Model-Based Bayesian Sparse Sampling for Data Efficient Control

by

Timmy Rong Tian Tse

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2019

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

In this work, we propose a novel Bayesian-inspired model-based policy search algorithm for data efficient control. In contrast to other model-based approaches, our algorithm makes use of approximate Gaussian processes in the form of random Fourier features for fast online systems identification and computationally efficient posterior updates via rank one Cholesky updates. Furthermore, fast and tractable posterior updates permits policy optimization to leverage knowledge from posterior evolution tracking for a directed Bayesian approach to the exploration-exploitation dilemma. To address the optimization formulation involving belief monitoring as well as the potentiality of a loss surface with zero gradients everywhere, we leverage a blackbox optimizer in the form of covariance matrix adaptation evolution strategy (CMA-ES). We test our algorithm on four challenging control tasks and report the superior data efficiency as well as the exploration capabilities of our model.

Acknowledgements

First, I would like to thank my supervisor, Prof. Pascal Poupart and my co-supervisor, Prof. Edith Law, both for their support, patience and guidance they have provided me over the past years. In particular, I am grateful to Edith for accepting me as a student and giving me the invaluable chance to find my vocation. Also, many thanks to Pascal for taking me under his wing when, unacquainted at the time, I burst into his office proclaiming that we ought to work together!

I would also like to thank my committee members, Prof. Yaoliang Yu and Prof. Peter van Beek for taking the time to read, edit and analyze my thesis. Their feedback and suggestions will provide me with new perspectives for future work.

I would like to thank my colleagues and friends for making my time at the university fun and enjoyable.

Finally and of course the most, I thank my family for their love, support and patience.

Dedication

To my *mom*, *dad* and *sis*.

Table of Contents

List of Figures

Chapter 1

Introduction

One eventual goal of Artificial Intelligence is to get agents to autonomously make sequential decisions in an environment to realize a given task. A mathematical formulation between the interaction of an agent and its environment known as Reinforcement Learning (RL) provides a rich and attractive framework to approach this problem. The beginnings of Reinforcement Learning date back to the ideas of Bellman in the 1950s, but there has been a resurgence of interest in these ideas in recent times due to the recent success of Deep Learning, allowing a re-visitation of old ideas with new tools and more powerful hardware.

Despite the recent success, however, there remains problems in RL and optimal control for which a solution that is theoretically satisfying and computationally feasible would be desirable. One such problem, for example, concerns the sample inefficiency of current algorithms wherein these algorithms require an exorbitant number of interactions with the environment before learning a successful policy, rendering them impractical for real-world applications (e.g., mechanical systems such as robots that are prone to wear and tear).

In this thesis, we address the problem of sample efficiency along with the (related) problem of the so-called exploration exploitation dilemma (related in that improving the latter will also improve the former). This term describes the dilemma faced by an agent interacting with an environment wherein the agent must balance the selection of actions that either 1) explore by revealing reachable states that were previously unknown or 2) exploit by using its current knowledge of the environment to maximize the collected rewards. With too much exploration the agent will collect a suboptimal amount of rewards and with too much exploitation the agent may never reach a (near) optimal strategy; hence the dilemma and the need to aptly balance exploration and exploitation.

An elegant and theoretically justified approach in dealing with the exploration-exploitation

dilemma involves a technique known as model-based *Bayesian Reinforcement Learning* (BRL). In this formulation, we model both the transition and reward function with a probabilistic model and update the model upon new evidence based on Bayes' rule. Learning an optimal policy then involves unrolling the model into the future and searching for a set of actions that maximize the cumulative rewards of the trajectory. The presence of a distribution over world models allows the agent to "know what it does and does not know" rendering it capable of selecting actions that naturally balance exploration-exploitation by considering the statistical averages of all possible world models.

Historically, much of the work on BRL involved experiments that were limited to toy grid-world like problems which had discrete states and actions. In this thesis, we extend the BRL formulation to tasks that resemble more real-world control problems, that is, problems that involve continuous states and actions. We first introduce an approximate Gaussian process (GP) regression model in the form of Bayesian linear regression (BLR) with random Fourier features (RFFs) for learning the system and reward dynamics of an environment. This formulation allows us to approximately leverage the regression powers of Bayesian nonparametrics while retaining the benefits of fast online learning due to the fact that covarianace matrices possess constant space complexity in primal space. The value of a policy is given by the expected cumulative discounted rewards obtained via the unrolled trajectory following said policy with respect to the learned reward and system dynamics model. In accordance to the BRL formulation, we maintain an information state corresponding to the covariance and cross-covariance matrices of the system and reward dynamics model and during policy optimization, we track the posterior of the information state for all hypothetical experience tuples during trajectory rolllouts, thereby allowing the controller to optimize for a policy that balances exploration-exploitation via the Bayes-optimal formulation. Note that the repeated computation of the posterior during optimization further motivates a primal space formulation of the regression model. Finally, our optimization objective calls for a black-box optimizer and to this end, we make use of an algorithm call covariance matrix adaptation evolution strategy (CMA-ES).

1.1 Contributions

The main contribution of this work is that we extend the formulation of model-based BRL with its principled approach to the exploration-exploitation dilemma under the Bayes-optimality to Markov decision processes (MDPs) with continuous states and actions. We draw inspiration from pioneering ideas and leverage modern software tools to explore an algorithm that is not only data-efficient and theoretically-sound, but also scales to problem

domains beyond just grid-world. Specially, we contribute:

- A novel probabilistic non-linear regression method that allows for fast online learning as well as an efficient posterior updates via rank-one Cholesky updates. The regression method approximates a GP but maintains a form that is fast enough for posterior tracking during policy optimization in finding a policy that addresses the exploration-exploitation dilemma under the BRL formulation.
- An approximation to the loss function that reflects the Bayes-optimality condition under a continuous state and continuous action MDP. In addition, we propose a Monte Carlo sampling procedure in conjunction with a black-box optimizer to effectively solve for a policy.
- A set of experiments that demonstrate the data-efficiency of our model-based method compared to a model-free baseline as well as the ability of our model to perform few-shot learning on a set of classical control tasks. As well, we test our algorithm on more complex tasks and observe how well our model-based algorithm scales to problems with high-dimensional state and/or action spaces.

1.2 Thesis Organization

The remainder of the thesis is organized as follows:

- Chapter two presents the background on MDPs as well as algorithms for model-free RL including REINFORCE, Q-learning, SARSA and actor-critic methods and model-based reinforcement learning such as the Dyna architecture and the model predictive control (MPC) algorithm. In addition, we review GP regression as well as the CMA-ES algorithm as they are major components of this work.
- Chapter three describes the proposed algorithm which is the major focus of this thesis. We elucidate the two major components of the work, that is, regression of the system and reward dynamics and policy search. In addition, we discuss the design decisions of the algorithm including approximations that make the algorithm computationally tractable and addenda that allow the method to reflect the theory posited by BRL.
- Chapter four describes the set of experiments which were conducted to test the performance and data-efficiency of our method. The first set of experiments compares

our method to a model-free baseline on a set of toy control tasks, the second set of experiments scales the former set of experiments to higher dimensional control tasks and finally, the last set of experiments compares our method to the PILCO and deepPILCO algorithms [10, 27].

- Chapter five concludes the thesis by summarizing the contributions and outlining directions for future work.

Chapter 2

Background and Related Work

2.1 Markov Decision Processes

A Markov Decision Process (MDP) is a mathematical model that formalizes planning and decision making in uncertain environments. A MDP is defined by the tuple (S, A, P, R, γ), where

- $s \in S$ is the finite set of states;
- $a \in A$ is the finite set of actions;
- $P = Pr(s'|s, a)$ is the probability of transitioning to state s' when taking action a in state s;
- $R = Pr(r|s, a)$ is the probability of obtaining reward r when taking action a in state s;
- $\gamma \in [0, 1]$ is the discount factor, which is a scalar that represents the amount in which immediate rewards are valued in comparison to future rewards.

The events of a MDP proceed as follows: in the beginning, the current state of the world is set to some initial state s_0 drawn from an initial state distribution P_0. Upon the agent executing action a, the state of the world is updated to the next state s_1 obtained by sampling from the distribution $P(s_1|s_0, a)$. The environment emits rewards r_0 obtained by sampling the distribution $P(r_0|s_0, a)$. Given a time horizon T, the task of solving a

MDP involves finding a sequence of actions $a_0, a_1, ..., a_{T-1}$ so that the expected sum of the discounted rewards, that is,

$$\mathbb{E}[r_0 + \gamma r_1 + \cdots + \gamma^{T-1} r_{T-1}]$$

is maximized. We see that the discount factor determines how quickly the value of the future rewards are depreciated. For example, consider the extreme, undiscounted case where $\gamma = 1$ and thus immediate and all future rewards are weighted equally. At the other extreme, $\gamma = 0$ and hence, only immediate rewards are considered.

Roughly, in the case where the reward and next state distributions are not known by the agent, then the problem formulation becomes what is known as reinforcement learning.

2.2 Reinforcement Learning

Reinforcement learning (RL) enables an agent with a goal to learn by continuously interacting with an environment. At each time step, the agent exists in a given *state* within the environment and the agent executes an *action* in the environment where the environment provides feedback to the agent with the *next state* and *reward*. The goal of the agent is to find a policy $\pi : S \to A$ such that the expected cumulative rewards $\mathbb{E}[\sum_{t=0}^{T} \gamma^t r_t | a_t = \pi(s_t)]$ in the lifetime T of the agent is maximized. More formally, the *state-action quality* function describes the expected total rewards received by an agent in state s taking action a and then following policy π thereafter and is defined

$$Q^{\pi}(s, a) = \mathbb{E}[r_0 + \sum_{t=1}^{T} \gamma^t r_t | s_0 = s, a_0 = a, a_t = \pi(s_t)]. \tag{2.1}$$

One popular algorithm for obtaining the optimal state-action quality function Q^* is Q-iteration where the $Q(s_t, a_t)$ is updated using the observed reward and a bootstrapped estimate of the next value:

$$Q(s_t, a_t) \leftarrow (1 - \alpha) Q(s_t, a_t) + \alpha[r_t + \max_a Q(s_{t+1}, a)] \tag{2.2}$$

where $\alpha \in (0, 1]$ is the learning rate. After the algorithm has converged, the optimal policy π^* can be simply obtained from Q^* by picking the action that yields the highest value, i.e., $\pi^* = \arg\max_a Q^*(s, a)$.

One of the fundamental challenges faced by a RL agent, known as the *exploration-exploitation dilemma*, concerns the predicament faced by an agent wherein it must balance

the selection of actions that either explore the environment so as to discover regions of the state space that may yield higher rewards or exploit the environment by reaching states known to yield high rewards. One can imagine that the agent must find a good balance between the two as exploring too much and exploiting too little would hamper the total rewards collected by the agent, but the converse might also result in the agent getting stuck in a suboptimal policy. Some of the most popular algorithms used to tackle this dilemma falls under a category known as *undirected exploration*. These algorithms are "undirected" because they do not take into the account the agent's current state of knowledge. In other words, they are heuristic algorithms that simply ensure that enough (blind) exploration will be performed to guarantee convergence to an optimal policy. An example of such heuristics is the classic ϵ-greedy algorithm, which selects actions uniformly at random once in a while and otherwise exploits by choosing a greedy action with respect to the agent's current estimates. More formally, assuming a continuous action space, we may define the the choice of action to select a_{select} for a given time step t as

$$a_{\text{select}} = \begin{cases} a_{\text{unif}} \sim \text{unif}(a_{\min}, a_{\max}) & \text{if } p < \epsilon \\ a_{\text{greedy}} & \text{otherwise} \end{cases},$$

where $p \sim \text{unif}(0, 1)$, $\epsilon \in [0, 1]$ and a_{greedy} is the current best action predicted by the agent. We may fix ϵ to a constant but perhaps a more judicious choice would involve, for example, setting $\epsilon = \epsilon_{\min} + (\epsilon_{\max} - \epsilon_{\min})e^{-\tau t}$ where $0 \leq \epsilon_{\min} < \epsilon_{\max} \leq 1$ and τ is the decay rate, as this would allow ϵ to decrease exponentially over time, following the intuition that during the early stages of the algorithm, the agent would have little information about the environment and hence, ought to explore. Conversely, as t grows larger, the agent would have presumably explored much of the state space by then and thus should prioritize selecting actions that exploit.

In contrast, *directed exploration* is a term used to describe algorithms that do in fact, take into consideration the agent's current knowledge state in addressing the exploration-exploitation dilemma. In other words, these methods explicitly model the unknown distributions $P(s'|s, a)$ and/or $P(r|s, a, s')$ and the choice of action will depend on the posterior evolution of these distributions for a given planning trajectory. One example of such algorithm that falls under this category involves performing forward search in a Bayes-Adaptive MDPs (BAMDPs) [12]. BAMDPs can roughly be thought of augmenting MDPs the state space S with the posterior parameters of the transition function Φ, resulting in a joint space $S^+ = S \times \Phi$, which we call the *hyper-state*. This joint space models not only the physical state but also the information state of the agent, allowing it to track its belief of the system dynamics as it performs forward search planning. In more details, consider

Bellman's equation in classic RL

$$V(s) = \max_a \sum_{s'} P(s'|s,a)(R(s,a,s') + \gamma V(s'))$$

which gives us a mathematical condition to be satisfied for a policy to be optimal. In the case of BAMDPs, we augment this equation with the agent's belief state ϕ of the environment's transition dynamics:

$$V(s,\phi) = \max_a \sum_{s'} P(s'|s,\phi,a)(R(s,a,s') + \gamma V(s',\phi_{s,a,s'}))$$

where $\phi_{s,a,s'}$ is the posterior belief after observing transition tuple (s,a,s'). The belief ϕ tells us what parts of the model are not well known and therefore worth exploring. Under this framework, the exploration-exploitation dilemma gets lumped under a single objective of maximizing the expected total rewards in a given time horizon. If we can model these equations with high fidelity, then in theory, we can achieve an *optimal* exploration-exploitation trade-off known as *Bayes-optimality* condition.

The two aforementioned methods of exploration, directed and undirected, respectively do and do not require a model of the system dynamics and we call the former class of algorithms *model-based* and the latter *model-free*. Extending beyond exploration-exploitation, it turns out almost all algorithms in RL can be dichotomically categorized to either side and we take some time to discuss their similarities and differences as well as highlight a few example algorithms from both classes.

2.2.1 Model-free Reinforcement Learning

As suggested by the name, model-free algorithms are methods that do not model explicitly the state dynamics, but instead learn policies and/or values directly from the tuples (s,a,r,s'). There is a general consensus that model-free algorithms tend to have poor *sample efficiency* in the sense that relative to more sample efficient algorithms, model-free methods require more training tuples, hence more interactions with the environment, to learn a policy of the same performance. This bottleneck renders model-free algorithms inapplicable to, for example, low-data tasks such as the learning from scratch of real-world robots as these systems are constrained from repetitively performing actions for data gathering due to the subjection of wear and tear to physical systems. An advantage, however, is that these methods are generally more computationally efficient, resulting in procedures with shorter run-time duration than algorithms of a different class. Another benefit is that

model-free algorithms are generally simpler as a consequence of eschewing the need to fit model dynamics in favor of directly learning the policies and/or values.

The Q-learning update 2.6 discussed earlier is an example of a model-free algorithm since it directly updates its state-value estimation using its data tuples. It is also an instance of what is known as a *off-policy* technique in that it learns a value function that is based on a greedy policy, due to the presence of the max operator, when in fact, the agent does not necessarily follow a greedy policy. An example of a model-free *on-policy* algorithm is the the SARSA algorithm, standing for state-action-reward-state-action, obtained simply via replacing the max operation of Q-learning with the Q-value for the subsequent state s_{t+1} and the subsequent action a_{t+1} that the agent would take in s_{t+1} (and hence, on-policy):

$$Q(s_t, a_t) \leftarrow (1-\alpha)Q(s_t, a_t) + \alpha[r_t + Q(s_{t+1}, a_{t+1})]. \tag{2.3}$$

Both Q-learning and SARSA are *action-value* methods in that they implicitly derive a policy after first learning a state-action value function, in contrast to *policy-based* methods that directly learn a policy mapping $\pi : s \rightarrow a$ from state to action.

A popular policy-based method is the REINFORCE algorithm, derived from the idea that one can directly optimize the parameters of the policy via gradient descent to increase the probability of selecting actions towards a trajectory of high rewards under expectation. Formally, we define a class of parameterized policies $\Pi = \{\theta_\pi, \theta \in \mathbb{R}^m\}$ and for each policy we define its value $J(\theta)$ and the task is to find the optimal policy $\theta^* = \arg\max_\theta J(\theta)$. Mathematically, we write

$$\begin{aligned} J(\theta) &= \mathbb{E}\Big[\sum_{t\geq 0} \gamma^t | \pi_\theta\Big] \\ &= \mathbb{E}_{\tau\sim p(\tau|\theta)}[R(\tau)] \\ &= \int_\tau R(\tau) p(\tau|\theta) d\tau, \end{aligned} \tag{2.4}$$

where $r(\tau)$ is the reward of the trajectory $\tau = \{(s_t, a_t, r_t)\}_{t=0}^{T-1}$. Taking the derivative of $J(\theta)$ with respect to θ, we obtain

$$\begin{aligned} \nabla_\theta J(\theta) &= \int_\tau R(\tau) \nabla_\theta p(\tau|\theta) d\tau \\ &= \int_\tau R(\tau) p(\tau|\theta) \frac{\nabla_\theta p(\tau|\theta)}{p(\tau|\theta)} d\tau \\ &= \int_\tau R(\tau) p(\tau|\theta) \nabla_\theta \log p(\tau|\theta) d\tau \\ &= \mathbb{E}_{\tau\sim p(\tau|\theta)}[R(\tau) \nabla_\theta \log p(\tau|\theta) d\tau], \end{aligned} \tag{2.5}$$

which is a value that can be estimated using Monte Carlo methods. Furthermore, we have

$$p(\tau|\theta) = \prod_{t\geq 0} p(s_{t+1}|s_t, a_t)\pi_\theta(a_t|s_t),$$

thus

$$\log p(\tau|\theta) = \sum_{t\geq 0} \log p(s_{t+1}|s_t, a_t) + \log \pi_\theta(a_t|s_t),$$

and differentiating, we obtain

$$\nabla \log p(\tau|\theta) = \sum_{t\geq 0} \nabla \log \pi_\theta(a_t|s_t),$$

noting that the resulting gradient does not depend on the transition probabilities. Given a sample trajectory τ, the gradient can thus be estimated as

$$\nabla J(\theta) \approx \sum_{t\geq 0} R_t(\tau)\nabla \log \pi_\theta(a_t|s_t)$$

and applied via the learning update $\theta \leftarrow \theta + \alpha\nabla J(\theta)$. $R_t(\tau) = \sum_{k=t}^{T-1} \gamma^{k-t} r_k$ is the Monte Carlo estimate of the discounted sum of rewards from time t to $T-1$.

Rather than just learning the action-values or just the policy, it is possible to learn both, and indeed a class of model-free methods by the name of *actor-critic algorithms* do just that. The name derives from the notion that the "actor" maps states to actions and the "critic" maps state-action pairs to values, critiquing the performance of the actor. As a simple example, consider the scenario where we separately parameterize an action-value $Q_\theta(\cdot,\cdot)$ and policy $\pi_\vartheta(\cdot)$ mapping with parameters θ and ϑ respectively. One may use any action-value method to train $Q_\theta(\cdot,\cdot)$ (e.g., Q-learning or SARSA) and as a concrete example consider the use of an one step temporal difference (TD) *Q-learning* algorithm, with corresponding squared error loss

$$L(\theta) = (r_t + \gamma \max_{a'} Q_\theta(s_{t+1}, a') - Q_\theta(s_t, a_t))^2. \tag{2.6}$$

This error is referred to as one step TD because it unrolls Bellman's equation one step into the future, but in general any n step TD loss can be constructed by unrolling Bellman's equation n steps into the future. In addition, Q-iteration can roughly be understood as the term ascribed to the tabular case of Q-learning while 2.6 is appropriate for a parameterized class of action-value functions. The policy is trained using a REINFORCE-like algorithm

wherein Monte Carlo estimate is replaced by the learned action-value function, resulting in a gradient estimator of the form

$$\nabla J(\vartheta) \approx \sum_{t \geq 0} Q_\theta(s_t, a_t) \nabla \log \pi_\vartheta(a_t|s_t). \tag{2.7}$$

Learning is performed by alternating between minimizing loss 2.6 and applying gradient update 2.7.

2.2.2 Model-based Reinforcement Learning

As suggested by the name, model-based is the opposite of model-free wherein a part of the algorithm involves learning the system dynamics $P(s'|s, a)$ of the MDP. Model-based algorithms also possess many properties that are converse to model-free such as the notion that model-based methods have better sample efficiency, but are less computationally efficient. Likewise, this bottleneck renders model-based algorithms inapplicable to, for example, tasks that require a short response time such as robotic systems with real-time response. I give two examples below to better illustrate the algorithms belonging to this class.

A classical model-based method is the Dyna architecture [1], a framework inspired by the commonsense idea that planning is 'trying things in your head,' using an internal model of the world. In more technical details, Dyna, as with standard RL agents, uses reinforcements from real experience (i.e., (s, a, r, s') received from the environment), but in addition, the framework also uses this experience to construct what the author calls an *action model* that can be used to predict the results of actions. Unique to Dyna, this action model is used to generate hypothetical model-generated experiences based on which reinforcement are then used to improve the policy; this in effect is a planning process. Experiments in grid-world domains involving simple navigation tasks have demonstrated that the more hypothetical experiences using the world model (i.e., the more "planning steps" per interaction with the MDP), the faster an optimal path was found [42].

In spite of the theoretical appeal of Dyna, the success of the framework has mainly been most pronounced in grid-world toy tasks. When scaling the problem to higher dimensions and/or real-world control, a classical model-based technique by the name of model predictive control (MPC) has been shown to be fruitful. The method was derived from the field of classical control theory and had initial applications in chemical plants and oil refineries [15] and since then, it has found application in control problems in a diverse set

of fields ranging from power electronics [16] to wind turbines [26], planetary rovers [6] and agriculture [11]. Consider the generic constrained optimization problem

$$\begin{aligned} \underset{a_0,\ldots,a_{T-1}}{\text{minimize}} \quad & J = \sum_{t=0}^{T-1} \ell(s_t, a_t) \\ \text{subject to} \quad & s_{t+1} = f(s_t, a_t, e_t),\ t = 0, \ldots, T-1, \end{aligned} \tag{2.8}$$

that is, we wish to find a control sequence that minimizes the cumulative cost over a time horizon T subject to the dynamics given by f with random disturbance e_t. In some cases, the cost (i.e., negative reward) $\ell(\cdot, \cdot)$ and/or transition function f are given, but in the general scenario, they would have to be learned from data. The pseudocode for MPC is given by Algorithm 1. The time and space complexity of finding an (approximate)

Algorithm 1 MPC

1: **procedure** MPC($\ell(\cdot, \cdot)$, $\mathcal{S}$, $\mathcal{A}$) ▷ $s \in \mathcal{S}$, set of states; $a \in \mathcal{A}$, set of actions.
2: Initialize $\mathcal{E}$ ▷ Initialize the environment.
3: $s_{\text{current}} \leftarrow \mathcal{E}$`.reset()` ▷ Observe initial state of environment.
4: **for** $\mathcal{T}$ interactions with the environment **do**
5: $a_0^*, \ldots, a_{T-1}^* \leftarrow$ `solve(`$\ell(\cdot, \cdot)$`,` s_{current}`)` ▷ Solve 2.8 for $s_0 = s_{\text{current}}$ to obtain optimal trajectory.
6: $s_{\text{current}} \leftarrow \mathcal{E}$`.step(`$a_0^*$`)` ▷ Step through the environment with the first action.
7: **end for**
8: **end procedure**

solution to 2.8 (`solve()` in line 5) is vital to the scalability and applicability of MPC as the algorithm calls this sub-procedure at every time step. In the special case where the formulation is linear in system dynamics and quadratic in cost, in other words, they are respectively of the form $s_{t+1} = As_t + Ba_t + e_t$ and $J = \mathbb{E}_e[x_{T-1}Qx_{T-1}^\top + \sum_{t=0}^{T-2} x_tQx_t^\top + a_tRa_t^\top]$, where A, B, Q and R are square matrices, then the optimal control sequence can be computed in closed form by solving the *Algebraic Riccati equation*. In the general case, however, solving 2.8 may be intractable and one simple procedure to approximate its solution is the "*shooting method*", wherein a set of k trajectories $K = \{(a_0, \ldots, a_{T-1})_i\}_{i=0}^k$ are first sampled uniformly at random, then each trajectory is assigned a cost according to J and the trajectory with the lowest cost is chosen as the solution. Effectively, the algorithm plans for a horizon, takes one step into the environment and replans upon observing the new state. At the cost of the additional computation incurred with frequent replanning, this strategy makes the algorithm robust to non-stationary environments.

One of the main challenges to model-based methods that prevent them from being widely applied is the notion of *model-bias* [39, 4]. The idea is that when a regression model is fitted with limited training data, there may be areas in the parameter space in which there are very few examples for the model to base its predictions upon, yet it makes these inferences with one hundred percent confidence; in other words, the model has no notion of uncertainty. To illustrate the detrimental effects of model-bias, consider the example where an autonomous car is using a model-based RL algorithm to learn how to drive and the dynamics model is learned from training data gathered from on-road experience. If the model does not factor in any notion of uncertainty, then it would make predictions about the effects of both driving down the highway and driving off the cliff with full certainty! Needless to say, this may have catastrophic consequences if the predictions from the model are in fact wrong. Therefore, for the sake of reducing model-bias, it is paramount that probabilistic dynamics models are employed to represent uncertainty in the predictions. One of the most popular regression methods for such a task is the Gaussian process regression (GPR) and we elaborate on this technique below.

2.3 Gaussian Process Regression

Gaussian processes (GPs) are a powerful state-of-the-art Bayesian nonparametric regression method. GPs infer a distribution over functions and in the context of nonparametrics, this corresponds to lifting the parameter space to infinitely many dimensions where the function is modelled as a distribution over an infinitely long vector. This allows the model to be highly expressive, making underfitting an issue that typically does not occur. At the same time, the model is resistant to overfitting as the Bayesian approach provides a systematic approach in accounting for noise in data with a prior and a posterior update upon new observations [9].

The interpretation of GPs can be approached from what is called a *function-space view* or a *weight-space view* [37] and in this exposition, we proceed with the latter. Consider a training set of $\mathcal{D}$ of M observations, $\mathcal{D} = \{(\boldsymbol{x}_i, y_i) | i = 0, \ldots, M-1\}$ where $\boldsymbol{x}$ is an input column vector of size n and y is a scalar. Denote the design matrix $\boldsymbol{X} \in \mathbb{R}^{M \times n}$ as the aggregate of the input vectors and the column vector $\boldsymbol{y}$ as the collation of the output scalars. The weight-space view begins with the *Bayesian linear regression* model where it assumes a linear relation between $\boldsymbol{x}$ and $\boldsymbol{y}$ corrupted with Gaussian noise, that is, $\boldsymbol{y} = \boldsymbol{X}\boldsymbol{w} + \boldsymbol{\epsilon}$ where $\boldsymbol{\epsilon} = [\epsilon_0, \ldots, \epsilon_{M-1}]^\top$, $\epsilon_i \sim \mathcal{N}(0, \sigma^2)$, for all $i = 0, \ldots, M-1$. The noise and independence assumption along with the model definition give rise to a *likelihood*

function given by the expression

$$\begin{aligned} p(\boldsymbol{y}|\boldsymbol{X}, \boldsymbol{w}) &= \prod_{i=0}^{M-1} p(y_i|\boldsymbol{x}_i, \boldsymbol{w}) \\ &= \prod_{i=0}^{M-1} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(y_i - \boldsymbol{x}_i \boldsymbol{w})^2}{2\sigma^2}} \\ &= \frac{1}{(2\pi\sigma^2)^{\frac{M}{2}}} e^{-\frac{\|\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w}\|_2^2}{2\sigma^2}} \\ &= \mathcal{N}(\boldsymbol{y}|\boldsymbol{X}\boldsymbol{w}, \sigma^2 I). \end{aligned}$$

The Bayesian formulation requires a *prior* over the weights specifying the belief of $\boldsymbol{w}$ before any observations are made. We proceed by specifying a normal prior with zero mean and covariance $\boldsymbol{V}_0$

$$\boldsymbol{w} \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{V}_0).$$

The posterior distribution of the model is computed according to Bayes' rule which states that the posterior is equal to the likelihood times the prior divided by the marginal likelihood, that is,

$$\text{posterior} = \frac{\text{likelihood} \times \text{prior}}{\text{marginal likelihood}}.$$

Applying Bayes' rule, the posterior of the model is given by the expression

$$p(\boldsymbol{w}|\boldsymbol{X}, \boldsymbol{y}) = \frac{p(\boldsymbol{y}|\boldsymbol{X}, \boldsymbol{w}) p(\boldsymbol{w})}{\int p(\boldsymbol{y}|\boldsymbol{X}, \boldsymbol{w}) p(\boldsymbol{w}) d\boldsymbol{w}}.$$

Focusing on the numerator, the likelihood times the prior, then "completing the square", yields the expression

$$\begin{aligned} p(\boldsymbol{w}|\boldsymbol{X}, \boldsymbol{y}) &\propto \mathcal{N}(\boldsymbol{y}|\boldsymbol{X}, \sigma^2 I) \times \mathcal{N}(\boldsymbol{0}, \boldsymbol{V_0}) \\ &\propto \exp\Big(-\frac{1}{2\sigma^2} (\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w})^\top (\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w}) \Big) \exp\Big(-\frac{1}{2} \boldsymbol{w}^\top \boldsymbol{V}_0^{-1} \boldsymbol{w} \Big) \\ &\propto \exp\Big(-\frac{1}{2} (\boldsymbol{w} - \boldsymbol{w}_N)^\top (\boldsymbol{V}_0^{-1} + \frac{1}{\sigma^2} \boldsymbol{X}^\top \boldsymbol{X}) (\boldsymbol{w} - \boldsymbol{w}_N) \Big). \end{aligned}$$

We recognize that the posterior is also a Gaussian distribution with covariance $\boldsymbol{V}_N = \sigma^2(\sigma^2 \boldsymbol{V}_0^{-1} + \boldsymbol{X}^\top \boldsymbol{X})^{-1}$ and mean $\boldsymbol{w}_N = \frac{1}{\sigma^2} \boldsymbol{V}_N \boldsymbol{X}^\top \boldsymbol{y}$. In the Bayesian model, predictions are made by averaging the output over all possible function parameters, in contrast to the non-Bayesian approach which produces a single output prediction. Hence, the predictive

distribution, $f_* \triangleq f(\boldsymbol{x}_*)$, at a given a test point $\boldsymbol{x}_*$, is obtained by marginalizing over the model parameters of the posterior distribution, given by the expression

$$\begin{aligned} p(f_*|\boldsymbol{x}_*, \boldsymbol{X}, \boldsymbol{y}) &= \int p(f_*|\boldsymbol{x}_*, \boldsymbol{w}) p(\boldsymbol{w}|\boldsymbol{X}, \boldsymbol{y}) d\boldsymbol{w} \\ &= \mathcal{N}(f_*|\boldsymbol{x}_*^\top \boldsymbol{w}_N, \sigma^2 + \boldsymbol{x}_*^\top \boldsymbol{V}_N \boldsymbol{x}_*). \end{aligned}$$

The predictive distribution is again a Gaussian distribution with mean prediction centred at $\boldsymbol{x}_*^\top \boldsymbol{w}_N$ and variance $\sigma^2 + \boldsymbol{x}_*^\top \boldsymbol{V}_N \boldsymbol{x}$ representing the uncertainty in the prediction. Note that two factors affect the predictive uncertainty, namely, the uncertainty that is inherent in the noise of the measured data, represented by the additive term σ^2, as well as the uncertainty reflected by the proximity of the test point to the training set, given by the term $\boldsymbol{x}_*^\top \boldsymbol{V}_N \boldsymbol{x}_*$.

A drawback to Bayesian linear regression is that its expressive power is limited to modelling linear patterns in data. One simple way that Bayesian linear regression can be extended to perform non-linear regression is by first mapping the inputs into a higher dimensional space via a non-linear basis function $\phi(\cdot)$ and then performing linear regression on these high dimensional features rather than directly on the inputs. Similar to before, denote $\phi(\boldsymbol{x}_i)$ as the column basis vector obtained after applying mapping $\phi(\cdot)$ to input $\boldsymbol{x}_i$ and the matrix $\Phi(\boldsymbol{X})$ as the concatenation of the input basis vectors. We proceed as before where we assume a linear relation between $\phi(\boldsymbol{x})$ and $\boldsymbol{y}$ corrupted with Gaussian noise:

$$\boldsymbol{y} = \Phi(\boldsymbol{X})\boldsymbol{w} + \boldsymbol{\epsilon}.$$

The derivation is unchanged as well except for the minor difference where every instance of $\boldsymbol{X}$ is substituted with $\Phi(\boldsymbol{X})$. Hence, for a test data point $\boldsymbol{x}_*$, the posterior predictive distribution is given by the expression

$$p(f_*|\boldsymbol{x}_*, \boldsymbol{X}, \boldsymbol{y}) = \mathcal{N}\Big(f_*\Big|\frac{1}{\sigma^2}\phi(\boldsymbol{x}_*)^\top \boldsymbol{V}_N \phi(\boldsymbol{X})\boldsymbol{y}, \phi(\boldsymbol{x}_*)^\top \boldsymbol{V}_N \phi(\boldsymbol{x}_*)\Big). \tag{2.9}$$

Using the *Woodbury matrix identity*, one can show that Equation 2.9 has the following equivalent form:

$$\begin{aligned} p(f_*|\boldsymbol{x}_*, \boldsymbol{X}, \boldsymbol{y}) = \mathcal{N}\big(f_*\big|&\phi(\boldsymbol{x}_*)^\top \boldsymbol{V}_0 \phi(\boldsymbol{X})\big(\phi(\boldsymbol{X})^\top \boldsymbol{V}_0 \phi(\boldsymbol{X}) + \sigma^2 \boldsymbol{I}\big)^{-1}\boldsymbol{y}, \\ &\phi(\boldsymbol{x}_*)^\top \boldsymbol{V}_0 \phi(\boldsymbol{x}_*) - \phi(\boldsymbol{x}_*)^\top \boldsymbol{V}_0 \phi(\boldsymbol{X})\big(\phi(\boldsymbol{X})^\top \boldsymbol{V}_0 \phi(\boldsymbol{X}) + \sigma^2 \boldsymbol{I}\big)^{-1} \\ &\phi(\boldsymbol{X})^\top \boldsymbol{V}_0 \phi(\boldsymbol{x}_*)\big). \end{aligned} \tag{2.10}$$

Note that in this expression, the basis functions and the covariance matrix always appear together in the form $\phi(\cdot)^\top \boldsymbol{V}_0 \phi(\cdot)$. Let us define $k(\boldsymbol{x}, \boldsymbol{y}) = \phi(\boldsymbol{x})^\top \boldsymbol{V}_0 \phi(\boldsymbol{y})$, which is called a *kernel*. Since a covariance matrix is symmetric positive definite, we can apply the Cholesky decomposition to decompose $\boldsymbol{V}_0$ into an upper triangular matrix, U, and a symmetric lower triangular matrix, $U^\top$, such that $\boldsymbol{V}_0 = U^\top U$. Defining $\varphi(\boldsymbol{x}) = U\phi(\boldsymbol{x})$, the kernel, $k(\boldsymbol{x}, \boldsymbol{y}) = \varphi(\boldsymbol{x}) \cdot \varphi(\boldsymbol{y})$, can then be expressed as an inner product of two basis vectors. Replacing all instances of the basis vector inner products with a kernel, the posterior predictive distribution then takes the form

$$p(f_*|\boldsymbol{x}_*, \boldsymbol{X}, \boldsymbol{y}) = \mathcal{N}(f_*|\bar{f}_*, \text{var}(f_*)), \tag{2.11}$$

where

$$\begin{aligned} \bar{f}_* &= (k(\boldsymbol{X}, \boldsymbol{X}) + \sigma^2 I)^{-1} \boldsymbol{y}, \\ \text{var}(f_*) &= k(\boldsymbol{x}_*, \boldsymbol{x}_*) - k(\boldsymbol{x}_*, \boldsymbol{X})(k(\boldsymbol{X}, \boldsymbol{X}) + \sigma^2 I)^{-1} k(\boldsymbol{X}, \boldsymbol{x}_*). \end{aligned} \tag{2.12}$$

Through kernelizing the input features, the resulting model 2.12 is known as a *Gaussian process*.

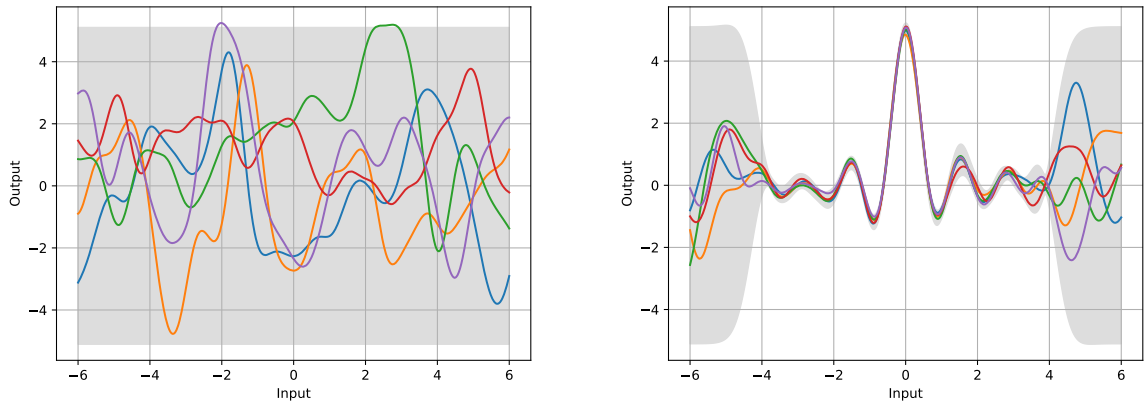
As an example to demonstrate the regression capabilities of GPs, consider a training set of 100 pairs of scalar observations where $x_i \sim \text{unif}(-4, 4)$ and $y_i = \frac{\sin(5x_i)}{x_i} + \epsilon_i$ where $\epsilon_i \sim \mathcal{N}(0, 0.04)$ for all $i = 0, \ldots, 99$. The covariance function is chosen to be the *squared exponential kernel* (SE) defined as

$$k(\boldsymbol{x}, \boldsymbol{y}) = \sigma_f^2 e^{-\frac{\|\boldsymbol{x}-\boldsymbol{y}\|_2^2}{2\ell^2}},$$

where ℓ is the *length-scale* which controls the smoothness of the function and the scaling factor σ_f^2 is the *signal amplitude* which determines the range and rate of change of the function [13]. Consider a vector $\boldsymbol{x}_* = [-6, \ldots, 6]^\top$ consisting of 1000 evenly spaced test points in the range $[-6, 6]$. The prior of the GP evaluated at the test points is given by the distribution

$$\boldsymbol{f}_* \sim \mathcal{N}(\boldsymbol{f}_*|\boldsymbol{0}, k(\boldsymbol{x}_*, \boldsymbol{x}_*))),$$

Figure 2.1a depicts five functions sampled from the prior distribution and the gray-shaded regions represent three standard deviations from the mean centered at zero. As expected from an uninformed prior, the error bars have "uniform" lengths and the sampled curves exhibit characteristics of random functions. The posterior distribution evaluated on the test points is given by $\text{var}(f_*)$ and in the same vein, Figure 2.1b illustrates five functions sampled from distribution 2.12 and the gray-shaded regions, likewise, represent three standard deviations from the mean prediction (shown in Figure 2.2 by the dashed red line). In



(a) Prior (b) Posterior

Figure 2.1: Five functions sampled from a GP (a) prior and (b) posterior. The gray-shaded regions represent three standard deviations from the mean.

contrast, the error bars are narrow within the interval $[-4, 4]$ and wide outside of this region, representing the notion that in the domain with training data, the model is very confident and vice versa in said domain's complement. Indeed, this idea is corroborated by the sampled functions as their deviation is minuscule within $[-4, 4]$ and large outside of this interval. Finally, Figure 2.2 plots the training data, alongside the mean fit accompanied by their corresponding confidence intervals, and the ground truth function, illustrating the non-linear regression capabilities of GPs.

Covariance functions are a crucial component to the GP as they encode assumptions about the functions that we wish to learn such as, for example, the smoothness of the learned curves. From another viewpoint, they could be thought of as a metric specifying the distance similarity between two points under the assumption that closer points ought to be more similar [37]. In the example above, the squared exponential (SE) kernel is used, which is one among others, each with varying characteristics. To illustrate this, we give examples of two other kernels and touch upon their properties. The SE kernel is a special case of a class of kernels known as the *Matérn covariance function*, defined

$$k_{\text{Matérn}}(\boldsymbol{x}, \boldsymbol{y}) = \frac{2^{1-\nu}}{\Gamma(\nu)} \left(\frac{\sqrt{2\nu} r}{\ell} \right)^{\nu} K_{\nu} \left(\frac{\sqrt{2\nu} r}{\ell} \right)$$

where $r = \|\boldsymbol{x} - \boldsymbol{y}\|_2$, with positive hyperparameters ν and ℓ and K_{ν} is the modified Bessel function of second kind of order ν [3]. The ν parameter controls the smoothness of the

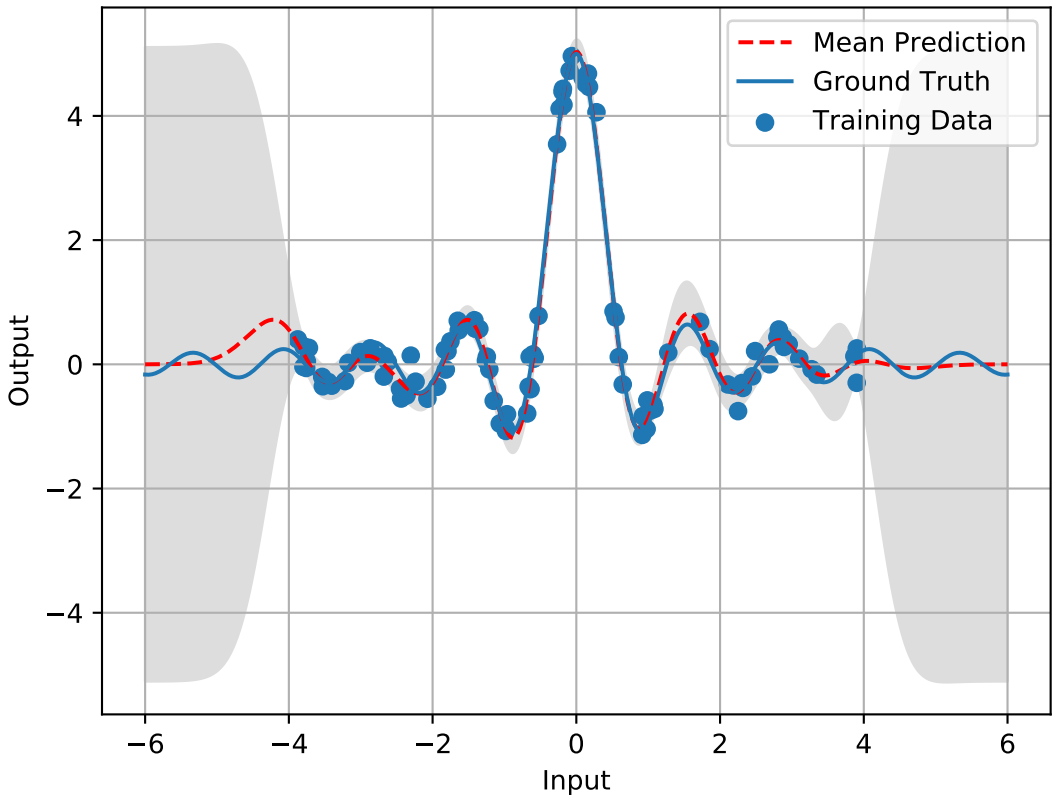


Figure 2.2: A GP fitted to a training set of 100 pairs of scalar observations where $x_i \sim \text{unif}(-4, 4)$ and $y_i = \frac{\sin(5x_i)}{x_i} + \epsilon_i$ where $\epsilon_i \sim \mathcal{N}(0, 0.04)$ for all $i = 0, \ldots, 99$. The gray-shaded regions represent three standard deviations from the mean.

resulting function and the Matérn kernel reduces to an SE kernel for $\nu \to \infty$. For the case that $\nu = 1/2$, the kernel becomes

$$k_{\nu=1/2} = e^{-\frac{r}{\ell}},$$

known as the *Ornstein-Uhlenbeck process* [46] which was introduced to model the behaviour of a particle undergoing Brownian motion. Another example, consider the *periodic kernel* given by

$$k_{\text{periodic}}(\boldsymbol{x}, \boldsymbol{y}) = \exp\left(- \frac{2\sin^2\left(\frac{\pi\|\boldsymbol{x}-\boldsymbol{y}\|_2}{p}\right)}{\ell^2} \right)$$

which, as the name suggests, encodes a periodic assumption in the function, making them well-suited for sinusoidal-like data. The period p determines the distance between repetitions and ℓ parameter functions the same way as in the SE kernel.

In the discussions above, there is a notion of hyperparameters. Yet it was not elaborated as to how one may go about choosing these hyperparameters in the model, but fortunately, the Bayesian formulation offers a principled approach to this task. Consider the *marginal likelihood*, or the marginalization of a distribution over the parameter $\boldsymbol{\theta}$, given by the expression

$$p(\boldsymbol{y}|\boldsymbol{X}) = \int p(\boldsymbol{y}|\boldsymbol{\theta}, \boldsymbol{X}) p(\boldsymbol{\theta}|\boldsymbol{X}) d\boldsymbol{\theta}$$

and for GPs, the log of the marginal likelihood can be shown to be

$$\log p(\boldsymbol{y}|\boldsymbol{X}) = -\frac{1}{2}\boldsymbol{y}^\top (K + \sigma^2 I)^{-1}\boldsymbol{y} - \frac{1}{2}\log|K + \sigma^2 I| - \frac{n}{2}\log 2\pi. \tag{2.13}$$

Selecting the hyperparameters would involve maximizing 2.13 with respect to the hyperparameters. This amounts to maximizing the *model evidence* which is a principled and automatic way of selecting parameters that balance between expressivity and simplicity by Occam's razor [36].

2.4 Covariance Matrix Adaptation Evolution Strategy

Covariance matrix adaptation evolution strategy (CMA-ES) is a stochastic, or randomized, method for real-parameter (continuous domain) optimization of non-linear, non-convex functions [18]. We first explain the concepts of evolution strategy and genetic algorithms and then the extensions made by CMA-ES.

Evolution strategy could roughly be understood as a term that describes any algorithm that produces a set of *candidate solutions* to evaluate a problem. The quality of the candidate solutions are measured based on what is known as a *fitness function* and the *fitness* of each solution is fed back to the evolution strategy where then the algorithm produces the next generation of (hopefully) fitter candidate solutions. In more detail, the candidate solutions are generated by sampling an arbitrary number of times from a distribution and upon receiving fitness scores of each candidate solution, the algorithm updates the distribution to reflect the new gain in information. As a concrete example, consider a two-dimensional fitness function $F(x, y)$ and we wish to find the parameters x and y such that $F(x, y)$ is maximized. One simple evolution strategy for this objective would be to define a normal distribution with mean $\boldsymbol{\mu}$ and covariance $\boldsymbol{\Sigma}$ where the latter is held fix. $\boldsymbol{\mu}$ is initially set to the origin and the algorithm proceeds by sampling N candidate solutions and $\boldsymbol{\mu}$ is set to the candidate solution with the highest fitness. The algorithm then repeats by sampling around the new mean.

Informally, the term "genetic" in genetic algorithms refer to the idea of injecting variations in the process of selecting/generating the next generation of candidate solutions. In our two-dimensional example, diversity could be incorporated into our procedure by, for instance, keeping only the top ten percent of the candidate solutions and letting the remaining prospects die. Upon updating the new mean, we randomly select two solutions in the survivor pool and recombine their parameters, that is, use the x from one solution and the y from the other solution as the parameters of the new mean.

One of the limitation of the evolution and genetic algorithms described above is that their covariance function is held fix. CMA-ES is an algorithm that addresses this shortcoming with a way to adapt the covariance function at every iteration of the search procedure. To understand how CMA-ES works, we recall the formulas for estimating the mean and covariance of a distribution given N samples and continuing from our two-dimensional

example, these are given by

$$\mu_x = \frac{1}{N}\sum_{i=1}^{N} x_i,$$
$$\mu_y = \frac{1}{N}\sum_{i=1}^{N} y_i,$$
$$\sigma_x^2 = \frac{1}{N}\sum_{i=1}^{N} (x_i - \mu_x)^2,$$
$$\sigma_y^2 = \frac{1}{N}\sum_{i=1}^{N} (y_i - \mu_y)^2,$$
$$\sigma_{xy} = \frac{1}{N}\sum_{i=1}^{N} (x_i - \mu_x)(y_i - \mu_y).$$

Of course, applying these equation would only lead to the estimation of the mean and covaraince of the current generation. CMA-ES modifies these equations such that it can dynamically increase the variance when the optimum is far away and conversely, narrow the distribution if the optimum is close. CMA-ES works by keeping only the best N_{best} candidate solutions of a population sample and it calculates the mean of the next generation $(g+1)$ using the best N_{best} solutions, i.e.,

$$\mu_x^{(g+1)} = \frac{1}{N_{\text{best}}}\sum_{i=1}^{N_{\text{best}}} x_i,$$
$$\mu_y^{(g+1)} = \frac{1}{N_{\text{best}}}\sum_{i=1}^{N_{\text{best}}} y_i.$$

CMA-ES modifies the calculation of the covariance values by using the mean of the current generation g, rather than the next generation $(g+1)$, yielding

$$\sigma_x^{2,(g+1)} = \frac{1}{N_{\text{best}}}\sum_{i=1}^{N_{\text{best}}} (x_i - \mu_x^{(g)})^2,$$
$$\sigma_y^{2,(g+1)} = \frac{1}{N_{\text{best}}}\sum_{i=1}^{N_{\text{best}}} (y_i - \mu_y^{(g)})^2,$$
$$\sigma_{xy}^{(g+1)} = \frac{1}{N_{\text{best}}}\sum_{i=1}^{N_{\text{best}}} (x_i - \mu_x^{(g)})(y_i - \mu_y^{(g)}).$$

With the updated parameters $\mu_x^{(g+1)}$, $\mu_y^{(g+1)}$, $\sigma_x^{2,(g+1)}$, $\sigma_y^{2,(g+1)}$ and $\sigma_{xy}^{(g+1)}$, the algorithm can iterate by sampling to generate the population of the proceeding generation [2].

2.5 Related Work

The foundational work of DYNA has inspired numerous work in model-based RL including recent work by [17] where the authors demonstrated that challenging continuous control problems, such as simulated contact robotics, could be solved with less interactions with the environment if a learned system dynamics model augmented the data buffer with *imagination rollouts*. Further, experiments suggested that while prior methods proposed a variety of models for system identification including neural networks, Gaussian process and locally-weighted regression, but the authors found that they were able to obtain good results with a technique known as iteratively refitted time-varying linear models.

The seminal work of [10] introduced the PILCO algorithm which was able to learn optimal policies with unprecedented data efficiency on a set of simulated cart-pole and unicycle riding control problems as well as on a real cart-pole-swing-up environment. The PILCO algorithm consists of first learning a one-step system dynamics model using a GP with SE kernel and then using this learned model, perform trajectory rollouts in search for an optimal policy. Trajectory unrolling calls for the propagation of distributions through the model which are intractable in exact inference and instead, the algorithm circumvents this by opting for approximate inference at every unroll step via a moment matching method. The controller is parameterized by a non-linear radial basis function (RBF) network and policy improvement is performed using analytic gradients. The cost function is exposed to the controller and is given as the RBF of the absolute difference between the controller's current position and the target position. The authors of PILCO stressed the need to overcome model-bias and demonstrated that by leveraging a probabilistic dynamics model in the form of GPs, control polices can be learned from scratch with unprecedented data efficiency. The success of PILCO has spurred numerous research that builds upon the original algorithm such as, for example, PILCO with partially observable states [28], combining PILCO with MPC [20] and extending PILCO to deepPILCO by replacing the GP with a Bayesian neural network (BNN) [27].

With the advent of Deep Learning (DL) and thus, deep RL (DRL), there have been countless works that approach the paradigm of model-based RL from the viewpoint of blackbox neural network architectures and we outline a few of these below. In one example, [43] proposed a neural network architecture called Value Iteration Network (VIN) which implements the idea of enforcing a structural prior on a network based on a stack of

convolutional neural net layers in the goal of mimicking the update operation in the classical value iteration algorithm upon recurrent forward passes through the architecture. The algorithm is not exactly considered a model-based method as it does not learn the system dynamics but it does possess traits of a model-based algorithm in that the value iteration-like update allows the algorithm to perform planning, making it what the authors call a planning-based method. Experiments have demonstrated superior performance on maze navigation tasks that require a notion of planning to reach long-term goal states. In addition, the authors argue that the network was able to generalize by showing that the algorithm was able to learn policies that performed well on unseen environments after it was trained on several instances of the domain with different start state, goal and obstacle positions.

On a similar thread of work, the Predictron [41] introduced the idea of planning in abstract space where at every step, a model is applied to an internal state that simulates a trajectory of next state, action and rewards which are used to predict future value function. Each forward pass of the Predictron computes internal rewards and value functions across multiple planning steps. The architecture is trained end-to-end so that its simulated value functions accurately approximate the true value functions. Experiments were performed on a set of maze problems and pool domain and the authors argued that the controller demonstrated successful maze navigation and sophisticated pool trajectories and cited this as evidence that the Predictron learned to plan.

In yet another similar work, [34] proposed Prediction Network (VPN) which is neural network architecture that attempts to combine model-based and model-free RL in a united network by, respectively, learning the dynamics of an abstract state space for computing future rewards and values, and mapping the learned abstract state space to rewards and values. In other words, VPN learns to predict values via Q-learning and rewards via supervised learning and at the same time, perform lookahead planning to choose actions and compute bootstrapped Q-values. Experiments on a set of 2D grid world domains with the task of collecting as many goals within a time limit shows that the VPN performed better than DQN and that the VPN's performance increases as the number of planning steps increases. In addition, the authors demonstrated the viability of their method in environments with complex visual observations by outperforming DQN in 7 out of 9 Atari games [5].

On the front of vision-based RL, [33] introduced a convolutional autoencoder neural network architecture that conditionally predicts future frames based on a provided action input to the latent features. Using a suite of Atari games, the authors were able to show that the architecture was able to accurately generate up to 100-step action-conditional frames into the future. Furthermore, this strategy of video prediction was put forward as a

method to perform informed exploration whereby the controller would select sequences of actions that lead to a frame that has been visited least often. Experiments on Atari games using DQN with proposed video prediction exploration strategy showed that informed exploration improved DQN's performance in three of five games.

Returning to the area of continuous control robotic systems, the work of [31] investigated the viability of model-based DRL to challenging high-dimensional contact simulated locomotion tasks from the MuJoCo benchmark system [44]. The work parameterized the learned dynamics function as a deep neural network and the parameters were learned via gradient descent on a mean squared error loss. To cope with the model-bias incurred by a non-probabilistic dynamics model, the controller used the MPC algorithm so that only the actions with the lowest uncertainty in the predictions (i.e., actions for the immediate next step) were executed. The work used a simple shooting method where K candidate action sequences are randomly generated, the corresponding state evolutions are calculated using the dynamics model, the rewards of each sequence are calculated, and the sequence that yielded the highest discounted sum of rewards is chosen. The work corroborates the general perception that compared to model-free, model-based methods initially learn faster, but do not match the final performance of model-free algorithms. The work addresses this by combining the strengths of both approaches by first learning a policy with their model-based algorithm and then using this policy as an initialization to the model-free algorithm, Trust Region Policy Optimization (TRPO) [40]. Experiments demonstrated superior performance for the hybrid model compared to both either model-free or model-based alone in all four different MuJoCo location tasks.

[32] extends the previous work by scaling their model-based algorithm to control a real robotic system and in addition, endowing the robot with the ability to automatically learn the visual cues of the terrain that affect its dynamics by augmenting the inputs of the dynamics function with image observations. The resulting MPC controller was tried on the VelociRoACH, a small, mobile, highly dynamic, and bio-inspired hexapedal millirobot on a set of terrain-varying track navigation tasks and the VelociRoACH demonstrated successful and robust policies across all terrains.

One of the most successful and influential work in RL robotics is the Guided Policy Search (GPS) [23] algorithm. This method makes use of a classical trajectory optimization technique called *differential dynamic programming* (DDP) which yields approximate closed form solutions to the state and reward dynamics under linear quadratic assumptions. For a set of tuples generated by a policy, the DDP algorithm solves for an optimal trajectory which then serves as a "guide" to a neural network policy by having it imitate the DDP output. The neural network policy is then executed on the environment to obtain a new trajectory and from there, the algorithm repeats by solving for a new trajectory using DDP.

GPS has inspired many other research that use and build upon the original work such as, for example, the application of GPS to visuomotor control [14, 22] and to learning contact-rich manipulation skills [24], the demonstration that GPS can be cast as an approximation to mirror descent [29], the augmentation of memory states to GPS [47], and finally, the extension of the algorithm to environments with unknown dynamics [21]. The success of GPS and its related work has demonstrated its potential to be applied to control in real-world robotic systems.

Chapter 3

Proposed Model

The work in this thesis is inspired by the PILCO algorithm described in Section 2.5. Like the PILCO algorithm, our algorithm performs system identification by learning a function $f : S \times A \to S$ that maps state-action pairs to next states. Unlike PILCO, however, we instead opt to use an approximate GP based on random Fourier features (RFFs) [35] for the system dynamics function f rather than an exact GP. The rationale behind this decision is that using RFFs allows the regression model to remain in primal space, meaning the design matrix $\boldsymbol{X}^\top \boldsymbol{X}$ does not grow in size with respect to the amount of training data, making the model more applicable to an online learning setting. Using the learned dynamics model, we perform policy search by unrolling states into the future to find the sequence of actions that yields the highest expected cumulative discounted rewards. Following the formulation of BAMDPs [12], our algorithm augments the state space with a belief state constructing a set of hyperstates that are used by the policy search algorithm to optimize for a policy that performs directed exploration by taking into account the uncertainty in the state space given by the belief state. The tracking of hyperstate evolution through future time steps calls for a tree search-like operation, which does not suit well with gradient-based optimizers, hence to optimize the parameters of the policy, we opt instead for a black box optimizer in the form of covariance matrix adaptation evolution strategy (CMA-ES) [18]. Our algorithm thus can be naturally divided into three components: system identification, Bayesian belief tracking, and policy search optimization. We outline each constituent in detail below.

3.1 System Identification

The model used for learning the system dynamics is Bayesian linear regression using RFFs which can be interpreted as an approximation to GP regression [35]. Recall that all kernels can be written in the form $k(\boldsymbol{x}, \boldsymbol{y}) = \phi(\boldsymbol{x}) \cdot \phi(\boldsymbol{y})$, that is, a dot product of two possibly infinite length basis vectors $\phi(\boldsymbol{x})$ and $\phi(\boldsymbol{y})$. Hence, one strategy to approximate the kernel would be to come up with a finite length basis function $\psi(\cdot)$ such that $k(\boldsymbol{x}, \boldsymbol{y}) \approx \psi(\boldsymbol{x}) \cdot \psi(\boldsymbol{y})$. Given a shift-invariant kernel, i.e., $k(\boldsymbol{x} - \boldsymbol{y}) \triangleq k(\delta)$, and is properly scaled, then Bochner's theorem [38] guarantees that its Fourier transform is a proper probability distribution, i.e.,

$$\begin{aligned} k(\boldsymbol{x} - \boldsymbol{y}) &= \int_{\mathbb{R}^d} p(\boldsymbol{\omega}) e^{j\boldsymbol{\omega}^\top (\boldsymbol{x} - \boldsymbol{y})} d\boldsymbol{\omega} \\ &= \mathbb{E}_{\boldsymbol{\omega}}[z(\boldsymbol{x}) z(\boldsymbol{y})], \end{aligned}$$

where $z(\boldsymbol{x}) = \sqrt{2}\cos(\boldsymbol{\omega}^\top \boldsymbol{x} + b)$, $\boldsymbol{\omega}$ is drawn from $p(\boldsymbol{\omega})$ and b is drawn from $\text{unif}(0, 2\pi)$. Hence, the shift-invariant kernel can be approximated via a simple summation average,

$$\begin{aligned} \mathbb{E}_{\boldsymbol{\omega}}[z(\boldsymbol{x}) z(\boldsymbol{y})] &\approx \frac{2}{D} \sum_{i=0}^{D-1} \cos(\boldsymbol{\omega}_i^\top \boldsymbol{x} + b_i) \cos(\boldsymbol{\omega}_i^\top \boldsymbol{y} + b_i) \\ &= \psi(\boldsymbol{x}) \cdot \psi(\boldsymbol{y}), \end{aligned}$$

where $\psi(\boldsymbol{x}) = \sqrt{\frac{2}{D}}[\cos(\boldsymbol{\omega}_0^\top \boldsymbol{x} + b_0), \ldots, \cos(\boldsymbol{\omega}_{D-1}^\top \boldsymbol{x} + b_{D-1})]$ and D is the number of samples used in the approximation. We arrive at a simple yet powerful algorithm that allows us to approximate a GP by projecting the input into a feature space with a random matrix sampled from $p(\boldsymbol{\omega})$ then passing that result through a cosine function. It turns out that the Fourier transform of a Gaussian is a Gaussian and hence we can show that a GP with a SE kernel can be approximated by the basis function

$$\psi_{\text{SE}}(\boldsymbol{x}) = \sigma_f \sqrt{\frac{2}{D}} \Big[\cos\Big(\frac{\boldsymbol{\omega}_0^\top \boldsymbol{x}}{\ell} + b_0\Big), \ldots, \cos\Big(\frac{\boldsymbol{\omega}_{D-1}^\top \boldsymbol{x}}{\ell} + b_{D-1}\Big)\Big],$$

where $\boldsymbol{\omega}_i \sim \mathcal{N}(\boldsymbol{0}, I)$, $b_j \sim \text{unif}(0, 1)$ for all i, j and σ_f and ℓ are respectively, the signal amplitude and length-scale parameters of the GP.

Next, we detail how f, that is, the function that maps state action pairs to next state, $\boldsymbol{s}_{t+1} = f(\boldsymbol{s}_t, \boldsymbol{a}_t)$ is learned using Bayesian linear regression (BLR) with RFF basis functions. Consider a set of M experience tuples, $\{(\boldsymbol{s}_i, \boldsymbol{a}_i, \boldsymbol{s}_i') | i = 0, \ldots, M-1)\}$ where $\boldsymbol{s}'$ is the next state resulting from taking action $\boldsymbol{a}$ in state $\boldsymbol{s}$. We take a simple approach

where we learn a separate BLR model for every state dimension. Define $\boldsymbol{x}_i \triangleq [\boldsymbol{s}_i^\top, \boldsymbol{a}_i^\top]^\top$ and $y_i \triangleq \boldsymbol{s}'_{i,j}$ where $\boldsymbol{x}_i$ is the result of concatenating the state and action vector and y_i is the jth dimensional output of next state $\boldsymbol{s}'_i$. Thus for every output dimension, we have a training set $\mathcal{D} = \{(\boldsymbol{x}_i, y_i) | i = 0, \ldots, M-1\}$ and the design matrix $\boldsymbol{X}$ and column vector $\boldsymbol{y}$ are as defined in Section 2.3, but with the slight difference that $\boldsymbol{X}^\top \boldsymbol{X}$ is constructed after each input $\boldsymbol{x}_i$ is transformed via RFF basis $\psi(\cdot)$, i.e., $\boldsymbol{X}^\top \boldsymbol{X} = \sum_{i=0}^{M-1} \psi(\boldsymbol{x}_i)\psi(\boldsymbol{x}_i)^\top$. Given training set $\mathcal{D}$ and with this notation redefinition, we may proceed like we did before and thus, the posterior is a Gaussian distribution with covariance

$$\boldsymbol{V}_N = \sigma^2 \Big(\frac{\sigma^2}{\tau^2} I + \boldsymbol{X}^\top \boldsymbol{X} \Big)^{-1}$$

where we assume $\boldsymbol{V}_0 = \tau^2 I$ and mean

$$\boldsymbol{w}_N = \frac{1}{\sigma^2} \boldsymbol{V}_N \boldsymbol{X}^\top \boldsymbol{y},$$

and given a test point $\boldsymbol{x}_*$, the posterior predictive distribution is also given by a Gaussian distribution, but with variance

$$\sigma_N^2(\boldsymbol{x}_*) = \sigma^2 + \boldsymbol{x}_*^\top \boldsymbol{V}_N \boldsymbol{x}_*$$

and mean

$$\mu_N(\boldsymbol{x}_*) = \boldsymbol{x}_*^\top \boldsymbol{w}_N.$$

The hyperparameters of the model, that is, σ, σ_f and ℓ are chosen by maximizing the log marginal likelihood, which for BLR, is given by the expression

$$\log p(\boldsymbol{y}|\boldsymbol{X}) = -\frac{N}{2}\log(2\pi) - \frac{N}{2}\log(\sigma^2) - \frac{1}{2}\log\left(\frac{|\boldsymbol{V}_0|}{|\boldsymbol{V}_N|}\right) - \frac{\boldsymbol{y}^\top \boldsymbol{y}}{2\sigma^2} + \frac{1}{2}\boldsymbol{w}_N^\top \boldsymbol{V}_N \boldsymbol{w}_N.$$

With this formulation, we now have a probabilistic model that yields one-step predictions of the state dynamics.

We take a moment to touch upon the pros and cons in opting for an approximate model of the GP. One of the most important ingredient to both the GP and BLR model is the covariance matrix. For the GP, the covariance matrix takes the form $\text{var}(f_*)$ (Equation 2.12) and we take note that computing this result requires the inversion of the gram matrix (i.e., $k(\boldsymbol{X}, \boldsymbol{X})$). This matrix grows quadratically with respect to the amount of training data, hence making it unscalable when naively applied to domains with vast amounts of training data (i.e., an online learning setting). On the other hand, the covariance matrix of the BLR model is given by the $\boldsymbol{V}_N$ which in this case, requires the inversion of the matrix

$\boldsymbol{X}^\top \boldsymbol{X}$. Under this formulation, however, the size of the matrix $\boldsymbol{X}^\top \boldsymbol{X}$ does not grow with respect to the training set and thus is more applicable under large training sets.

Given $\boldsymbol{x}_t$, the BLR model maps a vector to a distribution over all the possible values that the next state $\boldsymbol{s}_{t+1}$ may take on. In order to predict future states thereafter, i.e. $\boldsymbol{s}_{t+2}$, we would have to calculate a marginal distribution that averages over all the possible input values over the output distribution. In general, the exact computation of this resulting distribution is intractable. The PILCO algorithm [10] circumvents this problem by approximating the intractable distribution using *moment-matching*, but in this work, we take a simpler approach by sampling P states, propagating the predicted states through the model and approximating the resulting distribution via an average of the P point evaluations.

We close this section by remarking that it is possible and computationally efficient to formulate the model so that it can have multi-output (i.e., an output for each state dimension). In this case, the covariance matrix $\boldsymbol{V}_N$ will remain unchanged and will be shared across each output dimension. The mean will differ slightly where the vector $\boldsymbol{y}$ will be replaced by the matrix $\boldsymbol{Y}$, i.e., $\boldsymbol{w}_N = \frac{1}{\sigma^2}\boldsymbol{V}_N \boldsymbol{X}^\top \boldsymbol{Y}$. The obvious computational benefit arises from the fact that since $\boldsymbol{V}_N$ is shared across the output dimensions, the prediction of the next state would call for only a single inversion of the covariance matrix as opposed to one for each output dimension for the case where we have a BLR model for each output dimension. Sharing the covariance matrix across all output dimensions also has drawbacks, since the covariance matrix encodes the uncertainty in our predictions, having a single covariance matrix would imply that there would also only be a single uncertainty value for all output dimensions. This is especially an issue if the smoothness of the state dimensions vary greatly between the dimensions. The hyperparameters of the BLR dictate the smoothness of the model and hence if a single set of hyperparameters is responsible for all of the output dimensions, then if would be difficult to find hyperparameter values that will provide a good fit for all of the output dimensions.

3.2 Policy Search Optimization

We detail our policy improvement algorithm in this section. The policy $\pi_\theta(\cdot)$ is parameterized as a neural network with parameters θ. Given P samples of the initial state distribution

$\{\boldsymbol{s}_{0,p}|p = 0, \ldots, P-1\}$ and a time horizon of T, we wish to solve the optimization problem

$$\begin{aligned} \underset{\theta}{\text{minimize}} \quad & L(\theta) = -\frac{1}{P}\sum_{p=0}^{P-1}\sum_{t=t'}^{T-1}\gamma^t R(\boldsymbol{s}_{t,p}, \pi_\theta(\boldsymbol{s}_{t,p})) \\ \text{subject to} \quad & \boldsymbol{s}_{t+1,p} \sim f(\boldsymbol{s}_{t,p}, \pi_\theta(\boldsymbol{s}_{t,p})),\ t = t', \ldots, T-1,\ p = 0, \ldots, P-1, \end{aligned} \tag{3.1}$$

where $\boldsymbol{s}_{t+1,p}$ is sampled from f, our probabilistic system dynamics model given by our regression model described above. The reward function $R(\cdot, \cdot)$ may be given, but if it is not, then it may be learned in a similar fashion described in Section 3.1 which is what was done for some of our experiments.

One of the obvious strategies to optimize 3.1 is to use gradient descent methods to minimize $L(\theta)$. However, preliminary experiments have demonstrated that under certain circumstances, gradient-based methods may not at all be able to optimize our objective. A salient example involves the classical control problem known as the Mountain car problem [30] (i.e., `MountainCarContinuous-v0` on OpenAI Gym [8]). The environment exhibits a sparse reward structure in that under the formulation provided by OpenAI Gym, the agent receives a slight punishment for exerting actions and gets a large reward for reaching the goal (i.e., the top of the mountain). Put it another way, the reward function of the environment can effectively be thought of as step function with respect to the position of the cart. Thus, we would expect gradient-based optimizers to not work in this environment because the loss landscape contains zero gradients everywhere. In this work, we opt instead for a black-box optimizer in the form of CMA-ES [18] for this reason, but also for the purpose of Bayesian belief tracking which we will elaborate in the section below.

3.3 Bayesian Belief Tracking

Different from other work in model-based RL we attempt to perform directed exploration by incorporating Bayesian belief tracking in our policy search algorithm. Our approach follows that of BAMDPs which was briefly covered in Section 2.2 and we expand on this in the following section. If one is willing to adopt a Bayesian perspective, then the problem of exploration-exploitation can be settled. The formulation of policies that optimally balance the exploration-exploitation trade-off, known as *Bayes-optimal policies* begins with abstracting the notion of "state" to be an ordered pair tuple (s, x) consisting of the "physical state" s and the information state "information state" x. One may think of the physical state as the traditional notion of state in that it describes the current condition and circumstance of the agent and the information state as the set of parameters that summarizes the entire observation history of the agent.

As a concrete example (adopted from [12]), consider the problem where a medical researcher is presented with two types of drugs and twenty critically ill patients. The task is to administer the drugs to the patients so as to maximize the number of survivors. It is not hard to see that there is an exploration-exploitation component to this problem: we must spend actions to explore which of the drugs is more promising and at the same time, commit to exploiting the promising drug once we have discovered this drug. In the Bayesian approach, we would model the success and failure rate of each drug with a probability distribution and upon observing a new outcome, we would update our distribution via Bayes' rule. Due to the nature of this problem where the action is discrete and the outcome set is binary, an appropriate distribution to use for modelling the success and failure of each drug would be the *Beta distribution* with its PDF given by

$$p(x|\alpha,\beta) = \frac{x^{\alpha-1}(1-x)^{\beta-1}}{B(\alpha,\beta)} \text{ where } B(\alpha,\beta) = \frac{\Gamma(\alpha)\Gamma(\beta)}{\Gamma(\alpha+\beta)}.$$

Roughly, one may think of the Beta distribution as a model for an experiment with a binary output where the two parameters α and β parameterize the two outcomes, where in the context of our problem, would be the success and failure rate of a drug. The mean of a random variable X with a Beta distribution is given by $E[X] = \frac{\alpha}{\alpha+\beta}$. One can show that upon receiving a new observation, the posterior update of a Beta distribution simply amounts to incrementing the corresponding α or β parameter associated with the new data point. In the drug allocation example, α_1 may represent the number of times that drug 1 has succeeded and β_1 may represent the number of times that drug 1 has failed (subscripts of α and β are used to indicate that they correspond to drug 1). Upon administering the drug to a new patient and making the observation that the trial was a success, then the posterior PDF and mean are respectively given by $p(x|\alpha_1+1,\beta_1)$ and $\frac{\alpha_1+1}{\alpha_1+1+\beta_1}$. The information state corresponds to the tuple consisting of α_1 and β_1, i.e., $x_1 = (\alpha_1,\beta_1)$. Under this framework, the Bellman equation gets augmented with the information state allowing uncertainty to be factored into the calculation of the value function. Abstractly, one could imagine that this amounts to performing tree search to find a set of actions that maximize the return averaged over the model uncertainty subject to the hyperstate evolution at every tree depth. Figure 3.1 depicts the posterior evolution of the Beta distribution given a sequence of administered drugs and their corresponding successes/failures. On the leftmost path, for example, where in the hypothetical scenario drug 1 always succeeds, then we can see that the model gets updated in such a way that the distribution heavily skews towards 1 (i.e., it is becoming more confident that drug 1 has a high probability of success). Consider Bellman's equation in classic RL given by

$$V(s) = \max_a \sum_{s'} p(s'|s,a)(R(s,a,s') + \gamma V(s')),$$

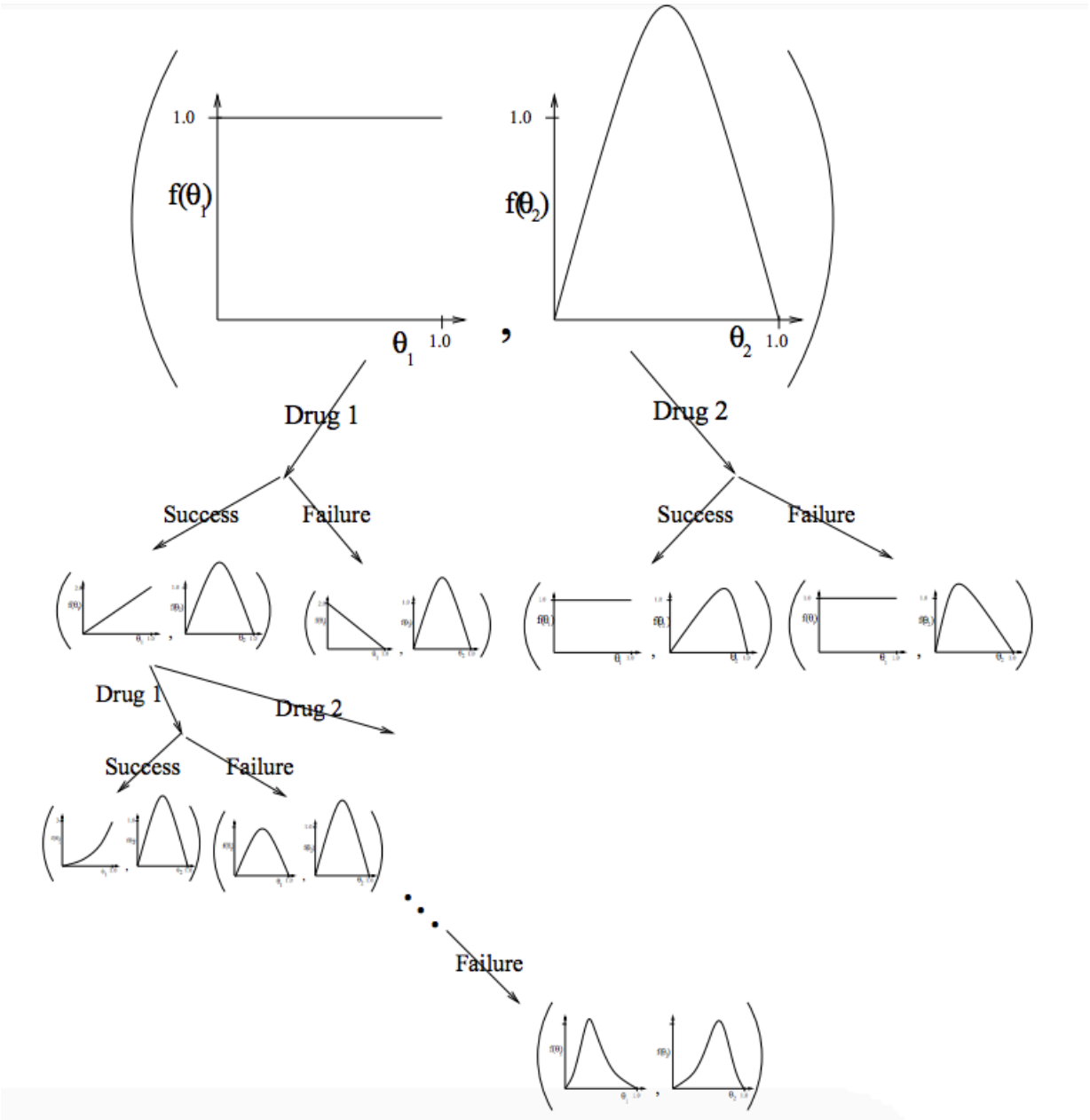


Figure 3.1: An information-state transition diagram for the drug allocation problem (adopted from [12]).

augmenting this equation with an information state and assuming deterministic posterior updates, we obtain

$$V(s,x) = \max_a \sum_{s'} p(s'|s,a,x)(R(s,a,s') + \gamma V(s', x_{s,a,r,s'}))$$

where $x_{s,a,r,s'}$ is the posterior information update after observing transition tuple (s,a,r,s'). In summary, $p(s'|s,a,x)$ corresponds to the depicted distributions in Figure 3.1 and tracking the evolution of the information state allows us to model the information gain when making a hypothetical action. When these distributions are factored into the calculation of the value function, then the Bellman backups at every time step will be averaged over the uncertainties of the model, yielding values that naturally balance exploration and exploitation.

In this work, we extend the ideas from BAMDPs to the problem domain of continuous states and actions. As aforementioned, the one step state dynamics is modelled using BLR with RFF basis functions, i.e., $p(\boldsymbol{s}'|\boldsymbol{s},\boldsymbol{a}) = \mathcal{N}(\boldsymbol{s}'|\boldsymbol{\mu},\boldsymbol{\Sigma})$, and extrapolating from the drug allocation problem, the information state corresponds to the tuple $x = (\boldsymbol{\mu},\boldsymbol{\Sigma})$. Letting $\boldsymbol{a}_t = \pi_\theta(\boldsymbol{s}_t)$, then

$$\begin{aligned} V(\boldsymbol{s}_t, x_t, \theta) &= \max_\theta \int_{\boldsymbol{s}_{t+1}} p(\boldsymbol{s}_{t+1}|\boldsymbol{s}_t, \pi_\theta(\boldsymbol{s}_t, x_t), x_t)[R(\boldsymbol{s}_t, \pi_\theta(\boldsymbol{s}_t, x_t), \boldsymbol{s}_{t+1}) \\ &\quad + \gamma V(\boldsymbol{s}_{t+1}, x_{t+1})]d\boldsymbol{s}_{t+1} \\ &\approx \mathbb{E}\Big[\sum_{\tau=t}^{t+T-1} \gamma^\tau R(\boldsymbol{s}_\tau, \pi_\theta(\boldsymbol{s}_\tau, x_\tau))\Big] \\ &\approx \frac{1}{P}\sum_{p=0}^{P-1}\sum_{\tau=t}^{t+T-1} \gamma^\tau R(\boldsymbol{s}_{\tau,p}, \pi_\theta(\boldsymbol{s}_{\tau,p}, x_{\tau,p})). \end{aligned} \tag{3.2}$$

To arrive at our final optimization objective, we first make two approximations to our augmented Bellman's equation. In the first approximation, we unroll Bellman's equation up to only T time steps into the future due to computational restrictions. Given that the state space is continuous, exact inference calls for integrating over the value function with respect to the next state which in general, is intractable and hence, the second approximation amounts to a Monte Carlo estimation of this integral. After these two approximations, we

arrive at our final objective

$$\begin{aligned} \underset{\theta}{\text{minimize}} \quad & L(\theta) = -\frac{1}{P}\sum_{p=0}^{P-1}\sum_{\tau=t}^{t+T-1} \gamma^{\tau} R(\boldsymbol{s}_{\tau,p}, \pi_\theta(\boldsymbol{s}_{\tau,p}, x_{\tau,p})) \\ \text{subject to} \quad & \boldsymbol{s}_{\tau+1,p} \sim f(\boldsymbol{s}_{\tau,p}, \pi_\theta(\boldsymbol{s}_{\tau,p}, x_{\tau,p})), \\ & x_{\tau+1,p} = \text{BayesUpdate}(x_{\tau,p}, \boldsymbol{s}_{\tau,p}, \pi_\theta(\boldsymbol{s}_{\tau,p}, x_{\tau,p}), \boldsymbol{s}_{\tau+1,p}), \\ & \tau = t, \ldots, t+T-1, \\ & p = 0, \ldots, P-1, \end{aligned} \tag{3.3}$$

where $x_{\tau+1,p}$ is the posterior information update after observing transition tuple $(\boldsymbol{s}_{\tau,p}, \pi_\theta(\boldsymbol{s}_{\tau,p}, x_{\tau,p}), R(\boldsymbol{s}_{\tau,p}, \pi_\theta(\boldsymbol{s}_{\tau,p}, x_{\tau,p})), \boldsymbol{s}_{\tau+1,p})$, f is the probabilistic dynamics model and P is the number of Monte Carlo samples used to approximate the integral.

We close this section by making a few remarks. The optimization problem in Equation 3.3 requires the calculation of the posterior distribution after observing a hypothetical tuple and this further motivates the use of a BLR model and a black-box optimizer. BLR provides a easy way to calculate the posterior, that is, after observing a new data tuple $(\boldsymbol{x}, y)$ calculating the posterior simply amounts to making a rank-one update to the covariance and cross-covariance matrices, i.e., $\boldsymbol{X}^\top\boldsymbol{X} \leftarrow \boldsymbol{X}^\top\boldsymbol{X} + \boldsymbol{x}^\top\boldsymbol{x}$ and $\boldsymbol{X}^\top\boldsymbol{y} \leftarrow \boldsymbol{X}^\top\boldsymbol{y} + \boldsymbol{x}^\top y$. Further, rank-one updates are operations that do not suit well with computational graphs hence gradient-based optimizers may not be applicable, motivating the need for black-box optimizers.

The policy network also takes as input the information state, allowing the policy to select actions that reflect the uncertainty of the model. In addition, the information state oftentimes takes the form of a very large matrix (e.g., if we use basis dimension of size 256, then the covariance matrix will be of size 256×256) and to reduce the size of this matrix, we use a straightforward, random projection technique [7] to embed the information state to a lower dimension.

3.4 Algorithm Overview

Algorithm 2 provides a pseudocode summary of the overall algorithm.

Algorithm 2 Bayesian Policy Search

Gather initial training data $\mathcal{D} = \{(\boldsymbol{s}_i, \boldsymbol{a}_i, r_i, \boldsymbol{s}'_i) | i = 0, \ldots, M-1)\}$
Sample initial state distribution $\boldsymbol{S}_0 = \{\boldsymbol{s}_{0,p} | p = 0, \ldots, P-1\}$
for `e` number of `epochs` **do**
    Fit approx GP models $\mathcal{G}$ with $\mathcal{D}$ to obtain distributions over $p(\boldsymbol{s}'|\boldsymbol{s}, \boldsymbol{a})$ and $p(r|\boldsymbol{s}, \boldsymbol{a})$
    Optimize 3.3 using CMA-ES with inputs $\boldsymbol{S}_0$, $\mathcal{G}$ and $\pi_\theta(\cdot)$ ▷ remembering to update the belief state at every time step
    Initialize environment and observe initial state `s`
    while epoch is not over **do**
        Take action `a` according to policy network given state `s` and belief `x`
        Observe immediate reward `r` and next state `s'`
        Cache training tuple `(s, a, r, s')`
        `s` ← `s'`
    end while
    Update $\mathcal{D}$ with new data
end for

Chapter 4

Experiments and Results

In this chapter, we describe the set of experiments that were conducted to test the efficacy of our algorithm.

4.1 Classical Control Problems

The first set of experiments were conducted on various classical continuous control tasks simulated in OpenAI Gym. In the next subsections, we first describe the set of environments used in our experiments then we detail the methodology and result of each of our experiments.

4.1.1 Environments

We briefly describe the two OpenAI Gym environments used in the first set of experiments.

`Pendulum-v0`

The first problem domain is `Pendulum-v0` and this environment is a basic control task that involves a hinged pendulum that swings up when torque is applied. The state $\boldsymbol{s} = (\cos(\theta), \sin(\theta), \dot{\theta})$ where $\theta \in [-\pi, \pi]$ is the angle of the pendulum and $\dot{\theta} \in [-8, 8]$ is the angular velocity and the action (applied torque), $a \in [-1, 1]$. The precise equation for the reward is $r_t = -(\theta_t^2 + 0.1\dot{\theta}_t^2 + 0.001a_t^2)$. Therefore, the lowest reward is $-(\pi^2 + 0.1 \times 8^2 +$

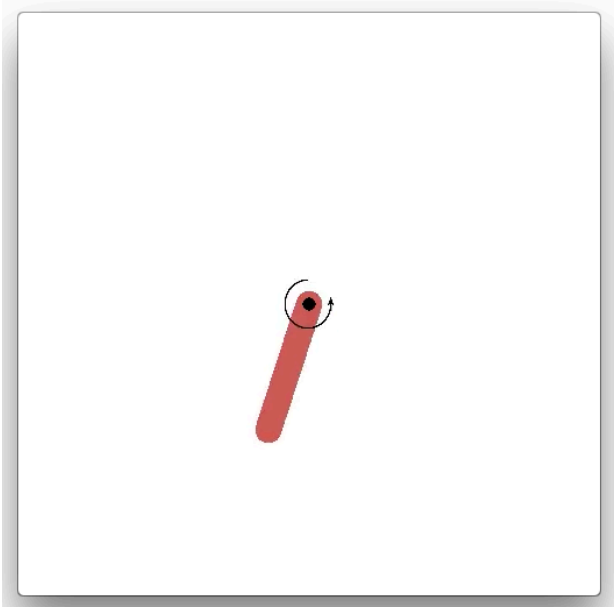


Figure 4.1: A screenshot of the `Pendulum-v0` environment.

$0.001 \times 2^2) = -16.2736044$ and the highest reward is 0. Essentially, the goal is to have the pendulum standing vertically with the least rotational velocity and the least effort. Figure 4.1 depicts a screenshot of the `Pendulum-v0` environment.

`MountainCarContinuous-v0`

The second problem domain is `MountainCarContinuous-v0` which is a variant of the classical Mountain car problem originally introduced in [30]. In this domain, an under-powered car is situated in a valley and must drive up a steep hill. Since gravity is stronger than the car even at full throttle, it must first drive backwards to leverage potential energy and then drive forward to reach the top of the hill. The state $\boldsymbol{s} = (p, v)$, where $p \in [-1.2, 0.6]$ and $v \in [-0.07, 0.07]$ is the position and velocity of the cart, respectively, and the action (applied force), $a \in [-1, 1]$. The reward is given by

$$r_t = \begin{cases} 100 - a_t^2 & \text{if } p_t \geq 0.45 \\ -a_t^2 & \text{otherwise.} \end{cases}$$

Figure 4.2 depicts a screenshot of the `MountainCarContinuous-v0` environment.

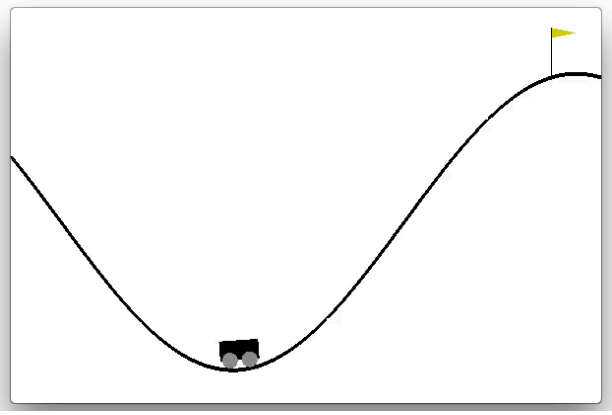


Figure 4.2: A screenshot of the `MountainCarContinuous-v0` environment.

4.1.2 Systems Dynamics Regression

We conduct a preliminary experiment as a proof of concept to evaluate the capabilities of our method. The purpose of the first experiment is to visualize and do a sanity check of the regression power of our model and this is done by performing simple tests on time series forecasting in `Pendulum-v0` and `MountainCarContinuous-v0`.

`Pendulum-v0`

We trained a BLR model with 256 SE RFF basis functions that makes one step predictions $\boldsymbol{s}_{t+1} = f(\boldsymbol{s}_t, a_t)$ using a training set consisting of three epochs of data (600 data points) and then we make a comparison between the true and predicted $\boldsymbol{s}_{t+1}$ given an input sequence $\{(\boldsymbol{s}_t, a_t)\}_{t=0}^{199}$. As depicted in Figure 4.3, the regression model is very accurate in that the predictions match the true states closely and the green errors bars are small, indicating that the model is very confident in its predictions (which are correct in this case). To add a bit more difficulty to the task, we also conducted an experiment where the model was queried to predict the sequence of states that would result from an action sequence $\{a_{t=0}^{199}\}$ starting from a given initial state $\boldsymbol{s}_0$. As illustrated in Figure 4.4, the predicted trajectories, which are sampled 50 times to visualize uncertainty, all closely match the true trajectory. Finally, we visualize the predicted reward function in Figure 4.5 and we see that the predicted reward function is also quite accurate.

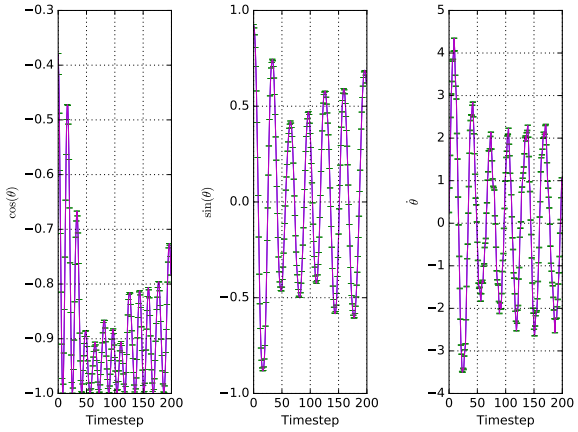


Figure 4.3: A comparison between the true and predicted $\boldsymbol{s}_{t+1}$ given an input sequence $\{(\boldsymbol{s}_t, a_t)\}_{t=0}^{199}$. The blue and purple curves are the true and predicted curves respectively. The predicted curve is accompanied by green error bars which represent one standard deviation of uncertainty. The training set consists of three epochs of data (600 data points).

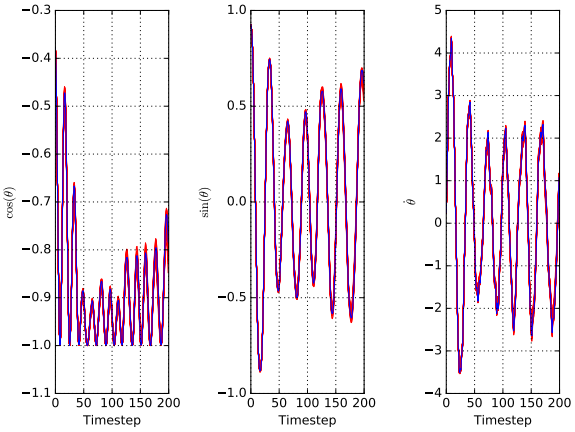


Figure 4.4: A comparison between the true trajectory and 50 samples of the predicted trajectories given a seed state $\boldsymbol{s}_0$ and action sequence $\{a_t\}_{t=0}^{199}$. The blue and red curves are the true and predicted curves, respectively. The training set consists of three epochs of data (600 data points).

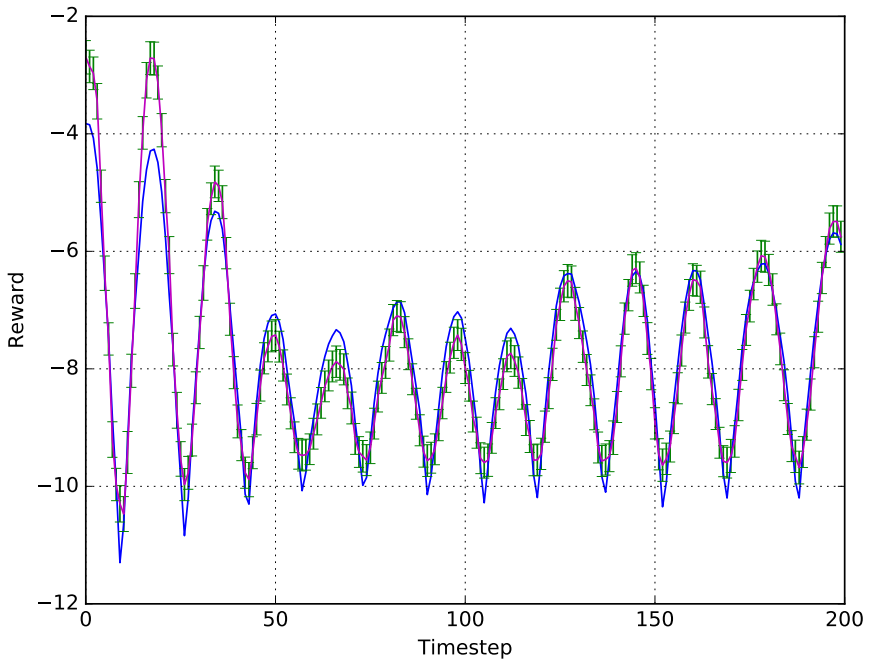


Figure 4.5: A comparison between the true and predicted rewards given an input sequence $\{(\boldsymbol{s}_t, a_t)\}_{t=0}^{199}$. The blue and purple curves are the true and predicted, respectively. The predicted curve is accompanied by green error bars which represent one standard deviation of uncertainty. Training set consists of three epochs of data (600 data points).

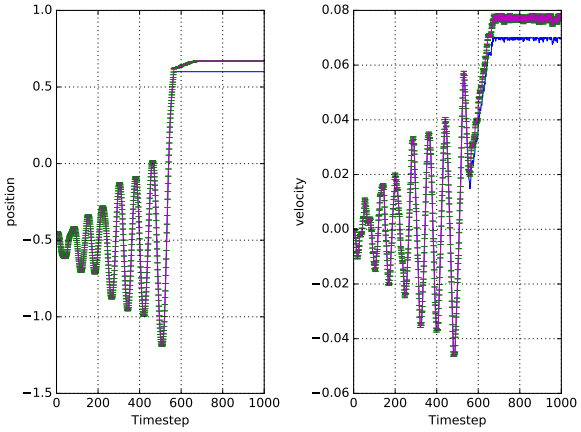


Figure 4.6: A comparison between the true and predicted $\boldsymbol{s}_{i+1}$ given an input sequence $\{(\boldsymbol{s}_t, a_t)\}_{t=0}^{998}$. The blue and purple curves are the true and predicted curves respectively. The predicted curve is accompanied by green error bars that represent one standard deviation of uncertainty. The training set consists of on epoch of data (999 data points).

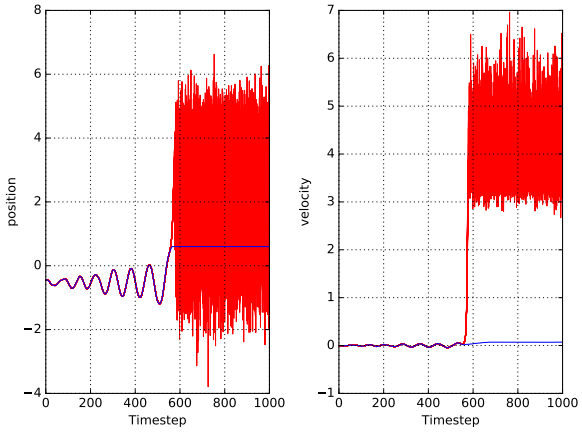


Figure 4.7: A comparison between the true trajectory and 50 samples of the predicted trajectories given a seed state $\boldsymbol{s}_0$ and action sequence $\{a_t\}_{t=0}^{998}$. The blue and red curves are the true and predicted, respectively. Training set consists of one epoch of data (999 data points).

`MountainCarContinuous-v0`

Similar experiments are performed on `MountainCarContinuous-v0` task using a BLR model with the same settings described above and a training set consisting of one epoch of data (999 data points). Analogous Figures of Figure 4.3, Figure 4.4 and Figure 4.5 are Figure 4.6, Figure 4.7 and Figure 4.8.

However, we notice that the `MountainCarContinuous-v0` domain illustrates something that `Pendulum-v0` does not. To understand this, we first realize that the goal of the agent is to reach the top of the mountain in the `MountainCarContinuous-v0` domain. The experiment was setup so that the training set *did not* have the agent reach the goal whereas the testing set *did*. Hence, we see that the model predicted an area of the state which it has never seen before, that is the goal state, with very large uncertainty. This is what is desired and expected as there is potential to exploit this feature in the model to perform directed exploration-exploitation.

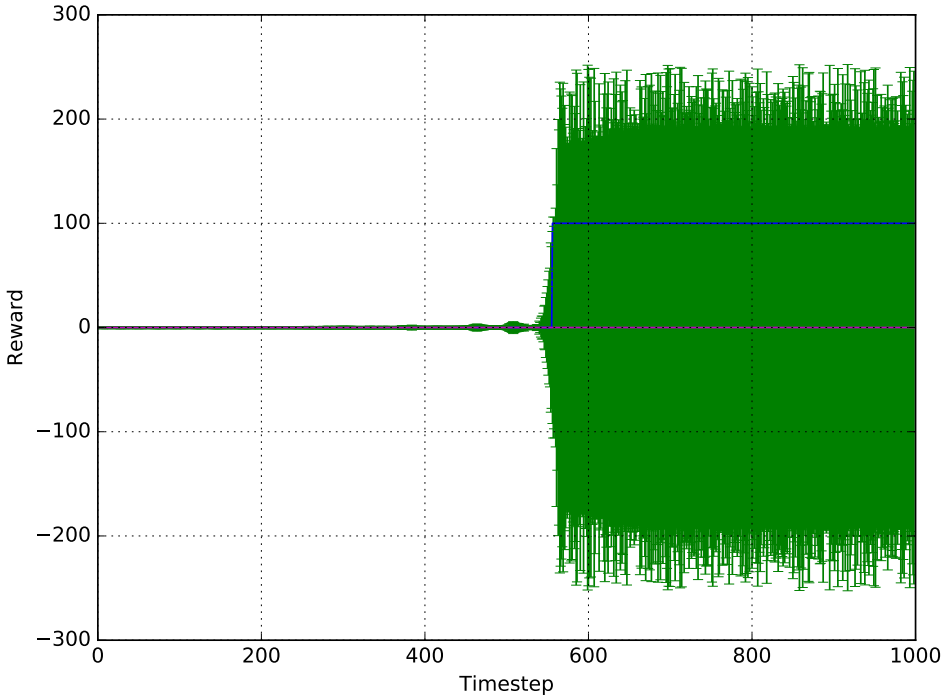


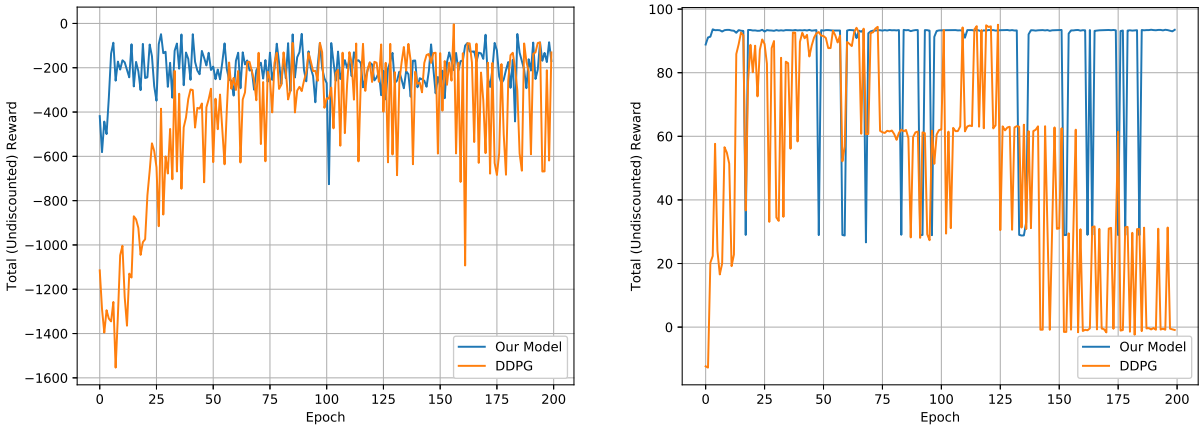
Figure 4.8: A comparison between the true and predicted rewards given an input sequence $\{(\boldsymbol{s}_i, a_t)\}_{t=0}^{998}$. The blue and purple curves are the true and predicted, respectively. The predicted curve is accompanied by green error bars which represent one standard deviation of uncertainty. Training set consists of one epoch of data (999 data points).

4.1.3 Classic Continuous Control

The next experiment involves actually learning a policy and seeing how well our controller performs. The baseline for our comparison is the deep deterministic policy gradient (DDPG) [25] method which is a pioneer model-free actor-critic algorithm for continuous states and actions and has since become a popular baseline for other algorithms that operate over a continuous state action space. We parameterize a policy network $\pi_\theta(\cdot)$ as an artificial neural network (ANN) with weights θ. Specifically, the architecture of the policy network is a feedforward ANN with an input layer, one hidden layer and an output layer and the number of hidden units is 8. All units used the $\tanh(\cdot)$ activation and the output layer was scaled so that the resultant values were clipped to match the minimum and maximum action values specified by the environment. The discount factor γ is set to 0.995 for both environments, the unroll steps T are 35 and 120 and the number of Monte Carlo trajectories are 30 and 7 for `Pendulum-v0` and `MountainCarContinuous-v0`, respectively. For `Pendulum-v0` we trained a BLR model with 50 SE RFF basis functions for the system dynamics and another BLR model with 150 SE RFF basis functions for the reward dynamics while for `MountainCarContinuous-v0` we trained a BLR model with 50 SE RFF basis functions for the system dynamics and another BLR model with 50 SE RFF basis functions for the reward dynamics. The BLR model for the system dynamics remain unchanged from the model used in Section 4.1.2. The training curves comparing our model to the DDPG baseline for `Pendulum-v0` and `MountainCarContinuous-v0` are depicted in Figure 4.9(a) and Figure 4.9(b), respectively. Each curve is generated from averaging the results of three trials of the experiment. These results corroborate the hypothesis that our model is much more sample efficient in that it was already able to learn a near optimal policy during the initial epochs.

4.2 `pybullet` Environments

We attempted to scale our algorithm by testing it on environments involving simulated high-dimensional contact robotics such as the tasks found in the MuJoCo benchmark system [45]. However, due to license restrictions, we opted instead for the `pybullet` robotics reinforcement learning environments which is an open-source competitor to MuJoCo for simulating contact robotics tasks for RL. The `pybullet` suite provides a rich set of locomotion tasks including minitaur robot, humanoid walking, ant walking, hopper, cheetah running and walker (`MinitaurBulletEnv-v0`, `HumanoidBulletEnv-v0`, `AntBulletEnv-v0`, `HopperBulletEnv-v0`, `HalfCheetahBulletEnv-v0` and `Walker2DBulletEnv-v0`). Figure



(a) (b)

Figure 4.9: A comparison of learning curves between our model and DDPG for (a) `Pendulum-v0` and (b) `MountainCarContinuous-v0`.

4.12 depicts screenshots of a few of these environments. However, preliminary experiments suggest that our Gaussian process regression was unable to learn the contact discontinuities involved in such robotics environments. In similar vein to Figures 4.3 and 4.4 and Figures 4.6 and 4.7, Figure 4.10 and Figure 4.11 depicts the regression of the next state given state-action and the predicted trajectories given a seeding state, respectively for one particular state dimension of the `AntBulletEnv-v0` environment. These two figures highlight the difficulties in learning contact dynamics as the true underlying function exhibit characteristics of a square function but our regression model fails to learn these "jumps" in the data under the smoothness assumptions made by the GP model. As a result, we observe from the plots that our regression model makes predictions that are inaccurate along with high uncertainty in many regions. Unsurprisingly, it produces trajectories that do not resemble any coherent pattern.

This shortcoming unfortunately limits the applicability of our regression model to environments with dynamics that exhibit smooth transitions and in the suite of `pybullet` environments, only two pendulum-related tasks, `InvertedPendulumBulletEnv-v0` and `InvertedDoublePendulumBulletEnv-v0`, satisfy this criterion, making them the only tasks considered in this experiment. `InvertedPendulumBulletEnv-v0` is a classical pole on a cart task where the pole is initially standing upright on the cart with state $\boldsymbol{s} \in \mathbb{R}^5$ and the goal of the controller is to apply a force $a \in [-1, 1]$ so as to counteract the gravitational force to keep the pole upright. In this instance, the environment is formulated in such a way that

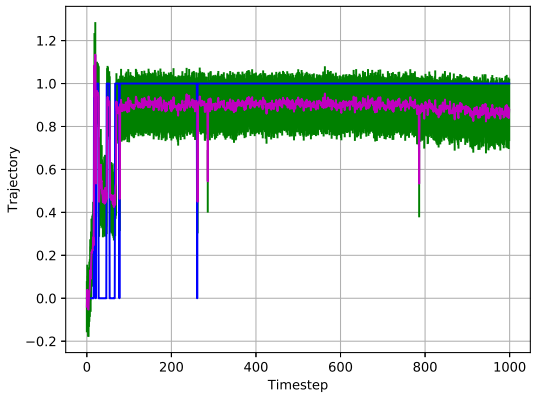


Figure 4.10: A comparison between the true and predicted $\boldsymbol{s}_{t+1}$ given an input sequence $\{(\boldsymbol{s}_t, a_t)\}_{t=0}^{199}$. The blue and purple curves are the true and predicted curves respectively. The predicted curve is accompanied by green error bars which represent one standard deviation of uncertainty.

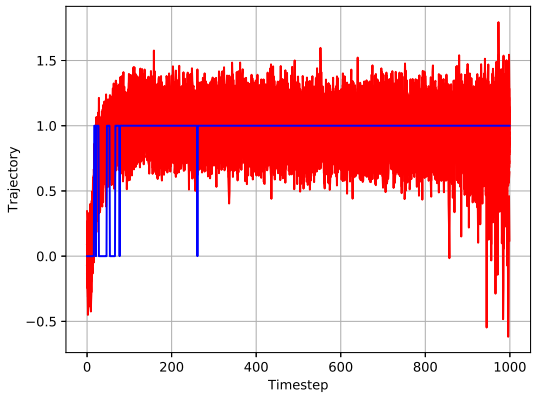
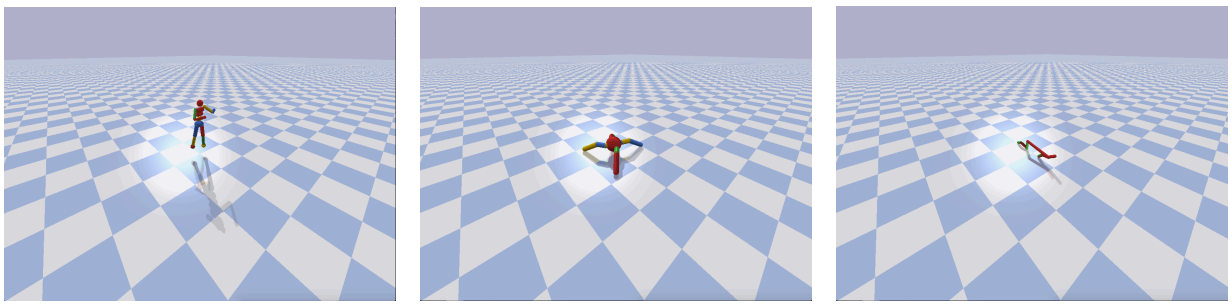


Figure 4.11: A comparison between the true trajectory and 50 samples of the predicted trajectories given a seed state $\boldsymbol{s}_0$ and action sequence $\{a_t\}_{t=0}^{199}$. The blue and red curves are the true and predicted curves, respectively.

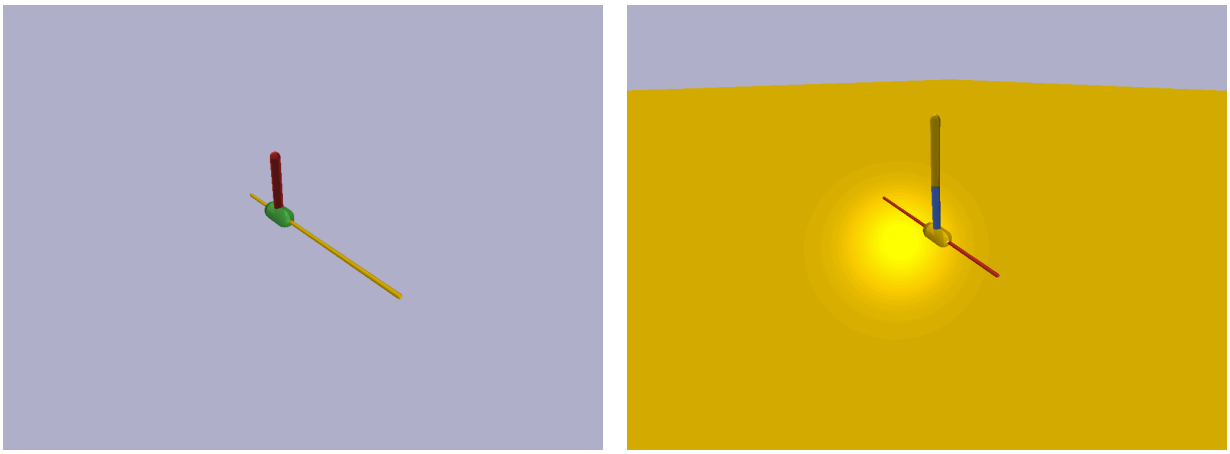


(a) (b) (c)

Figure 4.12: Screenshot of (a) `HumanoidBulletEnv-v0`, (b) `AntBulletEnv-v0`, and (c) `HalfCheetahBulletEnv-v0`.

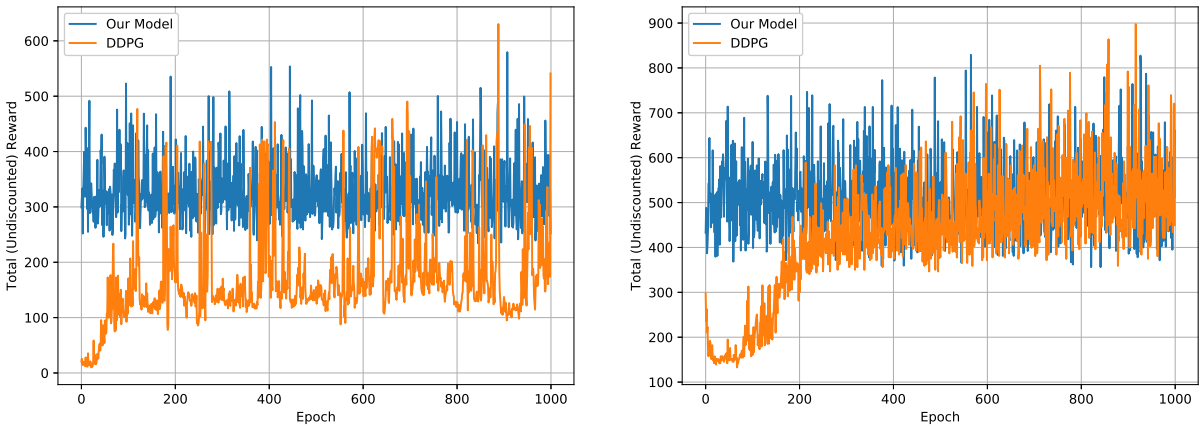
the rail with which the cart runs on has finite length with walls existing at both terminals and we note that the presence of these walls introduce discontinuities in the system dynamics if the controller drives the cart to the edge of the rail. The episode ends when the pole falls below a certain angle measured from the upright position or when it has elapsed a maximum number of time steps and the agent receives a reward of one at every time step until the episode terminates (i.e., the agent is incentivized to keep the episode alive for as long as possible). We point out that the reward function as described is problematic and does not lead to convergence when we apply our algorithm. The reason being is that our model-based method has no notion of end-of-episode but rather it only attempts to maximize the expected discounted sum of rewards up to some horizon T under the assumption that the episode will always remain alive for the next T time steps. This is in contrast to model-free algorithms where, for example, in the Q-learning algorithm, end-of-episode is implied when the unrolled next state action value is zero (i.e., if T is the time step where the episode terminates, then $Q(\boldsymbol{s}_T, \boldsymbol{a}_T) = R(\boldsymbol{s}_T, \boldsymbol{a}_T)$ implies end-of-episode). This lack of notion of end-of-episode in our model-based method coupled with a flat reward function (i.e., the controller always observes a reward of one) will fool our model in predicting the same loss for all trajectories leading to a flat fitness function with no policy improvement in our policy search algorithm. This is rectified via reward shaping where the reward function is instead set to be the cosine of the angle of the pole measured from the upright position thereby providing the controller with a difference between the "goodness" of various states resulting in a loss function with a varying landscape upon which the policy algorithm can optimize. As the name suggests, the `InvertedDoublePendulumBulletEnv-v0` environment is a more advanced version of `InvertedPendulumBulletEnv-v0` where the pendulum attached to the cart is hinged midway creating a "double" pendulum whilst the rest of the dynamics remain same with `InvertedPendulumBulletEnv-v0`. The double pendulum environment does not require reward shaping as the reward function is already a function of the pendulum angles. Figure 4.13 depicts a screenshot of the `InvertedPendulumBulletEnv-v0` and `InvertedDoublePendulumBulletEnv-v0` environment.

$\boldsymbol{s}_{\mathrm{I}} \in \mathbb{R}^5$ and $T_{\mathrm{I}} = 150$ where $\boldsymbol{s}_{\mathrm{I}}$ and T_{I} are respectively the state vector and number of unroll steps for `InvertedPendulumBulletEnv-v0` and similarly, $\boldsymbol{s}_{\mathrm{ID}} \in \mathbb{R}^9$ and $T_{\mathrm{ID}} = 150$ where $\boldsymbol{s}_{\mathrm{ID}}$ and T_{ID} are respectively the state vector and number of unroll steps for `InvertedDoublePendulumBulletEnv-v0`. The number of Monte Carlo samples is 7 for both environments and the discount factor as well as the specifications of the policy network remain unchanged from the experiments in Section 4.1.3. The state dimensions of the environments were large enough that we opted for multi-output regression model to speed up the computation time at the expense of having the same uncertainty across each output dimension. We noticed that our state and reward dynamics model saturated



(a) (b)

Figure 4.13: Screenshot of (a) `InvertedPendulumBulletEnv-v0`, and (b) `InvertedDoublePendulumBulletEnv-v0`.



(a) (b)

Figure 4.14: A comparison between my model configured with multi-output regression and DDPG for (a) `InvertedPendulumBulletEnv-v0` and (b) `InvertedDoublePendulumBulletEnv-v0`.

quickly and a near-optimal policy was learnt in the first few epochs of interaction of the environment with negligible policy improvements for epochs thereafter. Hence, in this experiment we use the first few epochs for training and the remaining to just interact with the environment. The training curves comparing our model to the DDPG baseline for `InvertedPendulumBulletEnv-v0` and `InvertedDoublePendulumBulletEnv-v0` are depicted in Figure 4.14(a) and Figure 4.14(b), respectively. The plots provide some corroboration for our hypothesis that our model-based algorithm is more sample efficient than the DDPG baseline in that our model was able to leverage the smoothness assumption of the system dynamics and learn a policy with high return at the initial epochs whereas it took DDPG to reach and saturate to the same performance at approximately epoch 200 for `InvertedDoublePendulumBulletEnv-v0` while `InvertedPendulumBulletEnv-v0` never quite reached the same performance of our model with the same stability.

The main limitation to our method is the computationally complexity of the algorithm. This computational complexity is high because of the following reasons:

1) we are constrained to use for loops to compute the prediction for every Monte Carlo sample as opposed to the more efficient numerical "broadcasting" operation due to the posterior updates enforcing the covariance matrices to be different across Monte Carlo samples and

2) we compute the posterior distribution for the system and potentially reward dynamics model.

Note that we incur the inefficiencies of both 1) and 2) for each Monte Carlo sample and time step unroll in each fitness evaluation. Let us attempt to quantify the inefficiencies for both 1) and 2) and assume that we are modelling only the system dynamics and that we are using a basis function of size B. Consider the first point 1), where we are making a prediction and we assume that prediction is dominated by the complexity of matrix multiplication which is $O(B^3)$. Assuming that we use S Monte Carlo samples and an unroll horizon of T time steps, then the algorithm will make a total of ST predictions resulting in a complexity of $O(STB^3)$. This is as opposed to a complexity of $O(TB^3)$ without incurring the need to perform a separate matrix multiplication across each Monte Carlo sample (due to an unaltered covariance matrix) and without quantifying the computational efficiency gained in numerical broadcasting. The second point 2), calls for computing the posterior for an additional data point and two of the most computationally fast methods are rank-one Cholesky updates and Sherman-Morrison formula as both have a complexity of $O(B^2)$ and we opt for the former as it exhibited better numerical stability in our experiments. Posterior updates were necessary for each Monte Carlo sample and time step, hence we

have a total complexity of $O(STB^2)$ for the second point. Therefore, the computation complexity for a single evaluation of the fitness function is $O(STB^3 + STB^2)$ if we perform Bayesian belief tracking as compared to a computational complexity of $O(TB^3)$ if we do not. Note that the computational complexity would be magnified if we opt for a separate regression model for each output dimension.

4.3 Comparisons to PILCO and deepPILCO

In the last set of experiments, we attempt to compare our model against PILCO and deepPILCO. We use an open-source implementation of PILCO and deepPILCO which constitutes a component of the code repository, entitled `kusanagi`, released for the experiments conducted in [19]. PILCO and deepPILCO were designed to have access to the cost function (as opposed to samples from the reward function in the case of traditional RL) and for example, consider the cost

$$c(\boldsymbol{x}) = 1 - e^{\frac{\|\boldsymbol{x}-\boldsymbol{x}_{\text{target}}\|^2}{\sigma_c^2}} \tag{4.1}$$

which is the squared exponential subtracted from unity, where $\boldsymbol{x}$ is the current location of the controller, $\boldsymbol{x}_{\text{target}}$ is the target location and σ_c is a parameter that controls the width of the cost. The motivating choice for cost 4.1 is that it allows the expected cost at each time step, that is,

$$\mathbb{E}_{\boldsymbol{x}_t}[c(\boldsymbol{x}_t)] = \int c(\boldsymbol{x}_t)\mathcal{N}(\boldsymbol{x}_t|\mu_t, \boldsymbol{\Sigma}_t)d\boldsymbol{x}_t$$

to be calculated in closed form. In light of these requirements, the `Pendulum-v0` and `MountainCarContinuous-v0` environments were modified so that the instead of returning reward samples, the target goals of both environments were provided to construct the cost function as specified in Equation 4.1 and were provided to PILCO and deepPILCO. The settings for our model and DDPG remain unchanged from the set of experiments specified in Section 4.1.3 and the default settings from the `kusanagi` package were used for PILCO and deepPILCO. The training curves comparing our model, DDPG, PILCO and deepPILCO for `Pendulum-v0` and `MountainCarContinuous-v0` are depicted in Figure 4.15(a) and Figure 4.15(b), respectively. Each curve is generated from averaging the results of three trials of the experiment. As depicted in the figures, our model outperforms PILCO and deepPILCO for both environments despite the additional domain knowledge that is supplied to PILCO and deepPILCO. We suspect PILCO and deepPILCO's shortcoming is potentially due to the model's difficulty in environments with long time horizons. Indeed,

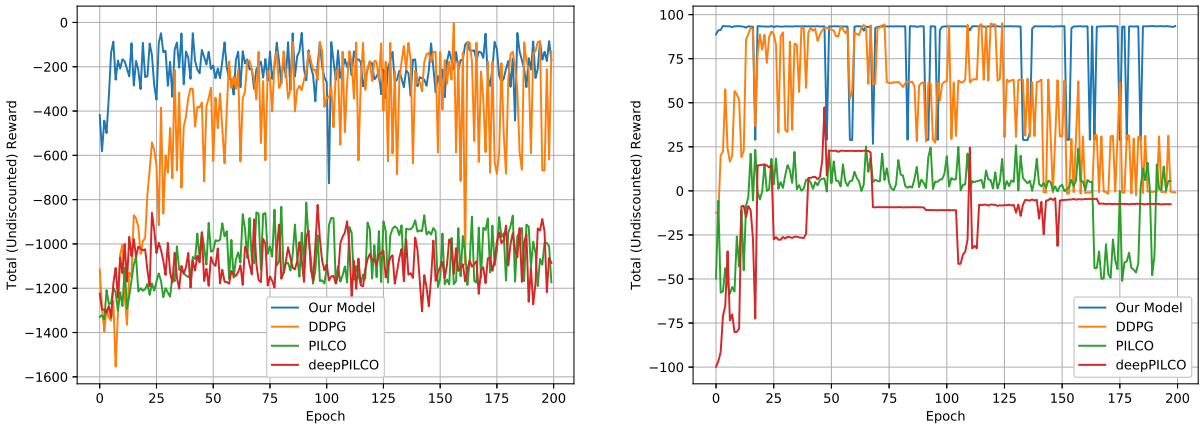


Figure 4.15: A comparison of learning curves between our model, DDPG, PILCO and deepPILCO for (a) `Pendulum-v0` and (b) `MountainCarContinuous-v0`.

the test-bed environments for PILCO and deepPILCO (i.e., the environments that come along with the `kusanagi` package) have time horizons that range from 25 to 40 whereas the time horizons are 200 and 999 for `Pendulum-v0` and `MountainCarContinuous-v0`, respectively. In some environments, for instance, `MountainCarContinuous`, long rollouts are necessary due to sparse rewards and we can see that our model was able to capture the long-term payoff whereas PILCO and deepPILCO learned to just stay still and remained stuck in this local optimal policy.

Chapter 5

Conclusion and Future Work

In this work, we explored a model-based Bayesian sparse sampling method for data-efficient control. This work was inspired by the success of PILCO which used a GPR model for learning the system dynamics and we follow in this direction but with the slight deviation in that we used an approximate GP in the form of RFF thereby providing us with the advantage of a design matrix that does not grow with respect to the size of the training data. Posterior updates then correspond to a simple rank-one Cholesky update, making the model well-suited for an online learning setting. In addition, our work takes a step towards the possibility of directed model-based Bayesian RL exploration inspired by the work in BAMDPs, but we extend the previous research by applying the theory to continuous states and actions and we proposed a simple Monte Carlo sampling for approximating the intractable integrals that result from this formulation. With our Bayesian approach, optimization involves tracking the belief state of the agent at every unrolled step of the optimization procedure, which further motivates the use BLR as the alternative, that is with GPs, posterior updates would correspond to augmenting the gram matrix with the hypothetical tuple at every time step, making the model computationally infeasible. For the choice of optimizer, we chose a gradient-free black-box optimizer in the form of CMA-ES mainly for the reasons that 1) computational graphs in gradient-based optimizers do not mend well with operations involving posterior updates and 2) black-box optimizers cope well with environments that exhibit loss functions with zero gradients everywhere (i.e., mountain car problem).

We compared our model to a baseline model-free method and showed that our model-based method was able to learn optimal policies within the first few epochs in the OpenAI's pendulum and mountain car tasks. Due to the smoothness assumption made by our regression model, our algorithm struggles in robotics tasks simulated using the `pybullet`

physics engine but our model still demonstrated superior data-efficiency in two of the environments that had minimal contact dynamics. In our last set of experiments, we compared our model against PILCO and deepPILCO on the pendulum and mountain car task and showed that our algorithm was able to learn better and more data-efficient policies than the baselines.

There are a few directions for future research, and we close by enumerating two salient potential future areas to explore and improve. Open problems/questions for future research may include:

1. An obvious future direction is the address the difficulties of contact dynamics in some environments. Assuming a model-based method, how would be designed a regression technique that is able to faithfully model the contact discontinuities in such environments?

2. As evidenced through the experiments, our method is still computationally demanding due to the need to propagate the Bayesian belief state at every consecutive time step. Are there any ways to work around this and make our algorithm more computationally tractable? Further, if we are able to design it so that we can do away with a black-box optimizer in favor of a gradient-based optimizer, then this would add an extra boost in computational tractability as it is known that gradient-based optimizers learn faster than gradient-free optimizers.

In this thesis, we introduced a framework for model-based Bayesian RL algorithm that is though computationally very demanding, we hope that it will provide as an example for the direction of principled exploration-exploitation via model-based Bayesian techniques for the challenging domain of continuous states and actions.

References

[1]

[2] A visual guide to evolution strategies. http://blog.otoro.net/2017/10/29/visual-evolution-strategies/. Accessed: 2019-03-15.

[3] Milton Abramowitz and Irene A. Stegun. *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*. Dover, New York, ninth dover printing, tenth gpo printing edition, 1964.

[4] C. G. Atkeson and J. C. Santamaria. A comparison of direct and model-based reinforcement learning. In *Proceedings of International Conference on Robotics and Automation*, volume 4, pages 3557–3564 vol.4, April 1997.

[5] Marc G. Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *CoRR*, abs/1207.4708, 2012.

[6] Giovanni Binet, R Krenn, and Alberto Bemporad. Model predictive control applications for planetary rovers. 01 2012.

[7] Avrim Blum. Random projection, margins, kernels, and feature-selection. In Craig Saunders, Marko Grobelnik, Steve Gunn, and John Shawe-Taylor, editors, *Subspace, Latent Structure and Feature Selection*, pages 52–68, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[8] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[9] Marc Deisenroth. Efficient reinforcement learning using gaussian processes. 11 2010.

[10] Marc Peter Deisenroth and Carl Edward 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*, ICML'11, pages 465–472, USA, 2011. Omnipress.

[11] Ying Ding, Liang Wang, Yongwei Li, and Daoliang Li. Model predictive control and its application in agriculture: A review. *Computers and Electronics in Agriculture*, 151:104 – 117, 2018.

[12] Michael O'Gordon Duff. *Optimal Learning: Computational Procedures for Bayes-adaptive Markov Decision Processes*. PhD thesis, 2002. AAI3039353.

[13] David Duvenaud. *Automatic model construction with Gaussian processes*. PhD thesis, 11 2014.

[14] Chelsea Finn, Xin Yu Tan, Yan Duan, Trevor Darrell, Sergey Levine, and Pieter Abbeel. Learning visual feature spaces for robotic manipulation with deep spatial autoencoders. *CoRR*, abs/1509.06113, 2015.

[15] Carlos E. Garca, David M. Prett, and Manfred Morari. Model predictive control: Theory and practicea survey. *Automatica*, 25(3):335 – 348, 1989.

[16] Tobias Geyer. *Model Predictive Control of High Power Converters and Industrial Drives*. 09 2016.

[17] Shixiang Gu, Timothy P. Lillicrap, Ilya Sutskever, and Sergey Levine. Continuous deep q-learning with model-based acceleration. *CoRR*, abs/1603.00748, 2016.

[18] Nikolaus Hansen. The CMA evolution strategy: A tutorial. *CoRR*, abs/1604.00772, 2016.

[19] Juan Camilo Gamboa Higuera, David Meger, and Gregory Dudek. Synthesizing neural network controllers with probabilistic model based reinforcement learning. *CoRR*, abs/1803.02291, 2018.

[20] Sanket Kamthe and Marc Peter Deisenroth. Data-efficient reinforcement learning with probabilistic model predictive control. *CoRR*, abs/1706.06491, 2017.

[21] Sergey Levine and Pieter Abbeel. Learning neural network policies with guided policy search under unknown dynamics. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'14, pages 1071–1079, Cambridge, MA, USA, 2014. MIT Press.

[22] Sergey Levine, Chelsea Finn, Trevor Darrell, and Pieter Abbeel. End-to-end training of deep visuomotor policies. *CoRR*, abs/1504.00702, 2015.

[23] Sergey Levine and Vladlen Koltun. Guided policy search. 06 2013.

[24] Sergey Levine, Nolan Wagener, and Pieter Abbeel. Learning contact-rich manipulation skills with guided policy search. *CoRR*, abs/1501.05611, 2015.

[25] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *CoRR*, abs/1509.02971, 2015.

[26] Wai Hou Lio, J. A. Rossiter, and B. L. Jones. A review on applications of model predictive control to wind turbines. In *2014 UKACC International Conference on Control (CONTROL)*, pages 673–678, July 2014.

[27] Rowan McAllister and Carl E. Rasmussen. Improving pilco with bayesian neural network dynamics models. 2016.

[28] Rowan McAllister and Carl Edward Rasmussen. Data-efficient reinforcement learning in continuous state-action gaussian-pomdps. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 2040–2049. Curran Associates, Inc., 2017.

[29] William Montgomery and Sergey Levine. Guided policy search as approximate mirror descent. *CoRR*, abs/1607.04614, 2016.

[30] Andrew William Moore. Efficient memory-based learning for robot control. Technical report, 1990.

[31] Anusha Nagabandi, Gregory Kahn, Ronald S. Fearing, and Sergey Levine. Neural network dynamics for model-based deep reinforcement learning with model-free fine-tuning. *CoRR*, abs/1708.02596, 2017.

[32] Anusha Nagabandi, Guangzhao Yang, Thomas Asmar, Gregory Kahn, Sergey Levine, and Ronald S. Fearing. Neural network dynamics models for control of under-actuated legged millirobots. *CoRR*, abs/1711.05253, 2017.

[33] Junhyuk Oh, Xiaoxiao Guo, Honglak Lee, Richard L. Lewis, and Satinder P. Singh. Action-conditional video prediction using deep networks in atari games. *CoRR*, abs/1507.08750, 2015.

[34] Junhyuk Oh, Satinder Singh, and Honglak Lee. Value prediction network. *CoRR*, abs/1707.03497, 2017.

[35] Ali Rahimi and Ben Recht. Random features for large-scale kernel machines. In *In Neural Infomration Processing Systems*, 2007.

[36] Carl Edward Rasmussen. Gaussian processes in machine learning. In *Advanced Lectures on Machine Learning, ML Summer Schools 2003, Canberra, Australia, February 2-14, 2003, Tübingen, Germany, August 4-16, 2003, Revised Lectures*, pages 63–71, 2003.

[37] Carl Edward Rasmussen and Christopher K. I. Williams. *Gaussian Processes for Machine Learning (Adaptive Computation and Machine Learning)*. The MIT Press, 2005.

[38] W. Rudin. *Fourier Analysis on Groups*. Dover Books on Mathematics. Dover Publications, 2017.

[39] Jeff G. Schneider. Exploiting model uncertainty estimates for safe dynamic control learning. In *Proceedings of the 9th International Conference on Neural Information Processing Systems*, NIPS'96, pages 1047–1053, Cambridge, MA, USA, 1996. MIT Press.

[40] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. Trust region policy optimization. *CoRR*, abs/1502.05477, 2015.

[41] David Silver, Hado van Hasselt, Matteo Hessel, Tom Schaul, Arthur Guez, Tim Harley, Gabriel Dulac-Arnold, David P. Reichert, Neil C. Rabinowitz, André Barreto, and Thomas Degris. The predictron: End-to-end learning and planning. *CoRR*, abs/1612.08810, 2016.

[42] Richard S. Sutton. Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *In Proceedings of the Seventh International Conference on Machine Learning*, pages 216–224. Morgan Kaufmann, 1990.

[43] Aviv Tamar, Sergey Levine, and Pieter Abbeel. Value iteration networks. *CoRR*, abs/1602.02867, 2016.

[44] Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033, 2012.

[45] Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. pages 5026–5033, 10 2012.

[46] G. E. Uhlenbeck and L. S. Ornstein. On the theory of the brownian motion. *Phys. Rev.*, 36:823–841, Sep 1930.

[47] Marvin Zhang, Sergey Levine, Zoe McCarthy, Chelsea Finn, and Pieter Abbeel. Policy learning with continuous memory states for partially observed robotic control. *CoRR*, abs/1507.01273, 2015.