Memory-based Modeling and Prioritized Sweeping in Reinforcement Learning

Thijs Ramakers

Master of Science Thesis

Delft University of Technology

Delft Center for Systems and Control

Memory-based Modeling and Prioritized Sweeping in Reinforcement Learning

Master of Science Thesis

For the degree of Master of Science in Systems and Control at Delft University of Technology

Thijs Ramakers

August 20, 2010

Faculty of Mechanical, Maritime and Materials Engineering (3mE) · Delft University of Technology

Delft University of Technology
Department of
Delft Center for Systems and Control (dcsc)

The undersigned hereby certify that they have read and recommend to the Faculty of Mechanical, Maritime and Materials Engineering (3mE) for acceptance a thesis entitled

Memory-based Modeling and
Prioritized Sweeping in
Reinforcement Learning

by

Thijs Ramakers

in partial fulfillment of the requirements for the degree of

Master of Science Systems and Control

Dated: August 20, 2010

Supervisor(s): Prof.dr. R. Babuška

Dr.ir. G. Lopes

Ir. E. Schuitema

Reader(s): Dr.ir. M. Wisse

Abstract

Reinforcement Learning (RL) is a popular method in machine learning. In RL, an agent learns a policy by observing state-transitions and receiving feedback in the form of a reward signal. The learning problem can be solved by interaction with the system only, without prior knowledge of that system. However, real-time learning from interaction with the system only, leads to slow learning as every time-interval can only be used to observe a single state-transition. Learning can be accelerated by using a Dyna-style algorithm. This approach learns from interaction with the real system and a model of that system simultaneously. Our research investigates two aspects of this method: Building a model during learning and implementing this model into the learning algorithm.

We use a memory-based modeling method called Local Linear Regression (LLR) to build a state-transition model during the learning process. It is expected that the quality of the model increases as the number of observed state-transitions increases. To assess the quality of the modeled state-transitions we introduce prediction intervals. We show that LLR is able to model various systems, including a complex humanoid robot.

The LLR model was added to the learning algorithm to generate more state-transitions for the agent to learn from. We show that an increasing number of experiences leads to faster learning. We introduce Prioritized Sweeping (PS) and Look Ahead Dyna (LA Dyna) as possibilities to use the model more efficiently. We show how prediction intervals can be used to increase the performance of the various algorithms. The learning algorithms were compared using an inverted pendulum simulation, which had to learn a swing-up control task.

Table of Contents

List of Figures

List of Tables

Acknowledgments

Just like reinforcement learning, scientific research has a trial-and-error nature. The research for this master's thesis was no exception. Although a project plan was made at the start, the obtained results forced the project in certain directions that were not always foreseen in the beginning. Every choice made during research led to new questions and new options.

I would like to thank my supervisors for providing lots of ideas and suggestions, while allowing me the freedom to make my own choices in the end. I owe many thanks to Prof.dr. Robert Babuška for his broad knowledge, numerous ideas and useful feedback. I would also like to thank my daily supervisors dr.ir. Gabriel Lopes, for his useful suggestions and enthusiasm, and ir. Erik Schuitema, for his advice and for placing this research in a practical context.

Finally, I would like to thank my family, Margot and friends. Although they could not give me answers to my research questions, they did provide the support that was needed for finalizing my master.

Delft, University of Technology Thijs Ramakers
August 20, 2010

Chapter 1

Introduction

Reinforcement Learning (RL) is a machine learning method inspired by human and animal psychology. RL can be used in a variety of applications. It is used in optimization problems or complex problems in general, for which a solution might be difficult to find. Autonomous robots that have to act in unknown or complex environments can use RL to learn or adapt their behavior. RL is based on receiving rewards for good actions or penalties for bad actions.

RL can solve various problems, ranging from optimization problems to actual control applications. Each application has its own solution methods and associated difficulties, as we describe in Chapter 2. In order to identify useful methods, it is important to describe the particular setting we are considering in this thesis.

RL has been successfully applied to easy to moderately complex problems, often assuming discrete state- and action-variables. However, many possible real-world applications are more complex and have high-dimensional, continuous state-variables. In this work we search for solutions to speed-up reinforcement learning that can be used on real systems. A typical system could be a humanoid robot that has to learn how to walk or an autonomous robot that has to navigate an unknown environment. Such complex, real-world systems have a number of important properties that make RL challenging. We will describe these properties here, because they are of major influence on the RL methods that we chose.

Some systems can be described by a discrete set of states. Such systems can be found in optimization problems in game theory and various logistic problems. RL can solve such problems by using algorithms that visit each state continuously. Dynamic systems however, will typically have a continuous, high dimensional state-space. For such systems it is impossible to visit all states and it takes long to visit a substantial part of the state-space. Therefore, solving a RL problem on such systems is difficult.

In practice, a complete model of the system is not always available a priori. We suppose that at the beginning of the learning process, there is no knowledge of the system available to the learning agent. This includes both the system dynamics and the reward function. The only 'knowledge' available to the agent is the state-vector at a certain moment in time and the set of available control actions. This limitation implies that methods that use full knowledge of the environment are excluded from this research. Also methods that supply a priori knowledge to the system (either by changing the reward structure, or by supplying some initial control policy) will not be discussed. We are interested in model-learning methods. These methods start with no a priori knowledge but use the experience gained during learning to build a model. The built model can than be used to increase learning speed. Such 'model-learning' methods will be investigated in depth in this work. They are interesting, as they can easily be used on a variety of complex systems.

Finally, we are interested in using RL for control of real systems. Application on real systems leads to specific problems. Two important consequences will be taken into account. The first aspect is the presence of noise. Disturbances can never be neglected in real systems as it will affect the quality of the measurements available to the agent. The second aspect is sampling. Control of a real system implies that a control action has to be sent to the system at every sampling interval. This means that the learning algorithm has limited time available to do calculations and therefore computation speed is of great importance in real-time experiments.

Considering the properties described, we can formulate the following research question:

> How can we on-line build and use a model in order to speed up a learning process on complex systems?

The methods presented in this work are suitable to be applied on complex systems. However, throughout this thesis, we will show results that were obtained using easier settings. For example, we will use 1-dimensional functions, simulations and simple learning problems.

This report is structured as follows. We start with an introduction to RL in Chapter 2. We introduce the general framework and discuss possible solution methods. In view of the setting we have sketched, we argue to use model-learning to solve the RL problem. This breaks the problem down into two parts. The first part is modeling, which is discussed in Chapter 3. We will argue that memory-based modeling is interesting in a RL setting and will introduce Local Linear Regression (LLR) as a particular method. The second part is the RL algorithm itself. In Chapter 4 we will combine the LLR model with several RL algorithms. We will investigate Prioritized Sweeping (PS) as a method to increase the speed of learning.

Chapter 2

Reinforcement learning

2-1 Introduction

Reinforcement Learning (RL) was inspired by human and animal learning [1]. It is derived from the idea that behavior can be learned by receiving rewards or punishments for good or bad actions. The result of random actions in different situations is used to decide on actions in future situations that are similar. RL has a distinct 'trial and error' nature. Random actions are being tried until the best action for a certain situation is found. In this chapter we introduce the RL framework and introduce various ways in which a RL problem can be solved. After a general overview of the RL method, we give a formal description of the framework. Finally, several solution methods are introduced and some interesting research possibilities are indicated.

In RL, learning is done by an *agent* in close interaction with its *environment* (Figure 2-1). The agent is the 'learner' and the 'decision maker'. It decides on what *action* to take and evaluates the result of this action. The agent interacts with the environment and receives feedback in the form of a scalar *reward* that indicates how 'good' the previous action was. In control applications, the agent will be represented by a computer program following a certain learning algorithm and the environment will be a system (either a real setup or a mathematical model) that reacts to the actions the agents takes. The environment also gives a reward signal to the agent. We are focusing on control applications, so we assume that the decisions of the agent take place at discrete time intervals.

In many problems, a certain *goal* state exists that the agent must try to reach. If this goal state is reached, the learning is stopped and the process is repeated. These separate experiments are called *episodes* or *trials*. The reward function can be defined in a number of ways. Usually, the reward will be positive if the goal is reached and negative or zero in all other cases. Note that the reward only qualifies the direct result of a certain action and not the long-term effects of that action. The task of the agent

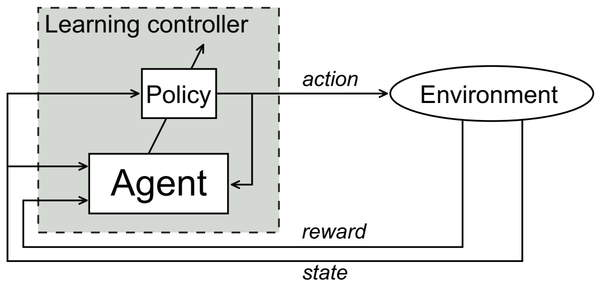


Figure 2-1: A schematic overview of the RL framework. The agent interacts with an environment and learns using a scalar reward.

will be to maximize the total (accumulated) reward, which requires the agent to asses the long-term effects of actions.

After a learning process, the agent 'knows' how it can reach the goal. To be more precise, it knows what action it must take in every state in order to maximize the total reward. The set of rules that describe what action the agent will take in every state is called the *policy*.

So, in summary, the goal of a RL system is to find an optimal policy so that the total reward received during a trial is maximized. This is learned by an agent in close interaction with its environment.

2-2 MDP framework

In the previous section we introduced the general idea behind RL and some important terms used in RL. In this section, a more formal description of a RL problem will be given.

A requirement in RL is that the environment should have the Markov property. This means that if the current state is known, then the transition to the next state is independent of all previous states. However, this transition may still be either deterministic or stochastic. The state therefore contains all the information that describes the environment. In every state the agent has to make a decision based on an environment that has the Markov property, the learning task is therefore called a Markov Decision Process (MDP). When the state is only partially observable for the agent, such a task is called a Partially Observable Markov Decision Process (POMDP). In this work we only consider situations in which the full state is available to the agent.

Formally a MDP is a 4-tuple $(\mathcal{S}, \mathcal{A}, T, \rho)$. $\mathcal{S}$ is a finite set of states s and $\mathcal{A}$ a finite set of actions a. $T : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ is the state transition function which defines the probability of getting from state s to s' when action a is taken. ρ: $\mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ defines the immediate reward r for getting from state s to s' after taking action a.

The policy is a mapping π: $\mathcal{S} \to \mathcal{A}$ that assigns action selection probabilities to $\forall a \in \mathcal{A}$ in each state. The objective of the agent is to learn a policy that maximizes the expected total reward in the entire episode. The accumulated total reward is called the return. The expected return R_t after an action at time step t is:

$$R_t = E\left\{r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ...\right\} = E\left\{\sum_{t=0}^{\infty} \gamma^t r_{t+1}\right\}$$

where $\gamma \in (0, 1]$ is the discount factor. In other words, R_t is the sum of the expected reward for the current action and the discounted rewards of future actions when using the current policy. For problems with a finite time horizon, the upper bound of the summation is the final time step t_f. The discount factor can be viewed in two different ways. The first view is purely mathematical. It makes sure that if the time horizon is infinite, the total reward converges to some finite value and will not go to infinity. The second view is more intuitive. By using a discount factor, rewards in the near future have an exponentially greater effect than rewards received in the far future. Because the discount factor is part of the learning task, the MDP is sometimes explicitly written as $(\mathcal{S}, \mathcal{A}, T, \rho, \gamma)$ to emphasize the discounting.

2-2-1 Value functions

We can now proceed to assign a value to every state. The value of a state $V(s)$ is the expected total reward in that state under a certain policy. This value function is therefore defined as:

$$V^\pi(s) = E^\pi\left\{R_t \mid s_t = s\right\}$$

where $E^\pi\{\cdot\}$ is the expected value when following policy π. Because the goal of the agent is to learn which action is best in every state, it is also useful to define a value function that assigns a value to state-action pairs. To this end, the action-value function $Q(s, a)$ is defined:

$$Q^\pi(s, a) = E^\pi\left\{R_t \mid s_t = s, a_t = a\right\}$$

which also depends on the policy followed. Typically, for problems in which the agent knows how it can reach a certain state, the value function $V(s)$ is convenient to use. For problems in which the agent does not know this, the action-value function $Q(s, a)$ can be used to explicitly store the reward for certain actions. In most control problems the action-value function is used.

2-2-2 Optimal policy

As mentioned before, the goal of the agent is to take actions which will result in the highest return R. In other words, it has to find a policy π that maximizes the value function $V^\pi(s)$ (or equivalently, $Q^\pi(s, a)$). This policy, which is better than all other policies, is called an optimal policy π^*. The value function that belongs to this policy is the optimal value function $V^*(s)$ and is defined as:

$$V^* = \max_\pi V^\pi(s) \tag{2-1}$$

An optimal policy has an optimal action-value function $Q^*(s,a)$:

$$Q^*(s,a) = \max_{\pi} Q^{\pi}(s,a) \tag{2-2}$$

which can also be written in terms of the optimal value function:

$$Q^*(s,a) = E\left\{r_{t+1} + \gamma V^*(s_{t+1}) \mid s_t = s, a_t = a\right\}$$

So, if the optimal (action-)value function is known, the optimal policy is the argument that maximizes (2-1) or (2-2):

$$\pi^* = \arg\max_a Q^*(s,a) = \arg\max_a E\left\{r_{t+1} + \gamma V^*(s_{t+1}) \mid s_t = s, a_t = a\right\}$$

Because this policy selects the action for which the (action-)value function is maximum, this policy is called a greedy policy.

Important for a large number of solution techniques are the so called Bellman equations. For the value function, these can be written as:

$$\begin{aligned} V^{\pi}(s) &= E^{\pi}\left\{\sum_{t=0}^{\infty} \gamma^t r_{t+1} \mid s_t = s\right\} \\ &= E^{\pi}\left\{r_{t+1} + \gamma \sum_{t=0}^{\infty} \gamma^t r_{t+2} \mid s_t = s\right\} \\ &= E^{\pi}\left\{r_{t+1} + \gamma V^{\pi}(s') \mid s_t = s, s_{t+1} = s'\right\} \\ &= \sum_{s'} T(s,a,s')\left[\rho(s,a,s') + \gamma V^{\pi}(s')\right] \end{aligned}$$

Recall that $T(s,a,s')$ is the transition probability from s to s' under action a and $\rho(s,a,s')$ is the scalar reward for that transition. Using Eq. (2-1) we can derive the Bellman optimality equation for the value function:

$$V^*(s) = \max_a \sum_{s'} T(s,a,s')\left[\rho(s,a,s') + \gamma V^{\pi}(s')\right] \tag{2-3}$$

and the equivalent Bellman optimality equation for the action-value function:

$$Q^*(s) = \sum_{s'} T(s,a,s')\left[\rho(s,a,s') + \gamma \max_{a'} Q^*(s',a')\right] \tag{2-4}$$

2-3 Solution methods

Several methods for solving RL problems exist. 'Solving' in a RL context means computing the optimal policy, which is generally derived from the optimal (action-)value function. The methods can be divided into three different types: model-based, model-free and model-learning methods. Model-based methods assume that a complete model of the environment is available. Model-free methods start with no knowledge of the environment at all and learn from interaction with the environment. Model-learning methods build a model during the learning process. Although prior knowledge of the environment can be incorporated, the main feature of this method is the continuous improvement of the model during learning.

2-3-1 Model-based methods

Model-based RL methods assume that the MDP is completely known and available to the agent. Not only is the state fully observable by the agent, also the state transition function T and reward function ρ are known. This means that the agent has a perfect and complete model of the environment and its dynamics. The agent can calculate for every state the possible next states and their rewards. The Bellman optimality equations Eq. (2-3) and Eq. (2-4) lead to a system of equations with only $V^*(s)$ and $Q^*(s,a)$ as unknown variables and can therefore be solved. The need for a perfect model and the computational effort needed to solve these equations, makes the practical usability for these methods limited. However, these algorithms form the base of other methods. These model-based solution techniques are called Dynamic Programming (DP) [2]. We will briefly describe the Q-iteration method.

Q-iteration

For control applications, using the action-value function $Q(s,a)$ is usually preferred over the value function $V(s)$. Q-iteration is a model-based solution method that iteratively computes the optimal action-value function $Q^*(s,a)$. The algorithm continuously visits all state-action pairs (s,a) and updates the value of $Q(s,a)$ according to the reward. The Q-iteration method is shown in Algorithm 1. For discrete state-action spaces the algorithm will always converge to the optimal policy.

Algorithm 1 Q-iteration

```
1: Initialize $Q_0(s,a)$, $\gamma$
2: $k \leftarrow 0$
3: repeat
4:     for each state $s$ do
5:         for each action $a$ do
6:             $s' \leftarrow T(s,a)$
7:             $Q_{k+1}(s,a) \leftarrow \rho(s,a) + \gamma \max_a Q_k(s',a)$
8:         end for
9:     end for
10:    $k \leftarrow k+1$
11: until Convergence
```

2-3-2 Model-free methods

The methods presented in Section 2-3-1 assume that a full model of the environment is available. In practice, this is often not the case. Model-free methods can be used to solve a RL problem without using a model. These solution methods rely on interaction with the environment only. Model-free learning methods that learn from interaction with the environment are also known as *on-line* learning methods, as opposed to model-based methods which are also known as *off-line* methods.

Temporal Difference methods

Temporal Difference (TD) methods are on-line methods that update the value function every time step. At every time step t, the agent takes an action a for which it receives an immediate reward r. At every time step the value function is updated according to:

$$V(s_t) \leftarrow (1-\alpha_t)V(s_t) + \alpha_t \left(r_{t+1} + \gamma V(s_{t+1})\right)$$

with α_t the learning rate at time step t. The learning rate indicates how strong a new experience influences the current estimate of the value function. A frequently used notation for the TD update is:

$$V(s_t) \leftarrow V(s_t) + \alpha_t \delta_{V,t} \tag{2-5}$$

with $\delta_{V,t}$ the so-called TD-error. TD methods can also be easily altered to be used with action-value functions. The update rule then becomes:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha_t \delta_{Q,t} \tag{2-6}$$

with $\delta_{Q,t}$ the TD-error for the action-value function.

A problem that exists in model-free methods is the dilemma of exploitation versus exploration. The agent should use (exploit) experience gained to optimize the current policy. On the other hand, the agent should maintain its search in the state-space (explore) to find possible better policies. In fact, most proofs of the policy converging to the optimal policy only hold when the total state-space is constantly visited. Different strategies exist that describe how the agent should explore the state-space. ϵ-greedy is the most straightforward exploration method. The exploration rate $\epsilon \in [0,1]$ gives the probability of choosing an explorative action, instead of the current optimal (greedy) action. The exploration rate does not have to be constant during the learning task, but can be gradually decreased allowing less explorative actions as learning proceeds.

We now describe two of the most used TD algorithms that use Q-values: SARSA and Q-learning.

SARSA SARSA [3] is probably the most basic TD algorithm that uses $Q(s,a)$-values instead of $V(s)$-values for learning. As mentioned before, using state-action values is more convenient for control applications than storing state values only. It uses the update rule in Eq. (2-6) with the following TD-error:

$$\delta_{Q,t} = r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)$$

The SARSA algorithm uses five values $(s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1})$ to update Q which is the reason for the algorithm's name. The implementation of the SARSA algorithm is shown in Algorithm 2.

Algorithm 2 SARSA

```
1: Initialize $Q(s, a)$
2: for each time step $t$ do
3:     $s \leftarrow$ Current state
4:     Select $a$ using policy
5:     Execute $a$ and observe $s'$, $r$
6:     $\delta \leftarrow r + \gamma Q(s', a') - Q(s, a)$          ▷ TD-error
7:     $Q(s, a) \leftarrow Q(s, a) + \alpha\delta$          ▷ Update $Q$ function
8: end for
```

Q-learning Q-learning [4] is also a TD method that estimates the action-value function $Q(s, a)$. As opposed to SARSA which is *on-policy*, Q-learning is an *off-policy* algorithm as it learns state-action values that are not necessarily on the policy that is followed. The Q-learning algorithm uses the same update rule as SARSA (Eq. (2-6)), but uses a different TD-error:

$$\delta_{Q,t} = r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)$$

The implementation of the Q-learning algorithm is the same as SARSA and both algorithms are encountered frequently in literature. SARSA is often preferred, because of better convergence properties, for examples when used with function approximators [5]. Q-learning is preferred when off-policy learning is desired.

2-3-3 Model-learning methods

We have introduced two types of RL methods. One based on the availability of a complete model of the environment and one based on the situation in which no model is present. As described in Chapter 1, we consider problems in which initially no model is present. For these cases, one would have to use a model-free method. Two questions arise: could we use past experience to estimate a model during learning? And can we then use this model to speed up the learning process? In this section we introduce model-learning methods that use observed state-transitions to build a model upon which model-based techniques can be applied. Methods that use combine model-free and model-based learning are called Dyna-style methods.

Dyna

Sutton [6] argues there are great similarities between model-based and model-free RL methods. Different methods could therefore be combined, which he calls: integrating *learning*, *planning* and *acting* (see Figure 2-2). Where the term planning is used to refer to off-line, model-based techniques (DP) and learning to on-line, model-free techniques (TD). Acting refers to on-line learning situations in which interaction with the environment (controlling actuators, reading sensor data) consumes an important part of every time step.

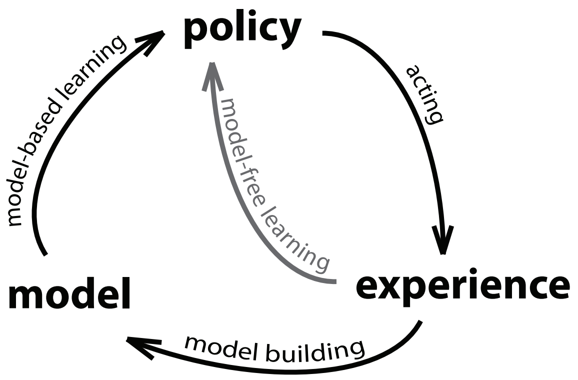


Figure 2-2: Integrating learning, planning and acting in the Dyna framework.

Dyna is the general term that is used for a class of algorithms that combine learning from real experiences with learning from simulated experiences generated using a (learned) model. We use the general term 'experience' to refer to a (s, a, s', r)-sample, which can be either obtained from observing the real system or by using the model. Various Dyna methods differ in the algorithms and the type of value function that are used. For example, Sutton introduces Dyna-Q and Dyna-PI that use Q-learning and Policy Iteration as learning algorithms respectively.

In a Dyna setting, every sampling interval starts with interaction with the real system. The observed state-transition and the resulting reward are used to update the value function and to update the model. Thereafter, a number of state-transitions are generated using the model. In short, the agent interacts with the system and the learned model alternatively (Figure 2-3).

The number of model-generated state-transitions can be either fixed or variable depending on the problem. In a real-time experiment for instance, the sampling interval can be used for model-based learning as long as there is time left. The ratio between real world and simulated samples can therefore vary. Algorithm 3 shows Dyna-style learning in pseudo-code. The number of model-generated state-transitions every time step is fixed to N_m. Notice that the first part of the algorithms is on-line learning (identical to model-free SARSA learning), while the second part is off-line learning (based on the model). We have not explicitly indicated the building of the model in the algorithm. Depending on the implementation this could happen after every time step or episode.

There is no restriction in the Dyna algorithm on the type of model that can be used. Also the needed accuracy of the model is not discussed in the literature. A general statement about the model can be made: the accuracy of the model should be high enough so that the model-based updates do not disturb the learning process. This statement is also true for real experiences, which can be noisy or disturbed measurements. The optimal ratio of real experience versus modeled experience might vary during the learning process. Early in the learning experiment, the model might still be inaccurate due to the low number of experiences. However, when the accuracy of the model increases, model-generated experiences might be a better representation of the

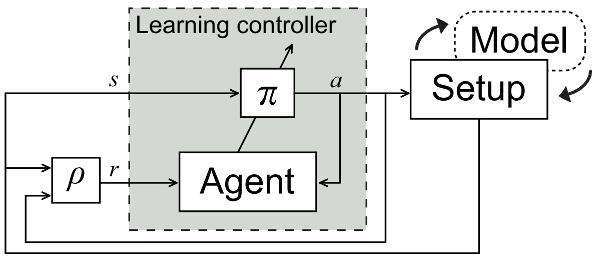


Figure 2-3: Schematic overview of the Dyna algorithm, which switches between interaction with the real system and its model.

Algorithm 3 Dyna

```
1: Initialize $Q(s,a)$ randomly
2: for each time step $t$ do
3:     $s \leftarrow$ Current state
4:     Select $a$ using policy
5:     Execute $a$ and observe $s'$, $r$
6:     $\delta \leftarrow r + \gamma Q(s',a') - Q(s,a)$
7:     $Q(s,a) \leftarrow Q(s,a) + \alpha\delta$
8:     loop $N_m$ times
9:         Select $(s,a)$ randomly
10:        $(s',r) \leftarrow Model(s,a)$
11:        $\delta \leftarrow r + \gamma Q(s',a') - Q(s,a)$
12:        $Q(s,a) \leftarrow Q(s,a) + \alpha\delta$
13:    end loop
14: end for
```

true system than noisy measurements in the case of a deterministic system. Furthermore, the model might not have to be accurate in the entire state-space. The agent might also be able to find the goal if it has an accurate model along trajectories directed towards the goal only.

The Dyna algorithm does not include a description of how to select the states that are updated in the planning. Is is generally assumed that the states are chosen randomly. However, this seems not to be the best choice for at least two reasons. Early in the learning process, most states will not contain any information regarding the goal state. The state's value will still be zero because the goal has never been reached from that state. Using these states in the planning will not lead to a useful value function update. Furthermore, in very large state-action spaces, not all states will be equally important for solving the learning problem. Only along trajectories towards the goal a good policy is required.

For these two reasons, it makes sense to concentrate the computational effort to areas of the state-action space where it is most effective. In other words, we want to prioritize important states in some way. A method that was introduced to update more important states first, is introduced next.

Prioritized Sweeping

Moore [7] and Peng [8] independently developed strategies to speed up the planning in Dyna by introducing a priority queue. Moore focused mainly on explicitly learning the state-transition model with all its transition probabilities. They named their method Prioritized Sweeping (PS). Peng used Dyna-Q as learning algorithm, and also used a priority queue to speed up learning. They named their method *Queue-Dyna*. In both approaches, a queue is maintained that determines the order in which states should be updated during planning. The two methods are almost identical. They only differ in which states they allow onto the priority queue. Where PS allows all predecessors which have a predicted change on the queue, Queue-Dyna allows only states that have a change greater than some threshold value. As mentioned, these differences are small and probably only influence memory usage in practice, but not learning speed.

The importance of a state that determines its place in the queue can be determined in several ways. The general way is to take the absolute value of the TD-error δ_t. The larger this error, the more the value for a certain state has changed and thus the more influence it has on the total value function. Algorithm 4 shows the PS algorithm. Notice that the algorithm consists of an on-line and off-line part, which is similar to the Dyna algorithm. An important step in the PS algorithm is determining lead-in states (Algorithm 4, line 11). Lead-in states are state-action pairs $(\bar{s}, \bar{a})$ that lead to state s. It is unclear how these states should be determined. In the literature one simply assumes that the model can produce them. In Chapter 4 we will describe our approach to this problem.

Algorithm 4 Prioritized Sweeping

```
 1: Initialize $Q(s,a)$ randomly and $Queue$ empty
 2: for each time step $t$ do
 3:     $s \leftarrow$ Current state
 4:     Select $a$ using policy
 5:     Execute $a$ and observe $s'$, $r$
 6:     $\delta \leftarrow r + \gamma Q(s',a') - Q(s,a)$
 7:     $Q(s,a) \leftarrow Q(s,a) + \alpha\delta$
 8:     Insert $(s,a)$ into $Queue$ with priority $|\delta|$
 9:     loop $N_m$ times
10:         Select $(s,a)$ from $Queue$
11:         for each lead-in state $(\bar{s},\bar{a})$ do
12:             $\delta \leftarrow r + \gamma Q(s,a) - Q(\bar{s},\bar{a})$
13:             $Q(\bar{s},\bar{a}) \leftarrow Q(\bar{s},\bar{a}) + \alpha\delta$
14:             Insert $(\bar{s},\bar{a})$ into $Queue$ with priority $|\delta|$
15:         end for
16:     end loop
17: end for
```

Most of the articles dealing with PS techniques report improved learning speed compared to standard Dyna [7], [8], [9], and also [10]. However, it has been reported that PS can also lead to sub-optimal solutions for some specific tasks. For example [11] introduces a navigation task with one goal and several sub-goals. Using PS in this task resulted in finding a sub-optimal solution. Grześ argues that this effect is due to the many sub-optimal solutions for the task. PS would lead to insufficient exploration in such cases. However, it is not clear how the type of exploration, the exploration rate and learning rate influence this effect.

2-4 Conclusions & research opportunities

In this chapter, three different types of solution methods to the Reinforcement Learning problem were presented. Model-based methods use an available model of the environment to iteratively calculate the optimal value function. Model-free methods are used in an on-line setting where no explicit model of the environment is available. A combination of these two leads to Dyna-style methods in which model-free and model-based learning techniques are combined. Prioritized Sweeping is an improvement of Dyna, as states are no longer updated randomly, but according to some priority measure.

As described in Chapter 1, many interesting systems do not have an explicit model available. Therefore, purely model-based methods are of limited use in practical applications. On the other hand, purely model-free methods need long learning times because they can only interact with the real system once in every time interval. Therefore, model-learning methods seem to be the most promising method for the applications we consider in this work. The question how to build a model during learning is investigated further in Chapter 3. In order to have the most profit of the learned model, PS seems to be an interesting technique to research. However, this technique has never been used on problems with a continuous state-space. The implementation of the model and PS in the learning algorithms are described in Chapter 4.

Chapter 3

Memory-based modeling

3-1 Introduction

As described in Section 2-3-3, using a state-transition model of a system can be useful in speeding up the learning process. Typically, such a model is not available for real systems. Hence, such a model has to be constructed. In this work, we focus on building a model during the learning process[1]. The only data available for building a model, are the transitions that are experienced by the system during learning. This approach has the virtue that the model can be adapted to possible changes in the system dynamics. This is an advantage for autonomously operating systems. Furthermore, there is no need for separate identification experiments (which could be difficult or expensive in real-world situations). Building a model using identification experiments and then applying them in a Reinforcement Learning (RL) setting is also possible and has been done successfully in the past [12], [13], [14].

In this chapter we argue that a memory-based approach to building a state-transition model is interesting in a RL setting. The method used is Local Linear Regression (LLR). This method assumes the state-transition model to be locally linear. Upon a query input, a nearest neighbor search is carried out and a linear model is fitted through the obtained set of nearest neighbors. The fitted model is used to estimate an output.

This chapter is organized as follows. First an introduction to memory-based modeling is given in Section 3-2. In Section 3-3 the Local Linear Regression method is introduced and some statistical methods are introduced. Finally Section 3-4 presents the results obtained with three different setups.

[1]In literature, some authors use the term model 'learning' to refer to the construction of a model. In this thesis, the term *learning* will be used for reinforcement learning methods only. The term *building* will be used to refer to the construction of the model. This in order to emphasize the difference between the reinforcement learning methods and modeling methods.

3-2 Memory-based modeling

Memory-based modeling is a class of modeling methods that memorizes all samples and delays the estimation of a model and output until a query is made. Since modeling is deferred to the moment a query is made, this approach is also known as lazy-learning. Its opposite is eager-learning, which adjusts the model for every new sample that is obtained. Eager-learning methods (such as artificial neural networks) assume that a global parametric model structure is known and adjust its parameters according to some error-measure. In general, eager-learning methods discard the data they were trained on after the parameters have been tuned.

In this research, the stored samples are state-transitions $(\mathbf{x}_t, \mathbf{u}_t) \rightarrow \mathbf{x}_{t+1}$, the input-query $(\mathbf{x}_q, \mathbf{u}_q) = (\mathbf{x}_t, \mathbf{u}_t)$ is a state-action pair at time t and the estimated output $\hat{\mathbf{x}}_{t+1}$ is the resulting next state. In order to avoid confusion and to comply with the usual terminology in modeling, we will use the terms (query) *input* for $(\mathbf{x}_t, \mathbf{u}_t)$ and *output* for $\mathbf{x}_{t+1}$, instead of referring to states explicitly.

Advantages Memory-based modeling is also known as nonparametric modeling, as no global (parametric) model structure is used. Instead, for every query a model is estimated that is valid in the vicinity of the query input. In general, one uses only samples close to the query input to estimate the model. This can lead to a model that is able to estimate local nonlinearities very accurately. When enough samples are available around a query point, estimations can be accurate locally, even if large parts of the state-space have rarely been visited. Therefore, memory-based modeling can lead to accurate estimates after relatively few observation. This is very useful in a RL setting, in which an accurate model is required although the number of observed state-transitions might be limited.

A second advantage is that no detailed knowledge about the system dynamics is needed to obtain good quality estimates. Obtaining a representative parametric model of a system might be difficult and the obtained global model might not represent local nonlinearities accurately. Memory-based models can even be used if the knowledge about the system to model is very limited. This makes memory-based modeling useful for a wide range of applications. This is an advantage for a RL agent that possibly has no a priori knowledge about the environment.

Compared to eager-learning, memory-based learning seems to be favorable in a real-time experiment in which new samples are obtained continuously. Incorporating new samples in memory-based methods is straightforward. It only involves storing the new sample. In an eager-learning method, incorporating new samples means re-tuning the model parameters. This typically involves some iterative search for nonlinear models. If the tuning of the parametric model is very expensive, a memory-based approach might be more favorable.

Disadvantages Estimating a model solely based on stored samples also has disadvantages. The main disadvantage is its sensitivity to noise. Since no model-structure is imposed on the data, noisy data or outliers can lead to locally inaccurate estimates. However, we can use several statistical tools to cope with noise and faulty data in order to increase the accuracy and reliability of the estimation.

Another disadvantage of memory-based methods is the need to store all samples. This leads to two problems. First, many observed state-transitions lead to a large dataset which requires large amounts of memory to store. A second problem is that the estimation of an output can become computationally heavy since the estimation is possibly based on all stored samples and has to be re-calculated for every input query. For these reasons, memory-based modeling methods in real-time applications need to have some sort of memory management in order to prevent the number of samples in the memory to become too high. The most basic form of memory management is limiting the number of samples that are stored in the memory by replacing the oldest samples with new ones once a memory limit is reached. This approach is also logical when the system dynamics gradually change in time.

3-3 Local Linear Regression

In this section we will introduce the LLR method that will be used throughout this report. LLR is a memory-based modeling method that fits a linear model to a set of data points. These points are a subset of all the stored points and are selected according to their similarity to a query input (see Figure 3-1). Typically a vector norm is used as a similarity measure, which is generally called a distance measure. The first step is to select the subset of the memory data that is closest to the query input. This search is called a K nearest neighbors search, as the K nearest samples are selected. This aspect of the method will be discussed first. The second step is fitting a linear model to the set of nearest neighbors, which will be discussed thereafter. In the last part of this section, several methods from the field of statistics will be used to assess the quality of the estimated model.

Alternatives In this work we fit a linear model to the set of nearest neighbors. Linear regression (Section 3-3-2) is a computationally expensive step. Other options to obtain an estimate of the output are also possible. The easiest option is to simply use the output of the nearest neighbor as an estimate of the output. Also the average output of the K nearest neighbors could be used as an estimate for the output. Our main reason for estimating a linear model is to reduce the influence of noise and thereby improving the estimation accuracy. In Section 3-4-2 we compare LLR with the nearest neighbor and K nearest neighbors approaches.

A more sophisticated memory-based method is Locally Weighted Regression (LWR) [15]. This method can improve the estimation quality compared to LLR by introducing two types of weights. The first is a weighted distance measure which increases the influence of certain state-variables in the distance metric. The second type is a weighted linear regression procedure that increases the influence of nearby samples in the linear

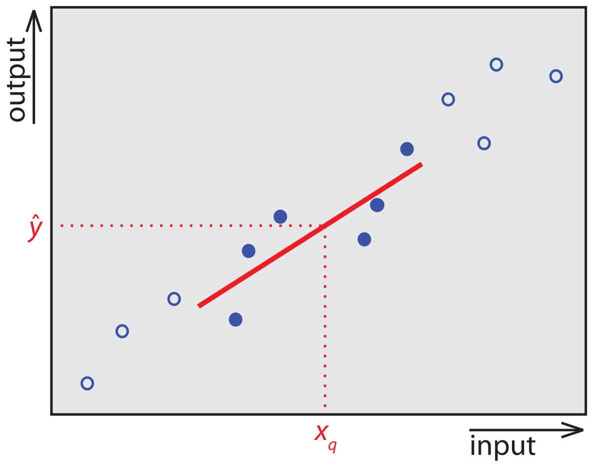


Figure 3-1: LLR on a set of data points. Out of a set of 13 samples (circles), a subset of 6 nearest neighbors (solid circles) is selected that are closest to the query input x_q. The estimated output $\hat{y}$ is found by fitting a linear model through the nearest neighbors and evaluating the obtained model for $x = x_q$.

regression. This method is more complex than LLR and introduces more tuning parameters compared to LLR. The statistical results derived in this section also hold - apart from a weighting factor - for LWR. Although it is not clear how the experimental results would be influenced. However, LWR could be used as a good alternative to LLR if the accuracy of the model would need to be increased.

3-3-1 Nearest neighbors search

As mentioned, the first step in the LLR algorithm is searching the memory for the set of nearest neighbors according to some distance metric D. The number of neighbors to be returned is K. The distance metric can be chosen freely, but is generally assumed to be a mathematical vector norm. We used an unweighted vector norm. Preliminary experiments showed little difference between different norms. In our experiments we have used the 1-norm (or Manhattan norm), which gave slightly better results:

$$D(\mathbf{a}, \mathbf{b}) = \|\mathbf{a} - \mathbf{b}\|_1 = \sum_{i=1}^{k} |a_i - b_i|$$

Where the ith component of the vector $\mathbf{a}$ is denoted by a_i. The dimension of the samples is k. In its simplest implementation, a nearest neighbors search is composed of the following steps:

1. Calculate the distance of the query point to every sample in the memory
2. Sort the samples according to their distance
3. Select the K closest samples

For a memory of size N this means calculating $D(\mathbf{x}_q, \mathbf{x}_j)$ for $j = 1, 2, \ldots, N$, so this implementation has complexity $O(N)$. This approach is most straightforward and is also called the naive approach. For large memory sizes, the naive approach becomes computationally infeasible due to its scaling with N. For small problems (low dimensional and/or small number of samples) or off-line problems (without a limit on the computation time) this approach is generally used. For complex, on-line problems a faster method for the nearest neighbors search is needed. The search speed can be increased by using a kd-tree (short for k-dimensional tree) to represent the memory. In this section we will introduce the kd-tree concept and explain how it is implemented to execute a nearest neighbor search efficiently.

Storing samples in a kd-tree

A kd-tree is a binary search tree that stores k-dimensional data in a tree structure. In this section we will introduce the concept of a tree and explain how it can be built and used for a fast nearest neighbor search.

Tree representation A tree is a structure that represents a set of stored data points. The tree-analogy is used throughout[2] (Figure 3-2): the construction of a tree starts at its *root*, which contains the entire set of data points. Moving up the tree, points are divided into subsets called *branches*. Finally, all the points are represented by *leafs*. In every node, the set of points is split into two subsets, according to some splitting criterion. This criterion is a value in one of the k dimensions. All points that are smaller or equal to the splitting value are diverted to the 'left' part of the three, the remaining points are stored in the 'right' part. For every subset of data, a new splitting criterion is applied and again two subsets of data are created. This process is repeated until a subset contains only one data point. This subset is then called a leaf. With this choice of creating the tree, all points will end up in their own leaf. Alternative implementations allow for multiple points to be stored in a single leaf. Building a kd-tree from a set of data points means selecting and applying the splitting criteria of the nodes. Choosing a splitting criteria will be discussed next.

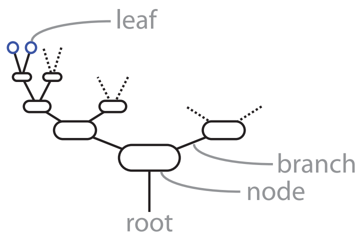


Figure 3-2: Naming in a tree structure. The root contains the total memory, a leaf contains a single sample.

[2]In this thesis the tree is visualized in the intuitive way: placing the root at the bottom and moving up towards the branches and the leafs. This is contrary to the common practice in literature, which is to neglect the tree-analogy and draw the tree 'upside-down'.

Splitting criterion The splitting criterion consists of two parts: the splitting-dimension and the splitting-value. The dimension is chosen based on the current level in the tree. In this way, the splitting-value cycles through all dimensions. The choice of the splitting-value is based on the values of the points (in the splitting-dimension) in the considered subset. We choose the median value. Figure 3-3 shows how a kd-tree is built, based on a data set of 16 samples.

Using the median value to split along a dimension leads to a balanced tree. In a perfectly balanced tree, all nodes have an equal number of 'left' and 'right' branches. A balanced tree is in general preferable, as an unbalanced tree could in theory lead to sub-optimal performance when searching for nearest neighbors. If the distribution of the data is not uniform over the state-space, a different splitting criterion might be preferable in order to prevent bad performance in the nearest neighbor search.

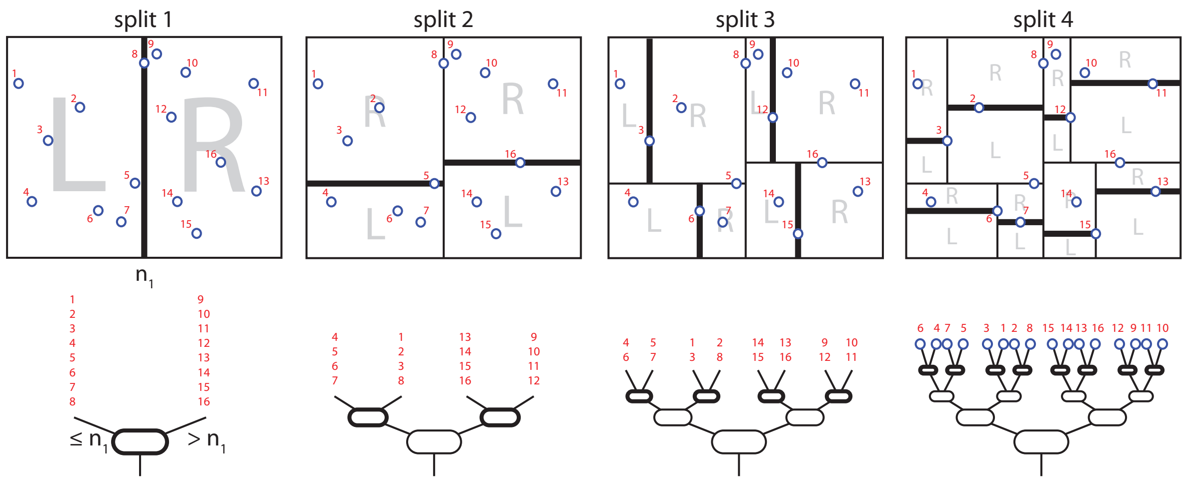


Figure 3-3: Creating a kd-tree structure to represent a set of 16 data samples in a two-dimensional space. 'L' indicates the points 'left' of a node, 'R' indicates the points 'right' of a node.

Adding and removing samples Although building a new tree leads to a perfectly balanced tree, it takes considerable time for a large number of samples. This is caused by the fact that choosing the splitting values, requires sorting of all samples which is a slow process. For a set of N samples of dimensionality k, constructing a tree is of complexity $O(kN \log N)$ [16]. Fortunately, re-building the entire tree is not always needed. Using the existing tree structure, new samples can be added to the tree. Adding samples to an existing tree is straightforward. Starting from the root of the tree, the tree is traversed. At every node, the choice 'left' or 'right' is based on the splitting criterion and the value of the to-be-added sample. In this way, the new sample ends up in a certain leaf. Because we only allow for one sample to be stored in a leaf, this leaf has to be converted into a new node that leads to two leafs: one containing the existing sample and one containing the to-be-added sample (see Figure 3-4).

Continually adding new samples to an existing tree can cause the tree to become unbalanced. Especially when the number of added samples is relatively large compared to the original tree size. An unbalanced tree could lead to performance issues in the nearest neighbors search. A solution to this problem is to re-build the entire tree whenever the tree appears to become unbalanced. The approach in our experiments was to re-build the tree when the number of samples was less than 1000 and add new samples to the existing tree thereafter. This value was chosen because re-building a tree for 1000 samples was still relatively fast. Re-building a tree for more samples took too long to be convenient. With this approach, loss of performance due to an unbalanced kd-tree was never an issue.

Removing samples from an existing tree is simply the reverse of adding a sample. The to-be-deleted sample is removed and the node that led to that sample is converted into a leaf that contains the remaining sample. With the possibility of adding and removing samples, all tools are available to include memory management using a kd-tree to represent the data.

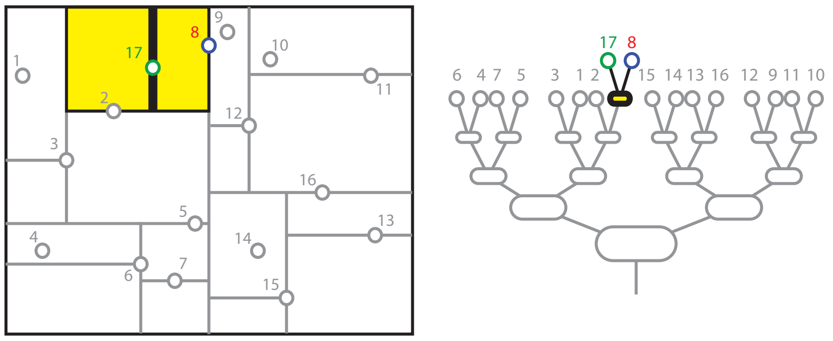


Figure 3-4: Adding a new sample to the existing kd-tree constructed in Figure 3-3. Sample 17 is added by creating a new node at the position of sample 8.

Nearest neighbors search using a kd-tree

Our reason to store all data in a kd-tree is to speed up the nearest neighbors search. We will describe the search algorithm for finding a single nearest nearest neighbor, according to [17]. For this method it is assumed that the distance measure D is monotonically increasing, which is true for all vector norms:

$$D_i(x_{q,i}, y_i) \geq D_i(x_{q,i}, z_i), \quad \text{if } y_i > z_i \tag{3-1}$$

With $i = 1, 2, ..., k$. D_i is the distance measure for the ith dimension.

We will describe the search for the nearest neighbor to a query point in a 2-dimensional space (Figure 3-5). The search starts with moving down the tree to determine the leaf that would contain the query point. This is the same procedure as when a new sample was added to an existing tree. The point in the leaf is stored as 'current best'. The real nearest neighbor should be at least as close to the query point as this first leaf point. Therefore, a sphere with a radius equal to the distance to the current best is created around the query point. Using the values of the splitting criteria, the nodes that intersect with this sphere are identified. These nodes could possibly contain points that are closer than the current best and are therefore inspected. If a closer point is found, this is stored as current best. With this approach, large parts of the tree do not have to be visited. This approach is also valid in higher dimensions. For a more elaborate description of the search algorithm, see e.g. [18], [17].

A K nearest neighbors search is carried out in the same way. The only difference is that a list of K current best points is stored, instead of a single best point. A new point is added to this list when it is closer that the K nearest point.

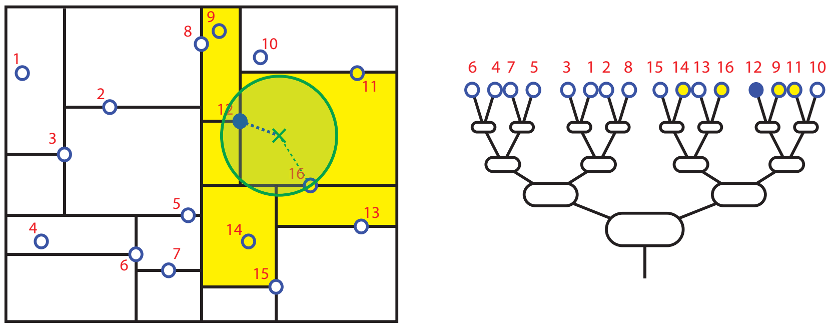


Figure 3-5: Nearest neighbor search using a kd-tree structure. The green cross represents the query point. The first neighbor found is sample 16. All nodes that intersect with the green circle are then visited and eventually sample 12 is identified as the nearest neighbor. Using the kd-tree, the distance to the query point of only five samples (9, 11, 12, 14, 16) out of a total of sixteen samples had to be calculated.

Performance The reason for using a kd-tree to store memory samples is to reduce the computational effort in the nearest neighbors search. Theoretically, the nearest neighbor search using a tree has complexity $O(\log(N))$ [17], excluding the time needed to build the tree. So for large data sets, this method is preferred over a naive search, which has complexity $O(N)$.

The nearest neighbors search discussed here, always returns the exact K closest neighbors. A way to reduce computation time even more, is to use an approximate search algorithm. Such an algorithm also returns K neighbors, but not necessarily the closest. For on-line applications with a large number of memory samples a fast search might be more important than an exact search. The approximate nearest neighbors search might be useful to reduce computation time in such cases. However, approximate nearest neighbor methods are outside the scope of this research.

Limitations

Although using a tree structure to find the set of nearest neighbors significantly increases the searching speed, some limitations do exist. We will discuss two problems.

Wrapping angles Many systems with rotational degrees of freedom, have rotational symmetry. Adding a full 2π rotation to a variable does not change the physical state of the system. Therefore, those state-variables are usually wrapped on a 2π domain

($[0, 2\pi)$ for instance). The nearest neighbors search should be able to deal with this wrapped domain when calculating distances. For instance, the points $x_1 = 0$ and $x_2 = 2\pi$ should have a distance of 0, instead of 2π.

In the naive approach dealing with wrapping can be done by, for instance, translating all data points such that the query point is in the center of the wrapped domain or by introducing an alternative distance measure that incorporates wrapping. These approaches are not possible when a tree structure is used. Manipulating data points would require re-building of the entire tree, which is unwanted for large data sets. An alternative distance measure would violate the monotonicity requirement Eq. (3-1), which would cause the nearest neighbors search to fail. To our knowledge an implementation of kd-trees that deals with wrapped domains does not exist. Development of such a 'circular' kd-tree could be an interesting and useful future research topic.

In this work we will simply ignore the wrapping problem. For data sets with a sufficiently high sample density near the edges of the wrapped domain, this should not lead to problems.

Building time The reason we are using a kd-tree, is for speeding-up the K nearest neighbors search. As we have shown, this process is significantly faster using a tree than using a sorting algorithm. However, the time needed for building the tree is still large. This is due to the fact that selecting the median values in a certain dimension to use as splitting values still involves sorting all data points. If one would continuously re-build the entire tree, this would be a major issue. In our experiments, we have chosen the following approach to deal with this problem: We build an initial tree from a large number of data point and add new points to the existing tree. The possible loss of performance was not observed, possibly because the initial tree was built using a sufficiently large number of data points.

3-3-2 Linear regression

Once the set of K nearest neighbors is determined, the second step of the LLR method is fitting a linear model to the obtained set of data. Linear regression is a well known technique and has been thoroughly analyzed in the literature (see for example [19]). In this section, we will select useful results from the literature that will be used in this work. First, we will introduce the linear regression model. Thereafter we will describe prediction intervals and outlier detection as possible techniques to assess and increase the quality of the estimates.

Estimating a linear model

We assume that the state-transition model can be represented locally by a linear model with the following structure:

$$\begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_{d_y} \end{pmatrix} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_{d_x} \end{pmatrix}^T \begin{pmatrix} \beta_{1,1} & \beta_{1,2} & \dots & \beta_{1,d_y} \\ \beta_{2,1} & \beta_{2,2} & \dots & \beta_{2,d_y} \\ \vdots & \vdots & \ddots & \vdots \\ \beta_{d_x,1} & \beta_{d_x,2} & \dots & \beta_{d_x,d_y} \end{pmatrix} + \begin{pmatrix} \epsilon_1 \\ \epsilon_2 \\ \vdots \\ \epsilon_{d_y} \end{pmatrix}$$

Or in vector notation:

$$\mathbf{y} = \mathbf{x}^T\boldsymbol{\beta} + \boldsymbol{\epsilon} \tag{3-2}$$

Where $\mathbf{y}$ is a $d_y \times 1$ output vector, $\mathbf{x}$ is a $d_x \times 1$ input vector[3], $\boldsymbol{\beta}$ is a $d_x \times d_y$ matrix. The regression variable $\boldsymbol{\beta}$ represents the local state-transition model and consists of $d_x d_y$ regression coefficients. The $d_y \times 1$ vector $\boldsymbol{\epsilon}$ is the noise vector. By definition, the error vector has the following properties:

1. $E[\boldsymbol{\epsilon}] = \mathbf{0}$, hence $E[\mathbf{y}] = E\left[\mathbf{x}^T\boldsymbol{\beta}\right]$
2. $\text{cov}(\boldsymbol{\epsilon}) = \sigma^2\boldsymbol{I}$, hence $\text{cov}(\mathbf{y}) = \sigma^2\boldsymbol{I}$

Where $E[\cdot]$ denotes the expected value. The first property assumes that the error is zero mean and that the samples represent a linear function. The second property assumes that the individual noise terms are independent and their variance equals σ^2.

The linear model is estimated based on K observations (which result from the nearest neighbors search). These can be written in matrix form as:

$$\begin{pmatrix} \mathbf{y}_1^T \\ \mathbf{y}_2^T \\ \vdots \\ \mathbf{y}_K^T \end{pmatrix} = \begin{pmatrix} \mathbf{x}_1^T \\ \mathbf{x}_2^T \\ \vdots \\ \mathbf{x}_K^T \end{pmatrix} \boldsymbol{\beta} + \begin{pmatrix} \boldsymbol{\epsilon}_1^T \\ \boldsymbol{\epsilon}_2^T \\ \vdots \\ \boldsymbol{\epsilon}_K^T \end{pmatrix}$$

Or in short:

$$\mathbf{Y} = \mathbf{X}\boldsymbol{\beta} + \boldsymbol{E}$$

With $\mathbf{X}$ the input matrix and $\mathbf{Y}$ the output matrix. The regression variable $\boldsymbol{\beta}$ has $d_y d_x$ regression coefficients. In order to uniquely identify $\boldsymbol{\beta}$ from K samples, it is required that $\text{rank}(X) \geq K$. This means that we require at least $K \geq d_x$ observations. The LLR model is estimated using the least squares approach. We try to minimize the sum of the squared errors:

$$\sum_{i=1}^{K} \hat{\mathbf{e}}_i^2 = \sum_{i=1}^{K} (\mathbf{y}_i - \hat{\mathbf{y}}_i)^T (\mathbf{y}_i - \hat{\mathbf{y}}_i)$$

[3]We will use $\mathbf{x}$ and $\mathbf{y}$ to represent input and output in this section. Note that in a real application $\mathbf{x}$ would contain states x_t and actions u_t and $\mathbf{y}$ would contain next states x_{t+1}, as explained in 3-2

Where $\hat{\mathbf{y}}_i = \mathbf{x}_i^T \hat{\boldsymbol{\beta}}$ is the prediction of $\mathbf{y}_i$. We use $\hat{\cdot}$ to denote a predicted quantity. In matrix notation, the sum of squared errors is:

$$\hat{\boldsymbol{E}}^T \hat{\boldsymbol{E}} = \left(\boldsymbol{Y} - \boldsymbol{X}\hat{\boldsymbol{\beta}}\right)^T \left(\boldsymbol{Y} - \boldsymbol{X}\hat{\boldsymbol{\beta}}\right)$$

Differentiating the squared error with respect to $\boldsymbol{\beta}$ and setting the result equal to zero, leads to the normal equations:

$$\boldsymbol{X}^T \boldsymbol{X} \hat{\boldsymbol{\beta}} = \boldsymbol{X}^T \mathbf{Y}$$

Therefore, the least squares solution to Eq. (3-3-2) is:

$$\hat{\boldsymbol{\beta}} = \left(\boldsymbol{X}^T \boldsymbol{X}\right)^{-1} \boldsymbol{X}^T \boldsymbol{Y}$$

Note that in practice, calculating $\left(\boldsymbol{X}^T \boldsymbol{X}\right)^{-1}$ explicitly is wanted nor needed. Instead, the normal equations are solved using matrix manipulations (such as Gaussian elimination).

Model accuracy

Estimation error The obtained linear model is used to estimate a state-transition. Or, using input-output terminology, estimate the output for a given input query:

$$\hat{\mathbf{y}}_q = \mathbf{x}_q \hat{\boldsymbol{\beta}}$$

We would like to have a measure to assess the accuracy of the predicted output. In the field of system identification, the quality of a model is usually assessed by comparing the estimated (model) output to the measured (real) output. For a single prediction, we can simply use the residual vector $\mathbf{e}$:

$$\mathbf{e} = |\hat{\mathbf{y}}_q - \mathbf{y}_q|$$

If we want to assess the quality of a set of estimates, we can use the Root Mean Squared Error (RMSE) of a set of M predictions:

$$\text{RMSE} = \sqrt{\frac{1}{M} \sum_{t=1}^{M} \left(\hat{\mathbf{y}}(t) - \mathbf{y}(t)\right)^T \left(\hat{\mathbf{y}}(t) - \mathbf{y}(t)\right)}$$

A measure comparing the estimated output to the measured output, will be referred to as the Root Mean Squared (RMS) *estimation error*.

However, such methods can only be used if the real output is available. In our case, we want to apply the LLR model in a RL setting. We will not always have the real output available to compare the estimated output to. So we would like to have an accuracy measure that is based on the data used in the linear regression and not on a measured output. Such a measure will be called the *prediction accuracy*. The expressions for the prediction accuracy will be discussed next.

Prediction accuracy The accuracy of a predicted output will be given as a confidence interval around the estimated output. It can be shown (see Appendix A-2) that the t-statistic:

$$t = \frac{\mathbf{y}_q - \hat{\mathbf{y}}_q}{\sigma\sqrt{1 + \mathbf{x}_q^T \left(\boldsymbol{X}^T \mathbf{X}\right)^{-1} \mathbf{x}_q}}$$

is zero mean and has a $t(K - d_x)$ distribution. The noise variance σ^2 can be estimated by s^2 (Eq. (A-2)). Using the probability density function of the t-distribution, we can obtain a $100(1 - \alpha)\%$ confidence interval for the estimate $\hat{\mathbf{y}}_q$:

$$\mathbf{y}_q = \hat{\mathbf{y}}_q \pm t_{\alpha/2, K-d_x} s\sqrt{1 + \mathbf{x}_q^T \left(\boldsymbol{X}^T \boldsymbol{X}\right)^{-1} \mathbf{x}_q}$$

These formulas can be used as a quality measure of the estimated outputs, based solely on information that can be obtained from the dataset. This measure will be called a prediction interval I and we can write:

$$\mathbf{y}_q = \hat{\mathbf{y}}_q \pm I$$

Throughout this report we have used $\alpha = 0.05$, which gives a 95% confidence interval around a predicted output.

It is important to note that these equations are statistical measures that are derived, based on assumptions on the data-generating function. The most important assumption is that the generating function is linear. In our experiments, we assume that the functions we are estimating are locally linear and that the nearest neighbors represent this local linearity. Therefore we argue that the statistical results are valid. However, the nearest neighbors might not represent a linear function when the target function is highly nonlinear or the sample density is low. In these cases the samples used in the linear regression do not represent a linear system Eq. (3-3-2) and the equations are not valid. In fact, the prediction interval may severely underestimate the possible prediction error in the output. Therefore, this measure has to be used with care. The same holds for the equations derived for outlier detection in the next section.

Outlier detection

A set of data, obtained from measurements on a real system, typically contains a certain amount of faulty data. These faulty samples can be the result of disturbances or measurement errors. LLR is a method that is sensible to noise because no global structure is imposed on the data. Since faulty samples could lead to an inaccurate model, it is useful to be able to identify these samples and optionally ignore them in the estimation process. Samples that are far away from the linear model that is fitted to the data are called outliers.

We again use a measure from the field of statistics to determine the presence of outliers. We use the PREdiction Sum of Squares (PRESS) statistic:

$$\text{PRESS} = \sum_{i=1}^{K} \left(\frac{\mathbf{y}_i - \mathbf{x}_i^T \boldsymbol{\beta}}{s\sqrt{1 - \mathbf{x}_i^T \left(\boldsymbol{X}^T \boldsymbol{X}\right)^{-1} \mathbf{x}_i}} \right)^2$$

The individual contribution of each data point to PRESS is:

$$e_i = \frac{\mathbf{y}_i - \mathbf{x}_i^T \boldsymbol{\beta}}{s\sqrt{1 - \mathbf{x}_i^T \left(\boldsymbol{X}^T \boldsymbol{X}\right)^{-1} \mathbf{x}_i}}$$

This measure has a normal distribution with zero mean and unit variance. If the residual of a point is larger than a certain threshold, the point is called an outlier. We use 1.96, which is the value that cuts off the 5% area of a normal distribution.

3-3-3 Computational complexity

An advantage of memory-based modeling, is that at every moment the full dataset is available to perform various analysis and optimization calculations. In this section we have only discussed prediction intervals and outlier detection but many more useful calculations could be thought of. In simulation, the addition of extra calculations is virtually unlimited. In a real-time system however, the amount of time available to make calculations is limited and is dictated by the sampling time. So, all calculations must be performed in the given sampling interval, which means that the number of calculations that can be performed is limited.

In this work, LLR is used as a model building technique in a RL setting. We are using LLR as a way to increase learning speed by generating state-transitions. We are not necessarily interested in estimating a very accurate model. In Chapter 4 we will show an example in which it is more useful to estimate many inaccurate state-transitions than to estimate less very accurate state-transitions. So whether or not the statistical methods are useful in practice might depend on the specific application.

3-4 Experimental results

In Section 3-3 we introduced LLR as a modeling method and introduced statistics that can be used to assess and improve the estimation accuracy of the model. In this section we will apply LLR in three settings. First we will estimate a 1-dimensional nonlinear function. Using this example, we can easily investigate and visualize the results of the LLR method. Thereafter we will use LLR as a method to model state-transitions on two different setups. The first setup is a simulation of a two-link manipulator. This system has got four state-variables and two inputs. The second setup is a humanoid robot setup. This is a complex setup that has 18 state-variables and 6 inputs.

3-4-1 1D nonlinear function

Before using LLR to model state-transitions of a real system, we will investigate how the results of Section 3-3-2 can be used. To easily visualize the results of this section, we use the following 1-dimensional, nonlinear test function:

$$y = x - \sin^3(2\pi x^3)\cos(2\pi x^3)\exp(x^4) \tag{3-3}$$

on the domain $x \in [0, 1]$. This nonlinear test function is useful, because it contains both high-frequency and low-frequency components. We estimate a model $\hat{\boldsymbol{\beta}}$ that estimates the output $\hat{y}$ using K inputs samples around x_q:

$$\hat{y} = \begin{bmatrix} x_q \\ 1 \end{bmatrix}^T \hat{\boldsymbol{\beta}}$$

We add 1 to the input vector to prevent that the linear model has to cross the origin. A dataset of $N = 100$ randomly distributed samples was generated and a white noise signal with $\sigma^2 = 0.01$ was added to the output (see Figure 3-6(a)). Using this nonlinear function we would like to answer the following questions regarding the LLR method:

1. Can the prediction interval be used to assess the quality of the estimate?
2. Is outlier detection useful to increase the estimation accuracy?
3. Can we locally tune the number of nearest neighbors to increase the estimation quality?

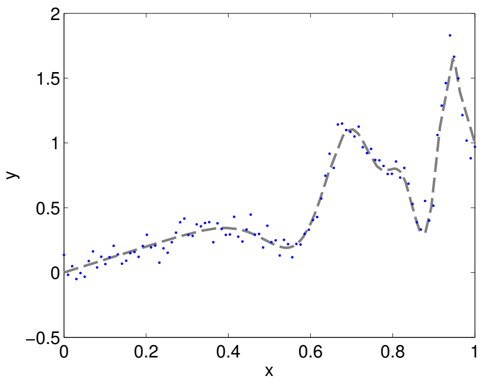
Prediction intervals

Figure 3-6(b) shows the LLR estimate (using $K = 4$) of the nonlinear test function with prediction intervals that indicate the quality of the estimate. The target function was estimated at $x_q = 1, 0.01, ..., 1$. The estimation is reasonably good in low-frequency areas (small values of x) but in high frequency areas (large values of x), the estimate is less accurate. The reason is that the sample density is too low to represent the fast dynamics of the target function accurately. This leads to a set of K samples that do not represent a linear function around x_q. This is represented by the prediction intervals that are large for these areas, but much smaller for the more accurate estimates.

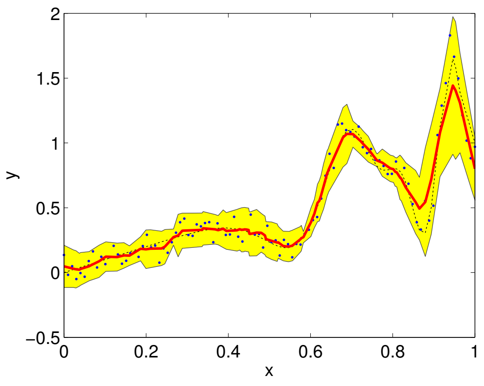
Figure 3-6(c) shows the size of the prediction interval compared to the estimation error $|\hat{y} - y|$. We can clearly see that the prediction interval is no actual accuracy measure: the shape of the interval does not exactly match the shape of the error. However, the prediction interval is significantly larger for large values of x and smaller for small values of x. This effect follows from the definition of the prediction interval. It should be used as a measure of linearity of samples instead of accuracy of prediction. However, if the samples are highly nonlinear, we can expect the estimated output to be inaccurate (or at least uncertain). So we argue that this measure can still be used to assess the quality of the prediction.

Outlier detection

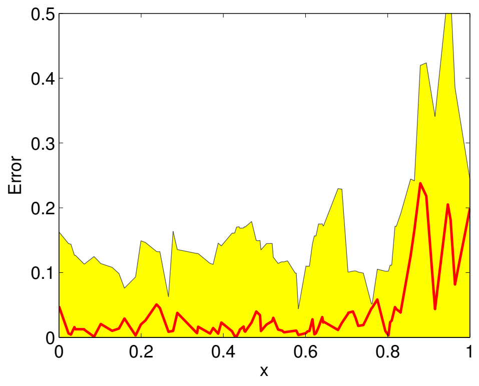
To test if we can use the methods of Section 3-3-2 to identify faulty data, we replaced four samples of the original dataset of Figure 3-6(a) by faulty data. Figure 3-7(a) shows the result that the outliers have on the estimation of the target function. We notice that the prediction is severely effected by the presence of the outliers. This is in turn



(a) Dataset obtained from the nonlinear function



(b) LLR estimate of the nonlinear function



(c) Prediction interval compared to the estimation error

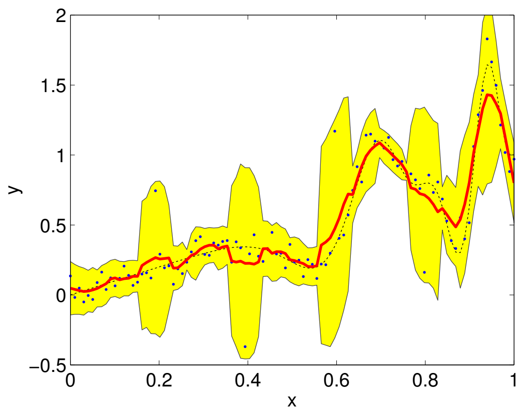
Figure 3-6: LLR estimate of a 1-dimensional nonlinear function. (a) shows 100 noisy samples (dots) obtained from the nonlinear function Eq. (3-3) (dashed line). (b) shows the resulting LLR estimate (solid line) of the nonlinear function (dashed line) using $K = 4$ with prediction intervals (shaded area) estimated at 100 query points. (c) shows the estimation error $|\hat{y} - y|$ (solid red line) compared to the size of the prediction interval (shaded area).

represented by the prediction intervals, which are very large for the estimates that use the outliers.

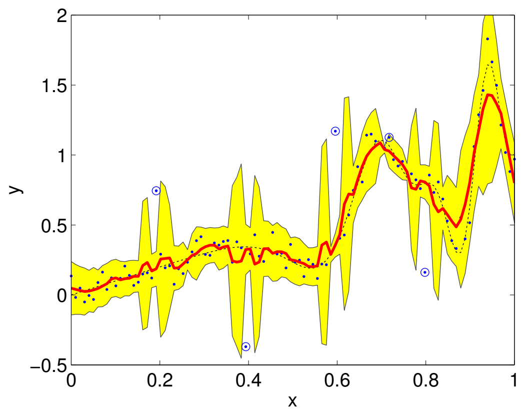
Figure 3-7(b) shows the result when outlier detection is used. The data points that are identified as outliers are indicated by circles. When outliers were detected the LLR estimate was computed again, neglecting the outlier(s). The added outliers are identified correctly. The result is clearly visible by the prediction intervals that are much smaller when the outliers are neglected. Also the LLR estimate $\hat{y}$ becomes better when the outliers are neglected.

We notice that not only the added outliers are detected, but also the sample at $x = 0.72$ is identified as outlier. This is caused by the fact that the noisy data samples around this point are almost linear. This causes the identified linear model to match all points very accurately except for one point, which is therefore identified as outlier.

In our approach, we simply neglected outliers that were detected. An other approach could be to remove detected outliers from the memory. We do not opt for this approach because a sample that appears to be an outlier for a certain query point at one moment in time, could prove to be an usable sample for a different query point or at a later stage when more samples have been added to the memory. In general, we do not encourage removing or changing memory data.



(a) Without outlier detection



(b) With outlier detection

Figure 3-7: LLR estimate with $K = 4$ (solid line) of a nonlinear function (dashed line) with outliers added at $x = 0.2, 0.4, 0.6, 0.8$. (a) shows the resulting estimate when the outliers are used in the estimation. (b) shows the result when outliers are detected and neglected in the estimation. Data points that are identified as outliers are indicated by circles.

Optimizing the number of neighbors

One of the attractive properties of LLR is that the method has only got one parameter to tune: the number of nearest neighbors K used in estimating the linear model. The optimal number of neighbors leads to the best estimation of the output (i.e., the smallest estimation error). The optimal number can vary throughout the state-space and depends locally on a number of factors:

- Linearity of the function (highly nonlinear, smaller K)
- Sample density (more samples, higher K)
- Amount of noise (more noise, higher K)

Because these factors can vary over the state-space, the optimal number of nearest neighbors can also vary over the state-space. Figure 3-8 schematically shows the estimation error as a function of K for three different target functions:

1. Linear function with noise
2. Nonlinear function without noise
3. Nonlinear function with noise

In the first case (Figure 3-8(a)), the number of neighbors should be high as possible to cancel the effect of noise. In fact, for white noise and $K \to \infty$ the LLR estimate $\hat{y}(x_q)$ is an unbiased estimator of $y(x_q)$.

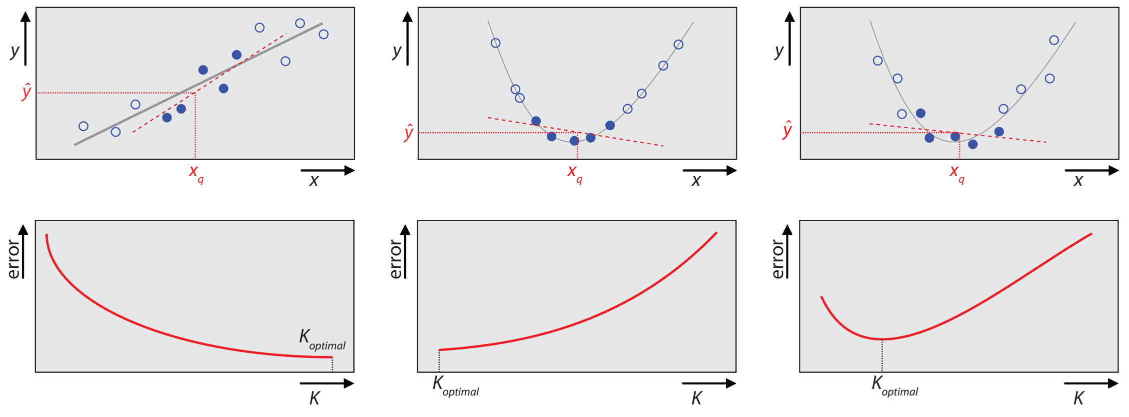
In the second case (Figure 3-8(b)), the optimal choice is to simply take the output of the nearest neighbor as the estimated output. Including more neighbors will introduce a bias, since the target function is locally nonlinear.

The third case is the most interesting since this situation will generally be encountered in practice. Starting from $K = 1$, increasing the number of nearest neighbors will reduce the estimation error, because the effect of noise is reduced. Further increasing K, will increase the error as the nonlinearity results in a biased estimate. So an optimal value for K exists for which the estimation error is smallest.

Determining the optimal value for K can be problematic, since no analytical expression to determine the optimal value for K exists. Furthermore, even if we have a measure for the estimation error for a certain value of K, we do not have a method to determine whether this error is due to noisy samples or due to local nonlinearity of the target function. In the first case K has to be increased, while in the second case it has to be decreased. In the next paragraph we will describe how the prediction interval could be used to tune the number of nearest neighbors.

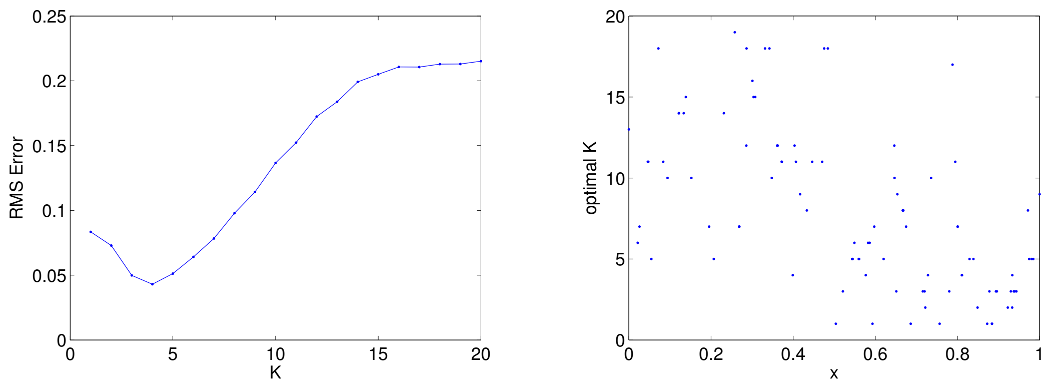
Globally optimal K The general approach to tuning the value of K is using the globally optimal value. A global optimum means a single fixed K value that leads to the best estimation error on average. In practice, this means computing a model for several values of K for a set of query points. The value that leads to the smallest total residual value will be chosen as globally optimal. This approach will typically lead to a model that is too general in fast dynamics regions and too noise sensitive in slow dynamics regions.

Figure 3-9(a) shows the RMS error of the estimation of the dataset of Figure 3-6(a) for a range of different K values. Since the target function is noisy and nonlinear, we expect the behavior as depicted in Figure 3-8(c). The expected behavior (first a decrease in error, then an increase) is indeed visible. Based on this figure, we select the globally optimal value $K = 4$.



(a) Linear target function with noise (b) Nonlinear target function without noise (c) Nonlinear target function with noise

Figure 3-8: Optimal number of neighbors for three different cases. (a) linear target function with added noise, (b) nonlinear target function without noise and (c) nonlinear target function with added noise. The top figures show a typical distribution of data points and the resulting linear regression. The bottom figures show schematically the absolute value of the estimation error of the real output and the estimated output as a function of the number of neighbors K.



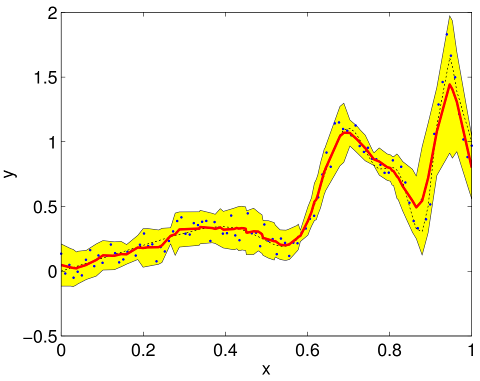
(a) Global optimum: $K = 4$ (b) Locally optimal K

Figure 3-9: Comparison between global and local optimal K values for the nonlinear target function Eq. (3-3). (a) shows the RMS error of the estimation for a range of different K values. (b) shows the locally optimal value of K for a range of x values.

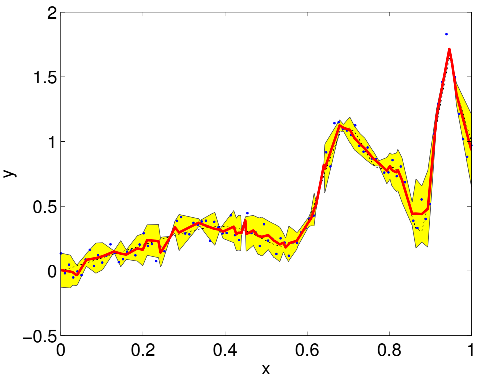
Locally optimal K A more advanced method of selecting K is choosing a specific value for every query point. Using different values for every query point could lead to a better estimation quality on average. Unfortunately, an analytical formula for finding the optimal value does not exist, since the effect of an single unknown sample on the estimation quality is not known[4]. Therefore, the locally optimal value has to be found by computing the model for a set of K values and selecting the one that is best according to some quality measure. Notice that this quality measure may only depend on the memory samples and not on the actual output, since this may be unknown. A possible quality measure could be the prediction interval discussed in Section 3-3-2. We select the K value that leads to the smallest prediction interval as optimal.

Figure 3-9(b) shows the locally optimal value for K for the nonlinear test function Eq. (3-3). It is clear that the optimal value varies heavily over the domain of the function. On closer inspection, we notice that the optimal value is in general smaller for larger x-values. This is due to the faster dynamics of the test function for these values of x. For smaller x-values the optimal value is larger due to the slower dynamics.

The difference between a locally and globally optimal value of K is shown in Figure 3-10. The global K value is set to $K = 4$. We notice that the prediction interval is smaller using a locally optimal K. This is as expected, since this is the quantity we minimized in selecting K. However, if we look at the estimated values, the results vary. In high frequency regions (near $x = 0.85$ and $x = 0.95$) we notice that the estimate improves when a locally optimized K value is used. In low frequency regions (near $x = 0.05$ and $x = 0.45$ for example), the estimation becomes worse. This is due to the fact that a linear model is fitted to a small number of noisy samples.



(a) Globally fixed $K = 4$



(b) Locally optimal K

Figure 3-10: LLR estimate (solid red line) of a nonlinear function (dashed line) with prediction intervals (shaded area). (a) shows the resulting estimate for a globally optimal (fixed) value of K. (b) shows the result for a locally optimal (varying) value of K.

Whether or not a locally optimized K-value leads to better estimations than a globally optimized value will depend on the situation. In very noisy applications, using the

[4]Incremental linear regression methods that solve a least squares problem by adding samples one-by-one do exist. However, they do not give an analytical expression for the influence of a single sample on the output or the prediction accuracy.

prediction interval to determine a locally optimal value might lead to bad estimates. A globally optimal K value is probably a good choice is most situations. In the remaining experiments, we selected a globally optimal K value in a way very similar to the approach described in this section.

3-4-2 Two-link manipulator

In this section we will use LLR to model state-transitions of a realistic setup. We consider a two-link manipulator system (see Figure 3-11). It was chosen because of its relative simplicity and because of its connection with a humanoid robot setup. The manipulator resembles one leg of a humanoid robot, actuated at the knee and hip joints. If this setup leads to good results, it gives confidence that LLR might also be used to model a real humanoid robot setup.

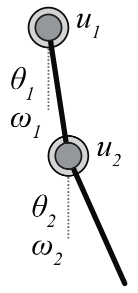


Figure 3-11: Two-link manipulator setup.

The two-link manipulator consists of two actuated joints, connected with rigid arms. The system is fixed at one of the joints. The setup is placed vertically. The system has a 4 state-variables: the two angles $\boldsymbol{\theta} = \begin{bmatrix}\theta_1 & \theta_2\end{bmatrix}^T$ and two angular velocities $\boldsymbol{\omega} = \begin{bmatrix}\omega_1 & \omega_2\end{bmatrix}^T$ of both joints. The full state-vector is $\mathbf{x} = \begin{bmatrix}\boldsymbol{\theta} & \boldsymbol{\omega}\end{bmatrix}^T$. The input of the system are the output voltages to the motors: $\mathbf{u}= \begin{bmatrix}u_1 & u_2\end{bmatrix}^T$. The voltages are limited to $\begin{bmatrix}1.5 & 1\end{bmatrix}^T$. The angles are limited to $[-\pi \quad \pi]$. The system is controlled at discrete time intervals t, with sampling time $T_s = 0.05$ s. The equations of motion are nonlinear and are given by:

$$M(\boldsymbol{\theta})\dot{\boldsymbol{\omega}} + C(\boldsymbol{\theta}, \boldsymbol{\omega})\boldsymbol{\omega} + G(\boldsymbol{\theta}) = \mathbf{u}$$

$$\begin{aligned}
M(\boldsymbol{\theta}) &= \begin{bmatrix} P_1 + P_2 + 2P_3\cos\theta_2 & P_2 + P_3\cos\theta_2 \\ P_2 + P_3\cos\theta_2 & P_2 \end{bmatrix} \\
C(\boldsymbol{\theta}, \boldsymbol{\omega}) &= \begin{bmatrix} b_1 - P_3\omega_2\sin\theta_2 & P_3(\omega_1+\omega_2)\sin\theta_2 \\ P_3\omega_2\sin\theta_2 & b_2 \end{bmatrix} \\
G(\boldsymbol{\theta}) &= \begin{bmatrix} -g_1\sin\theta_1 - g_2\sin(\theta_1+\theta_2) \\ -g_2\sin(\theta_1+\theta_2) \end{bmatrix}
\end{aligned}$$

The abbreviations are defined as follows:

$$\begin{aligned} P_1 = &\quad m_1 c_1^2 + m_2 l_1^2 + I_1 \\ P_2 = &\quad m_2 c_2^2 + I_2 \\ P_3 = &\quad m_2 l_1 c_2 \\ g_1 = &\quad (m_1 c_1 + m_2 l_1) g \\ g_2 = &\quad m_2 c_2 g \end{aligned}$$

The parameters of the system and their physical meanings can be found in Table 3-1.

Table 3-1: Physical parameters of the two-link manipulator

Symbol	Parameter
$g = 9.81$ m/s^2	gravitational acceleration
$l_1 = 0.4$ m	length of first link
$l_2 = 0.4$ m	length of second link
$m_1 = 1.25$ kg	mass of first link
$m_2 = 0.8$ kg	mass of second link
$I_1 = 0.07$ kgm^2	inertia of first link
$I_2 = 0.04$ kgm^2	inertia of second link
$c_1 = 0.2$ m	center of mass of first link
$c_2 = 0.2$ m	center of mass of second link
$b_1 = 0.08$ kg/s	damping in first joint
$b_2 = 0.02$ kg/s	damping in second joint

The model was implemented as a simulation in Matlab. A white noise signal with $\sigma^2 = 0.01$ was added to each output. The following input-output relation was estimated:

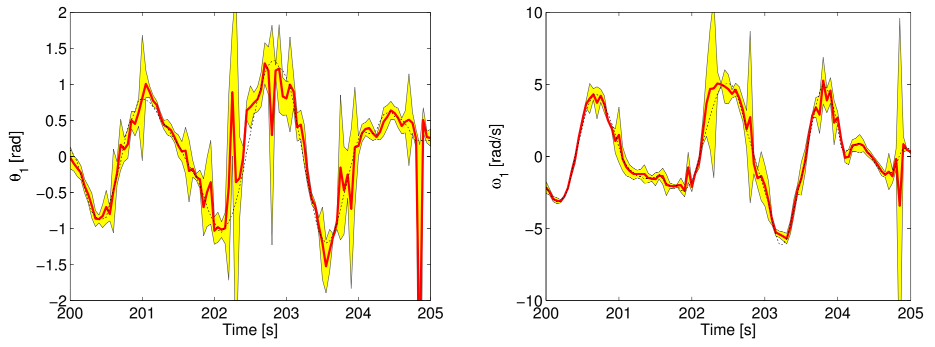
$$\hat{\mathbf{x}}_{t+1} = \begin{bmatrix} \mathbf{x}_t \\ \mathbf{u}_t \\ 1 \end{bmatrix}^T \hat{\boldsymbol{\beta}}$$

with $\hat{\boldsymbol{\beta}}$ the local linear model. A dataset was generated by applying a random input signal to the system and observing the resulting state-transitions for 250 seconds (5000 samples). The memory was filled with state-transition samples: $(\mathbf{x}_t, \mathbf{u}_t, \mathbf{x}_{t+1})$. The number of nearest neighbors used in the experiments was fixed to $K = 10$.

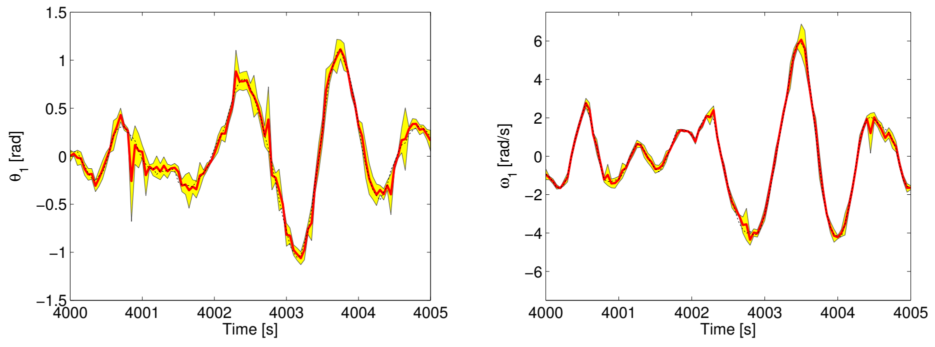
The two-link manipulator setup was used to investigate some of the properties of the LLR method in a setting that resembles a real learning experiment. We are mainly interested in the ability of LLR to model state-transitions and the influence of the memory size on the estimation accuracy. Furthermore, we are interested if fitting a linear model to a set of nearest neighbors improves the estimation compared to more simpler memory-based methods. We can summarize these questions as follows:

1. Can LLR be used to model state-transitions?
2. How does the memory size influence the estimation accuracy?
3. Is LLR an improvement over other memory-based methods?

Estimating state-transitions



(a) Memory size: $N = 200$ (10 s)



(b) Memory size: $N = 4000$ (200 s)

Figure 3-12: LLR estimate of state-transitions for the two-link manipulator for increasing memory size using $K = 10$. The figures show the improvement of the LLR estimate for θ_1 (left figures) and ω_1 (right figures). (a) is the estimate using a memory of size $N = 200$ (10 seconds), (b) the estimate for $N = 4000$ (200 seconds). The figures show the LLR estimate (red solid line), the measured value (black dashed line) and the prediction intervals (shades areas).

In a typical real-world learning experiment, the size of the memory available to the model increases during the experiment. We mimic this situation by increasing the size of the memory available to the LLR algorithm. At every time step, the model uses the memory samples observed up until that moment. These memory samples are used to estimate the transition from the current state to the next. We expect that the estimations become better with increasing memory size. Figure 3-12 shows the estimated state-transitions at two moments in the experiment for the angle θ_1 and angular velocity ω_1 of the first link. The state-variables of the second link show the same behavior and are therefore not shown.

The first observation that has to be made, is the fact that LLR is indeed able to model state-transitions. Using state-action pairs as input, the resulting next state can be estimated as output. Especially with a large memory (Figure 3-12(b)), the estimated state-transitions closely match the real state-transitions. It appears that the prediction intervals can be used as a measure for the estimation accuracy. We notice that state-transitions that are estimated inaccurately, also have a relatively large prediction interval. Especially with a large memory, the prediction intervals are small compared to the amplitude of the variables.

The improvement of the estimate with increasing memory size, is also visualized in Figure 3-13. The figure shows the RMS error of the estimated state-transitions. The estimation improves with the first 1000 samples and is more or less equal thereafter.

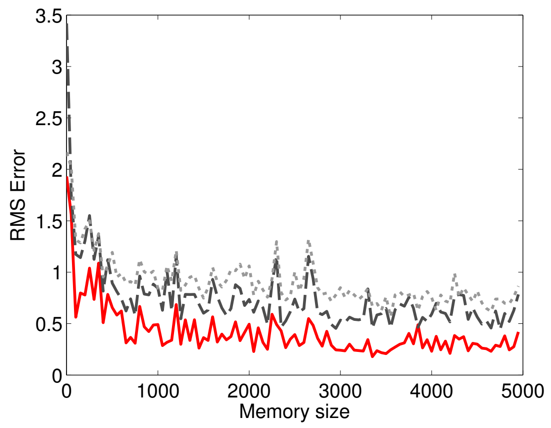


Figure 3-13: RMS estimation error for increasing memory size using different methods to estimate the two-link manipulator. The figure compares the nearest neighbor estimate ($K = 1$, dotted gray line), the K nearest neighbors average ($K = 10$, dashed gray line) and the LLR estimate ($K = 10$, solid red line).

Comparing memory-based modeling methods

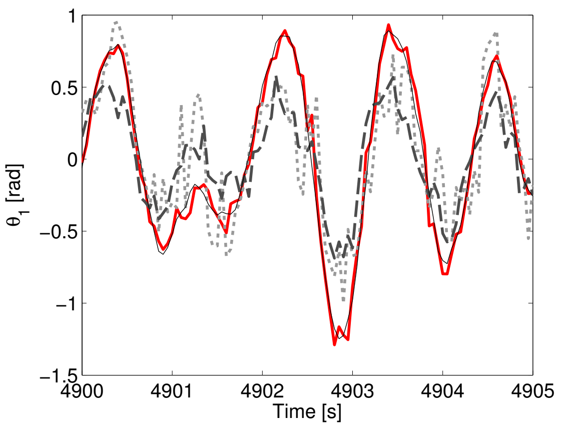


Figure 3-14: Comparison of three memory-based modeling methods on the two-link manipulator using a large memory ($N = 5000$). The figure shows the estimation of a trajectory (solid black line) using the nearest neighbor ($K = 1$, dotted gray line), the K nearest neighbors average ($K = 10$, dashed gray line), and the LLR estimate ($K = 10$, solid red line).

We have introduced LLR as a memory-based modeling method that fits a linear model to the set of nearest neighbors around the query input. Linear regression is a time consuming process, so the question rises whether fitting a linear model to the set of nearest neighbors is needed. Instead of estimating a model, we could also use the average output of the nearest neighbors as estimate (the K nearest neighbors average). An even faster option is to simply use the output of the nearest neighbor ($K = 1$) as an estimation for the output.

Figure 3-14 compares these three memory-based methods by estimating a 5-second trajectory using a random input signal and a large memory ($N = 5000$) to estimate state-transitions. We see that over the entire trajectory the LLR method outperforms the other memory-based methods. It seems that the other methods give an estimate that is too conservative. Figure 3-13 confirms this by showing that the RMS estimation error of LLR is smaller than the other memory-based methods for all memory sizes. Figure 3-14 shows that the difference in estimation quality is significant, so we conclude that fitting a linear model to the set of nearest neighbors leads to much better state-transition estimates. We argue that the improvement in estimation quality is worth the extra computational effort needed to estimate the linear model.

3-4-3 Humanoid robot

In this section we use a two-legged humanoid robot called 'Leo' as experimental setup. Leo is a humanoid robot developed at Delft University as a test platform to perform RL experiments (Figure 3-15). It was designed to be able to walk autonomously in circles. It is connected via a boom construction to a central rotating pivot, so it can theoretically walk infinitely long in circles. It has an arm that makes it possible to stand up when fallen. The setup is highly complex and is usable for performing advanced reinforcement learning experiments. We use it as a setup to test the ability of LLR to model complex, real-world systems. The robot is actuated at its angles, knees and

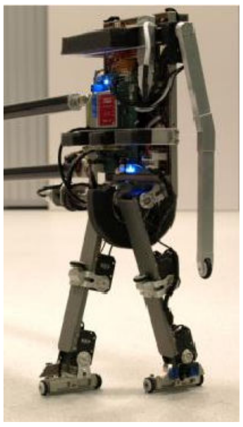


Figure 3-15: Two-legged humanoid robot 'Leo'.

hips. The motors also act as sensors that measure angle and angular velocity. Apart from the actuators, there are also sensors in the feet (that detect whether or not a foot touches the ground) and in the boom construction (measuring angle and angular velocity of the torso). In total, the robot has got 18 state-variables and 6 inputs. The states are concatenated in a 1×18 row vector $\mathbf{x}$ and the actions in a 1×6 row vector $\mathbf{u}$. The LLR model $\hat{\boldsymbol{\beta}}$ estimates the state-transition from a state $\mathbf{x}_t$ to the next state $\mathbf{x}_{t+1}$ when actuated with action $\mathbf{u}_t$:

$$\hat{\mathbf{x}}_{t+1} = \begin{bmatrix} \mathbf{x}_t \\ \mathbf{u}_t \\ 1 \end{bmatrix}^T \hat{\boldsymbol{\beta}}$$

A large number of memory samples was gathered by letting Leo walk for about two minutes sampling at 150 Hz using a pre-programmed controller. Although the walking motion looks to be the same in every step, closer inspection of the data reveals differences in every step (due to variations in the floor and other disturbances). The total dataset was split into two parts (both consisting of 8000 samples): one set for estimating the model and one for validating it. We used $K = 40$ for all Leo experiments. This value was determined empirically to give satisfactory results.

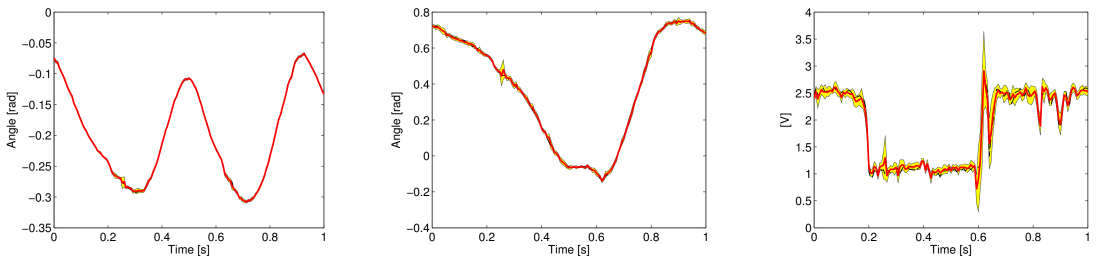
We have used LLR to model this system to investigate its performance on modeling complex systems. The questions to be answered are:

1. Can LLR be used to model a complex, high-dimensional system?
2. How large does the memory need to be for a reasonable estimation?

Modeling walking motion

To show that the LLR method is able to accurately model a high-dimensional system, we estimated the state-transitions of the validation dataset. The LLR estimates were determined using the estimation dataset ($N = 8000$) as memory. Figure 3-16 shows the response of the real system and its LLR estimate of one stride (two steps). For clarity we do not show all 18 state-variables but only the torso, the left hip and left foot (the estimates of all 18 state-variables can be found in Appendix B). We show these three variables, since they represent the remaining state-variables very well. Furthermore, since all estimated steps in the dataset are estimated approximately equally good, we show only one stride.

We notice that in all three cases the output is estimated very accurate. In fact, the measured states are barely visible due to the very close estimates of the LLR model. Also the prediction intervals are small (especially for the torso and the left hip), which indicates that the estimates are based on sets of very linear data. Although all variables are estimated accurately, the torso angle is clearly estimated best and the foot contact worst. This is reflected by the prediction intervals, which are small for the torso and hip, but larger for the foot. The main reason for this, is that the foot sensor data is very noisy. This is also visible from inspection of the measured data.



(a) Torso angle (b) Left hip angle (c) Left foot contact

Figure 3-16: LLR estimate ($K = 40$) of the walking motion of robot Leo using a memory consisting of 8000 samples. Three of the total of 18 state-variables are shown for one stride (two steps). (a) shows the angle of the torso, (b) shows the angle of the left hip, (c) the voltage of the left foot sensor. The figures show the LLR estimate (solid red line), the measured output (dashed black line) and the prediction interval (shaded area).

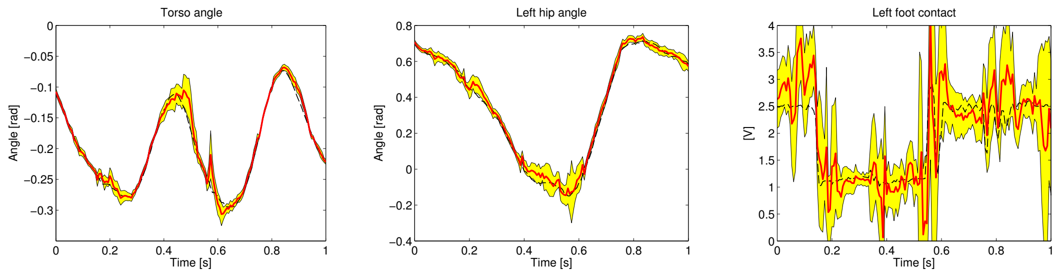
Modeling using increasing memory

In order to see the effect of the memory size on the estimation quality, we will again estimate the walking motion, but this time using a memory that increases in size. Starting with an empty memory, a new sample from the estimation dataset is added to the memory at every time step. This resembles a learning experiment which is started from no initial knowledge about the environment, but acquires one new state-transition at every time step.

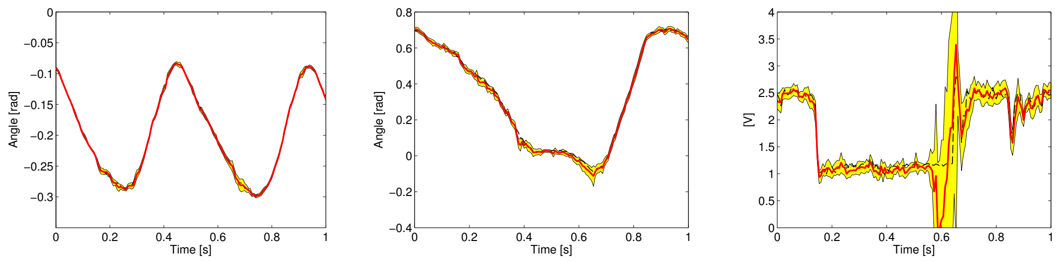
Figure 3-17 shows the improvement of the LLR estimate for state-variables torso angle, left hip angle and left foot contact. We show the LLR estimates of the three variables for increasing memory size. The torso angle and left hip angle are estimated quite accurately, even from a very small number of memory samples. This is probably due to the similarity between different footsteps for these variables. On the other hand, the foot contact variable is estimated relatively badly, particularly for a small number of memory samples. This is visualized by the bad estimates and the large prediction intervals. As noted before, the bad estimate of the foot sensor is probably due to the noisy and therefore unpredictable behavior of this variable.

Figure 3-18 shows the RMS error of the estimation for an increasingly large memory. The estimation error variates during a step. In order to obtain a more smooth graph, we averaged the RMS value over an estimated step (about 0.5 s). So, the graph shows the average RMS error of about 100 estimated steps. As expected, we see that the estimation error decreases with increasing memory size. The error decreases up to approximately $N = 2000$ and is more or less constant thereafter. This is approximately equal to 13 strides. So the LLR method is able to estimate future steps accurately from 26 previously observed steps. Along with the RMS error of the estimation, we also plotted the RMS value of the prediction interval. We notice that the prediction interval has a remarkably similar shape. Because the prediction interval can be considered as a measure of linearity, this indicates that the size of the estimation error can be related to the nonlinearity of the memory samples. This is a direct consequence of the lack of appropriate nearest neighbors, which then leads to an inaccurate estimate of the state-transition for a particular state-action pair.

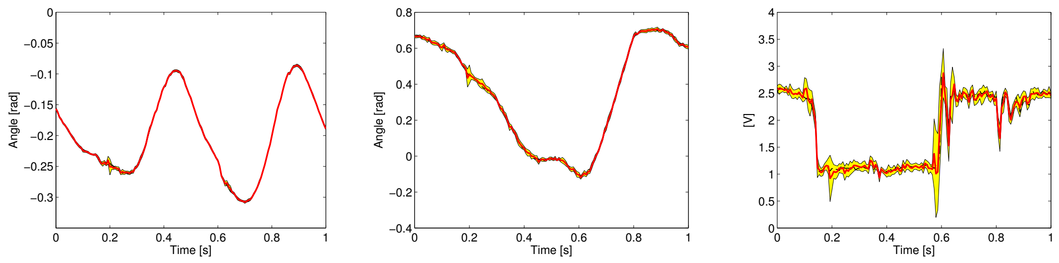
Although walking seems to be a motion that is the same in every step, in practice every single step is slightly different due to small disturbances such as ground level variations. This is reflected by sudden increases in the estimation error, for example around $N = 4500$. Apparently this step was very different from the previously observed steps and the lack of appropriate nearest neighbors led to a LLR estimate that is much worse than for previously estimated steps. This is confirmed by comparing the actual measurements of this step to other steps.



(a) Memory size: $N = 150$ (2 steps)



(b) Memory size: $N = 1000$ (14 steps)



(c) Memory size: $N = 6000$ (80 steps)

Figure 3-17: Improvement of the LLR estimate ($K = 40$) of the walking motion of Leo when the memory size increases. The figures show the improvement of the LLR estimate for the torso (left), the left hip (middle) and the left foot (right). (a) is the estimate for a memory of size $N = 150$ (roughly 2 steps), (b) the estimate for $N = 1000$ (14 steps) and (c) for $N = 6000$ (80 steps). The figures show the LLR estimate (solid red line), the measured value (dashed black line) and the prediction interval (shaded area).

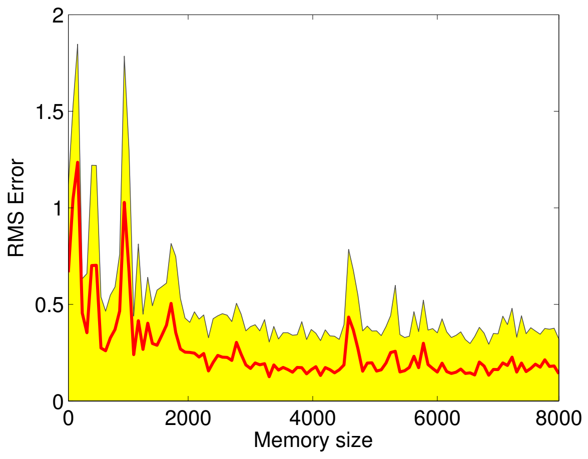


Figure 3-18: RMS value of the LLR estimation error (solid red line) for the walking motion of Leo for increasing memory size. The shaded area is the RMS value of the prediction interval.

3-5 Conclusions

In this chapter we have introduced Local Linear Regression as method for estimating a model from a set of observed state-transitions. Being a memory-based method, it is easy to implement several results from statistical analysis. Although the number of statistical quantities is vast, we have only described prediction intervals and outlier detection which could be useful in a reinforcement learning setting. In order to decrease the computation time, we introduced a kd-tree as a structure to store the samples in the memory. We have shown how the tree structure can be used to search for nearest neighbors more efficiently than a standard sorting approach.

The capabilities of the LLR method were shown in different settings. We used a 1-dimensional nonlinear function to research the possibility to increase the prediction accuracy. We showed that optimizing the number of nearest neighbors locally, can in some cases reduce the estimation error. Also outlier detection was able to identify spikes in the data and resulted in better estimates by neglecting these faulty samples. However, the extra computational effort that is needed to improve the estimate only slightly, makes it not interesting for on-line usage. The prediction interval is a measure for the linearity in the points used in the linear regression and can therefore be used as a rough estimate of the uncertainty in the prediction.

We applied LLR on a two-link manipulator simulation to test its ability to estimate state-transitions. We showed that LLR is indeed usable for estimating state-transitions accurately. As expected, increasing the number of samples in the memory, led to an improvement in the estimation accuracy. Furthermore, we showed that fitting a linear model to the set of nearest neighbors (as is done in LLR) makes the estimation more accurate compared to memory-based methods that use the average or the nearest neighbor as output.

An important question was whether the LLR modeling method still performed well on a complex, noisy system. We answered this question by applying LLR on a complex, humanoid robot setup. The LLR method proved to be able to model the walking motion of the robot accurately using a relatively small memory of a few thousand samples. One reason that the model is accurate lies in the fact that in all experiments the estimation and validation data are in the same region of state-space. Although this might seem to be an overly ideal situation, this is in fact a reasonable imitation of a real learning process. Not all parts of the state-space are expected to be equally important and we suspect that a perfect model is not needed in all parts, but mainly along trajectories that lead to goal states.

In this chapter, the accuracy of a model was only determined by looking at the response of the model compared to the real system and by inspecting the prediction intervals. The important question whether or not a model is accurate enough to be used in a RL setting is deferred to the next chapter where it will be discussed in more detail.

Chapter 4

Model-learning RL

4-1 Introduction

In this chapter, we combine Local Linear Regression (LLR) with Reinforcement Learning (RL). We use the LLR method as developed and investigated in Chapter 3. In this chapter, we describe how the LLR model is incorporated in on-line learning. We investigate whether it is possible to build a model during the learning process that can be used to accelerate the learning process. We use Dyna and Prioritized Sweeping (PS) as learning algorithms. We are not necessarily interested in achieving the best performance for each algorithm, but in investigating the difference between them. Therefore, we use a fixed problem setting to test the different algorithms.

This chapter is organized as follows. First, the learning problem is defined and some of the basic learning parameters are described in Section 4-2. In Section 4-3 we describe how the LLR model is implemented in the learning process and describe some of the difficulties. Finally, the results of the learning experiments are presented in Section 4-4. The results were obtained using SARSA, Dyna and PS.

This chapter only deals with on-line learning experiments. We have also tried an off-line approach to test the usability of a LLR model in a RL setting. In Appendix C, we describe off-line, model-based Q-iteration experiments that were conducted. However, these experiments did not lead to satisfactory results and were not researched in depth.

4-2 Problem setting: Inverted pendulum

In order to be able to compare different algorithms, we use a fixed learning problem. The system is a simulation of the DCSC inverted pendulum setup (Figure 4-1). The dynamics are given as:

$$J\dot{\omega} = mgl\sin\theta - \left(b + \frac{K_t^2}{K_R}\right)\omega + \frac{K_t}{K_R}u \tag{4-1}$$

The state-vector is two-dimensional and consists of the angle θ and the angular velocity ω. The action-space $\mathcal{A}$ is discretized and consists of three actions $u = \{-3V, 0, 3V\}$. The goal is to drive the pendulum from $[\pi, 0]$ to $[0, 0]$, i.e. to move the mass upwards. The control voltage is not high enough to drive the mass upwards in once, instead the mass has to rock back and forth to build up enough energy to make it to the top. This task can be seen as a variation of the well-known car on the hill problem that is often used in RL.

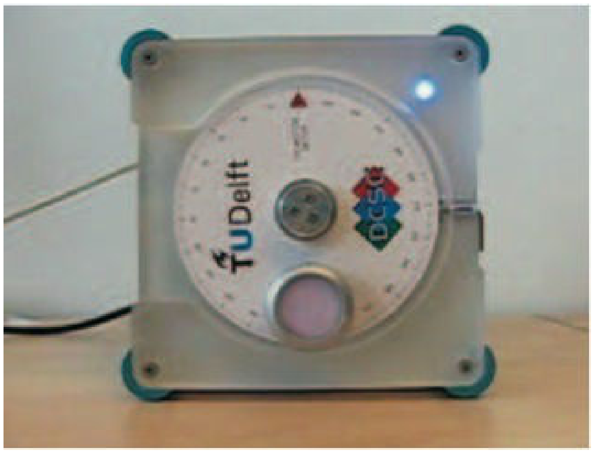


Figure 4-1: Inverted pendulum setup.

The inverted pendulum was simulated in MATLAB with a sampling time of $T_s = 0.03$ s. No noise was added to the system. The angle was wrapped to $\theta \in [-\pi, \pi]$ and the angular velocity was limited to $\omega \in [-10\pi, 10\pi]$.

Table 4-1: Physical parameters of the inverted pendulum.

Symbol	**Parameter**
$m = 0.1$ kg	Pendulum mass
$J = 1.91 \cdot 10^{-4}$ kg/rad^2	Pendulum inertia
$g = 9.81$ m/s^2	Gravitational acceleration
$l = 0.1$ m	Pendulum length
$b = 0.01$ kg/s	Damping
$K_R = 9.5\Omega$	Rotor resistance
$K_t = 5.36$ Nm/A	Torque constant

4-2-1 Reward function

The reward function can be implemented in two different ways. The first is a box-style reward. This reward originates from the idea that the goal state results in a reward and the other states in a penalty. As we are dealing with continuous states, we have to define a region around the goal that is considered close enough to the goal state. The box-style reward is defined as follows:

$$r(\theta, \omega) = \begin{cases} +10, & \text{if } |\theta| < \frac{10\pi}{180} \text{ rad and } |\omega| < 0.3 \text{ rad/s} \\ -1, & \text{elsewhere} \end{cases}$$

Where we have defined a region around zero that we consider as the goal region. With this type of reward, a learning trial is usually stopped when the goal region has been reached.

Another option that is often used, is to use a smooth reward that gives a (negative) reward for every state-action pair:

$$\rho(\theta, \omega, u) = -\begin{bmatrix} \theta & \omega \end{bmatrix} Q_{\text{lq}} \begin{bmatrix} \theta \\ \omega \end{bmatrix} - uR_{\text{lq}}u$$

with $Q_{\text{lq}} = \begin{bmatrix} 5 & 0 \\ 0 & 0.1 \end{bmatrix}$ and $R_{\text{lq}} = 0.01$. This is a lqr-style reward and it will also drive the system to the goal state $[0, 0]^T$. Preliminary experiments showed that this reward resulted in a much faster learning speed than the box-reward, so we choose the smooth reward for the learning experiments.

A learning problem with lqr-rewards does not have an explicit goal state so learning trials are continuing. We have chosen to stop a trial whenever the goal region (as defined in the box-style reward) has been reached. This made no difference for the learning speed or the resulting policy. This can be explained by the fact that the system does not learn anything by moving around in the goal region. Adding a stopping criterion did reduce the simulation time significantly, since a large number of trials is stopped before maximum time for the episode ran out.

It has to be noted that although both reward structures have $[0, 0]^T$ as goal, the learning problem is not the same. The box-style reward results in a solution that gives the time-optimal policy to drive the system to the goal. The smooth reward results in a solution that is optimal with respect to the cost function ρ. This function also depends on the input signal u, so the optimal policy will try to limit the control signal as well as getting to the goal. The resulting policy will therefore not be time-optimal.

4-2-2 Learning parameters

Tuning a RL algorithm boils down to tuning three learning parameters: the discount factor γ, the learning rate α and the exploration rate ϵ. The discount factor defines the solution to the learning problem and should therefore be fixed during the entire experiment. The learning rate influences how fast the value function is updated and thus how aggressively the agent learns. For noisy or stochastic experiments a high learning rate might lead to 'unlearning'. The exploration rate defines the percentage of random actions taken by the agent. The value may be constant or time-varying. The agent needs to explore in order to experience new state-action pairs and to avoid the learning of a sub-optimal policy. Unfortunately, no general rules for setting these parameters exist in the literature and the parameters have to be tuned empirically.

We want to investigate the difference in performance between the RL algorithms, we are not necessarily interested in the best performance for each case. Therefore, we did not try to fine-tune the parameters for every experiment, but set the tuning parameters to fixed values for all experiments. For γ and ϵ this seems reasonable. The first defines the solution and the second should have minor influence if multiple experiments are conducted. For the learning rate α this might be questionable, because this factor immediately influences the update of the value function (Eq. (2-6)). The learning rate might have an optimal value for different experimental settings. For noisy environments or inaccurate models for instance, a low learning rate might be preferred. In noise-free environments with a very accurate model, a high learning rate might be used. Also different algorithms might have different optimal learning rates.

The values of the learning parameters in a RL experiment can be chosen freely and no rules for determining the optimal values exist. We chose the parameter values based on initial experiments and scientific intuition. The learning parameters were set as follows:

$$\begin{aligned}\alpha &= 0.8\\ \gamma &= 0.98\\ \epsilon_{n_t} &= \max(0.1 \cdot 0.99^{n_t}, 0.001)\end{aligned}$$

Where n_t is the number of the current trial, so the exploration is decreased in every next trial, with a minimum of 0.001. A decreasing exploration rate led to faster learning than a fixed exploration rate and it led to a more 'smooth' learning curve. However, decreasing the exploration rate to zero led to occasionally finding a sub-optimal solution, so therefore the exploration was minimized to a small value.

4-2-3 Value function approximation: tile coding

In the introduction of the value function in Section 2-2-1, it was assumed that a value is assigned to every state. This would lead to a tabular implementation of the value function in which every state has a value. As our state-space is continuous, this approach is impossible. Even discretizing the state-space into discrete bins, would still lead to a very large table. Therefore, we have to approximate the value function in some way.

Tile coding (Figure 4-2) is a commonly used way to approximate functions, because it is fast and easy to use. We used tile coding as a function approximator for the action-value function. We used 20 tilings of 30×30 tiles each, which led to a final resolution of approximately $\Delta_\theta = 0.01$ for the angle θ and $\Delta_\omega = 0.1$ for the angular velocity ω. These values are exact only if the tilings are displaced evenly with respect to each other. Because we use random displacements, the given resolutions are approximations. The tiles were initialized with a random value in the interval $[0, 1]$. Again, we are not interested in achieving the best performance for every algorithm, but in creating a fixed experimental setting for all experiments. It is expected that the value-function will only influence the quality of the final policy and not the difference between the policies of different algorithms.

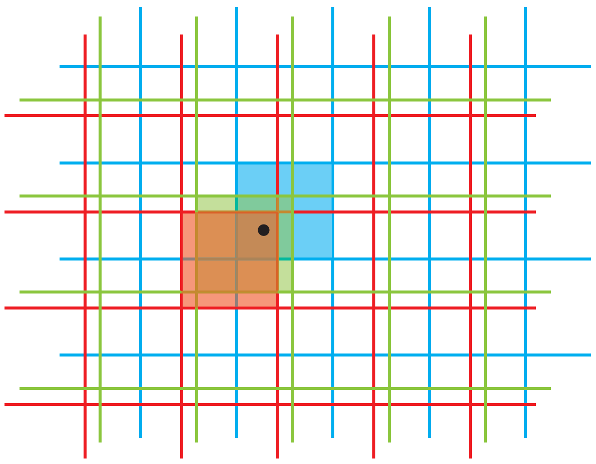


Figure 4-2: Schematic overview of Tile Coding. The black dot represents a continuous value. The colored grids represent three tilings and the continuous value is approximated by the shaded tiles.

In all our experiments the actions available to the controller form a discrete set, so only a limited number of actions can be used. This is a common approach in RL because it limits the number of state-action pairs. Hence the action-value function only has to generalize over the states and not over the actions.

4-3 Implementation of Local Linear Regression

In this chapter we use two model-learning methods: Dyna and PS. We use LLR as a model to generate state-transitions in Dyna and to determine lead-in states in PS. Prediction intervals are used as a possible accuracy measure of the modeled output. The number of nearest neighbors used in the linear regression will be fixed to $K = 5$, a value that is based on the results with the two-link manipulator in Section 3-4-2.

Every experiment started with an empty memory, which was updated with experienced state-transitions during the learning process. A kd-tree structure was used to represent the memory as described in Section 3-3-1. New samples were added to the kd-tree immediately. In order to maintain a balanced tree, we re-built the entire tree after a learning trial was completed. After the first 1000 samples, the tree is not re-built between trials anymore and samples are added to the existing tree only. The main reason for this is that the re-building consumes too much time for many samples. Furthermore, the sample-density will be sufficiently high so that new samples will not unbalance the tree severely. The 1000-sample border is chosen more or less arbitrarily, based on experience gained with the LLR experiments of Chapter 3.

4-3-1 Prediction intervals

It is expected that the accuracy of the LLR model improves as the number of memory samples increases and that it can vary over the state-space. Our approach to assess the quality of the model is to use prediction intervals around the modeled state-transition (see Section 3-3-2).

We will introduce a limit on the prediction interval which has to guarantee accurate model outputs. The calculated prediction interval I_x will be compared to an user imposed limit δ_x. Whenever the prediction interval is larger than the maximum value in any dimension, the modeled state-transition is discarded. We can write this mathematically as follows:

$$\hat{y}_q = \begin{cases} \text{LLR}(x_q) & , \quad \text{if } I_{x_i} \leq \delta_{x_i} \text{ for } i = 1, 2, \ldots, k \\ \text{no answer} & , \quad \text{if } I_{x_i} > \delta_{x_i} \text{ for } i = 1, 2, \ldots, k \end{cases}$$

If the prediction interval is smaller than this limit, the state-transition will be considered accurate enough to be used. If the prediction interval is larger, the estimated state-transition will be discarded and not used for learning. In our experiments a discarded estimation was not replaced by a different (possibly accurate enough) estimation. This replicates the real-time situation in which only a limited amount of time is available to estimate state-transitions. Endlessly calculating state-transitions until an accurate estimate is found, is therefore not possible.

As discussed earlier, we have no clear idea how small the interval has to be in order to indicate that the estimation is good enough to be used for RL. The main problem is that we have no quantity to relate the interval to. That is, if this quantity exists at all. It might also be true that very inaccurate predictions can be used as long as they meet certain restrictions. These restrictions may vary, depending on the learning problem.

At this point, we choose for the prediction interval a limit that is related to the final resolution of the value function approximator. The idea behind this choice is based on the discriminating power of the model and the value function approximator. It seems reasonable that the optimal situation is a value function approximator and a model that have about the same 'resolution'. Increasing the accuracy of only one of them, would only have a minor effect as the overall accuracy would be limited by the other. Computational resources are therefore used optimally if the model and the value function have a similar accuracy.

We will use two prediction interval limits. The first is based in the tile size in each tiling. As we divide each tiling into 30×30 tiles, the tile size is $\frac{2\pi}{30} \times \frac{20\pi}{30}$. The corresponding prediction interval limit is $[\delta_\theta, \delta_\omega] = [0.2094, 2.094]$. The second limit is based on the final resolution of the tile coding approximator. As we used 20 tilings, the approximate final resolution is $[\delta_\theta, \delta_\omega] = [\Delta_\theta, \Delta_\omega] = [0.01, 0.1]$. Figure 4-3 shows the prediction interval limit for a certain modeled state-transition.

4-3-2 Lead-in states

A crucial part in the PS algorithm is determining lead-in states (Algorithm 4, line 11). This process is not well described in literature. It is simply assumed that a model is available and that this model can be used to determine lead-in states. The type of model is not specified. We will describe our approach in this section.

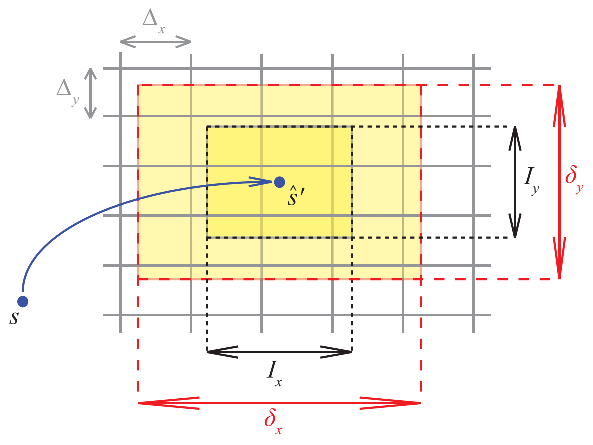


Figure 4-3: Limit of the prediction interval $[\delta_x, \delta_y]$ compared to the resolution of the value function $[\Delta_x, \Delta_y]$. The figure shows an estimated state-transition $s \to \hat{s}'$ with a prediction interval $[I_x, I_y]$. Because $[I_x, I_y]$ falls inside $[\delta_x, \delta_y]$, the corresponding prediction is considered accurate enough and is used for learning.

First we have to note that we are actually not searching for lead-in states, but for lead-in state-action pairs. Consider the LLR state-transition model M of the system. The state-transitions are then described by:

$$s_{t+1} = M \begin{bmatrix} s_t \\ a_t \\ 1 \end{bmatrix}$$

So we are actually searching for a state s_t and an action a_t that will lead to s_{t+1}. However, modeling the inverse mapping $s_{t+1} \to (s_t, a_t)$ would lead to two problems. First, one input could have several possible outputs (i.e., several state-action pairs (s_t, a_t) can lead to s_{t+1}). So a unique model for this mapping does not exist. The second problem is that when this model would be used, we would obtain continuous values of s_t and (more importantly) a_t. This is unwanted as we are assuming a discrete valued action-space.

Our approach tries to solve these two issues. We consider the mapping $(s_{t+1}, a_t) \to s_t$. This mapping can be modeled with an alternative model $\tilde{M}$ for which:

$$s_t = \tilde{M} \begin{bmatrix} s_{t+1} \\ a_t \\ 1 \end{bmatrix}$$

Because the action is now used as an input, we can choose a discrete value. The proposed mapping might seem counter intuitive, as the inputs are samples at different time steps. But this model should not be seen as an actual state-transition model. It should rather be regarded as a data mapping. We simply consider the sample (s_{t+1}, a_t) as the input that leads to the output sample s_t. Note that the model $\tilde{M}$ is different from the state-transition model M. It is again obtained using LLR techniques, but it requires a differently structured memory and a different input vector.

Now, for the target state s the lead-in state-action pairs ($\bar{s}$,$\bar{a}$) can be determined using the routine of Algorithm 5. Note that every state, can have multiple lead-in states. Furthermore, we use the prediction interval limit (as described in Section 4-3-1) to possibly discard an estimated lead-in state. The notation $\bar{s}$ is used to denote a lead-in state to s. Subscripts containing time are usually avoided.

Algorithm 5 Lead-in states

```
1: given $s$                                              ▷ Target state
2: for each $a \in \mathcal{A}$ do
3:     $\tilde{M} \leftarrow \text{LLR}(s, a)$            ▷ Estimate LLR model $\tilde{M}$
4:     $\hat{s} \leftarrow \tilde{M} \begin{bmatrix} s & a & 1 \end{bmatrix}^T$
5:     if $I_{\hat{s}} \leq \delta_s$ then                ▷ If prediction interval is sufficiently small
6:         $(\bar{s}, \bar{a}) \leftarrow (\hat{s}, a)$ is a lead-in state
7:     end if
8: end for
```

4-4 Experimental results

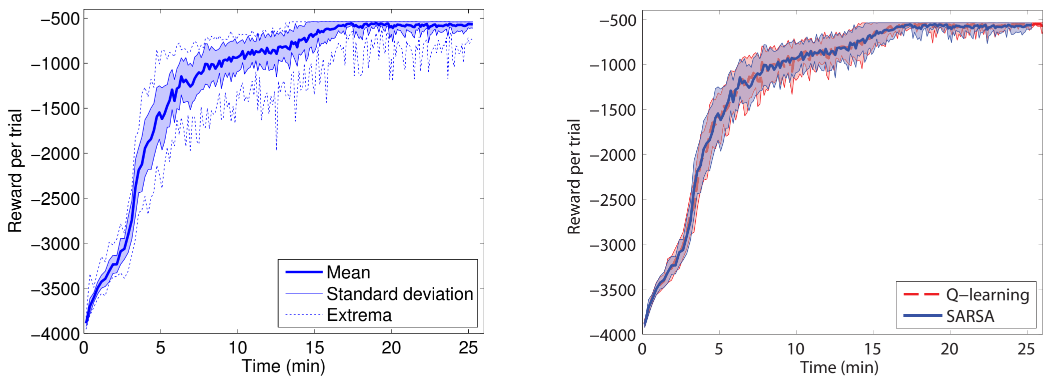
We now present our experimental results. As explained in the beginning of this chapter, we used the same experimental setup and parameters for all experiments. All results shown in this section, are averages of 30 separate learning experiments. Every experiment consists of 1000 learning trials and every trial (or episode) consisted of a maximum of 100 time steps (equivalent to 3 seconds simulated time). This results in a maximum learning time of 50 minutes per experiment. Every experiment starts with a cleared LLR memory and a randomly initialized value function.

4-4-1 SARSA

We have introduced the on-line, model-free algorithm SARSA in Section 2-3-2. This is the most basic form of on-line learning. At every time step, one value function update is executed based on one real state-transition. SARSA can be considered as a 'benchmark' to which Dyna and PS can be compared. In order for LLR to be useful, the model-learning algorithms should perform better than a purely model-free algorithm. Furthermore, SARSA was used to get insight in the learning parameters and the optimal policy.

Figure 4-4(a) shows the learning curve for the SARSA algorithm. The figure shows the reward per trial (a 3 second experiment), averaged over 30 separate experiments. We also show the standard deviation, minimum and maximum values in order to gain insight in the spread of the learning curve. The graph shows very rapid learning during the first 5 minutes. Thereafter, the policy strongly improves further until about 15 minutes of learning. The reason that the curve still fluctuates slightly thereafter is due to the exploration.

Closer inspection of the experimental data, showed that it takes on average about 3 minutes to reach the goal for the first time. Although the maximum duration of a learning experiment is 50 minutes, the added stopping criterion leads to an average experiment length of 25.4 minutes. During this period the SARSA algorithm used on average $5.1 \cdot 10^4$ real experiences to learn.



(a) SARSA learning curve (b) SARSA compared to Q-learning

Figure 4-4: Learning curves of model-free learning applied on the inverted pendulum simulation. (a) shows the reward per trial during an experiment consisting of 1000 trials using SARSA as learning algorithm. The graph shows the average value (solid line), the standard deviation (shaded area) and the extreme values (dashed lines) of 30 separate experiments. (b) compares SARSA and Q-learning.

We could also have used the model-free algorithm Q-learning (Section 2-3-2) instead of SARSA. Preliminary experiments with the two algorithms (Figure 4-4(b)) showed no significant difference in learning speed between the two algorithms. When combined with function approximators, SARSA has better convergence properties than Q-learning [5], so we will use SARSA to update the value function in the remaining of this chapter.

The learning curves presented in the chapter show the reward per trial. The results show that the agent learns a policy along the trajectory towards the goal. However, this does not necessarily mean that the policy has converged for the entire state-space. In Appendix D we show the final policy for all learning algorithms and compares them to the optimal policy.

4-4-2 Dyna

We now add the LLR model to the learning process and combine it with the Dyna algorithm (see Section 2-3-3). The modeled state-action pairs are chosen randomly throughout the state-space. The LLR model is built and used as described in Section 4-3. Initially the memory is empty and every new experience is immediately added to it. We generated $N_m = 3$ state-transitions every time step. Prediction intervals of the LLR model are used to possibly discard an estimate if it exceeds a given limit (see Section 4-3-1).

We compared several values for the maximum prediction interval. Due to the limit on the prediction interval, not all modeled state-transitions are used by the learning agent. Table 4-2 summarizes the number of iterations and value-function updates of the different settings. It is clear that setting this limit tighter, leads to more discarded model-outputs. In the case of a very tight limit of $[0.001, 0.01]$, 86 % of the modeled outputs get discarded. In the case of a less tight limit of $[0.2094, 2.0944]$, only 37% get discarded.

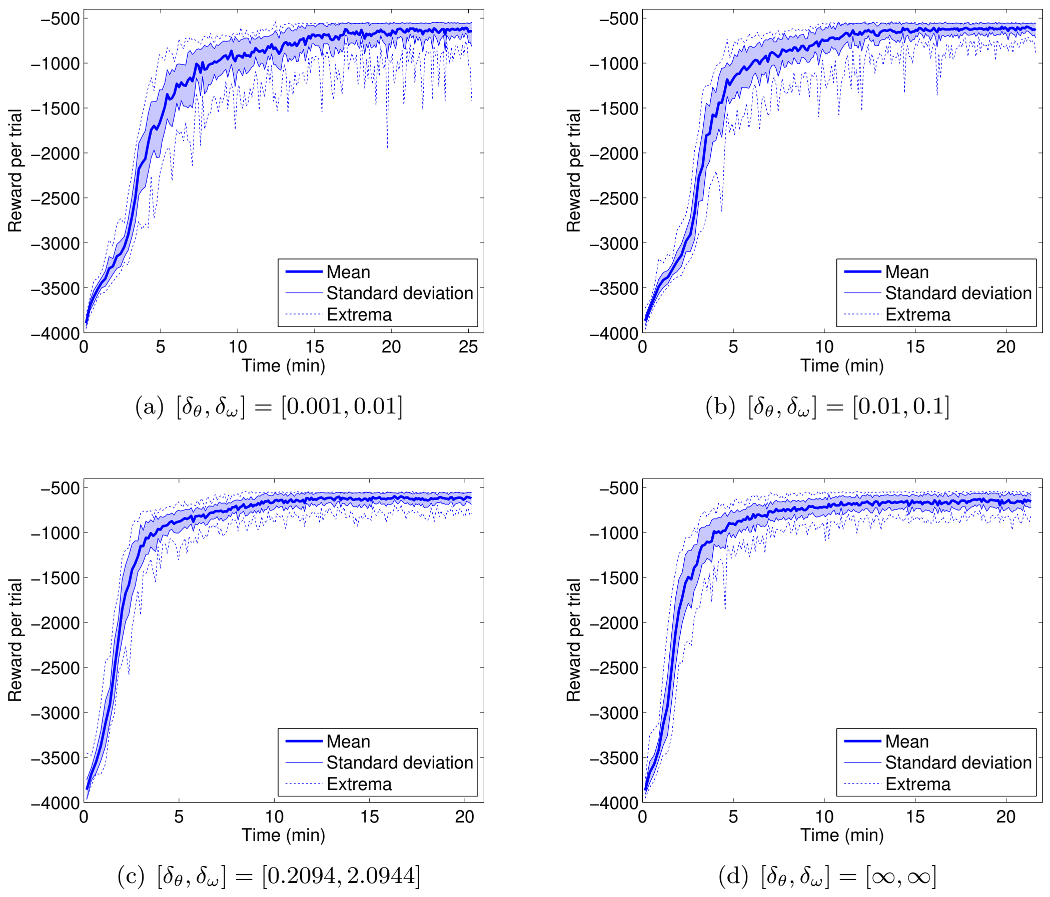
Table 4-2: Results of the learning experiments on the inverted pendulum using different algorithms. The results are averages of 30 experiments, consisting of 1000 trials, generating $N_m = 3$ modeled state-transitions per time interval. 'Iterations' are the total number of simulated time steps, 'Updates' are the total number of value-function updates (model-free and model-based), 'Ratio' is the ratio of model-free updates to model-based updates.

Algorithm	$[\delta_\theta, \delta_\omega]$	**Iterations** $[10^4]$	**Updates** $[10^4]$	**Ratio**
SARSA	-	4.95	4.95	1:0
Q-learning	-	5.11	5.11	1:0
Dyna	$[0.001, 0.01]$	4.94	6.98	1:0.41
(LLR model)	$[0.01, 0.1]$	4.22	8.73	1:1.07
	$[0.2094, 2.0944]$	3.97	11.93	1:2.01
	$[\infty, \infty]$	4.17	16.69	1:3
Dyna (exact model)	-	3.65	14.61	1:3
Prioritized Sweeping	$[0.001, 0.01]$	4.77	9.30	1:0.95
(LLR model)	$[0.01, 0.1]$	4.76	13.93	1:1.93
	$[0.2094, 2.0944]$	4.38	16.59	1:2.79
	$[\infty, \infty]$	6.31	25.24	1:3
Look Ahead Dyna	$[0.001, 0.01]$	4.71	12.61	1:1.68
(LLR model)	$[0.01, 0.1]$	4.42	15.38	1:2.48
	$[0.2094, 2.0944]$	4.14	16.01	1:2.87
	$[\infty, \infty]$	4.42	17.67	1:3

Table 4-3: Computation time needed to perform all calculations in a single time step of the different learning algorithms for $N_m = 3$. The table shows the mean time and the maximum time needed (during the entire experiment). The results were obtained using a 3 GHz desktop computer. Notice that the sampling time of the simulation is $T_s = 30 \cdot 10^{-3}$ s.

Algorithm	**Mean** $[10^{-3}$ s$]$	**Max** $[10^{-3}$ s$]$
SARSA	1.01	1.94
Dyna	2.75	5.97
Prioritized Sweeping	4.65	8.51
Look Ahead Dyna	2.38	4.71

We now continue with a closer inspection of the performance of the different LLR settings. Figure 4-5 shows the reward per trial during the learning process for the different settings. We notice that a large limit on the prediction interval leads to faster learning. The learning speed differs particularly in the first five minutes of the learning process. Furthermore, we notice that the spread in the learning curves increases with decreasing prediction interval limits. For very tight limits, the spread is very similar to the SARSA result. Using many uncertain outputs is preferred as this results in a smaller spread of the learning curve. This might be counter-intuitive as it is expected that highly inaccurate state-transitions lead to faulty value-function updates. However, this effect is apparently counteracted by the high number of value-function updates.

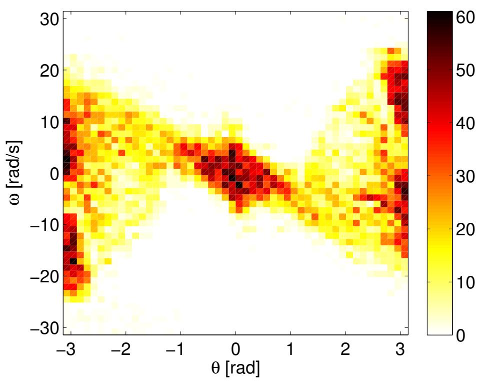


(a) $[\delta_\theta, \delta_\omega] = [0.001, 0.01]$ (b) $[\delta_\theta, \delta_\omega] = [0.01, 0.1]$ (c) $[\delta_\theta, \delta_\omega] = [0.2094, 2.0944]$ (d) $[\delta_\theta, \delta_\omega] = [\infty, \infty]$

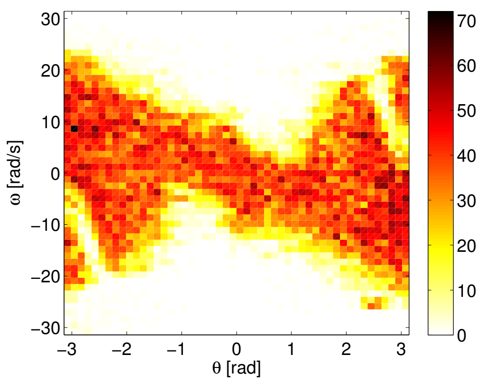
Figure 4-5: Learning curves of the Dyna algorithm applied on the inverted pendulum. (a), (b), (c) and (d) show the Dyna algorithm using LLR with increasing prediction interval limits.

Figure 4-6 shows another effect of the limit on the prediction interval. The figure shows the positions in state-space for which a modeled state-transition was generated and not discarded. It is clear that the sample density and the prediction interval are related. For a very tight limit (Figure 4-6(a)), only in the vicinity of the optimal trajectory enough samples are available for an estimated state-transition that is accurate enough. When the limit is stretched (Figure 4-6(b)-4-6(c)), so is the region in which the model is considered to be accurate enough. If no limit is imposed (Figure 4-6(d)), the modeled state-transitions in the entire state-space are used.

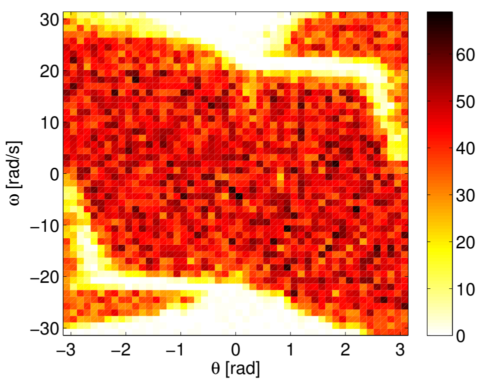
Figure 4-7 shows the average learning curve of the Dyna algorithms compared to model-free SARSA learning. We conclude that using model-generated experience in the learning process increases the learning speed. Increasing the limit on the prediction interval leads to more state-transition available to the agent. The graph shows that this leads



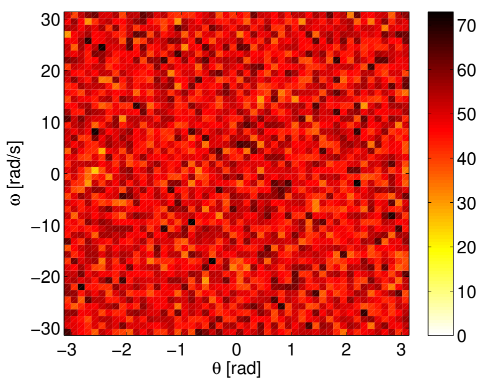
(a) $[\delta_\theta, \delta_\omega] = [0.001, 0.01]$



(b) $[\delta_\theta, \delta_\omega] = [0.01, 0.1]$



(c) $[\delta_\theta, \delta_\omega] = [0.2094, 2.0944]$



(d) $[\delta_\theta, \delta_\omega] = [\infty, \infty]$

Figure 4-6: Distribution of states for which a state-transition was generated using the LLR model in the Dyna setting. The figures only show the states for which the prediction interval was smaller than the given limit. The color indicates the number of times a state-transition was generated in a certain area.

to faster learning. The inaccuracy of these state-transition does not seem to influence the learning process negatively.

Although all algorithms converge, the final reward per trial is slightly higher for SARSA. This indicates that the final trajectory towards the goal for SARSA is slightly better, with respect to the reward function, than for Dyna. So we can conclude that adding state-transitions generated by the LLR model only has a minor negative effect on the resulting trajectory, even if highly inaccurate state-transitions are being used. Apparently, the number of real experience and accurate modeled state-transitions are high enough to counteract the negative effect of the inaccurate model outputs. Furthermore, it can be expected that the inaccurate state-transitions can mainly be found in rarely visited parts of the state-space (see also Figure 4-6). These regions have a limited effect on the trajectory towards to goal.

A question posed earlier was: is it more useful to use many inaccurate model-outputs or to use a limited number of very accurate outputs? If we regard the prediction interval as a good indication, we can answer this question by comparing the values in Table 4-2 with the learning curves in Figure 4-7. It can be concluded that the number of value-function updates is dominant over the quality of the state-transitions. In other words, the optimal setting for Dyna in this test setup is to use as many model-outputs as possible by setting the limit on the prediction interval to a high value.

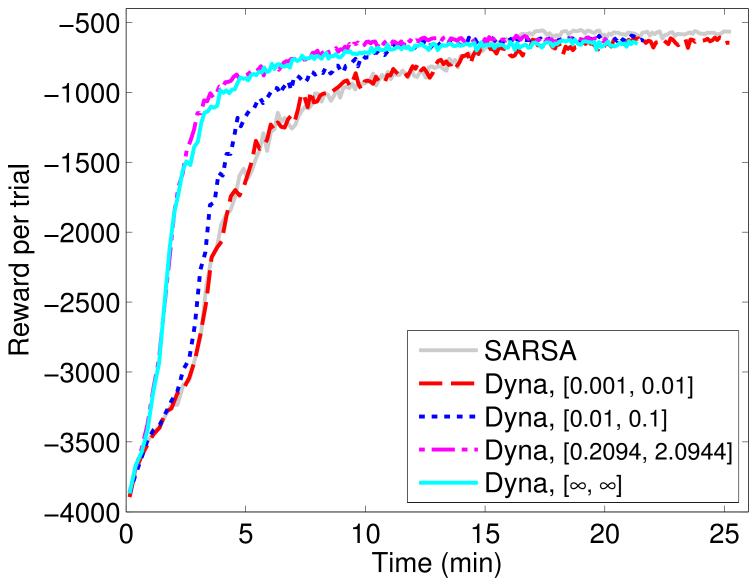


Figure 4-7: Comparison of the learning curves using SARSA and Dyna for the inverted pendulum for several values of the prediction interval limit $[\delta_\theta, \delta_\omega]$.

From the results in this section we can conclude that generating state-transitions using the LLR model can increase the learning speed compared to model-free SARSA. We have seen that limiting the prediction interval influences the learning speed. However, we do not have an indication of the accuracy of the modeled state-transitions. For this, we need the true state-transitions. These can be obtained from Eq. (4-1). We have applied Dyna using the exact model. This represents the optimal Dyna setting: a perfect model of the system throughout the state-space. We define the prediction interval of the exact model as $I = [0, 0]$. In Figure 4-8 we compared the exact model to the optimal LLR model.

We notice that the curves are very close together. This means that the LLR model does not disturb the learning process compared to the true model. In the first minutes of learning, both methods perform almost equally. This is surprising, as it is expected that early in the learning process the LLR is not very accurate yet because of the low number of memory samples. Apparently, possibly inaccurate model outputs are still suitable for learning in the beginning. In the remaining of the learning process, the exact model has a very slight advantage over the LLR model.

The inaccuracy of the LLR model is best seen in the later stages of the learning process. Although both learning curves converge to more or less the same value, the standard deviation of the exact model is about half the size of the LLR model. This is caused by the inaccurate model outputs that are used by the Dyna algorithm using the LLR model.

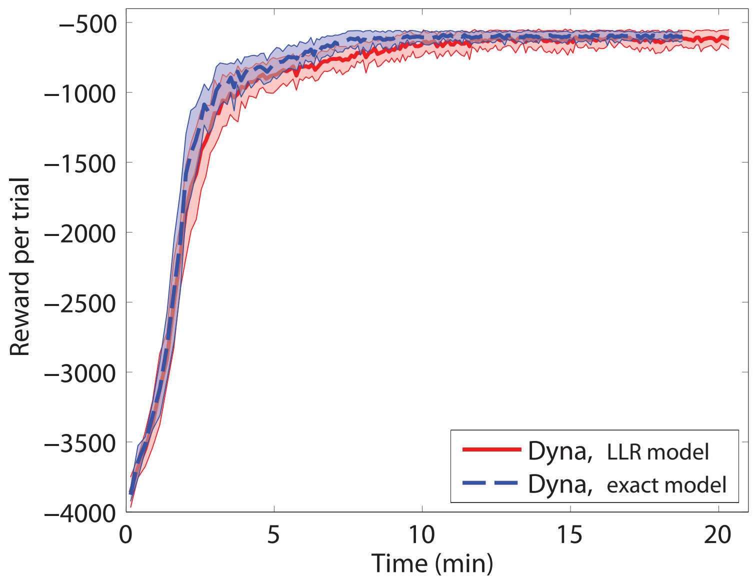


Figure 4-8: Comparison of the learning curves using Dyna with the LLR model and the exact model on the inverted pendulum. We used $[\delta_\theta, \delta_\omega] = [0.2094, 2.0944]$ for the prediction interval limit of the LLR model, which is the optimal setting.

In Section 3-3-1 we explained how we used a kd-tree to represent the memory in order to increase the computational speed of the LLR model. Although in these simulation experiments the computation time is not limited, in real-time experiments all the calculations need to be performed within the sampling interval. The sampling time of the simulation was set to $T_s = 0.03$ s. Table 4-3 shows the time needed by the learning algorithms in every sampling interval to perform all the calculations. The exact computation times will depend on the hardware and software used, so the values should be regarded only as indicative.

4-4-3 Prioritized Sweeping

We continue with PS as a learning algorithm. Our PS implementation will combine PS (Algorithm 3) with LLR. The lead-in states will be calculated as described in Section 4-3-2 and we set limits on the prediction interval in the same way as we did for Dyna.

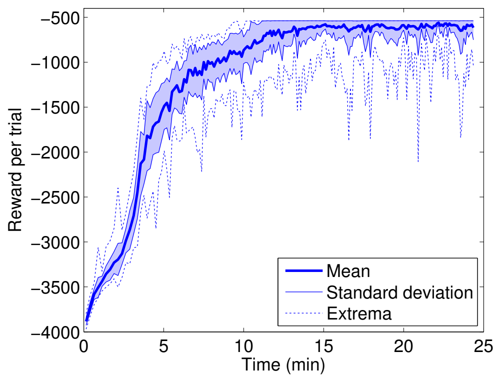
First we investigate the influence of the prediction interval limit. Figure 4-9 shows the learning curves for the PS algorithm for several limits. We notice that the results differ in many aspects from the Dyna results. First we notice that for PS we find an optimal value for the prediction interval limit. Out of the four intervals we have compared, $[\delta_\theta, \delta_\omega] = [0.2094, 2.0944]$ performs best. This limit performs best both in terms of learning speed and in the variance of the learning curves. The other interval limits lead to worse results, with the worst result for $[\infty, \infty]$. For this large limit, the learning curve shows very slow learning, a large spread and no convergence to the optimal policy. Apparently the number of inaccurate state-transitions is so large that they severely disturb the learning process.

Figure 4-10(a) compares the learning speed of the different PS settings. It is clear that PS needs a limit on the prediction interval. The setting with no limit leads to a very slow learning speed, even slower than SARSA. In the 1000-trial experiment, this setting did not converge to a solution. Also the setting with very tight bounds does not perform well. This setting leads to a learning speed that is similar to model-free SARSA.

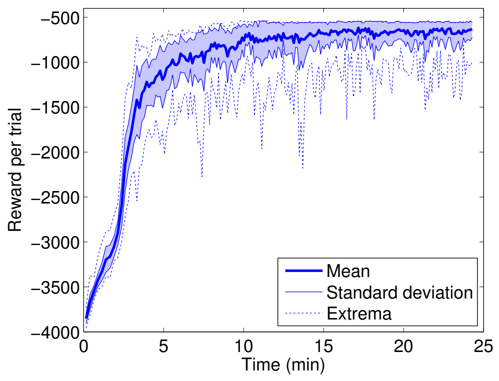
It is also interesting to look at the number of updates used by the best ($[0.2094, 2.0944]$) and worst ($[\infty, \infty]$) PS case (Table 4-2). We notice that the ratio's of the optimal (1:2.79) and worst (1:3) setting are actually very close together. Hence, the number of discarded state-transitions in the optimal case is not much higher. But apparently, the small amount of inaccurate estimates is enough to severely influence the learning process negatively.

If we compare the ratio of the model-generated versus model-free state-transitions of Dyna and PS we notice that this ratio is higher in PS. The priority queue leads to generated state-transitions that are close to the trajectory to the goal state. In these areas the number of memory samples is high and therefore accurate predictions can be made. Therefore, less model outputs are rejected than in the Dyna case where state-transitions are generated randomly in the entire state-space.

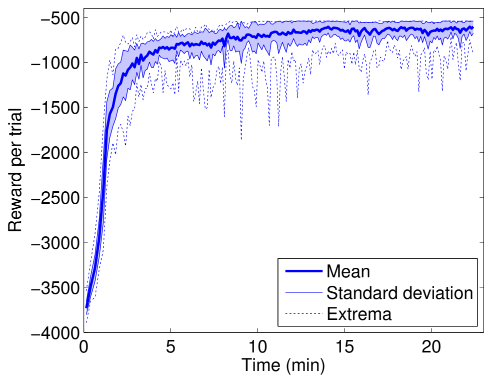
The best performance is obtained with the interval limit $[0.2094, 2.0944]$. Figure 4-10(b) shows the learning curve of the optimal PS algorithm compared to Dyna and SARSA. The figure shows that the PS algorithm learns faster than both Dyna and SARSA. Especially in the first minutes of learning, the prioritized updates of the value-function lead to faster learning than Dyna.



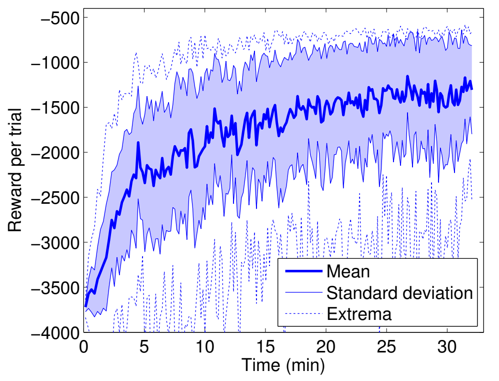
(a) $[\delta_\theta, \delta_\omega] = [0.001, 0.01]$



(b) $[\delta_\theta, \delta_\omega] = [0.01, 0.1]$

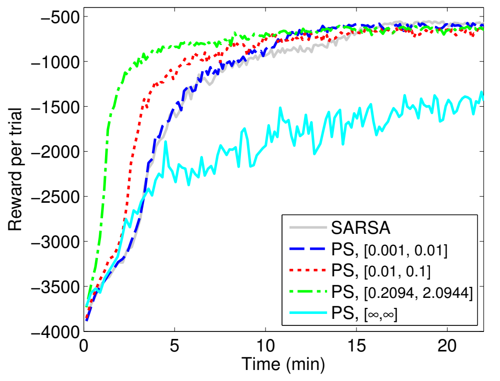


(c) $[\delta_\theta, \delta_\omega] = [0.2094, 2.0944]$

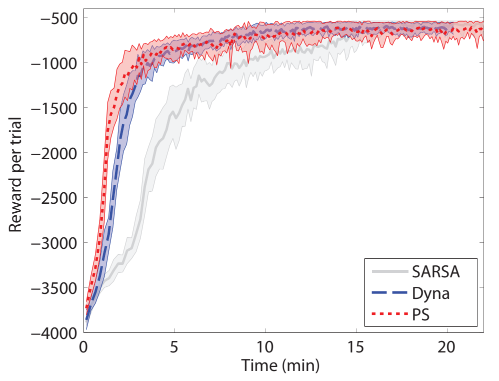


(d) $[\delta_\theta, \delta_\omega] = [\infty, \infty]$

Figure 4-9: Learning curves of PS applied on the inverted pendulum problem. (a), (b), (c) and (d) show the PS algorithm using LLR with increasing prediction interval limits.



(a) PS learning curves



(b) Comparison of PS, Dyna and SARSA

Figure 4-10: Learning curves for PS algorithm applied on the inverted pendulum setup. (a) compares PS using four different values for the prediction interval limit, (b) compares the optimal settings of PS, Dyna and SARSA.

4-4-4 Look Ahead Dyna

Let us consider PS simply as a method to generate specific model-based experiences instead of random experiences. We then might introduce other methods to generate useful state-transitions - not necessarily according to a priority measure. A possible method could be to use the model to 'look ahead' from the current state. Imagine a state-transition $s \rightarrow s'$. We then use the model to estimate $s' \rightarrow s''$ for all actions. The only issue is that the next state s' is not yet known when the model-based experiences are generated. Therefore, the next state has also got to be estimated. Algorithm 6 shows this method, which we will call Look Ahead Dyna (LA Dyna).

The Look Ahead approach partly originates from the earlier LLR experiments on the two-link manipulator and robot Leo (Sections 3-4-2 and 3-4-3). In those settings we used the LLR model to estimate a state-transition in the vicinity of the current state. LLR was able to generate very accurate estimates of complex systems using only a small number of experiences. We use this ability to model future state-transitions in this 'Look Ahead' setting.

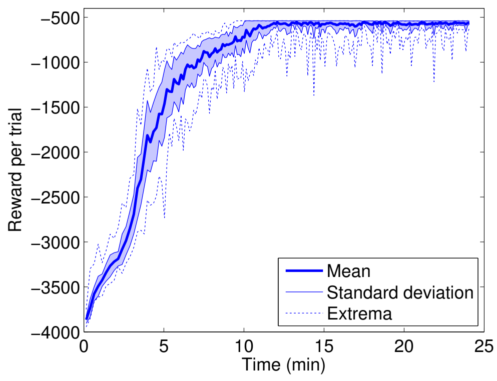
Algorithm 6 Look Ahead Dyna

```
1: Initialize $Q(s,a)$ randomly
2: for each time step $t$ do
3:     $s \leftarrow$ Current state
4:     Select $a$ using policy
5:     Execute $a$
6:     Estimate $\hat{s}' \leftarrow$ LLR $(s,a)$
7:     for each $a \in \mathcal{A}$ do
8:         $(\hat{s}'', r) \leftarrow Model(\hat{s}', a)$
9:         $\delta \leftarrow r + \gamma Q(\hat{s}'', a') - Q(\hat{s}', a)$
10:        $Q(s', a) \leftarrow Q(s', a) + \alpha\delta$
11:    end for
12:    Observe $s'$, $r$
13:    $\delta \leftarrow r + \gamma Q(s', a') - Q(s, a)$
14:    $Q(s, a) \leftarrow Q(s, a) + \alpha\delta$
15: end for
```

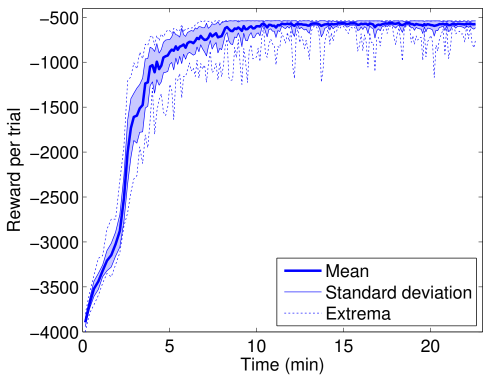
Results

We applied LA Dyna on the inverted pendulum simulation. Figure 4-11 shows the resulting learning curves for different values of the prediction interval limit. The prediction interval is needed in the LA Dyna algorithm. No limit on the prediction interval leads to severe 'unlearning', as can be seen in Figure 4-11(d). This effect was also seen in the case of the PS algorithm. Clearly, when the value function updates are close to the optimal trajectory, the estimated state-transitions need to be accurate.

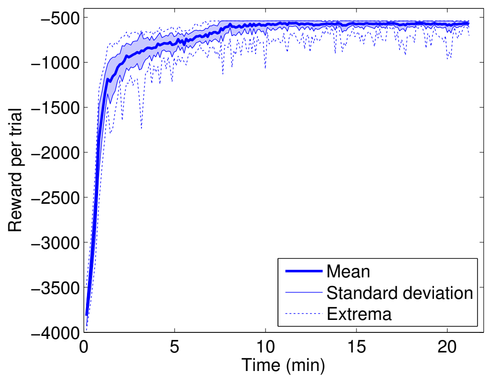
The optimal value for the prediction interval limit is $[0.2094, 2.0944]$, which is equal to the optimal value for Dyna and PS.



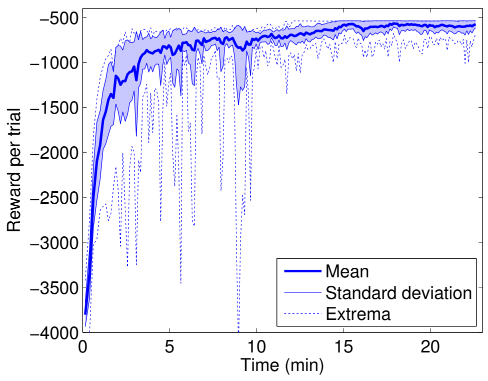
(a) $[\delta_\theta, \delta_\omega] = [0.001, 0.01]$



(b) $[\delta_\theta, \delta_\omega] = [0.01, 0.1]$

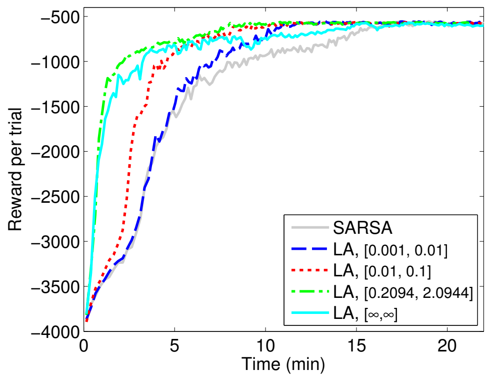


(c) $[\delta_\theta, \delta_\omega] = [0.2094, 2.0944]$

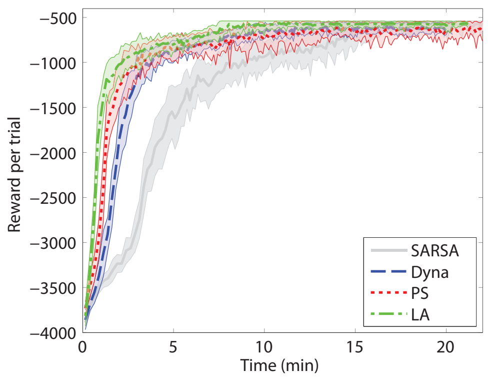


(d) $[\delta_\theta, \delta_\omega] = [\infty, \infty]$

Figure 4-11: Learning curves of the LA Dyna algorithm applied on the inverted pendulum. (a), (b), (c) and (d) show the LA Dyna algorithm using LLR with increasing prediction interval limits.



(a) LA Dyna learning curves

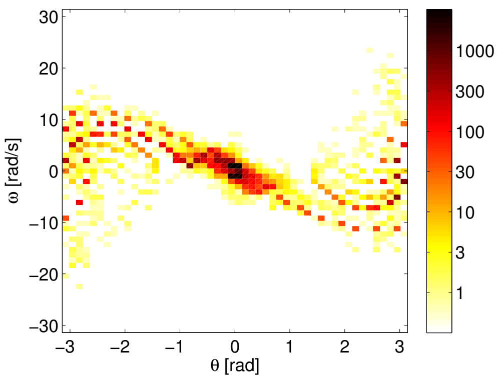


(b) Comparison of LA Dyna, PS, Dyna and SARSA

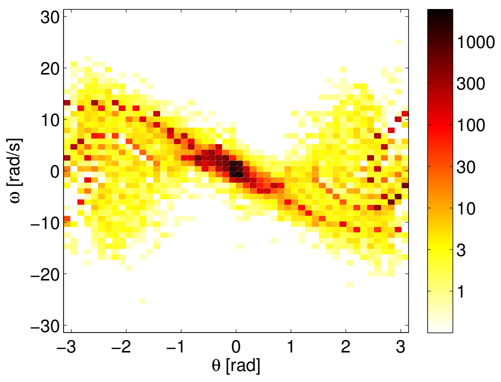
Figure 4-12: Learning curves for the LA Dyna algorithm applied on the the inverted pendulum setup. (a) compares LA Dyna using four different values for the prediction interval limit, (b) compares the optimal settings of the four different learning algorithms.

In Figure 4-12(b) we compared LA Dyna to SARSA, Dyna and PS. It can be seen that LA Dyna learns faster than the other algorithms. The faster learning can mainly be seen in the first part of the learning process. The LA Dyna algorithm also results in less spread in the learning speed. The variance of the LA Dyna is about half of the variance of the other algorithms.

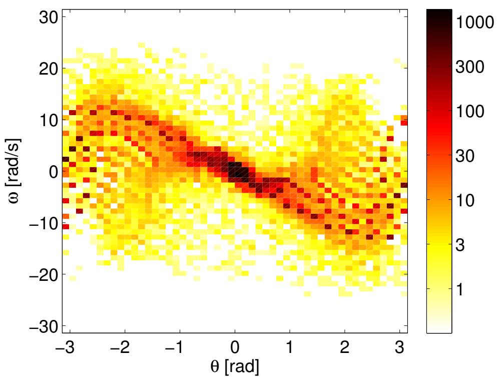
Figure 4-13 shows the distribution of the model-generated state-transitions. As expected, the state-transitions are located primarily in the neighborhood of the trajectories followed by the system. A possible reason for the fast learning of LA Dyna could be that the majority of the value function updates are very close to the current state of the real system.



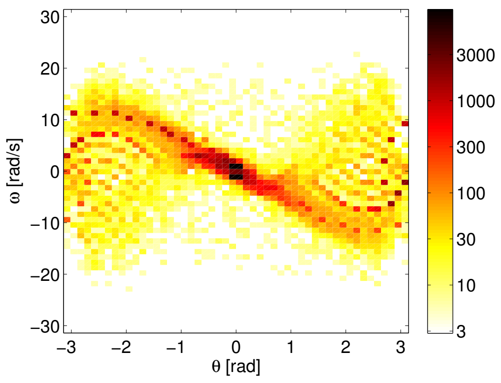
(a) $[\delta_\theta, \delta_\omega] = [0.001, 0.01]$



(b) $[\delta_\theta, \delta_\omega] = [0.01, 0.1]$



(c) $[\delta_\theta, \delta_\omega] = [0.2094, 2.0944]$



(d) $[\delta_\theta, \delta_\omega] = [\infty, \infty]$

Figure 4-13: Distribution of states for which a state-transition was generated using the LLR model in the LA Dyna setting. The figures only show the states for which the prediction interval was smaller than the given limit. The color indicates the number of times a state-transition was generated in a certain area.

4-5 Conclusions

In this chapter we have combined model-learning with Reinforcement Learning to speed up the learning process. We built upon the experience and insight of Chapter 3. Hence, we used Local Linear Regression as a model. Every experiment was started with an empty memory, which was filled with observed state-transitions during learning. We used an inverted pendulum setup in simulation as experimental setup. A swing-up control task was used to compare different learning algorithms.

Prediction intervals were used to assess the quality of a modeled state-transition. Whenever the prediction interval of a generated transition is larger than a certain limit, the model output is discarded. We compared four different interval limits, loosely based on the resolution of the value function approximator.

We compared model-free SARSA learning to model-learning Dyna. The Dyna algorithm used state-transitions that were generated randomly throughout the state-space. We showed that the Dyna algorithm performs at least as good as pure SARSA. The Dyna algorithm performed better as more model-generated state-transitions were added. This increased the learning speed and decreased the spread in the curves. Limiting the prediction interval had a negative influence for this setting, as it reduced the number of state-transitions available to the agent. For Dyna the optimal setting would be to use a very loose limit or no limit on the prediction interval at all.

Prioritized Sweeping was combined with LLR to determine lead-in state-action pairs. We showed that the limit on the prediction interval is vital for the performance of the PS algorithm. Imposing no limit led to very bad performance. In fact, the learning rate was even slower than for SARSA and the algorithm did not converge during a 1000-trial experiment using $[\delta_\theta, \delta_\omega] = [\infty, \infty]$. From the set of four compared prediction intervals, the optimal setting for PS was a limit of $[0.2094, 2.0944]$. This led to faster learning than the optimal Dyna setting.

Finally, we introduced Look Ahead Dyna as a learning algorithm. This method uses the LLR model to look ahead from the current state. In this way the effect of actions in future states can be assessed. The LA Dyna method showed faster learning than the PS algorithm. The high number of value-function updates close to the optimal trajectory, makes this method learn fast.

Chapter 5

Conclusions

Reinforcement Learning (RL) covers a wide variety of methods, ranging from off-line model-based to on-line model-free methods. In this work we focused on model-learning methods that build a model of the environment during the learning process. The implemented algorithms are Dyna-style methods, which interact with the model and the actual system simultaneously. The research was divided into two parts. First we investigated how a state-transition model could be built during learning. Thereafter we added the model to the learning process and investigated how the model could be implemented, such that the learning process benefits optimally from the state-transitions generated by the model.

5-1 Modeling

We used the memory-based Local Linear Regression (LLR) method to build a model. The memory consists of the observed state-transitions and can be updated continuously during the learning process. Memory-based methods have advantages that are useful in RL. It is very easy to improve the model during learning, as this simply means adding new experiences to the memory. Furthermore, results from statistical analysis can be used to assess the quality of a particular estimated state-transition. Two limitations of memory-based modeling are noise sensitivity and computational effort needed for a query. The first is counteracted by using linear regression, which estimates a linear function through the set of nearest neighbors. The computational effort is decreased by storing the memory samples using a kd-tree.

We showed that the LLR model is able to estimate state-transitions from relatively few observations. LLR was able to model the walking motion of a complex, humanoid robot setup using a memory of a few thousand samples. This shows that also high-dimensional setups can be estimated using LLR.

For use in a RL setting, it is desired to have a measure of the quality of the estimated state-transition. This measure should not be based on the real value, as this might not be available in a RL setting. We proposed the statistical notion of prediction intervals as a possible measure for the model accuracy. This measure is based on the residuals of the estimated model and only depends on the memory samples used in the linear regression.

5-2 Reinforcement Learning

We performed learning experiments on an inverted pendulum simulation that learned a swing-up task. The learning experiments were performed using model-learning algorithms, combining interaction with the real environment and the built model. We used LLR as a model, using the observed state-transitions as memory samples.

We showed that Dyna-style learning increases the learning speed, compared to model-free SARSA. However, the final performance of the Dyna algorithm is slightly worse than SARSA. This is probably due to the inaccurate state-transitions introduced by the model. For Dyna with randomly selected model-based updates, using many uncertain state-transitions resulted in faster learning than using few accurate state-transitions. For random Dyna, ignoring the prediction intervals made learning faster.

To generate specific model-based state-transitions instead of randomly chosen state-action pairs as in Dyna, two methods were introduced. Both Prioritized Sweeping (PS) and Look Ahead Dyna (LA Dyna) led to faster learning than random Dyna. As opposed to random Dyna, these methods needed a limit on the prediction interval. Without such a limit, severe 'unlearning' occurred and convergence to an optimal trajectory towards the goal was not always achieved. However, choosing this limit too tight resulted in slow learning. Out of a set of four compared prediction interval limits, the optimal value for PS and LA Dyna resulted in discarding 7% and 4% of the estimated state-transitions repectively.

5-3 Discussion and future work

In Chapter 3 we suggested local optimization of the number of nearest neighbors and outlier detection as methods to improve the quality of the model locally. In our learning experiments these methods were not used, primarily because they consume extra computation time. However, local optimization of K might be usable if sufficient computational power is available and outlier detection might be needed when LLR is used on real setups.

If a model is used to generate state-transitions, the model should be accurate enough to not disturb the learning process. In this work we have used the statistical notion of prediction intervals to assess the quality of the modeled state-transitions. We showed that in the case of PS and LA Dyna it is indeed needed to limit the prediction interval.

In random Dyna this was not needed. Further research is needed to better understand how the quality of the model affects the learning process.

We have briefly addressed the question how the limit of the prediction interval should be chosen. We have connected the prediction interval limit to the discriminating power of the value function approximator. It is not fully understood how the quality of the model and the resolution of the function approximator relate to each other. It might be true that a very accurate model is not needed if the resolution of the function approximator is relatively coarse.

The RL experiments in Chapter 4 were performed using an inverted pendulum simulation. The LLR model and the different algorithms were all applied in this experimental setting. For future research, more complex setups and real-time applications could be considered. Such an application could be the humanoid robot setup (see Section 3-4-3). The modeling results with this setup indicate that LLR is able to model such a system. Furthermore, all algorithms should be fast enough to be used in real-time applications, given that the sampling frequency is not too high. Therefore, learning experiments with the humanoid robot setup could also benefit from the methods introduced in this work.

We showed that PS and LA Dyna learn faster than Dyna with randomly selected model-based updates. PS and LA Dyna use different methods for selecting state-transitions. PS looks backwards, while LA Dyna looks ahead from the current state. In order to fully understand why these methods work and whether this is restricted to certain learning problems, further research is needed. Also different methods for selecting modeled state-transitions could be compared.

One of the major issues with PS is how to determine lead-in states. Our method calculates a lead-in state for every available action separately. This method becomes infeasible with a large or continuous action-space. It is not clear how lead-in states could be determined more easily.

5-4 Final conclusions

In the introduction of this thesis we formulated the research question, which can now be answered. The memory-based modeling method LLR showed very good modeling capabilities and can be implemented easily in a RL algorithm. LLR was able to model a complex humanoid robot setup surprisingly accurate using only a few thousand memory samples. The model can be used to generate state-transitions that can be used by the learning algorithm as if they were real observations. The learning speed is influenced by the position in state-space where the state-transitions are generated. PS and LA Dyna try to concentrate these state-transitions in the neighborhood of the trajectory towards the goal, which resulted in faster learning.

Our learning experiments were performed on a relatively simple system in simulation. However we are confident that the methods described in this thesis can also be used on more complex, real setups. The modeling capability of LLR and the computational speed of the algorithms are both sufficient to be applied to such systems.

Appendix A

LLR statistics

In this appendix we derive several expression for the statistics used with LLR. The results are valid for the set of K nearest neighbors used in the regression. These expressions are commonly used in statistics. For a more elaborate explanation, see e.g. [19].

A-1 Least squares solution

We estimate the following linear model:

$$\mathbf{Y} = \mathbf{X}\boldsymbol{\beta} + \boldsymbol{\epsilon} \tag{A-1}$$

With the following properties:

1. $E[\boldsymbol{\epsilon}] = \mathbf{0}$, hence $E[\mathbf{y}] = E\left[\mathbf{x}^T\boldsymbol{\beta}\right]$
2. $\text{cov}(\boldsymbol{\epsilon}) = \sigma^2\boldsymbol{I}$, hence $\text{cov}(\mathbf{y}) = \sigma^2\boldsymbol{I}$

The estimated model $\hat{\boldsymbol{\beta}}$ minimizes the squared error:

$$\hat{\mathbf{e}}^T\hat{\mathbf{e}} = \left(\boldsymbol{Y} - \boldsymbol{X}\hat{\boldsymbol{\beta}}\right)^T\left(\boldsymbol{Y} - \boldsymbol{X}\hat{\boldsymbol{\beta}}\right)$$

The solution is obtained by differentiating with respect to $\hat{\boldsymbol{\beta}}$ and setting the result equal to zero. This results in the well-known normal equations:

$$\boldsymbol{X}^T\boldsymbol{X}\hat{\boldsymbol{\beta}} = \boldsymbol{X}^T\mathbf{Y}$$

which leads to the solution:

$$\hat{\boldsymbol{\beta}} = \left(\boldsymbol{X}^T\boldsymbol{X}\right)^{-1}\boldsymbol{X}^T\boldsymbol{Y}$$

The obtained $\hat{\boldsymbol{\beta}}$ is the least squares solution of Eq. (A-1) and is an unbiased estimator of $\boldsymbol{\beta}$ (if $E[\mathbf{Y}] = \mathbf{X}\boldsymbol{\beta}$ holds).

We will now introduce some properties of linear regression that will be used in assessing the quality of an estimated model. First, we introduce the residual vector, which is an estimation of the noise:

$$\hat{\boldsymbol{\epsilon}} = \boldsymbol{Y} - \hat{\boldsymbol{Y}} = \boldsymbol{Y} - \boldsymbol{X}\hat{\boldsymbol{\beta}}$$

Using the expression for $\hat{\boldsymbol{\beta}}$, we obtain:

$$\hat{\boldsymbol{\epsilon}} = \boldsymbol{Y} - \boldsymbol{X}\left(\boldsymbol{X}^T\boldsymbol{X}\right)^{-1}\boldsymbol{X}^T\boldsymbol{Y}$$

$$\hat{\boldsymbol{\epsilon}} = \left(\boldsymbol{I} - \boldsymbol{X}\left(\boldsymbol{X}^T\boldsymbol{X}\right)^{-1}\boldsymbol{X}^T\right)\boldsymbol{Y}$$

By definition:

$$\mathrm{var}(\mathbf{y}_i) = \sigma^2 = E\left[\mathbf{y}_i - E[\mathbf{y}_i]\right]^2$$

Using $E[\mathbf{y}_i] = \mathbf{x}_i^T\boldsymbol{\beta}$, we get:

$$\sigma^2 = E\left[\mathbf{y}_i - \mathbf{x}_i^T\boldsymbol{\beta}\right]^2$$

Assuming the variance of the noise is constant, an unbiased estimator of the variance can be obtained by calculating the average value of the variance of the dataset. This estimator for σ^2 is s^2:

$$s^2 = \frac{1}{K - d_x}\sum_{i=1}^{K}\left(\mathbf{y}_i - \mathbf{x}_i^T\boldsymbol{\beta}\right)^T\left(\mathbf{y}_i - \mathbf{x}_i^T\boldsymbol{\beta}\right)$$

or:

$$s^2 = \frac{1}{K - d_x}\left(Y - \boldsymbol{X}\hat{\beta}\right)^T\left(Y - \boldsymbol{X}\hat{\beta}\right) \tag{A-2}$$

With $K - d_x$ the number of free parameters.

A-2 Prediction interval

The variance of the estimation error is given by:

$$\begin{aligned}
\mathrm{var}(e) = \mathrm{var}(\mathbf{y}_q - \hat{\mathbf{y}}_q) &= \mathrm{var}(\mathbf{y}_q) + \mathrm{var}(\hat{\mathbf{y}}_q)\\
&= \mathrm{var}(\mathbf{x}_q^T\boldsymbol{\beta} + \boldsymbol{\epsilon}) + \mathrm{var}(\mathbf{x}_q^T\hat{\boldsymbol{\beta}})\\
&= \mathrm{var}(\boldsymbol{\epsilon}) + \mathrm{var}(\mathbf{x}_q^T\hat{\boldsymbol{\beta}})\\
&= \sigma^2 + \sigma^2\mathbf{x}_q^T\left(\boldsymbol{X}^T\boldsymbol{X}\right)^{-1}\mathbf{x}_q\\
&= \sigma^2\left[1 + \mathbf{x}_q^T\left(\boldsymbol{X}^T\boldsymbol{X}\right)^{-1}\mathbf{x}_q\right]
\end{aligned}$$

Which follows from the assumptions made on the system equations. The noise variance σ^2 can be estimated by s^2. Using $E\left[\mathbf{y}_q - \hat{\mathbf{y}}_q\right] = 0$ and the fact that s^2 is independent of both $\mathbf{y}_q$ and $\hat{\mathbf{y}}_q$, it can be proved that the t-statistic

$$t = \frac{\mathbf{y}_q - \hat{\mathbf{y}}_q}{s\sqrt{1 + \mathbf{x}_q^T\left(\boldsymbol{X}^T\mathbf{X}\right)^{-1}\mathbf{x}_q}}$$

has a $t(K-d_x)$ distribution. Using the probability density function of the t-distribution, we can give a formula for the probability that the the t-statistic is within a certain interval:

$$P\left[-t_{\alpha/2,K-d_x} \leq \frac{\mathbf{y}_q - \hat{\mathbf{y}}_q}{s\sqrt{1+\mathbf{x}_q^T\left(\boldsymbol{X}^T\boldsymbol{X}\right)^{-1}\mathbf{x}_q}} \leq t_{\alpha/2,K-d_x}\right] = 1-\alpha$$

This inequality can be solved to obtain the $100(1-\alpha)\%$ prediction interval:

$$\hat{\mathbf{y}}_q - t_{\alpha/2,K-d_x} s\sqrt{1+\mathbf{x}_q^T\left(\boldsymbol{X}^T\boldsymbol{X}\right)^{-1}\mathbf{x}_q} \leq \mathbf{y}_q \leq \hat{\mathbf{y}}_q + t_{\alpha/2,K-d_x} s\sqrt{1+\mathbf{x}_q^T\left(\boldsymbol{X}^T\boldsymbol{X}\right)^{-1}\mathbf{x}_q}$$

or in a more compact form:

$$\begin{aligned}\mathbf{y}_q &= \hat{\mathbf{y}}_q \pm t_{\alpha/2,K-d_x} s\sqrt{1+\mathbf{x}_q^T\left(\boldsymbol{X}^T\boldsymbol{X}\right)^{-1}\mathbf{x}_q}\\ &= \hat{\mathbf{y}}_q \pm I\end{aligned}$$

with I the calculated prediction interval for the estimate $\hat{\mathbf{y}}_q$ of the true value $\mathbf{y}_q$.

Appendix B

Humanoid robot: LLR model

B-1 LLR estimate of all state-variables

In section 3-4-3 we have showed how Local Linear Regression (LLR) was used to estimate the walking motion of a humanoid robot. For clarity we showed only three representative state-variables. The LLR estimates of all 18 state-variables are shown in Figure B-1 and B-2 of this Appendix. The model was estimated using a large estimation dataset ($N = 8000$) as memory. The state-transitions were estimated using $K = 40$ nearest neighbors. As before, we only show the estimate of one stride (two steps). It is clear that the LLR model is able to model all 18 state-variables very accurately.

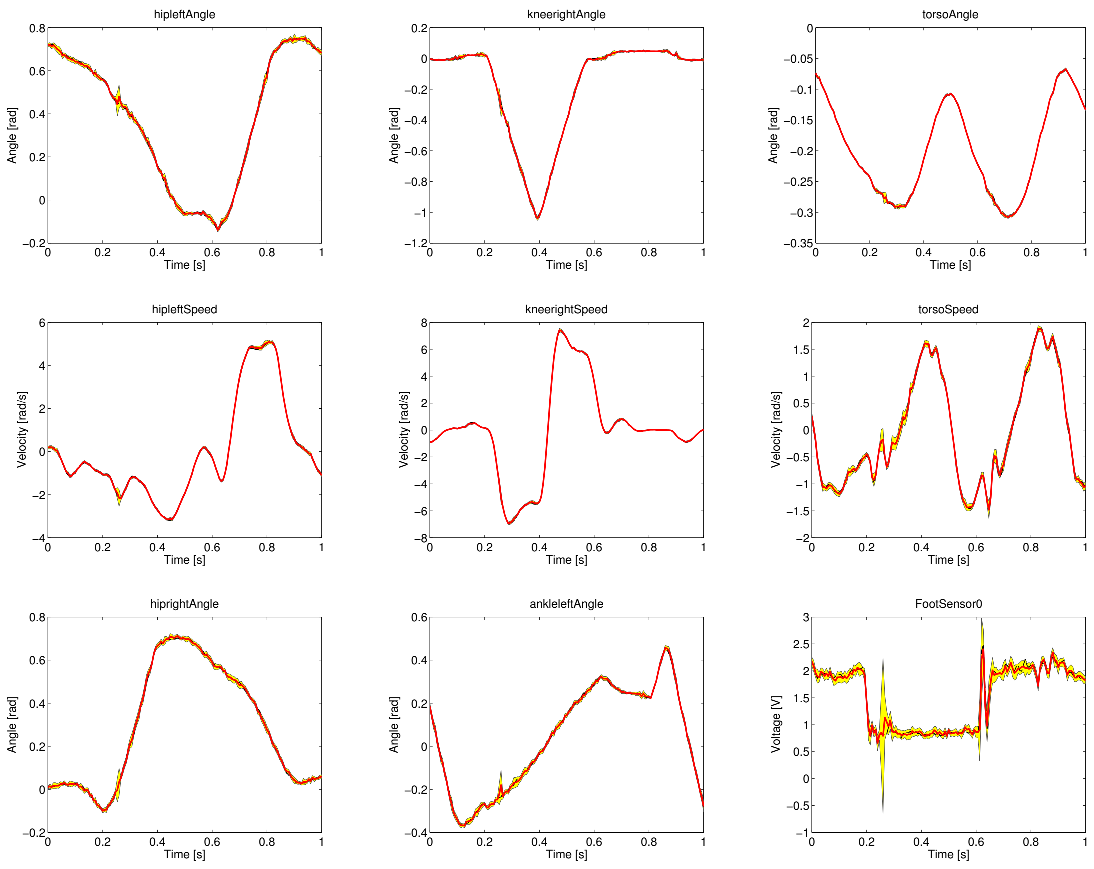


Figure B-1: LLR estimate $(K = 40)$ of the walking motion of robot Leo using a memory consisting of 8000 samples. The figures show 9 different state-variables. The figures show the LLR estimate (solid red line), the measured output (dashed black line) and the prediction interval (shaded area).

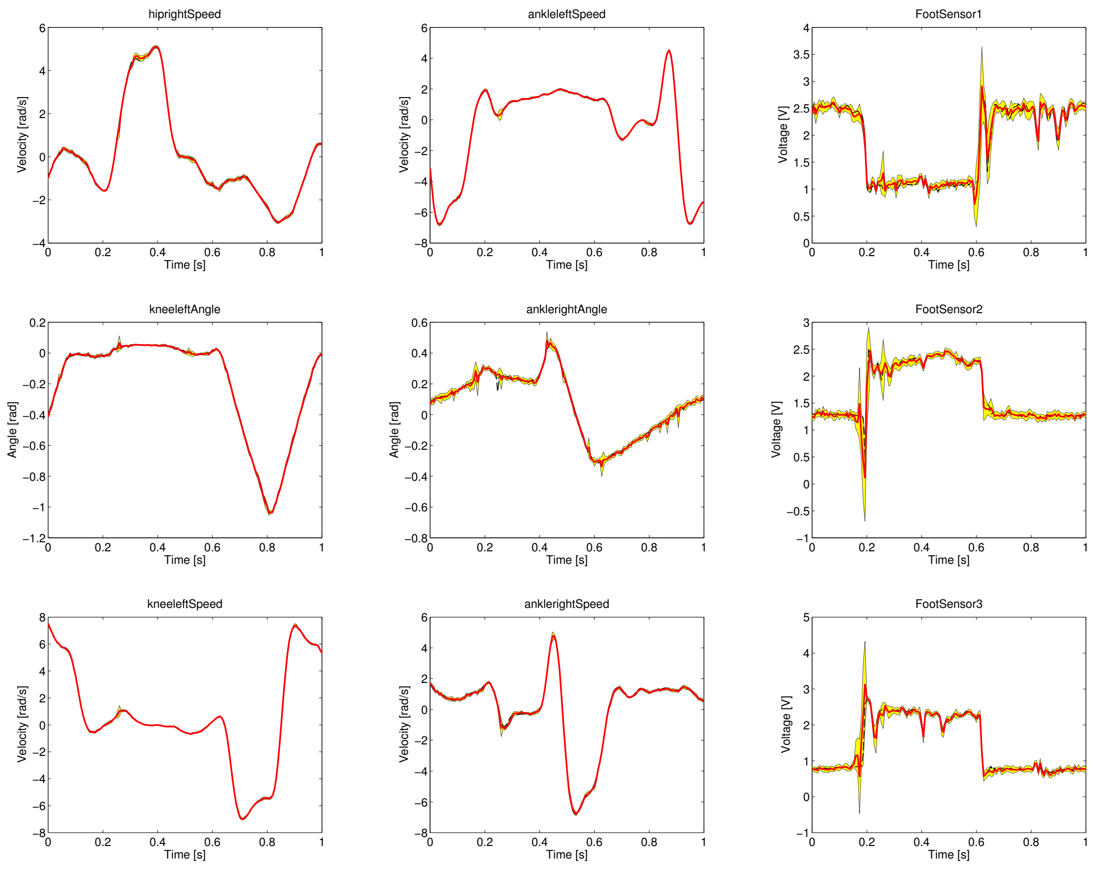


Figure B-2: LLR estimate $(K = 40)$ of the walking motion of robot Leo using a memory consisting of 8000 samples. The figures show 9 different state-variables. The figures show the LLR estimate (solid red line), the measured output (dashed black line) and the prediction interval (shaded area).

Appendix C

Q-iteration

C-1 Introduction

This Appendix shows the results of the model-based approach to solve a Reinforcement Learning (RL) problem using Local Linear Regression (LLR) as a model. This approach was tried for two reasons. First, we wanted to use the model-based policies as a benchmark result to which the model-free and model-learning methods could be compared. Furthermore, we searched for a way to compare LLR to the true system. Since iterative methods do not use exploration, the results are always the same and therefore easy to compare. In this way it would be easy to compare for instance the number of memory samples on the quality of the model and the resulting policy.

C-2 Experimental setup

The results are obtained using Q-iteration in a learning problem using the two-link manipulator (Section 3-4-2). Q-iteration (Section 2-3-1) is a model-based solution method that iterates over all states and actions to find a policy for the entire state-action space. The learning goal was to move the manipulator to 0. The reward was +10 for goal states and -1 elsewhere. We compared the results from the nonlinear model to the LLR model of the system. The results were obtained using Q-iteration and Fuzzy approximation of the value function [20]. The LLR model was obtained by using a memory consisting of $7.5 \cdot 10^4$ samples, using $K = 5$ nearest neighbors. These samples were distributed evenly over the state-space and were generated using the nonlinear model.

Performance measure Quality of an obtained policy and thus the performance of the model was assessed in two ways:

1. **Visual inspection of the policy:** The policy obtained for the noiseless case (Figure C-1(a)) is taken as the 'true' policy. The noisy and LLR policies should look similar to the true policy.
2. **Discounted reward J:** A set of 100 initial states is generated evenly over the state-space and the obtained policy is used to simulate trajectories starting from these initial states. The mean of the discounted rewards obtained in these trajectories is a measure for how good the obtained policy is (higher reward equals better policy).

$$J = \frac{1}{100}\sum_{i=1}^{100} J_i$$

C-3 Results

Figure C-1 shows the results obtained for Q-iteration applied on the two-link manipulator setup. The figures show the policy for the torques τ_1 and τ_2 in the two joints for $\omega_1 = 0$ and $\omega_2 = 0$.

The optimal result is the nonlinear model without noise. This result is shown in Figure C-1(a). The policy shows no artifacts and the performance of this policy is $J = -85.5$. The policy of the LLR model for the noise-free case is shown in Figure C-1(b). The policy still looks reasonably good, but the performance $J = -104.8$ is much worse than for the nonlinear model.

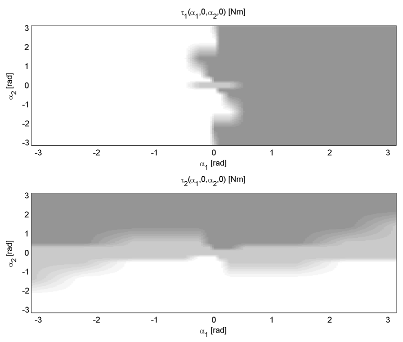
For both cases the policy becomes worse when a white noise signal is added to the output of the model. The nonlinear model still performs reasonably well for a noise signal with $\sigma^2 = 0.01$, but deteriorates for increasing noise. The LLR model also decreases in performance for increasing noise. In theory, the LLR model could decrease the effect of noise. However, this is not supported by these results as the LLR model always performs worse than the nonlinear model.

Several more experiments were carried out (not shown here) with various settings. The overall conclusions are the same for all cases: Adding noise quickly deteriorates the quality of the resulting policy and LLR never leads to better results than the nonlinear model. Even a very large memory consisting of $3 \cdot 10^5$ samples, did not lead to good results. This surprising as the earlier modeling experiments 3-4-2 showed very good modeling accuracy for only a few thousand samples.

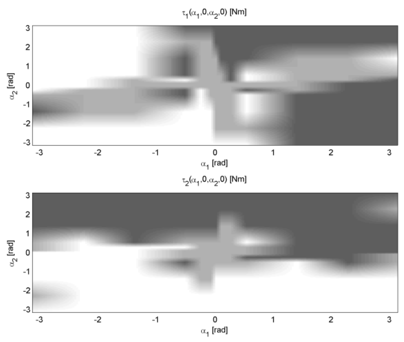
C-4 Conclusion & discussion

We used Q-iteration to obtain results that could easily be compared for different parameters and could be used as benchmarks for the on-line learning learning methods. However, the results are not satisfactory. The results obtained with the LLR model are far worse than those obtained with the nonlinear model. Also the number of memory samples needed was very high. Much higher than when LLR was used to estimate trajectories.

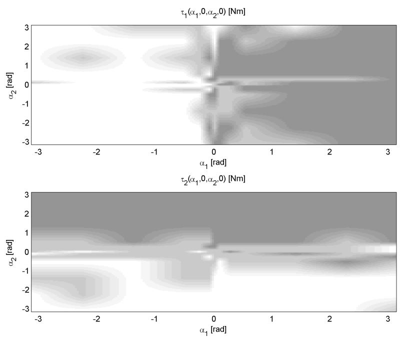
Although Q-iteration has advantages over an on-line learning method because it limits the number of tuning/learning parameters, it might not be the best setting to test LLR. Especially in view of the final application of the model: application of LLR in a Dyna/Prioritized Sweeping (PS) setting. The model will not be used to model the entire state-space, but only certain parts (the parts that have been visited by the agent and trajectories that lead to high rewards). In Q-iteration one assumes complete knowledge of the entire state-space.



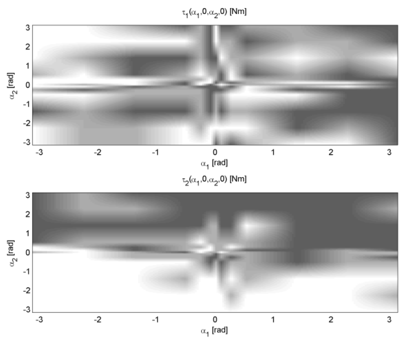
(a) nonlinear model, no noise, $J = -85.5$



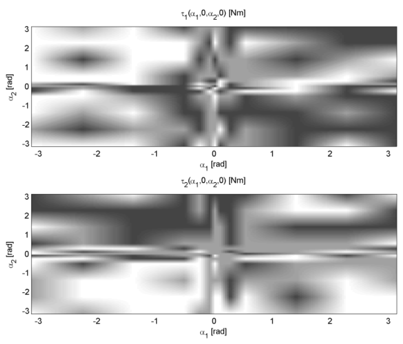
(b) LLR model, no noise, $J = -104.8$



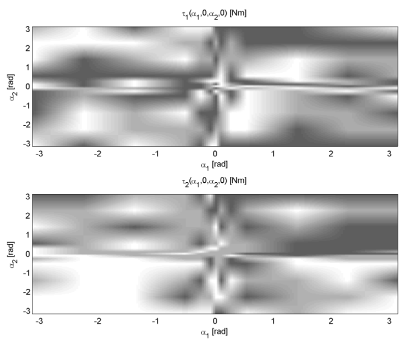
(c) nonlinear model, noise ($\sigma^2 = 0.01$), $J = -96.4$



(d) LLR model, noise ($\sigma^2 = 0.01$), $J = -133.5$



(e) nonlinear model, noise ($\sigma^2 = 0.05$), $J = -149.6$



(f) LLR model, noise ($\sigma^2 = 0.05$, $J = -202.6$

Figure C-1: Q-iteration policies for several settings with increasing noise. C-1(a), C-1(c), C-1(e) are the results for the nonlinear model; C-1(b), C-1(d), C-1(f) are for the LLR estimate. The white, gray and black areas correspond to control voltages of +3V, 0 and -3V respectively.

Appendix D

Learned policies

In Chapter 2-3, we mentioned that 'solving' in a Reinforcement Learning (RL) context means finding an optimal policy. An optimal policy assigns an action to every state. However, our learning experiments did not lead to a (converged) policy for the entire state-space. We were only interested in finding (sub-)optimal trajectories towards the goal. This is true for most RL experiments. Convergence to a solution in the entire state-space is almost never possible for model-free or model-learning methods as parts of the state-space will never or only rarely be visited by the system. In practice however, it is often not needed to converge to a solution in the entire state-space as we only need a control policy along interesting trajectories (towards the goal).

In this Appendix we show the final polices of the different algorithms that we have used in our research.

D-1 Converged policy

In order to get insight in how the optimal policy looks, we executed a (random) Dyna experiment using the exact model. We executed 10000 trials and used the exact model to generate $N_m = 10$ state-transitions per time step. These settings should lead to a converged solution throughout the entire state-space.

Figure D-1(a) shows the resulting policy. Although some minor artifacts are still visible, the obtained policy will be used as (an approximation of) the fully converged optimal policy.

D-2 SARSA policy

The obtained policy using SARSA is shown in Figure D-1(b). The learning results showed that the reward per trial converged to a final value. The policy however, has not converged in the entire state-space. The regions that show a converged policy, are the parts that were visited frequently by the system.

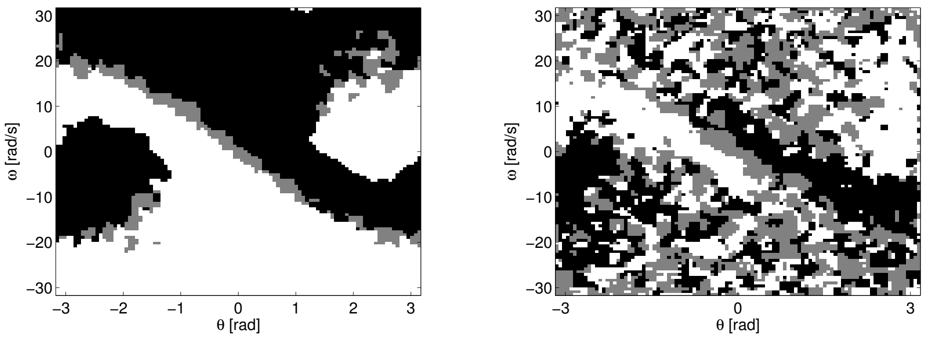
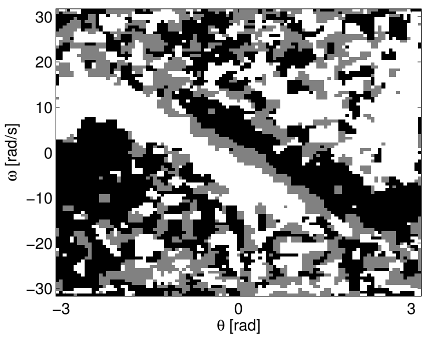


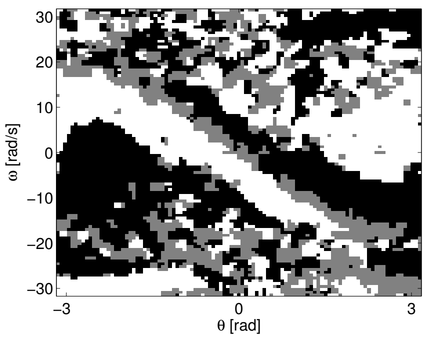
Figure D-1: Comparison of the optimal policy to the obtained policy using SARSA. (a) shows the optimal policy obtained using Dyna and the exact model, (b) shows the final SARSA policy. The white, gray and black areas correspond to control voltages of +3V, 0 and -3V respectively.

D-3 Dyna policies

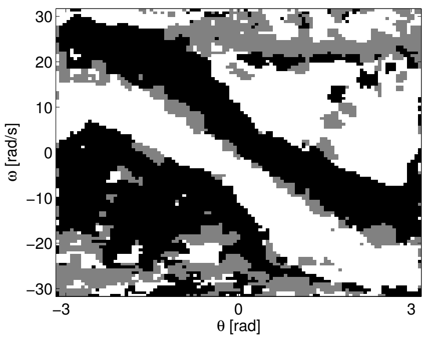
The obtained policies using Dyna are shown in Figure D-2. The policies are converged in a larger part of the state-space than the SARSA algorithm. This is due to the randomly selected model-based updates. This leads to learning in a larger part of the state-space than with SARSA.



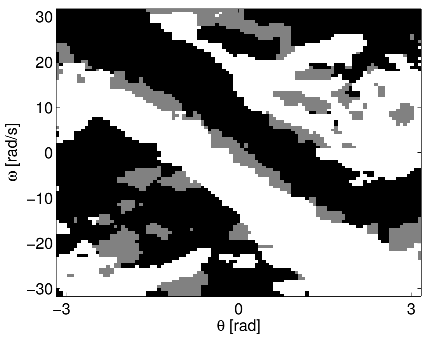
(a) $[\delta_\theta, \delta_\omega] = [0.001, 0.01]$



(b) $[\delta_\theta, \delta_\omega] = [0.01, 0.1]$



(c) $[\delta_\theta, \delta_\omega] = [0.2094, 2.0944]$

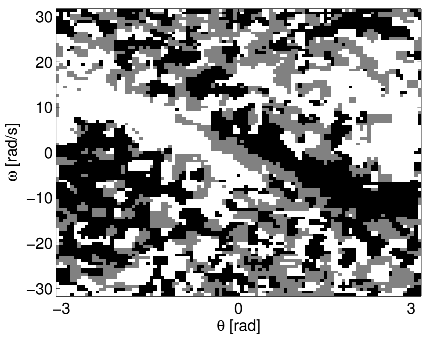


(d) $[\delta_\theta, \delta_\omega] = [\infty, \infty]$

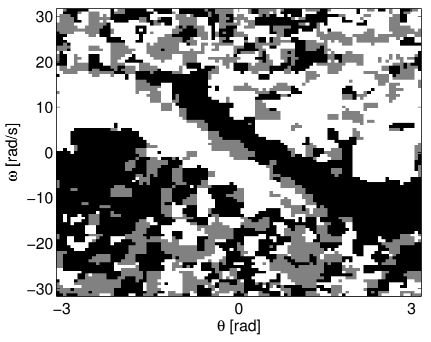
Figure D-2: Comparison of the final policies using Dyna. (a), (b), (c) and (d) show the obtained policies using LLR with increasing prediction interval limits. The white, gray and black areas correspond to control voltages of +3V, 0 and -3V respectively.

D-4 Prioritized Sweeping policies

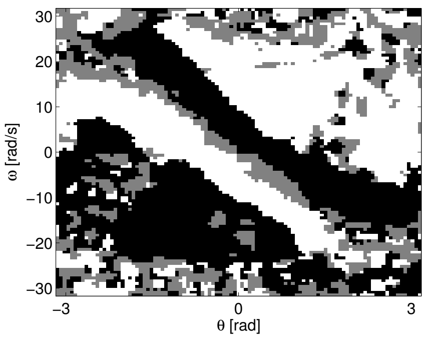
The obtained policies using Prioritized Sweeping (PS) are shown in Figure D-3. Although PS resulted in faster learning than Dyna, the resulting policies look worse than in the Dyna setting. This indicates that the resulting trajectory towards the goal may be found fast, but this does not mean that a policy has been found for the entire state-space.



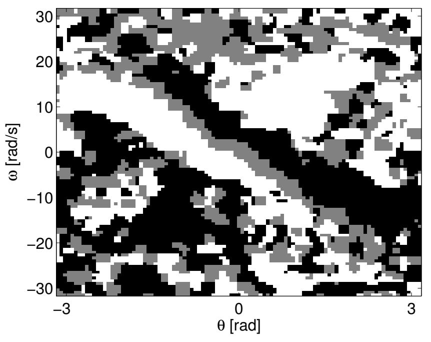
(a) $[\delta_\theta, \delta_\omega] = [0.001, 0.01]$



(b) $[\delta_\theta, \delta_\omega] = [0.01, 0.1]$



(c) $[\delta_\theta, \delta_\omega] = [0.2094, 2.0944]$

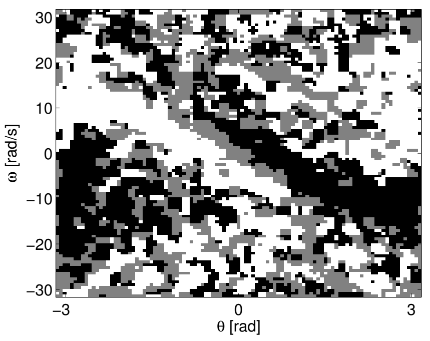


(d) $[\delta_\theta, \delta_\omega] = [\infty, \infty]$

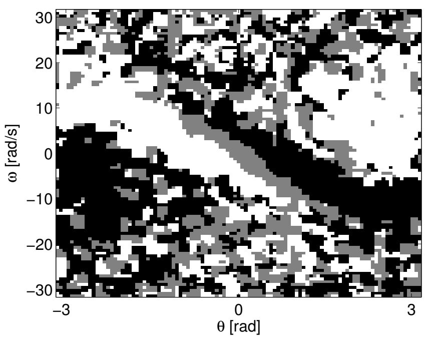
Figure D-3: Comparison of the final policies using PS. (a), (b), (c) and (d) show the obtained policies using LLR with increasing prediction interval limits. The white, gray and black areas correspond to control voltages of $+3$V, 0 and -3V respectively.

D-5 Look Ahead Dyna policies

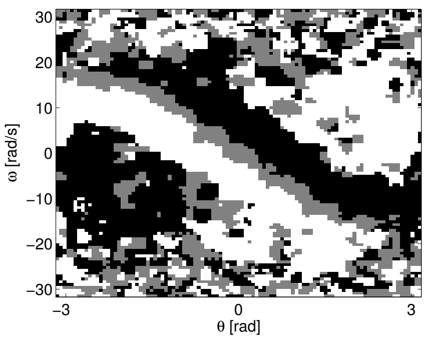
The obtained policies using Look Ahead Dyna (LA Dyna) are shown in Figure D-4. The policies have about the same quality as the for the PS case. Again, faster learning (compared to Dyna) does not result in a better final policy.



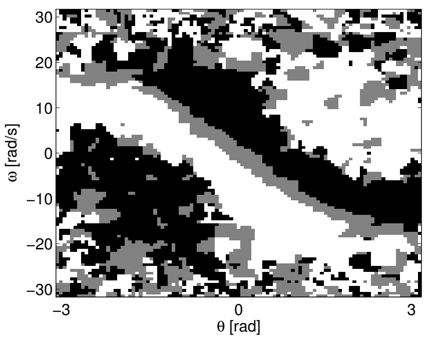
(a) $[\delta_\theta, \delta_\omega] = [0.001, 0.01]$



(b) $[\delta_\theta, \delta_\omega] = [0.01, 0.1]$



(c) $[\delta_\theta, \delta_\omega] = [0.2094, 2.0944]$



(d) $[\delta_\theta, \delta_\omega] = [\infty, \infty]$

Figure D-4: Comparison of the final policies using LA Dyna. (a), (b), (c) and (d) show the obtained policies using LLR with increasing prediction interval limits. The white, gray and black areas correspond to control voltages of +3V, 0 and -3V respectively.

References

[1] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction.* Cambridge, US: MIT Press, 1998.

[2] R. Bellman, *Dynamic programming.* Princeton University Press, Princeton, 1957.

[3] G. A. Rummery and M. Niranjan, "On-line q-learning using connectionist systems," tech. rep., 1994.

[4] C. J. C. H. Watkins and P. Dayan, "Q-learning," *Machine Learning*, vol. 8, pp. 279–292, May 1992.

[5] G. J. Gordon, "Stable function approximation in dynamic programming," 1995.

[6] R. S. Sutton, "Integrated architectures for learning, planning, and reacting based on approximating dynamic programming," in *Proceedings Seventh International Conference on Machine Learning (ICML-90)*, (Austin, Texas, US), pp. 216–224, June 21–23 1990.

[7] A. W. Moore and C. G. Atkeson, "Prioritized sweeping: Reinforcement learning with less data and less time," *Machine Learning*, vol. 13, pp. 103–130, 1993.

[8] J. Peng and R. J. Williams, "Efficient learning and planning within the dyna framework," in *Adaptive Behavior*, pp. 437–454, 1993.

[9] C. D. Rayner, K. Davison, V. Bulitko, K. Anderson, and J. Lu, "Real-time heuristic search with a priority queue," pp. 2372 – 2377, 2007.

[10] D. Wingate and K. D. Seppi, "Prioritization methods for accelerating mdp solvers," *J. Mach. Learn. Res.*, vol. 6, pp. 851–881, 2005.

[11] M. Grześ and D. Kudenko, "An empirical analysis of the impact of prioritised sweeping on the dynaq's performance," in *ICAISC '08: Proceedings of the 9th international conference on Artificial Intelligence and Soft Computing*, pp. 1041–1051, 2008.

[12] A. Y. Ng, H. J. Kim, M. I. Jordan, and S. Sastry, "Autonomous helicopter flight via reinforcement learning," in *International Symposium on Experimental Robotics*, MIT Press, 2004.

[13] B. Bakker, V. Zhumatiy, G. Gruener, and J. Schmidhuber, "A robot that reinforcement-learns to identify and memorize important previous observations," tech. rep., In Proceedings of the 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems, 2003.

[14] C. G. Atkeson and S. Schaal, "Robot learning from demonstration," in *ICML '97: Proceedings of the Fourteenth International Conference on Machine Learning*, (San Francisco, CA, USA), pp. 12–20, Morgan Kaufmann Publishers Inc., 1997.

[15] S. Schaal and C. Atkeson, "Robot juggling: implementation of memory-based learning," *Control Systems Magazine, IEEE*, vol. 14, pp. 57–71, Feb 1994.

[16] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf, *Computational Geometry: Algorithms and Applications.* Springer-Verlag, second ed., 2000.

[17] J. H. Friedman, J. L. Bentley, and R. A. Finkel, "An algorithm for finding best matches in logarithmic expected time," *ACM Trans. Math. Softw.*, vol. 3, pp. 209–226, September 1977.

[18] A. Moore, "A tutorial on kd-trees," 1991. Available from http://www.cs.cmu.edu/ãwm.

[19] A. C. Rencher, *Linear Models in Statistics.* Wiley-Interscience, 2008.

[20] B. D. S. R. B. L. Busoniu, D. Ernst, "Approximate dynamic programming with a fuzzy parametrization," *Automatica*, vol. 46, no. 5, pp. 804–814, 2010.

Glossary

List of Acronyms

DCSC	Delft Center for Systems and Control
DP	Dynamic Programming
LA Dyna	Look Ahead Dyna
LLR	Local Linear Regression
LWR	Locally Weighted Regression
MDP	Markov Decision Process
POMDP	Partially Observable Markov Decision Process
PRESS	PREdiction Sum of Squares
PS	Prioritized Sweeping
RL	Reinforcement Learning
RMS	Root Mean Squared
RMSE	Root Mean Squared Error
TD	Temporal Difference

List of Symbols

α	Learning rate
$\boldsymbol{\beta}$	Regression variable
$\boldsymbol{\epsilon}$	Noise vector

$\delta_{Q,t}$	TD-error for $Q(s,a)$
$\delta_{V,t}$	TD-error for $V(s)$
ϵ	Exploration rate
γ	Discount factor
ω	Angular velocity
π	Policy
π^*	Optimal policy
ρ	Reward function
σ^2	Noise variance
θ	Angle
$\bar{a}$	Lead-in action to s
$\bar{s}$	Lead-in state to s
δ_x	Imposed prediction interval limit for variable x
$\hat{y}$	Estimated output
$\mathbf{e}$	Residual vector
$\mathbf{u}$	Input vector
$\mathbf{X}$	Input matrix
$\mathbf{x}$	Input vector
$\mathbf{Y}$	Output matrix
$\mathbf{y}$	Output vector
$\mathcal{A}$	Set of actions
$\mathcal{S}$	Set of states
$\tilde{M}$	LLR model of the mapping $(s_{t+1}, a_t) \to s_t$
a	Single action
I_x	Calculated prediction interval for variable x
K	Number of nearest neighbors
k	Dimension of a sample
M	LLR model of the mapping $(s_t, a_t) \to s_{t+1}$
N	Number of samples in the memory
N_m	Number of model-generated state-transitions per time step
n_t	Trial number
$Q(s,a)$	Action-value function
r	Immediate reward
R_t	Return
s	Single state
s^2	Variance estimator
t	Time
T_s	Sampling time
$V(s)$	Value function
x_q	Query input
T	Transition function