Model-Based Active Learning in Hierarchical Policies

by

Vlad M. Cora

B.Sc., Simon Fraser University, 2002

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

The Faculty of Graduate Studies

(Computer Science)

The University Of British Columbia

(Vancouver)

April, 2008

Abstract

Hierarchical task decompositions play an essential role in the design of complex simulation and decision systems, such as the ones that arise in video games. Game designers find it very natural to adopt a divide-and-conquer philosophy of specifying hierarchical policies, where decision modules can be constructed somewhat independently. The process of choosing the parameters of these modules manually is typically lengthy and tedious. The hierarchical reinforcement learning (HRL) field has produced elegant ways of decomposing policies and value functions using semi-Markov decision processes. However, there is still a lack of demonstrations in larger nonlinear systems with discrete and continuous variables. To narrow this gap between industrial practices and academic ideas, we address the problem of designing efficient algorithms to facilitate the deployment of HRL ideas in more realistic settings. In particular, we propose Bayesian active learning methods to learn the relevant aspects of either policies or value functions by focusing on the most relevant parts of the parameter and state spaces respectively. To demonstrate the scalability of our solution, we have applied it to The Open Racing Car Simulator (TORCS), a 3D game engine that implements complex vehicle dynamics. The environment is a large topological map roughly based on downtown Vancouver, British Columbia. Higher

level abstract tasks are also learned in this process using a model-based extension of the MAXQ algorithm. Our solution demonstrates how HRL can be scaled to large applications with complex, discrete and continuous non-linear dynamics.

Table of Contents

List of Tables

List of Figures

Acknowledgements

I would like to acknowledge my supervisor, Dr. Nando de Freitas, for guiding me along the multitude of machine learning research paths, Dr. Michiel van de Panne for valuable feedback and proof-reading the thesis, Matthew Hoffman for many useful discussions, Eric Brochu for help with the Matlab implementation of the Bayesian Active Learning algorithm, and my family and friends for their support all throughout this project. Many thanks are also due to Next Level Games, for supporting me throughout my studies, and delivering on their promise of being a company focused on employee growth and balance.

Chapter 1

Introduction

It is unreasonable to expect that an artificial agent can learn how to behave intelligently from scratch, in the real-world or even in the complex video game worlds created today. Functional regions of the human brain common to most individuals seem to indicate that it is not a tabula-rasa at birth, but has been at least partly hard-wired by millions of years of evolution. While the brain is capable of amazing adaptation, areas such as the motor cortex and occipital lobe perform the same functions in most of us. Therefore, at least some of our incredible learning ability is more or less fine-tuning and reinforcement of pre-existing structures. The human mind is also guilty of countless (deemed irrational) biases that nevertheless enable people to function in an incredibly complex, partially observed, and uncertain world. In this thesis, we have used these "limitations" as an excuse to simplify the problem for ourselves: we require a task hierarchy to be specified by a designer, and abandon the quest for optimal un-biased solutions. We focus instead on possibly sub-optimal, but robust and easily tuneable algorithms. When creating an intelligent agent, we consider these two main factors: i) the ease of entering prior knowledge and amount of hand-tuning required for satisfactory results, and ii) the total cost of the learning algorithm in terms

of both world sampling complexity as well as computational complexity.

In general, problem solving and planning becomes easier when it is broken down into subparts. Variants of functional hierarchies appear consistently in video game AI solutions, from behaviour trees, to hierarchically decomposed agents (teams vs. players), implemented by a multitude of customized hierarchical state machines. The benefits are due to isolating complex decision logic to fairly independent functional units (or tasks). The standard game AI development process consists of the programmer implementing a large number of behaviours in as many ways as there are published video games, and reducing the problem to a more manageable number of tuneable parameters. We present a class of algorithms that attempts to bridge the gap between game development and general reinforcement learning. We aim to reduce the amount of hand-tuning traditionally encountered during game development, while still maintaining the full flexibility of manually hardcoding a policy when necessary.

The Hierarchical Reinforcement Learning field [4] models repeated decision making by structuring the policy into tasks (actions) composed of subtasks that extend through time (temporal abstraction) and are specific to a subset of the total world state space (state abstraction). Many algorithms have recently been developed, and are described further in Section 3.1. The exploration policies typically employed in HRL research tend to learn slowly in practice, even after the benefits of state abstraction and reward shaping. We demonstrate an integration of the MAXQ hierarchical task learner with Bayesian active exploration that significantly speeds up the learning process, applied to hybrid discrete and continuous state and action spaces.

1.1 Motivating Work

Manually coding hierarchical policies is the mainstay of video game AI development. The requirements for automated HRL to be a viable solution are that it must be easy to customize task-specific implementations, state abstractions, reward models, termination criteria and it must support continuous state and action spaces. Out of the solutions investigated, MAXQ [8] met all our requirements, and was the easiest to understand and get positive results quickly. The other solutions investigated include HAR and RAR [10] which extend MAXQ to the case of average rewards (rather than discounted rewards). Their implementations are very similar, and gave roughly the same results in our experiments. Hierarchies of Abstract Machines (HAM) [21] and ALisp [1] are exciting new developments that have been recently applied to a Real-Time-Strategy (RTS) game [18]. ALisp introduces programmable reinforcement learning policies that allows the programmer to specify choice points for the algorithm to optimize. Although the formulation is very nice and would match game AI development processes, the underlying solver based on HAMs flattens the task hierarchy by including the program's memory and call-stack into a new joint-state space, and solves this new MDP instead. It is less clear how to extend and implement per-task customized learning with this formulation. Even if this difficulty is surmounted, as evidenced by the last line in the concluding remarks of [18], there is an imperative need for designing faster algorithms in HRL. This thesis aims to address this need with model-based and active learning.

In our solution, we still require a programmer or designer to specify the task hierarchy. In most cases breaking a plan into sub-plans is much easier than coding the decision logic. With the policy space constrained by the task hierarchy, termination and state abstraction functions, the rate of learning is greatly improved, and the amount of memory required to store the solution reduces. The benefits of HRL are very dependant however on the quality of these specifications, and requires the higher-level reasoning of a programmer or designer. An automatic solution to this problem would be an agent that can learn how to program, and anything less than that would have limited applicability. We have not tackled the complex problems of automatic state abstraction and hierarchical structure learning in this thesis.

1.2 Overview

Our experiments use The Open Racing Car Simulator (TORCS), a 3D game engine that implements complex vehicle dynamics complete with manual and automatic transmission, engine, clutch, tire, suspension and aerodynamic models, described in section 2.1. Hierarchical Reinforcement Learning decomposes the policy into tasks, where each task is effectively a separate, self-contained decision process. This feature has allowed us to integrate active learning for discrete map navigation with continuous low-level vehicle control. The details of this task hierarchy are described in section 2.2.

The original MAXQ algorithm [8] is based on the model-free iterative Q-learning algorithm. It maintains the expected utility of taking action a in state s, and updates this value directly using an iterative backup equation.

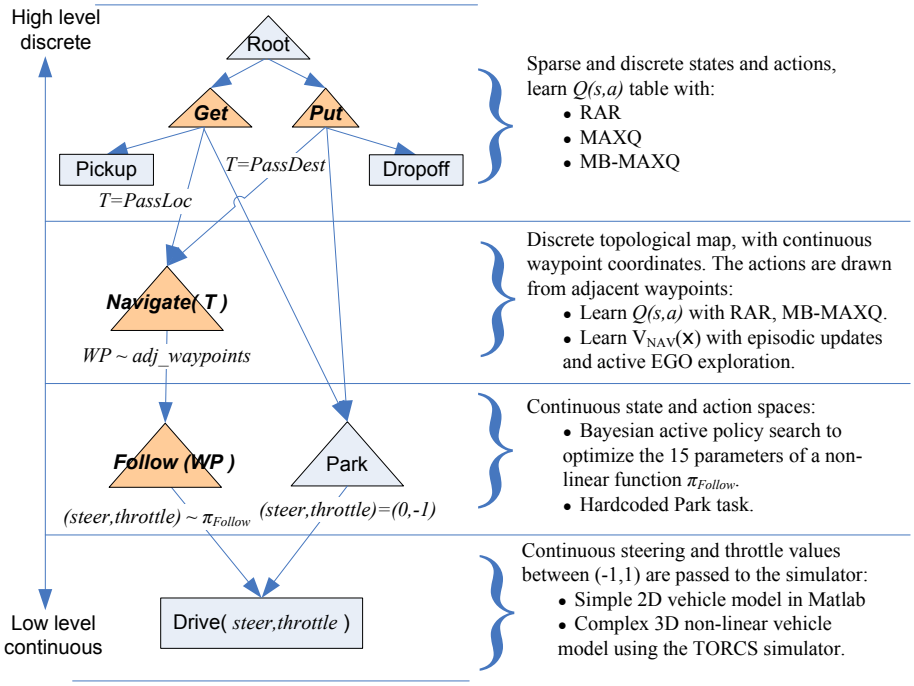

Figure 1.1: **Adaptive Hierarchical Task Tree**. Quick summary of the different task levels in the hierarchical policy, and the algorithms applied. The tasks highlighted learn their policies. A sample run would be *Root* invoking *Get*, which decides between *Pickup*, *Park* or *Navigate* based on whether the taxi is at a legal pickup location, and whether it is stopped. The same *Navigate* task is reused by both *Get* and *Put* by passing different values to the Target parameter T. Subsequently, the *Navigate* task learns path finding across the topological map to arrive to destination T, by repeatedly choosing target waypoints WP from adjacent intersections, and invoking the *Follow* task. This task is a previously trained tracking controller that follows a trajectory, and terminates when the taxi is at WP. The *Drive* action sends the driving control values to the simulator, and returns the new state. A negative reward value is accumulated with every timestep, with positive reward received when the taxi arrives at T, performs a successful pickup, or successful dropoff. See section 2.3 for further details.

The algorithm is model-free because it does not maintain any information about the dynamics of the environment. It requires very little computation for planning. However it learns too slowly, and needs an unreasonable number of samples from the world. We have extended MAXQ with model-based planning (section 3.1.3) and a more expensive exploration algorithm based on Gaussian Processes (GP), that not only interpolates existing observations to unobserved areas of the state space, but also maintains the agent's uncertainty over its knowledge (described in section 3.2). The agent intelligently explores uncertain actions, and exploits the ones with the highest value quickly. We have shown that our exploration policy using GPs requires significantly less samples from the world (or simulator), at the expense of more computation and planning by the agent (see section 4.2).

The lowest level task in the hierarchy generates continuous steering and throttle control values, given errors between the vehicle's position & heading and a given trajectory (see section 3.2.1). We have designed a non-linear policy function with 15 free parameters inspired by the parameterized policy used by [20] to fly a real model-helicopter. The free parameters $\mathbf{x}$ are trained through active policy search. First we generate a small number of samples of $\mathbf{x}$ and estimate the $V(\mathbf{x})$ value function through Monte-Carlo sampling of trajectories simulated in an empty parking lot. The Bayesian active learner fits a Gaussian Process regression model on the estimated $V(\mathbf{x})$ function and generates a new set of parameters $\mathbf{x}$ to evaluate next. It does this by maximizing an Expected Improvement function, computed from the GP by summing the estimated $V(\mathbf{x})$ with the uncertainty (or variance). We have control over exploration/exploitation by tuning the weight of the

uncertainty.

The next level up in the hierarchy is a discrete task that learns a sequence of waypoints for the lower level navigation task to follow, in order to get to the target destination. In this case, we take advantage of the continuous nature of map coordinates to fit a Gaussian Process over the value function $V(\mathbf{x})$. The parameters are $\mathbf{x} = \{x_C, y_C, x_T, y_T\}$, the 2D coordinates of the current and target destinations. At each waypoint, we update the value of navigating to the final destination (see section 3.2.2). The original MAXQ algorithm treats each pair of start/end waypoints as a separate task to learn. However, by fitting a Gaussian Process over the discretely sampled $V(\mathbf{x})$ function results in a form of transfer learning, where proximity to a known location guides the agent quicker to an unknown one. In addition, as described in the previous paragraph, we also generate exploratory actions by maximizing the Expected Improvement function, for quicker convergence to the optimal policy. And finally, the top-most tasks are discrete, and choose between navigating, picking up, or dropping off the passenger; they are simple enough for the original MAXQ algorithm, without active exploration.

All of the algorithms we have employed and extended are general and cleanly integrated using the MAXQ task hierarchy. We do not claim that they are the best solution for the task of map navigation or vehicle control. The domain chosen was meant to serve as a complex demonstration of the benefits and versatility of active learning using Gaussian Processes, combined with Hierarchical Reinforcement Learning. They have enabled a learning solution to scale to a more realistic state space: a map roughly based on downtown Vancouver, with complex hybrid discrete and continu-

ous dynamics. Experimental details and results are described in section 4.

Chapter 2

Application Domain

In order to frame our algorithmic research, we first introduce the motivating application. The original Taxi Domain of [8] is a small, fully discrete, well known benchmark application for hierarchical reinforcement learning. It consists of a taxi in a 5x5 grid world, with 4 possible passenger pickup and dropoff locations. The taxi has 6 primitive actions: North, South, East, West, Pickup and Dropoff. The hierarchical policy on the left of Figure 2.3 greatly speeds up learning over a flat policy by taking advantage of state abstraction, and reusing the *Navigate* task in both the *Get* and *Put* subtasks. We have extended this application to a more realistic setting: a topological map roughly half the size of downtown Vancouver, BC, with continuous vehicle dynamics. The agent needs to first learn how to drive a car, then it learns how to navigate the topological map, and, lastly when to park and pickup or dropoff passengers. This chapter describes the details of our domain, including the task hierarchy implemented.

2.1 The Open Racing Car Simulator

The simulator used for continuous vehicle dynamics is The Open Racing Car Simulator [30]. It is a 3D game engine that implements complex vehicle

dynamics complete with manual and automatic transmission, engine, clutch, tire, suspension and aerodynamic models. The engine architecture makes it very easy to write and integrate custom AI drivers, as well as tuning the hundreds of parameters defining the vehicle physics and road properties. All too often, AI researchers devise their own simple dynamics simulators then tune the learning algorithms to perform well on the simplified model. Subsequently, a lot of difficulty may be encountered when applying the general learning algorithm to other domains. An important criteria in our research was to have no knowledge of the underlying simulation implementation when designing and tuning the learning algorithms. This approach has led to more robust behaviour, in a more realistic setting of learning in an unknown or roughly specified domain.

2.2 The Vancouver Taxi Domain

Table 2.1: Comparing Domain Size

Domain	Size of final policy
5x5 Taxi Flat	$\sim 12,200$ *bytes*
Vancity Flat	$\sim 1,417,152$ *bytes*
Vancity Hierarchical	$\sim 18,668$ *bytes*

Our world is a city map roughly based on a portion of downtown Vancouver, British Columbia, illustrated in Figure 2.2. The data structure is a topological map (a set of intersection nodes and adjacency matrix) with 61 nodes and 22 possible passenger pickup and dropoff locations. The total navigable distance is roughly 28 kilometers, taking into account both direc-

Figure 2.1: **The Open Racing Car Simulator** is a full featured vehicle simulator with complex non-linear dynamics, complete with manual and automatic transmission, engine, clutch, tire, suspension and aerodynamic models.

tions of traffic. The state model includes both discrete variables used in the top layers of the task hierarchy, as well as continuous variables used by the *Follow* task that tracks a trajectory, and are described in Table 2.2.

Table 2.1 makes a rough comparison between the size of our extended application and the original taxi domain. Ignoring the continuous trajectory states (including the *Stopped* flag) and assuming the taxi hops from one intersection to an adjacent one in a single timestep results in a fully discrete problem. A flat learning solution scales poorly, not only in terms of world

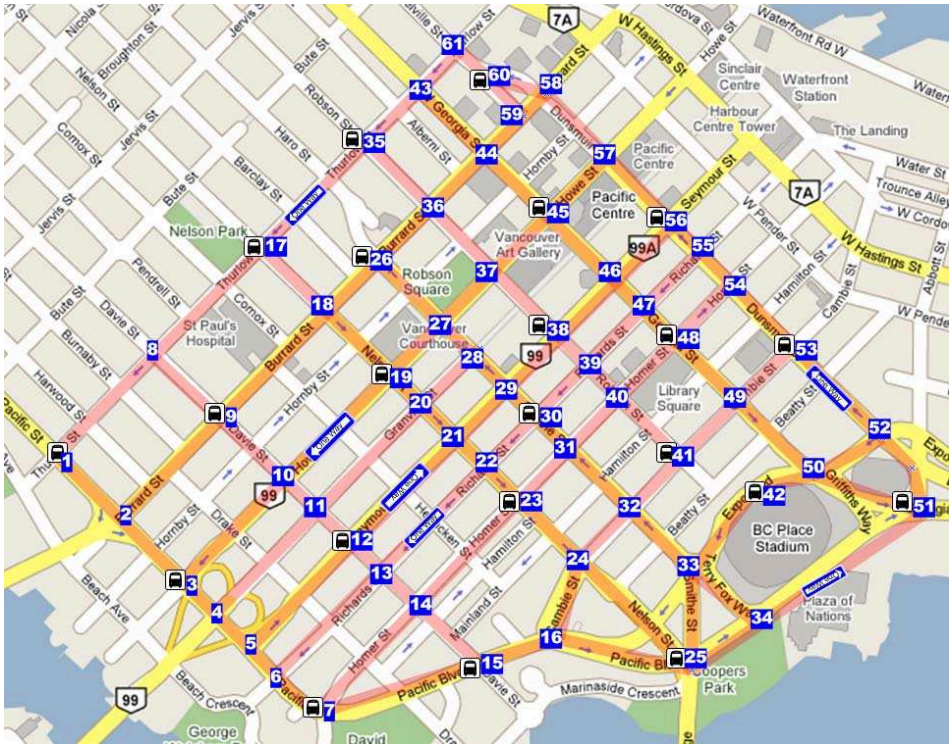

Figure 2.2: **Vancity Experiment** on a map (orange overlay) roughly based on downtown Vancouver, and used by the TORCS simulator. Each waypoint is labeled, and pickup and dropoff locations are marked by the Taxi icons. One way streets are accounted for in the waypoint adjacency matrix. Source image care of Google Maps.

samples required, but also in the size of the computed policy (if represented in a discrete table). As shown in the table, the extended task hierarchy illustrated in Figure 2.3 requires just a little bit more memory than the small 5x5 taxi domain.

2.3 Hierarchical Abstraction of the Vancouver Taxi

Taking advantage of the benefits of Hierarchical RL, requires a designer to decompose the policy into tasks and subtasks, and to specify the relevant state abstraction and reward functions, per task. The decomposition is designed to maximize the potential for state abstraction, and for applying different learning algorithms to different levels. Figure 2.3 compares the original Taxi task hierarchy, with our extended version that includes continuous trajectory following and a hardcoded *Park* task. The state abstraction function filters out irrelevant states when computing the hash key for looking up and updating the value function table defining the hierarchical policy (described further in section 3.1). The *Follow* task is trained with the Active Policy optimizer from section 3.2.1, and the resulting policy parameters are fixed before learning the higher level tasks. Algorithms RAR, MAXQ and Model-Based (MB)-MAXQ are applied to all the tasks above and including *Navigate*, which also uses the Active Path learning algorithm from section 3.2.2. Here is a summary of each task i, including its reward model, termination predicate T_i, and state abstraction function Z_i:

Root - this task selects between *Get* and *Put* to pickup and deliver the passenger. It requires no learning because the termination criteria of the subtasks fully determine when they should be invoked. $T_{Root} = (PassLock = PassDest)$, $Z_{Root} = \{\}$.

Get - getting the passenger involves navigating through the city, parking the car and picking up the passenger. In this task, the *LegalLoad* state is

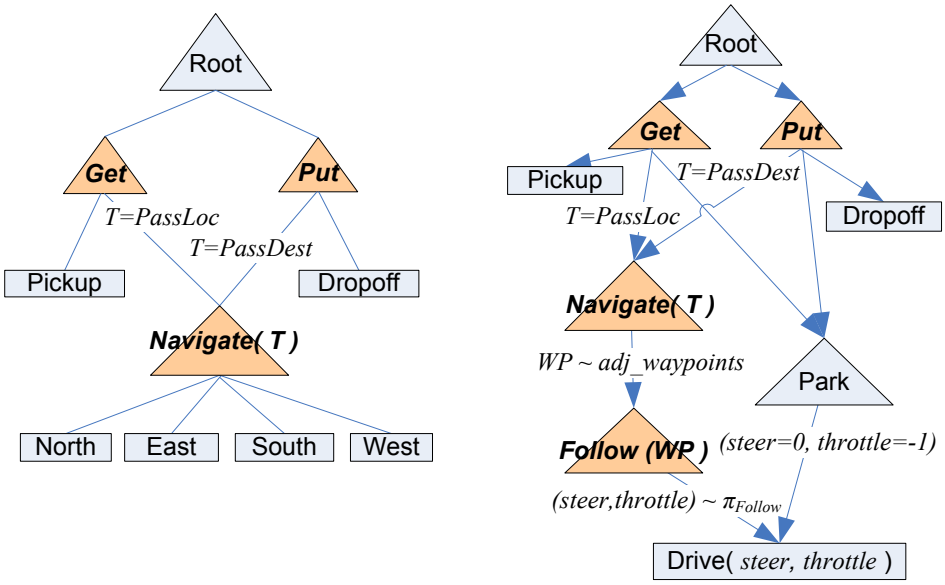

Figure 2.3: **Task Hierarchies.** Each composite task illustrated is an SMDP composed of other SMDPs, highlighting the adaptive ones in bold. Triangles are composite tasks, and rectangles are primitive actions. The left illustration is from the original discrete Taxi domain of [8]. The hierarchy on the right extends the problem to continuous state, by replacing the discrete actions N/S/E/W with one continuous *Drive(steer, throttle)* task. The driving controls are generated by the parameterized policy π_{Follow}.

true when the taxi is at the passenger's location. Receives a reward of 750 when the passenger is picked up, $T_{Get} = ((PassLoc = 0) \vee (PassLoc = PassDest))$, $Z_{Get} = \{\}$.

Put - similar to *Get*, also receives reward of 750 when the passenger is successfully delivered. The passenger destination *PassDest* is passed to the *Navigate* task. The abstracted *LegalLoad* state is true when the taxi is at the passenger's destination location. $T_{Put} = ((PassLoc > 0) \vee (PassLoc = PassDest))$, $Z_{Put} = \{\}$.

Table 2.2: States and Task Parameters

Name	Range/Units	Description
$TaxiLoc$	{0,1,..61}	current taxi waypoint #, or 0 if in transit between waypoints
$PassLoc$	{0,1,..22}	passenger waypoint #, or 0 if in taxi
$PassDest$	{1,2,..22}	passenger destination waypoint #
$LegalLoad$	$\{true, false\}$	true if taxi is empty and at passenger, or loaded and at target
$Stopped$	$\{true, false\}$	indicates whether the taxi is at a complete stop
T	{1,2,..22}	Target passenger location or destination parameter passed into *Navigate*
WP	{1,2,..22}	waypoint parameter adjacent to *TaxiLoc* passed to *Follow*
Y_{err}	meters	lateral error between desired point on the trajectory and vehicle
V_y	meters/second	lateral velocity (to detect drift)
V_{err}	meters/second	error between desired and real speed
Ω_{err}	radians	error between trajectory angle and vehicle yaw

Pickup - this is a primitive action, with a reward of 0 if successful, and -2500 if a pickup is invalid (if the taxi is not stopped, or if *LegalLoad* is false). $Z_{Pickup} = \{LegalLoad, Stopped\}$.

Dropoff - this is a primitive action, with a reward of 1500 if successful, and -2500 if a dropoff is invalid. $Z_{Dropoff} = \{LegalLoad, Stopped\}$.

Navigate - this task learns the sequence of intersections WP from the current *TaxiLoc* to a target destination T. By parameterizing the value function of this task, we can apply Active Path learning as described in Section 3.2.2. $T_{Navigate} = (TaxiLoc = T)$, $Z_{Navigate} = \{TaxiLoc, T\}$.

Follow - this is the previously trained continuous trajectory tracking task that takes as input an adjacent waypoint WP, and generates continuous

steering and throttle values to follow the straight-line trajectory from *TaxiLoc* to *WP*. $T_{Follow} = (TaxiLoc = WP)$, $Z_{Follow} = \{WP, \Omega_{err}, V_{err}, Y_{err}, V_y\}$.

Park - this is a hardcoded task which simply puts on the brakes ($steer = 0, throttle = -1$). The abstraction function is $Z_{Park} = \{LegalLoad, Stopped\}$. It might be intuitive at first glance for the *Park* task to be invoked by *Navigate*, but notice that the two tasks depend on completely different states. Parking the car depends on the same states as deciding whether to *Pickup/Dropoff* or *Navigate*: if the taxi is at a legal (un)loading location, and whether the taxi is stopped. This policy is learned by the *Get* and *Put* tasks, whereas the *Navigate* task is limited to pathfinding only.

Drive - this performs one timestep of the physics simulation, with the given steer and throttle inputs. The default reward per timestep of driving is -0.75.

Chapter 3

Algorithmic Details

3.1 Hierarchical Reinforcement Learning

This section describes the algorithmic details of Hierarchical RL. The MAXQ learner is capable of learning the entire hierarchical policy online. It does not require the child tasks to be optimal before learning the parents. However, this does lead to slower convergence, since the policies for higher level tasks have high variance during the learning phase, until the lower level tasks improve. So it usually makes sense, if possible, to learn the bottom-most layers of the hierarchy first. In some cases, this is required, such as learning how to control a car through steering and throttle action values, before learning how to navigate a map. Usually, the lower level tasks operate at shorter timescales than the higher levels. Since the observed rewards are aggregated at each task, the reward function at different layers needs to be scaled accordingly. For example if the penalty for picking up the passenger at the wrong place in the *Get* task is less than the cost of *Navigate*, the taxi would sit there, repeatedly invoking *Pickup*. An automatic solution to this problem is left to future research.

The following sections summarize the theoretical background of Hierarchical RL, and present a Model-Based extension to the MAXQ learner.

3.1.1 Semi-MDPs

Each task in a MAXQ hierarchy is a semi-Markov Decision Process [28], that models repeated decision making in a stochastic environment, where the actions can take more than one timestep. Formally, an SMDP is defined as a tuple: $\{S, A, P(s', N|s, a), R(s, a)\}$ where S is the set of state variables, A is a set of actions, $P(s', N|s, a)$ is the transition probability of arriving to state s' in N timesteps after taking action a in s, and $R(s, a)$ is the reward received. The solution of this process is a policy $\pi^*(s) \in A$, that selects the action with the highest expected discounted reward in each state. The function $V^*(s)$ is the value of state s when following the optimal policy. Equivalently, the $Q^*(s, a)$ function stores the value of taking action a in state s and following the optimal policy thereafter. These quantities follow the classical Bellman recursions, with $\gamma \in [0, 1)$ being the tuneable discount factor that specifies how much we value recent outcomes over future ones:

$$V^*(s) = max_{a \in A} \left[R(s, a) + \gamma \sum_{s', N} P(s', N|s, a) \gamma^N V^*(s') \right] \tag{3.1}$$

$$Q^*(s, a) = R(s, a) + \gamma \sum_{s', N} P(s', N|s, a) \gamma^N V^*(s') \tag{3.2}$$

3.1.2 Hierarchical Value Function Decomposition

A composite task i in MAXQ is defined as a tuple: $\{A_i, T_i(s), Z_i(s), \pi_i(s)\}$ where s is the current world state, A_i is a set of subtasks, $T_i(s) \in \{true, false\}$ is a termination predicate, $Z_i(s)$ is a state abstraction function that returns a

subset of the state relevant to the current subtask, and $\pi_i(s) \in A_i$ is the policy learned by the agent (or used to explore during learning). The primitive tasks perform atomic actions in the world and last only one timestep. Each composite task contains subtasks, and is effectively a separate, decomposed SMDP that has allowed us to integrate active learning for discrete map navigation with continuous low-level vehicle control. This is accomplished by decomposing the Q function into two parts:

$$\begin{aligned} a &= \pi_i(s) \\ Q^\pi(i,s,a) &= V^\pi(a,s) + C^\pi(i,s,a) \\ C^\pi(i,s,a) &= \sum_{s',N} P_i^\pi(s',N|s,a)\gamma^N Q^\pi(i,s',\pi_i(s')) \\ V^\pi(i,s) &= \begin{cases} Q^\pi(i,s,\pi_i(s)) & \text{if composite} \\ \sum_{s'} P(s'|s,i)R(s'|s,i) & \text{if primitive} \end{cases} \end{aligned} \tag{3.3}$$

Here, γ is the discount factor, i is the current task, and a is a child action given that we are following policy π_i. The Q function is decomposed into two parts: the value of V^π being the expected one step reward, plus C^π which is the expected completion reward for i after a completes. V is defined recursively, as the expected value of its child actions, or the expected reward itself if i is a primitive (atomic) action. The MAXQ learning routine is a simple modification of the typical Q-learning algorithm. In task i, we execute subtask a, observe the new state s' and reward r. If a is primitive, we update

$V(s,a)$, otherwise we update $C(i,s,a)$, with learning rate $\alpha \in (0,1)$:

$$\begin{aligned} V(a,s) &= (1-\alpha)V(a,s) + \alpha r \\ C(i,s,a) &= (1-\alpha)C(i,s,a) + \alpha\gamma^N \max_{a'} Q(i,s',a') \end{aligned} \tag{3.4}$$

An important consideration in HRL is whether the policy calculated is hierarchically or recursively optimal. Recursive optimality, satisfied by MAXQ and RAR, means that each subtask is locally optimal, given the optimal policies of the descendants. This may result in a suboptimal overall policy because the effects of tasks executed outside of the current task's scope are ignored. For example if there are two exits from a room, a recursively optimal policy would pick the closest exit, regardless of the final destination. A hierarchically optimal policy (computed by the HAR [10] and HAM [1] three-part value decompositions) would pick the exit to minimize total traveling time, given the destination. A recursively optimal learning algorithm however generalizes subtasks easier since they only depend on the local state, ignoring what would happen after the current task finishes. So both types of optimality are of value in different degrees for different cases. The MAXQ formulation gives a programmer or designer the ability to selectively enable hierarchical optimality by including the relevant state features as parameters to a task. However, it may be difficult to identify the relevant features, as they would be highly application specific.

3.1.3 Model-Based MAXQ

To reduce the amount of sampling from the world or simulator, we have extended the original model-free MAXQ algorithm with Real-Time Dynamic Programming (RTDP) [3] model-based planning. In the original model-free MAXQ algorithm, the value $C^{\pi}(i,s,a)$ defined in the previous section is computed iteratively using the following backup equation, with N being the number of timesteps action a took to complete and arrive in state s'.

$$C_{t+1}(i,s,a) = (1-\alpha_t(i))C_t(i,s,a) + \alpha_t(i)\gamma^N \max_{a'} Q(i,s',a') \tag{3.5}$$

Building the empirical model from observations entails estimating the probability distribution $P(s',N|s,a)$ of arriving at state s' in N steps, after taking action a in state s. This is not feasible, so instead we have approximated equation 3.3 with the following:

$$C(i,s,a) \approx \sum_{s'} P_i(s'|s,a)\gamma^{D_i(s,a,s')}Q(i,s',\pi(s')) \tag{3.6}$$

where $D_i(s,a,s')$ is the estimated length of time action a takes to arrive at state s'. Now in addition to performing the backup equation 3.4, in each composite task i, we also update from observations the internal empirical model:

$$P(s'|s,a) = n_i(s,a,s')/n_i(s,a) \tag{3.7}$$

$$D_i(s,a,s') = d_i(s,a,s')/n_i(s,a) \tag{3.8}$$

Algorithm 1 The pseudo-code is the same as Diettreich's original MAXQ algorithm, with the added RTDP style updates at lines 13 and 21.

1: **function** $MBMAXQ(Task\ i, State\ s)$
2: **let** $trajectory$=() be the sequence of states visited while executing i
3: **if** i is a primitive action **then**
4: execute i, receive r_t, and observe result state s'
5: $V_{t+1}(i,s) = (1-\alpha_t)V_t(i,s) + \alpha_t r_t$
6: push s into the beginning of $trajectory$
7: **else**
8: **while** $T_i(s)$ is false **do**
9: choose an action a according to exploration policy $\pi_x(i,s)$
10: **let** $childSeq = MBMAXQ(a,s)$
11: observe result state s'
12: **let** N = length($childSeq$)
13: **UpdateEmpiricalModel(** i,s,a,s',N **)**
14: **for all** s in $childSeq$ **do**
15: $C_{t+1}(i,s,a) = (1-\alpha_t)C_t(i,s,a) + \alpha_t\gamma^N \max_{a'} Q_t(i,s',a')$
16: $N = N - 1$
17: **end for**
18: **append** $childSeq$ onto the front of $trajectory$
19: $s = s'$
20: **end while**
21: **RunValueIteration(** $trajectory$ **)**
22: **end if**
23: **return** $trajectory$

24: **function** $UpdateEmpiricalModel(i,s,a,s',d)$
25: $n_i(s,a) = n_i(s,a) + 1$
26: $n_i(s,a,s') = n_i(s,a,s') + 1$
27: $P_i(s'|s,a) = n_i(s,a,s')/n_i(s,a)$
28: $d_i(s,a,s') = d_i(s,a,s') + d$
29: $D_i(s,a,s') = d_i(s,a,s')/n_i(s,a)$

30: **function** $RunValueIteration(trajectory)$
31: updateStates = SelectStatesToUpdate(trajectory)
32: **for all** s in $updateStates$ **do**
33: **for all** a in A_i **do**
34: $C_{t+1}(i,s,a) = \sum_{s'} P_i(s'|s,a)\gamma^{D_i(s,a,s')} \max_{a'} [V_t(a',s') + C_t(i,s',a')]$
35: **end for**
36: **end for**

With these estimates, we run a few steps of value iteration on equation 3.6 upon completion of task i, in order to propagate the observations further across states visited previously. This simple method reduces the total number of samples required from the world, at the expense of more internal computation on the empirical model. Refer to Algorithm 1 for the full pseudocode.

3.2 Bayesian Active Learning

The objective of Bayesian active learning is to learn properties of the value function or policy with as few samples as possible. In direct policy search, where this idea has been explored previously [23], the evaluation of the expected returns using Monte Carlo simulations is very costly. One, therefore, needs to find a peak of this function with as few policy iterations as possible. As shown here, the same problem arises when we want to learn an approximation of the value function only over the relevant regions of the state space. Bayesian active learning provides an exploration-exploitation mechanism for finding these relevant regions and fitting the value function where needed. Unlike traditional active learning, where the focus is often only in exploration (e.g. query the points with the maximum variance, entropy or other information-theoretic measures [29, 11]), here the goal is to balance exploitation and exploration. That is, to save computation, we only want to approximate the value function accurately in regions where it is profitable to do so. We do not need to approximate it well over the entire state space.

Bayesian active learning, also known as average-case analysis, involves three stages. First, a rough prior distribution is defined over the object being analyzed (for example the value function $V(\cdot)$). This can be non-informative, however the results improve with an informative prior, when available. Second, a set of N previously gathered observations $D_{1:N} = \{\mathbf{x}_i, V(\mathbf{x}_i)\}_{i=1}^{N}$ is combined with the prior, through Bayes rule, to obtain the posterior distribution over the object. Finally, the posterior risk (improvement) is minimized (or maximized) so as to determine which new data should be gathered. Mathematically, the point of maximum expected improvement is given by:

$$\mathbf{x}_{N+1} = \arg\max_{\mathbf{x}} \mathbb{E}\left[\max\{0, V(\mathbf{x}) - V_{\max}\} | D_{1:N}\right] \tag{3.9}$$

where $\max\{0, V(\mathbf{x}) - V_{\max}\}$ denotes the improvement over a defined standard (say the best value so far). The expectation is taken with respect to the posterior distribution $P(V(\cdot)|D_{1:N})$. In contrast with the popular worst-case (minimax) approach, this average-case analysis can provide faster solutions in many practical domains where one does not believe that the worst case scenario is very probable.

To implement the first stage, we place a Gaussian process (GP) prior over the value function: $V(\cdot) \sim GP(m(\cdot), K(\cdot,\cdot))$, see e.g. [24] for details on Gaussian process regression. The inherent assumption here is one of smoothness. Although we actually learn the mean function in the manner proposed in [23], for presentation clarity let us assume that it is the zero function as it is assumed often in the machine learning literature. We adopt the standard Gaussian and Matern kernel functions to describe the com-

ponents of the kernel matrix $\mathbf{K}$. The parameters of these functions (say kernel width in the Gaussian case) can be learned by maximum a posteriori inference, but since in our active learning setting we don't have many data points, the priors need to be fairly informative. That is, one has to look at the data and get a rough estimate of the expected distance between data points in order to choose the smoothing kernel width.

It is then easy to obtain an exact expression for the mean, μ, and variance, σ^2, of the posterior distribution:

$$\begin{aligned} \mu(\mathbf{x}) &= \mathbf{k}^T\mathbf{K}^{-1}(V_{1:N}) \\ \sigma^2(\mathbf{x}) &= k(\mathbf{x},\mathbf{x}) - \mathbf{k}^T\mathbf{K}^{-1}\mathbf{k}. \end{aligned} \tag{3.10}$$

where $V_{1:N} = (V_1, \ldots, V_N)$, $\mathbf{K}$ denotes the full kernel matrix and $\mathbf{k}$ denotes the vector of kernels $k(\mathbf{x}, \mathbf{x}_i)$ for $i = 1, \ldots, N$. The algorithm requires $O(N^2)$ operations to compute the correlation (kernel) matrix $\mathbf{K}$. As the number N of query points increases, computing $\mathbf{K}^{-1}$ can be expensive, and is required with each new observation.

Let $V_{\max}$ denote the best value of the value function at the current iteration. The expectation of the improvement function $I(\mathbf{x}) = \max\{0, V(\mathbf{x}) - V_{\max}\}$ with respect to the posterior distribution $\mathcal{N}(V(x); \mu(\mathbf{x}), \sigma^2(\mathbf{x}))$ results in the following expression:

$$EI(\mathbf{x}) = \begin{cases} (\mu(\mathbf{x}) - V_{\max})\Phi(d) + \epsilon \cdot \sigma^2(\mathbf{x})\phi(d) & \text{if } \sigma^2 > 0 \\ 0 & \text{if } \sigma^2 = 0 \end{cases} \tag{3.11}$$

where ϕ and Φ denote the PDF and CDF of the standard Normal distri-

bution and $d = \frac{\mu(\mathbf{x}) - V_{\max}}{\sigma^2(\mathbf{x})}$. The exploration multiplier ϵ allows control over preferring exploration or exploitation. Finding the maximum of the expected improvement function is a much easier problem than the original one because it can be fairly cheaply evaluated. Maximizing the expected improvement rather than the value function implies that with each observation, the response surface changes to reflect the updated uncertainty. The next query point suggested by the algorithm will be one that results in useful information (high variance due to lack of observations), or is expected to have a high value based on previous observations. In our implementation, we used the DIRect algorithm of [13], a global optimizer with any-time stopping ability. We have limited the number of iterations for practical reasons, and expect even better results with longer running times. Other methods such as sequential quadratic programming could also be adopted.

The overall procedure is shown in Algorithm 2. Many termination criteria are possible, including time and other computational constraints. When carrying out direct policy search [19], the Bayesian active learning approach has several advantages over the policy gradients method [5]: it is derivative free, it is less prone to be caught in the first local minimum, and it is *explicitly designed to minimize the number of expensive value function evaluations*. We do assume a continuous state space of reasonable size (15 or less dimensions), and a fairly smooth value function represented by the GP.

The Bayesian active learning approach has a long history, starting with early work by Kushner with Wiener processes [15]. It has been successfully applied to derivative-free optimization and experimental design [14] and has

Algorithm 2 Bayesian Active Learning with GPs

1: At iteration N:
2: Update the Gaussian posterior and EI over $D_{1:N}$
3: Choose $\mathbf{x}_{N+1} = \arg\max_{\mathbf{x}} EI(\mathbf{x})$.
4: Evaluate $V_{N+1} = V(\mathbf{x}_{N+1})$ and halt if a stopping criterion is met.
5: Augment the data $D_{1:N+1} = \{D_{1:N}, (x_{N+1}, V_{N+1})\}$.
6: $N = N + 1$ and go to step 2.

recently begun to appear in the machine learning literature [16]. There exist several consistency proofs for this algorithm in the one-dimensional setting [17] and one for a simplification of the algorithm using simplicial partitioning in higher dimensions [31].

It should be clear to readers with a good grasp of decision theory that the expected improvement function is simply a myopic (one-step-ahead) utility function. Although, not pursued in the present thesis, it is possible to look-ahead more steps using Bellman's backward induction method; see for example [6, 27] for details. Note that although we are replacing the original stochastic programming problem with another one, it is important to notice that the latter (optimizing the expected improvement function) is easier because the objective function is cheaper to evaluate.

3.2.1 Active Policy Optimization

In practice, the low-level *Follow* task uses the parameterized function illustrated in Figure 3.1 to generate continuous steer and throttle values, within the range of (-1,1). The $|\mathbf{x}| = 15$ parameters (weights) are trained using the Bayesian active policy learning Algorithm 2, with $\mathbf{x} = \{w1, w2, ..., w15\}$. The DIRect optimizer is limited to $x \in (0, 1)$, and we re-scale the weights

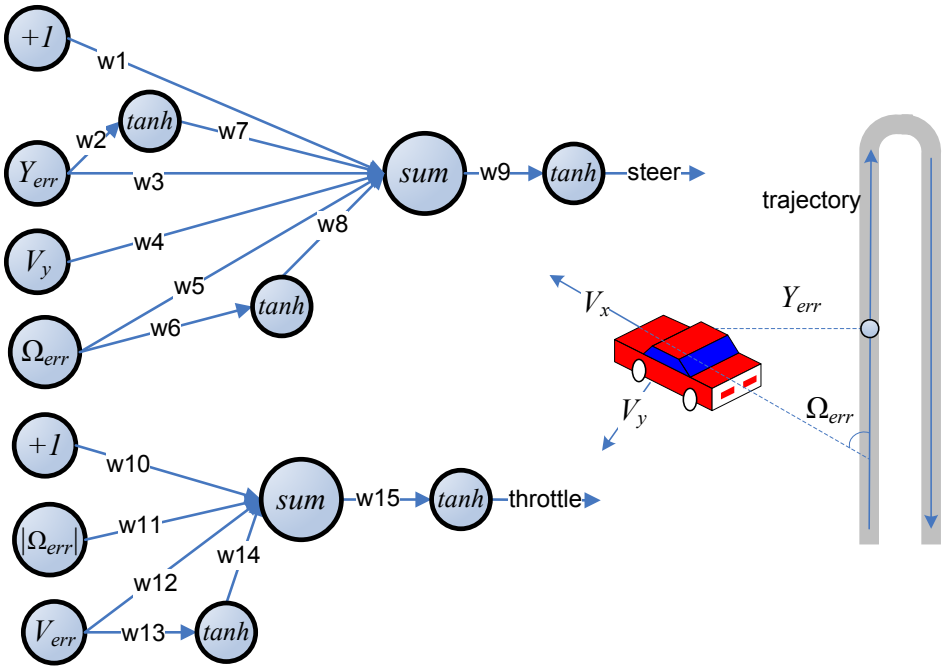

Figure 3.1: **Trajectory Tracking Policy:** This parameterized policy, inspired by [20] minimizes the error between the vehicle's heading and velocity while following a trajectory. The positional error Y_{err} is in trajectory coordinates, Ω_{err} refers to the difference between the current heading and the trajectory tangent, and V_{err} is the difference between the real and desired velocities.

to lie within $(-1, 1)$.

The intuition behind the policy function is that steering the vehicle depends on how far the taxi is from the trajectory, Y_{err}, as well as the orientation error Ω_{err}. The *tanh* is a decision function that maps its input to $(-1, 1)$, and results in a more flexible and expressive policy. The lateral velocity V_y is meant to help with control during a lateral drift, in which case one should steer into the skid. However in our experiments, this value was usually 0, and could be removed from the policy. Controlling the throttle depends only on the forward velocity error V_{err}, as well as the absolute value

of the error between the car's yaw and the desired trajectory $|\Omega_{err}|$. This allows the agent to learn to slow down and brake when facing in a completely wrong direction.

We first generate and evaluate a set of 45 samples of $\mathbf{x}$ from a latin-hypercube and store them and corresponding values vector V in the data matrix D. The value of a trajectory is the negative accumulated error between the car's position and velocity, and the desired position and velocity. A Latin square is a 2-dimensional grid with only one sample in each row and column. The Latin hypercube generalizes this concept to $n-$dimensions. Sampling from this hypercube results in a more even spread of initial samples than drawing randomly.

The policy evaluation consists of averaging 10 episodes along the same trajectory but with different, evenly spaced starting angles. An episode consists of the car accelerating from rest up to 60 km/hr, tracking the trajectory to the first waypoint, performing a U-turn, and driving back to the starting location. The target velocity is always 60 km/hr, and the optimal policy learns to break while performing the U-turn.

In a noisier environment, more samples would be necessary to properly evaluate a policy. Stochastic learning algorithms sometimes estimate the noise of the world in lieu of modeling the entire dynamics. The TORCS simulator is deterministic, and our policy was able to capture the control requirements without a noise estimate or representing the full vehicle dynamics such as tire slip or roughness of the road. The 10 different starting angles were sufficient for evaluating a policy in our experiments. Subsequently, we perform the iteration described in Algorithm 2 to search for the

best instantiation of the parameters.

3.2.2 Active Value Learning

The *Navigate* task learns path finding from any intersection in the topological map to any of the destinations. Although this task operates on a discrete set of waypoints, the underlying map coordinates are continuous, and we can again apply active exploration with GPs.

Unlike the previous algorithm that searches for a set of optimal parameters, Algorithm 3 learns the value function at a finite set of states, by actively generating exploratory actions; it is designed to fit within a MAXQ task hierarchy. The 4-dimensional value function $V(\mathbf{x})$ in this case is parameterized by two 2D map coordinates $\mathbf{x} = \{x_C, y_C, x_T, y_T\}$, and stores the sum of discounted rewards while traveling from the current intersection $|C| = 61$ to the target $|T| = 22$. The sampled instances of $|\mathbf{x}| = 1342$ and corresponding $V(\mathbf{x})$ vector are stored in the data matrix D; it is initialized with $V(x_T, y_T, x_T, y_T) = 0$ for all target destinations T, which enables the GP to create a useful response surface without actually having observed anything yet.

In the $\epsilon-$greedy experiments, a random adjacent intersection is chosen with chance 0.1, and the greedy one with chance 0.9. For the active exploration case, we fit a GP over the data matrix D, and pick the adjacent intersection that maximizes EI. We parameterize this function with an annealing parameter e that decays over time such that initially we place more importance on exploring [15].

The true value will not be known until the *Navigate* task reaches its

Algorithm 3 Active Path Learning - $V_{Nav}GP$

function $NavigateTaskLearner(Navigate\ i, State\ s)$
let $trajectory$=() - list of all states visited in i
let $intersections$=() - intersection states visited in i
let $visits = 0$ - # of visits at an intersection in i
while $Terminated_i(s)$ is false **do**
  choose adjacent intersection WP using ϵ-greedy or Active exploration.
  let $childSeq = Follow(WP, s)$
  append $childSeq$ onto the front of $trajectory$
  observe result state s'
  $N =$ length($childSeq$)
  $R = \sum_{j=1}^{N} \gamma^{N-j} r_j$ be the total discounted reward from s to s'
  $V'_s = V(TaxiLoc_{s'}, Target_i)$ { guaranteed <= 0}
  $V_s = V(TaxiLoc_s, Target_i)$ { guaranteed <= 0}
  if $Terminated_i(s')$ is true **then**
    $V_s \leftarrow (1-\alpha)V_s + \alpha R$
    for all $j = 1$ to length(intersections) **do**
      $\{s', N', R'\} =$ intersections(j)
      $R \leftarrow R' + \gamma^{N'} R$
      $V'_s \leftarrow V(TaxiLoc_{s'}, Target_i)$
      $V'_s \leftarrow (1-\alpha)V'_s + \alpha R$
    end for
  else
    append $\{s, N, R\}$ onto the front of $intersections$
    $visits(TaxiLoc_s) \leftarrow visits(TaxiLoc_s) + 1$
    $penalty \leftarrow V_s visits(TaxiLoc_s)$ {prevent loops}
    $V_s \leftarrow (1-\alpha)V_s + \alpha(penalty + R + \gamma^N V'_s)$
  end if
  $s = s'$
end while
return $trajectory$

destination and terminates, but we still need to mark visited intersections to avoid indefinite looping. Lines 24-26 compute an estimated value for $V(s)$ by summing the immediate discounted reward of executing $Follow(WP, s)$ with the discounted, previously recorded value of the new state $V(s')$, and a heuristic penalty factor to avoid looping. Once we reach the destination of this task, $Target_i$, we have the necessary information to propagate the discounted reward to all the intersections along the trajectory, in lines 15-21.

Chapter 4

Experimental Results

The experimental results have been generated on a laptop running Windows XP SP2, with Dual Core Intel T7400 @ 2.16 GHz Mobile Processor and 2 GB of RAM. The learning algorithms are implemented in Matlab, communicating with the TORCS C++ simulator through an inter-process shared memory map.

4.1 Learning to Drive

The nature of the domain requires that we run policy optimization first to train the *Follow* task. This is reasonable, since the agent cannot be expected to learn map navigation before learning to drive the car. Figures 4.1 and 4.2 compare the results of running the active policy optimization algorithm from section 3.2.1 on two cars with vastly different properties. For both cars, the desired velocity is 60 km/hr, a timestep lasts 0.25 seconds, and the trajectory reward

$$R = -\sum_t \left[1 \times \tilde{Y}_{err}^2 + 0.95 \times \tilde{V}_{err}^2 + 1 \times \tilde{\Omega}_{err}^2 + 0.5 \times \tilde{\mathbf{a}}'\tilde{\mathbf{a}}\right] \tag{4.1}$$

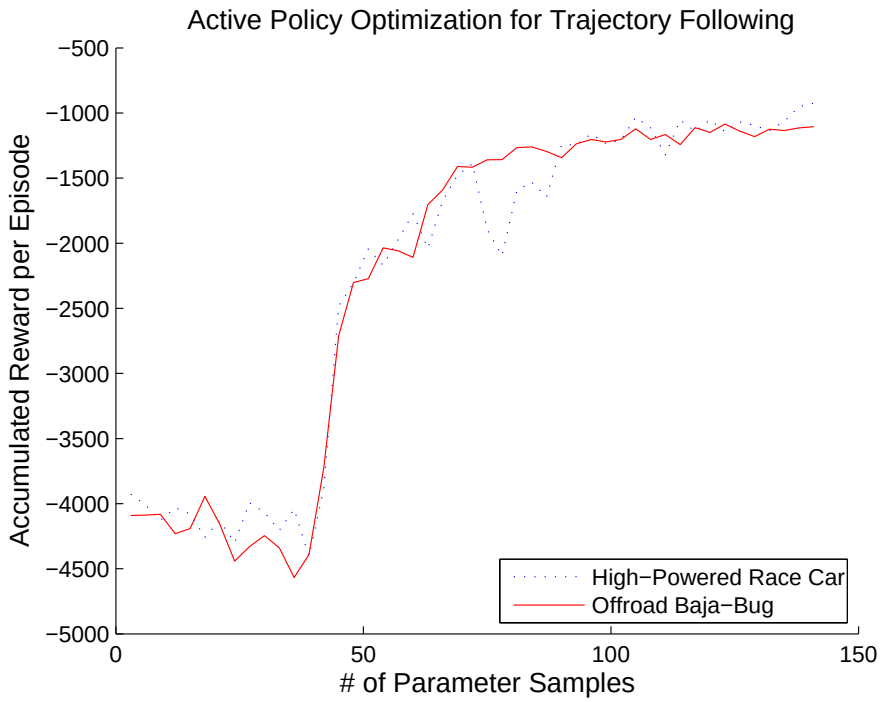

Figure 4.1: **Active Policy Optimizer:** searching for the 15 policy parameters, and comparing the results on two cars with different properties: a high-powered race car with stiff suspensions, and a low-powered offroad buggy with soft suspensions. A total of 15 experimental runs were averaged for this plot.

is the negative weighted sum of normalized squared error values between the vehicle and the desired trajectory, including $\mathbf{a} = [steer, throttle]$ to penalize for abrupt actions. Approximately 40 more parameter samples after the initial 45 random samples, the learner has already found a good policy. A stochastic hill climbing technique such as policy gradient would be very unstable in this case because of the *tanh* function. Hill climbing algorithms are local optimizers that use the derivative of the policy function to adjust the parameters $\mathbf{x}$ in the direction of high reward. The *tanh* function has a

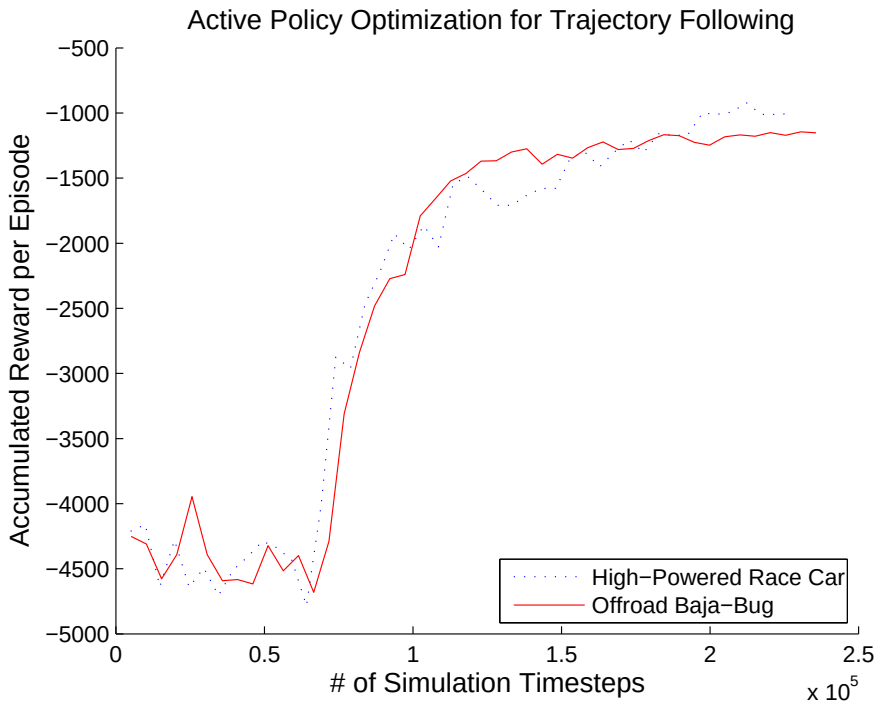

Figure 4.2: **Active Policy Optimizer:** This plot displays the total number of simulation timesteps required to evaluate the parameter samples. Each timestep lasts 0.25 seconds, therefore an entire experimental run takes $\sim$ 15 hours of simulated driving. To make the experiments more bearable, we have accelerated the simulation (without any effect on the results), and a total run takes $\sim$ 2 hours of computation time.

large derivative at the decision point, which causes instability in the step size, requiring a large amount of hand-tuning to bound the maximum weight step size. Being a local search method, hill climbing also suffers from a tendency of getting stuck in local minima, and is not well suited for optimizing multi-modal functions.

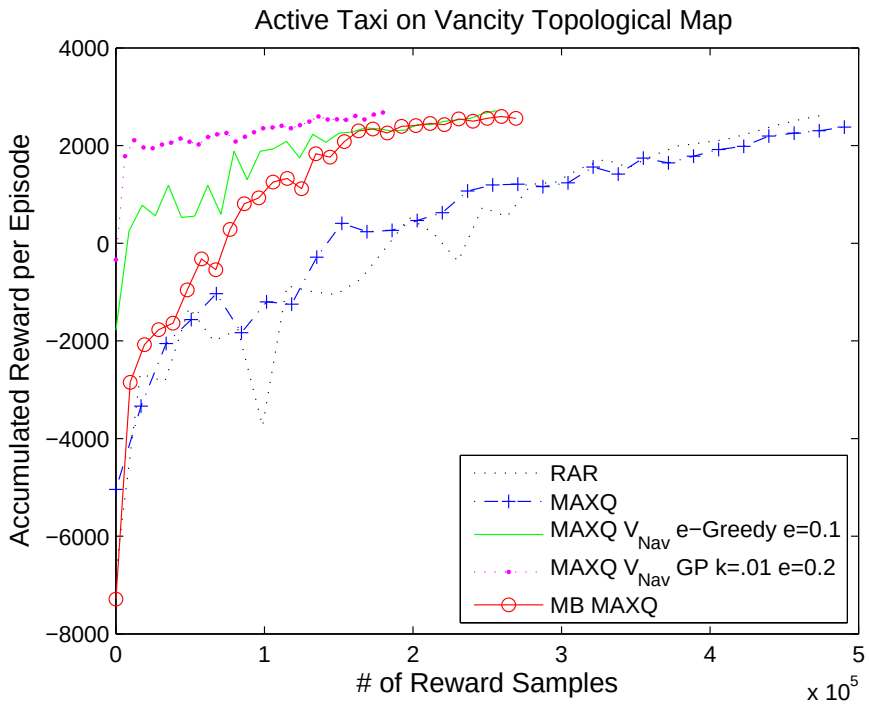

Figure 4.3: **Parameterized V_{Nav} vs. RAR and (MB-)MAXQ:** These experiments compare the original Recursive Average Reward (RAR) and MAXQ (discounted reward) algorithms against Model-Based MAXQ and the parameterized V_{Nav} path learner with $\mathbf{x} = \{TaxiLoc, WP\}$, learning rate $\alpha = 0.4$, and discount rate $\gamma = 0.999$.

4.2 Path Finding

Subsequently, the best parameters are fixed inside the *Follow* task, and we run the full hierarchical learners, with results in Figure 4.3. We averaged the results from 10 runs of RAR, MAXQ, MB-MAXQ, and the value learning Algorithm 3 (V_{Nav}) applied only to the *Navigate* task (with the rest of the hierarchy using MAXQ). All the experiments use the hierarchical task model presented in Section 2.3. Each reward timestep lasts 0.3 seconds, so

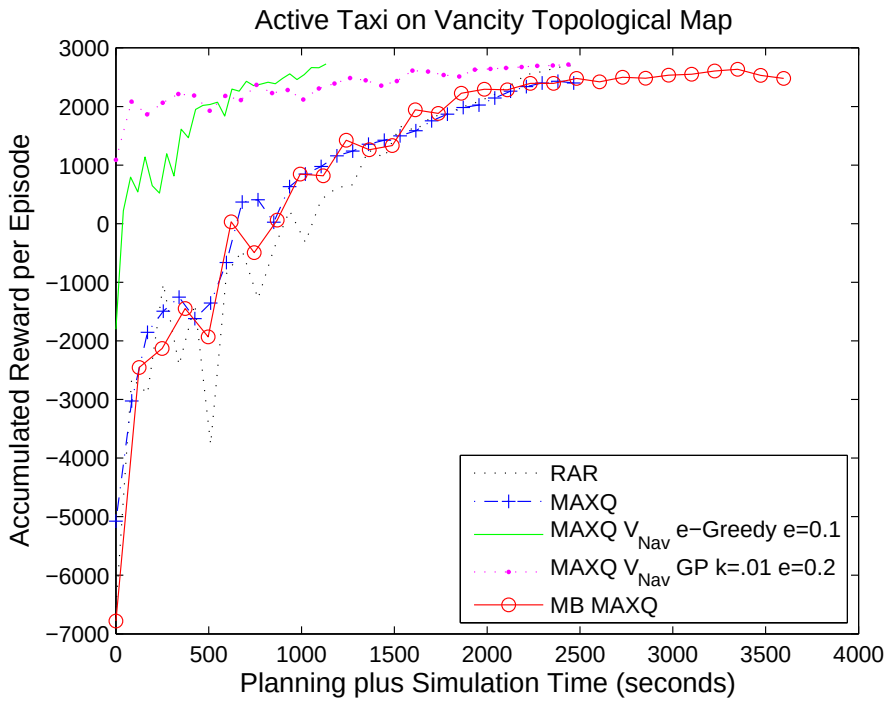

Figure 4.4: **Parameterized V_{Nav} vs. RAR and (MB-)MAXQ:** The more complex algorithms require more computation time to reduce the number of samples from the real world. It is up to the designer to evaluate which is more costly, world sampling, or computation, and choose the right algorithm accordingly.

the fastest learner, V_{Nav} GP with $\epsilon = 0.2$ drove for ~ 4 hours (simulated) real-time at ~ 60 km/hr before finding a good approximation of the V_{Nav} value function. For comparison, to reach the same performance level, it took roughly ~ 11 hours of driving for V_{Nav} $\epsilon - greedy$, and ~ 30 hours for MAXQ and RAR.

To make the experiments bearable, the simulator was sped up to $100\times$ real-time. Refer to Figures 4.5 and 4.6 for an intuition of how fitting the GP over the sampled values transfers observations to adjacent areas of the

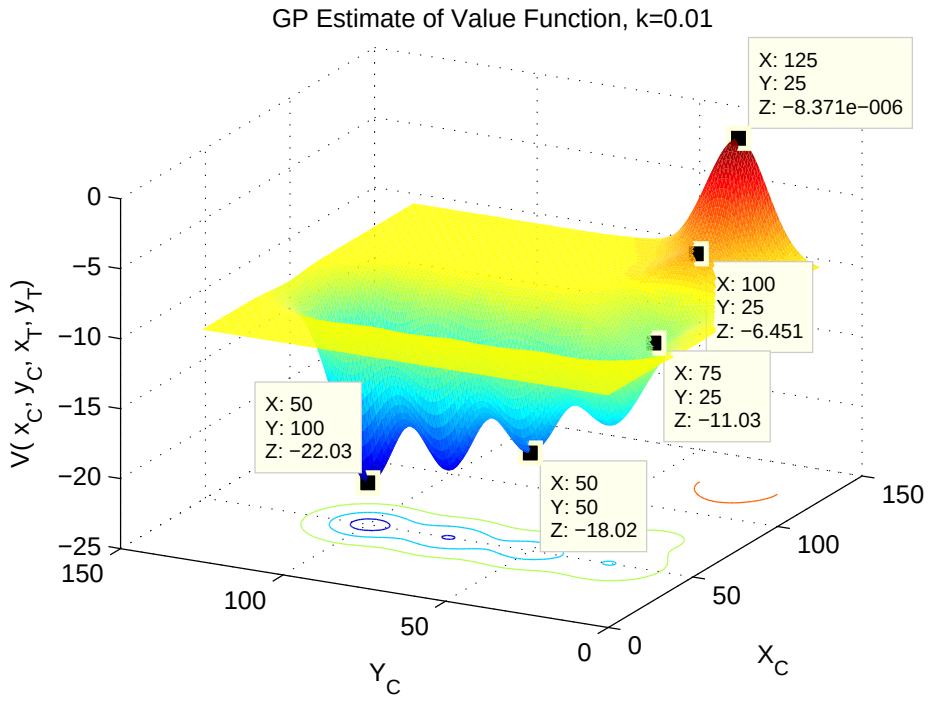

Figure 4.5: **Gaussian Process estimate of V_{Nav}.** A small kernel value narrows the 'footprint' of an observation. The topological map in this graph is a 5x5 grid with $25m$ between intersections. The Datatips highlight 5 of the 7 observation samples along the trajectory. The start of the trajectory is $(50, 100)$ and the Target point $(X_T, Y_T) = (125, 25)$.

state space. Just 7 observations (only 5 are highlighted) results in a good estimate of the value function in areas of the state space that matter. Active algorithms use these estimates of $V(\mathbf{x}) \approx \mu(\mathbf{x})$ and uncertainty $\sigma(\mathbf{x})$ to spend more time exploring areas of high-interest. While this application is specific to navigating a topological map, the algorithm is general and can be applied to any continuous state spaces of reasonable dimensionality.

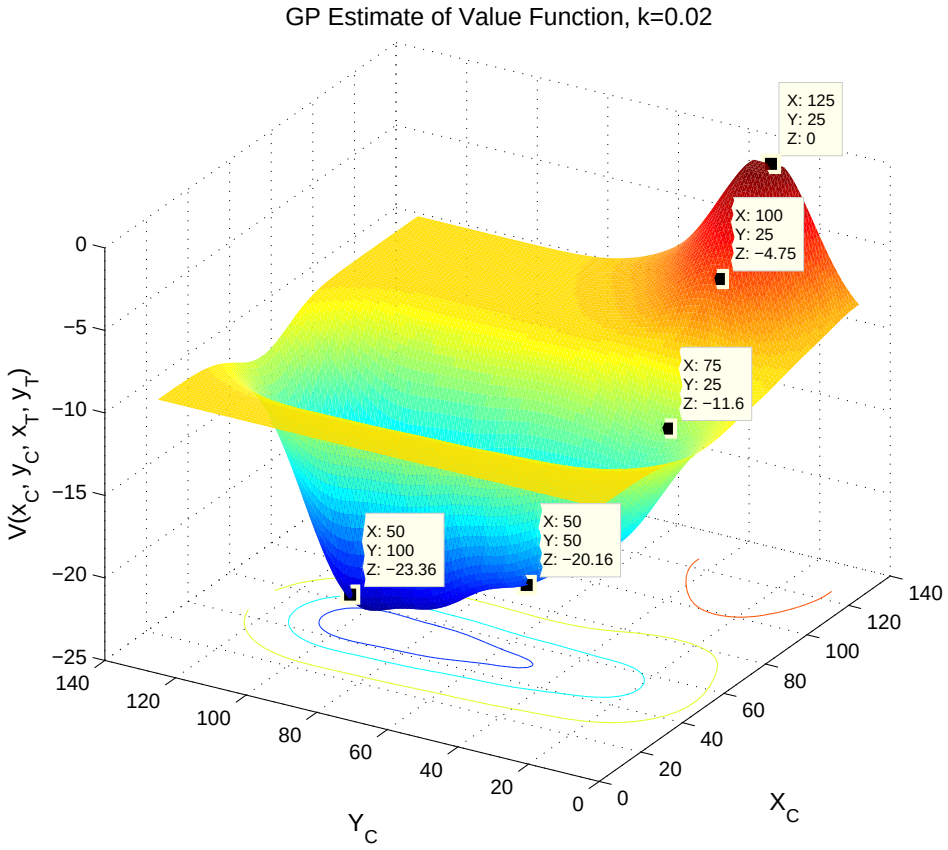

Figure 4.6: **Gaussian Process estimate of V_{Nav}.** When the gaussian kernel parameter k is increased, observations affect the neighboring space more. In our experiments, the kernel is cooled linearly between two roughly tuned bounds. It is isotropic (of equal scale in all state dimensions), but this limitation could be lifted, with different k values for different dimensions. The start and end points of the episode are the same as Figure 4.5.

4.3 Manual Tuning

One of the goals of this project was to find ways of reducing the amount of manual parameter tuning. While the algorithms investigated were able to learn desired behaviours, they still require some manual tuning. The usual approach to designing game AI behaviours has been to manually hardcode a policy. Applying reinforcement learning algorithms shifts the work from coding the policy, to designing and tuning a reward function describing the desired behaviour, and tuning the parameters controlling the algorithms.

Tuning a reward function can be tricky depending on the required outcome. Trajectory tracking is a fairly simple behaviour, that still required normalizing the individual error measures before summing them in equation 4.1. The factors were chosen empirically to normalize the different units to a relatively common measurement, simplifying the ability to weigh different features (for example preferring to minimize yaw error over action magnitudes). This reward function combined with the parameterized policy in Figure 3.1 enables automatic policy parameter learning for a wide variety of cars. The active policy search learner using GPs, with the same reward function, quickly found good policies for controlling both a high-performance race car and an off-road dune buggy.

The parameters controlling the active GP learner are the exploration rate ϵ, a multiplier on the uncertainty $\sigma(\mathbf{x})$ from equation 3.11, and the size of the Gaussian width kernel k used when computing the correlation matrix $\mathbf{K}$. In our experiments, both of these parameters are cooled linearly between roughly tuned bounds, initially favoring larger exploration

and interpolation to unobserved areas of the state or parameter space, and focusing over time to a narrower kernel and exploitation of observations, for fine-tuning. Minimal tuning was required to set the bounds of e and k for good results.

The three parameters controlling discrete learning in the task hierarchy are the discount rate γ, the learning rate α and the exploration rate ϵ. Since a trajectory length can be fairly large in our case (for example driving a few kilometers with a reward sample at every 0.3 seconds), the discount rate was set quite high, $\gamma = 0.999$, to value rewards received far into the future. The learning rate α controls the stability of policy updates during learning, and can be reduced over time to freeze the policy. In a highly stochastic environment, this value should be low, to reduce the noise in the learned value function. Since our application is mostly deterministic, we fixed $\alpha = 0.4$. Finally, the exploration rate ϵ was fixed to 0.1. The problem of tuning these parameters have spawned whole sub-fields of research within the RL community, which we do not cover in this thesis.

Chapter 5

Discussion

5.1 Related Work

Extending MAXQ with model-based updates is closely related to the Hierarchical Average Reward Learning (HARL) algorithm [25]. It also maintains an empirical transition model, and uses it to update the average gain of a subtask. In comparison, the hierarchical average reward learners of [10] are model-free, and use iterative temporal updates to compute the values of tasks. It has been demonstrated that model-based learners are quicker because they make better use of observations. Maintaining an empirical model allows for predictive planning, such as in Real Time Dynamic Programming (RTDP) applied to flat policies [26].

In our model-based solution, we have combined ideas from all of the mentioned papers, and applied it to the hierarchical discounted reward setting: while a task executes, we update the empirical model and also apply the original MAXQ temporal updates (equation 3.6). After the task completes, we run a set number of value update iterations similar to RTDP. Running the value iteration algorithm to completion after every task termination is too time-consuming, and many methods exist of selecting a subset of states to update, one of them being prioritized sweeping [2]. In our implemen-

tation, we opted for the simple heuristic of updating the states along the task's most recently executed trajectory, as well as any states with observed transitions to states in the current trajectory. This heuristic is only applicable in the discrete value learning case, whereas we used active policy search to optimize the policy parameters in the *Follow* task with continuous states and actions.

The related approach of [9] applies a factored variant of the $RMax$ [7] algorithm to the MAXQ value decomposition, using an empirical model to run a value iteration planning update when sufficient observations have been gathered. Their application is limited to deterministic domains, whereas the Model-Based MAXQ algorithm described in section 3.1.3 is applicable to stochastic domains as well.

Reinforcement learning has been applied successfully to simple games such as backgammon, and checkers, and has recently found limited success for specific applications in some published games. For example the creature in *Black & White* builds a decision tree representing the player, and uses it to act in accordance to the player's wishes. In *Age of Empires*, offline optimization is used to balance the settings of different troops and enemy types. Please refer to the *AI Game Programming Wisdom* series of books for an excellent reference of practical knowledge gathered from a large number of industry experts. The main hurdle of applying RL is its traditionally poor ability to scale to large state spaces, and slow learning rates. These shortcomings can be ameliorated by adopting more complex and flexible function approximation techniques, exploration policies and hierarchical task decompositions.

5.2 Future Work

We have not implemented support for multiple agents and traffic, but the task hierarchy will make this easier. A high-level task can be interpreted as an agent, with its own subtask decomposition and policy. By implementing various communication methods, the agents can share observed knowledge [10] and possibly coordinate actions through a global synchronizer [18]. We also assume full observability of the state space, which is acceptable in a simulated world, but not in the real one. An internal belief model would be updated from noisy observations made by limited sensors, from which the agents predict the true state of the world [12].

The recently introduced Interactive Dynamic Influence Diagrams [22] could be used to provide prior knowledge of the belief model an agent holds over other agents and the world. Algorithms that leverage the structure of the state dynamics encoded in an I-DID can run faster than otherwise, similar to the gains resulting from decomposing actions into sub-actions. We have also not addressed automatic learning of the task decomposition (and similarly learning the structure of an I-DID). This problem is very difficult, and currently remains unsolved.

An extension of the work that would be of great value to video game designers is to adopt inverse RL techniques to learn the reward functions in these hierarchical domains. This would enable the design of more realistic game behaviours, from demonstrations of human players.

5.3 Conclusion

The thesis demonstrates an integration of three methods from the reinforcement learning community for scaling RL to larger problems: i) hierarchical policy decomposition for state and temporal abstraction, ii) stochastic model-based planning, and iii) Bayesian active learning for continuous policy optimization. Although the continuous TORCS taxi domain is mostly deterministic, the algorithms presented are applicable to stochastic domains as well. The methodology is not limited to games, or simulations. It is directly applicable to other domains, including animation, hybrid control, planning and robotics.

Dealing with high-dimensions is one of the main challenges of the Bayesian active learning approach. However, this problem might not be so severe in HRL because, in this setting, the policy is partitioned into several lower-dimensional decision modules that can be trained separately. We showed that the method performed well in a module with fifteen parameters. In this sense, there seems to be a perfect marriage between Bayesian active learning and hierarchical decompositions.

The work presented in this thesis attempts to bridge the gap between software practices in the video games industry and what HRL can deliver. To apply the algorithms, a designer describes the desired behaviour through a reward function, and designs a hierarchical, parameterized policy of sufficient complexity to encode the solution. Subsequently, the learning algorithms optimize the parameters and value functions to maximize the specified reward. This process can save considerable time if the same policy and

reward functions are applicable to many instances of agents, with different (but functionally similar) properties (such as different cars, or character motion parameters).

Fully general learning algorithms that make no use of prior knowledge or domain properties tend to very slow in practice. A hierarchical policy allows the flexibility of partitioning the problem into sub-problems to which we can apply custom learning algorithms, state, reward and policy specifications. Hierarchical RL is key for scaling to hybrid continuous and discrete, high-dimensional action and state spaces.

Bibliography

[1] David Andre. *Programmable Reinforcement Learning Agents*. PhD thesis, University of California at Berkley, 2003.

[2] David Andre, Nir Friedman, and Ronald Parr. Generalized prioritized sweeping. In *NIPS '97: Proceedings of the 1997 conference on Advances in neural information processing systems 10*, pages 1001–1007, Cambridge, MA, USA, 1998. MIT Press.

[3] Andrew G. Barto, Steven J. Bradtke, and Satinder P. Singh. Learning to act using real-time dynamic programming. *Artificial Intelligence*, 72(1-2):81–138, 1995.

[4] Andrew G. Barto and Sridhar Mahadevan. Recent advances in hierarchical reinforcement learning. *Discrete Event Dynamic Systems*, 13(1-2):41–77, 2003.

[5] Jonathan Baxter, Peter L. Bartlett, and Lex Weave. Experiments with infinite-horizon, policy-gradient estimatio. *JAIR*, 15:351–381, 2001.

[6] B Betro. Bayesian methods in global optimization. *Journal of Global Optimization*, 1:1–14, 1991.

[7] Ronen I. Brafman and Moshe Tennenholtz. R-max - a general polynomial time algorithm for near-optimal reinforcement learning. *J. Mach. Learn. Res.*, 3:213–231, 2003.

[8] Thomas G. Dietterich. Hierarchical reinforcement learning with the MAXQ value function decomposition. *JAIR*, 13:227–303, 2000.

[9] Carlos Diuk, Alexander L. Strehl, and Michael L. Littman. A hierarchical approach to efficient reinforcement learning in deterministic domains. In *AAMAS '06: Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, pages 313–319, New York, NY, USA, 2006. ACM Press.

[10] Mohammad Ghavamzadeh. *Hierarchical Reinforcement Learning in Continuous State and Multi-agent Environments*. PhD thesis, University of Massachusetts Amherst, 2005.

[11] C Guestrin, A Krause, and A P Singh. Near-optimal sensor placements in Gaussian processes. In *International Conference on Machine Learning (ICML)*, 2005.

[12] N. Roy J. Pineau and S. Thrun. A hierarchical approach to pomdp planning and execution. In *ICML Workshop on Hierarchy and Memory in Reinforcement Learning*, June 2001.

[13] David R Jones, C D Perttunen, and B E Stuckman. Lipschitzian optimization without the Lipschitz constant. *Journal of Optimization Theory and Applications*, 79(1):157–181, October 1993.

[14] Donald R. Jones, Matthias Schonlau, and William J. Welch. Efficient global optimization of expensive black-box functions. *Journal of Global Optimization*, 13:455–492, 1998.

[15] H J Kushner. A new method of locating the maximum of an arbitrary multipeak curve in the presence of noise. *Journal of Basic Engineering*, 86:97–106, 1964.

[16] Daniel Lizotte, Tao Wang, Michael Bowling, and Dale Schuurmans. Automatic gait optimization with Gaussian process regression. In *IJCAI*, 2008.

[17] M Locatelli. Bayesian algorithms for one-dimensional global optimization. *Journal of Global Optimization*, 10:57–76, 1997.

[18] B. Marthi, D. Latham, S. Russell, and C. Guestrin. Concurrent hierarchical reinforcement learning. *IJCAI*, 2005.

[19] A Y Ng and M I Jordan. Pegasus: A policy search method for large MDPs and POMDPs. In *UAI*, 2000.

[20] Andrew Ng, H. Jin Kim, Michael I. Jordan, and Shankar Sastry. Autonomous helicopter flight via reinforcement learning. In *NIPS*, 2003.

[21] Ronald Edward Parr. *Hierarchical control and learning for markov decision processes*. PhD thesis, 1998. Chair-Stuart Russell.

[22] Kyle Polich and Piotr Gmytrasiewicz. Interactive dynamic influence diagrams. In *Autonomous Agents and Multi-Agent Systems.*, 2006.

[23] A. Doucet R. Martinez-Cantin, N. de Freitas and J.A. Castellanos. Active policy learning for robot planning and exploration under uncertainty. In *In Robotics: Science and Systems (RSS)*, 2007.

[24] Carl Edward Rasmussen and Christopher K I Williams. *Gaussian Processes for Machine Learning*. MIT Press, Cambridge, Massachusetts, 2006.

[25] Sandeep Seri and Prasad Tadepalli. Model-based hierarchical average-reward reinforcement learning. In *ICML '02: Proceedings of the Nineteenth International Conference on Machine Learning*, pages 562–569, San Francisco, CA, USA, 2002. Morgan Kaufmann Publishers Inc.

[26] Alexander L. Strehl, Lihong Li, and Michael L. Littman. Incremental model-based learners with formal learning-time guarantees. In *In the proceedings of the 22nd Conference on Uncertainty in Artificial Intelligence*, 2006.

[27] S Streltsov and P Vakili. A non-myopic utility function for statistical global optimization algorithms. *Journal of Global Optimization*, 14:283–298, 1999.

[28] Richard S. Sutton, Doina Precup, and Satinder P. Singh. Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112(1-2):181–211, 1999.

[29] S Tong and D Koller. Active learning for structure in Bayesian networks. In *IJCAI*, pages 863–869, 2001.

[30] Bernhard Wymann, Christos Dimitrakakis, and Charalampos Alexopoulos et al. The open racing car simulator (http://torcs.sourceforge.net/).

[31] A Zilinskas and J Zilinskas. Global optimization based on a statistical model and simplicial partitioning. *Computers and Mathematics with Applications*, 44:957–967, 2002.