Technische Universität Berlin
School IV - Electrical Engineering and Computer Science
Department of Computer Engineering and Microelectronics
Robotics and Biology Laboratory

A thesis submitted for the degree of Bachelor of Science

A Practical Guide to Transformed Predictive State Representations

---

Author:
Niklas Wolf Andreas Gebauer

Supervisor:
Prof. Dr. Oliver Brock
Second Supervisor:
Prof. Dr. Marc Toussaint
Thesis Advisor:
Rico Jonschkowski

September 2015

Abstract

Predictive state representations (PSRs) are gaining a lot of attention in the robotics community lately because, in theory, they promise a powerful model that might be learned directly from data. But the practical application of PSRs remains a difficult procedure. There have only been a few learning algorithms proposed so far and only a small number of successful attempts where PSRs of complex domains were learned are reported. In this practical guide we aim to *ease* and *encourage* practical work with PSRs. On the one hand we provide the theoretical background and practical instructions to PSRs and on the other hand we identify possible questions that ought to be investigated to improve their practical applicability. To this end, we have re-implemented an algorithm that learns a PSR of a simulated mobile robot environment. We guide from the theory that is needed to understand the implemented algorithm to practice and provide in-depth information on all parts of our implementation. In a line of experiments we do not only validate former results that the learned PSRs are accurate enough to enable successful reinforcement learning, but further investigate the quality of learned models and the empirical performance of the algorithm itself. Therefore we apply the learning algorithm to environments of different complexity and examine the practical limits of the implemented approach. One of the main challenges we faced was the tuning of parameters. We found that tedious, environment specific fine tuning is needed to reliably learn accurate representations and thus investigate the influence of parameters on the quality of learned representations in several experiments in greater detail. The results are guidance for future work as well as they show possible problems that need to be tackled in order to improve PSR learning and make it applicable to complex real world domains.

Zusammenfassung

Predictive State Representations (PSRs) bilden einen interessanten Ansatz, der im Fachgebiet der Robotik immer mehr Aufmerksamkeit erhält. Die Theorie hinter PSRs lässt die Hoffnung zu, dass mit ihnen Zustandsrepräsentationen direkt anhand von Trainingsdaten erlernt werden können. Die praktische Anwendung und Implementierung gestaltet sich allerding schwerer als erhofft. Es gibt nur wenige Lernalgorithmen und eine noch geringere Anzahl an dokumentierten Ergebnissen, in denen PSRs in komplexen Systemen gelernt wurden. Außerdem gibt es wenige detaillierte Beschreibungen von Implementierungen. Diese Bachelorarbeit soll dazu beitragen, zu praktischer Arbeit mit PSRs zu motivieren und diese zu erleichtern. Zu diesem Zweck finden sich sowohl eine Zusammenfassung des theoretischen Hintergrundes von PSRs als auch spezifische Hilfsstellungen zur Implementierung wieder. Des Weiteren werden offene Fragen und Problemstellungen herausgearbeitet und diskutiert, um mögliche Ansätze für zukünftige Forschung zu geben.

Für diese Arbeit wurde ein Lernalgorithmus nachimplementiert, mit dem PSRs in einer Simulationsumgebung mit einem Roboter gelernt werden können. Diese Implementierung ist bis ins kleinste Detail dokumentiert, um als Beispiel für zukünftige Arbeiten dienen zu können. Zusätzlich sind viele experimentelle Ergebnisse des nachimplementierten Lernalgorithmus beschrieben. Diese bestätigen nicht nur frühere Ergebnisse, dass PSRs in komplexen Umgebungen anhand von Trainingsdaten gelernt werden können, sondern untersuchen vor allem empirisch die Qualität der gelernten Repräsentationen und die Grenzen des Lernalgorithmus. Einerseits wurden simulierte Umgebungen verschiedener Komplexität verwendet, um praktische Limitierungen des Lernansatzes herauszuarbeiten. Andererseits wurde der Einfluss der änderbaren Parameter des Algorithmus auf die Qualität der gelernten Repräsentationen untersucht, da sich bei der Implementierung herausstellte, dass diese Parameter sehr genau und umgebungsspezifisch angepasst werden müssen. Somit sollen diese Experimente dabei helfen, Zusammenhänge zwischen Parametern aufzudecken und besonders problematische Parameter zu identifizieren. Dies kann Möglichkeiten aufzeigen, wie Parameter in Zukunft schneller und einfacher bestimmt werden können.

Insgesamt können die Implementierung und die Ergebnisse als Einleitung, Orientierung und Hilfestellung dienen und somit den Zugang zu PSRs und vor allem ihrer praktischen Anwendung vereinfachen.

Contents

List of Figures **III**

List of Tables **V**

List of algorithms **VI**

1 Introduction **1**
1.1 State Representation Learning 1
1.2 Predictive State Representations 3
1.3 Contribution and Remainder 3
1.4 Related Work 4

2 Theoretical Background **6**
2.1 Predictive State Representations 6
2.2 Transformed Predictive State Representations 14
2.3 Features for Transformed Predictive State Representations . . 19
2.4 From Theory to Practice: Questions and Problems 24

3 Implementation **28**
3.1 Data Gathering and Preprocessing 28
3.2 Simple Batch Learning Algorithm 33
3.3 Tracking States 36
3.4 Important Parameters 37

4 Experimental Results **39**
4.1 Experimental Environment 40
4.2 Quality of Learned Representations 42
4.3 Influence of Parameters 51
4.4 Memory of TPSRs 61

5 Conclusion **66**

Bibliography **69**

Appendices **72**

A Additional Results **73**
A.1 Fully Observable Environment: All Dimensions Plotted . . . 73
A.2 Distractors in Fully Observable Environment: All Dimensions Plotted . . . 74
A.3 Partially Observable Environment: All Dimensions Plotted . 75
A.4 State Dimensionality: 5-dimensional Representation 76
A.5 TPSR Memory: Alternative Trajectories 77
A.6 TPSR Memory: Length One Histories and Tests 78

List of Figures

1.1 Controlled Dynamical System 2

2.1 System Dynamics Matrix $\mathcal{D}$ 8
2.2 Core Tests in $\mathcal{D}$ 9
2.3 Predictions as Linear Combinations in $\mathcal{D}$ 10
2.4 PSR State Update in $\mathcal{D}$ 11

4.1 Mobile Robot Environment and Observations 40
4.2 Experimental Design - Quality of Representations 43
4.3 Embedded Histories and Insufficient Parameters 44
4.4 Embedded Histories vs Updated States 44
4.5 Learned Representations and Results (Fully Observable Environment) 46
4.6 Environment with Distractors 47
4.7 Learned Representations and Results (Distractors in Fully Observable Environment) 48
4.8 Learned Representations and Results (Partially Observable Environment) 50
4.9 Number of Kernel Centers: State Representations 52
4.10 Number of Kernel Centers: Results 53
4.11 State Dimensionality: Results 54
4.12 Lenght of Tests and Histories: Results 55
4.13 Kernel Center Widths: Structure of Data 57
4.14 TPSR Memory: Learned Representation 61
4.15 TPSR Memory: Trajectories in Environment 62
4.16 TPSR Memory: Results 1 63
4.17 TPSR Memory: Results 2 64

A.1 All Dimensions of Learned TPSR (Fully Observable Environment) 73
A.2 All Dimensions of Learned TPSR (Distractors in Fully Observable Environment) 74
A.3 All Dimensions of Learned TPSR (Partially Observable Environment) 75

A.4 State Dimensionality: 5-dimensional Representation 76
A.5 TPSR Memory: Alternative Trajectories 77
A.6 TPSR Memory: Histories and Tests of Length One 78

List of Tables

3.1 Data Gathering Output 29
3.2 Whitening Output 30
3.3 Feature Extraction Output 32
3.4 TPSR Batch Learning Output 36
3.5 Parameters 38

List of Algorithms

1 Data Whitening 30
2 Feature Extraction 32
3 TPSR Batch Learning 35
4 TPSR State Update 37

Chapter 1

Introduction

Predictive state representations (PSRs, Littman et al. (2002)) are gaining a lot of attention in the robotics community lately because, in theory, they promise a powerful model that might be learned directly from data. But the practical application of PSRs remains a difficult procedure. Most proposed learning approaches are either limited to simple synthetic domains or hard to re-implement due to a lack of implementation details. Therefore, we carefully documented every step of our re-implementation of the batch learning algorithm from Boots et al. (2011) in this practical guide. Since we needed a lot of manual parameter fine tuning to reliably learn PSRs of a mobile robot environment with continuous observation space, we conducted several experiments that examine the influence of parameters on learned representations. We furthermore show practical limits of the implemented approach and state possible reasons for these limitations. Altogether, we aim to provide a solid starting point for future practical PSR work by both simplifying the implementation and identifying open questions.
In the following we will briefly introduce the field of *state representation learning* to motivate work with PSRs. We then proceed to explain the basic concept of PSRs, formulate the main contributions and structure of this thesis, and complete the introduction with a summary of related work.

1.1 State Representation Learning

Many problems in robotics can be described by controlled dynamical systems as shown in figure 1.1. An Agent can perform actions that influence its environment and gets back an observation and a reward. Usually the agent should learn to maximize its future reward. Therefore it maintains a *state* that captures the situation of the whole system including the environment and itself.
Compact state representations are mandatory for reinforcement learning algorithms that find policies to maximize reward as they suffer from the curse

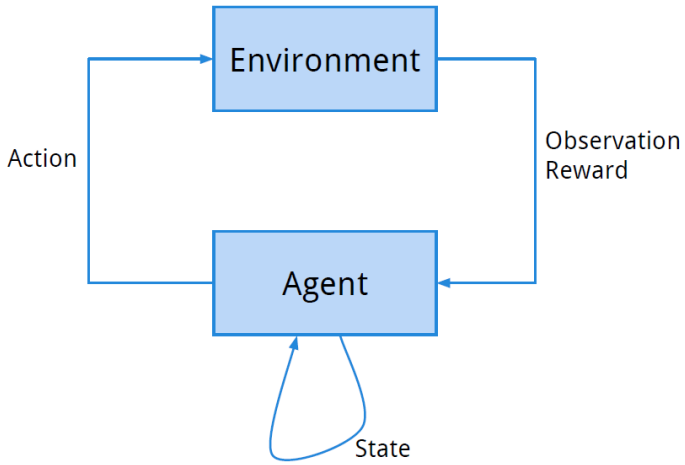


Figure 1.1: A general discrete time, controlled dynamical system: an agent performs actions which influence the environment and gets back an observation and reward from the environment. The agent itself has to hold a state which describes the current setting of the whole dynamical system in order to be able to make meaningful decisions.

of dimensionality. This means that they scale poorly with increasing dimensionality of the state space. Therefore the state representations that were used over the last decades in robotics were mainly hand engineered by humans who identified the main properties of the dynamical system and included task specific knowledge to build suitable, compact representations. But autonomous *state representation learning* is an emerging field in robotics (Böhmer et al., 2015) because learning these representations has several advantages.

First, modern robots face more complex tasks in complex environments where it is very time consuming, if not impossible to hand design suitable representations. Second, the tasks a robot has to solve can be diverse and need different representations. It might be more convenient to learn different representations instead of engineering them. Especially because third, maybe not all of the diverse tasks can be foreseen and thus learning representations on demand is necessary. Therefore fourth, learning representations allows to use robots more flexible and is a fine step forward to building truly autonomous agents.

One approach towards state representation learning is to find generic models that can describe any dynamical system. Partially observable Markov decision processes (POMDPs, Sondik (1971), Cassandra et al. (1994)) are a widely known and used model of such kind. They capture the system's state as a probability distribution over latent variables. But learning accu-

rate POMDPs from data that enable approximate planning techniques has only met with limited success (Boots et al., 2011). PSRs are an alternative model that can also describe dynamical systems in a general way. But they have been proposed with the focus on being easier to learn directly from data. We will introduce the basic concept of PSRs and why they might be easier to learn in the following section.

1.2 Predictive State Representations

PSRs capture the state of dynamical systems in terms of predictions about future events. A prediction is the probability of seeing something particular after acting in a certain way. For example, this could be the probability of hitting a wall when making three steps backwards or the probability of seeing a window when turning left. Usually, these probabilities are conditioned on the history of the system and therefore include information about the past. The idea is that maintaining a possibly small number of such predictions yields an adequate state representation. It has been shown that PSRs do not need more predictions to represent any dynamical system than the number of latent variables an unstructured POMDP of that system would have (Littman et al., 2002). Therefore, PSRs are *at least* as compact as POMDPs. Furthermore, it has been shown that PSRs have a greater representational capacity than POMDPs. In fact, POMDPs are a subset of PSRs (Singh et al., 2004). Additionally, it has been suggested that they might be easier to learn than POMDPs and related models. The reason for this is that predictions are *observable* quantities. We can do certain actions in the system and track resulting observations to gather data that is directly linked to the state representation, whereas POMDPs reason about *latent* variables that can never be observed directly (Littman et al., 2002). Nevertheless, whether PSRs can keep up with the promising theory in practice has yet to be answered. According to Boots et al. (2011) there are only a few algorithms that can learn PSRs from data and in the cited work they presented the first algorithm that learned PSRs which were accurate enough to enable subsequent approximate planning in a complex domain with continuous observation space. For this thesis we re-implemented this algorithm. To be precise, it learns a specific form of PSRs called *transformed PSRs* (TPSRs, Rosencrantz et al. (2004)) which we will thoroughly explain in section 2.2.

1.3 Contribution and Remainder

The main contribution of this thesis is twofold. First, we provide in-depth information on our implementation of the TPSR batch learning algorithm from Boots et al. (2011) and second, we evaluate weaknesses of the imple-

mented approach in a line of experiments.
The former is beneficial because despite the attention PSRs are gaining lately, the implementation and work with them in practice has proven difficult for us as not many practical details are given. Although the implemented algorithm mainly makes use of simple algebraic operations, there are many small details to decide and parameters to tune. Therefore we carefully documented *all* steps of our implementation in greater detail, raise awareness for all the tunable parameters involved, and thoroughly examine their influence on learned representations in our experiments. In this way, our work can serve as an example that simplifies the implementation for others and thus lowers the burden of working with PSRs. To this end, we also cover all the theory needed to understand the implemented algorithm from the first proposal of PSRs (Littman et al., 2002; Singh et al., 2004) over the introduction of spectral learning methods leveraging TPSRs (Rosencrantz et al., 2004) to the use of *features* to deal with more complex, continuous domains (Boots et al., 2011).
The second contribution is that we reveal open questions and problems that need to be tackled in future work. We analyze the empirical performance of the implemented approach in a set of experiments and determine possible sources of error. We show that the learned TPSRs are limited in practice and emphasize that this is caused by the discrepancies between theoretical assumptions and practical conditions.
Both contributions support our goal to ease and encourage practical future PSR work. The remainder of this thesis is organized as follows: after summarizing related work in the next section, chapter 2 introduces the theory of PSRs including the extension to TPSRs and usage of features. Chapter 3 thoroughly describes all steps of our implementation including the data pre-processing, the TPSR batch learning algorithm, and how to use learned models. Chapter 4 presents our experimental results that give insight into the relation and influence of parameters and weaknesses of the learning approach in practice. In the end, we draw a brief conclusion in chapter 5.

1.4 Related Work

The implemented algorithm of Boots et al. (2011) builds upon the TPSR notion and learning algorithm of Rosencrantz et al. (2004). It is using spectral decomposition to discover a compact subspace and is therefore closely related to spectral algorithms for learning (reduced) hidden Markov models (HMMs, Hsu et al. (2012); Siddiqi et al. (2010)) and subspace identification for learning linear dynamical systems (van Overschee and de Moor, 1996).
There are just a few reports on successful PSR learning attempts that

enabled *reinforcement learning* and *planning* in fairly complex domains. Stork et al. (2015) implemented a similar form of the TPSR batch learning algorithm to model robotic in-hand manipulation but used *string kernel features* instead of radial basis function (RBF) kernel features which were used in this and the original work of Boots et al. (2011). These string kernel features allowed them to incorporate information on taken actions, whereas RBF kernel features only take observations into account (for more details on features see section 2.3). Hamilton et al. (2013) presented a similar algorithm to learn PSRs of systems with discrete observation and action spaces by using *compression* methods which can be seen as a problem independent, automated way of finding features. Furthermore, there have been proposed extensions to the implemented TPSR learning algorithm from batch to online learning using random Fourier features (Boots and Gordon, 2011) and to Hilbert space embedding of non-parametric distributions for continuous action and observation spaces (Boots et al., 2013, 2014).
On a last note, there is a line of recent work that deals with the improvement of PSRs in practice (Kulesza et al., 2015a,b). Our experiments show that the practical performance of (T)PSRs is still far from theoretical expectations. This is mainly caused by the complexity of real world tasks so that theoretical assumptions do not hold. The mentioned approaches already start to exhibit what we hope to encourage with this thesis: PSR work that aims to automate parameter tuning and cope with problems of real world tasks that are mostly ignored in theory.

Chapter 2

Theoretical Background

This chapter will deal with the theory and main ideas of PSRs that are needed in order to understand the implemented algorithm and experiments we have done. We will build upon the basic notion of PSRs step by step: First, we will provide intuition into why and how PSRs work in theory and cover the basic mathematical formulation. Afterwards, we will introduce a generalization of PSRs called *transformed predictive state representations* (TPSRs), which simplify learning in many ways. As these basic concepts are limited to discrete action and observation spaces of moderate cardinality, we will proceed to explain how *features of observations* will allow us to learn TPSRs of more complex domains with continuous observations. In the end, we will give a brief summary and emphasize possible questions and problems concerning the practical application of the plain theory.

2.1 Predictive State Representations

When PSRs were first proposed, the derivation strongly relied on the notion of POMDPs (Littman et al., 2002). But Singh et al. (2004) introduced a more general and independent derivation using the concept of a *system dynamics matrix*. We decided to stick with this second derivation in the following because it provides intuition into why and how PSRs work in an understandable way.
The goal of PSRs is to model discrete time, controlled dynamical systems like sketched in figure 1.1. As already mentioned in the introduction, PSRs capture such systems in terms of predictions about the future considering the past. We will first define *tests*, which describe the future, *histories*, which describe the past, and *predictions* thereof to formalize this basic notion of PSRs. Building on the knowledge of predictions we can afterwards introduce the concept of the *system dynamics matrix* $\mathcal{D}$ and proceed to derive PSRs using $\mathcal{D}$. We will then complete the section by giving the mathematical formulation of PSRs and explaining ways to learn them.

Tests, Histories and Predictions

Predictions of tests and histories form the basis of PSRs. We will define them formally in the following. Assume that we are given a dynamical system with finite, discrete sets of observations $o \in \mathcal{O}$ and actions $a \in \mathcal{A}$. Each time step k in our system consists of one action-observation pair $a^k o^k$: at time k the agent has just taken an action $a^k = a \in \mathcal{A}$ and seen an observation $o^k = o \in \mathcal{O}$. We can define *tests* $t_j = (a_1^{t_j} o_1^{t_j} a_2^{t_j} o_2^{t_j} ... a_{|t_j|}^{t_j} o_{|t_j|}^{t_j})$ that describe possible futures of our system. A tests is an ordered sequence of any number of action-observation pairs and its length $|t_j|$ is defined as the number of contained action-observation pairs. The superscripts note the corresponding test and the subscripts describe the order of actions and observations. Accordingly, $a_x^t o_x^t$ is the x-th action-observation pair of a test t and $a_x^t o_x^t$ can be any combination of action in $\mathcal{A}$ and observation in $\mathcal{O}$. As we can define tests of arbitrary length, there exist infinitely many possible tests.
A dynamical system has a certain probability that a test t_j succeeds, this means a certain probability that we exactly get to see the observation sequence $o_1^{t_j} ... o_{|t_j|}^{t_j}$ when we execute the action sequence $a_1^{t_j} ... a_{|t_j|}^{t_j}$ as specified in t_j. Formally, we call this a *prediction* p of test t_j:

$$p(t_j) = prob(o^{k+1} = o_1^{t_j}, ..., o^{k+|t_j|} = o_{|t_j|}^{t_j} | a^{k+1} = a_1^{t_j}, ..., a^{k+|t_j|} = a_{|t_j|}^{t_j})$$

In the same way, we can define *histories* $h_i = (a_1^{h_i} o_1^{h_i} a_2^{h_i} o_2^{h_i} ... a_{|h_i|}^{h_i} o_{|h_i|}^{h_i})$ which describe the past of our system. A history is an ordered sequence of any number of action-observation pairs, which indicates the actions that have been executed and observations that have been seen so far. Again, there are infinitely many possible histories and the length $|h_i|$ of a history is defined as the number of contained action-observation pairs. Of course different histories can lead to different predictions for the same tests. To this end, predictions about the future can be conditioned on the past:

$$\begin{aligned} p(t_j|h_i) = prob(&o^{k+1} = o_1^{t_j}, ..., o^{k+|t_j|} = o_{|t_j|}^{t_j} | \\ &a^0 = a_1^{h_i}, o^0 = o_1^{h_i}, ..., a^k = a_{|h_i|}^{h_i}, o^k = o_{|h_i|}^{h_i}, \\ &a^{k+1} = a_1^{t_j}, ..., a^{k+|t_j|} = a_{|t_j|}^{t_j}) \end{aligned}$$

We colored actions and observations belonging to the test blue and those belonging to the history orange. The formula expresses that prediction $p(t_j|h_i)$ is the success probability of test t_j in a system with current history h_i. For the sake of readability, we will write predictions as the one above in a simplified way from here on:

$$p(t_j|h_i) = prob(o_1 ... o_{|t_j|} | h_i, a_1 ... a_{|t_j|})$$

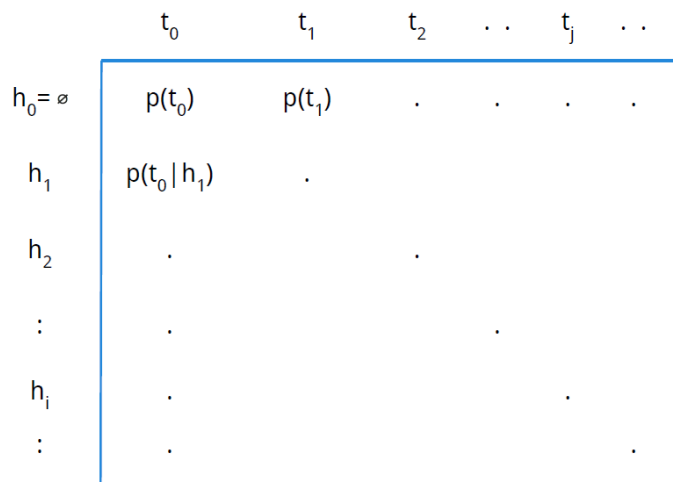


Figure 2.1: The *system dynamics matrix* $\mathcal{D}$ of infinite size contains all possible predictions one could make. h_0 represents the empty history after system reset. (Own version of figure 2 in Singh et al. (2004))

Here we summarized the past action-observation pairs which define time steps $0...k$ into h_j, so that we can drop the superscript from the remaining actions and observations as they all belong to the test t_j and write them as sequences which implicitly describe the future time steps $k+1...k+|t_j|$.

System Dynamics Matrix

The concept of the *system dynamics matrix* $\mathcal{D}$ is that predictions can completely describe a dynamical system. If we know the predictions of all possible tests for all possible histories, we know everything there is to know about the system in order to plan future steps[1] . Thus, the system dynamics matrix $\mathcal{D}$ consists of all possible predictions (see figure 2.1). Each row of $\mathcal{D}$ corresponds to a certain history, each column corresponds to a certain test, and each entry is the prediction of the column's test conditioned on the row's history:

$$\mathcal{D}_{ij} = p(t_j|h_i)$$

The basic idea of describing dynamic systems in terms of a *system dynamics matrix* is that the goal of every model of such a system is to make predictions. In fields like robotics where many problems deal with planning and control, this assumption is a valid one, because the predictions enable agents to reason about the effect of their future actions. Therefore, being able to infer all predictions provides a basis for solving reinforcement learning and planning tasks. In fact, Singh et al. (2004) have shown that common models

[1]Note that the rows of $\mathcal{D}$ are uniquely specified by its first row corresponding to the empty history. For further details, see Singh et al. (2004)

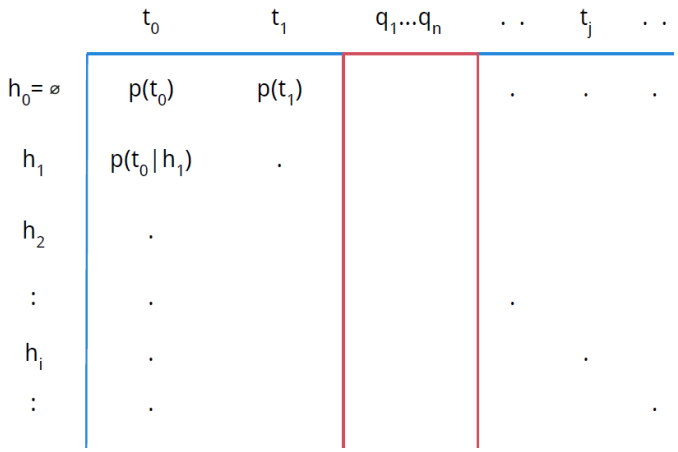


Figure 2.2: The n linearly independent columns of $\mathcal{D}$ correspond to certain set Q of tests $q_1, ..., q_n$. They form a submatrix $\mathcal{D}(Q)$ (marked red). (Own version of figure 4 in Singh et al. (2004))

like POMDPs can also be used to build the system dynamics matrix $\mathcal{D}$ and thus infer all possible predictions.

Derivation of PSRs

The novel idea of PSRs is that they cannot only make all predictions, but they directly use a set of predictions in order to represent the system's state. In the following, we will make use of the *system dynamics matrix* $\mathcal{D}$ to find a suitable PSR state and explain how PSRs work in theory.
The matrix $\mathcal{D}$ of any dynamical system has a rank n. Singh et al. (2004) define this rank as the *linear dimension* of the dynamical system. This *linear dimension* is an indicator for the complexity of a dynamical system. As $\mathcal{D}$ is of infinite size, the rank could possibly be infinite. For now, we assume that the rank n of $\mathcal{D}$ is finite. This means that $\mathcal{D}$ has n linearly independent columns, which each correspond to a certain test. We will call these n tests $q_1, q_2, ..., q_n$ and label the set containing these tests Q (see figure 2.2).
The n linearly independent columns in $\mathcal{D}$ form a submatrix $\mathcal{D}(Q)$. Since the rank of $\mathcal{D}$ is n and we have n linearly independent columns in $\mathcal{D}(Q)$, we can compute all the other columns of $\mathcal{D}$ as linear combinations of the columns in $\mathcal{D}(Q)$. This means that knowing the predictions of the tests $q_1, ..., q_n$ allows us to compute the predictions of any other test t_j for any history h_i using a row vector of weights $m_{t_j} \in \mathbb{R}^{1\times|Q|}$. If we let $Q(h_i)^T \in \mathbb{R}^{1\times|Q|}$ be the row of $\mathcal{D}(Q)$ corresponding to history h_i, we can write:

$$p(t_j|h_i) = m_{t_j} Q(h_i)$$

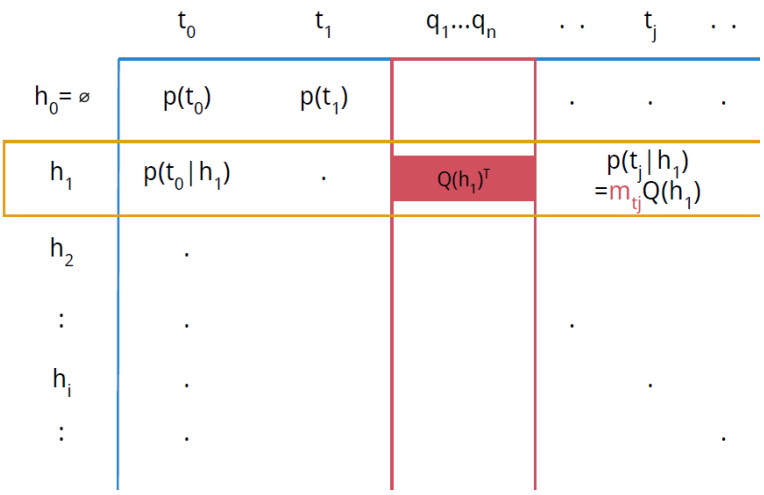


Figure 2.3: We can compute the predictions of any test t_j as a linear combination of the *core tests* $q_1, ...q_n$ using a weight vector m_{t_j}. Note that m_{t_j} is valid for all histories.

An illustration of this can be found in figure 2.3. As the predictions of tests $q_1, ..., q_n$ form a sufficient statistic for calculating predictions of all possible tests, they are called *core tests* (Singh et al., 2004).

The state of a PSR consists exactly of the predictions of these *core tests*. Hence, one row of $\mathcal{D}(Q)$ is equivalent to a PRS state. If the history of our system is h_i, the PSR state that represents the current situation is $Q(h_i)$, a vector of *core test* predictions conditioned on the current history:

$$Q(h_i) = \begin{pmatrix} p(q_1|h_i) \\ . \\ . \\ . \\ p(q_n|h_i) \end{pmatrix}$$

In the following we will describe how to maintain this state. If the agent executes an action a and gets to see an observation o, we need to update the state. The old history h_i will be extended with the new action-observation pair ao (see figure 2.4). Thus, the new state will be $Q(h_iao)$. To update we only need the prediction of the one step test containing the current action-observation pair ao and the predictions of the one step extended core tests aoq_j where the current action-observation pair is appended at the beginning. These predictions can be calculated as linear combinations of the current state with weight vectors m_{ao} and m_{aoq_i}. Then we can update the current state $Q(h_i)$ *entry wise* by updating the predictions of the core tests $q_j \in Q$

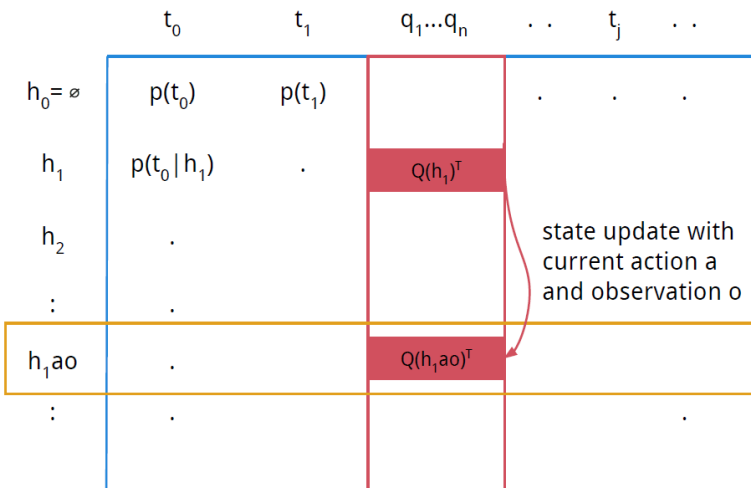


Figure 2.4: When doing an action a and observing an observation o, the state of our system needs to be updated. In this example we are looking for a transition from $Q(h_1)$ to $Q(h_1ao)$, which corresponds to another row in our submatrix Q.

using Bayes' rule (Littman et al., 2002):

$$p(q_j|h_iao) = prob(o_1...o_{|q_j|}|h_iao, a_1...a_{|q_j|}) = \frac{prob(oo_1...o_{|q_j|}|h_i, aa_1...a_{|q_j|})}{prob(o|h_i, a)}$$
$$= \frac{p(aoq_j|h_i)}{p(ao|h_i)} = \frac{m_{aoq_j}Q(h_i)}{m_{ao}Q(h_i)}$$

Again we colored actions and observations belonging to the core test q_j blue, those belonging to the current history h_i orange, and the new pair of action we executed and observation we noted green.
Let $M_{ao} \in \mathbb{R}^{|Q|\times|Q|}$ be the matrix with rows m_{aoq_j}. Then the update of the *complete* state can be written as:

$$Q(h_iao) = \frac{M_{ao}Q(h_i)}{m_{ao}Q(h_i)} \tag{2.1}$$

Together with the set of core tests Q, the parameters $\{m_{ao}\}_{a\in\mathcal{A},o\in\mathcal{O}}$ and $\{m_{aoq_j}\}_{a\in\mathcal{A},o\in\mathcal{O},q_j\in Q}$ form a *predictive state representation* of a dynamical system. They are sufficient to maintain state and make predictions. To be precise, the weight vector m_t of any test $t = (a_1^t o_1^t a_2^t o_2^t ... a_{|t|}^t o_{|t|}^t)$ can be calculated from the above parameters as follows (Littman et al., 2002):

$$m_t = m_{a_{|t|}^t o_{|t|}^t} M_{a_{|t|-1}^t o_{|t|-1}^t} ... M_{a_1^t o_1^t} \tag{2.2}$$

Therefore, PSRs can maintain the state when actions are taken and observations are noted and make any possible prediction using just the predictions

of *core tests* and the mentioned parameters. This means that we do not have to explicitly store the current history h as it is implicitly expressed in the *core tests'* predictions. Thus, a PSR state $Q(h)$ is a sufficient statistic of history.
The main insight of PSRs is that a possibly small number of predictions about the future are sufficient to capture information arbitrarily far back in the past and reason about effects of actions arbitrarily far in future. Therefore, in theory, PSRs have unlimited memory and an unlimited prediction horizon.
Another benefit that was highlighted when PSRs were proposed is that they are expressed using observable quantities. Predictions can be observed or estimated because we can execute the actions in tests and record the observations. Hence, it was said that learning PSRs could potentially be easier than learning POMDPs or HMMs which are expressed in terms of latent variables that can never be observed (Littman et al., 2002). Despite the expressed hopes, there was no learning algorithm or sketch given. After describing PSRs formally in the next part, we will briefly summarize some learning attempts that have been proposed during the last decade and then proceed to provide the extensions to basic PSRs needed for the learning algorithm we re-implemented in this work.

Mathematical formulation and learning of PSRs

A PSR can formally be expressed as a quintuple $(\mathcal{A}, \mathcal{O}, Q, F, m_1)$ (Boots et al., 2011). As in the derivation above, $\mathcal{A}$ and $\mathcal{O}$ are finite sets of actions and observations, respectively and Q is a set of *core tests*. Q is called *minimal* if it contains only linearly independent tests. This means that no prediction of any test $q_i \in Q$ should be a linear combination of the predictions of other tests in Q. The set F contains prediction functions, so that the prediction of any test t can be calculated as a function m_t of the *core tests*: $p(t|h) = m_t(Q(h)) \quad \forall h$. In this work we will restrict the prediction functions to be linear[2] Hence, they can be expressed as a vector $m_t \in \mathbb{R}^{1\times|Q|}$. PSRs using linear prediction functions are formally called *linear PSRs*, but we will dropped the term linear in this work in favor of readability and note that there is barely any work on non-linear PSRs despite that of Rudary and Singh (2004). From the derivation in the previous part we know that F consists of the parameters $\{m_{ao}\}_{a\in\mathcal{A},o\in\mathcal{O}}$ and $\{m_{aoq_i}\}_{a\in\mathcal{A},o\in\mathcal{O},q_i\in Q}$, which are sufficient to calculate every other possible prediction function. m_1 is the initial prediction vector of *core tests* without having seen any history: $m_1 = Q(\emptyset)$. This parameter can only be calculated if we can reset our dynamical system to a common starting state. If we cannot reset our system to a common starting state, as it is the case for most real world problems,

[2]It is important to note that the use of linear prediction functions does *not* limit PSRs to only learn linear dynamical systems (Littman et al., 2002).

we can instead estimate any arbitrary feasible starting state m_* to enable prediction. This approximate starting state will influence the first predictions, but the difference will disappear over time as the process mixes (Boots et al., 2011).
The learning of PSRs can be devided into two sub-problems, namely the *discovery problem* and the *learning problem* (James and Singh, 2004). Here the *discovery problem* describes the problem of finding the core tests, whereas the *learning problem* deals with learning of the parameters m_{ao} and m_{aoq_i} that maintain the state. In the end, a state representation should enable solving planning or control problems. Since these problems suffer from the curse of dimensionality, state representations should be as compact as possible. As the dimensionality of a PSR is defined by its number of *core tests* $|Q|$, finding a *minimal* set of *core tests* is important and aggravates the *discovery problem.*
Learning approaches for standard PSRs usually involve repeated estimation of parts of the *system dynamics matrix* $\mathcal{D}$ to determine its rank and to find the *core tests* in order to solve the *discovery problem* (James and Singh, 2004; Wolfe et al., 2005). The *learning problem* can then be solved in a straight forward manner by regression using the found set of *core tests* Q and the estimated *system dynamics matrix* $\mathcal{D}$. However, solving the *discovery problem* in this way has high computational complexity and limits those learning algorithms to very small domains with just a few actions and observations (Hamilton, 2014).
Thus, in the following section, we will introduce a certain form of PSRs called *transformed PSRs* (Rosencrantz et al., 2004) that allow a different approach to solving the *discovery problem.* But there is one more thing that we want to note beforehand. (Boots et al., 2011) do not use m_{ao} parameters in order to calculate the prediction of a test ao by $p(ao|h) = prob(o|h, a) = m_{ao}Q(h)$ (where h is the current history). Instead, they calculate another parameter $m_\infty \in \mathbb{R}^{|Q|\times 1}$ that is defined by $m_\infty^T Q(h) = 1 \quad \forall h$. Thus, this parameter is a normalizer that marginalizes out the probability of the *core tests'* predictions so that $m_\infty^T M_{ao} Q(h) = m_{ao}Q(h) = prob(o|h, a)$. Recall that $M_{ao} \in \mathbb{R}^{|Q|\times|Q|}$ is the matrix with rows m_{aoq_i}. Hence, we can rewrite the state update from equation 2.1 using m_∞ instead of m_{ao}:

$$Q(hao) = \frac{M_{ao}Q(h)}{m_{ao}Q(h)} = \frac{M_{ao}Q(h)}{m_\infty^T M_{ao}Q(h)} \tag{2.3}$$

In a similar way, we can rewrite equation 2.2 that gives us weight vector m_t for any arbitrary test $t = (a_1o_1a_2o_2...a_{|t|}o_{|t|})$:

$$m_t = m_{a_{|t|}o_{|t|}} M_{a_{|t|-1}o_{|t|-1}} ... M_{a_1o_1} = m_\infty^T M_{a_{|t|}o_{|t|}} M_{a_{|t|-1}o_{|t|-1}} ... M_{a_1o_1} \tag{2.4}$$

In the remainder of this work we will therefore refer to the matrices M_{ao} and the vectors m_∞ and m_* if we are speaking of the PSR parameters that need to be found by solving the *learning problem.*

2.2 Transformed Predictive State Representations

In this section we will introduce a variant of PSRs called *transformed predictive state representations* (TPSRs), explain how they allow to mitigate the discovery problem, and show how they can be learned.

Definition of TPSRs

The parameters of a PSR are only determined up to a similarity transform as we will explain below. This means that we can learn linear transformations of normal PSR parameters which will make exactly the same predictions as the original ones. A TPSR consists of these linear transformed PSR parameters. Let J be an invertible matrix. Then we can define the TPSR parameters as follows (Boots and Gordon, 2011):

$$\begin{aligned} B_{ao} &= JM_{ao}J^{-1} \\ b_\infty^T &= m_\infty J^{-1} \\ b_* &= Jm_* \end{aligned}$$

The state $b(h)$ of a TPSR is a linear transformation of the underlying normal PSRs state $Q(h)$ as well:

$$b(h) = JQ(h)$$

We can show that in fact the prediction for a test $t = (a_1o_1a_2o_2...a_{|t|}o_{|t|})$ in PSR state $Q(h)$ is exactly the same as the prediction of t using the TPSR state and parameters:

$$\begin{aligned} p(t|h) &= m_t Q(h) \\ &\overset{2.4}{=} m_\infty^T M_{a_{|t|}o_{|t|}} M_{a_{|t|-1}o_{|t|-1}} ... M_{a_1o_1} Q(h) \\ &= m_\infty^T (J^{-1}J) M_{a_{|t|}o_{|t|}} (J^{-1}J) M_{a_{|t|-1}o_{|t|-1}} (J^{-1}J) ... (J^{-1}J) M_{a_1o_1} (J^{-1}J) Q(h) \\ &= (m_\infty^T J^{-1})(JM_{a_{|t|}o_{|t|}}J^{-1})(JM_{a_{|t|-1}o_{|t|-1}}J^{-1})(J...J^{-1})(JM_{a_1o_1}J^{-1})(JQ(h)) \\ &= b_\infty^T B_{a_{|t|}o_{|t|}} B_{a_{|t|-1}o_{|t|-1}} ... B_{a_1o_1} b(h) \end{aligned}$$

Additionally, we can update a TPSR state just as a PSR state:

$$\begin{aligned} b(hao) = JQ(hao) &\overset{2.3}{=} \frac{JM_{ao}Q(h)}{m_\infty M_{ao}Q(h)} \\ &= \frac{JM_{ao}(J^{-1}J)Q(h)}{m_\infty (J^{-1}J) M_{ao} (J^{-1}J) Q(h)} \\ &= \frac{(JM_{ao}J^{-1})(JQ(h))}{(m_\infty J^{-1})(JM_{ao}J^{-1})(JQ(h))} \\ &= \frac{B_{ao}b(h)}{b_\infty B_{ao} b(h)} \end{aligned} \tag{2.5}$$

Thus TPSRs form a generalization of PSRs that include PSRs as a special case, since the transformation J can be the identity.

Mitigating the Discovery Problem

Rosencrantz et al. (2004) have shown that TPSRs can mitigate the *discovery problem*. To this end, the transformation J needs to be factorized into two matrices: $J = U^T R$. Let $\mathcal{T}$ be a large set of tests. We can define $R \in \mathbb{R}^{|\mathcal{T}| \times |Q|}$ to be the matrix that maps the state of the underlying (normal) PSR to the predictions of the tests $t_j \in \mathcal{T}$:

$$[p(t_1|h), ..., p(t_{|\mathcal{T}|}|h)]^T = RQ(h)$$

Then the transpose of $U \in \mathbb{R}^{|\mathcal{T}| \times |Q|}$ maps the predictions of tests in $\mathcal{T}$ to the TPSR state:

$$b(h) = JQ(h) = U^T RQ(h) = U^T [p(t_1|h), ..., p(t_{|\mathcal{T}|}|h)]^T$$

Therefore, if we factorize J like above, the TPSR state will consist of *linear combinations* of predictions of $|\mathcal{T}|$ tests' in contrast to predictions of *core tests* which form a usual PSR state.
The advantage of this definition is that we can *specify* the set of tests $\mathcal{T}$ which we want to use to learn a TPSR. We do not have to explicitly *obtain* core tests anymore but can take any set $\mathcal{T}$ that we think is big and varied enough to exhibit the behavior of the dynamical system we want to model (Rosencrantz et al., 2004). With this set $\mathcal{T}$ we can then calculate all TPSR parameters without knowing the core tests. We can calculate the transformation U straight forwardly using singular value decomposition and find the parameters by regression as we will explain later.
Nevertheless, the dimensionality of the TPSR is determined by the transformation matrix U independently of the number of tests in $\mathcal{T}$. In this way, TPSRs alleviate the *discovery problem*, but do not completely overcome it. The set of tests $\mathcal{T}$ still needs to contain the core tests in order to be able to learn an exact TPSR that is truly equivalent to the underlying PSR (Boots et al., 2011). The learning approach we will describe in the next part assumes that we specify a *sufficient* set of tests $\mathcal{T}$ that contains the core tests. The possibility to take large sets eases this step, but of course the assumption does not always hold in practice. We will further examine practical problems that could arise based on theoretical assumptions in section 2.4. But for the following we assume that we can provide a *sufficient* set $\mathcal{T}$.

Learning TPSR Parameters and Transformation

In this part we will show how the parameters and transformation matrix U of a TPSR can be learned. They can be computed in a straight forward manner since they correspond to *observable quantities* just like PSR parameters. This means that we can solve for them using estimates of matrices directly built from execution traces of tests and histories. In the following

we will introduce three matrices called P-statistics and defined by Boots et al. (2011), explain how they can be estimated from data, and then show how they allow to solve for the transformation matrix U and the TPSR parameters.

P-statistics

The definition of the matrices relies on a sufficient set of tests $\mathcal{T}$. As mentioned above, we can specify $\mathcal{T}$ by taking a set that is large and varied enough to contain the core tests.
Similarly, we need a set of histories. We will therefore partition the infinite set of all possible histories into a finite set $\mathcal{H}$ of mutually exclusive sets H_i of histories. These mutually exclusive sets H_i are called *indicative events*. We can for example sort histories by equality of their last two observations or same first three actions after reset into *indicative events*. Again, there is an assumption that the set of indicative events $\mathcal{H}$ fulfills a certain sufficiency criterion that we will explain after equation 2.7.
The matrix $P_{\mathcal{H}} \in \mathbb{R}^{|\mathcal{H}|\times 1}$ contains the probability of every *indicative event* $H_i \in \mathcal{H}$:

$$[P_{\mathcal{H}}]_i = prob(H_i) \tag{2.6}$$

To gather data for estimation of $P_{\mathcal{H}}$ we can just repeatedly generate histories by sampling action-observation sequences following an exploration policy (which could be uniformly random) and count how often histories $h \in H_i$ occur. This approach requires us to be able to reset our system. But we can also estimate $P_{\mathcal{H}}$ without reset as we will briefly explain after we introduced the other two matrices that need to be estimated.
The matrix $P_{\mathcal{T},\mathcal{H}} \in \mathbb{R}^{|\mathcal{T}|\times|\mathcal{H}|}$ is similar to the *system dynamics matrix*. But instead of probabilities conditioned on histories, it contains joint probabilities of tests $t_i \in \mathcal{T}$ and *indicative events* $H_j \in \mathcal{H}$ and is therefore of finite size. Each row of $P_{\mathcal{T},\mathcal{H}}$ corresponds to a certain test and each column corresponds to a certain *indicative event*. Let t_i^O denote the observations and t_i^A denote the actions specified in test $t_i \in \mathcal{T}$. Then we can define $P_{\mathcal{T},\mathcal{H}}$:

$$[P_{\mathcal{T},\mathcal{H}}]_{ij} = prob(H_j, t_i^O | t_i^A) \tag{2.7}$$

Each entry is the probability of seeing a history $h \in H_j$ and then seeing observations t_i^O given that we execute actions t_i^A. This matrix has *at most* rank $|Q|$ (Boots et al., 2011). Recall that $|Q|$ is the number of *core tests* of the underlying PSR and thus the *linear dimension* of the system we want to model. Now we can define *sufficiency* of $\mathcal{H}$. The partition is *sufficient*, if the rank of $P_{\mathcal{T},\mathcal{H}}$ is equal to $|Q|$.
In order to estimate $P_{\mathcal{T},\mathcal{H}}$, we can repeatedly sample a history h as above, execute the actions t_i^A of a test, count how often the generated observations

match t_i^O and combine this information with the probabilities of the *indicative events*.
The last matrix $P_{\mathcal{T},ao,\mathcal{H}} \in \mathbb{R}^{|\mathcal{T}|\times|\mathcal{H}|}$ contains joint probabilities of an *indicative event*, the next action-observation pair, and a following test. Therefore we need to estimate a $P_{\mathcal{T},ao,\mathcal{H}}$ matrix for each action-observation pair $ao \in \mathcal{A} \times \mathcal{O}$.

$$[P_{\mathcal{T},ao,\mathcal{H}}]_{ij} = prob(H_j, o, t_i^O | a, t_i^A) \tag{2.8}$$

We can gather data to estimate $P_{\mathcal{T},ao,\mathcal{H}}$ in a similar manner to above, but we will need to take action a after sampling history h and before taking actions t_i^A and then compare the resulting observation sequence with ot_i^O instead.

Gathering Data Without Reset

All the described data gathering approaches would require a reset of the system. If we only have one single long trajectory of action-observation pairs, we can divide it into smaller sequences and pretend each sequence started with a reset (*suffix-history* algorithm, Wolfe et al. (2005)). Statistically seen, this will lead to more inaccurate estimates since the samples are not independent of each other. But these approximations usually are sufficient in practice (Boots et al., 2011).
We can furthermore avoid to explicitly having to execute actions of tests by using importance sampling (Bowling et al., 2008) if we know the exploration policy used. For example, we can just explore uniformly randomly and take sequences where the actions match the actions specified in a test as samples for that particular test. If the exploration policy is not uniformly random, we need to weight the samples according to the probabilities of the actions taken. We will be using this strategy when learning TPSRs with features as examined in section 2.3.

Finding Transformation Matrix U

The P-statistics we defined above can be related to the TPSR parameters and transformation U. We will first explain how U can be obtained and then proceed to show how the parameters can be found by regression in the following.
Recall that we need to find a matrix U so that $U^T R = J \in \mathbb{R}^{|Q|\times|Q|}$ is invertible. With the assumption from above that the rank of $P_{\mathcal{T},\mathcal{H}}$ is $|Q|$, this is equivalent to the constraint that $U^T P_{\mathcal{T},\mathcal{H}} \in \mathbb{R}^{|Q|\times|\mathcal{H}|}$ has full row rank (Boots et al., 2011). In order to find a transformation U with this property, we can make use of the *singular value decomposition* (SVD) of $P_{\mathcal{T},\mathcal{H}}$:

$$\text{SVD}(P_{\mathcal{T},\mathcal{H}}) = U_{orig} \Sigma V^T$$

Here we have matrices $U_{orig} \in \mathbb{R}^{|\mathcal{T}|\times|\mathcal{T}|}$ which contains the left singular vectors as columns, $\Sigma \in \mathbb{R}^{|\mathcal{T}|\times|\mathcal{H}|}$ which has singular values in decreasing order on its diagonal, and $V \in \mathbb{R}^{|\mathcal{H}|\times|\mathcal{H}|}$ which contains the right singular vectors as columns. Σ has as much non-zero singular values on its diagonal as the rank of the decomposed matrix $P_{\mathcal{T},\mathcal{H}}$. U_{orig} and V are orthogonal matrices, so that $U_{orig}^T U_{orig} = I^{\mathcal{T}\times\mathcal{T}}$ and $V^T V = I^{\mathcal{H}\times\mathcal{H}}$, where I is the identity matrix (Cline and Dhillon, 2013).
Thus, if $P_{\mathcal{T},\mathcal{H}}$ has rank $|Q|$, Σ will have $|Q|$ singular values greater than zero in decreasing order on its diagonal and the first $|Q|$ columns of U_{orig} are the left singular vectors corresponding to those non-zero singular values. Then, if we define U to consist of these columns of U_{orig} corresponding to non-zero singular values, the matrix product $U^T P_{\mathcal{T},\mathcal{H}}$ consists of $|Q|$ linearly independent rows and therefore has full row rank. Accordingly, the SVD of $P_{\mathcal{T},\mathcal{H}}$ constitutes a convenient method to obtain a suitable transformation U if the assumption that $P_{\mathcal{T},\mathcal{H}}$ has rank $|Q|$ holds. In order to tune the dimensionality of our model, we could pick less singular vectors than $|Q|$ from U_{orig}. The payoff is that this can lead to a loss in prediction accuracy, especially for long term predictions (Rosencrantz et al., 2004).

Learning the Parameters

With the transformation U, we can learn the TPSR parameters by regression using $P_{\mathcal{H}}$, $P_{\mathcal{T},\mathcal{H}}$, and $P_{\mathcal{T},ao,\mathcal{H}}$. Proofs of the following formulas can be found in Boots et al. (2011).

$$\begin{aligned} b_* &= U^T P_{\mathcal{T},\mathcal{H}} 1_{|\mathcal{H}|} \\ &= (U^T R) m_* && (2.9) \\ b_\infty^T &= P_{\mathcal{H}}^T (U^T P_{\mathcal{T},\mathcal{H}})^\dagger \\ &= m_\infty^T (U^T R)^{-1} && (2.10) \\ B_{ao} &= U^T P_{\mathcal{T},ao,\mathcal{H}} (U^T P_{\mathcal{T},\mathcal{H}})^\dagger \\ &= (U^T R) M_{ao} (U^T R)^{-1} && (2.11) \end{aligned}$$

Here $1_{|\mathcal{H}|} \in \mathbb{R}^{|\mathcal{H}|}$ is a vector consisting of only ones and $(U^T P_{\mathcal{T},\mathcal{H}})^\dagger$ marks a pseudo inverse. The first part of each formula shows how we can obtain each TPSR parameter using simple algebraic operations on the P-statistics and the second part shows the relation to the underlying PSR parameters. Thus, as explained in the beginning, a TPSR can be learned in this way just using matrices that we can estimate from data without having to know the *core tests*. Still, it will make exactly the same predictions as the underlying PSR under the assumption that $\mathcal{T}$ and $\mathcal{H}$ are sufficient so that $P_{\mathcal{T},\mathcal{H}}$ has rank $|Q|$.
This algorithm enables us to learn compact representations of systems that are fairly complex compared to systems that use the direct PSR approach

where we need to explicitly find the *core tests*. Nevertheless, we are restricted to discrete sets of actions and observations. Furthermore, in more complex systems where these sets are large, it might be impossible to provide reasonable sized sufficient sets of tests $\mathcal{T}$ and *indicative events* $\mathcal{H}$ (Boots et al., 2011). To learn such systems, the TPSR learning algorithm can be generalized to work with features as we will explain in the next section.

2.3 Features for Transformed Predictive State Representations

In this section we will explain how the use of *features* allows to learn TPSRs of more complex domains with continuous observations. Instead of working with tests and *indicative events* directly, we can also use *features* of tests and *indicative events*. Using features means to extract information from data and usually simplifies machine learning problems. A famous example where there is extensive use of features is object recognition. Instead of working on the prohibitively large space of image pixel data, features of images are extracted (e.g. see Lowe (1999)). These features can for example be certain shapes or edges. In our case, features of *indicative events* or tests could be the number of times we have seen a certain observation or the number of times we have done a certain action. Similar to the original implementation in Boots et al. (2011), we use *gaussian radial basis function* (RBF) kernels on observations to define features in our implementation. Therefore, in the following we will use gaussian RBF kernels to explain how to use features in accordance with our implementation, but should note that the approach is not limited to these features. For example random fourier features (Boots and Gordon, 2011) and string kernel features (Stork et al., 2015) have been used to learn TPSRs. The compression approach of Hamilton et al. (2013) to learn compressed PSRs can also be seen as a general way of extracting features for TPSRs.
We will first introduce gaussian RBF kernels for histories and tests and then define adapted versions of the P-statistics that are defined in terms of these features. Afterwards we will show how the parameters of a TPSR with features can be learned and draw a brief conclusion of learning TPSRs with features.

Gaussian RBF Kernels

Gaussian RBF kernels form a similarity measure between two observations. We will clarify this in terms of tests first. In addition to our set of tests $\mathcal{T}$, we will have another set of tests $\mathcal{T}'$. Each test $t'_i \in \mathcal{T}'$ is a so called *kernel center* and each *kernel center* will serve as one *feature*. The value of such a feature for a test $t_j \in \mathcal{T}$ will be the similarity of that test to the *kernel*

center t'_i. This similarity is defined by the *activation* of the kernel center by t_j, which can be written as $K(t'_i, t_j)$. Making use of these *features*, we can replace every test $t_j \in \mathcal{T}$ by a *feature vector* ϕ^{t_j} that consists of the activations of all kernel centers $t'_i \in \mathcal{T}'$ induced by t_j:

$$\phi^{t_j} = \begin{pmatrix} K(t'_1, t_j) \\ . \\ . \\ . \\ K(t'_{|\mathcal{T}'|}, t_j) \end{pmatrix}$$

Let $\phi^{\mathcal{T}} \in \mathbb{R}^{|\mathcal{T}'| \times |\mathcal{T}|}$ be the matrix whose columns are the feature vectors of all tests in $\mathcal{T}$. This *feature matrix* will form the data that we are working with instead of using tests directly. The activation of a gaussian RBF kernel is defined as follows:

$$K(t'_i, t_j) = exp(-\frac{||t'^O_i - t^O_j||^2}{2\sigma^2})$$

σ is a free parameter called *kernel width* and $||t'^O_i - t^O_j||^2$ is the squared euclidean distance between the observation sequences of test *kernel center* t'_i and test t_j. Observations are usually given as vectors of real numbers and we can compute the distance between two sequences of observations by stacking their observation vectors and using the standard definition of euclidean distance of two real valued vectors. The closer the two vectors representing observation sequences are, the higher is the kernel activation. At this point, two direct limitations emerge from using RBF kernel features. First, the similarity is only defined on observations, thus we lose information about the actions specified in tests, and second, we are limited to tests of equal length because otherwise the distance between their observation sequences cannot be calculated. We will discuss possible consequences of these limitations in section 2.4 and examine the quality of TPSR models learned with kernel RBF features in the experiments in chapter 4.
We can define *features* of *indicative events* in a similar way. In fact, when working with RBF kernel features we do not need to split histories into mutually exhaustive and exclusive sets of indicative events but we can take single histories as kernel centers. Therefore, we will have a set of *histories* $\mathcal{H}$ instead of indicative events and a set of *history kernel centers* $\mathcal{H}'$. Each *kernel center* will consist of an observation sequences just like test *features* defined above. We can build *feature vectors* which contain the activations of all history *kernel centers* for one history from $\mathcal{H}$ and *feature matrix* $\phi^{\mathcal{H}} \in \mathbb{R}^{|\mathcal{H}'| \times |\mathcal{H}|}$ contains the *feature vectors* of all histories as columns.

P-statistics in Terms of Feature Matrices

In the following we will describe the adjusted versions of the observable P-statistics (equations 2.6 - 2.8) and how they can be estimated in terms of the feature matrices $\phi^{\mathcal{H}}$ and $\phi^{\mathcal{T}}$. We will first state the equations of $P_{\mathcal{H}}$ and $P_{\mathcal{T},\mathcal{H}}$. Then we explain how to discretize the observations space with gaussian RBF kernel features as we would otherwise have to calculate intractably many $P_{\mathcal{T},ao,\mathcal{H}}$ matrices when dealing with continuous or very large observation spaces. The discretization will allow us to finally state the adjusted version of $P_{\mathcal{T},ao,\mathcal{H}}$.
To explain how the P-statistics can be estimated from data, we assume that we have a set E of training episodes. One episode consists of action-observation sequences that form one history, followed by a current action-observation pair and a subsequent test. Thus, $h_i \in E$ and $t_n \in E$ are all the histories and tests in our training data, respectively. $|E|$ is the number of training episodes.
We shifted our interest from probabilities of tests and *indicative events* to activations of test *features* and history *features*. Therefore the P-statistics will no longer consist of probabilities, but of expected values of features or products of features. In the following formulas, let $\phi_i^{\mathcal{T}}$ be the row of $\phi^{\mathcal{T}}$ consisting of the activations of test *kernel center* $t_i' \in \mathcal{T}'$ and $\phi_j^{\mathcal{H}}$ be the row of $\phi^{\mathcal{H}}$ consisting of the activations of history *kernel center* $h_j' \in \mathcal{H}'$.

Adjusted Definition of $P_{\mathcal{H}}$

The adjusted version of $P_{\mathcal{H}}$ in terms of *feature matrix* $\phi^{\mathcal{H}}$ is:

$$\begin{aligned}[P_{\mathcal{H}}]_j &= \mathbb{E}(\phi_j^{\mathcal{H}}) \\ &\approx \frac{1}{|E|} \sum_{h \in E} K(h_j', h) \end{aligned} \tag{2.12}$$

Each entry of $P_{\mathcal{H}}$ contains the expected value of one history *feature*. The second row shows how we can approximate P_H in practice using training data. We can use the empirical expectation of one *feature* which is the mean of all activations of the particular *kernel center* induced by each history in our training data.

Adjusted Definition of $P_{\mathcal{T},\mathcal{H}}$

Each entry of $P_{\mathcal{T},\mathcal{H}}$ is the expected value of the product of one test *feature* and one history *feature*:

$$\begin{aligned}[P_{\mathcal{T},\mathcal{H}}]_{ij} &= \mathbb{E}(\phi_i^{\mathcal{T}} \phi_j^{\mathcal{H}} | (\phi_i^{\mathcal{T}})^A) \\ &\approx \frac{1}{|E|} \sum_{n=1}^{|E|} \beta_n K(t_i', t_n) K(h_j', h_n) \end{aligned} \tag{2.13}$$

Here $(\phi_i^{\mathcal{T}})^A$ refers to the actions taken in the tests used to calculate *feature* $\phi_i^{\mathcal{T}}$ and β_n is an importance weight according to the exploration policy used to sample E. Again, we can approximate this matrix as the empirical expectation of products of test *features* and *indicative features* using our set E of training episodes.

Discretizing the Observation Space

When dealing with continuous or very large observation spaces, we would need to calculate intractably many $P_{\mathcal{T},ao,\mathcal{H}}$ matrices. But, in a similar notion to the features defined before, we can discretize the observation space using RBF kernel *features* on single observations. In a mathematical sense, we will model the observation probability density function (PDF) using kernel density estimation (KDE, Silverman (1986)).
Let $\mathcal{O}'$ be a set of observations that will serve as *kernel centers*. Just like before, we can use these *kernel centers* to define *feature vectors* ϕ^{o_j} of observations $o_j \in \mathcal{O}$. In contrast to history feature vectors and test feature vectors, the observation feature vectors have to be *normalized* according to KDE theory. Thus, a feature vector of observation o_j in terms of *kernel centers* $o'_i \in \mathcal{O}'$ is defined as:

$$\phi^{o_j} = \begin{pmatrix} K(o'_1, o_j) \\ \cdot \\ \cdot \\ \cdot \\ K(o'_{|\mathcal{O}'|}, o_j) \end{pmatrix} \frac{1}{Z_{o_j}}$$

Here Z_{o_j} is a normalizer for observation o_j and defined as the sum over all *kernel centers*' activations induced by that observation:

$$Z_{o_j} = \sum_{o'_i in \mathcal{O}'} K(o'_i, o_j)$$

Let, similar to $\phi^{\mathcal{T}}$ and $\phi^{\mathcal{H}}$, the matrix $\phi^{\mathcal{O}} \in \mathbb{R}^{|\mathcal{O}'| \times |\mathcal{O}|}$ contain the future vectors of all observations $o_j \in \mathcal{O}$ as columns.

Adjusted Definition of $P_{\mathcal{T},ao,\mathcal{H}}$

We can make use use of the discretization defined above to define an adjusted version of $P_{\mathcal{T},ao,\mathcal{H}}$. Instead of defining matrices $P_{\mathcal{T},ao,\mathcal{H}}$ for every action-observation pair, we can now define the matrices $P_{\mathcal{T},ao',\mathcal{H}}$ for every pair of

observation *kernel center* and action.

$$[P_{\mathcal{T},ao',\mathcal{H}}]_{ij} = \mathbb{E}(\phi_i^{\mathcal{T}} \phi_j^{\mathcal{H}} \phi_{o'}^{O} | a, (\phi_i^{\mathcal{T}})^A)$$
$$\approx \frac{1}{|E_a|} \sum_{n=1}^{|E|} \beta_n K(t_i', t_n) K(h_j', h_n) K(o', o_n) \frac{1}{Z_{o_n}} \delta(a_n) \tag{2.14}$$

Each entry of $P_{\mathcal{T},ao',\mathcal{H}}$ is the expected value of the product of one test *kernel center*, one history *kernel center* and the corresponding observation *kernel center*. We can approximate it as an empirical expectation using the set of training episodes E. E_a are all those episodes where the current action is a and $\delta(a_n)$ is an indicator function. In this way, only those training episodes are taken into account, where the current action corresponds to the action of the respective $P_{\mathcal{T},ao',\mathcal{H}}$ matrix.

Learning TPSR Parameters with Features

In this part we will state formulas to learn TPSR parameters in terms of the adjusted P-statistics with features. The transformation U can still be obtained by SVD of (the adjusted) $P_{\mathcal{T},\mathcal{H}}$ as described for standard TPSRs in section 2.2. For the parameters we can recover the old formulas of TPSR parameters (see equations 2.9 - 2.11) with only slight adaptions in order to learn a TPSR with *features*. The only difference is that we need a vector $d \in \mathbb{R}^{|\mathcal{H}'|}$ so that $(\phi^{\mathcal{H}})^T d = 1_{|\mathcal{H}|}$, where $1_{|\mathcal{H}|}$ is an all ones vector of length $|\mathcal{H}|$, to define b_* and that $B_{ao'}$ matrices are now defined in terms of observation *kernel centers* $o' \in \mathcal{O}'$ instead of observations:

$$b_* = U^T P_{\mathcal{T},\mathcal{H}} d \tag{2.15}$$
$$b_\infty^T = P_{\mathcal{H}}^T (U^T P_{\mathcal{T},\mathcal{H}})^\dagger \tag{2.16}$$
$$B_{ao'} = U^T P_{\mathcal{T},ao',\mathcal{H}} (U^T P_{\mathcal{T},\mathcal{H}})^\dagger \tag{2.17}$$

Since $\phi^{\mathcal{H}}$ is a matrix of *features*, we can easily specify d if we add a constant *feature* to our set of history features. For example the last *feature* could be $\frac{1}{1000}$ regardless of the history and hence d could be a vector of zeros except for the last entry, which would be 1000.
The $B_{ao'}$ matrices defined in terms of kernel centers can be used to compute B_{ao} matrices defined in terms of observations as needed. Whenever we want to update a TPSR state and are given a current action-observation pair, we can calculate B_{ao} as a weighted combination of all $B_{ao'}$ matrices corresponding to the same action. The weights are determined by the activation of the *kernel centers* o' induced by the current observation o, which are the entries of *feature vector* $\phi(o)$. Accordingly, we can use the following formula:

$$B_{ao} = \sum_{o' \in \mathcal{O}'} B_{ao'} \phi_{o'}(o) = \sum_{o' \in \mathcal{O}'} B_{ao'} K(o', o) \frac{1}{Z_o} \tag{2.18}$$

With this knowledge, we introduced everything needed to model a dynamical system using a TPSR with *features*. We can update the state of such a TPSR just like a normal TPSR state (see equation 2.5).

Using Features: Conclusion

Boots et al. (2011) have proven that a TPSR with *features* learned in the described manner will still converge to the true underlying PSR up to a linear transform. This prove is subject to certain assumptions which we will discuss in more detail in the next section. When using features, the transformation J is factorized into three matrices. The feature matrix $\phi^{\mathcal{T}}$ is added in between R and U and projects tests' predictions to a *feature vector*, which is then mapped to a TPSR state by U as before:

$$b(h) = JQ(h) = (U^T \phi^{\mathcal{T}} R)Q(h)$$

In conclusion, features relieve us from the need to define sufficient sets of tests $\mathcal{T}$ and *indicative events* $\mathcal{H}$. Instead, we need to define sets of test *features* and history *features*. Nevertheless, these also need to be sufficient in the sense that $P_{\mathcal{T},\mathcal{H}}$ (see equation 2.13) has rank $|Q|$, which is the linear dimension of the dynamical system. But in practice it seems to be easier to ensure sufficiency with a moderate number of features compared to working with tests and *indicative events* directly (Boots et al., 2011). Furthermore, using *features* allows us to model domains with continuous observation spaces, as we can express the $B_{ao'}$ matrices in terms of observation *kernel centers* and then generate the B_{ao} parameters as needed.

2.4 From Theory to Practice: Questions and Problems

Predictive state representations form an interesting approach to modeling dynamical systems in a general way. We have seen that every dynamical system with finite linear dimension n can be modeled by a PSR wit n *core tests*. This means that we can represent the state of a dynamical system solely in terms of n probabilities of future events conditioned on the history. These n probabilities are sufficient to predict *everything* that could happen in the future if we act in certain ways and they capture information about what happened arbitrarily far back in the past. Hence, a PSR relies completely on *observable* quantities, because we can take actions and track observations in the system in order to estimate and verify such probabilities. It has been suggested that learning PSRs from data could therefore be much easier than learning latent variable models like POMDPs that reason about *non-observable* hidden states. Despite this hope, the first approaches to learn PSRs were limited to small discrete domains due to an expensive

combinatorial search for *core tests*. To learn models of more complex domains, we went all the way from TPSRs to specifying *features*. TPSRs make use of spectral decomposition and regression in order learn a linearly transformed version of a PSR in a compact subspace. *Features* allow us to deal with continuous observation spaces and extract information from training data to compress it and thus simplify learning. Using these two ideas, Boots et al. (2011) have come up with a consistent PSR learning algorithm that gives globally optimal parameters of a TPSR with *features* in closed-form. Accordingly, looking at the theory, we have an appealing approach towards modeling dynamical systems. It allows to learn compact representations directly from execution traces in a statistically consistent way.
The theory involves many assumptions, though, that are not grantable in practice. Only practical experiments can show how well we can model dynamical systems with TPSRs. As PSRs are thought as a general representation for all dynamical systems, it will especially be interesting to see how many parameters have to be tuned and how dependent these parameters are on the particular problem that is modeled. The less we have to adapt parameters to the underlying problem, the more general the approach can be used and the easier it can be implemented. Therefore, we will now look at the assumptions made and try to identify parameters and probable sources of error that stem from the discrepancies between theoretical assumptions and practical conditions. We will later analyze these problems in our experiments in chapter 4.

Number of Samples

Since we learn a representation directly from (possibly noisy) data, we will estimate the P matrices $P_{\mathcal{H}}$, $P_{\mathcal{T},\mathcal{H}}$ and $P_{\mathcal{T},ao',\mathcal{H}}$ from a finite number of samples. The law of large numbers guarantees that the estimates converge to the true matrices when we include more data (Boots et al., 2011). In the derivation of the algorithm we always assumed that our estimates are correct, but this is by no means guaranteed in practice. Inadequate estimates can lead to numerous errors. For example the rank of $P_{\mathcal{T},\mathcal{H}}$ might be higher or lower than the linear dimension of the underlying system if we have bad estimates so that we can no longer rely on the number of non-zero singular values when specifying transformation U. Thus we will have to examine the number of sampled episodes in our experiments and it will be interesting to see how many samples will be needed in practice to learn representations that enable successful reinforcement learning. Especially because gathering many samples without knowledge of the environment can be a time-consuming and costly task in real world scenarios.

Number of History and Test Kernel Centers

The algorithm assumes a sufficient set of *features* $\mathcal{T}'$ and $\mathcal{H}'$, where sufficient means that $P_{\mathcal{T},\mathcal{H}}$ has rank $|Q|$, the linear dimension of the underlying system. This is of great importance, since the transformation U depends on $P_{\mathcal{T},\mathcal{H}}$ and its rank. If the rank is smaller than Q, we cannot learn an exact TPSR.
There are many difficulties when it comes to this assumption. First of all, we can usually not verify this assumption, because the linear dimension of the modeled dynamical system is unknown. Trying to find it means to discover the *core tests* Q of the system and this is what we wanted to evade by learning a TPSR. Therefore we cannot check whether the rank equals $|Q|$ because Q is unknown. Furthermore, Kulesza et al. (2015b) stress that most real world systems have an unbounded or very high rank so that we cannot provide sets of sufficient size without exceeding the computational limits. The SVD of $P_{\mathcal{T},\mathcal{H}}$ limits us to sets of a few thousand test and history *features*. Kulesza et al. (2015b) also provide an algorithm to improve those sets to get better empirical performance (for the discrete TPSR case without *features*). Nevertheless, the assumption seems hardly satisfiable in more complex domains and we will thoroughly examine the influence of different sets of *kernel centers* on the empirical performance in our experiments.
We should also note that the *features* we define can greatly impact learning success. The choice of *features* is often domain specific and should not be underestimated. If we choose *features* that do not cover the system's pertinent properties, the assumption of sufficiency will likely not be satisfiable regardless of the number of features used. Our RBF kernel *features* for example only represent similarity between observation sequences and therefore ignore information about the actions specified in tests and histories. Using this kind of *features* connotes that we think (or hope) that the properties of the dynamical system are apparent in the observation sequences.

Number of Observation Kernel Centers

Similarly to the number of *kernel centers* for tests and histories, the observation kernel centers also play an important role. They will not influence the rank of $P_{\mathcal{T},\mathcal{H}}$, but the accuracy of B_{ao} parameters computed by $B_{ao'}$ matrices. In order to get accurate B_{ao} parameters, we need to model the observation *probability density function* (PDF) with *kernel density estimation* (KDE) as well as possible. According to KDE theory the KDE estimator converges to the observation PDF as the number of *kernel centers* goes to infinity and the *kernel width* approaches zero (Silverman, 1986). Of course we cannot use infinitely many observation *kernel centers*, but we will need enough centers to sufficiently model the observation PDF. On the other hand, we need to store $|\mathcal{A} \times \mathcal{O}'|$ $B_{ao'}$ matrices and calculating these ma-

trices is the second computationally expensive part besides the SVD. This means that we are also limited in the number of observation *kernel centers* and found another parameter that we will have to tune and analyze in our experiments in chapter 4.

Chapter 3

Implementation

The algorithm that we re-implemented and will describe in this chapter is basically the same as found in Boots et al. (2011). For this documentation we decided to stick as close to our actual implementation as possible. Therefore, the pseudo code given here will have minor differences to the general algorithm presented in the original paper and we will specifically note these slight changes to avoid any confusions. Furthermore, besides providing the core algorithm that uses spectral decomposition and regression to learn the TPSR parameters (section 3.2), we will also provide pseudo code and guidance to all the other steps involved in learning a TPSR with *features* from execution traces. This includes the gathering and preprocessing of data, covered in section 3.1, and instructions on how to use the learned TPSR in order to track states (section 3.3). In the end, we will summarize all the tunable parameters arising from this specific implementation in section 3.4. We hope to provide a solid starting point for future practical work with (T)PSRs by giving as much details as possible in this chapter. After each step there will be a table that summarizes the matrices that we just computed and that we will need for further computations.

3.1 Data Gathering and Preprocessing

In this section we will show how to gather training data, whiten it, choose kernel centers and calculate features. The goal of gathering data and preprocessing it will be to get matrices of *features* which allow us to estimate $P_{\mathcal{H}}$, $P_{\mathcal{T},\mathcal{H}}$, and $P_{\mathcal{T},ao',\mathcal{H}}$ in the learning algorithm. We first need to sample action-observation sequences, cut these into episodes of histories, current action-observation pair, and tests, and then extract features from these quantities. The RBF kernel features we use require the data to be whitened. For this chapter we assume that our observations are k-dimensional real vectors, thus $\mathcal{O} = \mathbb{R}^k$, and actions are a finite set of single numbers: $\mathcal{A} \subset \mathbb{Z}^1$.

Gathering Data

As already mentioned in section 2.2, we can gather data in multiple ways. For our experiments we assumed that reset is not available and gathered one long trajectory of action-observation pairs by taking actions uniformly randomly. Then we split the trajectory into episodes. The set of episodes will be called E. Due to our choice of features, we are limited to histories of same length and tests of same length. Thus, we can have histories of length m and tests of length n. Then one episode would consist of m action-observation pairs that form a history, followed by one current action-observation pair, followed by n action-observation pairs that form a test. We decided to slice the trajectory into episodes that do not overlap. Therefore the first $m+1+n$ action-observation pairs form the first episode, the following $m+1+n$ pairs form the second episode, and so on. In a scenario where there is only a very short trajectory available, we could also try to slice overlapping episodes to get more training samples, which would be more correlated, though.
Now we can form three matrices of *observations*. $\mathcal{H} \in \mathbb{R}^{|E|\times(m\cdot k)}$ consists of the observations from all sampled histories, $\mathcal{T} \in \mathbb{R}^{|E|\times(n\cdot k)}$ consists of the observations from all sampled tests and $O \in \mathbb{R}^{|E|\times k}$ consists of all the current observations. Here the rows correspond to the episode where the history, test, or current observation is taken from. The columns are vectors of k-dimensional observations which are stacked for histories and tests if these are longer than one action-observation pair. As our features are defined solely in terms of observations, we do not need to store the actions taken in tests and histories. For further calculations we only need one matrix of *actions* $A \in \mathbb{R}^{|E|\times 1}$ that includes the *current action* of each episode.

Matrix	Dimension	Description
$\mathcal{H}$	$\lvert E\rvert \times (m \cdot k)$	observations of gathered histories
$\mathcal{T}$	$\lvert E\rvert \times (n \cdot k)$	observations of gathered tests
O	$\lvert E\rvert \times k$	gathered current observations
A	$\lvert E\rvert \times 1$	gathered current actions

Table 3.1: Output after gathering a set E of training episodes with k-dimensional observations, histories of length m and tests of length n.

Whitening

Working with RBF kernels usually involves whitening of the training data. Whitening means to transform the data matrices so that the different data point dimensions show covariance of zero and a variance of one. We will make use of *principal component analyses* (PCA) in order to whiten our data. Algorithm 1 will take a $r \times b$ matrix X containing r data points of b dimensions and a parameter d^X that describes how many dimensions

we want to keep after whitening. Thus, we also use whitening to reduce the dimensionality of our data points. We will keep the d^X dimensions corresponding to the largest eigenvalues of our data. The algorithm returns a whitened $r \times d^X$ matrix X_{white} of the data, the mean of all data points $mean^X \in \mathbb{R}^{1\times b}$, and a transformation matrix $W^X \in \mathbb{R}^{b\times d^X}$. It holds:

$$X_{white} = (X - mean^X)(W^X)$$

Here and in the algorithm $X - mean^X$ subtracts $mean^X$ from each row in X so that the data is centered.

Algorithm 1 Data Whitening

In: $X_{1:r}$ matrix with r data points as rows, d^X number of dimensions to keep

Out: $X_{white}, mean^X, W^X$

$mean^X = \frac{1}{r}\sum_{i=1}^{r} X_i$
$X_{centered} = X - mean^X$ ▷ subtract mean from each row
$\langle U, eigenvalues\rangle = \text{PCA}(X_{centered})$
if $d^X <$ original data dimension b **then**
    $eigenvalues = d$ largest $eigenvalues$
    $U =$ eigenvectors of d largest $eigenvalues$
$S = \text{diag}(\frac{1}{\sqrt{eigenvalues}})$
$W^X = US$
$X_{white} = X_{centered}W^X$

We use this procedure to whiten the three observation matrices $\mathcal{H}$, $\mathcal{T}$, and O. The parameter d can be chosen at will or linked to the magnitude of the dropped or kept eigenvalues. We will explain our choice of parameters along with our experiments in chapter 4. Besides the whitened data, we also need to keep the mean and whitening transformation of the *current observations* for later calculations when updating the state.

Matrix	Dimension	Description
$\mathcal{H}_{white}$	$\lvert E\rvert \times d^{\mathcal{H}}$	whitened observations of gathered histories
$\mathcal{T}_{white}$	$\lvert E\rvert \times d^{\mathcal{T}}$	whitened observations of gathered tests
O_{white}	$\lvert E\rvert \times d^{O}$	whitened gathered current observations
$mean^O$	$1 \times k$	mean of current observation data points
W^O	$k \times d^O$	whitening transformation for current observations

Table 3.2: Output after whitening of the three observation matrices when keeping d of k original dimensions. d can be chosen for histories, tests, and current observations individually.

Feature Selection

As we want to make re-implementation as easy as possible, we will explain how to generate the *features* that we used for our experiments. The *features* we used are RBF kernel *features* and we already briefly introduced them in section 2.3. Nevertheless, we should note that the learning algorithm is not limited to this kind of features (Boots et al., 2011).
Our RBF kernel *features* only indicate the similarity of observations. Therefore we will take some of the observations from $\mathcal{H}_{white}$, $\mathcal{T}_{white}$, and O_{white} as *kernel centers* and one *feature* will consist of the similarity between the *kernel center* and an observation (or observation sequence for tests and histories longer than one). To be precise we take the first $l^{\mathcal{H}}$ rows of $\mathcal{H}_{white}$ as *history kernel centers*, the first $l^{\mathcal{T}}$ rows of $\mathcal{T}_{white}$ as *test kernel centers*, and the first l^{O} rows of O_{white} as *observation kernel centers*. Lets call these sub-matrices $\mathcal{H}^{centers}$, $\mathcal{T}^{centers}$, and $O^{centers}$. The remaining $t = |E| - max\{l^{\mathcal{H}}, l^{\mathcal{T}}, l^{O}\}$ rows of the whitened observation matrices serve as training data $\mathcal{H}^{train}$, $\mathcal{T}^{train}$, and O^{train}. Thus the training data matrices contain the whitened observations of the last t episodes as rows.
Algorithm 2 shows how to compute a *feature matrix* ϕ^X given $X^{centers}$ and X^{train}. Each entry in ϕ^X is the activation of one *kernel center* induced by one training data sample, where the rows correspond to centers and the columns correspond to training data samples. Additionally, we have to provide the algorithm with a kernel width σ^X, a boolean *normalize* that indicates whether we normalize the feature vectors, and a boolean *add_constant* that indicates whether we add a constant feature.
Using this feature extraction algorithm, we compute *feature matrices* $\phi^{\mathcal{H}}$ of histories, $\phi^{\mathcal{T}}$ of tests, and ϕ^{O} of current observations using our split whitened data sets of centers and training samples. We chose kernel widths by trial and error (see chapter 4). Since we want to model the observation probability density function with kernel density estimation (see section 2.3), we need to normalize the *feature vectors* of current observations in ϕ^{O}. The *feature matrices* $\phi^{\mathcal{H}}$ of histories and $\phi^{\mathcal{T}}$ of tests *do not necessarily* need to be normalized. In the experiments of the paper where this learning algorithm was proposed the authors *did* normalize these matrices as well (Boots et al., 2011). For our experiments, we *did not* normalize history and test *feature vectors*, as it gave us more consistent empirical results (see chapter 4).
Furthermore, we need to add a constant feature to the *history features* to ensure the constraint that $(\phi^{\mathcal{H}})^T e = 1_t$ with vector $e \in \mathbb{R}^{(l^{\mathcal{H}}+1)\times 1} = [0, 0, ...0, z]^T$. Here $1_t \in \mathbb{R}^{t\times 1}$ is a vector of ones. The value for the constant feature is chosen arbitrarily and can be changed in the algorithm and e accordingly. We used $\frac{1}{1000}$ as constant feature and $z = 1000$ as last entry in e in our experiments. Although not required in theory, we also added the constant feature to our *test features*. Empirically, this didn't influence the learned representations, but provided a simple way to maintain symmetry

between the dimensions of $\phi^{\mathcal{H}}$ and $\phi^{\mathcal{T}}$. Symmetry is not mandatory in this case, but simplified some of our experiments.

Algorithm 2 Feature Extraction

In: $X^{centers}$, X^{train}, σ^X, *normalize*, *add_constant*
Out: ϕ^X

for row index $j \in X^{train}$ **do** ▷ iterate over training data
  $sum = 0$
  for row index $i \in X^{centers}$ **do** ▷ iterate over kernel centers
    $dist = ||X_i^{centers} - X_j^{train}||^2$ ▷ squared euclidean distance
    $\phi_{i,j}^X = exp(-\frac{dist}{2(\sigma^X)^2})$ ▷ kernel activation
    $sum = sum + \phi_{i,j}^X$ ▷ sum over centers
  if *normalize* **then**
    for row index $i \in X^{centers}$ **do** ▷ iterate over kernel centers
      $\phi_{i,j}^X = \frac{\phi_{i,j}^X}{sum}$ ▷ normalize activation
  if *add_constant* **then**
    $\phi_{i+1,j}^X = \frac{1}{1000}$ ▷ add additional constant feature

Matrix	Dimension	Description
$\phi^{\mathcal{H}}$	$(l^{\mathcal{H}}+1) \times t$	unnormalized history feature matrix (with additional constant feature)
$\phi^{\mathcal{T}}$	$(l^{\mathcal{T}}+1) \times t$	unnormalized test feature matrix (with additional constant feature)
ϕ^O	$l^O \times t$	normalized feature matrix of current observations
$O^{centers}$	$l^O \times d^O$	observation kernel centers
σ^O	constant value	kernel width for observation kernel centers
e	$(l^{\mathcal{H}}+1) \times 1$	vector so that $(\phi^{\mathcal{H}})^T e = 1_t$

Table 3.3: Output after replacing our data with feature matrices. l is the number of chosen samples as *kernel centers* and can be specified for histories, tests, and current observations individually. $t = |E| - max\{l^{\mathcal{H}}, l^{\mathcal{T}}, l^O\}$ is the number of remaining training samples.

3.2 Simple Batch Learning Algorithm

In this section we will describe our implementation of the TPSR batch learning algorithm from Boots et al. (2011). It consists of the following steps: First, it estimates the P-statistics $P_{\mathcal{H}}$, $P_{\mathcal{T},\mathcal{H}}$, and $P_{\mathcal{T},ao,\mathcal{H}}$, then it performs singular value decomposition (SVD) on $P_{\mathcal{T},\mathcal{H}}$ to find the TPSR subspace transformation U, and finally it computes the remaining TPSR parameters b_*, b_∞, and $B_{ao'}$. The last step also leverages the SVD of $P_{\mathcal{T},\mathcal{H}}$ as the decomposition simplifies the parameter equations. We will introduce these easier formulas after recalling the meaning of the P-statistics and noting some slight changes that we made to the original algorithm concerning normalization. At the end, pseudo code wraps up the TPSR learning steps.

P-statistics in Terms of Feature Matrices

In the following we make use of equations 2.12 - 2.14 from the theory part which define the P-statistics in terms of feature matrices. Since we gather our data uniformly randomly, all actions are equally likely to be taken and thus the importance weights β in these equations are constant by design and cancel out. Therefore, we can estimate the P-statistics using *feature matrices* $\phi^{\mathcal{H}}$ of histories, $\phi^{\mathcal{T}}$ of tests, and ϕ^{O} of current observations as follows. The hat of $\widehat{P}$ marks that it is an empirical estimate of P.

$$\widehat{P}_{\mathcal{H}} = \frac{1}{t}\sum_{i=1}^{t} \phi^{\mathcal{H}}_{:,i} \tag{3.1}$$

$\phi^{\mathcal{H}}_{:,i}$ is column i of matrix $\phi^{\mathcal{H}}$. Therefore, $\widehat{P}_{\mathcal{H}} \in \mathbb{R}^{(l^{\mathcal{H}}+1)\times 1}$ is the mean of all columns of $\phi^{\mathcal{H}}$ and thus consists of the empirical expected value of each *history feature*.

$$\widehat{P}_{\mathcal{T},\mathcal{H}} = \frac{1}{t}\phi^{\mathcal{T}}(\phi^{\mathcal{H}})^T \tag{3.2}$$

$\widehat{P}_{\mathcal{T},\mathcal{H}} \in \mathbb{R}^{(l^{\mathcal{T}}+1)\times(l^{\mathcal{T}}+1)}$ consists of the empirical expected values of the product of one *test feature* and one *history feature*.
The $\widehat{P}_{\mathcal{T},ao',\mathcal{H}} \in \mathbb{R}^{(l^{\mathcal{T}}+1)\times(l^{\mathcal{T}}+1)}$ matrices for each action $a \in \mathcal{A}$ and observation *kernel center* $o' \in O^{centers}$ consist of the empirical expected value of the product of one *test feature*, one *history feature*, and one *observation feature* when the current action is a. Therefore, we filter our set of current actions A so that Y_a is the set of all indices where the current action is a and t_a is the number of those indices $|Y_a|$. $\phi^{\mathcal{H}}_{:,i}$ and $\phi^{\mathcal{T}}_{:,i}$ give us the i-th columns which correspond to training sample i and $\phi^{O}_{o',i}$ gives us an entry that corresponds to *kernel center* o' and training sample i. Then the $\widehat{P}_{\mathcal{T},ao',\mathcal{H}}$ matrices can

be calculated as follows:

$$\widehat{P}_{\mathcal{T},ao',\mathcal{H}} = \frac{1}{t_a} \sum_{i \in Y_a} \phi^{\mathcal{T}}_{:,i} (\phi^{\mathcal{H}}_{:,i})^T \phi^{O}_{o',i} \tag{3.3}$$

Normalizing Empirical Expectations

In contrast to the algorithm in Boots et al. (2011), we normalized our estimates of $\widehat{P}_{\mathcal{T},ao',\mathcal{H}}$ and $\widehat{P}_{\mathcal{T},\mathcal{H}}$. This is not necessary and in fact, we can come along without normalizing any of the empirical estimates as the normalization factors cancel out in the calculation of parameters (Boots and Gordon, 2011). But we noticed that they influence the transformation U in the sense that it scales the codomain of U^T and the magnitude of singular values of $\widehat{P}_{\mathcal{T},\mathcal{H}}$. In the end, both versions will lead to equally good predictions as long as no numerical issues occur. In line 8 of algorithm 3 we invert the singular values. Here very high or very low values can lead to numerical instability in further calculations and thus it is important to keep the singular values in a reasonable domain. Hamilton (2014) suggested to aim for singular values around 1 to circumvent problems and by normalizing we usually had singular values close to 1 or higher, but still stably invertible, depending on the chosen kernel widths. When normalizing the *features* of histories and tests, also normalizing the $\widehat{P}$ matrices will lead to rather low singular values. Thus, whether to normalize the empirical expectations or not strongly depends on the form of $\phi^{\mathcal{H}}$ and $\phi^{\mathcal{T}}$ and therefore on the choice of features.

SVD for Simple TPSR Parameter Learning

In our computations we will be using the *compact* singular value decomposition (SVD_c) of $\widehat{P}_{\mathcal{T},\mathcal{H}}$. If a matrix has dimensionality $m \times n$ and rank r so that $r \leq m \leq n$ or $r \leq n \leq m$, its compact SVD will give a vector $s \in \mathbb{R}^r$ of r positive singular values in decreasing order, a $n \times r$ matrix U with the corresponding r orthogonal left singular vectors as columns, and a $m \times r$ matrix V with the corresponding r orthogonal right singular vectors as columns (referred to as third SVD form in Cline and Dhillon (2013)). Then S is the $r \times r$ matrix with singular values s on its diagonal and it holds $\widehat{P}_{\mathcal{T},\mathcal{H}} = USV^T$. Not only will U be our TPSR subspace transformation, but we can also use this compact SVD to easily compute the TPSR parameters. It is particularly helpful because we can easily compute a pseudo inverse of $U^T \widehat{P}_{\mathcal{T},\mathcal{H}}$. Note that $S^T = S$ because S is diagonal. Since the columns in V and U are orthogonal, $V^T V$ and $U^T U$ are identity matrices. Thus we can write:

$$\begin{aligned}(U^T \widehat{P}_{\mathcal{T},\mathcal{H}})^\dagger &= (U^T U S V^T)^\dagger = (SV^T)^\dagger = (SV^T)^T (SV^T (SV^T)^T)^{-1} \\ &= VS(SV^T VS)^{-1} = VS(SS)^{-1} = VSS^{-1}S^{-1} = VS^{-1}\end{aligned} \tag{3.4}$$

As S is a diagonal matrix, its inverse S^{-1} can be computed straight forward by taking the reciprocals of the singular values on its diagonal. With this knowledge, we can easily compute the TPSR parameters using the following equations:

$$\begin{aligned} b_* &\overset{2.15}{=} U^T P_{\mathcal{T},\mathcal{H}} e \\ &= U^T U S V^T e \\ &= S V^T e \end{aligned} \tag{3.5}$$

$$\begin{aligned} b_\infty^T &\overset{2.16}{=} P_{\mathcal{H}}^T (U^T P_{\mathcal{T},\mathcal{H}})^\dagger \\ &\overset{3.4}{=} P_{\mathcal{H}}^T V S^{-1} \end{aligned} \tag{3.6}$$

$$\begin{aligned} B_{ao'} &\overset{2.17}{=} U^T P_{\mathcal{T},ao',\mathcal{H}} (U^T P_{\mathcal{T},\mathcal{H}})^\dagger \\ &\overset{3.4}{=} U^T P_{\mathcal{T},ao',\mathcal{H}} V S^{-1} \end{aligned} \tag{3.7}$$

The whole procedure to calculate the TPSR parameters is summarized in algorithm 3. Instead of keeping all singular values greater than zero, we can also decide to keep less singular values and crop s, U and V accordingly. Therefore, we provide a *rank* parameter r and keep only the r largest singular values. This will possibly result in a loss of prediction accuracy. Kulesza et al. (2014) have shown that even omitting the smallest singular value(s) can lead to arbitrarily high prediction errors. We will further reason about the rank parameter in our experiments in chapter 4.

Algorithm 3 TPSR Batch Learning

In: $\phi^{\mathcal{H}}$, $\phi^{\mathcal{T}}$, ϕ^{O}, $O^{centers}$, e, r, $\mathcal{A}$ (all actions), A (matrix of current actions)
Out: U, b_*, b_∞, $B_{ao'}$

$\widehat{P}_{\mathcal{H}} = \frac{1}{t} \sum_{i=1}^{t} \phi_{:,i}^{\mathcal{H}}$ ▷ mean of all columns
$\langle U, s, V \rangle = SVD_c(\frac{1}{t} \phi^{\mathcal{T}} (\phi^{\mathcal{H}})^T)$ ▷ compact SVD of $P_{\mathcal{T},\mathcal{H}}$
if $r <$ number of singular values in s **then**
    $s =$ first r singular values in s ▷ take largest r singular values
    $U =$ first r columns of U ▷ and associated left singular vectors
    $V =$ first r columns of V ▷ and associated right singular vectors
$S = \text{diag}(s)$ ▷ singular values on diagonal
$S^{-1} = \text{diag}(\frac{1}{s})$ ▷ reciprocals on diagonal
$b_* = S V^T e$ ▷ equation 3.5
$b_\infty^T = P_{\mathcal{H}}^T V S^{-1}$ ▷ equation 3.6
for $a \in \mathcal{A}$ **do** ▷ iterate over actions
    $Y_a = \text{where}(A == a)$ ▷ indices of episodes with current action a
    for $o' \in O^{centers}$ **do** ▷ iterate over observation kernel centers
        $B_{ao'} = \frac{1}{t_a} \sum_{i \in Y_a} U^T (\phi_{:,i}^{\mathcal{T}} (\phi_{:,i}^{\mathcal{H}})^T \phi_{o',i}^{O}) V S^{-1}$ ▷ equations 3.3 & 3.7

Matrix	Dimension	Description
U	$(l^{\mathcal{T}}+1) \times r$	transformation matrix that defines the subspace of the TPSR
b_*	$r \times 1$	feasible starting state of the TPSR
b_∞	$r \times 1$	normalizer of the TPSR
$B_{ao'}$	$r \times r$	update parameters of the TPSR, one matrix for each action-observation-kernel-center pair

Table 3.4: Output after learning the TPSR parameters. r is the rank we chose for our TPSR or the number of non-zero singular values if we did not limit it.

3.3 Tracking States

In this work we aim to use the learned representation to solve a reinforcement learning task. In order to use our representation for planning, we need to be able to update the system's TPSR state b_s at time step s whenever we take an action a_s and note a new observation o_s. Using equation 2.5, the recursive state update from time step s to $s+1$ is given as (Boots et al., 2011):

$$b_{s+1} = \frac{B_{a_s o_s} b_s}{b_\infty^T B_{a_s o_s} b_s} \tag{3.8}$$

The base case for the first action observation pair a_1, o_1 simply uses the starting state b_*. As described in equation 2.18, we can calculate the $B_{a_s o_s}$ matrix as a weighted sum over the $B_{a_s o'}$ matrices, that we calculated during the TPSR batch learning algorithm, for each observation kernel center $o' \in O^{centers}$:

$$Ba_s o_s = \sum_{o' \in O^{centers}} \phi_{o'}^{o_s} B_{a_s o'} \tag{3.9}$$

Here $\phi_{o'}^{o_s}$ is entry o' of feature vector ϕ^{o_s} of the current observation and weights the sum. It is the activation of kernel center o' induced by the new observation o_s.

Algorithm 4 summarizes how to update the state given the new action-observation pair a_s, o_s, the current TPSR state b_s, the TPSR parameters, and the observation kernel centers and width $O^{centers}$ and σ^O. We also need to provide the whitening parameters to project the new observation into the space of whitened current observations. Otherwise we could not calculate the features of the new observation, which we need to weight the sum when calculating $B_{a_s o_s}$. If the system was just started and did not track any action-observation pairs yet, the current state will be the starting state b_*. We assume that the current observation $o_s \in \mathbb{R}^{1 \times k}$ is given as a *row vector*.

Algorithm 4 TPSR State Update

In: a_s, o_s, b_s, b_∞, $B_{a_s o'}(\forall o' \in O^{centers})$, $O^{centers}$, σ^O, $mean^O$, W^O
Out: b_{s+1}

1: $o_s^{white} = (o_s - mean^O)W^O$ ▷ whiten observation
2: $\phi^{o_s^{white}} = \text{extract\_features}(O^{centers}, o_s^{white}, \sigma^O, normalize)$
▷ compute normalized feature vector, see algorithm 2
3: $B_{a_s o_s} = \sum_{o' \in O^{centers}} \phi_{o'}^{o_s^{white}} B_{a_s o'}$ ▷ compute update parameter (eq.3.9)
4: $b_{s+1} = \frac{B_{a_s o_s} b_s}{b_\infty^T B_{a_s o_s} b_s}$ ▷ compute new state(eq.3.8)

3.4 Important Parameters

Using the algorithms and steps described in this chapter there are a lot of different parameters that have to be chosen during the learning of a TPSR. With regard to the next chapter where we will present our experimental results, we compactly summarize all the parameters from above in table 3.5. For our experiments, we coupled some of the parameters. For example, histories and tests will always have the same length. Therefore, in the following table the column '*In Experiments*' shows how we named and possibly coupled the parameters for our experiments.
Coupling parameters shrinks the search space when looking for appropriate settings but can also prevent successful learning if parameters are coupled that need to be tuned individually. For example we should not couple the kernel width for current observations with the width for histories and tests because both sets have completely different theoretical demands. The feature matrices of histories and tests should capture the correlations in the data to pass on information about the system dynamics. Thus, we found rather high kernel widths for tests and histories so that the different features correlate were effective compared to lower kernel widths for current observations. The reason for this is that the features of current observations model the observation probability density function and therefore require rather narrow kernel widths. For more details on settings we used and relations we found, see chapter 4.
We coupled the length of histories and tests as they are only defined in terms of observation sequences due to our choice of features and therefore pretty similar. Furthermore, they both have the same sufficiency criterion that the rank of $P_{\mathcal{T},\mathcal{H}}$ equals the linear dimension of the system. Consequently, we also coupled the kernel widths and the number of dimensions to keep after whitening for histories and tests.

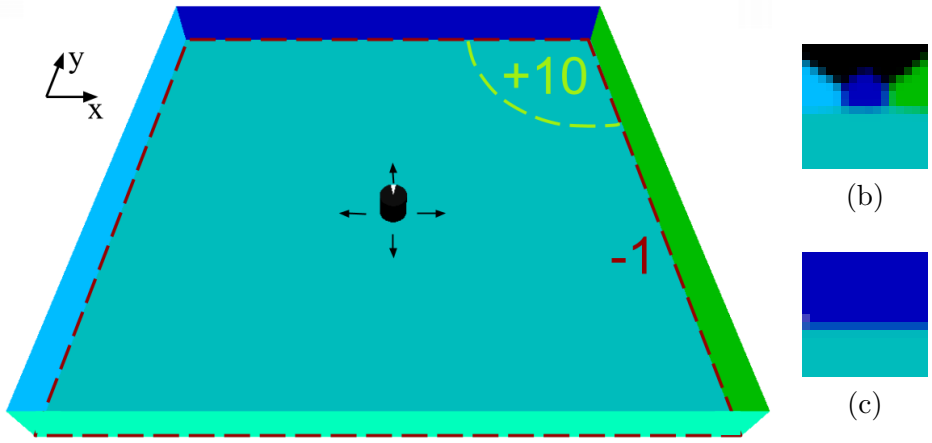
Parameter	Description	In Experiments
m	length of histories	h/t-length
n	length of tests	
$\|E\|$	number of episodes (1 history + current action-observation pair + 1 test) to gather (kernel centers and training data combined)	episodes
d^O	number of dimensions to keep after whitening current observations	obs-dim
$d^{\mathcal{H}}$	number of dimensions to keep after whitening histories	h/t-dim
$d^{\mathcal{T}}$	number of dimensions to keep after whitening tests	
l^O	number of observation kernel centers	obs-centers
$l^{\mathcal{H}}$	number of history kernel centers	h/t-centers
$l^{\mathcal{T}}$	number of test kernel centers	
σ^O	kernel width for observation features	σ^{obs}
$\sigma^{\mathcal{H}}$	kernel width for history features	$\sigma^{h/t}$
$\sigma^{\mathcal{T}}$	kernel width for test features	
r	dimension of the learned TPSR, defined by the number of singular values to keep after compact SVD of $P_{\mathcal{T},\mathcal{H}}$	dim

Table 3.5: All the parameters that have to be chosen during learning of the TPSR. The last column shows how we will refer to the parameters when presenting experimental results in chapter 4.

Chapter 4

Experimental Results

As we discussed in section 2.4, there are several theoretical assumptions that lead to concerns about the performance of PSRs in practice. In this chapter we present empirical results of different experiments to evaluate the practical applicability. Similar to the results of Boots et al. (2011), who proposed the used learning algorithm, we evaluate the quality of a learned model by analyzing the state space visually and by solving a planning task with reinforcement learning. But rather than just presenting results of one successfully learned TPSR, we will also concentrate on the empirical performance of the learning algorithm itself. Since Boots et al. (2011) have already shown successful planning in a learned model, the interesting question to us is how much effort is needed in order to learn such useful representations. This involves the amount of data needed, the effort into finding appropriate parameter settings, and the repeatability of successful learning attempts. By investigating these properties we want to provide further insights into the practical applicability of PSRs in complex domains and determine possible problems and questions that encourage future work. Therefore, section 4.2 will not only analyze the quality of a learned representation, but also include comparisons to representations learned using principal component analysis (PCA) and provide empirical results of many learning attempts with different amounts of data. Section 4.3 is dedicated to the challenge of finding appropriate parameter settings. It is thought of as a helpful resource for re-implementation and also shows possible aspects that could be improved in future work. A last line of experiments described in section 4.4 will examine the quality of the learned model by testing the memory of a TPSR in a partially observable domain.
Before we present the experimental results, we will describe the experimental environment in the following section 4.1. This includes the dynamical system that was learned and the reinforcement learning algorithm used to plan in the learned representation.



(a) (b) (c)

Figure 4.1: Simulated mobile robot environment with negative reward for hitting walls and positive reward in the top right-hand corner (a). The robot's current observations for the fully observable version with 300° field of view (b) and the partially observable version with only 60° field of view (c). (Own version of figure 3 in Jonschkowski and Brock (2015))

4.1 Experimental Environment

For our experiments we simulated a simple mobile robot navigation task similar to that in the work of Boots et al. (2011). But in contrast to the value iteration algorithm used for reinforcement learning there, we used a form of fitted Q-iteration for continuous state spaces. Both, the learning algorithm and the simulated environment are taken from Jonschkowski and Brock (2015), where they were used to learn state representations with robotics specific prior knowledge. In the following, we will first introduce the simulated environment and then proceed to describe the policy learning algorithm.

Mobile Robot Environment

The environment can be seen in figure 4.1 and consists of a simulated mobile robot in a squared room with differently colored walls. The size of the room is 45×45 units, the walls are 4 units high, and the robot has a height and diameter of 2. The only input available to the robot are 16×16 *pixel camera images* of RGB-values. Therefore, one observation is $16 \cdot 16 \cdot 3 = 768$ dimensional. By adjusting the field of view of the camera lens, we created a fully observable and a partially observable environment. In the fully observable case the robot's camera has a field of view of 300° (like in the experiments given in Jonschkowski and Brock (2015)), whereas it is restricted to only 60°

for the partially observable environment. Example observations for both settings can be found in figure 4.1.
The task of the robot is to learn to reach the upper right-hand corner of the room without hitting walls. Therefore, it gets a positive reward of 10 if that corner is closer than 15 units and negative reward of -1 for colliding with any wall. All remaining positions yield zero reward.
The robot's orientation is fixed and it can control its up-down and left-right velocity by 0, ± 3, or ± 6 units per time step to move throughout the room. Hence the robot can choose from 25 different actions. We add Gaussian noise with 0 mean and standard deviation of 10% of the chosen velocity to the actions to simulate uncertainty in movement execution. Aside from the narrowed field of view for the partially observable environment, these are the exact same settings as used in Jonschkowski and Brock (2015).

Reinforcement Learning

To determine a policy that the robot can follow greedily, we learn a Q-function. That is a function which assigns a value to each state-action pair. The higher the value, the higher the expected future reward if taking the particular action in the particular state. Hence, if a is an action and b is a state, our policy is given as:

$$\pi(b) = \underset{a}{\operatorname{argmax}}[Q(a, b)]$$

We use a fitted Q-iteration algorithm that utilizes RBF kernel features to deal with continuous state spaces. It is exactly the same as has been used in the experiments of Jonschkowski and Brock (2015).
The algorithm needs to be provided with sample trajectories of states, actions, and rewards. Therefore, we use the learned TPSR parameters to track a single long trajectory gathered in our environment. This will generate a sample state space of our TPSR. The trajectories we use will be several thousand steps long and therefore we demand high quality TPSR parameters that do not drastically drift. We think that this is a mandatory property of the parameters because real world problems often need the ability of tracking for a long period with reliable results. Another possible solution would be to only track for the length of one or several episodes and provide shorter trajectories to the Q-iteration algorithm.

4.2 Quality of Learned Representations

In this section we examine the performance of the learning algorithm described in chapter 3 by analyzing the quality of learned representations. We compare learned TPSRs to representations learned by PCA. To this end, we analyze plots of the learned representations and show that they capture the pertinent properties of the environment. Then we statistically *measure* the quality of the learned representations by comparing their reward scores after learning a policy for the mobile navigation task. We conducted experiments in three different versions of the mobile navigation task. We used the fully observable and partially observable standard versions as described above. Additionally we set up a harder version of the fully observable environment by including task irrelevant distractors. In the simple fully observable environment both approaches lead to pleasant results. But we will show that TPSRs are also sufficient to learn decent policies in the two more complex environments that can hardly be captured by the observation state mappings of PCA. Before presenting the results for the three different environments, we describe the general experimental setup and how we populated example state spaces for the reinforcement learning algorithm.

Experimental Design

In order to gather training data for the state representation learning (SRL), we let the robot take a few thousand actions uniformly randomly. In this way we created one long trajectory of action-observation pairs. The number of steps taken has been varied throughout the experiments to determine how the algorithm improves when using more input data. The robot then learned a TPSR (or PCA) model using the gathered data as described in chapter 3. The exact settings, like the number of kernel centers or the kernel width σ, can be found in a table that we will provide for each line of experiments. Afterwards we tracked the state of the robot using a *separately* (uniformly randomly) sampled trajectory to generate sample states that we could plot and use as input for the reinforcement learning (RL) algorithm. We will refer to this second trajectory as RL trajectory in contrast to the first SRL training trajectory used to learn the model parameters. These extra RL trajectories were only sampled *once* prior to starting the experiments. Therefore, throughout all the experiments in this section, we used the same RL trajectory to sample a state space from learned TPSRs and PCA models. More details on the generation of the sample state space and the used RL trajectory can be found after this subsection.
The sample state space was used to learn a policy with the Q-iteration algorithm. We then set the robot to ten different starting positions, let it follow the learned policy greedily for 50 steps, and noted the mean reward over these ten test trials.

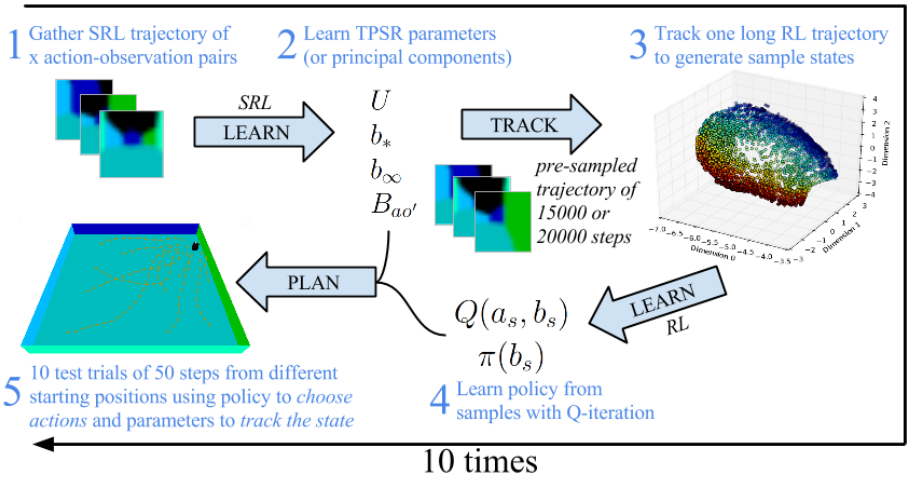


Figure 4.2: Experimental design to determine the quality of learned TPSR representations. We varied the amount x of training steps between 6000 and 24000. For each particular value of x the whole learning-planning loop is repeated 10 times. We will examine visualizations of the learned representations by plotting the sample states from step 3 and measure the quality by taking the mean, standard error of the mean, and median of the reward over the repeated ten complete learning-planning trials. *SRL* and *RL* mark the *state representation learning* and *reinforcement learning* step, respectively.

This procedure was done ten times for each amount of training data. This means that we for example learned ten TPSRs using ten different trajectories of 6000 steps. From each of the ten TPSRs we got a mean reward score over ten 50 steps trials after learning a policy. We then computed the mean and the median of the mean reward scores of these ten complete learning-planning iterations. Additionally, we computed the standard error of the mean. A pictorial summary of this process can be found in figure 4.2.

Generating a Sample State Space

As mentioned above, we need to generate a sample state space as input for the Q-iteration algorithm. In the following we will give more details about this procedure.
The length of the used RL trajectory was either 15000 steps for the fully observable case or 20000 steps for the harder version with distractors and the partially observable environment. We took a high number of steps as we wanted to make sure that the RL algorithm did not impair the learned policy. To find this number we tested the Q-iteration algorithm using the true underlying position as input. Since the true underlying position would be a

perfectly suitable two-dimensional representation for the mobile navigation task, it can be taken as a baseline. Using this perfect representation, the algorithm needed about 10000 steps to reliably learn flawless policies. Thus we decided to allow 15000 to 20000 steps for our learned representations to make sure that worse reward scores indicate inadequate representations rather than bad performance of the Q-iteration algorithm.

In the original work of Boots et al. (2011) the sample state space for the reinforcement learning method was generated by projecting histories with the TPSR transformation U. We suggest that this is not the general way to generate state samples and recommend to use the parameters instead. There are two main reasons for this.

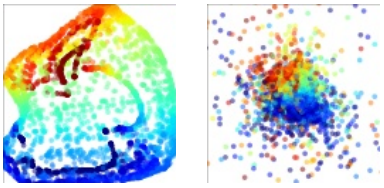
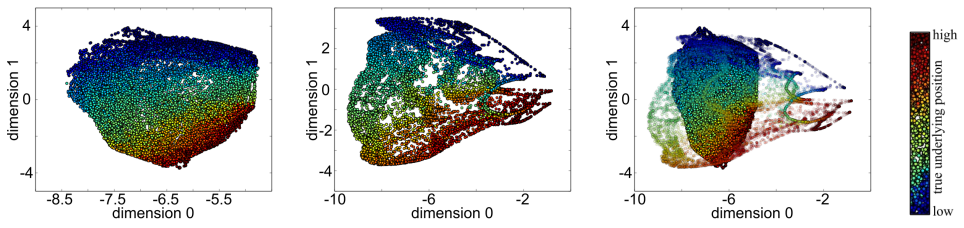


Figure 4.3: Left: Embedded histories with TPSR transformation matrix U (smooth state space) Right: Using learned TPSR update parameters (degenerated state space)

First, we found that embedding the histories with U will in many cases yield a smooth representation although the parameters might be ill-conditioned and lead to degenerated representations. An example where two such representations are compared is given in figure 4.3. In this case the learned policy will be useless as the ill-formed TPSR parameters can usually not be used to track states.
Second, even though the embedded histories sometimes approximate the learned state space, we often encountered TPSRs where the state space generated by updating with parameters was differently shaped than the space of embedded histories (see figure 4.4). Here using the embedded histories as sample state space leads to an inferior policy. Therefore we highly recommend to generate the space by tracking (longer or shorter) trajectories using the learned parameters.



(a) (b) (c)

Figure 4.4: The space generated by using the update parameters (a) has a different shape than the space generated by embedding histories (b). This can be seen when we plot both spaces into one image (c), where the embedded histories are slightly transparent. The plots are colored by the true underlying x-position.

Results in Fully Observable Environment

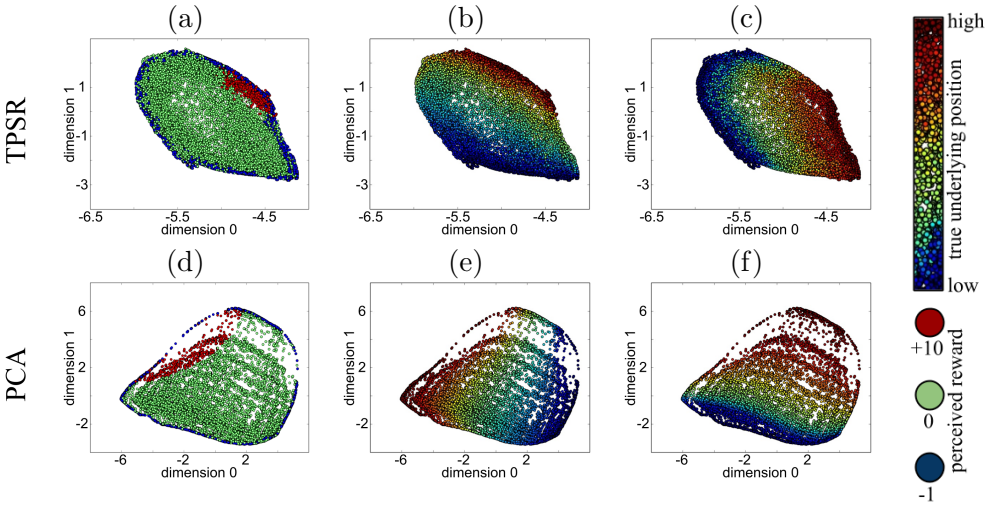
In this part we will present the results of the learning procedure in the fully observable environment. First, we want to analyze the learned representations visually. Figure 4.5 (a)-(d) shows plots of the first two dimensions of state samples generated by tracking one long trajectory of 15000 steps with the learned parameters. We can see that both representations clearly capture the topology of the robot's visual environment. The true underlying x-position and y-position are orthogonal and the positive reward (red, plots a and d) is in that corner where both underlying positions have high values (red, plots b, c, e, and f), which is the upper right-hand corner in the *original* environment. Furthermore, the states where the robot hit the wall and therefore got negative reward (blue, plots a and d) are enclosing the other states.

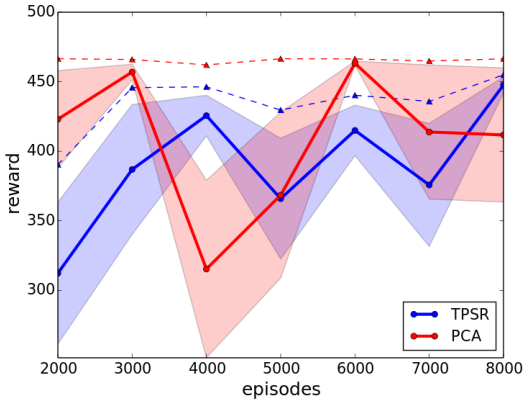
The chart below the plotted sample states reveals that both approaches achieved good average reward scores. Since we took 50 steps for testing and can at most get +10 reward per step, the maximum reward is 500. But as we are starting from positions which are not in the positive reward zone, the actual maximum reward was about 470.

On average, the PCA model (blue) achieves higher rewards, except for the trials with 4000 and 8000 episodes. Especially the median of the PCA shows that for each amount of training data more than half of the trials were very close to the maximum reward. The drops in the mean and high standard error of the mean at some points occurred due to single very bad results. We observed that the PCA models worked either perfectly, or, in very few cases, poorly.

The TPSR models (red) also achieved good mean rewards and a consistently high median value which indicates many trials with high scores. In contrast to the experiments with PCA, it also included rewards in the range between perfect behavior and total failure. But overall, TPSRs showed a slightly worse performance. The median is lower in all cases and the mean is lower in most cases compared to the PCA results. The worse performance gets even clearer if we take the computational effort into account. The whitening step alone includes a principal component analysis and is therefore as demanding as the whole PCA procedure. Furthermore, PCA shows consistently good results even for the smallest amount of training data (2000 episodes) and was learned with one dimension less, hence it led to more compact representations. We also tried to learn TPSRs with two dimensions but they were not sufficient to reliably learn good high quality policies.

As was to be expected, a simple method like PCA can outperform TPSRs in our fully observable setting. There seems to be a large overhead in the TPSR learning procedure as it is thought to deal with partially observable domains and high uncertainty. Thus we will proceed to show that these capabilities will proof to be of advantage in more complex domains.





	TPSR	PCA
SRL steps	$3 \times episodes$	
RL steps	15000	
dim	3	2
h/t-length	1	-
obs-centers	500	-
h/t-centers	1000	-
σ^{obs}	1.0	-
$\sigma^{h/t}$	2.0	-
obs-dim	10	-
h/t-dim	10	-

Figure 4.5: SRL and RL results in the fully observable environment. The upper plots show the first two dimensions of state samples from a learned TPSR (a,b,c) and the same trajectory projected onto the first two principal components (d,e,f). The states are colored by the perceived reward (a,d), the true underlying x-position (b,e), and the true underlying y-position (c,f). The chart shows RL results for TPSR and PCA models learned with varying amounts of training episodes. Thick lines show the mean reward, dashed lines show the median and surfaces indicate the standard error of the mean. The settings can be found in the table next to the chart. The representations (a)-(d) are examples from a run with 24000 SRL training steps (8000 episodes). A figure with all three TPSR dimensions plotted against each other can be found in the appendices (see figure A.1).

Results in Fully Observable Environment with Distractors

To raise uncertainty in our environment, we added visual distractors to the fully observable system (see figure 4.6). As the plots of the learned PCA state spaces in figure 4.7 (d)-(f) reveal, this drastically impairs the observation state mapping. While the true underlying x-position is still captured quite well (plot e), the y-position is not captured at all (plot f) and several positive and negative reward states are spread throughout the space (plot d). The TPSR plots (a)-(c) on the other hand still capture the visual environment of the robot. The shape degenerated to a circle, but the walls still surround the other states (blue, plot a), the underlying x-position and y-position are orthogonal, and the reward area (red, plot a) is at the right spot with high underlying position values (red, plot b and c).

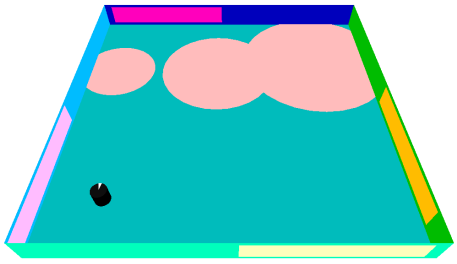
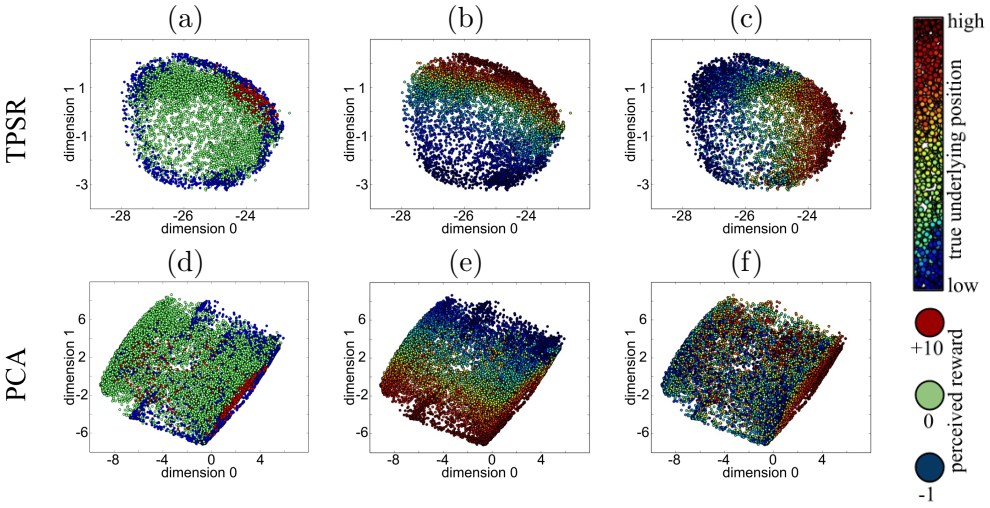
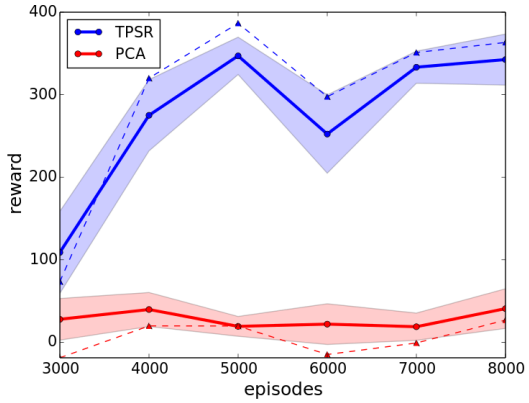


Figure 4.6: We added three randomly moving distractors on the floor and one at each wall. They differ from their backgrounds only in the red color channel of the RGB values. The idea and environment are taken from Jonschkowski and Brock (2015)

The reinforcement learning results seen in the chart below the plots confirm this visual impression. The PCA representations (red) lead to poor results with just barely any rewards above zero, whereas the TPSRs (blue) achieved mean reward of about 300 and medians above that. Hence many trials led to satisfactory results. The scores are worse than the results in the environment without distractors, though. In contrast to results like those of Jonschkowski and Brock (2015) where the presented SRL method can score just as good with or without distractors, the distractors interfere with the performance of the TPSRs. Note that we also had to take more kernel centers for observations, tests, and histories than before to achieve these results. The adjusted settings can be found in the table next to the chart.
We think that the ability of TPSRs to capture the history of the system in its state and taking the last executed action into account helps to navigate in such noisy environments. But it can rather model the noise established by distractors than ignoring them. Nevertheless, this experiment clearly shows that the TPSR parameters allow to model systems with noisy observations that cannot be mapped directly to states (except by using prior knowledge as in Jonschkowski and Brock (2015)).



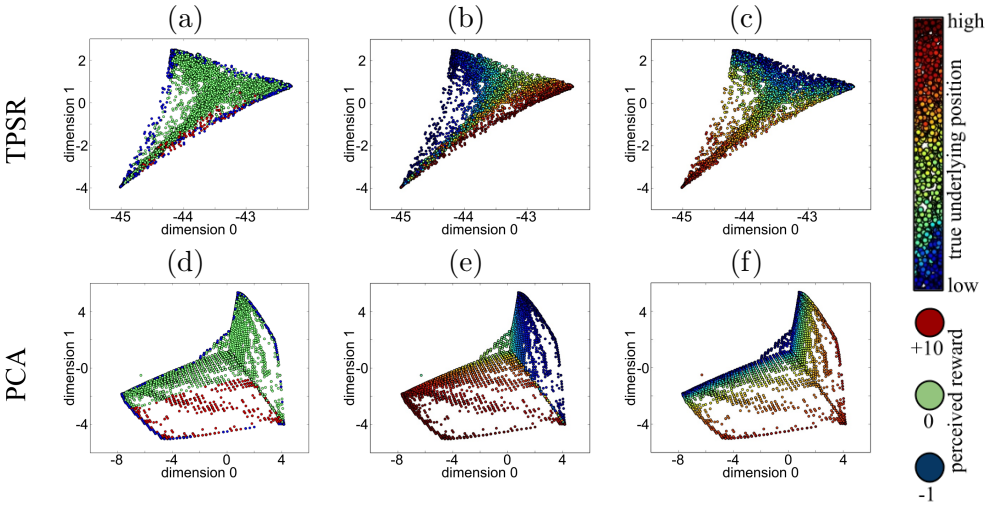


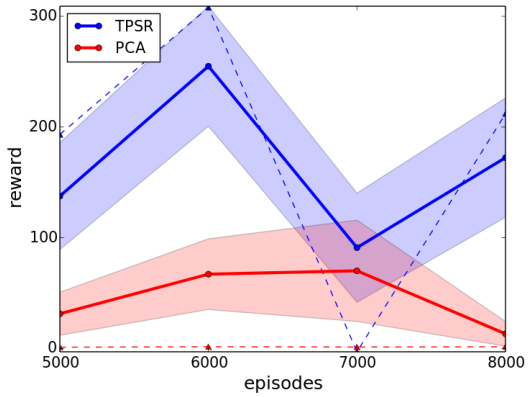
	TPSR	PCA
SRL steps	$3 \times episodes$	
RL steps	20000	
dim	3	3
h/t-length	1	-
obs-centers	1500	-
h/t-centers	2000	-
σ^{obs}	1.5	-
$\sigma^{h/t}$	4.0	-
obs-dim	10	-
h/t-dim	10	-

Figure 4.7: SRL and RL results in the fully observable environment with distractors. The upper plots show the first two dimensions of state samples from a learned TPSR (a,b,c) and the same trajectory projected onto the first two principal components (d,e,f). The states are colored by the perceived reward (a,d), the true underlying x-position (b,e), and the true underlying y-position (c,f).
The chart shows RL results for TPSR and PCA models learned with varying amounts of training episodes. Thick lines show the mean reward, dashed lines show the median and surfaces indicate the standard error of the mean. The settings can be found in the table next to the chart. The representations (a)-(d) are examples from a run with 24000 SRL training steps (8000 episodes). A figure with all three TPSR and all three PCA dimensions plotted against each other can be found in the appendices (see figure A.2).

Results in Partially Observable Environment

The last environment we conducted our line of experiments in is the partially observable version. In this scenario multiple positions can lead to the same observations. For example positions close to the upper (dark blue) wall of the room will lead to the same observations and therefore it will be much harder for the robot to predict whether it is going to hit the wall or how close it is to the rewarding upper right-hand corner.
Accordingly, the plotted PCA state spaces in figure 4.8 (d)-(f) are not smooth anymore. They show gaps where states lead to similar observations and the shape and proportions are skewed. The wall still enclose the other states (blue, plot d), but states close to side walls where the underlying x-position is very high or very low (red and blue, plot e) are stretched and take a big amount of the space, whereas states close to the upper and lower wall (red and blue, plot f) are closer together.
The TPSR in plots (a)-(c) also shows some gaps, but it seems a little smoother and denser. Furthermore, the proportions are skewed in another way. States close to the upper wall with high underlying y-position (red, plot c) collapsed into a small amount of close states. Thus, we can see a triangle shape instead of a square. On a positive note, walls still enclose other states (blue, plot a).
Judging from the visual appearance of the first two dimensions, both representations look insufficient to reliably produce high quality policies. The reinforcement learning results in the chart below the plots confirm this impression. As the median indicates, half of the PCA (red) trials get less than zero reward and only a few good results can raise the mean reward above that mark. The TPSRs (blue) score remarkably better especially in the ten trials with 6000 training episodes. But the performance seems very volatile. We again adjusted the settings and increased the number of kernel centers as can be seen in the table right next to the chart. Still, the results can not keep up with the scores in the other environments.
As there are some trials with very high reward in the TPSR case, we think that with some tweaking of parameters and more computational power we could learn accurate representations more reliably. But with the high number of experiments we conducted we were limited to lower settings. It is, on the one hand, a remarkable result that this generic model *can* learn the complex environment with a good draft of training samples. But, on the other hand, it is also important to note that the learning method starts to exceed the computational limits quite fast. This simulated mobile robot task is far from real world complexity, but still proves to be hard to learn reliably with the used TPSR algorithm. We will further examine the quality of representations learned in the partially observable environment in section 4.4 by testing the capacity of the memory of TPSRs.





	TPSR	PCA
SRL steps	$3 \times episodes$	
RL steps	20000	
dim	3	3
h/t-length	1	-
obs-centers	2000	-
h/t-centers	4000	-
σ^{obs}	1.5	-
$\sigma^{h/t}$	5.0	-
obs-dim	10	-
h/t-dim	10	-

Figure 4.8: SRL and RL results in the partially observable environment. The upper plots show the first two dimensions of state samples from a learned TPSR (a,b,c) and the same trajectory projected onto the first two principal components (d,e,f). The states are colored by the perceived reward (a,d), the true underlying x-position (b,e), and the true underlying y-position (c,f). The chart shows RL results for TPSR and PCA models learned with varying amounts of training episodes. Thick lines show the mean reward, dashed lines show the median and surfaces indicate the standard error of the mean. The settings can be found in the table next to the chart. The representations (a)-(d) are examples from a run with 24000 SRL training steps (8000 episodes). A figure with all TPSR and PCA dimensions plotted against each other can be found in the appendices (see figure A.3).

4.3 Influence of Parameters

In this chapter we will examine the influence of the parameters, which were summarized in table 3.5 at the end of the implementation chapter, on the learned representations and the emerging reinforcement learning results. This will provide intuition into how different parameters relate to each other and how they can be tuned. Unfortunately, the tuning of parameters can be very tedious as in most cases we have to pass through a whole learning-planning loop or at least learn the parameters and plot state samples to get a visual impression. Therefore, we could not do as much experimental runs as in the previous section for all the parameters, but will ins some cases just describe our findings and strategies developed in the countless settings we tried. At the end of this section we will give a little summary and describe a reasonable way to tune the parameters.
Note that three of the parameters, namely the kernel centers, the kernel widths, and the dimensions to keep after whitening arise from our choice of RBF kernel features. The ones of those for histories and tests can be seen as the equivalent of finding a sufficient set of tests and histories if learning TPSRs without features. Boots et al. (2011) claim that sufficiency seems to be achieved easier with a set of features than histories and tests directly, but we want to note here that in return we have to tune 6 parameters (three for both histories and tests each) when using RBF kernel features instead of 2, which can take a lot of time.
In the following we will present our findings in a separate section for each parameter.

Kernel Centers

Assuming that we have adequate kernel widths and whitening settings, which we will discuss later on, the number of *history and test kernel centers* can be seen as the problem of finding a sufficient set of histories and tests. In our experiment we found this as the *most important* and demanding parameter to learn accurate TPSRs. The number of *observation kernel centers*, that determines how well we model the observation probability density function, on the contrary could usually be chosen a bit lower than the number of history and test kernel centers *without* limiting the RL success. An illustration of this can be found in figure 4.9. Here we can see that the representation hardly changed when we increased only the number of observation kernel centers (upper row in the figure), whereas the representation clearly improved when increasing only the number of history kernel centers and test kernel centers (lower row in the figure). In the last representation of the second row we again raised the number of observation kernel centers while keeping a high number of history and test kernel centers to check whether more observation kernel centers would make any difference in this

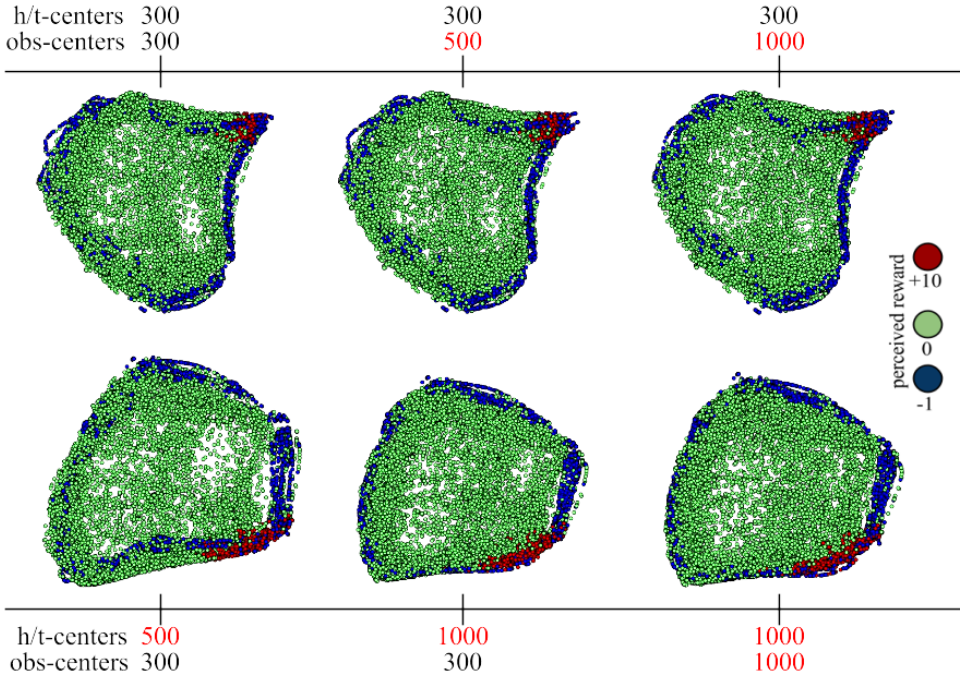
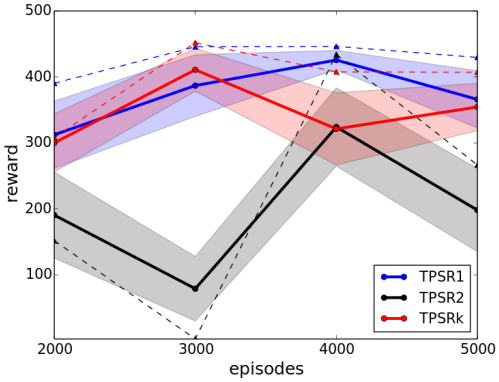


Figure 4.9: We learned several TPSRs with varying amounts of observation kernel centers and history and test kernel centers in the fully observable environment using the same training trajectory of 8000 episodes and plotted the first two state dimensions colored by reward. The first plot in the first row shows the base representation with 300 observation kernel centers, 300 history kernel centers and 300 test kernel centers. The value increased in each other state representation is marked red. The other settings are the same as in our experiments in the fully observable environment shown in figure 4.5.

case. But again, there is hardly any change. Thus, if we found an adequate number of observation kernel centers, we usually do not benefit much from raising it.

Increasing the number of history and test kernel centers seems to always pay off in RL results on the other hand. Figure 4.10 shows the RL results of two TPSRs with the exact same settings except for different amounts of kernel centers in the fully observable environment. In accordance with TPSR theory that a bigger number of features increases the probability of sufficiency, the TPSR with more kernel centers (blue) performs more consistently and has fewer runs with poor performance. Although the TPSR with less centers (black) could achieve maximum reward as well, this happened less often and therefore the mean and median are lower and the standard error of the mean is higher. In an attempt to increase the robustness with-



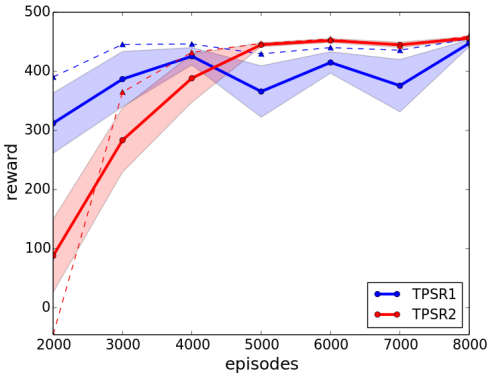
	obs-centers	h/t-centers
TPSR1	500	1000
TPSR2	300	300
TPSRk	300 k-means	300 k-means

Figure 4.10: RL results of three TPSRs learned in the fully observable environment with different amounts of kernel centers. All other settings are equal and stayed the same as in figure 4.5. For TPSRk we used k-means to determine the kernel centers. The thick lines show mean reward, dashed lines show the median and the surfaces indicate standard error of the mean.

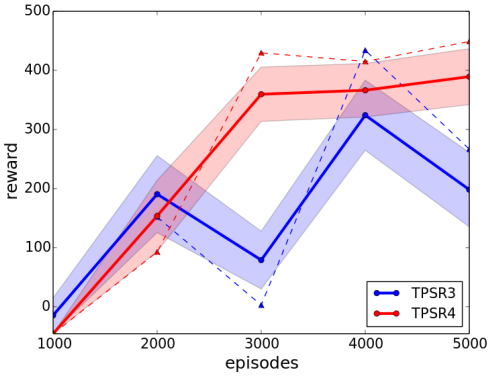
out needing many centers we learned a third kind of TPSRs (red). Here the 300 kernel centers were not taken as the first 300 episodes but found by a *k-means* algorithm on history, test and current observations separately. The chart shows that this approach in fact stabilizes the RL results compared to the second TPSR with just 300 kernel centers (black). But the TPSR with more centers chosen without k-means (blue) still performs slightly better. Hence also using k-means for a larger number of centers seems promising, but unfortunately the algorithm does not scale well with the number of kernel centers and training data. Even for the small amount of 300 centers the k-means algorithm took about two third of the whole TPSR parameter calculation. Thus k-means is intractable for the more complex domains which we could not learn as successful even with several thousand kernel centers.

TPSR Dimensionality

The dimensionality of a TPSR state should usually be as high as the linear dimension of the underlying system. It can be found by SVD of the $P_{\mathcal{T},\mathcal{H}}$ matrix if the history and test features are sufficient. But as we estimate the matrices from a finite amount of data and are limited in the number of kernel centers due to computational complexity of the SVD, we usually can not rely on the singular values of $P_{\mathcal{T},\mathcal{H}}$ as indicator. For example we tried to learn TPSRs in the fully observable environment with 15-dimensional states as the 15th singular value was still bigger than zero. But the resulting representation was completely skewed, had lots of state outliers and led to numerical failures when trying the Q-iteration algorithm on it. An attempt



	TPSR1	TPSR2
SRL steps	$3 \times episodes$	
RL steps	15000	
dim	3	5
h/t-length	1	
obs-centers	500	
h/t-centers	1000	
σ^{obs}	1.0	
$\sigma^{h/t}$	2.0	
obs-dim	10	
h/t-dim	10	



	TPSR3	TPSR4
SRL steps	$3 \times episodes$	
RL steps	15000	
dim	3	5
h/t-length	1	
obs-centers	300	
h/t-centers	300	
σ^{obs}	1.0	
$\sigma^{h/t}$	2.0	
obs-dim	10	
h/t-dim	10	

Figure 4.11: RL results for TPSRs of different dimensionality learned in the fully observable environment with varying amounts of training data. Thick lines show mean rewards, dashed lines show medians, and surfaces indicate standard error of the mean. Plots of a state space of a learned 5-dimensional TPSR can be found in the appendices in figure A.4.

with 10-dimensional states resulted in successful RL but could not achieve better scores than our 3-dimensional representations learned in the previous section.

But as can be seen in figure 4.11, learning 5-dimensional TPSRs (red) stabilized the RL results for TPSRs with many kernel centers (upper chart) and also for trials with less kernel centers (lower chart). Especially in the case with more kernel centers we can see that starting from 5000 training episodes the algorithm reliably learned perfect representations. The standard error of the mean is very low and the mean reward and median are close to the maximum reward of 470. But also in the lower chart with TPSRs with less kernel centers, the increased dimensionality led to a stabilization of the results with many training episodes.

Hence determining the dimensionality parameter includes a trade-off. Com-

pact representations with low dimensionality allow the usage of reinforcement algorithms that suffer from the curse of dimensionality. But increasing the dimensionality of the model can stabilize the procedure and lead to more consistent and better results. However, this statement is only valid if we have enough training data to learn a model with higher dimensionality. As the charts of our experiments in figure 4.11 show, the higher dimensional TPSRs perform worse in the RL task with smaller amounts of training data and only surpass the 3-dimensional competitors starting from 3000 or 5000 training episodes, respectively.

Length of Tests and Histories

The lengths of histories and tests increase the length of an episode and therefore inherently increases the amount of data to gather. If we gather 3000 episodes with histories of length 1, this will be 9000 training steps. For histories and tests of length 3 we would need 21000 training steps for 3000 episodes, though. Furthermore, the amount of possible histories and tests increases exponentially with their length. On the one hand, this is of advantage as we can decide to increase the length if for example histories and tests of length 1 do not show varied enough to capture the environment's properties. On the other hand, it means that the gathered histories and tests will be more different and therefore probably can be estimated worse with the same amount of episodes (which already include more data than with shorter lengths). Thus, the length of histories and tests is an important parameter to find sufficient sets of histories and tests, but can also lead to intractable amounts of data needed. Therefore, we always tried to model the environments with short histories and tests first.

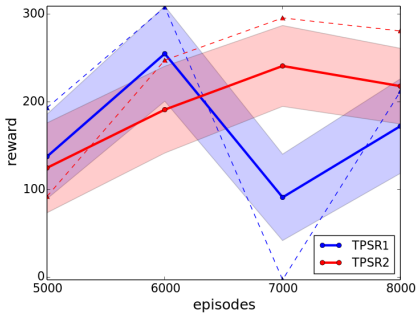


Figure 4.12: Results of learned TPSRs with length 1 (blue) and length 3 (red) histories and tests in the partially observable environment with different amounts of training data. Thick lines show mean rewards, dashed lines show medians and surfaces indicate standard error of the mean. The other settings are the same as given in the table in figure 4.8

In the partially observable environment integrating information over several steps can be helpful to learn more accurate representations. We learned TPSRs with histories and tests of length 3 and compared the results to TPSRs with length 1 histories and tests learned in the partially environment in section 4.2. Figure 4.12 summarizes the results and shows that the TPSR

with longer histories and tests performed slightly better. We think that the difference would be even bigger if we used more training data and more kernel centers, but the computational effort with higher settings was too high to gather empirical results over many trials.
In conclusion, we found a similar trade-off as for the TPSR state dimensionality above. Increasing the length of histories up to a certain length might improve the quality of learned representations, but the amount of data needed to build adequate estimates increases tremendously fast. Therefore we are limited to relatively short histories and tests in practice.

Dimensions to Keep After Whitening

To determine the number of dimensions of current observations, histories, and tests to keep after whitening, we can take into account the magnitude of the eigenvalues during the PCA step of whitening. This is a common approach when using principal components: we divide an eigenvalue by the sum of all eigenvalues to calculate the *explained variance* of the principal component of that particular eigenvalue. It describes how much of the original data's variance is captured by that principal component.
A reasonable rule of the thumb that provided us with good results was to take enough principal components to explain about 95% of the variance. We could roughly achieve this by keeping 10 dimensions in either of the two environments. We decided to take a fixed number of dimensions rather than using a cut-off to determine it variably, because the number of dimensions kept greatly impacted the kernel width we had to choose. Thus, it is more convenient to decide for a fixed number although this could mean that the used principal components sometimes explained a little less than 95% of the variance.
Increasing the length of histories and tests usually also means that we have to keep more dimensions after whitening to stay close to our 95% boundary. We can of course also decide to stay with our fixed number of dimensions and therefore drop more information. For example the first 10 principal components of data with histories and tests of length 4 in the partially observable environment still explained about 85% of the variance. With this amount of information dropped, we could still learn a smooth representation, whereas keeping only few dimensions with less than 80% variance explained led to poor results in the fully observable environment.
In the end, the actual amount we have to keep of course depends on the dynamical system we model and the task we want to solve. Taking a conservative boundary like 95% should decrease the possibility of failure due to information loss. Keeping all dimensions also led to poor results on the other hand, because projecting onto principal components with very small eigenvalues cause numerical issues in further computations.

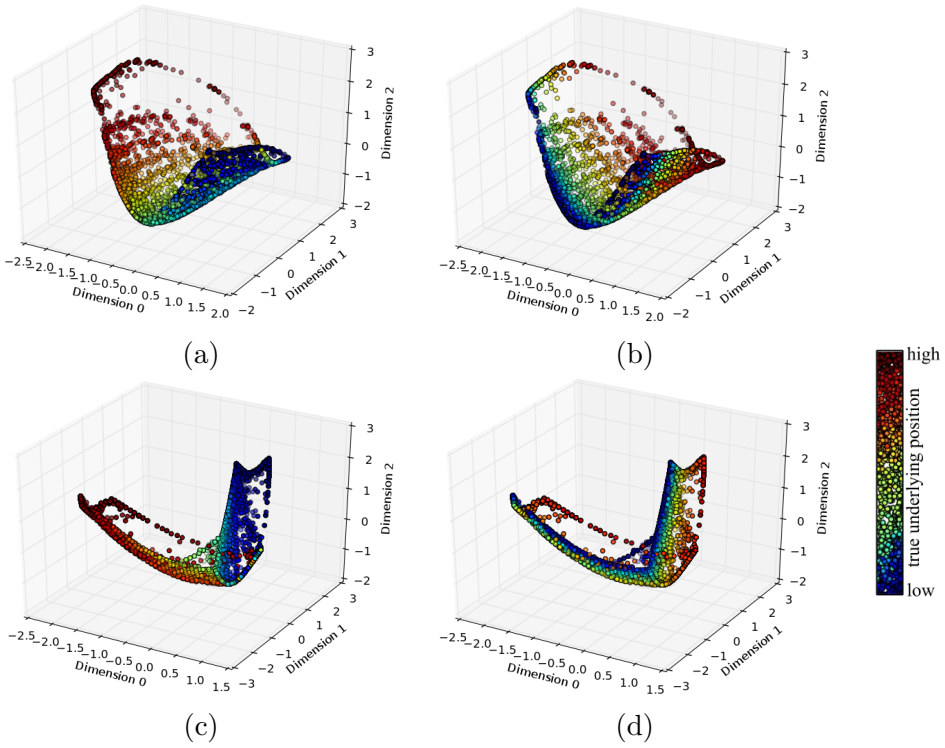


Figure 4.13: The first three dimensions of training data after whitening in the fully observable environment (a,b) and the partially observable environment (c,d). Colored by underlying x-position (a,c) and underlying y-position (b,d).

Kernel Widths

The three kernel widths for current observations, histories and tests are important parameters that highly influence if the RBF kernel features capture the pertinent properties of the system. The wider the width, the higher the activations of kernel centers by more distant samples. In all our experiments we observed that the kernel widths of histories and tests needs to be greater than the kernel width of current observations. This is caused by the fact that observation kernel centers should model the probability density function of observations and therefore require a narrow kernel width. History and test kernel centers on the other hand capture correlations in our data and benefit from wider kernel widths. In fact, we could raise the kernel widths for histories and tests by a significant amount without losing RL success. For example with settings as in the comparison between TPSRs and PCA in the fully observable environment in figure 4.5, values between 2.0 and 6.0 both worked perfectly. The observation kernel width on the other hand was more restricted and could range from 0.5 to 1.5 without impairing the results.

Lower or higher values would lead to skewed representations with outliers that do not allow reinforcement learning with Q-iteration.
However, the widths depend on various factors as the dimensions kept after whitening or the distribution of data. To illustrate this, we plotted the first three dimensions of data gathered in the fully observable and in the partially observable environment after whitening in figure 4.13. To successfully learn TPSRs with this data, the widths were 1.0 for observation kernel centers and 2.0 for history and test kernel centers in the fully observable environment, whereas the partially observable data needed widths of 1.5 for observations and 4.0 for histories and tests. For both environments we kept the same amount of dimensions after whitening. We guess that the difference is caused by the more equal distribution of the data in the fully observable case (plots a and b). In the partially observable case (plots c and d) there are more gaps between clusters of states.
In conclusion, suitable kernel widths are hard to predict, but extremely important to capture the structure of the underlying data and system. They are dependent on the underlying system and the parameters we choose in the whitening steps, which are then again dependent on the length of histories and tests.

Training Episodes

We have seen in several findings above that the amount of needed training data is influenced not only by the complexity of the underlying system, but also by the parameters we choose. While a higher number of kernel centers only moderately increased the needed amount to learn fine representations, taking longer histories and tests more significantly increased the amount of required training data in our experiments.
The RL results in the fully observable environment from figure 4.5 reveal that from 3000 to 4000 training episodes on, the reliability did not increase anymore. We think that the results were rather limited by other setting like the dimensionality of the TPSR and the number of kernel centers in the sense that there was still a small probability of having non-sufficient sets of tests and histories to model the system with 3-dimensional states. Hence increasing the amount of training data did not show significant improvements. In our tests with 5-dimensional TPSRs in the fully observable environment (see upper chart in figure 4.11) on the other hand, we can see the expected behavior that increasing the amount of training episodes leads to better RL results with decreasing growth.
Our experiments showed that in the fully observable environment with and without distractors we could gather enough training data without exceeding the computational limits in terms of time and memory. For the partially observable environment on the contrary, more training data might have led to better results, but the amount started to exceed the computational limits.

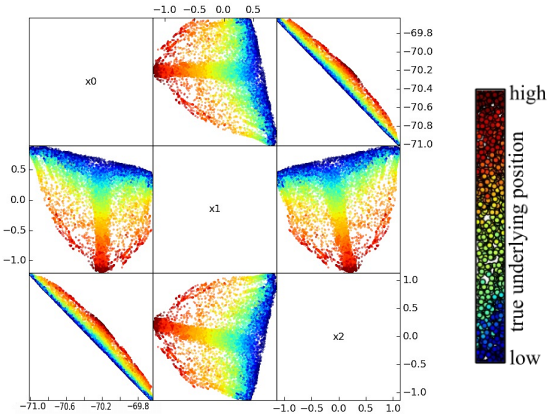
Summary

We showed that many of the parameters influence each other and tuning a certain parameter might lead to the need of tuning other parameters as well. Therefore, finding a suitable set of parameters is tedious as there are many possible sources of error and evaluating the parameter settings always includes learning a complete state representation.
A reasonable way we used to find parameters is to start with relatively high values for those parameters that only limit the learning success if their values are too low. This includes the number of kernel centers and the amount of training data. In contrast, we kept parameters that vastly increase the need of training data like the length of tests and histories and the state dimensionality on a lower level and set the dimensions to keep after whitening to a fixed value so that about 95% of the variance was explained. Then we started a coarse search over kernel widths, where we did not yet learn a policy but only plotted the learned state space (steps 1 to 3 in the experimental design, see figure 4.2). As soon as we found a smooth representation that captured the topology of the robot's visual environment, we started to use the Q-iteration algorithm on the learned representation. Then we were in a position to tune other parameters accordingly. If the RL results were good, we could for example lower the number of kernel centers or the amount of training data to ease the computations and check whether the results remained good. If the RL results were not satisfying, we could for example further increase the number of training steps, the number of kernel centers, or the dimensionality of the TPSR. We could also take longer histories and tests into account but should note that this again might lead to the urge of searching for new, higher kernel widths and adjustments of the number of dimensions to keep after whitening.
This procedure is by no means optimal and shows the difficulties of getting satisfactory results. Taking fewer kernel centers or training data seems tempting in the beginning as it allows faster computation, but actually we had more success by starting off with high values as it decreases the probability of failure due to these parameters.
Figure 4.3 (a) in the beginning of this chapter shows an insufficient representation of an unsuccessful TPSR learning attempt. It basically still seems to capture the topology of the room in the center but has many outliers and high diffusion. The appearance of this insufficient representation shows a typical example for wrong parameter settings in our experiments. Unfortunately, the outer appearance does not tell too much about which parameters were wrong. Wrong kernel widths, too less kernel centers, or too less training data all led to similarly degenerated representations.
Methods that allow the evaluation of parameters without having to learn all TPSR parameters or even solve RL tasks will be needed in order to make work with TPSRs convenient. Despite the the actual idea of replacing hand

modeled representations, adjusting the parameters to the different environments sometimes felt similar to engineering a representation concerning the effort. Thus, future work that deals with ways to automate the described process and limits the number of parameters that have to be tuned thoroughly by hand will be of great importance to reliably learn adequate representations in practice. We will further address such ideas in the conclusions in chapter 5.

4.4 Memory of TPSRs

In this section we will further examine the quality of learned TPSRs by testing their memory in the partially observable environment. The former experiments in this environment in section 4.2 have shown that states of the learned TPSRs close to the upper wall with high underlying y-position collapsed into a small region (see plot (c) in figure 4.8). These are the states where different underlying positions lead to the same observations and can thus only be distinguished by incorporating information from the past steps. To remember this history and hereby distinguish different states that have similar observations, TPSR states need to incorporate a *memory*.
In the following experiments we want to test how many steps of the past the TPSR memory can remember. The PSR theory suggests that if we have a core set of tests and the correct update parameters, the PSR state contains information from arbitrarily far back in history. The plots from TPSRs in the partially observable environment already suggest that this was not the case in our experiments.
We will show that a learned TPSR in the partially observable environment incorporates a memory of a few steps. This sets it apart from direct observation state mappings. Additionally, we will also see that actions in fact are taken into account when calculating the new state as intended.



	Settings
SRL steps	135000
RL steps	20000
dim	3
h/t-length	4
obs-centers	1500
h/t-centers	3000
σ^{obs}	1.5
$\sigma^{h/t}$	10.0
obs-dim	10
h/t-dim	10

Figure 4.14: 3 Dimensions of the learned TPSR plotted against each other and colored by true underlying y-position. We can see that true positions near the upper wall (high y-position values) collapsed into a small area in every dimension. The exact settings can be found in the table on the right-hand side.

Experimental Design

We learned a TPSR in the partially observable environment with tests and histories of length 4 and 15000 training episodes. This is a tremendous amount of SRL training steps (135000) compared to our earlier experiments in this environment with histories and tests of length 1 and maximally 8000 training episodes (24000 steps). Plots of all dimensions of the learned representation and the other settings can be seen in figure 4.14.

To test the memory of the learned TPSR we followed two fixed policies from a common starting state and tracked the states with the TPSR parameters. Plots of the two trajectories in the *actual* environment can be seen in figure 4.15.
The observations and actions from step 13 on in both trajectories are equal. Hence from this step on, the robot can only distinguish its position by using its memory. After 16 steps the robot reaches a position where it only sees the upper wall. But the trajectories did not end at this point. We let the robot stay (no acceleration in any direction) for several more time steps to see if the TPSR state drifts despite not moving the robot.

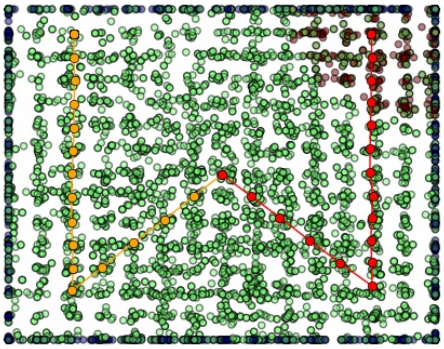


Figure 4.15: The two trajectories we followed plotted in the *real* system.

Results

Figures 4.16 and 4.17 show the results of our experiment. A description of the TPSR behavior as observations and actions become the same or are different in both paths can be found in the text under the figures. In conclusion, the memory of the TPSR seems to be no longer than four time steps. After two steps with the same actions and the same observations the TPSR states already lie very close to each other. The representation immediately starts to drift as soon as observations and actions become the same. It also drifts for a few steps while the robot stands still. If only one of both, actions or observations, are similar, it can distinguish the different positions reliably.
We conducted a similar experiment to verify these results. For that second experiment we let the robot stand still immediately after the first step backwards to the left or right, respectively. Again, the observations at these positions looked the same and from here on the robot took the same ac-

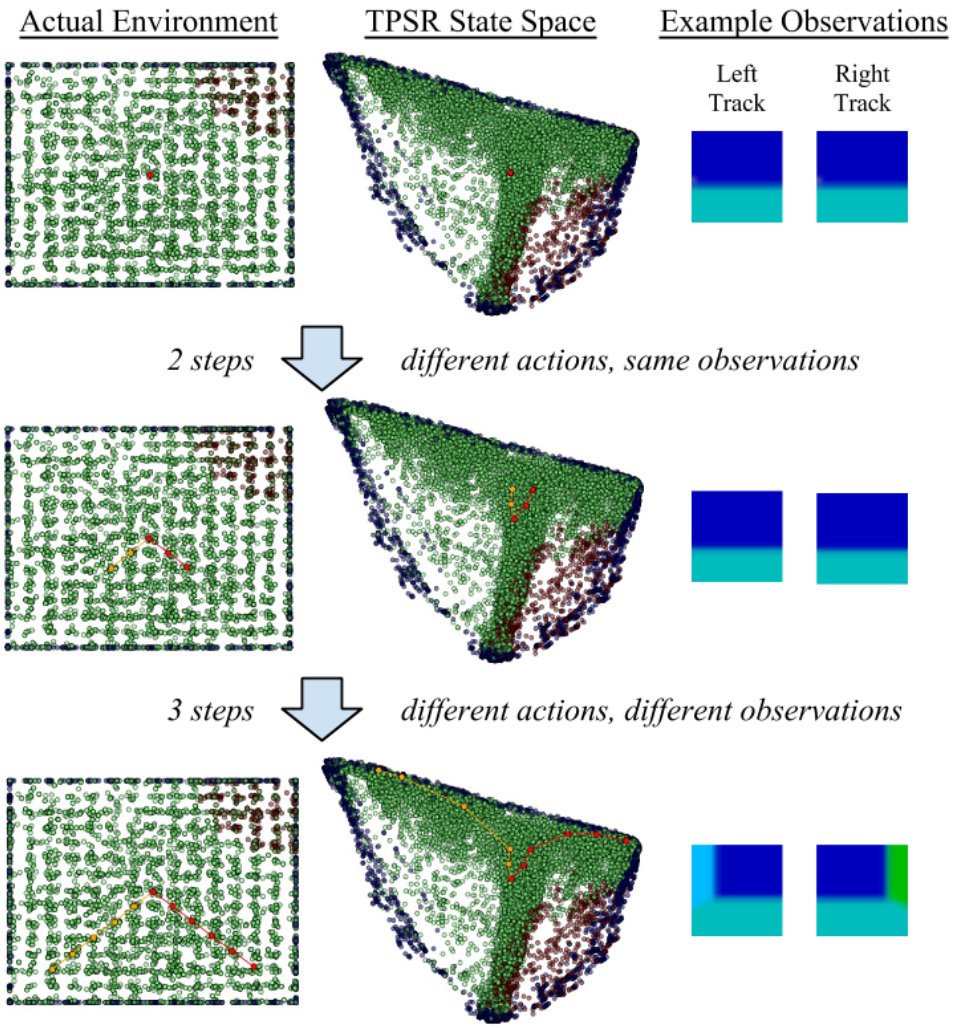


Figure 4.16: The trajectories in the actual environment, the tracked TPSR states in the learned representation and example observations for the two paths taken. In the first steps we have the same observations but different actions and we can see that the actions help the TPSR to distinguish the states, as the paths split up in both the actual environment and the TPSR state space. Afterwards, the states can be distinguished by actions and observations and the TPSR clearly maps to different states near the two bottom corners.

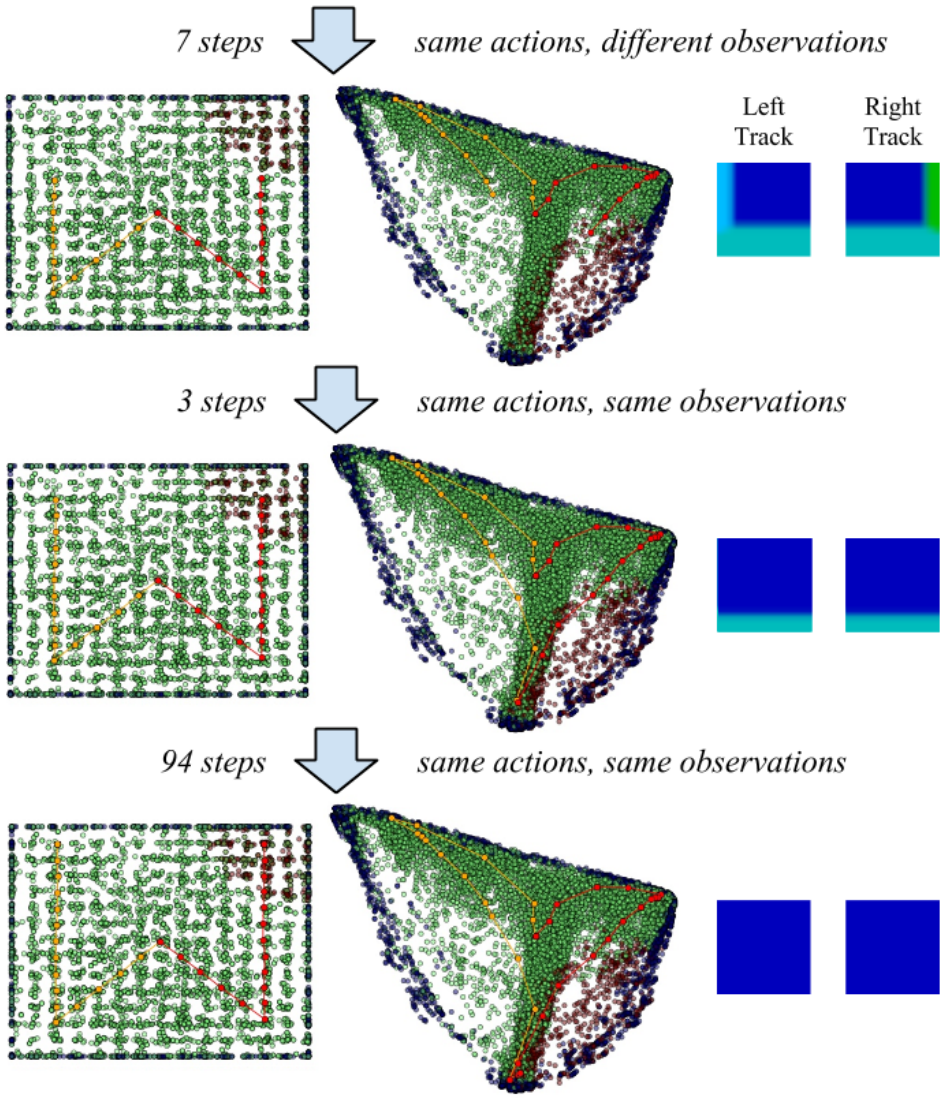


Figure 4.17: As soon as the observations and actions become the same, the TPSR states begin to drift towards each other. After three steps they are already very close (see the three added steps from the first to the second row). After taking one more step the robot reaches the final position and the TPSR states are already equal. In the last plot we can see that the states drift for several steps after reaching the final position although the robot does not move for 94 steps.

tions (standing still) in both trajectories. Plots of the resulting states can be found in the appendices in figure A.5. In that experiment even one step with the same observations and actions brought the states very close together. After three to four steps they overlapped completely.
This shows that our TPSRs memorizing capabilities are far from storing information arbitrarily far back. We could not find better settings to improve the memory and could not determine what caused this behavior. Maybe the TPSR would incorporate a longer history when using more training data or a bigger set of history and test features. Interestingly, we repeated the second experiment with a TPSR with *shorter* tests and histories of length 1 and *double the amount* of training episodes (30000 instead of 15000) and found similar results. The memory was about three to four steps long and one step with the same observations and actions already led to very close states.
Furthermore the choice of features could have influenced the results in a bad way. As we do not consider the taken actions when calculating feature matrices of histories and tests, we might have dropped valuable information about how to distinguish positions with similar observations. This information could have helped to achieve better performance in terms of the memory of the TPSR. Thus, a different choice of features might help to increase the memorizing capabilities and RL results, as well. Therefore, questions of how to improve the memory and representations in general provide a rich resource for future work. We think that especially choosing other features, which can also be seen as a step of providing prior knowledge, is an interesting and promising approach, as the features impact whether the pertinent properties of the environment can be captured. Thus more appropriate features might lead to more valuable sets of histories and tests.

Chapter 5

Conclusion

In the previous chapters we have guided from the underlying theory of (T)PSRs to the practical implementation and presented results of several experiments with learned TPSRs.
We have seen that in theory PSRs can model any dynamical system of finite linear dimension n with predictions of n *core tests* which describe future events. But the *discovery problem* of finding these core tests is intractable in complex systems with action and observation spaces of high cardinality and therefore limits the learnability. The concept of TPSRs allows to mitigate the discovery problem and combined with the idea of using features it lets us model more complex domains with continuous observation spaces.
We provided detailed instructions on how to implement TPSRs with features and also thoroughly described how we gathered and pre-processed data and how the learned model can be used.
Our experiments verified former findings that the implemented algorithm suffices to learn TPSRs that enable reinforcement learning in a complex mobile robot environment with continuous observation space directly from data. This means that the general notion of predictions about future events in fact can be used to adequately represent dynamical systems in practice. The experiments in a fully observable environment with task irrelevant distractors made clear that the learned TPSRs integrated information over time and of taken actions. Therefore they performd significantly better than basic observation state mapping methods like PCA in this setting.
Nevertheless, our experiments also showed that the practical performance lags behind the theoretical promises. This has for example been shown in the experiments about the memory of TPSRs, where states with similar observations in a partially observable environment could barely be distinguished. The reinforcement learning results in the partially observable environment revealed practical limitations as well as we were not able to *reliably* learn adequate representations in every trial.
We think that this is mainly caused by the discrepancies between theoreti-

cal assumptions and practical conditions. Although the concept of TPSRs makes learning easier as we can provide large sets of tests and histories which therefore have a higher probability of being sufficient, the number of tests and histories is limited in practice and finding sufficient sets still remains the main challenge. This challenge is further aggravated by the many parameters related to the quality of the sets of tests and histories especially when working with features. As we already explained, we have to decide for a number of history and test kernel centers which is limited in practice. Furthermore, appropriate kernel widths have to be found. We also have to decide for the length of histories and tests. The longer they are, the higher is the variety of available histories and tests. But we need much more training data with longer histories and tests, as well, which is again a limiting factor. Our experiments have shown that even more parameters which influence each other have to be adjusted for every environment separately. This is the second practical burden of the implemented approach. Even if sufficient sets can be provided as in the fully observable environment, the tuning of parameters is difficult and time consuming. It sometimes felt like hand engineering a representation although this should be avoided by *learning* representations.

Hence, there is still much progress to be made in order to make PSRs practical for real world robotic tasks. We hope that our thorough documentation can help others to start own work with PSRs that aims to improve the practical performance. There are several promising starting points for such efforts. From our experiments we drew the conclusion that the following two are especially important: First, the *discovery problem* has to be further alleviated in order to learn PSRs of more complex domains. Second, we need to find ways to tune parameters more efficiently and automated.

Considering the *discovery problem*, integrating actions and maybe even more prior knowledge like reward into the features might help to assure sufficiency and therefore be beneficial to improve the performance in more complex domains. But finding metrics to relate observations with actions or reward is a difficult task. Nevertheless, finding efficient and reliable methods to provide *suitable* sets of tests and histories (or test and history features) will be of great importance. We chose the word *suitable* instead of sufficient because assuring sufficiency in the theoretical sense is not always possible in practice. As Kulesza et al. (2015b) stress, the linear dimensionality of complex real world domains might be unbound or intractably high and therefore sufficient sets can practically not be found. Hence we have to find ways to adapt the approach to learn accurate representations under real world conditions.

The second aspect of finding methods to determine the quality of parameters without having to learn a TPSR (and maybe even a policy) completely would greatly ease the practical work with PSRs. There is already current work related to this issue. Kulesza et al. (2015b) proposed an algorithm to iteratively improve the sets of tests and histories for TPSRs without fea-

tures prior to parameter learning even if sufficient sets are impossible to find. Moreover, Kulesza et al. (2015a) introduced the use of a loss-function to bind the error to the magnitude of dropped singular values. This could be helpful in terms of choosing adequate TPSR state dimensionality. Keeping the dimensionality low and providing small, yet suitable sets of tests and histories will also help to lower the amount of data needed. Thus these two approaches demonstrate the goals future work should follow. We need to find methods that cope with the characteristics of practical environments where theoretical assumptions usually do not hold.
We have given a practical introduction to (T)PSRs and hope to have encouraged future work that will help to push the boundaries of these representations. Finding ways to further ease the discovery problem, deal with practical difficulties, and automate the tuning of parameters will be crucial for the success of PSRs as a useful representation of complex real world systems.

Bibliography

Böhmer, W., Springenberg, J. T., Boedecker, J., Riedmiller, M., and Obermayer, K. (2015). Autonomous learning of state representations for control: An emerging field aims to autonomously learn state representations for reinforcement learning agents from their real-world sensor observations. *KI-Künstliche Intelligenz*, pages 1–10.

Boots, B., Byravan, A., and Fox, D. (2014). Learning predictive models of a depth camera & manipulator from raw execution traces. In *Robotics and Automation (ICRA), 2014 IEEE International Conference on*, pages 4021–4028. IEEE.

Boots, B. and Gordon, G. J. (2011). An online spectral learning algorithm for partially observable nonlinear dynamical systems. In *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence*. Retrieved from: `http://www.aaai.org/ocs/index.php/AAAI/AAAI11/paper/view/3703`.

Boots, B., Gordon, G. J., and Gretton, A. (2013). Hilbert space embeddings of predictive state representations. In *Proceedings of Conference on Uncertainty in Artificial Intelligence (UAI2013)*, pages 92–101.

Boots, B., Siddiqi, S. M., and Gordon, G. J. (2011). Closing the learning-planning loop with predictive state representations. *The International Journal of Robotics Research*, 30(7):954–966.

Bowling, M., Johanson, M., Burch, N., and Szafron, D. (2008). Strategy evaluation in extensive games with importance sampling. In *Proceedings of the 25th international conference on Machine learning*, pages 72–79. ACM.

Cassandra, A. R., Kaelbling, L. P., and Littman, M. L. (1994). Acting optimally in partially observable stochastic domains. In *Proceedings of the 12th National Conference on Artificial Intelligence*, volume 94, pages 1023–1028.

Cline, A. K. and Dhillon, I. S. (2013). Computation of the singular value decomposition. In Hogben, L., editor, *Handbook of Linear Algebra*, Discrete Mathematics and Its Applications, pages 1027–1039. Chapman and Hall/CRC, second edition.

Hamilton, W. L. (2014). Compressed predictive state representation. Master's thesis, McGill University, Montreal.

Hamilton, W. L., Fard, M. M., and Pineau, J. (2013). Modelling sparse dynamical systems with compressed predictive state representations. In Dasgupta, S. and Mcallester, D., editors, *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, volume 28, pages 178–186. JMLR Workshop and Conference Proceedings.

Hsu, D., Kakade, S. M., and Zhang, T. (2012). A spectral algorithm for learning hidden markov models. *Journal of Computer and System Sciences*, 78(5):1460–1480.

James, M. R. and Singh, S. (2004). Learning and discovery of predictive state representations in dynamical systems with reset. In Greiner, R. and Schuurmans, D., editors, *Proceedings of the 21st International Machine Learning Conference*. ACM Press. Retrieved from: `http://www.machinelearning.org/proceedings/icml2004/papers/117.pdf`.

Jonschkowski, R. and Brock, O. (2015). Learning state representations with robotic priors. *Autonomous Robots*, pages 1–22.

Kulesza, A., Jiang, N., and Singh, S. (2015a). Low-rank spectral learning with weighted loss functions. In *Proceedings of the Eighteenth International Conference on Artificial Intelligence and Statistics*, pages 517–525.

Kulesza, A., Jiang, N., and Singh, S. (2015b). Spectral learning of predictive state representations with insufficient statistics. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, pages 2715–2721.

Kulesza, A., Rao, N. R., and Singh, S. (2014). Low-rank spectral learning. In *Proceedings of the Seventeenth International Conference on Artificial Intelligence and Statistics*, pages 522–530.

Littman, M. L., Sutton, R. S., and Singh, S. (2002). Predictive representations of state. In *Advances in Neural Information Processing Systems*, pages 1555–1561.

Lowe, D. (1999). Object recognition from local scale-invariant features. In *Computer Vision, 1999. The Proceedings of the Seventh IEEE International Conference on*, volume 2, pages 1150–1157.

Rosencrantz, M., Gordon, G., and Thrun, S. (2004). Learning low dimensional predictive representations. In Greiner, R. and Schuurmans, D., editors, *Proceedings of the 21st International Machine Learning Conference*. ACM Press. Retrieved from: `http://www.machinelearning.org/proceedings/icml2004/papers/379.pdf`.

Rudary, M. R. and Singh, S. P. (2004). A nonlinear predictive state representation. In *Advances in Neural Information Processing Systems*, pages 855–862.

Siddiqi, S. M., Boots, B., and Gordon, G. J. (2010). Reduced-rank hidden markov models. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics, AISTATS 2010*, pages 741–748.

Silverman, B. (1986). *Density Estimation for Statistics and Data Analysis*. Chapman & Hall/CRC Monographs on Statistics & Applied Probability. Taylor & Francis.

Singh, S., James, M. R., and Rudary, M. R. (2004). Predictive state representations: A new theory for modeling dynamical systems. In *Proceedings of the 20th Conference on Uncertainty in Artificial Intelligence*, UAI '04, pages 512–519, Arlington, Virginia, United States. AUAI Press.

Sondik, E. J. (1971). *The optimal control of partially observable Markov processes*. PhD thesis, Stanford University.

Stork, J. A., Ek, C. H., Bekiroglu, Y., and Kragic, D. (2015). Learning predictive state representation for in-hand manipulation. In *IEEE International Conference on Robotics and Automation, ICRA 2015*, pages 3207–3214.

van Overschee, P. and de Moor, L. (1996). *Subspace identification for linear systems: theory, implementation, applications*. Kluwer Academic Publishers.

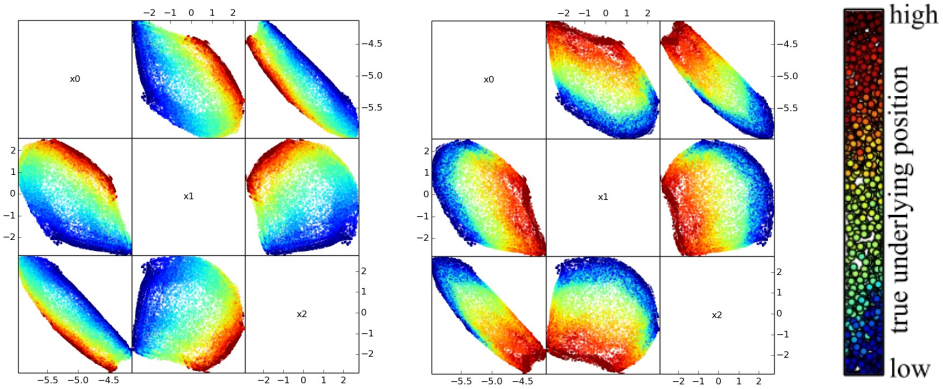
Wolfe, B., James, M. R., and Singh, S. (2005). Learning predictive state representations in dynamical systems without reset. In *Proceedings of the 22nd international conference on Machine learning*, pages 980–987. ACM.

Appendices

Appendix A

Additional Results

A.1 Fully Observable Environment: All Dimensions Plotted



(a) (b)

Figure A.1: All dimensions of the learned TPSR in the fully observable environment pairwise plotted against each other. Colored by true underlying x-position (a) and y-positon (b). The exact settings and planning results can be found in the table in figure 4.5.

A.2 Distractors in Fully Observable Environment: All Dimensions Plotted

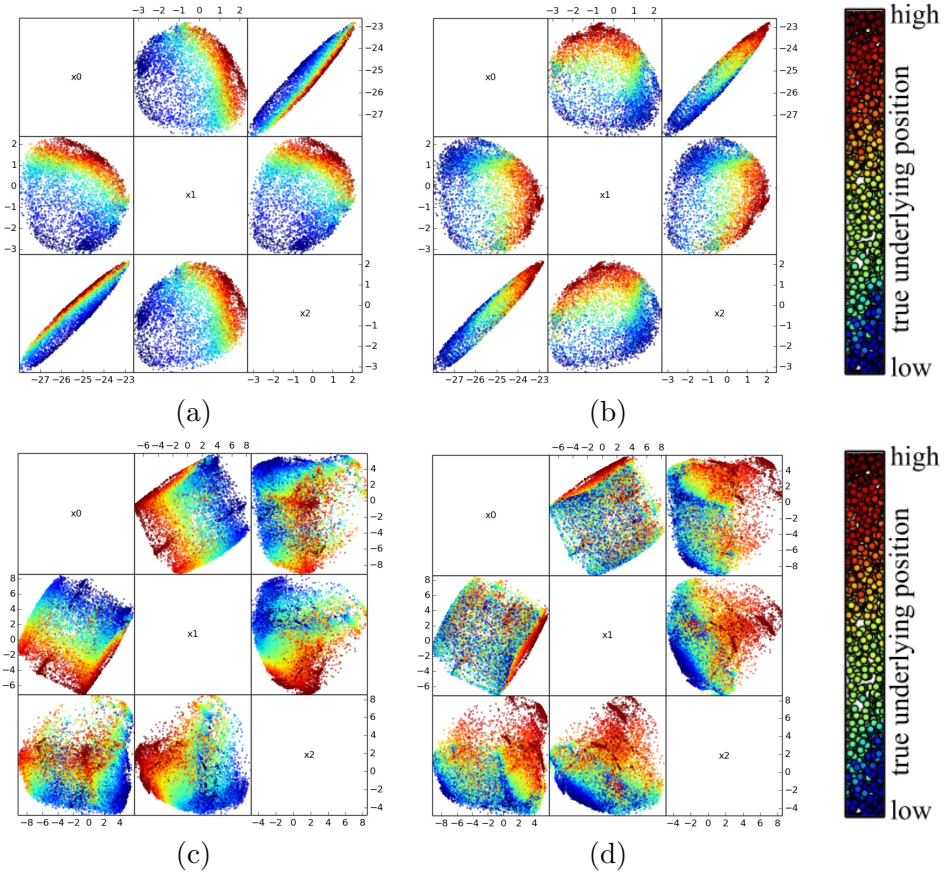


Figure A.2: All dimensions of the learned representations in the fully observable environment with distractors pairwise plotted against each other. The upper row shows the TPSR states, the lower one shows PCA states. Colored by true underlying x-position (a,c) and y-positon (b,d). It can be seen that the TPSR representation captures the topology of the visual environment more accurately. The exact settings and planning results can be found in the table in figure 4.7.

A.3 Partially Observable Environment: All Dimensions Plotted

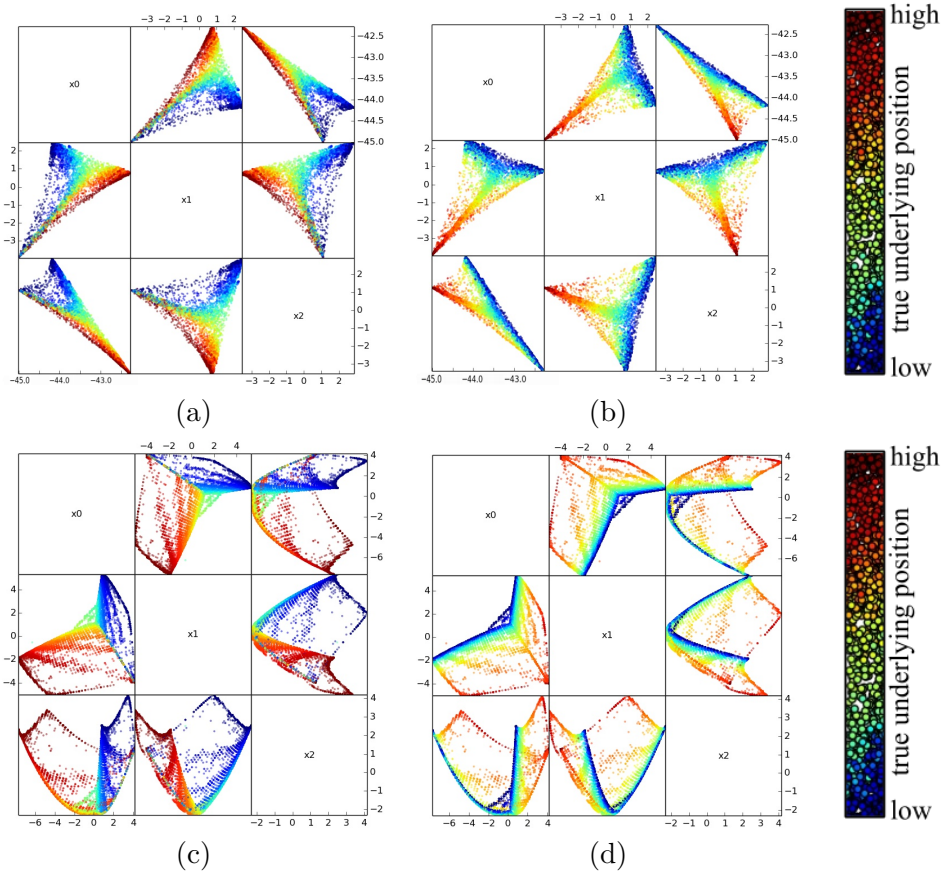
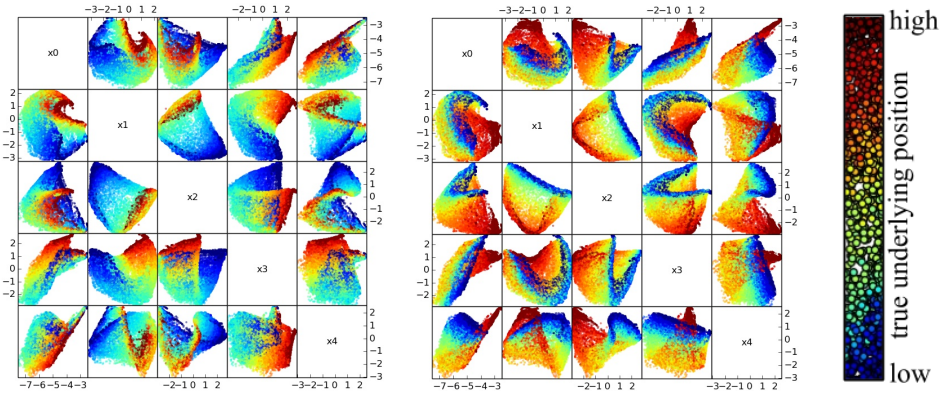


Figure A.3: All dimensions of the learned representations in the partially observable environment pairwise plotted against each other. The upper row shows the TPSR states, the lower one shows PCA states. Colored by true underlying x-position (a,c) and y-positon (b,d). It can be seen that the TPSR representation captures the topology of the visual environment more accurately. The exact settings and planning results can be found in the table in figure 4.8.

A.4 State Dimensionality: 5-dimensional Representation



(a) (b)

Figure A.4: All dimensions of a 5-dimensional learned TPSR in the fully observable environment pairwise plotted against each other. Colored by true underlying x-position (a) and y-positon (b). We can see that the properties of the visual environment are more spread between different dimensions than in the 3-dimensional case where the first two dimensional seem to perfectly capture the environment. Nevertheless, the 5-dimensional representation captures the important properties even better, since it consistently leads to higher score in the RL task. The exact settings and planning results in comparison can be found in the upper table of figure 4.11.

A.5 TPSR Memory: Alternative Trajectories

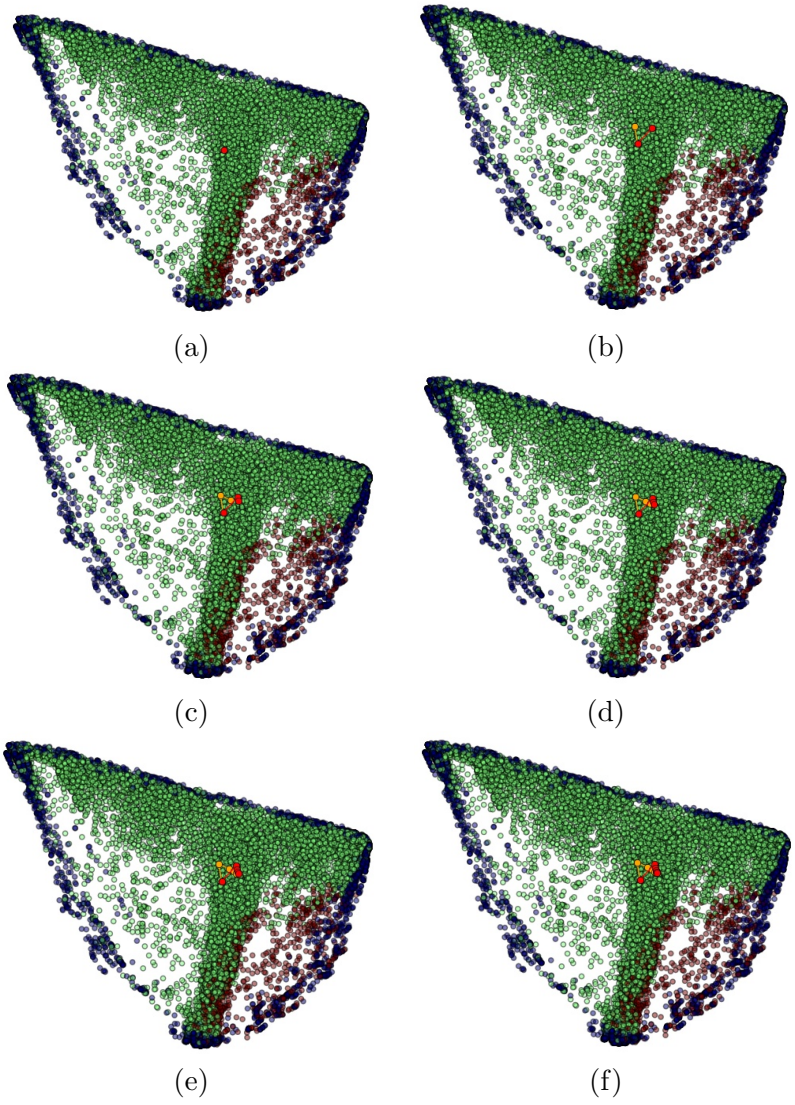
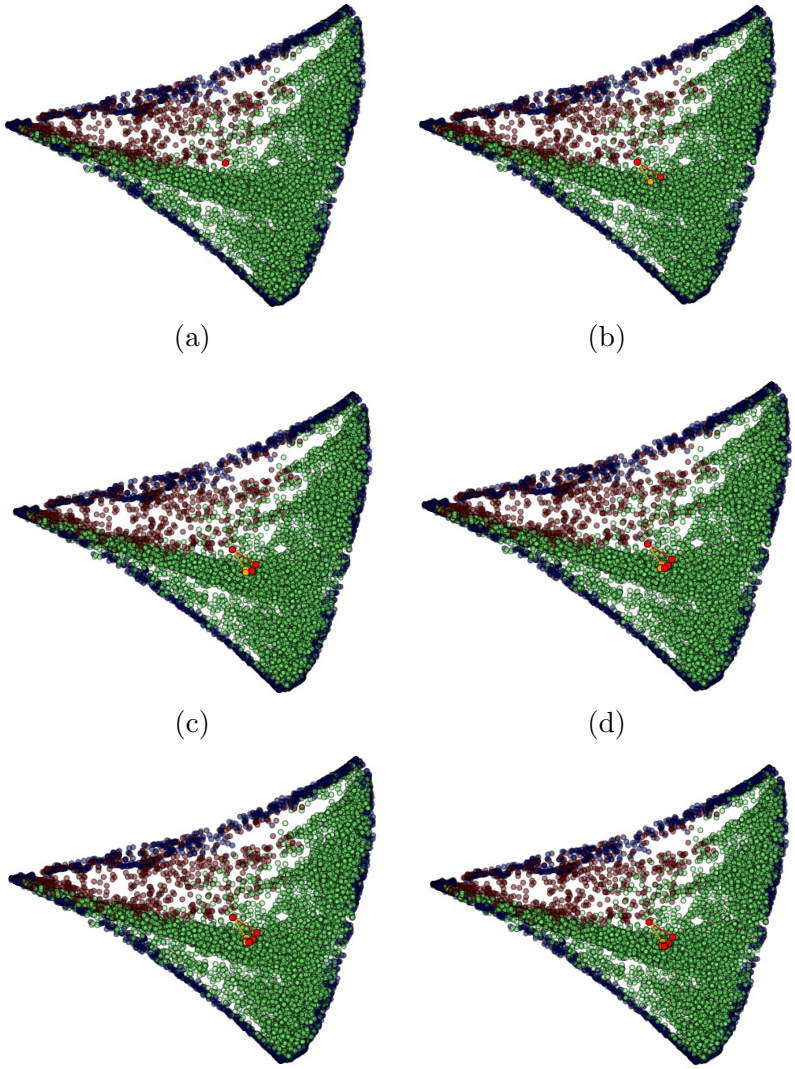


Figure A.5: Results of the second experiment to test the TPSRs' memory. The first step (a-b) are two different actions and the same observations. Afterwards we let the robot just stay, but the states in the TPSR nevertheless drifted and were already very close after one step (c). The following steps (d-f) brought them closer until they overlapped completely.

A.6 TPSR Memory: Length One Histories and Tests



(a) (b) (c) (d) (e) (f)

Figure A.6: Results of the second experiment to test the TPSRs' memory with different settings. We used histories and tests of length one and 30000 instead of 15000 training episodes, but still got similar results as in figure A.5. The memory seems to be no longer than 4 steps, actually one step with same actions and observations (c, where the robot didn't move) already brings the states close together.