang, Miahi, *et al.* 2024). This metric is highly related to the approximate Laplacian objective function, but does not depend on relative weighting between attractive and repulsive loss nor the effect of multi-step rollouts. Instead, we measure the distance between a state and its successor state relative to the distance between that state and a randomly sampled state from the dataset:

$$L_{DA} \doteq \frac{\sum_{s_i \sim \mathcal{D}} \left\| \psi(s_i) - \psi(s_{j \sim U(1,N)}) \right\| - \sum_{s_i \sim \mathcal{D}} \left\| \psi(s_i) - \psi(s_{i+1}) \right\|}{\sum_{s_i \sim \mathcal{D}} \left\| \psi(s_i) - \psi(s_{j \sim U(1,N)}) \right\|}. \tag{4.8}$$

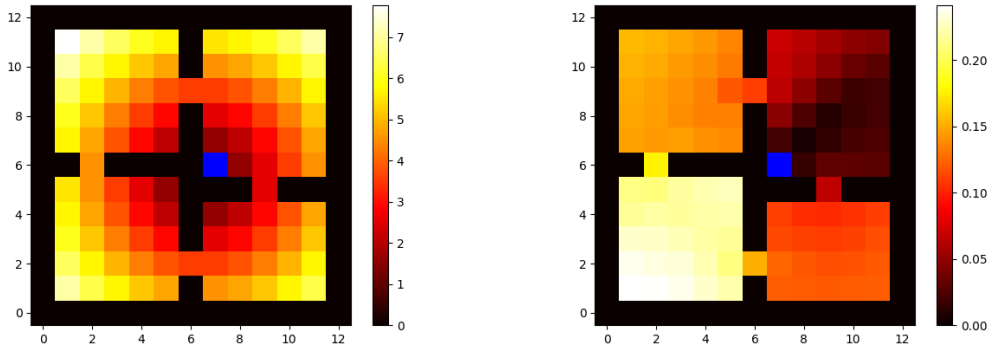
In practice, we find that the dynamics awareness score is not able to capture degeneration in the representation space, where either many or all states map to the same Laplacian representation. Clearly, this representation is undesirable as it does not provide any meaningful notion of distance between states - all states will have zero distance from each other in the degenerated space. To mitigate this, we introduce a secondary validation metric that we call *representation uniqueness*, which measures the proportion of unique state representations over the entire validation set:

$$L_{RU} \doteq \frac{| \Psi |}{N}, \tag{4.9}$$

where Ψ is the set of Laplacian representations of all states in the validation set, and N is the number of states in the validation set. A good uniqueness score should be close to 1.0, denoting that nearly all states have a unique representation. Thus, a good learned Laplacian representation can be selected by first filtering out representations

with a low uniqueness score (e.g. < 0.95) and then ranking representations in this filtered set using the dynamics awareness score.

An example of a Laplacian representation learned through this process in a four-room continuous gridworld environment is provided in Figure 4.1. Compared to raw Euclidean distance, the Laplacian distance is able to account for the obstacles in the environment.



(a) Euclidean distance

(b) Laplacian distance

Figure 4.1: Heatmap of distances calculated from the blue square to all other squares in a continuous state-space gridworld environment, using (a) L^2 distance in raw observation space and (b) L^2 distance in Laplacian representation space, where the Laplacian representation was approximated using the method described above. The gridworld was discretized into squares for the purpose of visualizing distances, and $d = 4$ was used for the Laplacian representation - further hyperparameters can be found in Table A.6 and training methodology in Section 5.2.4.

4.5.3 Discrete vs. Continuous Action Spaces

The KNN model can be used for both discrete and continuous action spaces, but the method of performing the nearest neighbor search will differ between the two. For discrete action spaces, the model will partition the dataset into subsets based on the action, and then perform a nearest neighbor search within the subset corresponding to the query action. For example, if the query action is $a_t = 1$, the model will perform a nearest neighbor search over the subset of dataset transitions where $a_i = 1$. For continuous action spaces, we are not able to partition the dataset in the same way. Instead, we will form a feature vector $x_t = [\psi(s_t), g(a_t)]$ by concatenating the state representation $\psi(s_t)$ with the action representation $g(a_t)$, and then perform a nearest neighbor search over the entire dataset using this feature vector. We opt to use an action representation $g(a_t)$ that simply normalizes a_t to be within the same range of

$\psi \in \Psi$ where Ψ is the set of all state representations over $\mathcal{D}$. Pseudo-code for the discrete and continuous action KNN next-state sampling algorithms can be found in Algorithms 1 and 2.

Note that more thought may be needed to handle continuous action spaces, as the naive approach of concatenating the state and action representations may generalize poorly. For example, the distance between two action representations may not be indicative of the true similarity between the actions. In this case, we may want to use a learned action representation $g(a_t)$ that maps the action to a space where the distance between representations is more indicative of action similarity. Further, it is not clear what the proper distance function is when jointly comparing states and actions. Currently, we take the union between the state and action representation spaces and compute L^2 distance within the joint space, such that the state and action features are weighed equally. While this equal weighting works to a reasonable extent in our experiments, it is not clear if this is an ideal approach.

Algorithm 1 Discrete-Action KNN Sampling

Input: s_t, a_t
  $(s'_i, r_i, d_i)_{i=1}^k \leftarrow \text{KDTreeSearch}(\psi(s_t), a_t, k)$
  if $\min_i d_i >$ threshold **then**
    return $\min_{r_t \in \mathcal{D}} r_t$, terminal ▷ Get min return from dataset
  end if
  $i \sim \text{SamplingDistr}(d_1, ..., d_k)$ ▷ E.g. distribution from equation 4.3
  return r_i, s'_i

Algorithm 2 Continuous-Action KNN Sampling

Input: s_t, a_t
  $x_t = [\psi(s_t), g(a_t)]$
  $(s'_i, r_i, d_i)_{i=1}^k \leftarrow \text{KDTreeSearch}(x_t, k)$
  if $\min_i d_i >$ threshold **then**
    return $\min_{r_t \in \mathcal{D}} r_t$, terminal ▷ Get min return from dataset
  end if
  $i \sim \text{SamplingDistr}(d_1, ..., d_k)$ ▷ E.g. distribution from equation 4.3
  return r_i, s'_i

4.6 Bootstrapped Ensembles

A bootstrapped ensemble of models is a technique that can be used to account for model uncertainty (also known as *epistemic uncertainty*) and improve confidence in

the model's predictions. It involves generating samples from a dataset with replacement, which can then be used to train a population, or *ensemble* of models on. While a single learned model may be prone to randomness in its training process, an ensemble of models can help to smooth out this randomness by averaging out their predictions and reducing the risk of learning a poor model (Dietterich 2000). Bootstrapped ensemble models have been used with success in PETS (Chua *et al.* 2018) and model-based policy optimization (MBPO) (Janner *et al.* 2019), where uncertainty estimates constructed from bootstrapped ensembles are used to plan and learn policies.

There are various ways to construct bootstrap samples and to leverage an ensemble of models. In our experiments, we will use *block bootstraps*, where the dataset is segmented into chunks and B bootstraps are constructed by holding out one of these chunks per bootstrap. A separate model is then trained on each bootstrap, and predictions across models are reconciled by taking the *worst rank* of a hyperparameter across models. For example, if we have three bootstraps and three hyperparameter settings, the bootstraps might produce rankings of [1, 2, 3], [2, 1, 3], [1, 2, 3], respectively, where the index in each list represents the hyperparameter setting, and the value represents the ranking of the setting according to the bootstrap. Using the worst rank approach, our final ranking in this case would be [2, 2, 3]. Note that while there are many ways that rankings can be reconciled across bootstraps, including various voting strategies, we opt to use worst rank because it is a way of using bootstraps to define a lower bound of performance for a hyperparameter setting.

4.7 Alternative Models

There are a variety of model architectures that have been used in previous model-based RL works, typically for the purpose of simulating rollouts for planning. Non-parametric, Gaussian process-based models were historically popular for learning robot dynamics (Deisenroth, Rasmussen, and Fox 2012; Ko *et al.* 2007; Nguyen-Tuong *et al.* 2009), and have also been leveraged to provide rollout uncertainty estimates to the agent during planning (Deisenroth and Rasmussen 2011). Linear-Gaussian dynamics models have also been used for guided policy search algorithms (Finn *et al.* 2016; Levine *et al.* 2016b), but these models are designed primarily to model local dynamics. Probabilistic dynamics model, which model distributions over predicted states, have been shown to capture stochastic dynamics and aleatoric uncertainty (Chua *et al.* 2018; Depeweg *et al.* 2018). State-space models, which make predictions in a latent space, have been used with success in the Dreamer algorithm (Hafner,

Lillicrap, Ba, *et al.* 2020), where the authors augment the model with recurrent and spatial inductive biases in order to predict pixel-based observations. In complex industrial systems, models that incorporate structured domain knowledge have been shown to significantly improve model accuracy (Zhan *et al.* 2022). Finally, one promising line of work uses a Lyapunov function constraint to encourage long-horizon rollout stability of the dynamics model (Manek and Kolter 2020). The non-generalizing KNN model that we spotlight in this work is an alternate perspective on building dynamics models, with a focus on remaining in the space of seen transitions, and a potential trade-off of short-term dynamics accuracy. We believe that this approach could be complementary to other models presented in this section when the goal is to achieve long-horizon rollout stability.

4.8 Model Evaluation

In model-based RL, learned transition models are often evaluated with respect to their utility in planning, by benchmarking performance on a downstream control task (Hafner, Lillicrap, Norouzi, *et al.* 2020; Kaiser *et al.* 2019; Talvitie 2014). Similarly, we are interested in understanding how a model performs with respect to its ability to rank and select hyperparameters of an RL algorithm. In addition to this measure, we are also interested in understanding the properties of model-generated trajectories, how to classify whether they are realistic or not, and how these properties relate to the model's ability to select hyperparameters. In this section, we will discuss the properties of model-generated trajectories that we are interested in, as well as some measures we can use to evaluate these properties.

4.8.1 Hyperparameter Selection

The most direct method of evaluating a calibration model is to test its performance on the task of selecting hyperparameters. One approach to this is to plot agent performance across different values of a single hyperparameter when run in the calibration model, and compare the curve to that when run in the true environment. For example, we can plot the total return of a DQN agent across different values of the optimizer learning rate, e.g. $\alpha \in \{1e{-}4, 1e{-}3, 1e{-}2, 1e{-}1\}$, when run in the calibration model, and compare this to the same curve when run in the true environment. Total return can be averaged across multiple runs of the agent, and confidence intervals can be used to express the variance in performance. If the relative ranking of each hyperparameter setting is similar in the calibration model and the true envi-

ronment, then we might conclude that the calibration model is effective at selecting hyperparameters. In the case of bootstrapped calibration models, we will use worst rank across bootstraps to generate the model ranking, as described in Section 4.6.

4.8.2 Multi-Step Prediction Error

One popular method for evaluating the quality of model-generated trajectories is to use the mean-squared error (MSE) between the model's predictions and the true environment's transitions over many timesteps. This can either be averaged across all timesteps in a trajectory, or *per* timestep across many trajectories. We will be using the latter method, where the MSE at each step can be defined as:

$$\text{MSE}_t \doteq \frac{1}{K} \sum_{k=1}^{K} \left\| s_t^{[k]} - \hat{s}_t^{[k]} \right\|_2^2, \tag{4.10}$$

where K is the total number of evaluation trajectories, $s_t^{[k]}$ is the true state at timestep t of trajectory k, and $\hat{s}_t^{[k]}$ is the model-predicted state at timestep t of trajectory k. Alternatively, we might want to express the MSE in the same units as the state variables, in which case we can use the root mean-squared error (RMSE):

$$\text{RMSE}_t \doteq \sqrt{\frac{1}{K} \sum_{k=1}^{K} \left\| s_t^{[k]} - \hat{s}_t^{[k]} \right\|_2^2}. \tag{4.11}$$

MSE has been used to showcase the tendency of models to exhibit compounding prediction error when rolled out for long horizons (Lambert *et al.* 2022) and is typically the metric used to evaluate model accuracy. However, a high MSE does not necessarily mean that a given trajectory is unrealistic, for example in chaotic systems where trajectories will diverge given slight differences, or for trajectories that are simply lagging by several timesteps. Works such as (Oh *et al.* 2015) recognize this and use a combination of quantitative analysis (MSE) and qualitative analysis (visual inspection) to evaluate the quality of model-generated trajectories. We will follow a similar approach in our experiments, using both quantitative and qualitative analysis to evaluate the quality of model-generated trajectories.

4.8.3 State-Space Distribution

Another approach to understanding how well a model simulates the true environment is to compare the distribution of states encountered in model-generated trajectories to those of the true environment. For a discrete state space, this can be done by counting

the number of times each state is visited in expectation over many trajectories, given some behavior policy, i.e.:

$$c(s) \doteq \mathbb{E}_\pi \left[\sum_{t=0}^{T} \mathbb{1}\{S_t = s\} \right]. \tag{4.12}$$

To handle continuous state spaces, one simple approach is to discretize each state feature into a set of B bins, and then count the number of times each bin is visited in expectation:

$$c(i,j) \doteq \mathbb{E}_\pi \left[\sum_{t=0}^{T} \mathbb{1}\{b_{\text{start}}^{[i,j]} \leq S_t^{[i]} < b_{\text{end}}^{[i,j]}\} \right], \tag{4.13}$$

where $i \in \{0, ..., |\mathcal{S}|\}$ is the index of the state feature, $j \in \{0, ..., B\}$ is the bin index, $S_t^{[i]}$ is the i'th feature of the random state, and $b_{\text{start}}^{[i,j]}$ and $b_{\text{end}}^{[i,j]}$ specify the range of the feature bin. The state-space distribution can then be visualized as a histogram, where each bin represents a value range of a state feature, and the height of the bin represents the number of times that feature range was visited in expectation over many trajectories. We will refer to this visualization as a *state distribution histogram*.

4.8.4 Invalid States and Transitions

Measuring the tendency of a calibration model to produce invalid states or transitions is another perspective on evaluating model quality. An invalid state is one that does not exist in the true environment's state space, while an invalid transition is a transition to a state that is not reachable from the previous state, according to the true dynamics. We can measure the frequency of invalid states or transitions in model-generated trajectories and compare this value across models. Counting invalid states is straightforward:

$$\mathbb{E}_\pi \left[\sum_{t=0}^{T} \mathbb{1}(S_t \notin \mathcal{S}) \right]. \tag{4.14}$$

However, counting invalid transitions is more difficult, for example in deterministic environments where any transition that does not follow the true dynamics exactly could be considered invalid. This could be mitigated by specifying a tolerance for the difference between the true and model-predicted transitions, however it's not immediately clear how to set this tolerance. For the purposes of our experiments, we will use environment-specific knowledge to determine what constitutes an invalid transition, e.g. for a gridworld environment, considering transitions through walls as invalid.

Chapter 5

Calibration Model Experiments

In this chapter, we will investigate the performance and stability of the NN, RNN, and KNN calibration models under selected conditions. We will use two simple environments, fixed-horizon acrobot and continuous gridworld, as our primary testbeds, alongside a combination of the metrics previously introduced to evaluate the quality of model-generated trajectories.

5.1 Fixed-Horizon Acrobot

For the first part of our experiments, our focus is twofold. First, we aim to understand the metrics introduced in Section 4.8, and the extent to which each maps to a useful notion of model quality. Second, we compare the performance of the NN, GRU, and KNN calibration models in a simple environment with a low-dimensional state space and simple dynamics. To this end, we will use a variant of the acrobot environment, with MSE, state-space distribution, and hyperparameter selection as our evaluation techniques.

5.1.1 Environment Description

The acrobot environment is a 2-link pendulum with a single actuator at the joint between the two links. The goal of the agent is to swing the end of the lower link up to a certain height. We use the Gym implementation of acrobot (Brockman *et al.* 2016) where the state space is 6-dimensional, consisting of the sine and cosine of the two joint angles and their angular velocities. The action space is a discrete and 1-dimensional, with three possible values representing the torque applied to the joint between the two links. The dynamics of the environment are governed by the equations of motion for a double pendulum, for which there is a detailed description in Sutton 1995.

For our experiments, we use a fixed-horizon variant of the acrobot environment, where the episode terminates after a fixed number of steps N_{traj}. Rather than using the default termination condition of the true environment, which terminates when the end of the lower link reaches a certain height, we instead allow the agent to continue swinging the pendulum until N_{traj} is reached. Reward is consequently changed to be -1 for every timestep that the pendulum is below the target height, and 0 for every timestep above the the target height. Because we'd like to understand compounding prediction error, using fixed-length trajectories allows us to have a constant number of samples to average across at each step in our per-step MSE analysis. A similar approach of using fixed-length trajectories was taken in Lambert *et al.* (2022). We primarily use $N_{\text{traj}} = 500$ for our experiments as empirically we found this gives sufficient time for the agent to learn, while also providing a large enough chunk of steps to investigate compounding prediction error.

5.1.2 Data Collection

As discussed in Section 4.3, a dataset with good coverage of the state-action space is generally a requirement in constructing a good model. To promote such coverage, we collect data using a learning agent that is trained to maximize the expected return in the true environment. The intuition here is that good coverage of acrobot is comprised of both sub-optimal behaviour for typical exploration, as well as near-optimal behaviour to reach certain parts of the state space (e.g. regions where the links need to reach a certain height). Hence a learning agent should provide broad coverage of the state-space, including the difficult to reach regions. This can alternatively be viewed as using a mixture of random, medium, and expert policies for data collection. We use a DQN agent with hyperparameters specified in Table A.1 and collect 200 trajectories with$N_{\text{traj}} = 500$ for a single training dataset. The DQN agent begins the training process with a randomly initialized policy, and improves its policy through the data collection process, reaching a near-optimal policy around 25% through the data collection process. Five datasets are collected using this process, only differing by random seed, to mitigate randomness associated with a single training run. These five datasets are subsequently used to train the NN, GRU, and KNN models.

5.1.3 Model Training

For the NN and GRU models, we split each dataset into a training set and a validation set, with 80% of the data used for training and 20% used for validation. The models are trained for 500 epochs to predict next state, reward, and termination,

$(S_{t+1}, r_t, term)$ with early stopping based on validation error. Because the transition models need to predict three separate targets, we train three separate networks of each model type, one for each prediction target. Network architectures can be tuned independently for each prediction target, e.g. for the acrobot GRU model, we use hidden layers of $[64, 64]$, $[32, 32]$, and $[16, 16]$ for next state, reward, and termination networks respectively. MSE loss is used for next state and reward prediction, while binary cross-entropy loss is used for termination. Hyperparameters for these models can be viewed in Tables A.4 and A.5. Notably, for the GRU model, we use a burn-in length of 5 timesteps, and a lookback sequence length of 10 timesteps for the one-step prediction. While the environment is fully observable and may not require a recurrent model, we include the GRU model in our experiments to provide a benchmark of its performance in a simple environment.

We also construct KNN models using both Euclidean and Laplacian distance metrics. For both the Euclidean and Laplacian KNN models, we use $k = 3$ nearest neighbors, which is the default used in H. Wang, Sakhadeo, *et al.* (2022). We also try eliminating stochasticity by setting $k = 1$ but find that this degrades model performance - see the next section for these results. There are no further hyperparameters required for the Euclidean KNN, however hyperparameters for the Laplacian representation training can be viewed in Table A.6.

We learn several unique models for each model class, where the models differ by their training dataset. Given four model classes and three datasets, we learn a total of twelve models. By having multiple models of each type, we hope to mitigate the effect of randomness in the training process, and to better understand the performance of each model type. Also note that hyperparameter tuning is performed for each model type using a basic grid search appraoach.

5.1.4 Evaluation

Root Mean Squared Error

We compute the per-step root mean squared error (RMSE) between model rollouts and true rollouts starting from the same initial state, and following the same fixed policy. The reported RMSE is averaged across nine runs of 200 trajectories each, where each run uses a unique combination of model and rollout policy (three distinct policies and three distinct models). The rollout policies were frozen from a DQN agent at different stages of training in the true environment are are different from the policies used during data dollection.

Following the approach taken in Lambert *et al.* (2022), we normalize state features

to $[0, 1]$ prior to computing RMSE, such that each feature contributes equally to the error. We include 95% confidence intervals for each model's RMSE in the plots. We also try measuring the RMSE in the learned Laplacian representation space, where we consider the L^2 distance between $\psi(s_t)$ and $\psi(\hat{s}_t)$ at each timestep.

As a baseline, we include results for a *static model* that always predicts the same next state, and a reward of -1. Effectively, this means that the model remains in the start state for the duration of each episode. This model provides a good reference point for how we expect a poor model to perform with respect to different evaluation metrics.

State-Space Distribution

Using the same rollout data as for the RMSE evaluation, we visualize the state-space distribution of each model by plotting histograms of the visitation counts for each feature bin during rollouts. Each feature is normalized to $[0, 1]$ and discretized to 20 bins. Histograms are overaged over nine runs of 200 trajectories. We expect that a good model will produce a state-space distribution that is similar to the true environment's state-space distribution.

Hyperparameter Selection

We plot hyper sensitivity curves over the Adam optimizer learning rate α for a DQN agent run in each of the calibration models, and compare this to the same curve when run in the true environment. We perform a grid search over the range $\alpha \in [1e{-}7, 1e{-}6, 1e{-}5, 1e{-}4, 1e{-}3, 1e{-}2, 1e{-}1]$, with 50 runs per hyper setting and a training budget of 20,000 steps. We then plot the average episode return of the DQN agent as our performance measure for each value of α. We expect that a good model will produce a hyper sensitivity curve that is similar to the true environment's hyper sensitivity curve - i.e. the ranking (as opposed to the raw performance value) of hyperparameters and the optimal hyperparameter value should be similar.

Altering Dataset Coverage

Finally, we briefly investigate the effect of reducing dataset coverage on the NN model's performance. We consider the standard dataset, which was collected using $N_{traj} = 500$ as described in Section 5.1.2 as our *high-coverage dataset*, and a secondary dataset collected with $N_{traj} = 50$ as our *low-coverage dataset*. Since the agent never escapes the region of the state-space close to the initial state, we expect the low-coverage dataset to contain a poor coverage of the state-space. We then compare performance of NN models trained on each dataset using RMSE, state-space

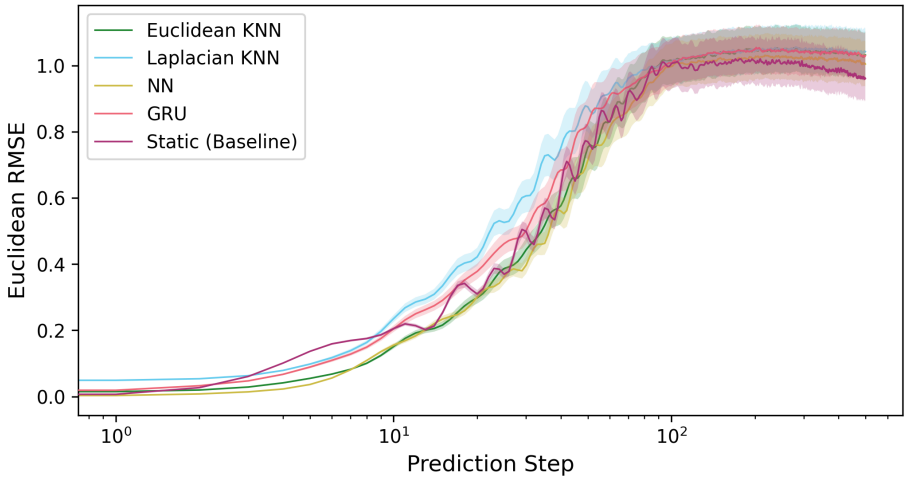


Figure 5.1: Per-step RMSE between model rollouts and true rollouts, averaged across nine runs of 200 trajectories each, where each run uses a unique combination of model and rollout policy. State features are normalized to $[0, 1]$ prior to computing RMSE and RMSE is computed in Euclidean space.

distribution, as well as a trajectory visualization.

5.1.5 Results

Root Mean Squared Error

We find that Euclidean RMSE has poor correlation with model quality. In Figure 5.1, we can see that the static model's RMSE curve roughly matches each of the other models, despite producing essentially useless trajectories. In fact, we see the static model's RMSE begin to decrease near the final prediction steps - this might be related to policy behaviour in later steps (e.g. a good policy in this environment might learn to swing the acrobot links around in circles). We also try computing the RMSE in the Laplacian representation space, and find that Laplacian RMSE is able to distinguish poor quality of the static model trajectories (see Figure 5.2). Note that several learned Laplacian representations were tried and yielded the same results. This suggests Laplacian RMSE may be a more meaningful metric for evaluating model quality than Euclidean RMSE, due to its ability to capture a notion of distance in the underlying MDP. Interestingly, the NN model performs best in the Laplacian space, even better than the Laplacian KNN which directly uses the Laplacian representation. One possible explanation for this is that learning one-step dynamics is a task closely related to learning the structure of the MDP, and so the NN model may implicitly learn to minimize the Laplacian error. However, more investigation would be needed to confirm this hypothesis.

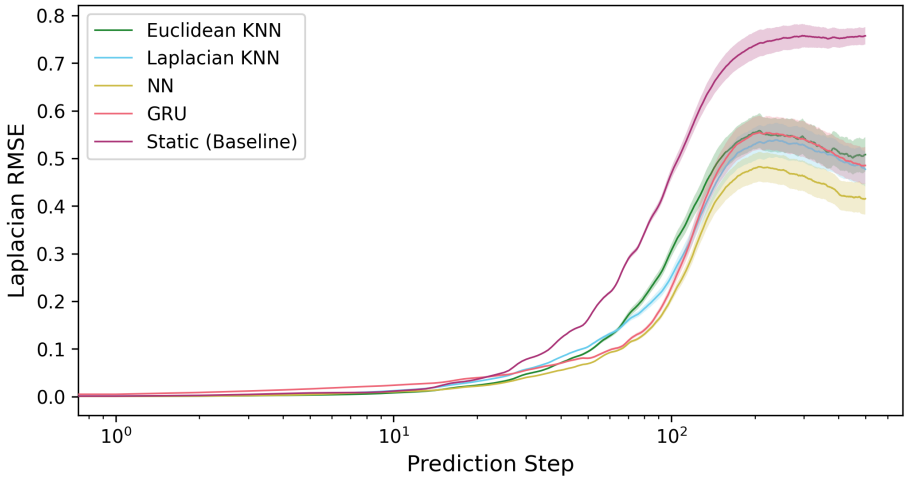


Figure 5.2: Same as Figure 5.1, but RMSE is computed in the Laplacian representation space.

State-Space Distribution

State-space distribution appears better able to capture the quality of model-generated trajectories than Euclidean RMSE. As we can see in 5.3, the static model produces feature histograms that are very different from the true environment's histograms, while the other models are all more closely matched. This suggests that state-space distribution may be a more meaningful metric for evaluating model quality than Euclidean RMSE. Also note that all (non-static) models perform well in the Fixed-Horizon acrobot environment with respect to state-space distribution, when trained on a dataset with good coverage. This is surprising, since we might have expected NN models to collapse due to compounding error and may suggest that realisitc multi-step prediction can in fact be achieved across a variety of model types in environments with simple dynamics.

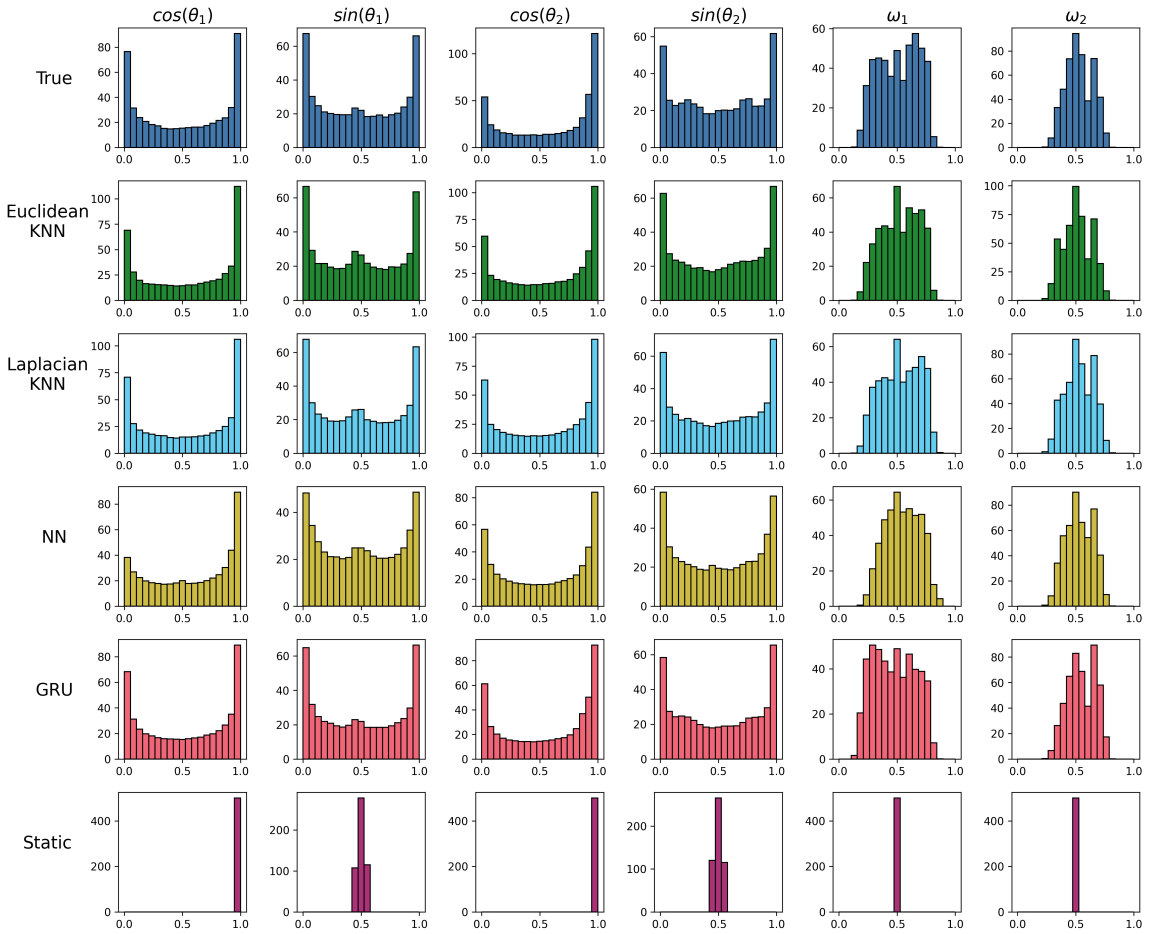


Figure 5.3: Histogram over binned features of the acrobot environment; each feature is normalized to [0, 1] and discretized to 20 bins. The histogram shows the number of timesteps spent in each feature bin, averaged across 200 rollouts from each model.

Hyperparameter Selection

All models perform relatively well from a hyperparameter selection perspective, as we can see in Figure 5.4 where the hyper sensitivity curves for each model are quite similar to the true environment's curve. Specifically, we can see that the learning rates $[1e{-}6, 1e{-}5, 1e{-}4]$ have overlapping confidence intervals for the true environment and can hence all be considered optimal (more runs would be needed to further distinguish between these). Each of the calibration models similarly selects optimal hyperparameters in this set, with most models typically selecting all three. Similarly, the rankings for the remaining hyperparameters are roughly consistent with the true environment ranking as well. This corroborates the findings from the state-space distribution histograms, and suggests that the good model accuracy determined by the state-space distribution perspective does in fact correlate with the type of model accuracy that is useful for selecting hyperparameters.

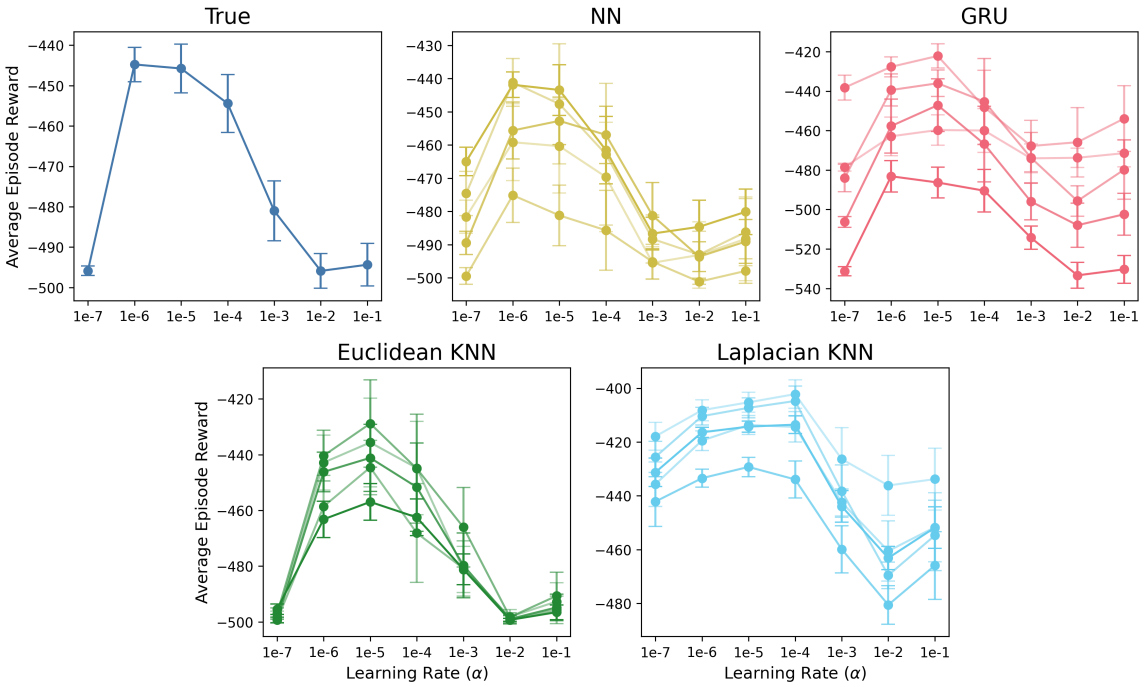


Figure 5.4: Hyper sensitivity curves over the Adam optimizer learning rate α for a DQN agent across the true environment and calibration models in fixed-horizon acrobot. Curves for five different models of each type are shown, each trained on a different dataset collected with a unique random seed. Each hyper setting is averaged across 20 random seeds, with a training budget of 20,000 steps per run. Transparency is used to visually distinguish between lines and does not encode any additional information.

Dataset Coverage

As expected, dataset coverage has a significant impact on the NN model's performance. Qualitatively, we can see that the low-coverage NN model produces trajectories that are quite different from the true environment's trajectories, as shown in Figure 5.5, which visualizes acrobot trajectories by stacking frames of the acrobot arm throughout a given rollout. This is similarly reflected in the state-space distribution histograms in Figure 5.6, where the low-coverage NN model produces a state-space distribution that is quite different from the true environment's distribution. Notably, the RMSE metric is *unable* to capture the degraded model quality of the low-coverage NN model, and actually appears to have slightly better performance according to Figure 5.7.

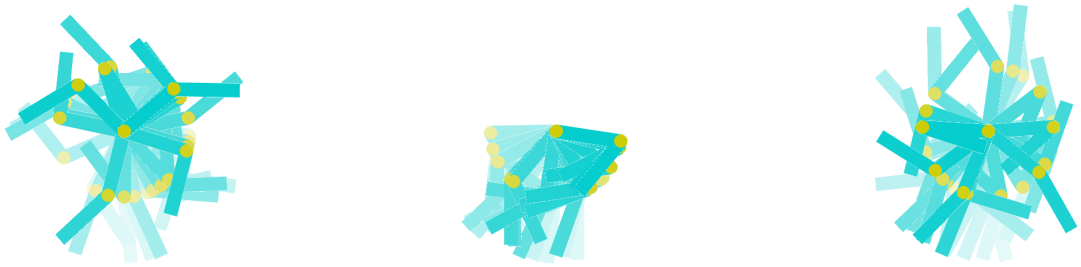


Figure 5.5: Visualizations of Fixed-Horizon acrobot rollouts from the true environment (left), a low-coverage NN model (middle), and a high-coverage NN model (right). Frames are subsampled every 15 steps from 500 step rollouts, with earlier frames shaded lighter and later frames shaded darker. Each rollout is initialized with the same start state and uses the same policy for selecting actions.

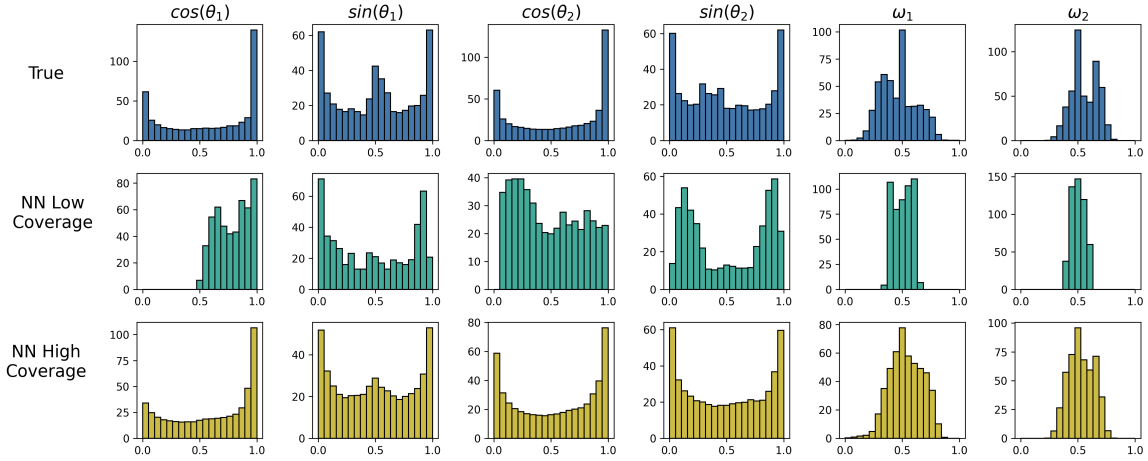


Figure 5.6: Feature histograms for NN models with varying dataset coverage.

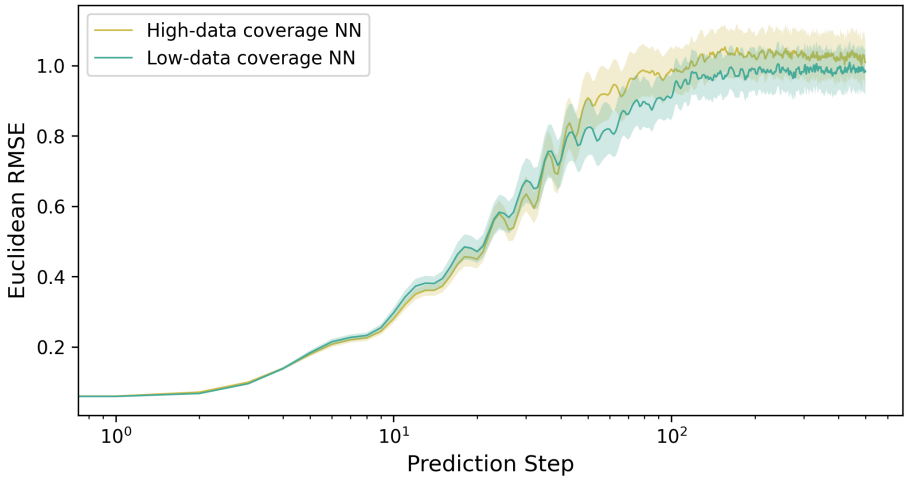


Figure 5.7: RMSE plot for NN models with varying dataset coverage.

KNN Model Sensitivity to k

The KNN model is quite sensitive to the number of nearest neighbors k used to sample the next state, as we can see in Figure 5.8. Some transition stochasticity (i.e. $k > 1$) appears necessary in order to stabilize the KNN rollouts.

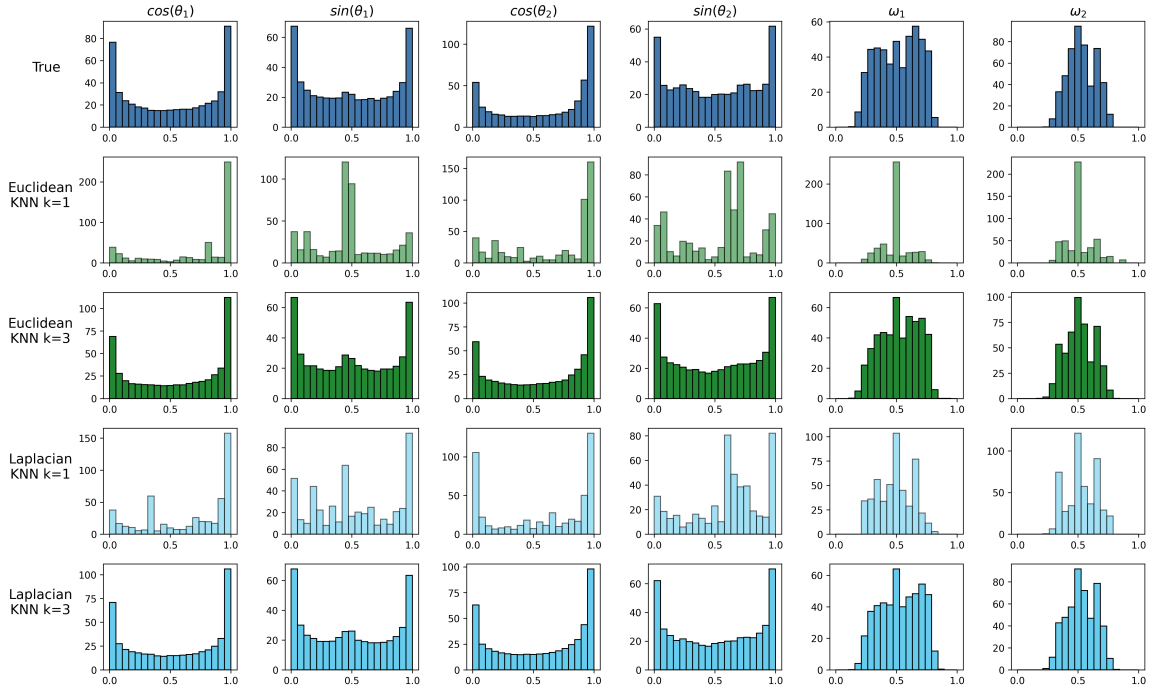


Figure 5.8: Histograms over binned features of the acrobot environment. Here we can see the effect of varying the number of nearest neighbors k used to predict the next state. Some transition stochasticity (i.e. $k > 1$) appears necessary in order to stabilize the KNN rollouts. The histograms are averaged across three different calibration models, each learned on a different dataset of 200 trajectories. The same rollout policy is used for all rollouts, which is different from the policy used to collect the training set.

5.2 Continuous Gridworld

For our next set of experiments, we would like to investigate the setting where we expect calibration models to produce unrealistic trajectories in an easily identifiable way. To achieve this, we will use a continuous gridworld environment with obstacles, where we might expect e.g. generalizing models such as the NN and GRU to predict next states where the agent transitions *into* or *through* obstacles. We aim to understand to what extent such models might be considered "unrealistic" and whether this has any bearing on their ability to select hyperparameters.

5.2.1 Environment Description

The continuous gridworld environment is a $w \times h$ 2D gridworld with continuous state and action spaces. The state space is 2-dimensional, with features representing the x and y coordinates of the agent, i.e. $s = (x, y)$ where $x \in [0, w]$ and $y \in [0, h]$. Similarly, the action space is 2-dimensional, where each dimension corresponds to a change in the agent's x and y coordinates. Given that the environment's dynamics are deterministic, they can be described by the equation $s_{t+1} = s_t + a_t$. While actions could be un-bounded, we limit them to the range $[-1, 1]$ for our experiments.

We primarily focus on a four-room gridworld of gridsize $w = h = 11$, with a goal state in the top-right room. Episodes terminate either after the agent reaches the goal state or after a fixed timeout. The reward is 0 for reaching the goal state and -1 for all other timesteps. Actions that would result in the agent moving into an obstacle or beyond the map boundary result in the agent staying at the map or obstacle boundary. The environment is deterministic, with the agent's action vector being clipped to $[-1, 1]$ at each timestep. The environment is visualized in Figure 5.9 with a near-optimal policy rollout.

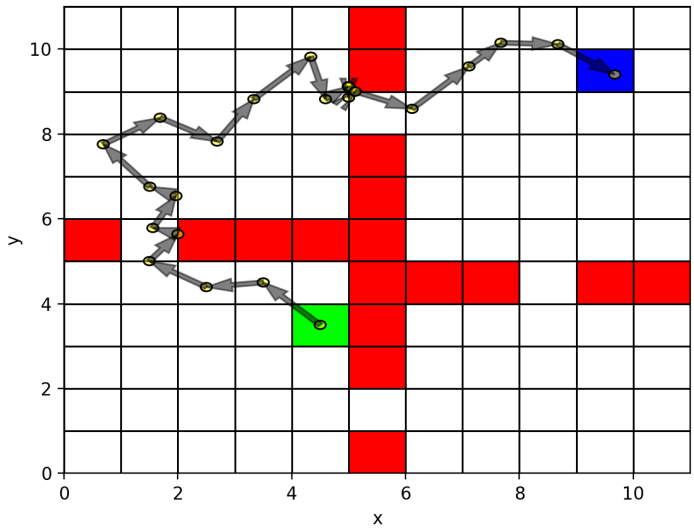


Figure 5.9: Four-room continuous gridworld environment with a near-optimal policy rollout. The green square represents the start state, the blue square is the goal state, red squares are obstacles, and arrows represent the agent's action vector at each timestep.

5.2.2 Agent Description

In order to handle the environment's continuous action space, we opt to use a Soft Actor-Critic (SAC) agent (Haarnoja *et al.* 2018) with a fixed entropy temperature α during training. Observations are normalized to $[-1, 1]$ prior to training, and the agent is trained according to the hyperparameters specified in Table A.2. The agent may select actions outside of the allowed range $[-1, 1]$, so we rely on either the true environment or calibration model to clip the actions appropriately. We use this SAC agent for both data collection and evaluation rollouts.

5.2.3 Data Collection

A good dataset for the continuous gridworld environment should include both sufficient coverage over the state-space as well as rollouts where the agent reaches the goal state, so that the calibration models can learn to predict the termination signal. To achieve this, we construct a dataset that is 95% random policy data and 5% near-optimal policy data. Specifically, we collect 19k transitions from a random policy agent and 1k transitions from a near-optimal policy agent. The near-optimal policy was learnt via SAC agent with hyperparameters specified in Table A.2 for 30k steps. Episode timeouts are set to 200 during data collection, which is also used in evaluation experiments. Three datasets are collected using this process, only differing by random seed, to mitigate randomness associated with a single training run. These three datasets are subsequently used to train the NN, GRU, and KNN models.

While actions are clipped to $[-1, 1]$ by the environment, we collect the pre-clipped actions in our dataset. This is because we want to see how the models handle invalid actions, and whether they are able to implicitly simulate the true environment's action clipping.

5.2.4 Model Training

Model training is similar to the acrobot environment as described in Section 5.1.2, but with different datasets and hyperparameters for the models. One other significant difference is in the algorithm used for the KNN models - for the discrete-action acrobot environment, we used a discrete-action KNN model, while for the continuous-action gridworld environment, we use a continuous-action KNN model as described in 2. As previously noted, the continuous-action KNN algorithm may be sub-optimal in its treatment of distance within the action space, as well as how it balances the importance of state and action distance, but we leave this as a topic for future work.

We use the same hyperparameters for the Euclidean KNN model as we did for the Laplacian KNN model, as we found that the Laplacian KNN model did not perform well in the continuous gridworld environment. Note that since the continuous gridworld is fully observable, for simplicity we opt to exclude the GRU model from this set of experiments. While the acrobot models were trained using unnormlized state features, we normalize state features to $[-1, 1]$ for training continuous gridworld models, including when learning a Laplacian representation. Hyperparameter tuning is performed for each model type using a basic grid search approach to arrive at the hyperparameters specified in the tables above.

5.2.5 Evaluation

Invalid transitions

We measure the number of invalid transitions in model-generated rollouts, where three types of invalid transitions are considered: transitions *into* an obstacle, transitions *through* an obstacle, and transitions with an *invalid magnitude*, i.e. where $|\Delta x| > 1$ or $|\Delta y| > 1$, given $\Delta x = x_{t+1} - x_t$ and $\Delta y = y_{t+1} - y_t$. To measure an expected number of invalid transitions, we perform nine runs of 5000 steps for each model, where each run is done using a unique combination of model and random seed (i.e. three models trained on different datasets, and three random seeds per model). We use a SAC agent that learns from scratch with hyperparameters specified in Table A.2 for rollouts. We then compute the number of transitions within each category as a percentage of the total number of transitions in each rollout and average this across all rollouts in the nine runs, also providing 95% confidence intervals.

Hyperparameter Selection

We follow the same procedure described for the acrobot environment in Section 5.1.4 for evaluating hyperparameter selection. Specifically, we plot hyper sensitivity curves over the SAC entropy temperature α for a SAC agent run in calibration models vs. the true environment. We perform a grid search over the range $\alpha \in [0.05, 0.2, 0.35, 0.5, 0.65, 0.8, 0.95]$, with 50 runs per hyper setting and a training budget of 30,000 steps.

Rollout Visualizations

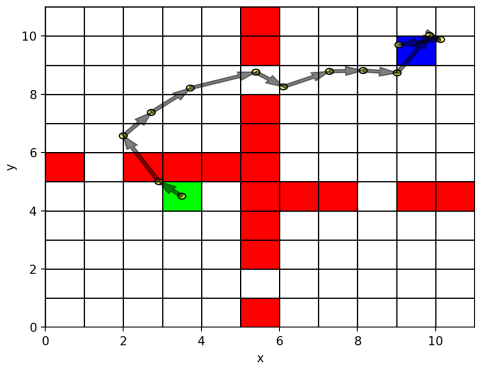
Lastly, we consider a qualitative approach to understanding rollout quality by looking at visualizations of rollouts from each model. We expect these visualizations to correlate with our measures of invalid transitions, and to provide insight into the nature

of each model's inaccuracies. We attempt to take rollout visualizations that are representative of a typical rollout from each model, while also showcasing inaccuracies specific to that model.

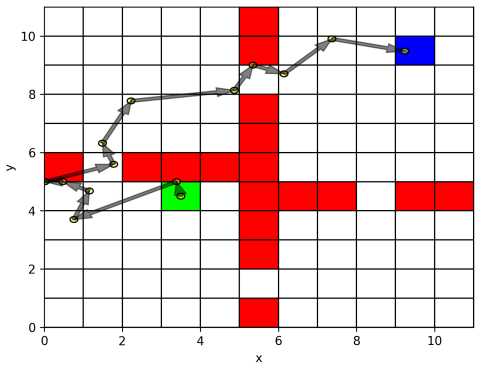
5.2.6 Results

Model Inaccuracies

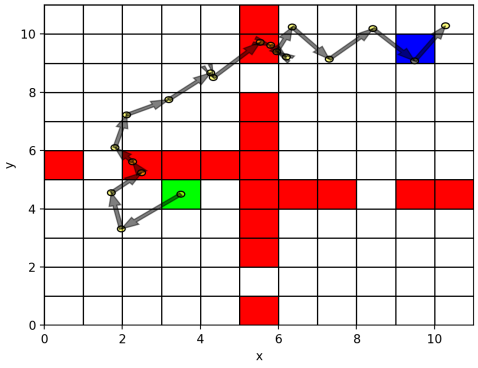
The rollout visualizations in Figure 5.10 and the invalid transition plots in Figure 5.11 help to paint a picture of the nature of the inaccuracies of each model. The NN model produces trajectories where there are seldom large jumps in the state space, but does produce invalid states (i.e. transitions *into obstacle*). In contrast, the KNN models disallow such invalid states by nature of only predicting real states from the training dataset, but do appear more prone to errors in one-step dynamics (i.e. transitions with an *invalid magnitude*). The Laplacian KNN is especially prone to these large jump transitions, which is perhaps not surprising, as the next predicted state is no longer selected based on distance in the original Euclidean space. The Laplacian KNN also reduces the number of invalid transitions *through obstacles* compared to the Euclidean KNN, due to its usage of the Laplacian distance metric, which is able to account for obstacles. Note that Figure 5.11 shows results for the setting where the agent's actions are not clipped prior to being passed to the calibration model. We also try clipping the agent's actions prior to being passed to the calibration model, and find that this decreases the number of invalid magnitude transitions for the NN model, but *increases* the number of invalid magnitude transitions for the KNN models, as shown in Figure 5.12.



(a) Euclidean KNN



(b) Laplacian KNN



(c) NN

Figure 5.10: Example rollouts of near-optimal policies in different calibration models of four-room continuous gridworld. Euclidean and Laplacian KNN rollouts tend to include more large magnitude jumps, while NN rollouts tend to include many invalid transitions into obstacles.

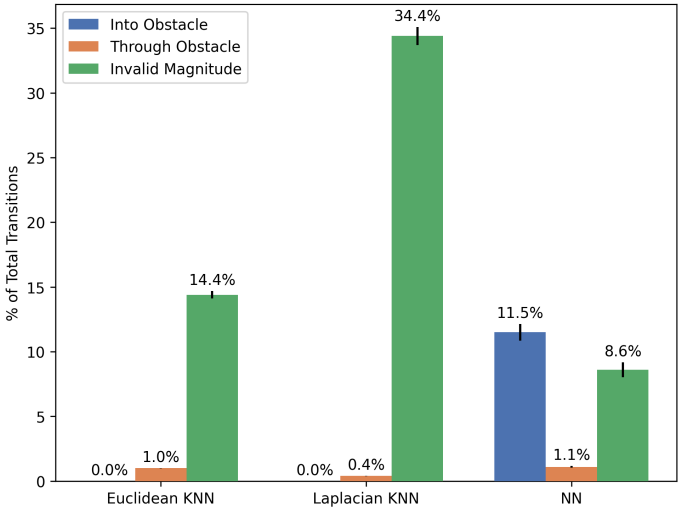


Figure 5.11: Count of invalid transitions as a percentage of total number of transitions across each model class.

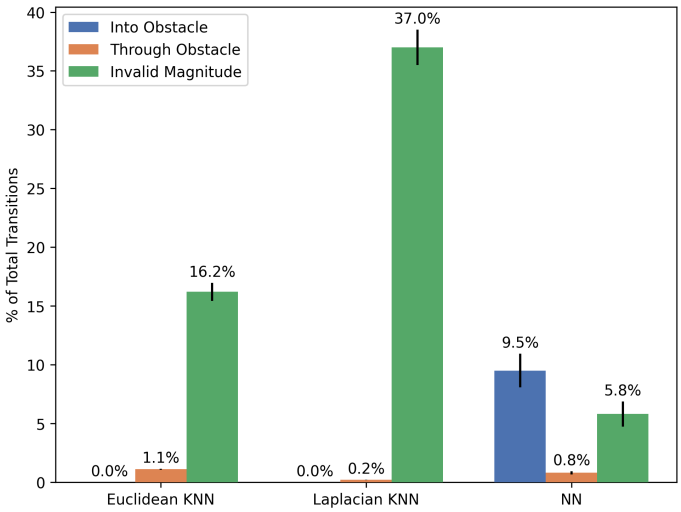


Figure 5.12: The same as Figure 5.11, except in this case agent actions are clipped prior to being passed to the calibration model.

Hyperparameter Selection

While the NN model is susceptible to producing invalid states, it appears that this does not have a significant negative impact on its ability to match the true environment's hyper sensitivity curve for the entropy temperature parameter α - as we can see in Figure 5.13, where the NN curve closely mirrors the true one. On the other hand, the invalid magnitude flavour of model inaccuracy presented by the KNN models appears to have a negative impact on the model's ability to select hyperparameters. Figure 5.13 highlights this, where both the Euclidean and Laplacian KNN models rank $\alpha = 0.2$ as a candidate for the best hyperparameter setting, while the true environment's curve suggests either $\alpha = 0.35$ or $\alpha = 0.5$ as the top choice. More investigation is needed to determine the precise relationship between model inaccuracies and hyperparameter selection, but we hypothesize that either the larger jumps allowed by the KNN models or the constraining of the MDP may provide an easier environment for the agent to exploit, thus allowing for better performance with less need for exploration (i.e. less policy entropy).

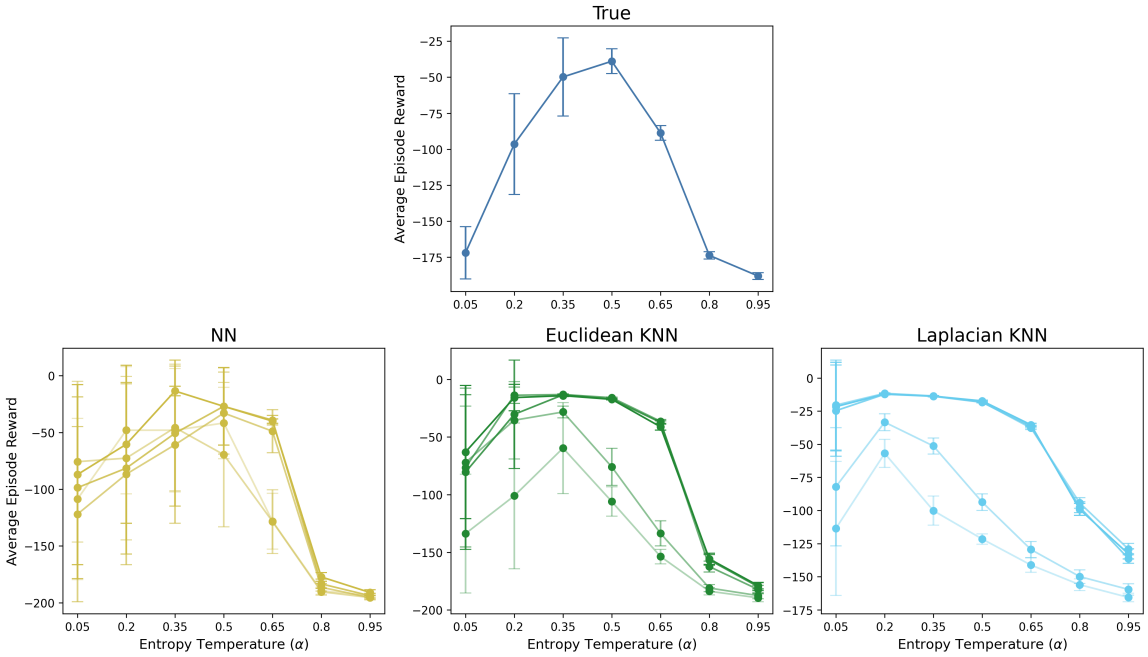


Figure 5.13: Hyper sensitivity curves over the SAC entropy temperature α for a SAC agent across continuous gridworld calibration models and true environment. Curves for five different models of each type are shown, each trained on a different dataset collected with a unique random seed. Each hyper setting is averaged across 20 random seeds, with a training budget of 30,000 steps per run. Transparency is used to visually distinguish between lines and does not encode any additional information.

5.3 Discussion

The first significant finding of this work is that MSE, and the multi-step compounding error typically associated with it, may not have much discriminatory power in distinguishing models of good and bad quality. In chaotic systems especially, like acrobot, a divergence from the true environment's trajectory does not necessarily imply that the model is bad. Instead, we find that other metrics such as state-space distribution may be better able to capture characteristics that we would consider indicative of a good model.

Further, model quality or accuracy can be considered in a variety of ways, including but not limited to the perspectives taken in this work. Namely, we can consider the state-space distribution produced over rollouts, the types and frequencies of invalid transitions produced, or the performance of the model on a downstream task, such as hyperparameter selection. Other perspectives that were not explored in this work but may be useful are measuring the error in the model's one-step dynamics, or assigning a likelihood that a model rollout was produced by the true environment.

Certain types of model inaccuracy may be more detrimental than others in the context of hyperparameter selection. While more evidence is required to suggest a causal relationship, we find the invalid magnitude transitions produced by the KNN models to correlate with worse hyperparameter selection performance, whereas the invalid states produced by the NN models do not. This suggests that the nature of model inaccuracies are important to consider when evaluating the quality of a model, and the significance of an inaccuracy is likely dependent on the downstream task at hand.

While we initially expected that generalizing models like the NN and GRU might be more prone to compounding prediction error, we found that these models performed well over long rollouts in the acrobot and gridworld environments. One interpretation of this result is that for environments with simple dynamics, a range of both generalizing and non-generalizing models can produce realistic multi-step predictions. This is an interesting finding, as it suggests that the choice of model may not be as important as we might have initially thought, at least in the context of simple environments.

Lastly, we recommend that more work be done to develop the continuous-action KNN algorithm. For discrete action spaces, the KNN algorithm addresses the issue of the raw Euclidean state space not matching the structure in the underlying MDP. However, for continuous action spaces, the algorithm also needs to account for distance in the action space, while further balancing the weighting between state and action distances when computing distance. A more sophisticated approach, e.g.

learning a distance metric in the action space and a weighting between state and action features, could be of benefit here.

In this chapter, we focused on understanding the performance of several calibration model implementations in the context of simple, simulated environments. In the next chapter, we will apply the calibration models to a real-world environment, and investigate the feasibility of the approach in a setting with more complex dynamics and high-dimensionality.

Chapter 6

Calibration Models for Water Treatment

For the final part of this thesis, we investigate the application of calibration models to a real-world water treatment plant. The goals of this study are to demonstrate the effectiveness of calibration models in a real-world setting, and to investigate the potential for using calibration models to select hyperparameters for a prediction agent. We will use the same calibration models as in the previous chapter, and focus on prediction of individual sensors in the water treatment plant.

6.1 Background

First, we provide background on the water treatment plant (WTP) dataset, the prediction task, and the use of dynamic time warping (DTW) as a metric for evaluating calibration models. We discuss the prediction task in detail using the general value function (GVF) framework, the computation of returns, and the evaluation metrics used in this study.

6.1.1 Water Treatment

In collaboration with the Drayton Valley water treatment plant located in Alberta, Canada, we collect real-time data from a range of sensors that are used to monitor the water treatment process. These sensors are not located on the full-scale plant, but rather on a miniature test plant that is fed the same incoming water, and is meant to emulate the filtration process used by the full plant. The plant uses a membrane filtration system to treat water by removing particulate matter such as sediment and microorganisms to a degree that satisfies regulatory and safety standards. Prior to passing through the filter membrane, the water is pre-treated with

chemicals to facilitate clumping of the particulate matter, which is then filtered by the membrane. There are a variety of control tasks that can be formulated within this system, some of which include controlling the chemical dosing rates, the membrane backwash frequency, and the water flow rates. For the purposes of this study, we focus on *prediction* of individual sensors, as this is a previously studied problem within the context of the Drayton Valley plant (Janjua *et al.* 2023) and offers a stable testing ground for our experiments.

The plant is equipped with a variety of sensors that monitor properties of the water chemistry and mechanical components of the plant at various stages in the water treatment pipeline. In total there are 480 sensors that monitor properties such as temperature, pH, turbidity, flow rate, and pressure, comprising a high-dimensional multivariate time series. Given the seasonal nature of incoming water quality, the changing conditions of the physical sensors (e.g. dirt or moisture building on a sensor over time), and the existence of sudden changes in plant operation (e.g. maintenance or repair), the sensor data is highly non-stationary, noisy, and complex. With this in mind, the algorithms we consider should be robust to a high degree of partial observability and noise. A visualization of several of these sensors is provided in 6.1.

The plant also does not have a simulator available, and thus deploying a reinforcement learning agent in the plant runs into the issues discussed in previous chapters for the NoSim setting, most notably the problem of hyperparameter tuning. Given the high-dimensionality, non-stationarity, and potentially complex dynamics of the sensor data, learning a calibration model in this environment is a challenging task and will provide a good litmus test for the effectiveness of the different types of calibration models. More details about the dataset, including pre-processing and augmentation techniques, will be discussed in 6.2.1.

6.1.2 Prediction

Prediction forms a critical element of any decision-making method, whether it be in value function estimation for action selection as in Q-learning, in constructing accurate models of the world as in model-based RL, or in directly anticipating future state variables as in nexting (Modayil *et al.* 2012). In the context of water treatment, Janjua *et al.* (2023) explore the utility and feasibility of predicting sensor values many steps into the future. While the focus of our experiments is on building and evaluating calibration models, the prediction tasks proposed in Janjua *et al.* (2023) provide a useful test-bed for such models.

To leverage RL methods for the purpose of sensor prediction, we use the GVF

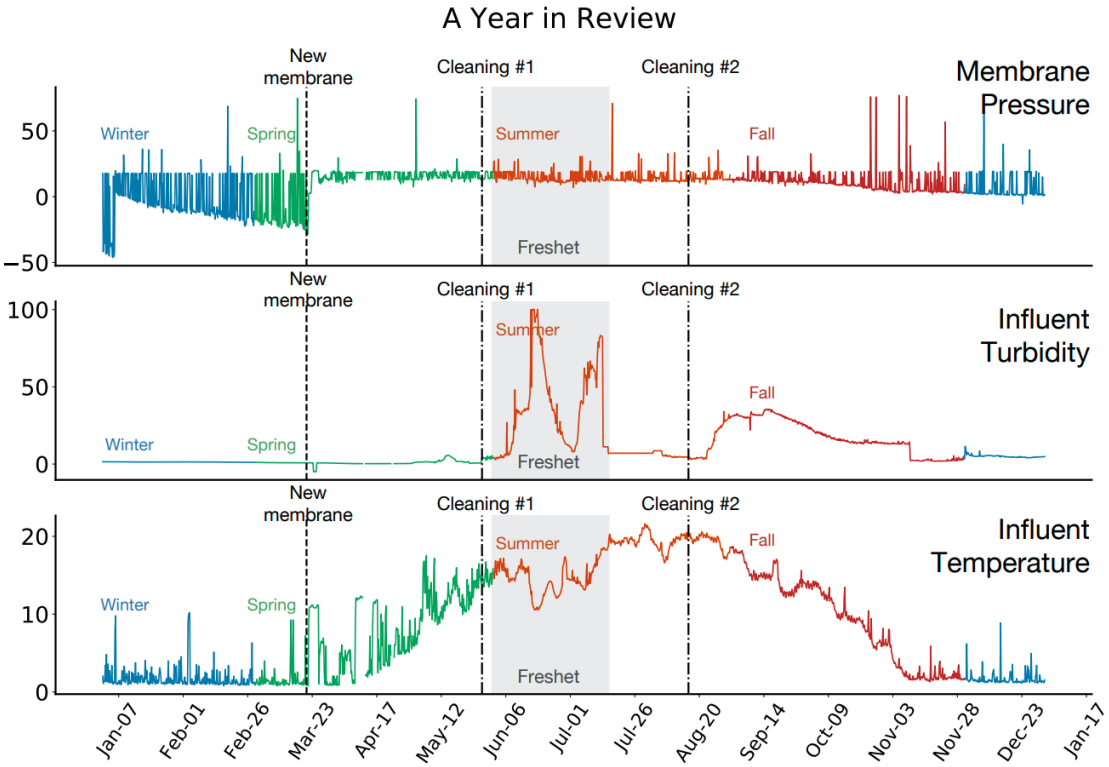


Figure 6.1: Plot taken from (Janjua *et al.* 2023) showing a subset of sensors from the water treatment plant over the course of a year. Different colours are used to reflect the change in seasons. This selection of sensors illustrates the effect of seasonality and plant maintenance procedures on the sensor readings.

framework, as described in Section 2.1.5. In the water treatment setting, cumulants are formulated as raw sensor readings o_t^i from the full observation vector $\boldsymbol{o}_t \in \mathcal{R}^d$, where d is the dimensionality of the observation vector, $i \in [1, d]$, and t is the timestep in a given trajectory.

Prediction can hence be framed as GVF, where the objective is to estimate the expected future sum of discounted cumulants $c_t^i = o_t^i$ from state s:

$$\begin{aligned} v_t^i(s) &\doteq \mathbb{E}\left[G_t^i \,\middle|\, s_t = s \right] \\ &= \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k c_{t+k+1}^i \;\middle|\; s_t = s \right] \end{aligned} \tag{6.1}$$

Following usual practice in the continuing RL setting, a discount rate $\gamma \in [0, 1)$ is used to ensure a finite sum. We model the problem as a Markov reward process (MRP), as is common for prediction tasks, since the observations contain information about any actions that might have been taken by the behavior policy.

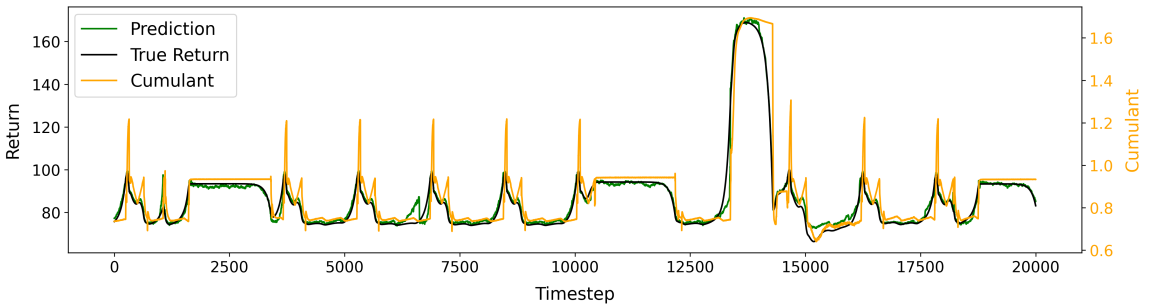


Figure 6.2: Cumulants and discounted returns for the PIT300 membrane pressure sensor for a 20k step slice of the training data. Returns are computed using $\gamma = 0.99$ and y-axes are of different scales. Returns are used as the prediction target for our learning agent.

While Janjua *et al.* (2023) considered five sensors of different categories in their investigation, we will restrict our focus to three sensors which we find maintains sufficient diversity for our analysis. The sensors we consider are the membrane pressure (PIT300), influent temperature (TIT101), and inlet turbidity (TUIT101). Figure 6.2 shows cumulants and returns of PIT300 for a 20k trajectory in the training dataset.

To measure prediction accuracy, we can compute the true return

$$G_t^i \doteq \sum_{k=0}^{\infty} \gamma^k c_{t+k+1}^i, \tag{6.2}$$

for each step during a rollout retroactively. The root mean-squared error (RMSE) between true and predicted returns can then be computed as $\|\hat{v}_t^i - G_t^i\|^2$ averaged over all T timesteps in a rollout:

$$\text{RMSE} \doteq \sqrt{\frac{1}{T}\sum_{t=0}^{T} \|\hat{v}_t^i - G_t^i\|^2}, \tag{6.3}$$

where $\hat{v}_t^i$ denotes the agent's discounted return estimate for sensor i at timestep t, and similarly for the true discounted Monte Carlo return G_t^i.

Because RMSE across models may be at different scales, we also use a normalized mean-squared error (NRMSE) metric, where the RMSE is normalized by the mean of the true discounted returns in the rollout, $\mu^i = \frac{1}{T}\sum_{t=0}^{T} G_t^i$:

$$\text{NRMSE} \doteq \frac{1}{\mu^i}\sqrt{\frac{1}{T}\sum_{t=0}^{T} \|\hat{v}_t^i - G_t^i\|^2}. \tag{6.4}$$

Both the RMSE and NRMSE are typically averaged across R rollouts.

6.1.3 Dynamic Time Warping for Rollout Similarity

As described in Section 2.4, the DTW algorithm can be used to compute similarity between two time series sequences that vary in length or frequency. In the context of calibration models, DTW can be used to compare rollouts produced by a calibration model to rollouts from the true dataset, providing us with one method of determining rollout accuracy. This is particularly useful when evaluating the quality of Laplacian distance metrics, as the quality of the rollouts produced by these metrics can vary significantly.

An important hyperparameter for the DTW algorithm is the step pattern, which characterizes the specific constraints imposed on the warping functions ϕ_x and ϕ_y. For our experiments, we opt to try four of the most common step patterns, as described in Giorgino (2009):

- **Symmetric2**: Imposes no limit on the number of elements that can be matched from one sequence to a single element in the other, no global path constraints, and requires all elements to be matched. This is the default step pattern used in the **dtw** python package (Giorgino 2009).
- **Asymmetric**: Includes a slope constraint between 0 and 2, and matches each element of the query sequence exactly once.
- **Sakoe-Chiba**: Slope-constrained patterns with slope parameter P and choice of symmetry (symmetric or asymmetric) (Sakoe and Chiba 1978). We use the symmetric variant with $P = 1$ for our experiments.
- **Rabiner-Juang-IV-c**: A characterization of step pattern introduced in Rabiner and Juang (1993), which offers a choice of local continuity constraint, slope weighting function $m_\phi(k)$, and smoothing (true or false) over slope weights. We use the type IV continuity constraint, with type c slope weighting, and smoothing set to true, following Giorgino (2009). For brevity, we will refer to this version of step pattern as *Rabiner-Juang-IV-c*.

A visual representation of the Symmetric2 and Rabiner-Juang-IV-c step patterns can be viewed in Figure 6.3.

We use DTW with either one or all of these step patterns as one form of evaluation for rollouts produced by different calibration models. Specifically, in this chapter, we will use DTW to evaluate different learned Laplacian representations, and in Chapter 7, we will use DTW to compare calibration models trained with different datasets. While it is unclear the degree to which DTW is suitable for evaluating calibration

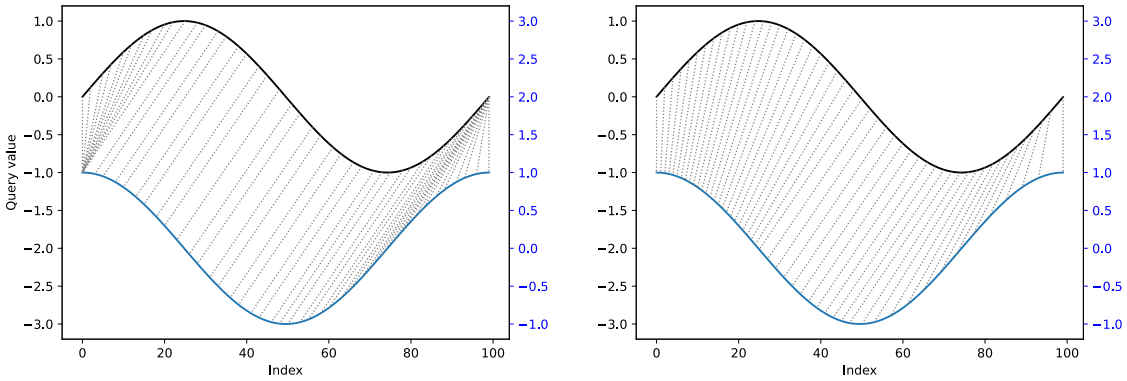


Figure 6.3: Example of dynamic time warping alignment between a sine and cosine wave, using the Symmetric2 (left) and Rabiner-Juang-IV-c (right) step patterns. The optimal alignment is found by warping the time axis to minimize the distance between corresponding points.

models rollouts, we believe that it is useful as a supplement to other evaluation techniques, such as rollout visualization and hyperparameter selection.

6.2 Experimental Setup

Next, we describe the methodology used for training and evaluating calibration models in the WTP. Similar to Chapter 5, we aim to understand the performance of each model with respect to the quality of the rollouts it produces, as well as to its ability to select good hyperparameters for a prediction agent. We discuss the dataset, the agent used for training, and the calibration models that we consider for this study. We also describe the hyperparameters used for training the calibration models, and the evaluation metrics used to assess the quality of the models.

6.2.1 Dataset

The dataset used for this study is collected from the Drayton Valley water treatment plant, and consists of sensor readings from a range of sensors that monitor the water treatment process. First, we will describe the data collection process in more depth, then discuss the normalization and augmentation techniques used to prepare the data for training the calibration models.

Data Collection

Each row in the dataset represents a single observation of the plant's state, which is a vector of sensor readings. The dimensionality of each observation is 480 (matching the number of sensors), and the data is collected at a frequency of one sample per second. The dataset spans a period of over two years, however we will not use the entire dataset for our experiments. Instead, we opt to use one week of data ($\sim$350k transitions) for our initial experiments, and scale up to a full year's worth of data in Chapter 7. Note that we do not have significant missing data issues for the one-week dataset. In the rare case where there are missing sensor readings, we use zero-imputation to fill in the missing values. For the full year dataset, there are 32 sensors that are missing data for extended periods of time, which we opt to remove from the observation space. This did not seem to significantly hurt performance of the prediction agent, so it is likely that these sensors are not critical for the prediction task.

Normalization and Augmentation

Prior to training the calibration model, we follow data normalization and augmentation techniques used in Janjua *et al.* (2023). First, we remove sensors that emit constant values in the dataset and then use percentile min-max normalization over the remaining sensor values to the range $[0, 1]$. This means that we take the 5th-percentile value as the min and 95th-percentile value as the max when performing min-max normalization so that the scaling is more robust to outliers. Next, we augment the dataset by employing several techniques, namely: zero-imputation for missing data and adding mode encodings to encapsulate timing of the plant transitioning between operation modes. We tested other augmentation techniques such as adding trace features to alleviate partial observability, and transforming continuous features to discrete ones using feature binning, however we did not find these improved performance, so we omit them in our experiments. A more detailed description of these techniques can be found in Janjua *et al.* (2023). Following normalization and augmentation, we are left with a feature dimensionality of 142.

6.2.2 Agent

Following the design of the online prediction agent in Janjua *et al.* (2023), we use a TD(0) agent with replay buffer to learn the GVF from equation 6.1. Specifically, we approximate the GVF using a neural network with 2 hidden layers of 512 units each. The network takes as input the augmented observation vector, or agent state $\boldsymbol{s_t}$ and

outputs a single value representing the predicted future sum of discounted values of the PIT300 sensor. We use a discount rate of $\gamma = 0.99$ and a replay buffer size of 1M samples with batch size of 256. The agent is trained using the Adam optimizer, where the learning rate α is the hyperparameter swept over for our experiments.

6.2.3 Calibration Models

We consider the same calibration models as in Chapter 5, namely a feedforward neural network, a recurrent neural network, a KNN model with Euclidean distance metric, and a KNN model with Laplacian distance metric. The implementation and training details for each model are described below.

KNN

We train a bootstrapped Laplacian KNN model consisting of 5 bootstraps over the training dataset, using $k = 3$. A sweep was performed over Laplace training hyperparameters, using standard hold-out validation on the training set. In addition to performing validation with the dynamics awareness and representation uniqueness scores described in 4.5.2, we found it necessary to introduce an additional step for evaluating learned Laplacian representations. This was due to the fact that distance metrics that scored well in terms of training loss and validation scores would sometimes vary in the visual quality of the rollouts they produced. Figure 6.4 shows rollouts from models that use Laplacian representations with similar dynamics awareness scores, but that differ in number of training steps and have very different rollout quality. To address this, we first selected the top 3 distance metrics based on the validation scores, and then selected the one with rollouts that most closely match the true environment using dynamic time warping with the Rabiner-Juang-IV-c step pattern to compute time series similarity. Note that we only evaluated rollout similarity for the PIT300 sensor. A final Laplacian representation was selected using these two measures and subsequently used to construct the bootstrapped calibration model.

NN and RNN

We train a 2-layer neural network and a 2-layer GRU model on the training dataset using the same training and validation splits as the KNN model. We use one-step ahead prediction as the training objective; we tried multi-step prediction target strategies (Talvitie 2014; Venkatraman *et al.* 2015) but did not find them to be effective at producing better rollouts. We also experiment with standard techniques for improving GRU performance, such as dropout and burn-in periods. We perform grid search

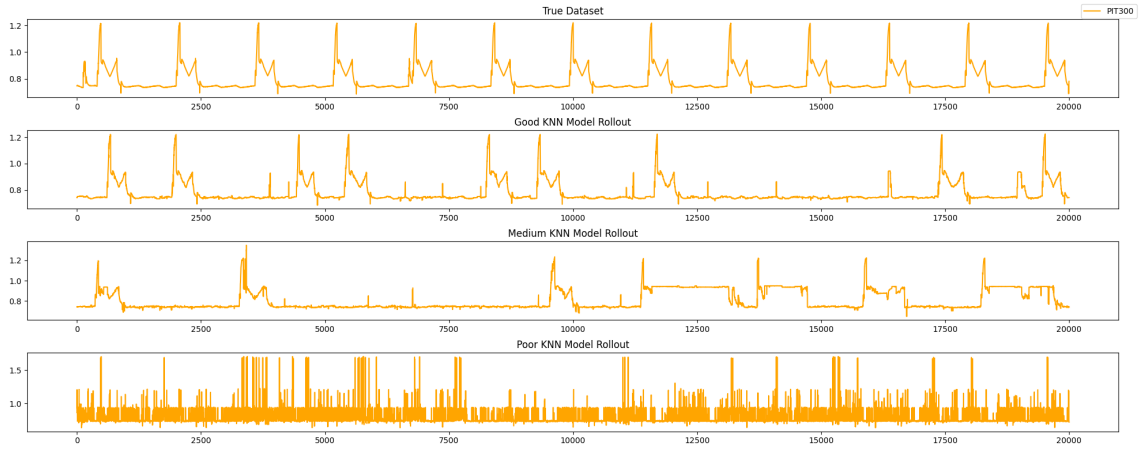


Figure 6.4: Sample rollouts of the PIT300 sensor using three distance metrics that produce good, medium, and poor rollouts compared to a slice of the true dataset. DTW yields normalized distances of 0.023, 0.043, and 0.080 respectively, giving us a useful way to rank distance metrics.

sweeps over various hyperparameter combinations to select the hyperparameters that perform best on the validation set - the final hyperparameters used can be found in Appendix A.1.

6.2.4 Evaluation

We evaluate the calibration models using two main approaches: the quality of the rollouts produced by each model, and the ability of the models to select hyperparameters for a prediction agent. These evaluation procedures are described below.

Rollout Quality

Defining a quantitative metric for evaluating rollout quality is difficult for the WTP. Specifically, time series comparisons are made tricky by the fact that there may exist temporal offsets or changes in frequency between two otherwise similar time series. While we could use a technique such as DTW to mitigate such issues, a single scalar metric is limited in its ability to convey information. Instead, we take a qualitative approach of visualizing individual sensor time-series and comparing between the model rollouts and the true dataset. Each model is rolled out for 30k steps starting from an identical start state, and individual sensors PIT300, TIT101, and TUIT101 are plotted.

Hyperparameter Selection

We sweep over Adam learning rates for the TD(0) prediction agent in the dataset and in the calibration models. Because we are doing prediction with respect to a fixed policy, the test dataset described in 6.2.1 can be used as a proxy for the online setting. We select a rollout length that is roughly one third the size of the online dataset (22,100 steps) such that there is some diversity across rollouts while still allowing enough steps to simulate a reasonable learning timeframe. The last 2,100 steps of each rollout are truncated for the RMSE computation. This truncation length k was selected such that $\gamma^k < 10^{-10}$, i.e. large enough that we can ignore the effect of rollout termination on computed returns. We use a discount rate of $\gamma = 0.99$ for the agent and computed returns. As in previous sections, we evaluate calibration models by their ability to match the sensitivity curves produced by the true environment.

Agent performance in the true environment and in non-bootstrapped calibration models is measured and plotted using NRMSE, as described in Equation 6.4. Agent performance in individual KNN bootstraps is done using RMSE, however due to how hyperparameter ranking is reconciled *across* bootstraps (see Section 4.6), we plot the hyperparameter *rank* directly instead of RMSE or NRMSE. Because rank and NRMSE are typically at different scales, we plot these on separate axes. NRMSE will typically be plotted along the left y-axis, and rank on the right y-axis. Since we care mainly about relative rankings of hyperparameters and the shape of the sensitivity curve, and less about the raw performance values, we are able to compare NRMSE and rank curves within a single plot.

6.3 Results

We present the results of our experiments in the WTP, focusing on the quality of the rollouts produced by each calibration model, and the ability of the models to select hyperparameters for a prediction agent. We first visualize the rollouts produced by each model, and then present the results of our hyperparameter sweeps. We find that the Laplacian KNN produces the best rollouts, followed by the Euclidean KNN, and then the NN and GRU models. We also find that the Euclidean KNN model is best at selecting hyperparameters, followed by the Laplacian KNN, and then the NN and GRU models.

6.3.1 Visualizing Rollouts

Plots of PIT300, TIT101, and TUIT101 sensor values for each model rollout can be seen in Figures 6.5, 6.6, and 6.7, respectively. Note that while each model predicts all 142 features in the state-space, we only visualize three of those sensors here. Overall, we find that KNN model rollouts match closely with the true data for PIT300, but appear to have more trouble with TIT101 and TUIT101. In each case, the Laplacian KNN appears to produce better rollouts than its Euclidean counterpart. On the other hand, the NN and GRU models struggle across all sensors. For PIT300, both appear to collapse towards predicting a constant value after 500 timesteps. For TIT101 and TUIT101, the NN model oscillates around some value in roughly the correct range as the true environment, but does not appear to match the true sensor patterns very well, while the GRU again predicts a constant value for TIT101, and produces a step function-like pattern for TUIT101. While rollouts are certainly not perfect for TIT101 and TUIT101 in the KNN models, it is possible that sufficient information is captured by these models for the purpose of hyperparameter selection (while policy transfer may be infeasible). Based on these results, we hypothesize that the Laplacian KNN should be best for selecting hyperparameters, followed by the Euclidean KNN, and then the NN and GRU models.

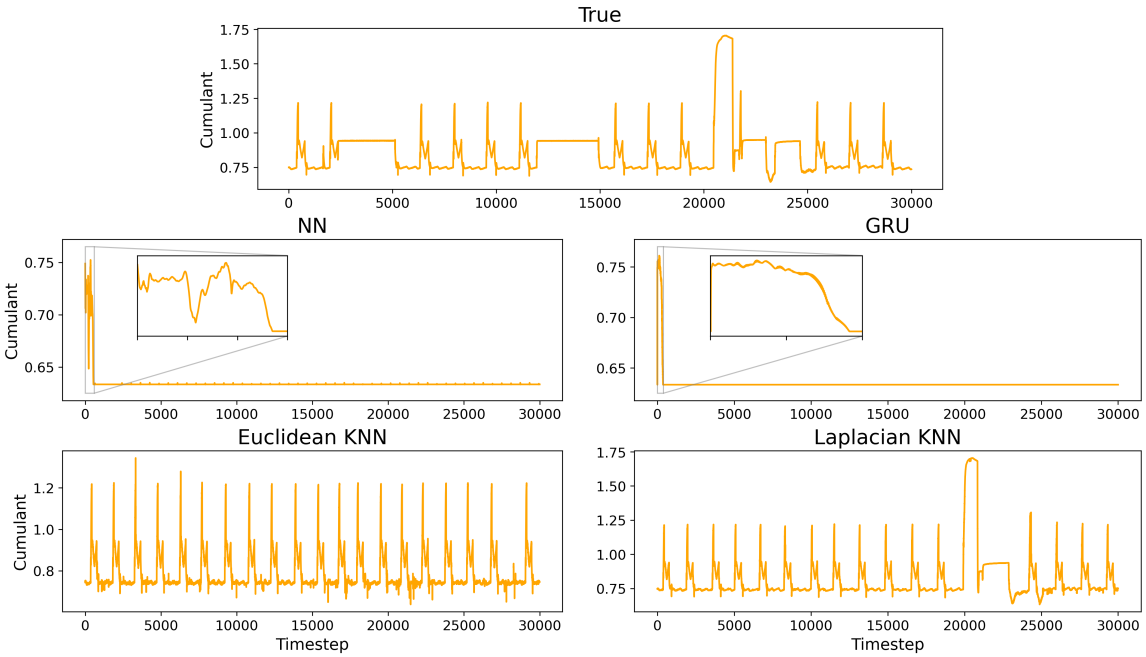


Figure 6.5: PIT300 sensor (membrane pressure) rollouts in the true environment and calibration models. Each model is rolled out for 30k steps, beginning from the same start state.

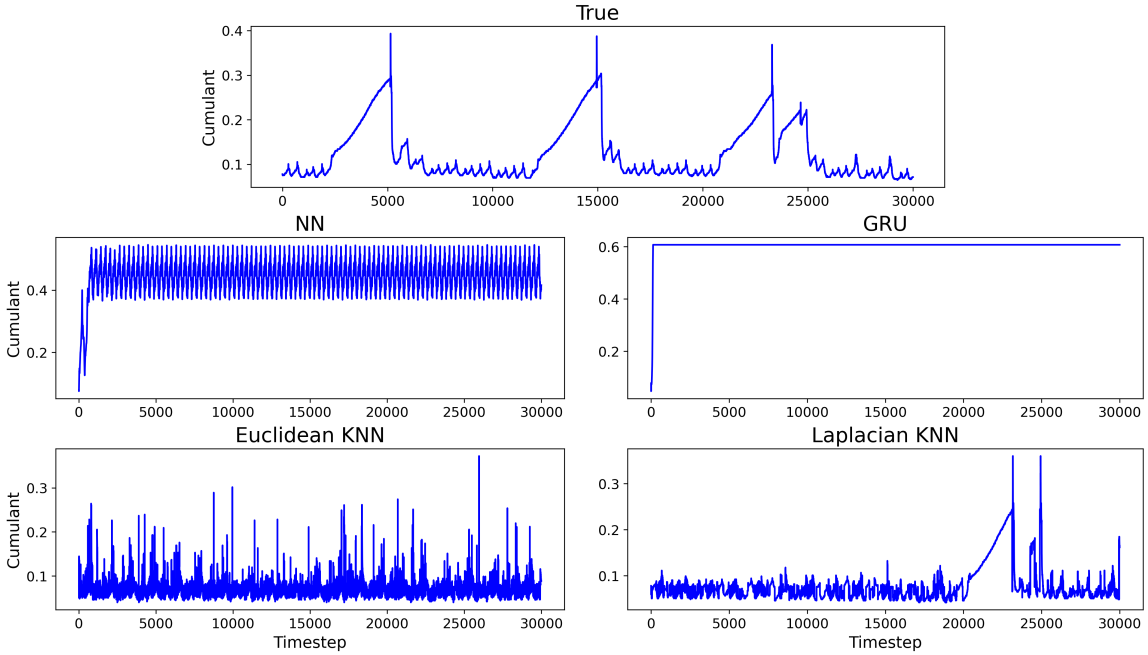


Figure 6.6: TIT101 sensor (influent temperature) rollouts in the true environment and calibration models. Each model is rolled out for 30k steps, beginning from the same start state.

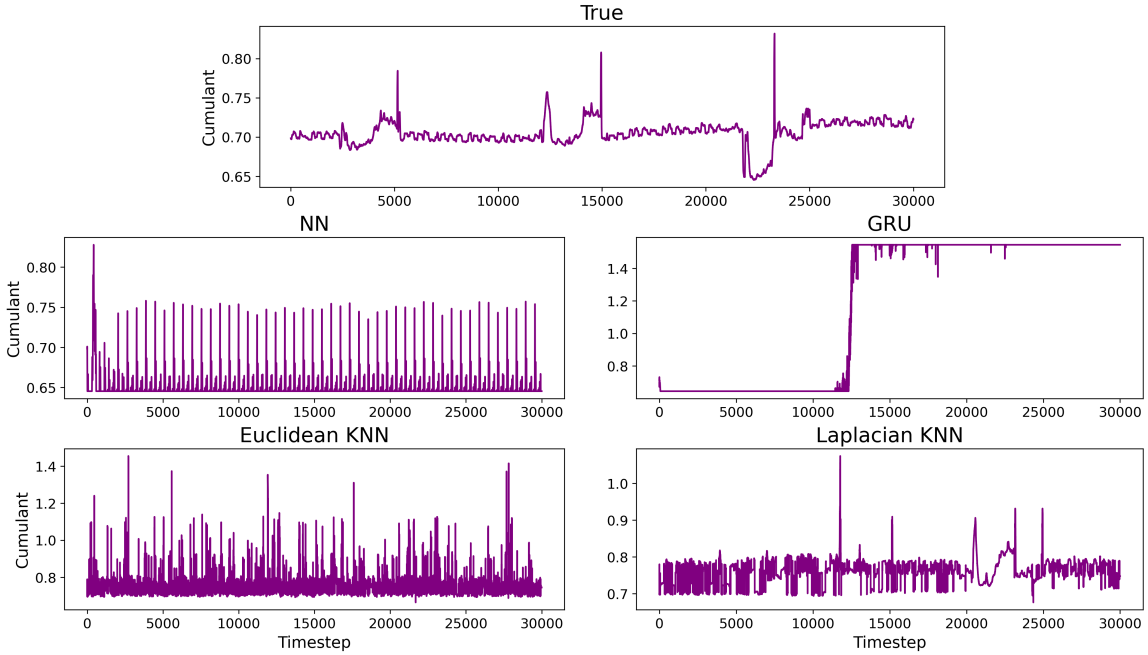


Figure 6.7: TUIT101 sensor (influent turbidity) rollouts in the true environment and calibration models. Each model is rolled out for 30k steps, beginning from the same start state.

6.3.2 Hyper Selection

Figure 6.8 shows the results of our hyperparameter sweeps for fine-tuning prediction on the PIT300, TIT101, and TUIT101 sensors. Across all sensors, we see that the Euclidean and Laplacian KNNs are typically able to match the online sensitivity curve, meaning that hyperparameters are ranked in roughly the same order in these calibration models as the true environment. For example, in the TIT101 plot of Figure 6.8, the online environment considers $[1e{-}5, 0.0001, 0.001, 0.01]$ as all having roughly equal (and optimal) performance, which is consistent for the Euclidean KNN, and nearly consistent for the Laplacian KNN, where 0.001 is ranked as worse, and 0.01 is ranked as better. In contrast, the NN and GRU models perform inconsistently across sensors. The GRU calibration model manages to roughly match the online sensitivity curve for PIT300, and arguably for TIT101, but not for TUIT101, while the NN calibration model is unable to match the online sensitivity curve in all three cases.

One potential issue with our usage of bootstraps is that by condensing multiple performance values into a single rank causes us to lose information about the variance *within* each model. For example, in the PIT300 plot of Figure 6.8, the online environment considers $[0.0001, 0.001, 0.01]$ as all having roughly equal (and optimal) performance, whereas the KNN models produce an ordering between these hyperparameter settings, without any uncertainty. It may be preferable for the bootstrap reconciliation method to incorporate performance uncertainty, though we will leave this for future work.

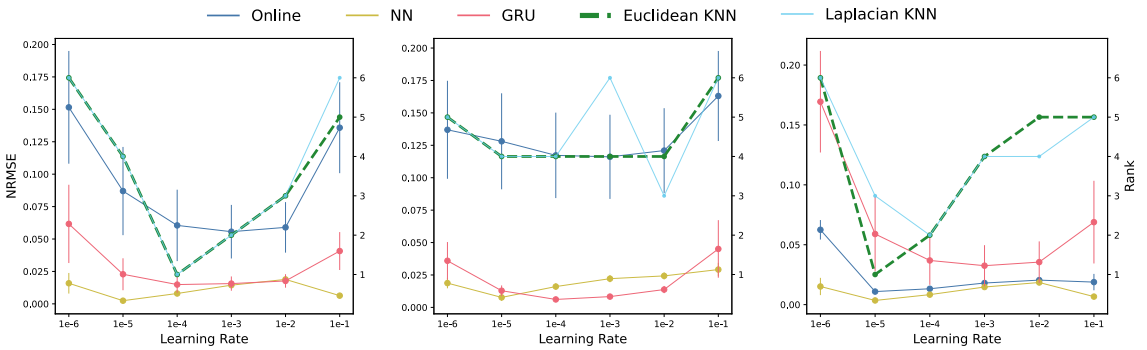


Figure 6.8: Learning rate sensitivity curves for the true (Online) environment and calibration models using the 1-week WTP dataset over three different sensors. From left to right, the plots are for sensors PIT300, TIT101, and TUIT101. NRMSE with 95% confidence intervals are shown for non-bootstrapped models, while worst rank is used for bootstrapped models (i.e. KNN models), which is plotted against the left rank y-axis. Wider, dotted lines are used for the Euclidean KNN to distinguish it from the Laplacian KNN line, which often overlap.

6.4 Discussion

Through the water treatment experiments, we have shown that the KNN calibration models show promise in their ability to produce realistic rollouts over long horizons and their subsequent utility towards hyperparameter selection. In more detail, we outline additional findings from these experiments in bullet-point:

- The current Laplacian learning process is not ideal. The dynamics awareness score does not always correlate with model quality, for example in Figure 6.4 the Laplacian representations used for each model had similar dynamics awareness score (with different amounts of training time), but much different rollout quality. To mitigate this, an extra step of either manual visual analysis of rollouts or usage of DTW to compute rollout similarity was needed. Improvements on this model might involve updating either of the training objective or validation metric, or both. Specifically, devising a differentiable loss function that incorporates rollout quality could be a useful direction.

- The NN and GRU models yielded poor rollouts over long horizons. More investigation needed is needed to understand the root causes here, especially since we found these models to be sufficient in Chapter 5 in simpler environments. Possible causes for this degradation include high dimensionality, non-stationarity, complexity in dynamics, etc.

- Interestingly, the Laplacian KNN had worse hyperparameter tuning performance than the Euclidean KNN for the TIT101 and TUIT101 sensors, as we can see in Figure 6.8. One possible explanation for this is that we selected a Laplacian representation that performed best for a single sensor (PIT300), but may have been suboptimal for other sensors. This points to the need for a better way to validate Laplacian representations, as pointed out above and in section 4.5.2.

- We do not have much intuition for the properties of the environment that are most important for hyperparameter selection. For example, we see that the GRU model is prone to predicting a constant sensor value, but is still able to produce reasonable looking hyperparameter curves. Developing models that are more tailored towards simulating properties important for hyperparameter selection, or more generally, value function estimation, is also an avenue of future work.

In this section, we demonstrated the effectiveness of calibration models in a real-world setting, and showed that non-generalizing KNN models yield better long-horizon rollouts than generalizing NN and GRU models within this environment. In the next section, we will explore several considerations for employing a calibration model in the real world, which we expect more closely simulate how one might be used in practice.

Chapter 7

Towards Real-World Deployment

There is often a sizable gap between an algorithm working within the context of a controlled research experiment and in the increased complexity of real-world application. For example, while a small dataset might be sufficient for demonstrating, by proof of concept, that a technique is feasible, it does not guarantee that the technique will work to the same degree when scaled up to a much larger, more diverse dataset. Further, certain experimental design decisions may not accurately represent the setting in which the algorithm will be deployed. In this section, we hope to bridge the gap towards application by exploring several modifications to the experimental setting that was followed in Chapter 6. We will investigate how the KNN calibration model might be scaled up to handle a full year's worth of data, how a dataset might be used for both agent pre-training and calibration model construction, and discuss the ramifications of distribution shift between the calibration model and the true environment at deployment time. Finally, we will show experimental results that illustrate how our approach can be modified to tackle these issues.

7.1 Scaling Up

In the water treatment setting, and similarly in industrial settings more generally, we have access to offline data logs on the order of years. Training our models on large proportions of this data means extracting all the information we can from the dataset, which typically involves learning representations that might help in tackling non-stationarity and seasonality, and ideally making the model more robust to a wide range of scenarios. To this end, we will consider the setting where we have access to one full year's worth of data, and discuss modifications made to the KNN calibration model to handle this scale. This is not the first study in which RL is applied towards large datasets. For example, Zhan *et al.* 2022 uses between one and two years of data

to train their models, while Luo *et al.* 2022 uses just under one year of data.

Given the standard observation sampling rate of once per second from the WTP, the one-year dataset consists of roughly 32M samples. While it might be possible to leverage this full dataset for learning, we hypothesize that such fine-grained sampling is not critical for prediction, and opt to sub-sample to reduce the size of the dataset. We sub-sample every 10th transition, changing our sampling rate to one sample per 10 seconds, and reducing our dataset size to ∼3.2M samples.

In terms of computational complexity of the KNN calibration model, constructing the model is a two-step process of first building the KD-tree of all transitions, and then converting this into a full lookup table (i.e. *neighbor table*) such that next-state prediction is constant time. KD-tree construction is $\mathcal{O}(nd \log n)$ (Brown 2014) and neighbor table construction is $\mathcal{O}(nk \log n)$, so the overall time complexity is $\mathcal{O}((d+k)n \log n)$, where n is the number of samples, d is the dimensionality of samples, and k is the number of neighbors. With $n \approx 3.2M$, $d = 142$, and $k = 3$, we find that this takes ∼10 hours on a 2 GHz Quad-Core Intel Core i5 processor.

While this is relatively costly, we find the time to be acceptable for our experiments since neighbor table construction only needs to be run once on the dataset. However, other techniques that can be used to speed up KNN construction include dimension reduction (e.g. PCA, autoencoder, etc.), prototype selection (Wilson and Martinez 2000) for further data reduction, and approximate nearest neighbor search (Arya *et al.* 1998; Indyk and Motwani 1998).

7.2 Fine-Tuning

Given a real-world deployment in the water treatment plant, we would like to make full use of our offline dataset towards solving the prediction or control task at hand. In practice, this means that we would likely want to use some form of pre-training to initialize the agent's policy to a good starting point, i.e. give the agent a "warm-start". This is especially important in the case where no simulator exists, and where a poor cold-started policy could be costly or dangerous to run. Once an agent has been pre-trained, we still want it to be able to adapt to the evolving conditions of the plant and water conditions. While transferring a policy from offline to online is a generally difficult problem (Zhu *et al.* 2023), we are primarily interested in the narrow scope of selecting hyperparameters for such a pre-trained agent. We will refer to this setting where an agent is first pre-trained on an offline dataset and then continues to learn once deployed as the *fine-tuning* setting.

A natural hyperparameter of interest in the fine-tuning setting is the fine-tuning

learning rate, which is simply the optimizer learning rate used for the deployed agent. Ostensibly, the pre-trained agent will have compressed a large amount of information about the task into its policy, but will still need to adapt online to achieve good performance. In the case of many learning algorithms, such as TD(0), this requires the agent to be deployed with a sensible learning rate. Hence, in the following experiments, we will shift our focus from evaluating "from-scratch" agent learning rate selection to evaluating the fine-tuning learning rate selection.

Using the offline dataset for both agent pre-training and for constructing a calibration model means that we should consider potential ways of sharing the dataset between these tasks. One simple approach is to partition the dataset and use all the transitions before a certain point for pre-training and all the transitions after for calibration model construction - such that the partitioning roughly simulates an offline-to-real transfer point. While there are likely more nuanced approaches to sharing data between the tasks, we will use this as a starting point for our experiments. Also note that partitioning will be sensitive to distribution shifts (i.e. non-stationarity) that may exist in the dataset, so understanding the dataset in advance is important.

7.3 Different Deployment Periods

In a complex, real-world system like the water treatment plant, the dynamics of the system may change drastically depending on the time period of interest. In Figure 6.1, we can see how sensor patterns change throughout the year - these changes could be due to varying conditions ranging from rainfall and temperature, to filter cleanliness and sensor drift within the plant. While previously we only used a test set that immediately proceeded the training set, in this section we will explore test sets that occur one week, one month, and three months after the training set. Note that test sets here are used to simulate a specific deployment period. As an example, the time-series for TIT101 in these time periods are shown in Figure 7.1, where we can see changing patterns across months.

In theory, scaling up the calibration model to a larger dataset should allow it to capture more diverse dynamics around e.g. sensor drift, seasonality in incoming water characteristics, changing plant conditions, etc. Hence, one goal for the calibration model is that it might be used to simulate *different deployment periods*. Of course, this may be extremely difficult to achieve given that we don't know what the future data distribution will look like, but it's possible that with enough data, good generalization is possible.

Given a calibration model $\hat{p}$, we can think of several techniques for sampling rollout

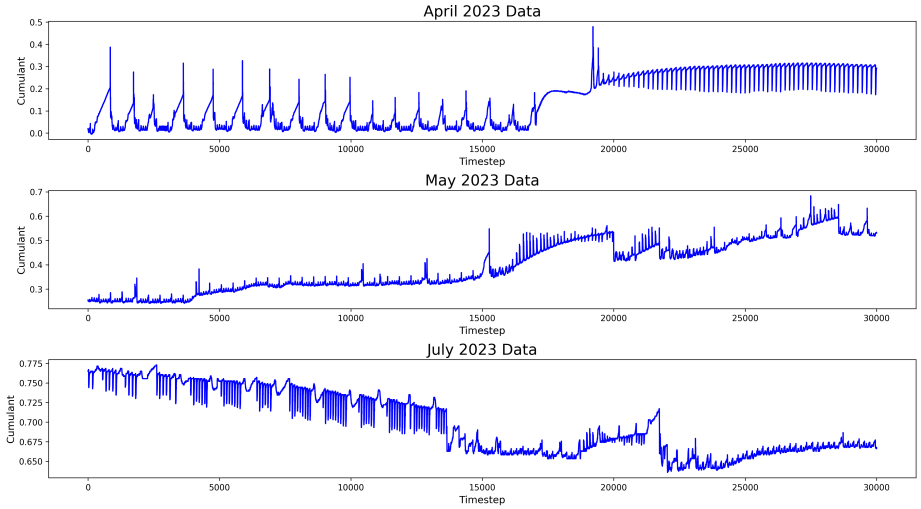


Figure 7.1: TIT101 time-series from several test deployment periods in the WTP. We show 30k timestep slices, which corresponds to ∼3.5 days at 10x sub-sampling.

start states $\mathcal{S}_0 = \{s_0^{[0]}, ..., s_0^{[r]}\}$, where r is the number of runs for each hyperparameter setting, to simulate a specific deployment period:

(i) Randomly select $\mathcal{S}_0$ from the entire dataset. This is a baseline method which we wouldn't expect to have much success.

(ii) Select $\mathcal{S}_0$ from the same month as the deployment period, if they exist in the training set. E.g. if the deployment period is July 2023, we will select start states from July 2022 in the training set.

(iii) Collect samples from the online period and find nearest neighbors in the dataset to use for $\mathcal{S}_0$, where the nearest neighbor is found using the Laplacian distance metric.

In our experiments, we will test (i) and (iii), hypothesizing that (iii) should more closely simulate the target deployment period. While (ii) is also interesting, it is not clear how closely sensor patterns from the same month in a previous year will map to a subsequent year, i.e. due to sensor drift and changing plant conditions. Further, our use of bootstrapping means that each individual start state will not exist in all bootstrap models.

7.4 Experimental Setup

We try two different schemes for sharing the training dataset between agent pre-training and calibration model construction:

(i) 50/50 split of training dataset for agent pre-training and calibration model construction. The first 6 months of data are used for pre-training and the second 6 months are used for the calibration model.

(ii) Full training dataset is used for both pre-training and calibration model construction.

We pre-train a prediction agent on the training dataset using TD(0) with replay. The replay buffer is filled with all transitions from the training data and the agent is pre-trained for 1M steps with early stopping based on RMSE over the held-out validation set. We find that an agent with learning rate 1e-5 trained for 1000 epochs with batch size 256 (i.e. $\sim$5.5M steps for the 6-month dataset and $\sim$11M steps for the full year dataset) produces high accuracy predictions on the validation set and freeze this network to use for fine-tuning - see a full list of hyperparameters in Table A.3. We find prediction accuracy to be fairly low with $\gamma = 0.99$ given this amount of pre-training, so we shorten the prediction horizon using $\gamma = 0.9$. Hyper sweep learning curves are shown in Figures A.1 and A.2.

To test performance of calibration models at selecting hyperparameters, we sweep over the optimizer learning rate $\alpha \in [1e{-}7, 1e{-}6, 1e{-}5, 1e{-}4, 1e{-}3, 1e{-}2, 1e{-}1]$. We perform 30 runs per learning rate in the online setting, and 10 per bootstrap in the KNN models, resulting in 50 total runs per learning rate for bootstrapped models. Performance is measured using RMSE averaged over the final 25% of each run, so that early training error does not dominate the metric. Note that all experiments in this section use a learned Laplacian distance metric.

7.5 Results

So far we've already seen that it is computationally feasible to scale up the KNN calibration model to a large dataset. In our remaining experiments, we seek to answer two further questions which are presented below.

Question 1: *Does scaling up the KNN calibration model improve its generalization capabilities?*

In training a calibration model on a larger dataset, one of the primary aims is for the model to improve its ability to generalize to a wider range of scenarios (i.e. deployment settings). To test this, we compare rollouts from a KNN trained on 12 months of data and a KNN trained on the one-week dataset from Chapter 6, which we refer to as the *12-month KNN* and *1-week KNN*, respectively. These rollouts are

evaluated by their ability to match true rollouts from three test datasets, giving us one view into the generalization capabilities of each model. For each test dataset, we select 30 random states from which to begin the true rollouts, and use their nearest neighbor state as start states for KNN rollouts, as described by technique (iii) in Section 7.3. Rollouts are performed for 10k steps, and we look at the PIT300, TIT101, and TUIT101 sensors in our analysis. We analyze the rollouts qualitatively through visualization, and quantitatively by using DTW to measure distance between individual sensor sequences from the true and model-generated rollouts.

In Figures 7.2, 7.3, and 7.4, we show a subset of rollouts for each sensor relative to the May 2023 test set. While it is difficult to definitively evaluate the rollouts by qualitative (i.e. visual) analysis, the general trend appears to be that the 12-month KNN is able to simulate a wider range of sensor patterns than the 1-week KNN. We also measure how similar each model rollout is compared with its corresponding rollout in the true dataset which begins from timestep t_0. Results using the DTW similarity metric are shown in Figure 7.1 across four common step patterns (Giorgino 2009). We find that results are mostly consistent between step patterns, with all patterns agreeing the 12-month KNN produces better rollouts for TIT101 and the 1-week KNN for TUIT101, and three out of four patterns agreeing that the 1-week KNN produces better rollouts for PIT300. Based on these three sensors, it is inconclusive whether the 12-month KNN yields better generalization from a quantitative standpoint.

	PIT300		TIT101		TUIT101	
	1-Week	12-Month	1-Week	12-Month	1-Week	12-Month
Symmetric2	$\mathbf{414.02 \pm 0.02}$	483.24 ± 0.02	910.20 ± 0.05	$\mathbf{782.05 \pm 0.04}$	$\mathbf{154.78 \pm 0.01}$	194.86 ± 0.01
Asymmetric	$\mathbf{224.0 \pm 0.02}$	283.38 ± 0.03	523.05 ± 0.05	$\mathbf{431.87 \pm 0.04}$	$\mathbf{100.92 \pm 0.01}$	153.81 ± 0.02
SymmetricP1	805.51 ± 0.04	$\mathbf{782.79 \pm 0.04}$	1031.66 ± 0.05	$\mathbf{898.61 \pm 0.04}$	$\mathbf{199.55 \pm 0.01}$	268.24 ± 0.01
RabinerJuang	$\mathbf{339.87 \pm 0.03}$	367.87 ± 0.04	521.66 ± 0.05	$\mathbf{447.82 \pm 0.04}$	$\mathbf{100.43 \pm 0.01}$	154.09 ± 0.02

Table 7.1: Dynamic time warping (DTW) distances computes between rollouts from KNN models (1-Week and 12-Month) and true rollouts, averaged over three test sets with 30 rollouts each, and computed across three sensors. Smaller distance is better, and the best model for a specific sensor is presented in bold font. We try four common step patterns for the DTW algorithm, with results for each occupying a separate row.

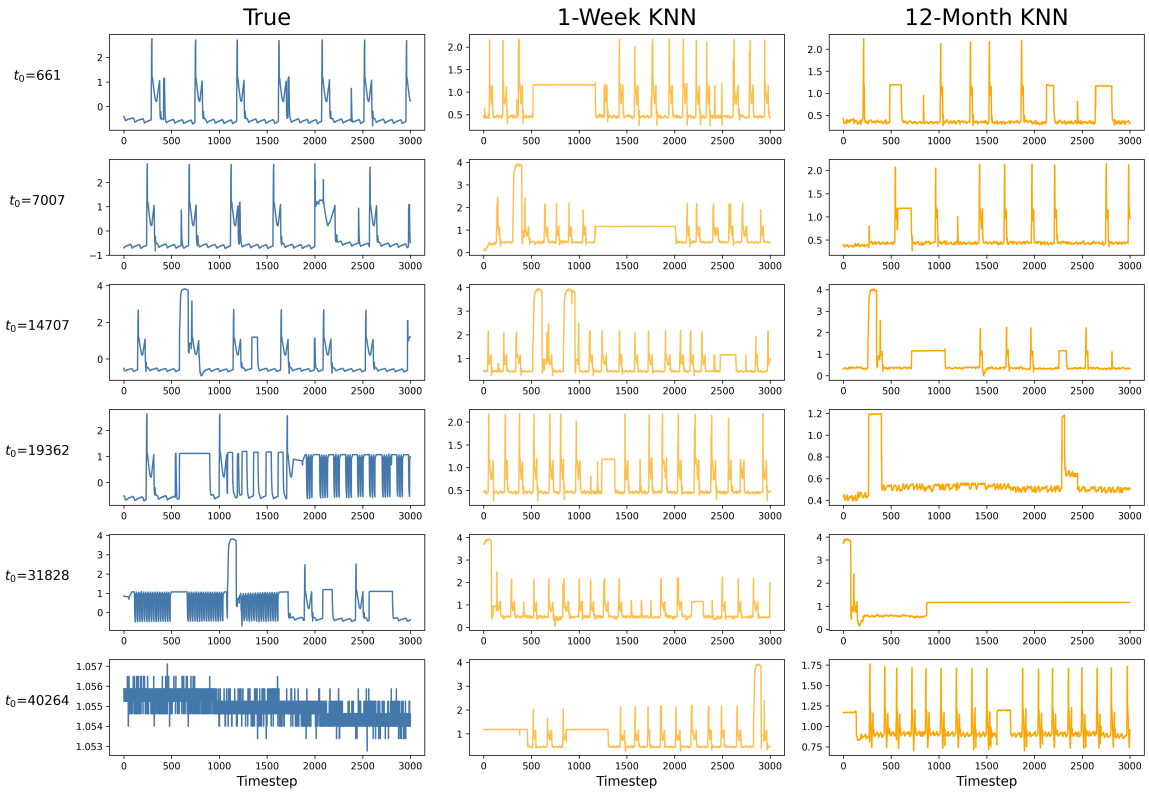


Figure 7.2: PIT300 rollouts for the 12-month and 1-week WTP KNN calibration models using targeted start states from the May 2023 dataset.

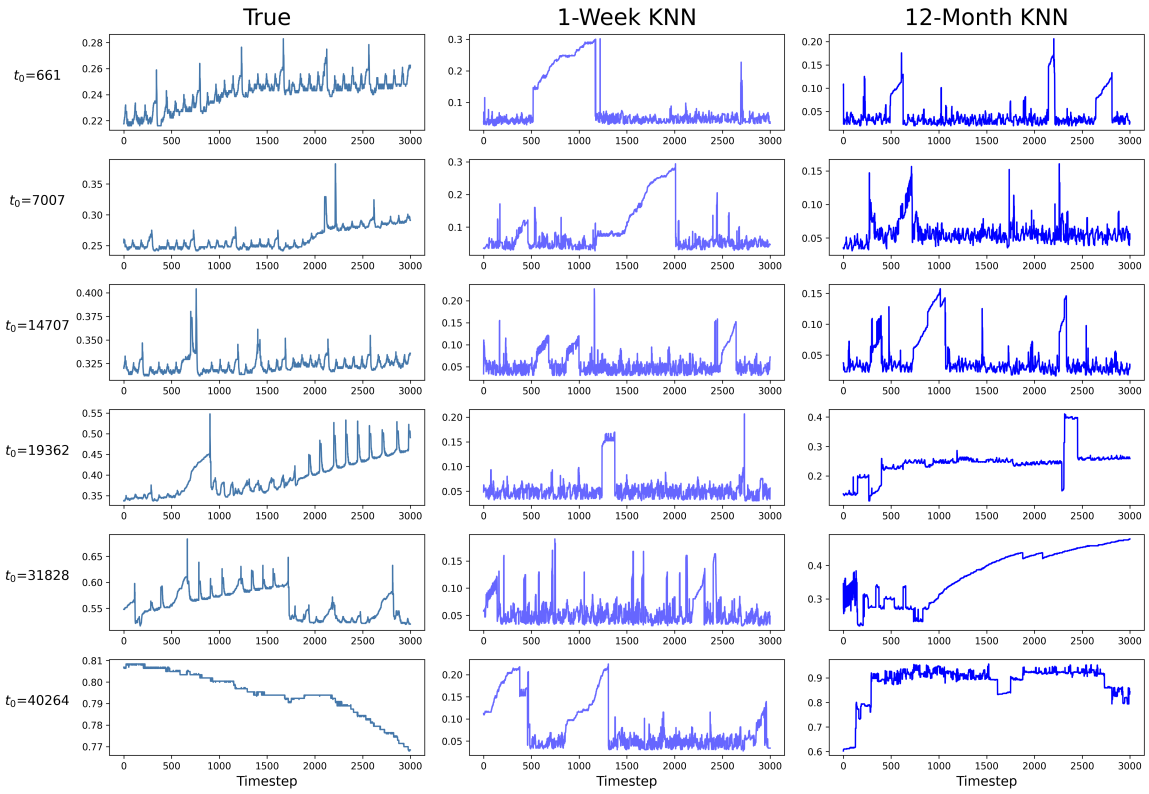


Figure 7.3: TIT101 rollouts for the 12-month and 1-week WTP KNN calibration models using targeted start states from the May 2023 dataset.

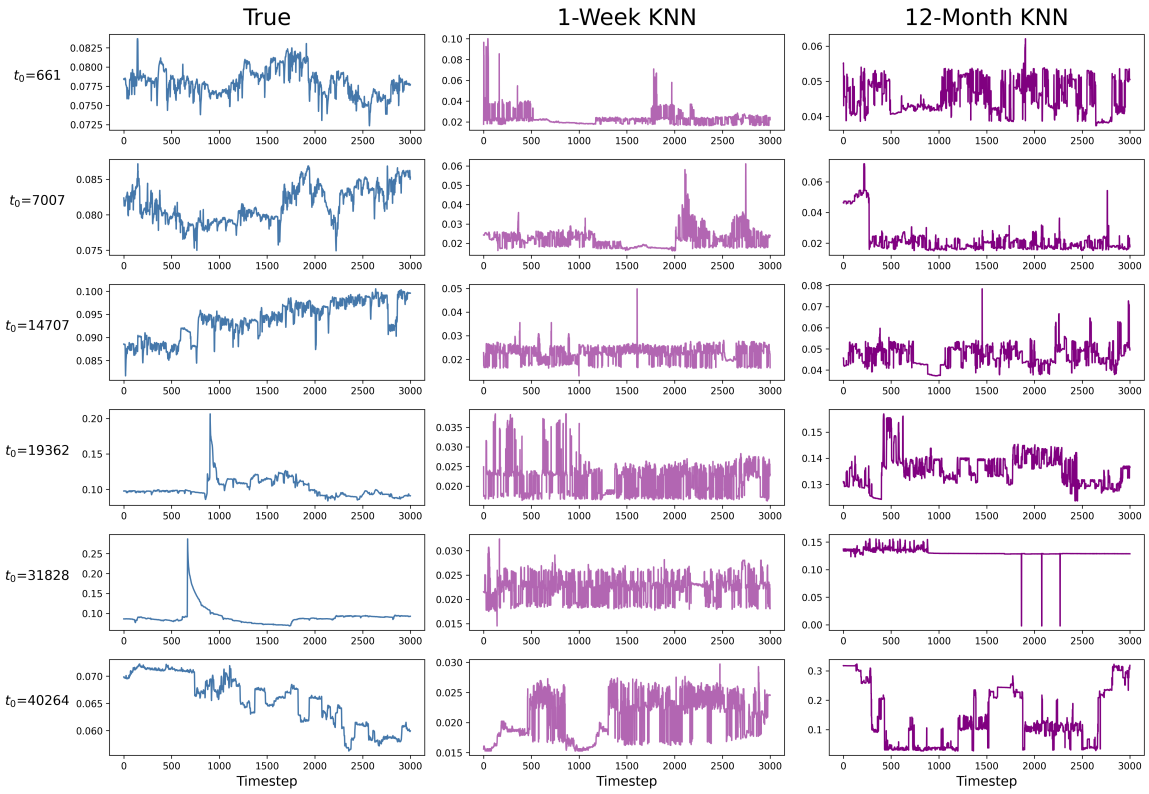
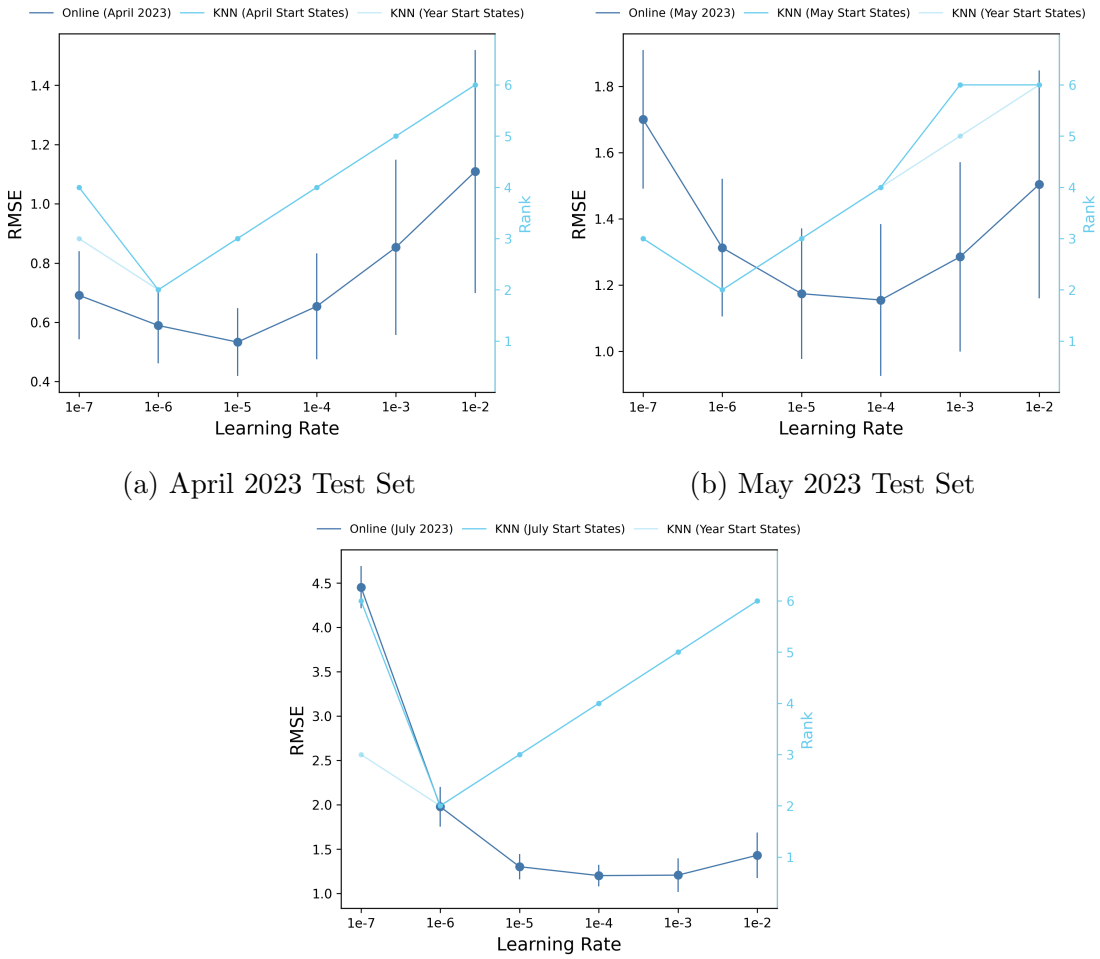


Figure 7.4: TUIT101 rollouts for the 12-month and 1-week WTP KNN calibration models using targeted start states from the May 2023 dataset.

Question 2: *Can the KNN calibration models be used to simulate a specific deployment period for the purpose of selecting the fine-tuning learning rate?*

By training the calibration model on a larger, richer dataset, we would expect it to offer greater generalization capabilities. However, an open question remains as to how these generalization capabilities can be realized, for example when attempting to simulate new, unseen deployment periods. Because we are specifically interested in generalization capabilities with respect to hyperparameter selection, our experiments look at how well the 50/50 partitioning and full dataset approaches perform at selecting a fine-tuning learning rate for different deployment periods, using start state sampling techniques (i) and (iii) described in the previous section.

In figure 7.5, we see that using the full dataset for both pre-training and model construction is not particularly useful for selecting the fine-tuning learning rate. Regardless of deployment period, the KNN produces roughly the same learning rate curve, even when using targeted start states, always finding a small learning rate of 1e−6 to be optimal. This is likely due to the fact that for fine-tuning, the agent has been pre-trained on the same data as the calibration model, and hence the calibration model produces a data distribution that the agent is already well-trained on. Conversely, in figures 7.6, where different dataset partitions are used for pre-training and model construction, we see that the calibration models do a better job at simulating the distribution shift, with learning rate curves closer to the true curves. We also find that using targeted start states give different results than random (i.e. full year) start states, but we do not find evidence that this strategy actually helps the model to simulate the desired time period.

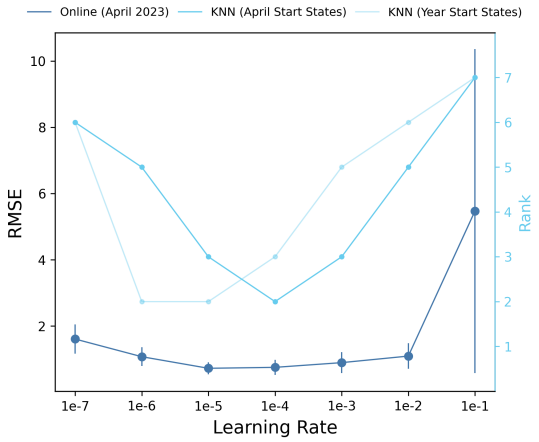


(a) April 2023 Test Set

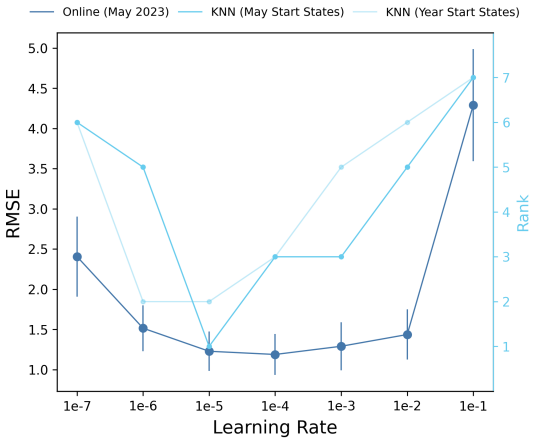
(b) May 2023 Test Set

(c) July 2023 Test Set

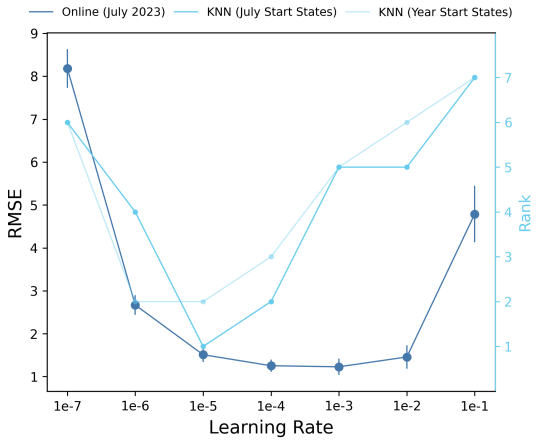
Figure 7.5: Learning rate sensitivity curves for the true (Online) environment and calibration models for PIT300 in the one-year WTP dataset with the full dataset used for both pre-training and model construction. RMSE with 95% confidence intervals are shown for non-bootstrapped models, while worst rank is used for bootstrapped models (i.e. KNN models), which is plotted against the rank axis. Learning rate 1e−1 is omitted here because RMSE values and confidence intervals are extremely large, which hurts plot scaling.



(a) April 2023 Test Set



(b) May 2023 Test Set



(c) July 2023 Test Set

Figure 7.6: Learning rate sensitivity curves for the true (Online) environment and calibration models for PIT300 in the one-year WTP dataset with 50/50 partitioning between pre-training and model construction. RMSE with 95% confidence intervals are shown for non-bootstrapped models, while worst rank is used for bootstrapped models (i.e. KNN models), which is plotted against the rank axis.

7.6 Discussion

- We show that the KNN model can be scaled up to upwards of 3M samples with minimal changes to the core algorithm, apart from reducing dataset size using subsampling. We also discussed possible strategies that could be pursued to reduce computational complexity, should the dataset size become untenable.

- The effectiveness of the calibration model may depend on the hyperparameter of interest. E.g. fine-tuning learning rate is difficult to tune due to its dependence on how much the data distribution during deployment has changed since the agent's training dataset was collected.

- The effectiveness of the calibration model also depends on how the offline dataset is used. For example, if the offline dataset is also used for agent pre-training, then a decision must be made on which portions of the dataset are used for model construction and pre-trainng. While we explored two strategies for splitting the dataset and found that a 50/50 split performed better than using the entire dataset for both uses, this result was largely contingent on the fact that we were trying to simulate a fine-tuning scenario where there is a distribution shift between the offline and online data. Again, we point to the fact that choosing a fine-tuning learning rate here is likely more dependent on how well the model is able to simulate the true distribution shift, which is in itself a difficult task.

- The KNN model appears capable of capturing information from larger datasets and simulating a range of patterns from within this dataset. This is promising because it suggests that it may be possible to augment the KNN calibration model using more data. However, there are two significant difficulties we encountered when using and evaluating the KNN model in this setting. First, it is tricky to take advantage of this augmented model towards hyperparameter tuning. Even if we have knowledge about the deployment period, it is necessary to develop a method which can encourage the model to behave in a way that is similar to the deployment period. Second, it is unclear what a good measure is to evaluate rollout quality. Because the sensor data can be viewed as a collection of univariate time-series, we decided to try a popular time-series distance measure which accounts for varying frequency and length. However, as we've discussed thoroughly in this work, a measure which computes some multi-step error over a rollout does not necessarily indicate whether this rollout is *realistic*, nor *effective for hyper tuning*. Further exploration into good quantitative measures for time-series rollouts would be beneficial.

Chapter 8

Conclusion & Future Work

Reinforcement learning is a powerful framework for solving sequential decision-making problems, and there are myriad real-world problems that could benefit from its application. Unfortunately, application is rarely straightforward, as agent designers frequently run into challenges ranging from partial observability and learning from limited samples, to reward function design and safety constraints. In this thesis, we've discussed one approach that uses a calibration model to mitigate the difficulties associated with hyperparameter tuning of RL algorithms in tasks where there is no simulator available. While the idea of learning a calibration model is a simple one, its requirement of producing realistic trajectories over long horizons is difficult to both characterize and satisfy. We presented several perspectives on how to characterize model quality, including metrics related to multi-step error, state-space distribution, and invalid transition count. We also introduced different candidate implementations for a calibration model, dividing them into *generalizing* and *non-generalizing* models, where we posit that non-generalizing models may be better at avoiding compounding prediction error and producing stable long-horizon rollouts. The generalizing models we explore are a simple feedforward neural network and a GRU, while the non-generalizing models include a KNN with Euclidean distance, and a KNN with Laplacian distance. Our first batch of experiments compare different model implementations, finding that both generalizing and non-generalizing models were able to perform well in environments with low-dimensionality and simple dynamics. This is a positive result, as it shows that one does not necessarily need a highly accurate model in order to select good hyperparameters, and validates the feasibility of the novel KNN methods. We also show that the usual MSE metric used to evaluate model accuracy is flawed, instead proposing that other perspectives such as state-space distribution could be more meaningful.

In the final chapters of this thesis, we provide the first application of calibration

models to a real-world environment, where we test different models in the Drayton Valley water treatment plant. In this setting, we find that NN and GRU models fail to produce realistic trajectories, while the KNN models perform surprisingly well for some sensors. We contribute to the KNN method in several ways. First, we uncover an issue where the learned Laplacian representation becomes degenerate, and propose a secondary validation score to mitigate this. Second, we demonstrate that the KNN model is in fact able to produce realistic trajectories in a high-dimensional, partially observable environment with complex dynamics. Lastly, we provide perspectives on how to bridge the gap to deploying the KNN model in a real-world system, and show how it can be scaled up to a full year's worth of data.

There are a variety of branches to be explored as natural extensions of this work:

- **Understanding generalizing model failure in WTP**: while we found that NN and GRU models failed to produce good trajectories in WTP, it is unclear the exact reasons why. More investigation here could be enlightening, and some properties to look into would be the high-dimensionality of observations (e.g. one could try dimension reduction techniques for observations), noise in sensor outputs, and the non-stationarity of the dataset.

- **Alternate long rollout models**: basic NN and GRU architectures were just the tip of the iceberg in terms of transition model architectures that we could use. It would be fascinating to combine model techniques discussed in Section 4.7, such as probabilistic ensembles, or Lyapunov stability, with the KNN approach of keeping trajectories within the observed state space.

- **Desired calibration model properties**: while we thoroughly presented views on rollout quality throughout this thesis, it is unclear exactly which properties are necessary to achieve good hyperparameter selection. This extends more broadly into understanding how and why hyperparameters are sensitive to different tasks. This is likely a deep rabbit hole, and it might be the case that the desired properties change depending on the target hyperparameter.

- **KNN improvements**: the KNN models were mostly successful in the tasks we tested, but there are multiple avenues of improvement. The first is improving the algorithm for continuous action spaces - more thought should be put into how to measure distance in the action space, and how to balance this with state-space distance. Second is around improving the Laplacian representation learning and validation process. Using two validation scores is not ideal, and finding a way to incorporate the desired result of long-horizon rollout quality directly

into the learning process (e.g. possibly using a differentiable KNN) could be largely beneficial. Lastly, more thought could be put into voting strategies for reconciling rankings across bootstrap models. While we use a simple worst rank protocol in our work, there is a large space of other strategies that could be taken.

We believe that the idea of learning simulators which require minimal domain knowledge is an enticing idea, and one that can be broadly useful for hyperparameter tuning and beyond. Perhaps, with more advances in this area, the myriad applications of RL in the real-world can move one step closer to a reality.

Bibliography

[1] M. M. Afsar, T. Crump, and B. Far, "Reinforcement learning based recommender systems: A survey," *ACM Computing Surveys*, vol. 55, no. 7, pp. 1–38, 2022.

[2] R. Agarwal, D. Schuurmans, and M. Norouzi, *An optimistic perspective on offline reinforcement learning*, 2020. arXiv: 1907.04543 [cs.LG].

[3] M. Andrychowicz *et al.*, *What matters in on-policy reinforcement learning? a large-scale empirical study*, 2020. arXiv: 2006.05990 [cs.LG].

[4] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu, "An optimal algorithm for approximate nearest neighbor searching fixed dimensions," *Journal of the ACM (JACM)*, vol. 45, no. 6, pp. 891–923, 1998.

[5] M. G. Bellemare *et al.*, "Autonomous navigation of stratospheric balloons using reinforcement learning," *Nature*, vol. 588, no. 7836, pp. 77–82, Dec. 1, 2020, ISSN: 1476-4687. DOI: 10.1038/s41586-020-2939-8. [Online]. Available: https://doi.org/10.1038/s41586-020-2939-8.

[6] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Communications of the ACM*, vol. 18, no. 9, pp. 509–517, 1975.

[7] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization," *Journal of Machine Learning Research*, vol. 13, no. 10, pp. 281–305, 2012. [Online]. Available: http://jmlr.org/papers/v13/bergstra12a.html.

[8] G. Brockman *et al.*, *Openai gym*, 2016. eprint: arXiv:1606.01540.

[9] R. A. Brown, "Building a balanced kd tree in o (kn log n) time," *arXiv preprint arXiv:1410.5420*, 2014.

[10] S. Chiappa, S. Racaniere, D. Wierstra, and S. Mohamed, *Recurrent environment simulators*, 2017. arXiv: 1704.02254 [cs.AI].

[11] K. Cho *et al.*, "Learning phrase representations using RNN encoder–decoder for statistical machine translation," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, A. Moschitti, B. Pang, and W. Daelemans, Eds., Doha, Qatar: Association for Computational Linguistics, Oct. 2014, pp. 1724–1734. DOI: 10.3115/v1/D14-1179. [Online]. Available: https://aclanthology.org/D14-1179.

[12] K. Chua, R. Calandra, R. McAllister, and S. Levine, "Deep reinforcement learning in a handful of trials using probabilistic dynamics models," *Advances in neural information processing systems*, vol. 31, 2018.

[13] M. P. Deisenroth and C. E. Rasmussen, "Pilco: A model-based and data-efficient approach to policy search," in *Proceedings of the 28th International Conference on International Conference on Machine Learning*, ser. ICML'11, Bellevue, Washington, USA: Omnipress, 2011, pp. 465–472, ISBN: 9781450306195.

[14] M. P. Deisenroth, C. E. Rasmussen, and D. Fox, "Learning to control a low-cost manipulator using data-efficient reinforcement learning," 2012.

[15] S. Depeweg, J.-M. Hernandez-Lobato, F. Doshi-Velez, and S. Udluft, "Decomposition of uncertainty in bayesian deep learning for efficient and risk-sensitive learning," in *International conference on machine learning*, PMLR, 2018, pp. 1184–1193.

[16] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[17] T. G. Dietterich, "Ensemble methods in machine learning," in *International workshop on multiple classifier systems*, Springer, 2000, pp. 1–15.

[18] G. Dulac-Arnold *et al.*, "Challenges of real-world reinforcement learning: Definitions, benchmarks and analysis," *Machine Learning*, vol. 110, no. 9, pp. 2419–2468, 2021.

[19] L. Engstrom *et al.*, "Implementation matters in deep rl: A case study on ppo and trpo," in *International Conference on Learning Representations*, 2020. [Online]. Available: https://openreview.net/forum?id=r1etN1rtPB.

[20] C. Finn, X. Y. Tan, Y. Duan, T. Darrell, S. Levine, and P. Abbeel, "Deep spatial autoencoders for visuomotor learning," in *2016 IEEE International Conference on Robotics and Automation (ICRA)*, IEEE, 2016, pp. 512–519.

[21] J. Fu, A. Kumar, O. Nachum, G. Tucker, and S. Levine, *D4rl: Datasets for deep data-driven reinforcement learning*, 2021. arXiv: 2004.07219 [cs.LG].

[22] S. Fujimoto, D. Meger, and D. Precup, *Off-policy deep reinforcement learning without exploration*, 2019. arXiv: 1812.02900 [cs.LG].

[23] T. Giorgino, "Computing and visualizing dynamic time warping alignments in r: The dtw package," *Journal of statistical Software*, vol. 31, pp. 1–24, 2009.

[24] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor," in *Proceedings of the 35th International Conference on Machine Learning*, J. Dy and A. Krause, Eds., ser. Proceedings of Machine Learning Research, vol. 80, PMLR, 2018, pp. 1861–1870. [Online]. Available: https://proceedings.mlr.press/v80/haarnoja18b.html.

[25] D. Hafner, T. Lillicrap, J. Ba, and M. Norouzi, *Dream to control: Learning behaviors by latent imagination*, 2020. arXiv: 1912.01603 [cs.LG].

[26] D. Hafner, T. Lillicrap, M. Norouzi, and J. Ba, "Mastering atari with discrete world models," *arXiv preprint arXiv:2010.02193*, 2020.

[27] D. Hafner, J. Pasukonis, J. Ba, and T. Lillicrap, *Mastering diverse domains through world models*, 2023. arXiv: 2301.04104 [cs.AI].

[28] P. Henderson, R. Islam, P. Bachman, J. Pineau, D. Precup, and D. Meger, "Deep reinforcement learning that matters," in *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence and Thirtieth Innovative Applications of Artificial Intelligence Conference and Eighth AAAI Symposium on Educational Advances in Artificial Intelligence*, ser. AAAI'18/IAAI'18/EAAI'18, New Orleans, Louisiana, USA: AAAI Press, 2018, ISBN: 978-1-57735-800-8.

[29] M. Hessel *et al.*, *Rainbow: Combining improvements in deep reinforcement learning*, 2017. arXiv: 1710.02298 [cs.AI].

[30] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, 1997, ISSN: 0899-7667. DOI: 10.1162/neco.1997.9.8.1735. [Online]. Available: https://doi.org/10.1162/neco.1997.9.8.1735.

[31] G. Z. Holland, E. J. Talvitie, and M. Bowling, *The effect of planning shape on dyna-style planning in high-dimensional state spaces*, 2019. arXiv: 1806.01825 [cs.AI].

[32] P. Indyk and R. Motwani, "Approximate nearest neighbors: Towards removing the curse of dimensionality," in *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, 1998, pp. 604–613.

[33] A. Jacobsen, M. Schlegel, C. Linke, T. Degris, A. White, and M. White, *Meta-descent for online, continual prediction*, 2019. arXiv: 1907.07751 [cs.LG].

[34] M. Jaderberg *et al.*, *Population based training of neural networks*, 2017. arXiv: 1711.09846 [cs.LG].

[35] M. Janjua, H. Shah, M. White, E. Miahi, M. Machado, and A. White, "Gvfs in the real world: Making predictions online for water treatment," *Machine Learning*, pp. 1–31, Nov. 2023. DOI: 10.1007/s10994-023-06413-x.

[36] M. Janner, J. Fu, M. Zhang, and S. Levine, "When to trust your model: Model-based policy optimization," *Advances in neural information processing systems*, vol. 32, 2019.

[37] L. Kaiser *et al.*, "Model-based reinforcement learning for atari," *arXiv preprint arXiv:1903.00374*, 2019.

[38] E. Kaufmann, L. Bauersfeld, A. Loquercio, M. Müller, V. Koltun, and D. Scaramuzza, "Champion-level drone racing using deep reinforcement learning," *Nature*, vol. 620, no. 7976, pp. 982–987, Aug. 1, 2023, ISSN: 1476-4687. DOI: 10.1038/s41586-023-06419-4. [Online]. Available: https://doi.org/10.1038/s41586-023-06419-4.

[39] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[40] J. Ko, D. J. Klein, D. Fox, and D. Haehnel, "Gaussian processes and reinforcement learning for identification and control of an autonomous blimp," in *Proceedings 2007 ieee international conference on robotics and automation*, IEEE, 2007, pp. 742–747.

[41] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, vol. 25, 2012.

[42] J. Kruskal and M. Liberman, "The symmetric time-warping problem: From continuous to discrete," *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*, Jan. 1983.

[43] A. Kumar, A. Zhou, G. Tucker, and S. Levine, *Conservative q-learning for offline reinforcement learning*, 2020. arXiv: 2006.04779 [cs.LG].

[44] T. Kurutach, I. Clavera, Y. Duan, A. Tamar, and P. Abbeel, "Model-ensemble trust-region policy optimization," *arXiv preprint arXiv:1802.10592*, 2018.

[45] N. Lambert, K. Pister, and R. Calandra, "Investigating compounding prediction errors in learned dynamics models," *arXiv preprint arXiv:2203.09637*, 2022.

[46] S. Levine, C. Finn, T. Darrell, and P. Abbeel, "End-to-end training of deep visuomotor policies," *Journal of Machine Learning Research*, vol. 17, no. 39, pp. 1–40, 2016.

[47] S. Levine, C. Finn, T. Darrell, and P. Abbeel, "End-to-end training of deep visuomotor policies," *Journal of Machine Learning Research*, vol. 17, no. 39, pp. 1–40, 2016.

[48] J. Luo *et al.*, *Controlling commercial cooling systems using reinforcement learning*, 2022. arXiv: 2211.07357 [cs.LG].

[49] M. C. Machado, M. G. Bellemare, and M. Bowling, "A laplacian framework for option discovery in reinforcement learning," in *International Conference on Machine Learning*, PMLR, 2017, pp. 2295–2304.

[50] S. Mahadevan and M. Maggioni, "Proto-value functions: A laplacian framework for learning representation and control in markov decision processes.," *Journal of Machine Learning Research*, vol. 8, no. 10, 2007.

[51] T. Mandel, Y.-E. Liu, E. Brunskill, and Z. Popović, "Offline evaluation of online reinforcement learning algorithms," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 30, 2016.

[52] G. Manek and J. Z. Kolter, *Learning stable deep dynamics models*, 2020. arXiv: 2001.06116 [cs.LG].

[53] J. Matas, S. James, and A. J. Davison, "Sim-to-real reinforcement learning for deformable object manipulation," in *Proceedings of The 2nd Conference on Robot Learning*, A. Billard, A. Dragan, J. Peters, and J. Morimoto, Eds., ser. Proceedings of Machine Learning Research, vol. 87, PMLR, 29–31 Oct 2018, pp. 734–743. [Online]. Available: https://proceedings.mlr.press/v87/matas18a.html.

[54] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, *et al.*, *Playing atari with deep reinforcement learning*, 2013. arXiv: 1312.5602 [cs.LG].

[55] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, Feb. 1, 2015, ISSN: 1476-4687. DOI: 10.1038/nature14236. [Online]. Available: https://doi.org/10.1038/nature14236.

[56] J. Modayil, A. White, and R. S. Sutton, "Multi-timescale nexting in a reinforcement learning robot," in *From Animals to Animats 12*, T. Ziemke, C. Balkenius, and J. Hallam, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 299–309.

[57] A. Nagabandi, G. Kahn, R. S. Fearing, and S. Levine, "Neural network dynamics for model-based deep reinforcement learning with model-free fine-tuning," in *2018 IEEE international conference on robotics and automation (ICRA)*, IEEE, 2018, pp. 7559–7566.

[58] D. Nguyen-Tuong, M. Seeger, and J. Peters, "Model learning with local gaussian process regression," *Advanced Robotics*, vol. 23, no. 15, pp. 2015–2034, 2009.

[59] J. Oh, X. Guo, H. Lee, R. L. Lewis, and S. Singh, "Action-conditional video prediction using deep networks in atari games," *Advances in neural information processing systems*, vol. 28, 2015.

[60] T. L. Paine *et al.*, *Hyperparameter selection for offline reinforcement learning*, 2020. arXiv: 2007.09055 [cs.LG].

[61] S. Paul, V. Kurin, and S. Whiteson, "Fast efficient hyperparameter tuning for policy gradient methods," *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[62] L. Rabiner and B.-H. Juang, *Fundamentals of speech recognition*. USA: Prentice-Hall, Inc., 1993, ISBN: 0130151572.

[63] A. A. Rusu, M. Vecerik, T. Rothörl, N. Heess, R. Pascanu, and R. Hadsell, *Sim-to-real robot learning from pixels with progressive nets*, 2018. arXiv: 1610.04286 [cs.RO].

[64] H. Sakoe and S. Chiba, "Dynamic programming algorithm optimization for spoken word recognition," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 26, no. 1, pp. 43–49, 1978. DOI: 10.1109/TASSP.1978.1163055.

[65] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, *Proximal policy optimization algorithms*, 2017. arXiv: 1707.06347 [cs.LG].

[66] K. Schweighofer *et al.*, "Understanding the effects of dataset characteristics on offline reinforcement learning," in *Deep RL Workshop NeurIPS 2021*, 2021.

[67] D. Silver *et al.*, "Mastering the game of Go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, Jan. 1, 2016, ISSN: 1476-4687. DOI: 10.1038/nature16961. [Online]. Available: https://doi.org/10.1038/nature16961.

[68] J. Snoek, H. Larochelle, and R. P. Adams, *Practical bayesian optimization of machine learning algorithms*, 2012. arXiv: 1206.2944 [stat.ML].

[69] R. S. Sutton, "Learning to predict by the methods of temporal differences," *Machine learning*, vol. 3, pp. 9–44, 1988.

[70] R. S. Sutton, "Generalization in reinforcement learning: Successful examples using sparse coarse coding," in *Advances in Neural Information Processing Systems*, D. Touretzky, M. Mozer, and M. Hasselmo, Eds., vol. 8, MIT Press, 1995. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/1995/file/8f1d43620bc6bb580df6e80b0dc05c48-Paper.pdf.

[71] R. S. Sutton, "The Grand Challenge of Predictive Empirical Abstract Knowledge," 2009.

[72] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[73] R. S. Sutton, J. Modayil, *et al.*, "Horde: A scalable real-time architecture for learning knowledge from unsupervised sensorimotor interaction," in *The 10th International Conference on Autonomous Agents and Multiagent Systems-Volume 2*, 2011, pp. 761–768.

[74] E. Talvitie, "Model regularization for stable sample rollouts.," in *UAI*, 2014, pp. 780–789.

[75] E. Talvitie, "Self-correcting models for model-based reinforcement learning," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 31, 2017.

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

[77] A. Venkatraman, M. Hebert, and J. Bagnell, "Improving multi-step prediction of learned time series models," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 29, no. 1, Feb. 2015. DOI: 10.1609/aaai.v29i1.9590. [Online]. Available: https://ojs.aaai.org/index.php/AAAI/article/view/9590.

[78] O. Vinyals *et al.*, "Grandmaster level in StarCraft II using multi-agent reinforcement learning," *Nature*, vol. 575, no. 7782, pp. 350–354, Nov. 1, 2019, ISSN: 1476-4687. DOI: 10.1038/s41586-019-1724-z. [Online]. Available: https://doi.org/10.1038/s41586-019-1724-z.

[79] H. Wang, E. Miahi, *et al.*, "Investigating the properties of neural network representations in reinforcement learning," *Artificial Intelligence*, vol. 330, p. 104 100, 2024, ISSN: 0004-3702. DOI: 10.1016/j.artint.2024.104100. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0004370224000365.

[80] H. Wang, A. Sakhadeo, *et al.*, *No more pesky hyperparameters: Offline hyperparameter tuning for rl*, 2022. arXiv: 2205.08716 [cs.LG].

[81] K. Wang, K. Zhou, J. Feng, B. Hooi, and X. Wang, *Reachability-aware laplacian representation in reinforcement learning*, 2022. arXiv: 2210.13153 [cs.LG].

[82] K. Wang, K. Zhou, Q. Zhang, J. Shao, B. Hooi, and J. Feng, "Towards better laplacian representation in reinforcement learning with generalized graph drawing," in *International Conference on Machine Learning*, PMLR, 2021, pp. 11 003–11 012.

[83] P. Werbos, "Backpropagation through time: What it does and how to do it," *Proceedings of the IEEE*, vol. 78, no. 10, pp. 1550–1560, 1990. DOI: 10.1109/5.58337.

[84] M. White and A. White, "A greedy approach to adapting the trace parameter for temporal difference learning," *arXiv preprint arXiv:1607.00446*, 2016.

[85] D. R. Wilson and T. R. Martinez, "Reduction techniques for instance-based learning algorithms," *Machine learning*, vol. 38, pp. 257–286, 2000.

[86] Y. Wu, G. Tucker, and O. Nachum, "The laplacian in rl: Learning representations with efficient approximations," *arXiv preprint arXiv:1810.04586*, 2018.

[87] Y. Wu, G. Tucker, and O. Nachum, *Behavior regularized offline reinforcement learning*, 2019. arXiv: 1911.11361 [cs.LG].

[88] T. Yu *et al.*, "MOPO: Model-based offline policy optimization," in *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., vol. 33, Curran Associates, Inc., 2020, pp. 14 129–14 142. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2020/file/a322852ce0df73e204b7e67cbbef0d0a-Paper.pdf.

[89] T. Zahavy *et al.*, "A self-tuning actor-critic algorithm," *Advances in neural information processing systems*, vol. 33, pp. 20 913–20 924, 2020.

[90] X. Zhan, H. Xu, Y. Zhang, X. Zhu, H. Yin, and Y. Zheng, *Deepthermal: Combustion optimization for thermal power generating units using offline reinforcement learning*, 2022. arXiv: 2102.11492 [cs.LG].

[91] Z. Zhu, K. Lin, A. K. Jain, and J. Zhou, "Transfer learning in deep reinforcement learning: A survey," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2023.

Appendix A: Appendix

A.1 Hyperparameters

Hyperparameters used for data collection, agent training, and model training in the fixed-horizon acrobot and continuous gridworld experiments in chapter 5 and water treatment experiments in Chapters 6 and 7. Hyperparameters were typically tuned specifically for each environment, hence the differences in values across environments.

Hyperparameter	**Symbol**	**Acrobot**
Optimizer	-	Adam
Learning Rate	α	1e−3
Discount Factor	γ	0.99
Batch Size	B	256
Hidden Layers	-	2
Hidden Units	-	256
Replay Buffer Size	-	100,000
Target Network Update Frequency	-	1
Training Frequency	-	1
Exploration Rate	ϵ	0.1

Table A.1: DQN hyperparameters used for data collection and evaluation in the fixed-horizon acrobot environment.

Hyperparameter	Symbol	GridWorld
Optimizer	-	Adam
Learning Rate	-	1e−3
Discount Factor	γ	0.99
Batch Size	B	256
Hidden Layers	-	2
Hidden Units	-	64
Replay Buffer Size	-	100,000
Target Network Update Frequency	-	1
Training Frequency	-	1
Entropy Temperature	α	0.25

Table A.2: SAC hyperparameters used for data collection and evaluation rollouts in the Continuous GridWorld environment. The same network architecture is used for both actor and critic networks.

Hyperparameter	Symbol	Water 1-Week (Online)	Water 12-Month (Pre-training)	Water 12-Month (Online)
Optimizer	-	Adam	Adam	Adam
Learning Rate	α	-	1e−5	-
Discount Factor	γ	0.99	0.9	0.9
Batch Size	B	256	256	256
Hidden Layers	-	2	2	2
Hidden Units	-	256	512	512
Replay Buffer Size	-	1M	3M	1M
Train/Validation Split	-	-	0.9/0.1	-
Epochs	-	-	1000	-

Table A.3: Hyperparameters used for the TD(0) prediction agent in water treatment experiments.

Hyperparameter	Symbol	Acrobot	GridWorld	Water 1-Week
Optimizer	-	Adam	Adam	Adam
Learning Rate	α	1e−3	1e−3	1e−3
Batch Size	B	256	256	256
Hidden Layers	-	2	2	2
State Model Hidden Size	-	64	128	512
Reward Model Hidden Size	-	32	32	-
Termination Model Hidden Size	-	32	32	-
Epochs	-	500	500	100

Table A.4: NN calibration model training hyperparameters. The same number of training epochs are used across state, reward, and termination models.

Hyperparameter	Symbol	Acrobot	Water 1-Week
Optimizer	-	Adam	Adam
Learning Rate	α	1e−3	1e−3
Batch Size	B	256	256
Hidden Layers	-	2	2
State Model Hidden Size	-	64	512
Reward Model Hidden Size	-	32	-
Termination Model Hidden Size	-	16	-
Burn-in Length	-	5	0
Sequence Length	-	10	20
State Model Epochs	-	400	100
Reward Model Epochs	-	50	-
Termination Model Epochs	-	50	-

Table A.5: GRU calibration model training hyperparameters. Different numbers of training epochs are used for state, reward, and termination models.

Hyperparameter	Symbol	Acrobot	GridWorld	Water 1-Week	Water 12-Month
Optimizer	-	Adam	Adam	Adam	Adam
Learning Rate	α	1e−3	1e−3	3e−4	1e−5
Batch Size	B	256	256	256	256
Hidden Layers	-	2	2	2	2
Hidden Units	-	64	32	256	256
Output Dimension	-	4	4	64	64
Training Steps	-	100,000	100,000	100,000	200,000
Train/Validation Split	-	0.9/0.1	0.9/0.1	0.8/0.2	0.8/0.2
Sequence Length	-	20	5	20	20
Kappa	κ	0.99	0.9	0.95	0.95
Beta	β	5	1.0	5	5
Zeta	ζ	0.05	0.01	0.05	0.05

Table A.6: Laplacian representation training hyperparameters.

A.2 Agent Pre-training

Training and validation curves for hyper sweeps of agent pre-training on the one-year WTP dataset.

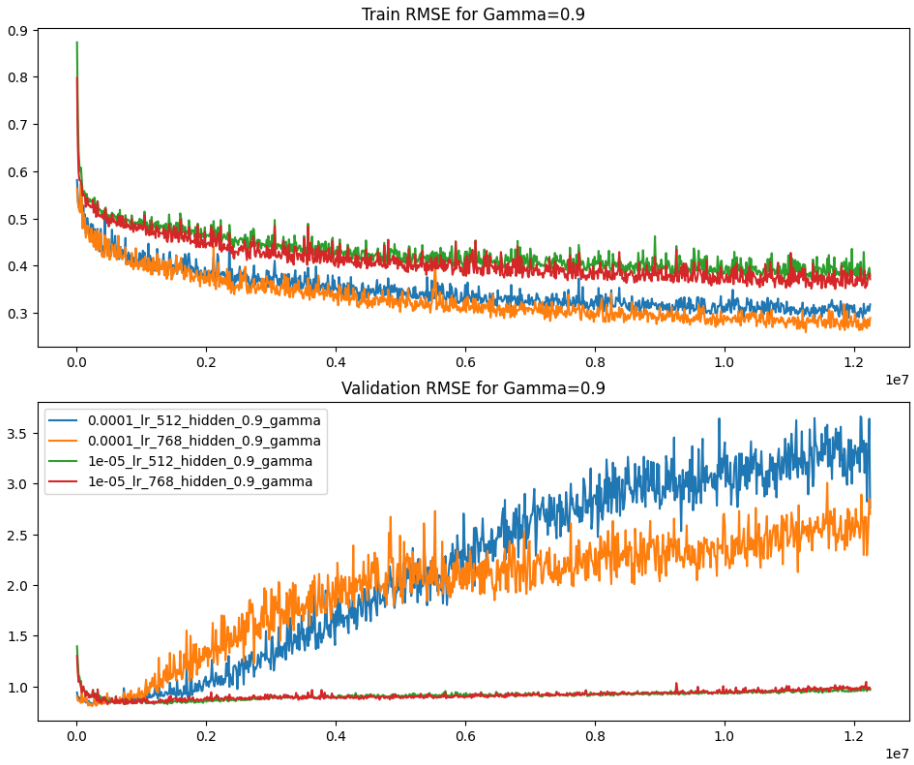


Figure A.1: Hyper sweep over learning rate and network size for TD(0) replay prediction agent on one-year WTP dataset, with $\gamma = 0.9$.

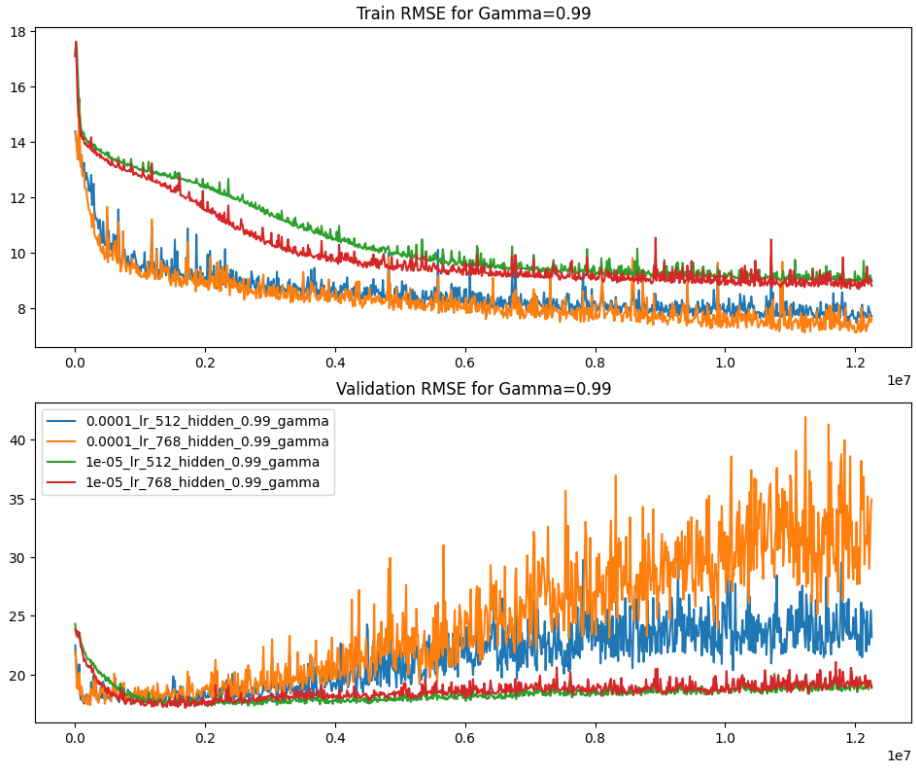


Figure A.2: Hyper sweep over learning rate and network size for TD(0) replay prediction agent on one-year WTP dataset, with $\gamma = 0.99$.

A.3 WTP Rollouts

Rollouts for the 12-month and 1-week WTP KNN calibration models using targeted start states from the April 2023 and July 2023 datasets. In Section 7.5, only rollouts for the May 2023 dataset were shown.

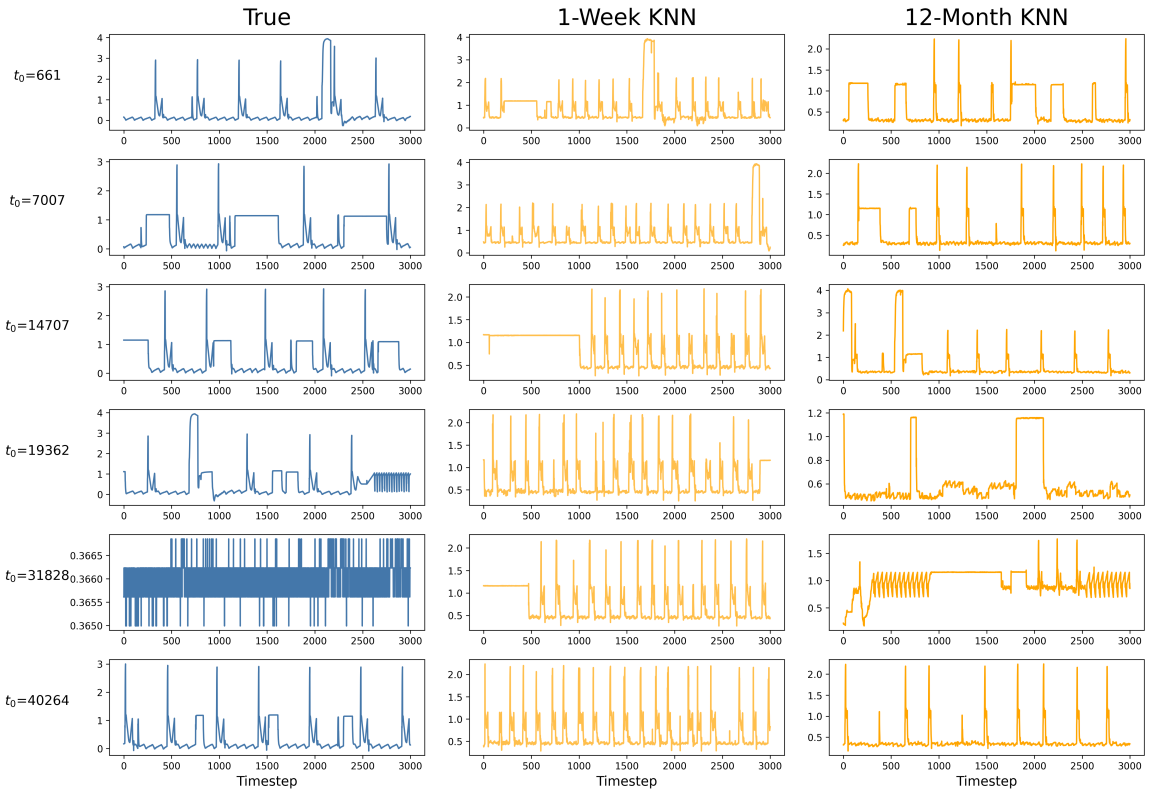


Figure A.3: PIT300 rollouts for the 12-month and 1-week WTP KNN calibration models using targeted start states from the April 2023 dataset.

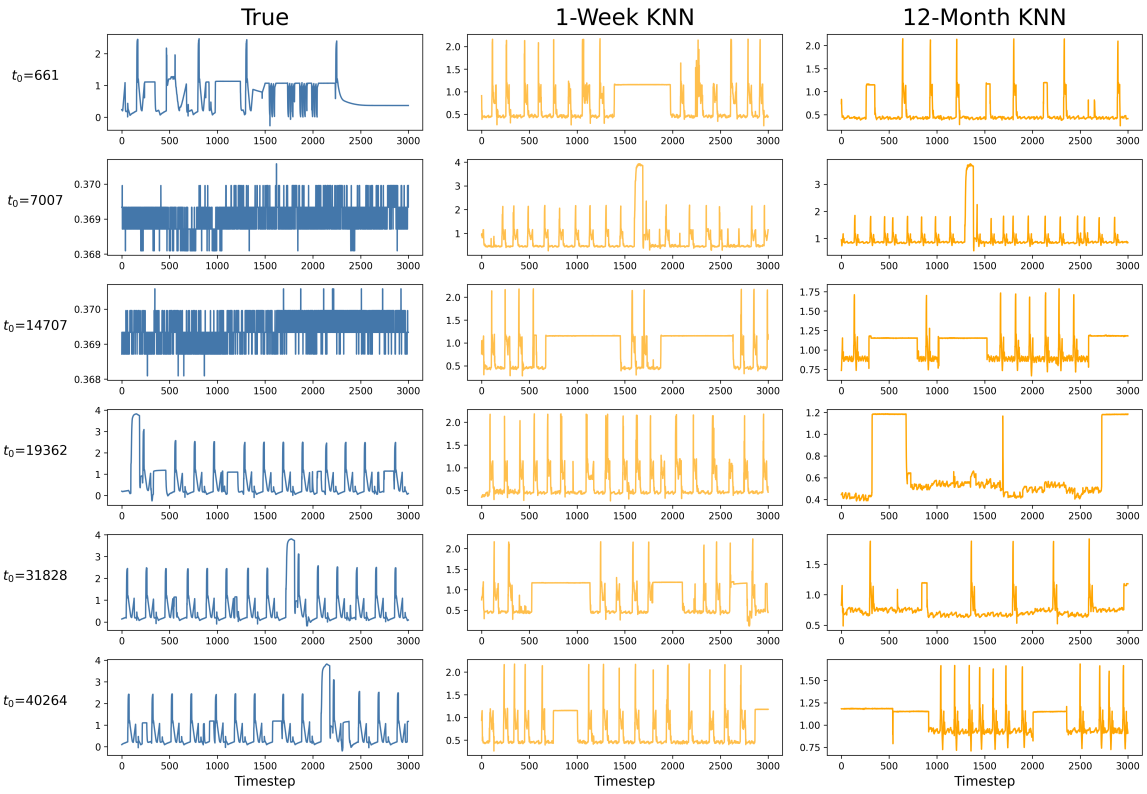


Figure A.4: PIT300 rollouts for the 12-month and 1-week WTP KNN calibration models using targeted start states from the July 2023 dataset.

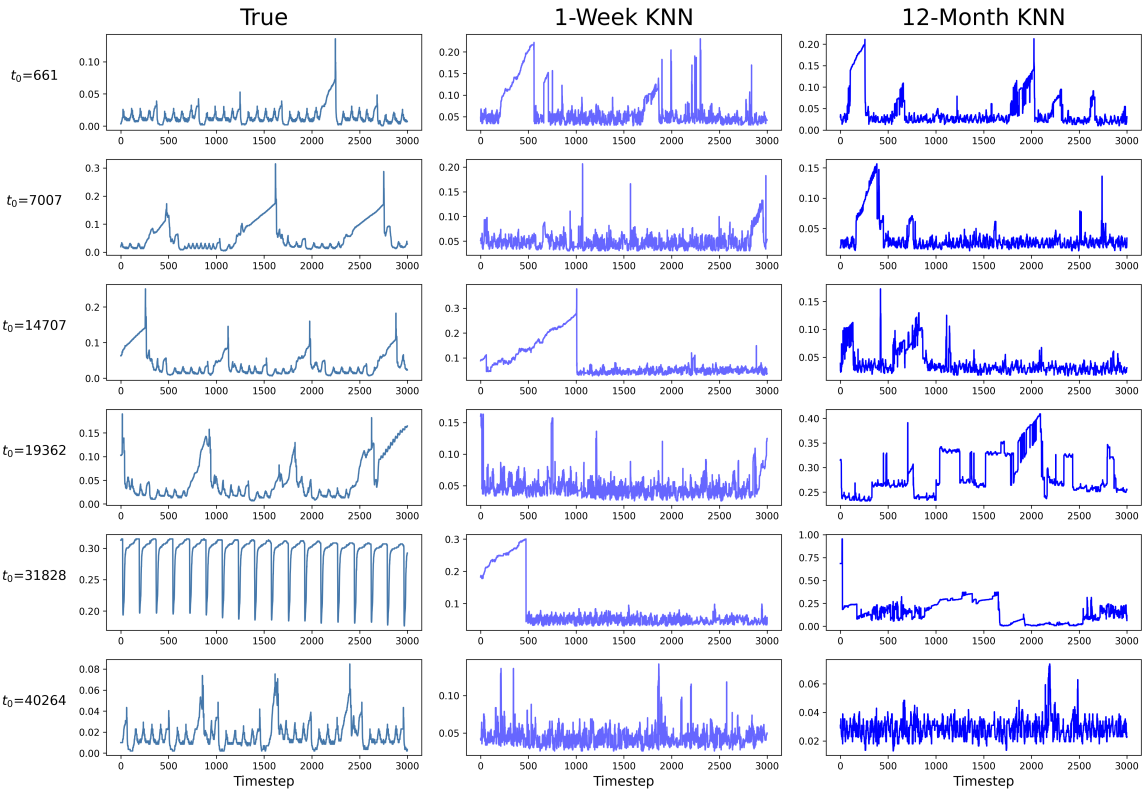


Figure A.5: TIT101 rollouts for the 12-month and 1-week WTP KNN calibration models using targeted start states from the April 2023 dataset.

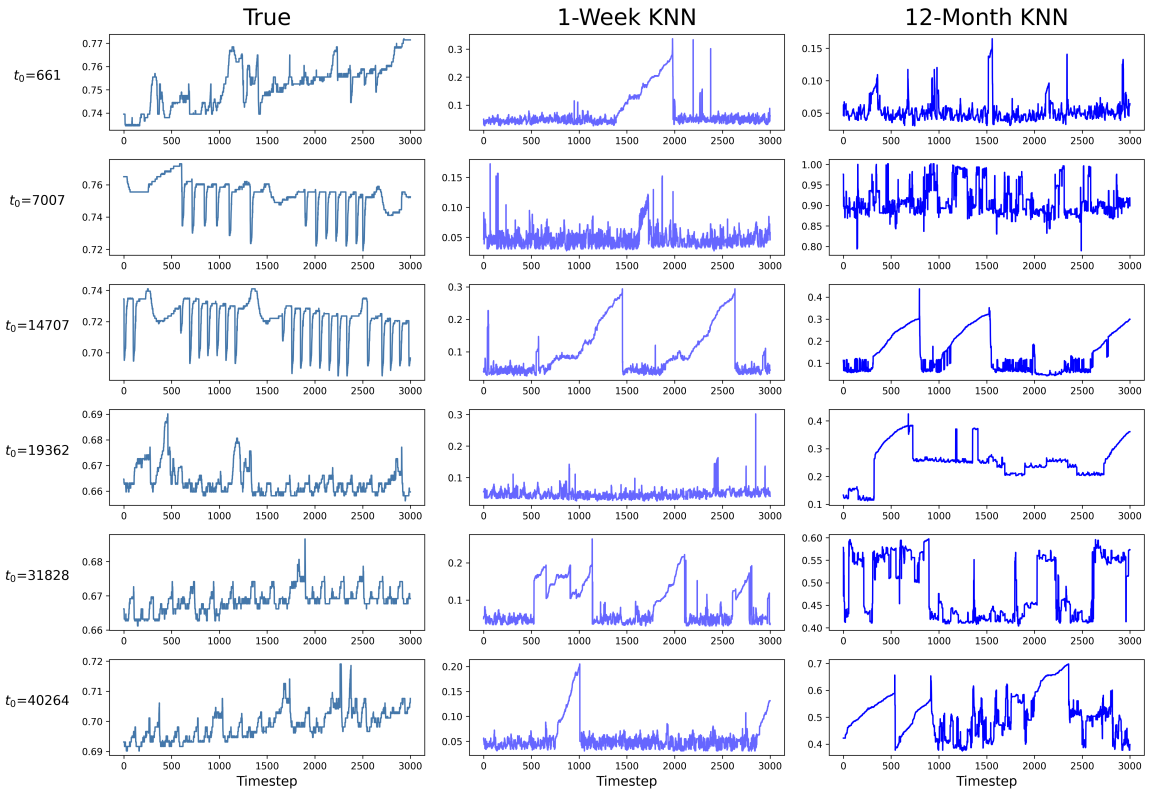


Figure A.6: TIT101 rollouts for the 12-month and 1-week WTP KNN calibration models using targeted start states from the July 2023 dataset.

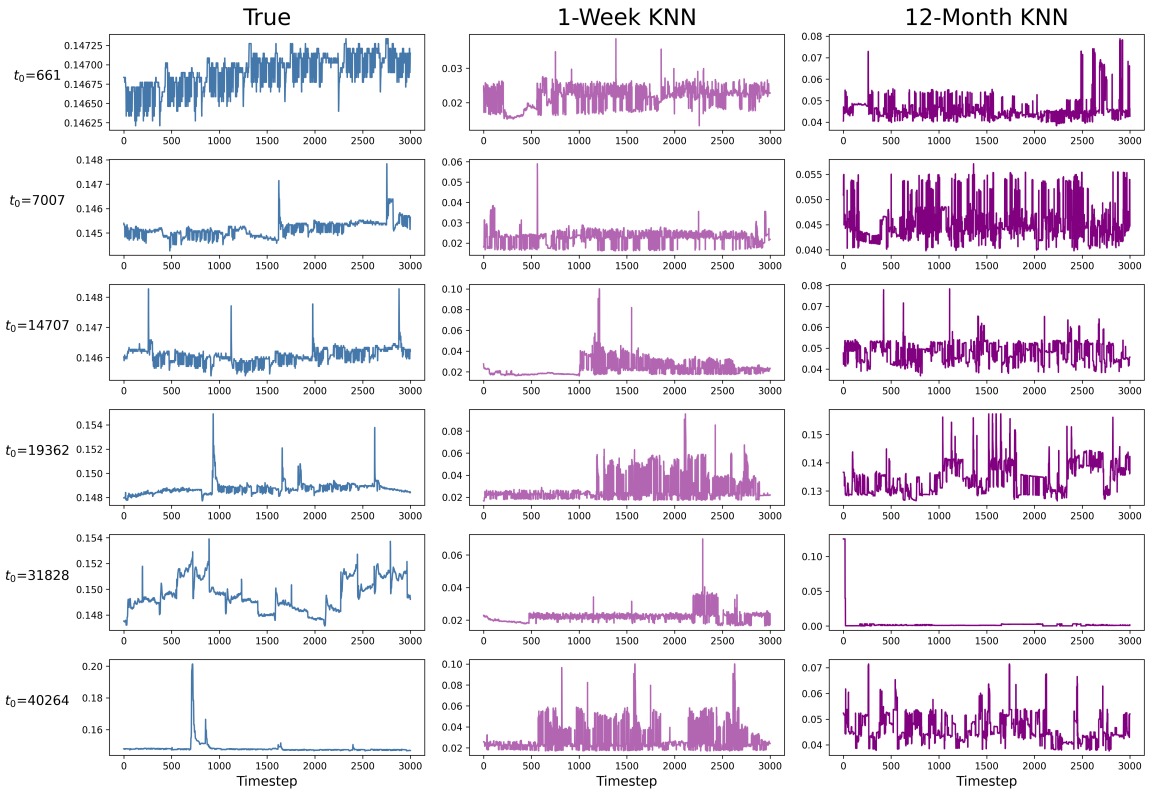


Figure A.7: TUIT101 rollouts for the 12-month and 1-week WTP KNN calibration models using targeted start states from the April 2023 dataset.

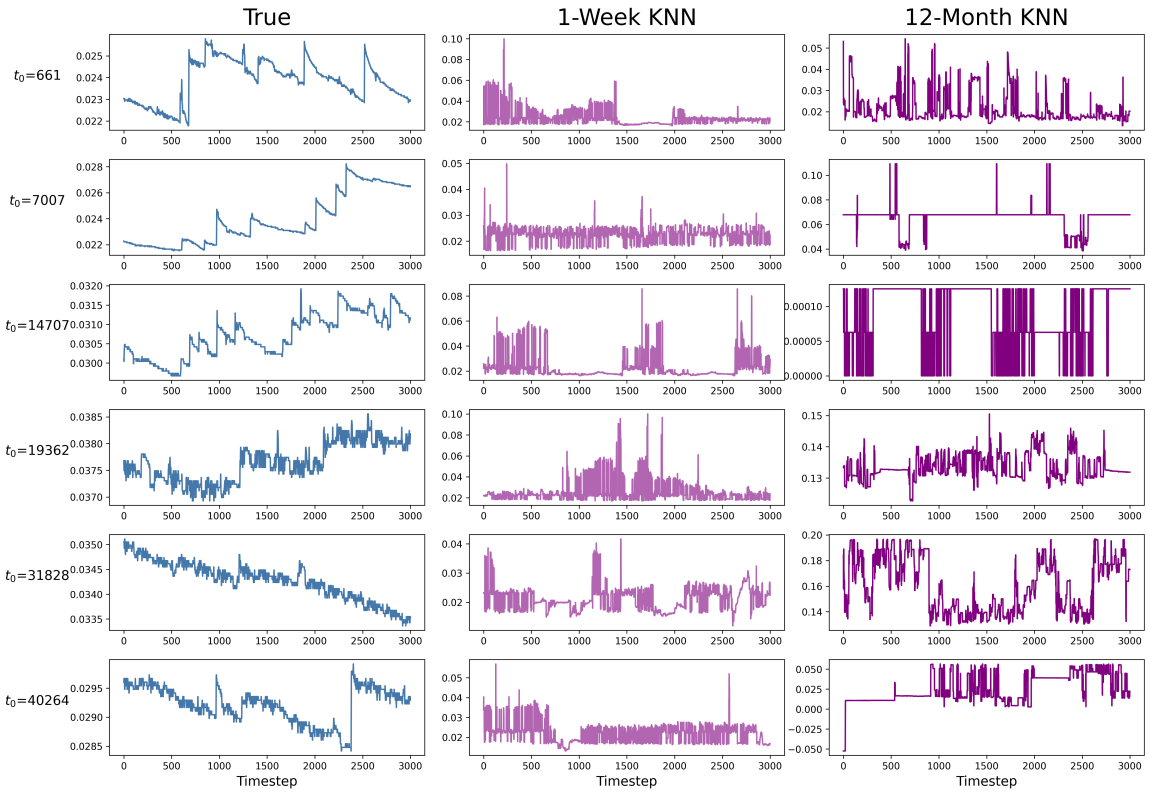


Figure A.8: TUIT101 rollouts for the 12-month and 1-week WTP KNN calibration models using targeted start states from the July 2023 dataset.