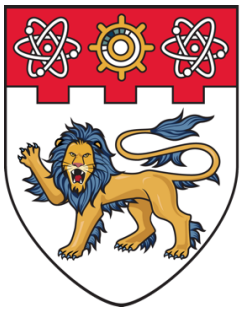


NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

Combining PSR theory with Distributional Reinforcement Learning

JINGZHE ZHOU

School of Computer Science and Engineering

2020

Combining PSR theory with Distributional Reinforcement Learning

JINGZHE ZHOU

School of Computer Science and Engineering

A thesis submitted to the Nanyang Technological University
in partial ful lment of the requirement for the degree of
Master of Engineering

Jan 2020

Statement of Originality

I hereby certify that the work embodied in this thesis is the result of original research, is free of plagiarised materials, and has not been submitted for a higher degree to any other University or Institution.

22/Jan/2020

. . . .

Date   ZHOUJINGZHE

Supervisor Declaration Statement

I have reviewed the content and presentation style of this thesis and declare it is free of plagiarism and of sufficient grammatical clarity to be examined. To the best of my knowledge, the research and writing are those of the candidate except as acknowledged in the Author Attribution Statement. I confirm that the investigations were conducted in accord with the ethics policies and integrity standards of Nanyang Technological University and that the research data are presented honestly and without prejudice.

22/Jan/2020

. . . .

Date   Asst. Prof Zinovi Rabinovich

Authorship Attribution Statement

*(A) This thesis does not contain any materials from papers published in peer-reviewed journals or from papers accepted at conferences in which I am listed as an author.

22/Jan/2020

. . . .

Date ZHOUJINGZHE

Acknowledgements

I would like to express my sincere gratitude and appreciation to my supervisor prof.Zinovi Rabinovich. He is an extremely knowledgeable, patient and understanding mentor who has a great sense of humour. He has the remarkable quality of explaining complex abstract concepts in a very easy-to-understand way, which helped me gain a deep understanding of reinforcement learning in partially observable environments. Under his guidance, I not only got in touch with the cutting-edge research eld of Reinforcement Learning, but also got inspired to be a down-to-earth human being.

I would also like to thank Miss Ridhima Bector who is an excellent lady with a great sense of responsibility. She is happy to help me whenever I get stuck in my research and I deeply bene t from the discussions I have with her. She is always patient while listening to my issues and has wonderful ideas to push me forward. Her most impressive quality is her eagerness to learn, which encourages me to think more.

I am also very thankful to all the people at Computational Intelligence Lab who helped me on this journey in many di erent ways.

Lastly, I would like to thank my family who have provided me with immense support, motivation and love at every step of my journey.

Abstract

This work focuses on using Distributional Reinforcement Learning (DRL) in a partially observable environment that is modelled via Predictive State Representation Theory (PSR). We aim to integrate the bene ts of DRL and PSR to obtain a model-based reinforcement learning method that is capable of providing complete (distributional) performance information about a policy using an observation-only environment model. PSR theory is one of the advanced techniques used to model a dynamical system on a partially observable environment. Unlike traditional partially observable Markov models, such as POMDP, which capture the uncertainty of the environment using belief states, PSR model describes the partially observable environment based on probabilities of executable and observable future events. Distributional Reinforcement Learning (DRL), proposed by MG Bellemare, is a learning paradigm that aims to improve learning by modelling the rewards as probability distributions instead of scalar expectations.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the last decade, there have been numerous accomplishments in the eld of Reinforcement Learning. It has been proven on value-based as well as policy-based algorithms that by learning from a large number of interactions with the environment, the agent can understand the rules governing the environment and is thereby able to learn a good policy to achieve it's goal [24, 48, 45, 51]. However, this task becomes increasingly di cult as we move from simple hypothesised environments to complex real-world-like environments. This is due to the fact that real world environments are generally highly dynamic and partially observable. The research objective in this thesis is to nd an e cient way to train an agent in a highly dynamic and partially observable environment. This research objective has two components which are; representations of the partially observable environment and learning algorithms that can learn based on these representations. In this work, we use Predictive State Representations (PSR) as it enables us to construct the environment model with completely observable quantities. This methodology has been proven to be more e ective than using latent variables [27]. Distributional Reinforcement Learning (DRL) is an advanced learning algorithm which is then used to train the agent in the environment model constructed using PSR. DRL enables the agent to learn to make decisions based on the complete distribution of rewards. We will now discuss DRL and PSR in greater detail.

Distributional Reinforcement Learning [6] proposes a risk-neutral strategy of working with the complete distribution of return instead of grounding the learning process on the expectation of returns. In Reinforcement Learning, the agent makes predictions on given states for each action, which enables the agent to calculate the overall bene t the agent expects to obtain until the end of games [44]. In general, the expected value of an action is approximated as the sum of the immediate reward and the expected value in the next step [23]. Therefore, in general, the agent's learning is based on \expected total rewards". This approach is based on the assumption that \expectation" is the best metric to learn on. This idea falls short in numerous cases, especially when the environment is highly dynamic.

Distributional Reinforcement Learning (DRL) on the other hand, is a powerful algorithm that enables agents to learn on complete distributions of rewards [6]. This provides huge exibility to the agents, as the agents can utilize the entire distribution for learning and thus choose a suitable metric on the y during the execution. Knowledge of the complete distribution allows advanced analysis of the given World. This is primarily advantageous in uncertain and dynamical environments where the knowledge of other \good" paths can immensely aid the agent in making useful decisions.

In order to enable an agent to learn well in partially observable environments, DRL is employed on Predictive State Representation [26] of the environment. In a discrete setting, the world exists as a set of snapshots where each snapshot is a World-state that identi es a unique status of the environment. This World-state changes in correspondence to the actions of the agent, and the agent tries to perceive these World-states with the help of sensors. However, the sensors not only have certain limitations but can also produce faulty readings. This leads to a discrepancy between the actual World-state and the agent's belief of the World-state (hereafter referred to as \belief"). The World-state thus becomes partially observable to the agent. In order to learn a good policy in this

partially observable environment, the agent first needs to build up a model of the World to understand its working better. Instead of creating a probabilistic model based on historical observations [32], Predictive State Representation (PSR) builds a model of the partially observable environment using completely observable future events.

This work aims to enable the agents to learn the rules and knowledge of partially observable environments by employing the Distributional Reinforcement Learning (DRL) on the Predictive State Representation of the environment.

Chapter 2

Literature-Review

In a partially observable environment, the state information is not visible to the agent, and the observations that the agent receives cannot uniquely determine a state. It is thus hard for an agent to understand it's current situation and thereby make long term goals only on the basis of observations. In order to overcome the bottleneck of incomplete state information, Partially Observable Markov Decision Process (POMDP) was proposed. POMDP is the extension of Markov Decision Process (MDP) [3, 16] that creates a probabilistic model of the world [32] to deal with partially observable environments. Due to incomplete state information in a partially observable environment, [31, 37], the agent does not know its current state. This problem is solved by the POMDP model by maintaining a belief vector that contains the likelihood of the di erent states being the current state [2, 50]. The belief vector can be seen as the output of a function of observations. Under the POMDP framework, numerous controlled methods have been proposed to nd an optimal policy in a partially observable environment, such as, policy based methods [42, 43], increment pruning based method [10], policy gradient algorithms [1], state aggregation method [28], state estimation methods [29, 36] etc. The POMDP belief vectors are probability distributions over the entire state space of the world. Learning an optimal policy on belief states is thus computationally intractable [11].

Predictive State Representation (PSR) is a new framework for modelling partially

Method	Difference 1	Difference 2	Difference 3
PSR	PSR is a probabilistic mode based on visible quantities	Features are directly measurable	Acceptable in large environments(CPSR)
POMDP	POMDP is a probabilistic model based on invisible quantities	Features are inferenced by observations	Compuational heavy in large environments

Table 2.1: Di erence Between POMDP and PSR

observable environments and unlike POMDP, PSR works explicitly on completely observable quantities [41]. The experience of McCallum shows that the model that learns on completely observable data is more e ective than models that make approximate estimations on partially observable data [27]. In Predictive State Representation (PSR), the agent tries to understand the environment by predicting how likely an event is to happen next. Instead of maintaining belief vectors regarding what the current state might be, which is a quantity that cannot be known for certain and thus cannot be learned directly; the predictions of PSR models can be correctly estimated with the help of records of execution of events. Moreover, it has been proved that the number of predictions required for representing the environment is not larger than the number of dimensions of the belief vectors [26]. This reduces the complexity of computation required while searching for the optimal policy. The minimal set of events used to represent the environment are known as the \core tests" [9]. Predictions of these \core tests", enable the agent to evaluate the outcomes of taking particular actions in the current step.

Along with the vanilla PSR model mentioned above, this work also experiments with two other PSR variants namely TPSR and CPSR. Unlike vanilla PSR which is generally created with the help of the POMDP model of the world [26], TPSR can be directly learned by extracting useful information from data (experience of the agent). Time spent on construction of the TPSR model however becomes unacceptably large when the environment has a large observation space. To overcome the bottleneck of time complexity

Method	Difference 1	Difference 2
Vanilla PSR	Search for \core tests" feature is based on knowledge of the environments	Little computation
Transformed PSR	Search for \core tests" feature is based on SVD of \sample matrices"	Computationally expensive in large and complex environments
Compressed PSR	Search for \core tests" feature is based on SVD of compressed \sample matrices"	Computationally acceptable in large and complex environments

Table 2.2: Di erences Between Vanilla PSR, Transformed PSR and Compressed PSR

of TPSR, Hamilton proposed to build the PSR model on compressed representations of tests and histories [17]. This reduces the dimensionality of the data matrices, thus shrinking the time complexity from $O(LjZj + jHjjTj^2)$ on TPSR to $O(LjZj)$ on CPSR. Here L is the maximum length of a trajectory and Z is the total number of trajectories in the simulation.

In the Reinforcement Learning eld, there are two main kinds of learning algorithms; value based learning algorithms [46] and policy based learning algorithms [45]. Value based algorithms try to estimate the utility of all the actions in all possible states. The original value based method is known as Q learning [51] .Q learning however tends to overestimate in stochastic environments. Double Q learning [19, 48] was then proposed to help solve the problem of overestimation. Policy based methods, on the other hand, do not try to explicitly learn the utility of each action in each state. They are only concerned about the best action in each state. One of problems in vanilla policy based algorithm is that the gradient would be back and forth in some cases. Trust Region Policy Optimization algorithm (TRPO) [38] puts constraints on gradient descent to avoid \move back". Proximal Policy Optimization algorithm [39] improves the complexity of TRPO with data e ciency. Deterministic Policy gradient algorithm [40] is to work with continuous actions. Compared with stochastic policy gradient algorithms, it is much

Method	Difference 1	Difference 2
Q-Learning	Decisions are based on maximum expected rewards	Bellman Equation is updated on scalar value
DRL	Decisions are based on maximum expectations of distributions	Bellman Equation is updated on distributions

Table 2.3: Di erences Between Q-Learning and DRL

more e cient to learn.

Distributional Reinforcement Learning (DRL) is a state-of-art value based learning algorithm. Gabriel [5] also researches on a policy based distributional learning method. The di erence of DRL with Q-learning algorithms is to transform a scalar expectation on a (state, action) pair into a probability distribution over a value range [6]. In categorical DRL [34], the range of probability distribution is de ned by a set of discrete value, each of which denotes a \canonical return" [6]. Given a state and an action, this method predicts the likelihoods on each \canonical return". In DRL with Quantile Regression [13], instead of xing the \atoms" in value range, it predicts expected values on each \atoms" on the cumulative probabilities.

In our work, we combine the di erent PSR models with the categorical distributional learning algorithm to reinforce the capability of an agent on the partial observable environment.

Chapter 3

Background

We begin this chapter with the introduction of POMDP model in section 3.1. In section 3.2 we introduce the PSR model and discuss the di erences between Vanilla PSR model, Transformed PSR model and Compressed PSR model. Lastly, we conclude the background study with Distributional Reinforcement Learning that is introduced and discussed in section 3.3.

3.1 Partially Observable Markov Decision Process (POMDP)

States in a Reinforcement Learning model capture the essence of the environment and help the agents to take \helpful" actions. In Markov Decision Processes, every world-state follows the Markov property, \the future is independent of the past, given the present" [30]. This property enables the agent to take actions solely based on current observations. However, more often than not, an agent is unable to perceive the complete world-state due to many reasons, such as poor sensors, complex environment, dynamic environment, obstructions etc. In such partially observable environments, where the agent cannot obtain complete state information, the agent has to rely on the past history to deduce the present state. POMDP deals with partial observability by maintaining a

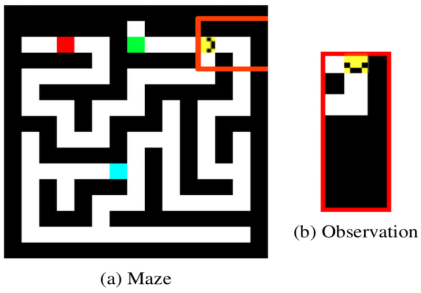


Figure 3.1: Maze illustration

belief vector. The belief vector holds the probability distribution over the state space which indicates how likely it is for a particular state to be the current state [25].

The di culties posed by a partially observable environment can easily be demonstrated with the help of an example. In a maze, like the own shown in Figure 3.1, if the agent can see the complete world-state, it can know its exact position. This enables the agent to easily estimate how good its position is, with the help of metrics such as Euclidean distance, Manhattan distance, etc. However, if in any position, the agent is only able to see whether or not there is a wall on its north, south, east or west, the agent cannot uniquely identify its position in the grid and can only estimate its probability of being in some of the positions. Thus, the agent constructs a belief vector with this probability distribution, which thereafter acts as the agent-state.

3.1.1 The definition of POMDP

The POMDP model can be described as a tuple $< S, A, O, R, T, \mathsf{O}, \mathbb{R}, b_0 >$ [25], where

S = $s_1, s_2, ..., s_n$ denotes state space

A = $a_1, a_2, ..., a_n$ denotes action space

O = $o_1, o_2, ..., o_n$ denotes observation space

R = $r_1, r_2, ..., r_n$ denotes reward space

T denoted the conditional probability $P(s'|s,a)$, $s',s,a \in S \quad S \quad A$

O denotes the conditional probability $P(o|s',a)$, $s',o,a \in S \quad O \quad A$

$\mathbb{R}$ is a reward function: $S \quad A \rightarrow R$

b_0 is the initial belief vector

At the beginning, the agent is instantiated with the belief vector b_0 and a random policy. Based on the random policy, the agent performs an action $a \in A$. This action takes the agent from an unobserved state $s_i \in S$ to an unobserved state $s_j \in S$ with probability $P(s_j|s_i,a)$. The probability of seeing any observation $o \in O$ in state s_j is $P(o|s_j,a)$. Therefore even though an observation cannot uniquely determine a state, it does have some dependency on the state. This allows the agent to estimate s_j as a probability distribution over the state space. After having performed the action a and seen the observation o, the belief vector is updated using Eq. 3.1 [10]. In this equation 1^n is an $n \quad 1$ normalization vector with each element being 1, b^T is a $1 \quad |S|$ vector, T^a is an $|S| \quad |S|$ matrix where each element $v_{i,j}$ denotes the likelihood $P(s_i|s_j,a)$, $O^{a,o}$ is an $|S| \quad |S|$ diagonal matrix where each element $v_{i,i}$ denotes the likelihood $P(o|s_i,a)$. Each update of the belief vector takes it one step closer to the actual World-state.

$$
\begin{aligned}
b' &= P(s'|a,o,b) \\
&= \frac{P(s'|a,b)P(o|s',a,b)}{P(o|a,b)} \\
&= \frac{\sum_s P(s)P(s'|a,s)P(o|s',a)}{\sum_s P(s)P(o|a,s)} \\
&= \frac{\sum_s P(s)P(s'|a,s)P(o|s',a)}{\sum_{s'}\sum_s P(s)P(s'|a,s)P(o|s',a)} \\
&= \frac{b^T T^a O^{a,o}}{b^T T^a O^{a,o} 1^n}
\end{aligned}
\qquad \text{(Eq. 3.1)}
$$

3.2 Predictive State Representation (PSR)

The learning algorithms used in the POMDP framework need to consider all possible probability distributions over the state space. This causes the representation of policy and value function in learning algorithms to be highly complex and computationally prohibitive in large and complicated environments [18].

Predictive State Representation [22, 26, 35, 41] overcomes the problem of high complexity of policy and value function representations by doing away with partially observable states and working purely on observable quantities [26] such as actions and observations. The predictions of tests in PSR theory can be treated as a \state". Its likelihood can be measured by executing it on the environment. A test consists of a sequence of action and observation pairs where each action (taken by the agent) results in the subsequent observation (produced by the environment). The actions that the agent takes and the resultant observations that the agent sees before reaching a state acts as the history for that particular state. The first state thus has null history. Prediction of a test in any given state is the likelihood of the test's observation sequence being produced by the environment when the test's action sequence will be played out by the agent, starting from the given state. The agent can thus infer the current state by making use of the likelihoods of tests which are measured by simulating the action sequence in the environment and recording the observation sequence. This makes learning the PSR model a lot easier than building stochastic models such as POMDP, as discussed in [22].

The utility of PSR model comes from the fact that the probability of a test has a strong correlation with the current state. Different tests can provide similar information and it is thus redundant to use all possible combination of action-observation sequences to determine the states of a particular world/environment/game. PSR theory aims to find a small set of tests, whose likelihoods in different states uniquely determine the states of that particular world. This filtering helps reduce space, time as well as computational

complexity. In order to nd this set of tests, [26] proposes to nd the tests that are linearly independent with respect to any given history and refers to this set of tests as the core tests. The core tests can be used to estimate the likelihood of any test on the given history. This property is used to calculate the likelihood of the core tests for the next state. In this work, the relationship amongst the core tests as well as between the core tests and any other test is linear. This however, is not necessary and a non-linear PSR has already been proposed in [35].

3.2.1 The definition of PSR

PSR can be described as a tuple $< A, O, T, H, Q, M, m_1 >$, where

A, O are de ned as in section 3.1.1.

T is a collection of tests.

H is a collection of histories.

Q is the set of core tests.

m_0 is the likelihood of core tests on null history, $P(Qj\phi)$.

M is a collection of M_t, t 2 T, M_t is a transformed matrix that is used to compute the likelihood of the test t.

m_1 is a normalization term.

PSR theory focuses on learning the likelihoods of tests on di erent histories. The likelihood of a test can be measured on the environment. Each test is a sequence of action-observation pairs. The likelihood for a test t where $t = a_1, o_1, ...a_n, o_n$, on a history h is the probability that the agent sees the observation o_1 on executing the action a_1, observation o_2 on executing the action a_2 and so on, in the next n steps and is represented as

$P(o_1, ..., o_n|h, a_1, ..., a_n)$. The likelihood of a test can be measured directly by repeatedly executing the action sequence in the environment and taking note of the observation sequence. Given a history h, a collection of tests and their likelihoods conditioned on the history, $P(T|h) = [P(t_1|h), P(t_2|h), ...]$ can be treated as the set of core tests if and only if these tests and their likelihood can be used to compute the likelihood of any arbitrary test on the same history and any core test cannot be replaced by other tests. The likelihood of the core tests is denoted as $P(Q|h) = [P(q_1|h), P(q_2|h), ..., P(q_n|h)]^T$. The core tests thus capture nearly complete information about the world, and the likelihood of any arbitrary test on the given history can be represented as a function of the likelihoods of the core tests Eq. 3.2, [41]:

$$P(t|h) = f_t(P(Q|h)) \quad \text{(Eq. 3.2)}$$

As shown in Eq. 3.3, the likelihood of any test is calculated by multiplying the linearly transformed matrix M_t and a normalization vector m_1 with likelihoods of core tests on the same history $P(Q|h)$. After having seen an action a and an observation o, the predictions of core tests are updated as in Eq. 3.4. Multiplying m_1^T on both sides of Eq. 3.4 gives Eq. 3.5. This shows that the product of the likelihoods of core tests on any history and m_1 is equal to one. This enables us to calculate m_1 with the help of the core tests' likelihoods, $P(Q|h)$. Subsequently, M_{ao} can be calculated using Eq. 3.3, [7].

$$P(t|h) = m_1^T M_t P(Q|h), M_t \in M \quad \text{(Eq. 3.3)}$$

$$P(Q|h, ao) = \frac{M_{ao} P(Q|h)}{m_1^T M_{ao} P(Q|h)} \quad \text{(Eq. 3.4)}$$

$$m_1^T P(Q|h, ao) = 1 \quad \text{(Eq. 3.5)}$$

3.2.2 Converting POMDP into vanilla PSR

Construction of PSR model is not an easy task. The hardest problem is to define the dimension of core tests as well as determine the action-observation sequence of the core

tests for the given world. This requires a large amount of data and a lot of statistical analysis. However, if we already have a POMDP model of the given world, it is relatively quite easy to find out the core tests Q and their likelihoods.

$$P(tjh) = P(tjS) \quad P(Sjh) \qquad \text{(Eq. 3.6)}$$

$P(TjH)$ is a $jTj \quad jHj$ matrix whose rows correspond to different tests while columns correspond to different histories. An element $v_{i,j}$ in $P(TjH)$ holds the conditional likelihood of test i given history j i.e. $P(tjh)$. This matrix is constructed with the help of data collected through simulations in the given world. On the other hand, $P(TjS)$ is the $jTj \quad jSj$ matrix whose rows correspond to different tests while columns correspond to different states. An element $v_{i,j}$ in $P(TjS)$ denotes the conditional likelihood of test i given state j i.e. $P(tjs)$. The size of $P(TjH)$ is extremely large because unlike the finite state space, the test space as well as the history space is countably infinite. It is thus substantially more efficient to search for the core tests in $P(TjS)$, than in $P(TjH)$. Eq. 3.6 [21] illustrates the relationship between $P(tjh)$ and $P(tjS)$ where S is the complete state space and $P(Sjh)$ is the belief vector.

$$P(t = a_1o_1jS) = [P(o_1js_1, a_1), ..., P(o_1js_n, a_1)]$$

$$= [\sum_{s^0} P(s^0, o_1js_1, a_1), ..., \sum_{s^0} P(s^0, o_1js_n, a_1)]$$

$$= [\sum_{s^0} P(s^0js_1, a_1)P(o_1js^0, a_1, s_1), ..., \sum_{s^0} P(s^0js_n, a_1)P(o_1js^0, a_1, s_n)] \quad (Bayes\ theorem)$$

$$= \sum_{s^0}[P(s^0js_1, a_1), ..., P(s^0js_n, a_1)] \quad [P(o_1js^0, a_1, s_1), ..., P(o_1js^0, a_1, s_n)]$$

$$= \sum_{s^0}[P(s^0js_1, a_1), ..., P(s^0js_n, a_1)] \quad [P(o_1js^0, a_1), ..., P(o_1js^0, a_1)] \quad (s_i\ is\ independent\ of\ o_1)$$

$$= T^{a_1}O^{a_1o_1}1^n$$

(Eq. 3.7)

$$
\begin{aligned}
& P(o_1, o_2, .., o_n | S_0, a_1, a_2, ..., a_n) \\
&= P(o_1 | S_0, a_1) ... P(o_n | S_0, a_1, o_1, ..., a_n) \\
&= T^{a_1} O^{a_1 o_1} T^{a_1} O^{a_2 o_2} ... T^{a_n} O^{a_n o_n} 1^n
\end{aligned}
\qquad \text{(Eq. 3.8)}
$$

Eq. 3.7 shows how to compute the likelihoods of one step tests for all the states and Eq. 3.8 demonstrates that the likelihood of any multi-step test can be calculated as the product of the likelihoods of the one-step tests that it is composed of T^a, $O^{a,o}$, 1^n. 1^n are de ned in Eq. 3.1 while denotes element-wise product. Since the observation o is independent of the previous state s_i, the last term in Eq. 3.7 becomes a constant vector. This vector is transformed into a diagonal matrix O in the last step of Eq. 3.7. Using these equations, the likelihood of any test can be computed. In PSR theory it has been established that the number of core tests is less than or equal to the number of states in the environment [26]. This property has been used in Algorithm 1, to nd all the core tests Q along with their likelihoods $P(Q|S)$. $P(Q|S)$ is a $|Q| \times |S|$ matrix where each element $v_{i,j}$ denotes the likelihood of core test q_i given state s_j i.e. $P(q_i|s_j)$. $P(Q|h)$ can be computed by taking a product between $P(Q|S)$ and the current belief vector $P(S|h)$ as shown in Eq. 3.6.

$$
m_0 = [P(Q, H)]_{,1}. \qquad \text{(Eq. 3.9)}
$$

$$
m_1^T = e^{|H|} (P(Q, H) N^{-1})^y \qquad \text{(Eq. 3.10)}
$$

$$
\begin{aligned}
& M_{ao} P(Q|h) = P(Q|h, ao) P(ao|h) \\
& for\ any\ history\ h
\end{aligned}
\qquad \text{(Eq. 3.11)}
$$

N is an $|H| \times |H|$ diagonal matrix and the diagonal element $v_{i,i}$ denotes the likelihood of seeing history h_i in simulations i.e. $P(h_i)$. $P(Q|H)$ is a $|Q| \times |H|$ matrix where each element $v_{i,j}$ denotes the likelihood $P(q_i|h_j)$. $P(Q|H)$ is equivalent to the product of the joint probability $P(Q, H)$ and N^y. The element $v_{i,j}$ in $P(Q, H)$ denotes the joint

likelihood of seeing a core test q_i and a history h_j simultaneously i.e. $P(h_j, q_i)$. The joint probability $P(Q, ao, H)$ is a $jQj \quad jHj$ matrix where each element $v_{i,j}$ denotes the likelihood of seeing the test q_i, a action-observation unit ao and history h_j at the same time, $P(h_j, ao, q_i)$. The reason behind using joint likelihood matrix rather than conditional likelihood matrix is that the joint likelihood of seeing a test t and a history h is easier to compute during the simulations. The initial core tests' likelihood m_0 is the rst columns of $P(Q, H)$, as shown in Eq. 3.9. The normalization vector m_1 can be calculated using Eq. 3.10 based on the rule of Eq. 3.5. e^{jHj} is a $1 \quad H$ vector with each element being 1. The linear transformed matrix M_{ao} for each ao in any history h can be computed using Eq. 3.11. M_{ao} computed on the basis of a single \h" has bias in highly dynamic environments due to high variance in data. To work in such environments, M_{ao} should be constructed using all history H. This extended M_{ao} can be constructed using Eq. 3.13 which is deduced from Eq. 3.12.

$$
\begin{aligned}
M_{ao}[P(Qjh_1), ..., P(Qjh_n)] &= [P(Qjaoh_1)P(aojh_1), ..., P(Qjaoh_n)P(aojh_n)] \\
&= [P(Qjaoh_1), ..., P(Qjaoh_n)] \quad [P(aojh_1), ..., P(aojh_n)] \\
&= P(QjaoH)P(aojH)
\end{aligned}
\tag{Eq. 3.12}
$$

$$
\begin{aligned}
M_{ao} &= P(Q, ao, H)P^{y}(a, o, H)P(a, o, H)P^{y}(H)P(H)P(Q, H) \\
&= P(Q, ao, H)(P(Q, H))^{y}
\end{aligned}
\tag{Eq. 3.13}
$$

3.2.3 Transformed Predictive State Representation(TPSR)

Construction of a PSR model using the POMDP model of the same system can be ine cient as construction of the POMDP model requires state transition probabilities as well as observation probabilities. A more e cient approach is to build the PSR model directly from the information available in the environment. Transformed PSR (TPSR)

Algorithm 1 Obtain core tests on $P(TjS)$

```
Input: a matrix P(TjS)
Output: Core tests' likelihood M
M =Vector(), x=0
for each test, t 2 T do
   M =Concatenate M and P(tjS)
   if rank(M) > x then
      x=rank(matrix)
      if x == jSj then
         return M
      end if
   else
      delete the vector P(tjS) on M
   end if
end for
return M
```

model, [33], is an advancement on vanilla PSR model that can be built directly from the data sampled by carrying out random actions in the environment. The Test-History matrix $P(T, H)$ is generated using all trajectories presented in the collected data. In $P(T, H)$, the element $v_{i,j}$ denotes the joint probability of seeing test t_i on history h_j i.e. $P(t_i, h_j)$. TPSR does away with the need of identifying the core tests by working with a set of \special" vectors X which are equivalent to the core tests but can be easily computed using the Singular Value Decomposition of $P(T, H)$, [33]. TPSR additionally o ers the exibility of changing the complexity of its representation by allowing us to choose the SVD dimensions accordingly.

In order to clearly distinguish between vanilla PSR and TPSR models, the two models have been introduced with di erent notations. TPSR is described with the help of the following 7-element tuple: $< A, O, T, H, X, B, \beta_1 >$, [17]:

A, O as de ned in section 3.1.1.

T, H as de ned in section 3.2.1.

X is the set of special vectors.

B is a collection of linearly transformed matrices $B_t \; where \; t \; 2 \; T$, and is used to map the likelihood of a test t onto the special vectors.

β_1 is the normalization vector.

$$USV^T = f_{SVD}(P(T,H)) \quad \text{(Eq. 3.14)}$$

$$Z = U^T \quad \text{(Eq. 3.15)}$$

$$XN = ZP(T,H) \quad \text{(Eq. 3.16)}$$

The matrix Z (Eq. 3.15) is equal to the transpose of the left singular matrix U, Eq. 3.14. The special vectors X are extracted by multiplying the left singular matrix Z with the Test-History matrix $P(T,H)$ as shown in Eq. 3.16. Here N is as described in section 3.2.2.

$$P(T,H) = RQN \quad \text{(Eq. 3.17)}$$

[17] has explained that the Test-History matrix, $P(T,H)$ is equal to the product of R, Q and N, as shown on Eq. 3.17 where R is a vector, $[m_1^T M_{t_1}, ..., m_1^T M_{t_n}]$, Q is $P(QjH)$ and N is as described in section 3.2.2.

$$\begin{aligned} XN &= ZRQN \\ &= JQN \end{aligned} \quad \text{(Eq. 3.18)}$$

Substituting the value of $P(T,H)$ from Eq. 3.17 into Eq. 3.16 gives Eq. 3.18. Eq. 3.18 shows that the vectors X is equal to a linear transformation of the likelihoods of the core

tests Q. This shows that the special vectors X has a linear relationship with the likelihood of any test and can therefore take the role of Q.

$$m_1^T P(Qj H) = e^{jHj} \quad \text{(Eq. 3.19)}$$

$$\beta_1^T X = e^{jHj} \quad \text{(Eq. 3.20)}$$

$$\beta_1^T = m_1^T (ZR)^y \quad \text{(Eq. 3.21)}$$

Eq. 3.19 is obtained by extending Eq. 3.5 to include the complete history space. Given that X is a linear transformation of $P(Qj H)$, the relationship in Eq. 3.19 for vanilla PSR model transforms to the one shown in Eq. 3.20 for TPSR model. e^{jHj} is the same as in Eq. 3.10. Thus β_1^T takes the place of m_1^T in TPSR and can be computed using Eq. 3.21. Similarly B_{ao} takes the place of M_{ao} in TPSR and can be computed using Eq. 3.22 as shown in [17].

$$\begin{aligned} B_{ao} &= ZP(T, ao, H)(ZP(T, H))^y \\ &= JM_{ao}QN(QN)^y J^y \\ &= JM_{ao}J^y \end{aligned} \quad \text{(Eq. 3.22)}$$

In conclusion, TPSR can be seen as an invertible transform of vanilla PSR model where the core tests of vanilla PSR model are replaced by special vectors. TPSR builds a predictive model of a partially observable environment directly from sampled data. When building a TPSR model of an unknown environment, it is important to carefully choose an appropriate dimension of the special vector X in order to ensure high quality of the model. As TPSR does not work with the actual set of core tests, its prediction' quality is lower than that of the corresponding vanilla PSR model built on the same world/game/environment.

3.2.4 Compressed Predictive State Representation(CPSR)

Compressed Predictive State Representation (CPSR) [17] is an extension of Transformed PSR model. CPSR improves the time and space complexity by compressing the Test-History matrix before taking its SVD decomposition. This is very helpful because if TPSR model is applied on an environment with a large number of observations, the time spent on matrix decomposition becomes unacceptable, [17]. CPSR uses Johnson-Lindenstrauss (JL) projection matrices [14, 15] to encode each test and history which considerably reduces the dimensionality of the Test-History matrix. The components of CPSR model are:

A, O as de ned in section 3.1.1.

T, H as de ned in section 3.2.1.

is a set of special vectors.

C is a collection of linearly transformed matrices $C_t \;\; where \;\; t \; 2 \; T$ that are used to map the likelihood of a test t onto the special vectors in CPSR.

c_1 is the normalization vector.

$$= \begin{bmatrix} 1,0 & 1,1 & & 1,n \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ j \;\; j,0 & j \;\; j,1 & & j \;\; j,n \end{bmatrix} \qquad \text{(Eq. 3.23)}$$

n is the projection dimension. is the random projection matrix generated based on

[14, 15], as shown on Eq. 3.23. is the number of events projected.

$$
{}^{0}_{H} = \begin{cases}
\begin{bmatrix} 1 & 1 & 1 & & 1 \\ 1 & {}_{1,0} & {}_{1,1} & & {}_{1,n} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & {}_{jHj,0} & {}_{jHj,1} & & {}_{jHj,n} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & & 1 \\ 1 & & & & \\ \vdots & & {}_{H} & & \\ 1 & & & & \end{bmatrix} \\
if \ \ agent \ \ starts \ \ at \ \ random \ \ states \\
\begin{bmatrix} 1 & 0 & 0 & & 0 \\ 0 & {}_{1,0} & {}_{1,1} & & {}_{1,n} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & {}_{jHj,0} & {}_{jHj,1} & & {}_{jHj,n} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & & 0 \\ 0 & & & & \\ \vdots & & {}_{H} & & \\ 0 & & & & \end{bmatrix} \\
if \ \ agent \ \ starts \ \ at \ \ same \ \ states
\end{cases}
\qquad \text{(Eq. 3.24)}
$$

$_H$ is the projection matrix with = jHj. The rst row denotes the representation of null history, whose value depends on the di erent situations as shown on Eq. 3.24. If the agent starts at a random state, the projection of null history would be 1^{n+1} vector, and any other projected vector for history extends 1 at the beginning. Otherwise, the null history would be $[1, 0, 0..., 0]$ and other history starts at 0 at the beginning. This is to ensure that ${}^{0}_{H}e$ = $[1, 1, ..., 1]$ if and only if the agent starts at an random position and ${}^{0}_{H}e$ = $[1, 0, ..., 0]$ if and only if the agent always starts at the same position, [17]. e is a n 1 vector. Besides the rst element is 1, others are 0. $_T$ is a random projection matrix with = jTj.

$$
\begin{aligned}
{}_{H} &= {}^{0}_{H}P_{H} \\
&= \sum_{l2L} \sum_{h_j 2H} I_{h_j}(l)\phi_H(h_j)
\end{aligned}
\qquad \text{(Eq. 3.25)}
$$

The $_H$ matrix presented in Eq. 3.25 is the compressed form of diagonal history matrix N de ned in section 3.2.2. l denotes a trajectory and L denotes the collection of all trajectories. The function I is used to measure the frequency of test and history in the sampled data and is thus used to construct the Test-History matrix. I_{h_j,t_i} is 1 when the given trajectory l can be split into h_j and t_i.

$$
\begin{aligned}
{}_{T,H} &= \quad {}_{T}P(T,H) \quad {}^{\emptyset T}_{H} \\
&= \sum_{l2L} \sum_{t_i,h_j2T} \quad {}_{H} I_{h_j,t_i}(l)[\phi_T(t_i) \quad \phi_H(h_j)]
\end{aligned}
\tag{Eq. 3.26}
$$

The ${}_{T,H}$ matrix present in Eq. 3.26 is the compressed form of Test-History matrix $P(T,H)$. $P(T,H)$ is de ned in Eq. 3.14. The di erent components of CPSR model can be computed using Eq. 3.27, Eq. 3.28 and Eq. 3.29 as explained in [7].

$$
= Z \quad {}_{T,H} e \tag{Eq. 3.27}
$$

$$
c_1^T Z \quad {}_{T,H} = \quad {}_{H} \tag{Eq. 3.28}
$$

$$
\begin{aligned}
C_{ao} &= (Z \quad {}_{T,ao,H})(Z \quad {}_{T,H})^{y} \\
&= \sum_{z2Z} \sum_{t_i,h_j2T,H} I_{h_j,ao,t_i}(z)[(Z\phi(t_i) \quad ((Z \quad {}_{T,H})^{y}\phi(h_j))]
\end{aligned}
\tag{Eq. 3.29}
$$

In conclusion, CPSR model is the compressed version of TPSR model that overcomes the \large observation space" bottleneck of TPSR model by compressing the data matrices. This compression does not break distances in \Euclidean space" and succeeds in speeding up decomposition of $P_{T,H}$ matrix. However, the compression also deteriorates the quality of CPSR model in comparison with TPSR model. If projection dimension is too small, the information may get lost in compression.

3.3 Distributional Reinforcement Learning

A model based learning system generally constitutes a model of the environment as well as a learning technique which can be used by the agent to make decisions in the given environment. In our work, we use di erent PSR models to represent the environment and Distributional Reinforcement Learning to enable an agent to learn and make decisions.

Majority of the learning algorithms such as Q-learning [51], SARSA [23] etc, enable an agent to choose an action in a particular state on the basis of the expected cumulative rewards that the actions will lead to. Distributional Reinforcement Learning, [5, 6, 13], on the other hand, allows an agent to access the entire distribution of cumulative future rewards to base their decision on. This is exceptionally important in dynamic environments where an agent can exploit this information in di erent ways. By enabling access of the entire reward distribution to the agent at each step, DRL enables the agent to choose from di erent metrics on the go without relearning anything. Two main types of DRL have been introduced in the literature which are; Categorical DRL and Quantile DRL. Categorical DRL uses a discrete distribution of rewards while Quantile DRL builds a probability distribution over the transpose of the discrete reward distribution. In this work we experiment with Categorical DRL on Predictive State Representations of the environment.

3.3.1 Categorical DRL

Categorical DRL [6] represents the rewards corresponding to all (action, state) pairs as discrete probability distributions where the minimum value V_{min} and the maximum value V_{max} are decided by expertise experience.

$$\begin{aligned} Z(x,a) &\overset{D}{=} R(x,a) + \gamma Z(X^{\prime}, a^{*}) \\ X^{\prime} \quad P(\,\cdot\,|x,a), a^{*} \quad & \underset{a \in A}{max} Z(X^{\prime}, a) \end{aligned} \tag{Eq. 3.30}$$

The distributional form of Bellman equation presented in Eq. 3.30. It is used to update the estimate of the probability distribution on every (action, state) pair, (x, a). In Eq. 3.30, Z represents the value distribution function, R represents a one-step reward function, $(X^{\prime}, a^{*})$ is a pair of next state along with the action executed on that next state.

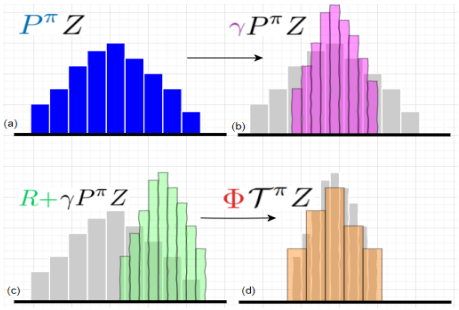


Figure 3.2: Distribution update [6]

Using Eq. 3.30 to update the value distribution shifts the minimal and maximal bounds of the distribution which is due to the discount rate and the reward that the agent gains. This shift creates the problem of inconsistent bounds of distributions during computation of loss. The issue is resolved by projecting the updated distribution to the original space. Figure 3.2 gives an overview of the update process.

In Eq. 3.31, the bellman operator in front of the distribution Z is to denote that the distribution Z has been updated based on Eq. 3.30, which causes the minimal and maximal bounds of the distribution to shift. Putting bellman operator in front of the distribution Z denotes that the distribution Z has been projected back into the original space. This projection is implemented using Eq. 3.31.

$$(\quad Z_\theta(x,a))_i = \sum_{j=0}^{N\ 1}[1 \quad \frac{j[\ z_j]_{V_{min}}^{V_{max}} \quad z_i}{z}]_0^1 p_j(x^\emptyset, \pi(x^\emptyset)) \qquad \text{(Eq. 3.31)}$$

In Eq. 3.31, N denotes the number of atoms on the discrete distribution, z is the incremental gain between two adjacent intervals. It illustrates to project the shifted distribution back to original value range, $[V_{min}, V_{max}]$ [6].

The loss between the predicted distribution and the updated distribution calculated by the distributional bellman equation is quanti ed by the Wasserstein metric [47]. Generally, most of the learning algorithms use a gradient descend optimizer to minimize the loss function [4, 8]. However, the loss function on this metric cannot be minimized by

gradient descend, as explained in [13] . The loss function is instead approximated by the KL divergence $D_{KL}($ $Z(x,a)jjZ(x,a))$, which is then readily minimized by gradient descent [13].

Overall, categorical DRL is a state-of-the-art technique that is especially useful on environments with high variance of rewards for the same action. The probability distribution over several outcomes provides substantially more information to the agent than the scalar expectation of returns. Categorical DRL has also been shown to achieve very good performances on Atari games [6].

3.4 Conclusion

In this chapter, we have described and analyzed the di erent PSR models and have discussed the advantages of DRL algorithm. In order to train an agent in partially observable environments, we combine these two methodologies. In some special environments, the PSR model, which is based on completely observable quantities, is unable to correctly distinguish between the di erent \states". To solve this problem, we have introduced an additional reward signal in the PSR model which reinforces the capability of the PSR model. We look into this extension of the PSR model in the next chapter.

Chapter 4

Contribution

The first contribution of this paper is extension of the PSR model to include a reward signal. This extension is very helpful in certain complex environments where the agent is unable to build up an accurate state representation of the world while relying only on observations. Moreover, the addition of reward to the PSR model enables it to be used as a replacement of the actual environment. This is highly useful when the cost of interaction with the actual environment is large.

The second contribution of this paper is the combination of DRL and PSR to create a state-of-the-art model-based learning methodology to enable an agent to learn in a dynamic partially observable environment. The agent uses its experience to build up a predictive model of the partially observable environment using PSR theory. The likelihoods of \core tests" in this predictive model are updated based on the observation (and reward) obtained by the agent on taking a particular action. This predictive model thus acts as the representation of the world in the agent's mind. In this work, vanilla PSR model along with its derivatives namely TPSR (Transformed PSR) and CPSR (Compressed PSR) are used to model the environment. Distributional learning algorithms are then used to help the agent make sequential decisions in the given worlds.

4.1 Extend PSR model to include Reward

Masoumeh T. Izadi and Doina Precup [20] have argued that reward signals carry useful information that helps disambiguate the hidden state. This is especially true in situations where the reward is determined by the agent's location and action. However, most of the time, action and observation information is su cient to disclose the hidden information i.e. the current state of the agent. Therefore, generally, even though the state and the observation are not equivalent, the agent is still able to roughly deduce it's state by calculating the conditional probability of the di erent states given its own historical experience.

However, in certain cases, the observation-only PSR models fail to accurately represent the world. For example, this happens in environments where observations can be unrelated to the state in which the agent took the action. Since rewards are generally highly correlated with the states, a reward signal can signi cantly help in distinguishing one state from another and can thus be used in such scenarios.

Introduction of rewards into the PSR model changes the composition of a test from action-observation units to action-observation-reward units. The de nition of a test's prediction is rede ned as the likelihood of seeing the given sequence of observations and gaining the given sequence of rewards on taking the given sequence of actions. The transformed matrix M is also expanded to the size: $jAj \quad jOj \quad jRj$. In Eq. 3.6, the predictions of a test on a history in the vanilla PSR model, not only considers the belief state $P(Sjh)$, but also considers the likelihood of the test in each of the states $P(tjS)$. The new de nition of $P(tjS)$ i.e. the likelihood of a single step test t where (t=a1,o1,r1),

due to inclusion of the reward signal is as shown in Eq. 4.1.

$$\begin{aligned} P(tjS) &= P(o, rjS, a) \\ &= \sum_{s^0} P(s^0, o, rjS, a) \\ &= \sum_{s^0} P(s^0jS, a)P(ojS, a, s^0)P(rjS, a, s^0, o) \\ &= P(S^0jS, a)P(ojS^0, a)P(rjS, a) \end{aligned} \qquad \text{(Eq. 4.1)}$$

Observations are not related to the previous states and the reward is only determined by the state where the agent takes the action, therefore, the S, S^0 and o in $P(ojS, a, S^0)$ and $P(rjS, a, S^0, o)$ can be omitted. $P(S^0jS, a)$, $P(ojS^0, a)$ is the transition matrix T^a and the observation matrix O^{ao} respectively which have been mentioned in Eq. 3.7. $P(rjS, a)$ is the likelihood of getting a reward r in each of the states by taking the action a.

$$P(a_1, o_1, r_1, ...a_n, o_n, r_n, jh) = b(s) \prod_i [(T^{a_i} O^{a_i, o_i}) \quad R^{a_i, r_i}] \qquad \text{(Eq. 4.2)}$$

Eq. 4.2 can be used to calculate the likelihood of a multi-step test. In Eq. 4.2 $R^{a_i r_i}$ is an S S diagonal matrix in which each element on the diagonal denotes the likelihood of getting the reward r_i on taking the action a_i on each of the states and is the same in Eq. 3.7. The equations for computing the m_1 and M_{aor} remain the same as the original PSR model. In conclusion, model construction in highly dynamic environments require the model to record sequences of action-observation-reward units instead of only action-observation units. The main process however remains the same.

4.2 Combination of Distributional Reinforcement Learning with PSR model

In this section we propose a way to combine DRL with PSR to train an agent on a partially observable environment. The rst scenario describes how to train the agent's learning algorithm (DRL) using data sampled directly from the environment simulation.

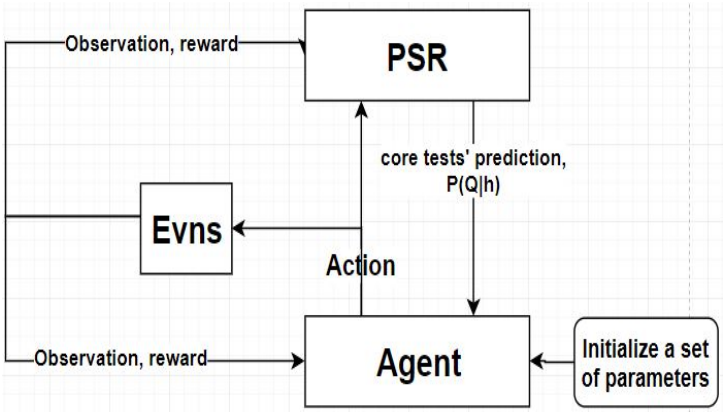


Figure 4.1: Interactions between environment and PSR model

However, sampling data directly from the environment can be computationally expensive. To address this issue, we propose a way to train the learning algorithm using data sampled from the PSR model of the environment, in the second scenario.

4.2.1 DRL with PSR on data collected from actual environment's simulation

In the PSR model the agent tries to understand its environment using likelihoods of only completely observable quantities of observations, actions and rewards; known as core tests. These core tests act as the replacement of belief vectors that are used in the POMDP framework.

Algorithm 2 outlines the process by which the agent learns to make sequential decisions in a partially observable environment modelled using PSR theory. L is a collection of trajectories where each trajectory l is a sequence consisting of action-observation (or action-observation-reward) units u. For every trajectory the PSR model is instantiated with ω as the predictive vector on null history. ω denotes $P(Qjh)$ in vanilla PSR model, X in TPSR model and c in CPSR model respectively. Each unit in the given trajectory is then used to update ω using Eq. 3.4. The value function Z is then updated using Eq. 4.3.

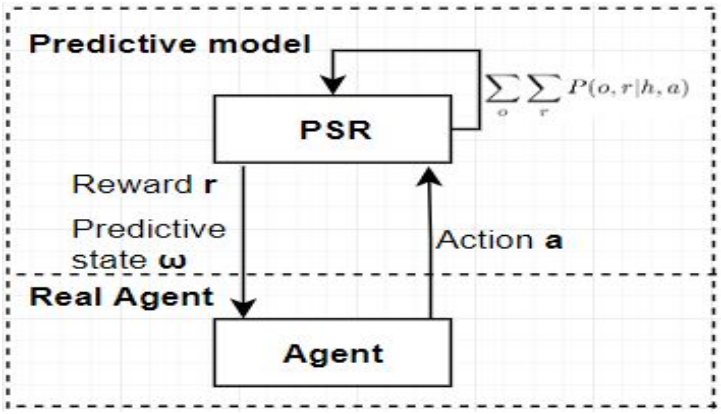


Figure 4.2: Interaction with PSR model only

Algorithm 2 Combination of PSR and reinforcement learning

Input: a collection of trajectories L
Output: a value function Z based on predictive vectors ω
taking Algorithm 3 to obtain PSR model S based on L
for each trajectory l 2 L **do**
  reset PSR model
  for each u 2 l **do**
    ω_0 is the current predictive vector of S
    update S with a action-observation unit u
    ω_1 is the current predictive vector of S
    $Z(\omega_0, a) = r + \gamma \max_{a2A} Z(\omega_1, a)$
  end for
end for
Using Regression method to approximate $Z(\omega, a)$

$$Z(\omega_i, a) = r + \gamma Z(\omega_{i+1}, a)$$
$$\omega_{i+1} = P(Q|h_{i+1}), a \qquad argmax_a Z(\omega_{i+1}, a) \tag{Eq. 4.3}$$

4.2.2 DRL with PSR on data collected from PSR model's simulation

In Algorithm 2, the data for training DRL is collected from the real environment, however interaction with the environment is often very costly. If there exists a high quality PSR

model of the environment, rewards and observations can be collected from the PSR model itself, instead of sampling from the real environment. Each time when PSR model receives an action-observation (or action-observation-reward) unit, PSR model can give a set of likelihoods for the possibility of occurrence of any test in the next step. The agent can thus use a batch of action-observation (or action-observation-reward) units for training. The architecture for using the PSR model itself as the simulation environment is presented below in Figure 4.2. The bellman update changes from Eq. 4.3 to Eq. 4.4 when DRL is trained using data from the PSR model of the environment.

$$Z(\omega, a) = \sum_{o2O, r2R} P(o, rja, h)(r + \gamma Z(\omega_{aor}, a\ )) \qquad \text{(Eq. 4.4)}$$

4.3 Conclusion

In this chapter, we explained how to reinforce the capability of PSR models by introducing the reward signal as well as how to combine the PSR model with di erent learning algorithms to aid the agent to learn in a dynamic partially observable environment. In the next chapter we will introduce the di erent experimental environments.

Chapter 5

Experiment

We experiment with several partially observable games to observe the performances of PSR models with di erent learning algorithms. First, we establish the usability of PSR models by testing them with Q learning on simple environments. Thereafter we implement PSR with DRL in simple environments to investigate the in uence of DRL. Lastly, we test the robustness of the PSR+DRL methodology by extending it to complex environments. The di erent experiment environments/games are: 1) Tiger95, 2) NiceEnv, 3) Shuttle, 4) Maze, 5) Standing Tiger and 6) PacMan.

In all the above mentioned experiments/games, avgValue of a \game" Eq. 5.2 is used to evaluate the agent's policy. Every \game" except PacMan ends after the agent takes 50 actions. In PacMan the \game" ends when the agent is caught by a ghost, when the agent collects all possible \points" or when the agent has taken 100 actions.

The avgValue of a \game" is the average cumulative rewards that the agent obtains in a game. In PacMan, except avgValue of a \game", avgLength of a \game" also contains valuable information. It denotes the average number of actions that the agent takes in a game. In PacMan, if the agent lives longer, the likelihood of wining the game and the rewards that the agent obtains during the game would be higher.

The quality of the PSR models are computed in terms of the KL-divergence which compares a set of predictions by TPSR or CPSR model with the actual likelihoods from

vanilla PSR model, Eq. 5.1. This measurement has been named as \predLoss". In Eq. 5.1 e is an event (observation-action unit or observation-action-reward unit), E is a collection of events, p_e is the prediction of the event e in TPSR or CPSR model and q_e is the prediction of the event e in vanilla PSR model.

To smooth the lines in \predLoss", the value on each epoch is averaged with values of three epoch before and include itself, as show on Eq. 5.3. n is the number of points in x-axis.

$$D_{KL}(p_e jj q_e) = \sum_{e2E} p_e \log \frac{p_e}{q_e} \quad \text{(Eq. 5.1)}$$

$$avgValue = \sum_{i=0}^{49} r_i \quad \text{(Eq. 5.2)}$$

$$p_i = \frac{p_{min(i\ \ 3,0)} + p_i}{2}, i\ 2\ [0, n) \quad \text{(Eq. 5.3)}$$

5.1 Tiger95

Tiger95 is a partially observable game in which there are two doors in front of the agent and a tiger is randomly placed behind a door that the agent is unaware of. There are three actions available to the agent; opening left door (Open-L-door), opening right door (Open-R-door) and listening (Listen). If the agent opens a door with a tiger behind, it is punished with -100 reward whereas if the agent opens a door without a tiger behind, the agent wins a reward of 10. Thus, the goal of the agent is to learn a policy to understand how to choose the door which does not have a tiger behind it. To aid this choice, the agent relies on the sound of the tiger. However, unfortunately, the \Listen" action is not perfect and helps locate the tiger with only 85 percent probability. For example, even if the tiger is behind the left door, and thus the current state is \Tiger-Left-St", there is still a 15 percent chance that the agent receives the observation \Tiger-Right-ob" on taking the \Listen" action. The cost of listening is -1 and the tiger cannot move while

the agent is listening. The \game" ends and manually resets after 50 actions. Once the agent chooses a door and receives a corresponding reward, the game resets and thus shifts to \Tiger-Left-Door" or \Tiger-Right-Door" state randomly. Thus, \Open-L-door" and \Open-R-door" are the internal reset actions.

Actions: Open-L-door, Open-R-door, Listen.

Observations: Tiger-Left-Ob, Tiger-Right-Ob.

States: Tiger-Left-Door, Tiger-Right-Door.

Rewards: 10.0, -100.0, -1.0.

In Tiger95, avgValue of a \game" is used to measure the performances of each agents' policy. The avgValue of Tiger95 is the the same as mentioned above. The quality of TPSR and CPSR models that are learned directly from data is evaluated with the help of \predLoss" measurement mentioned above.

5.2 NiceEnv

NiceEnv is a partial observable game in which there are ve latent states. The agent achieves the maximum reward of 100 when it stays in the middle state. However, if the agent is in the state that is next to the middle state, the agent gets a reward of -10. In case the agent is in the leftmost or rightmost state, the agent gets a reward of -50. The agent has three available actions; move-left, stay and move-right. However, while taking an action, the agent can fail to move or even move in the wrong direction with some likelihood. The likelihood of an erroneous move depends on the agent's current position and is illustrated in Figure 5.1. Whenever the agent takes an action, it receives an observation to illustrate its current state which itself is accurate only 80 percent of the time.

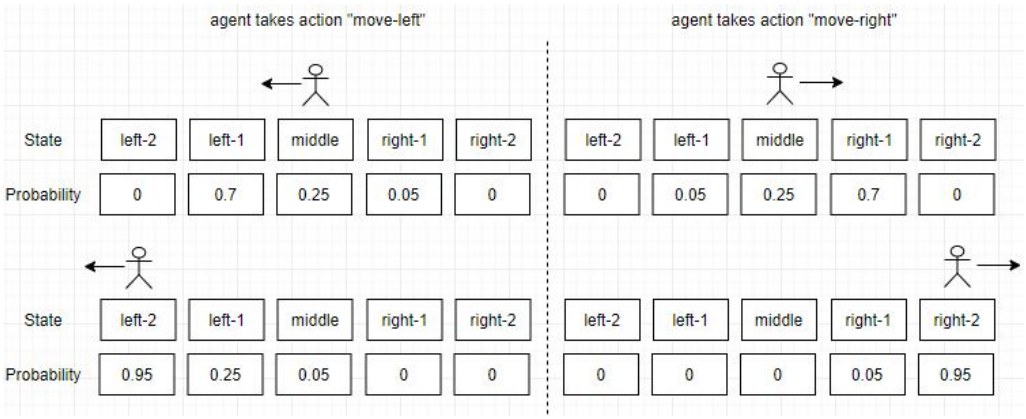


Figure 5.1: Agent's transition

Actions: move-left, stay, move-right.

Observations: there are ve observations corresponding to each state.

States: there are ve latent states from left to right.

Rewards: -50.0, -10.0, 100.0

In NiceEnv, avgValue of a \game" still remains the same meaning

5.3 Shuttle

The Shuttle is a partially observable game that has been described in [12]. In this game, there are two ports in a free ocean and a ship transports goods between them. Each time when the ship successfully transports the goods from the most recent visited port to the least recent visited port, it wins 10 reward. When the ship begins to transport goods, it should rst go forward towards the destination and in the next step as it has reached in front of the station; it should turn around. It must then enter the station by moving backward. If the ship tries to enter the station while facing the port, the ship collides with the port. If the ship collides with the port, it gets -3 reward. However, if the \Backup" action fails, the ship either remains in the same location or randomly turns

around. The ship has limited observation as even while it is near the port and is facing it, the ship only has a 70 percent chance to see it..

Actions: Turn-Around, Go-Forward, Backup.

Observations: ob-1, ob-2, ob-3, ob-4, ob-5.

States:st-1, st-2, st-3, st-4, st-5, st-6, st-7, st-8.

Rewards: -3.0, 10.0

The observations are: 1)See LRV (Least Recently Visited station) forward, 2)See MRV (Most Recently Visited station) forward, 3)See that we are docked in MRV, 4)See nothing and 5)See that we are docked in LRV. The states are: 1)Docked in LRV, 2)Just outside space station MRV, front of ship facing station, 3)Space facing LRV, 4)Just outside space station LRV, back of ship facing station. 5)Just outside space station MRV, back of ship facing station, 6)Space, facing LRV, 7)Just outside space station LRV, front of ship facing station, 8)Docked in MRV.

In Shuttle, the avgValue of a \game" still remain the same.

5.4 Maze

Maze is a navigation game in a partially observable environment. The Maze map is presented Figure 5.2. I the gure black squares stand for an obstacle or wall, the number represents the state ID. In this environment, the observation \left" indicates the existence of a wall in the agent's left. The observation \right" indicates the existence of a wall in the agent's right. The observation \neither" means that there is no wall in the agent's left or right. The observation \both" means that there is a wall in agent's left as well as right. The agent cannot see if there is a wall in agent's up and down direction. There are a total of 11 states, as shown in Figure 5.2. If the agent is in state \st-3", it receives the

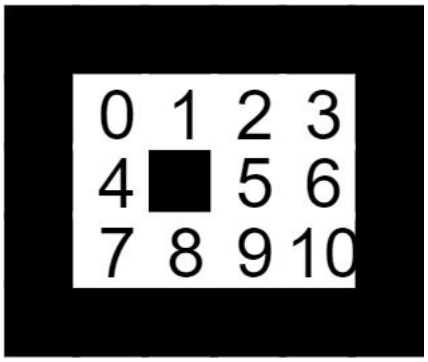


Figure 5.2: Maze

observation \good". If the agent is in state \st-6", it receives the observation \bad". The agent starts at any state except the state \st-3" and state \st-6". The agent is allowed to move left, right, up and down. When the agent arrives in state \st-3" and takes action \obtain", it gets a reward of 10. However, if the agent takes action \obtain" in state \st-6", it is penalized with a rewards of -100. Except for state \st-3" and state \st-6", the action \obtain" does not lead to any penalty or reward but costs 0.04 cost and does not lead to a shift in the agent's state. No matter where the agent takes action \obtain", it receives one out of the following observations randomly: left, right, neither, both. As each action costs 0.04, the agent should learn a policy to reach state \st-3" in minimum possible number of actions. Limited vision adds di culty in achieving this goal.

In Maze, each time the agent reaches state \st-3" or \st-6", the agent is randomly shifted to any state except \st-3" and \st-6".

Actions: Move-up, Move-down, Move-right, move-left, obtain.

Observations: left, right, neither, both, good, bad.

States: st-0, st-1, st-2, st-3, st-4, st-5, st-6, st-7, st-8, st-9, st-10.

Rewards: -0.04, 10.0, -100.0

5.5 StandTiger

StandTiger is an advanced version of Tiger95. This game has three doors and the agent can have two positions. The agent can either be sitting or standing. At the beginning of the game, the agent is sitting. The agent can switch to \standing" status by implementing action \Stands up". The agent can estimate where the tiger is by listening if and only if the agent is sitting. Otherwise, the agent receives the observations of \tiger-stand-left", \tiger-stand-middle", \tiger-stand-right" in equal likelihoods. This does not reveal any useful information but cut o the link between adjacent actions and observations. Moreover, the agent must stand up before opening a door. When the agent is standing up, it receives a random observation from [\tiger-stand-left", \tiger-stand-middle", \tiger-stand-right"]. If the agent tries to open a door while sitting, it causes a 1000 penalty. As in Figure 5.3, when the tiger is behind the middle door and the agent listens, the agent has 80 percent chance to nd out that the tiger is actually behind the middle door and 10 percent chance of thinking that the tiger is behind left or right door. When the tiger is behind the left or right door and the agent listens, the likelihood of nding out the actual place of the tiger by listening decreases to 75 percent. The likelihood of thinking that the tiger is behind the middle door is 15 percent and that the tiger is behind any other door is 10 percent. If the agent opens a door without a tiger behind, the agent wins 30 reward, whereas the agent is penalized with 100 reward upon choosing the door with a tiger behind. Opening a door gives the agent a random observation between tiger-sit-left, tiger-sit-middle, tiger-sit-right. The action \stands" and \listen" costs 1 reward. In this environment, \Stand up" and \Open a door" internally reset the environment but \Stand up" action does not cause the tiger to shift. Thus it is not complete reset. The common property of these two actions makes the environment being non-Markov, which means that the next action and observation after these two actions does not have any relation with the current action and observation. This however increases the di culty of

Figure 5.3: Observation's likelihood on StandTiger domain

constructing TPSR and CPSR models from data and requires the agent to know at least two step history.

Actions: Open-L-door, Open-M-door, Open-R-door, Listen, Stands up

Observations: Tiger-Sit-Left, Tiger-Sit-Middle, Tiger-Sit-Right, Tiger-Stand-Left, Tiger-Stand-Middle, Tiger-Stand-Right

States: Tiger-Left-Sit, Tiger-Middle-Sit, Tiger-Right-Sit, Tiger-Left-Stand, Tiger-Middle-Stand, Tiger-Right-Stand

Rewards: -1000.0, -1.0, 30.0, -100.0

In StandTiger, we still use avgValue of a \game" to evaluate the agent's policy and takes \predLoss" to measure the TPSR and CPSR models' quality.

5.6 PacMan

We use a partially observable version of a PacMan game [49]. The maze is a grid world and many pills are randomly distributed across the area where x-axis is not in [5,13] and y-axis is not in [6, 12]. The agent has to search for all of pills and meanwhile avoids to

power pill door

Figure 5.4: Tools in PacMan Maze

be caught by four ghosts who are wandering around the maze. At the most of time, the ghost are unconsciously moving, while if the distance between a ghost and the agent is less than 5 in Manhattan metric, the ghost are crazy to catch it, which the agent should be paid attention to. Due to the partial observability, the agent is only allowed to see the environment surrounding itself. The agent can understand if there are walls around it and it can see pills and ghosts if the food and ghosts are in its direct line sights. Another ability of the agent is that if the food is near the agent within 4 Manhattan distance, it can smell it but cannot gure out the exact location. When initializing the maze, each place has 0.5 probability to have a pill. The reward of eating a pill for the agent is 10 and if collecting all pills, the agent will be awarded with extra 100 reward. However, if the agent hits the wall, it will be penalized with 10 cost and if the agent is caught by a ghost, it will be penalized with 50 cost. Each action also costs the agent 1. The goal of the agent is to end the game with maximum rewards. The Maze structure is like Figure 5.5. In Figure 5.4, if the agent eats the power pill, the agent is not afraid of ghost during 15 seconds. If the agent passes through the doors, the agent can go to the another side immediately.

In PacMan, we do not have the ground trues of some event's likelihoods. Thus, we cannot calculate \predLoss". We measure the agent's performance by avgValue of a \game" and avgLength of a \game". In PacMan, the game ends when the agent is caught by ghosts, collects all \points" or the agent has taken 100 actions. Beside comparing the overall rewards during a game, the number of steps that the agent takes also reveals the

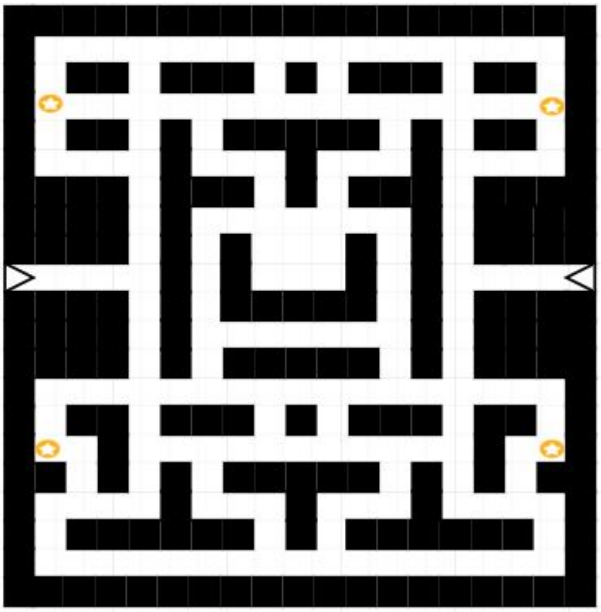


Figure 5.5: PacMan Maze

lifetime of the agent. If an agent acts more steps, it at least means the agent desires to survive and try to escape from ghost.

5.7 Conclusion

In this chapter, we introduced six partially observable environments which are Tiger95, NiceEnv, Shuttle, Maze, StandTiger and PacMan. The first five environments are small and simple, and are mainly used to prove the usability of the different PSR models. The last environment is to test whether the different PSR models can be extended to large and complex environments. Not all PSR models can be applied to all the different environments. In the next chapter, we will introduce the implementation details of the different methods.

Chapter 6

Implementation Details on PSR models' construction

In this chapter, we will introduce how to build a PSR model directly from data. In TPSR and CPSR methods, certain hyper-parameters and details are extremely critical and have considerable in uences on prediction quality but are also easy to oversee, and thus special attention will be drawn to them in this chapter.

Algorithm 3 Constructing PSR model

Input: a collection of trajectories L
Output: a set of M_{ao}, a, o 2 A O, m_1 and Core tests Q
TPSR: Q = X, m_1 = b_1
CPSR: Q = , m_1 = c_1
taking Algorithm 4 to gain the matrix $P(T, H)$
U, S, V = svd($P(T, H)$)
TPSR: X = $U^T P(T, H) N^{y}$, $b_1^T X$ = e^{jHj}
CPSR: = U^T $_{T,H}$, c_1^T = $_H$
taking Algorithm 5 to gain M_{ao}, a, o 2 A O,

The outline of the procedure is provided in Algorithm 3. The rst step is to collect the data, initialize the variables and ll-in the Test-History matrix, $P(T, H)$, by scanning the trajectories. The details of this step are captured in Algorithm 4. In theory, the Test-History matrix should include tests and histories of all possible lengths. However, in practice, using extremely long tests and history is not only computationally expensive

but can also harmful instead of helpful. This is due to the fact that a high dimensional Test-History matrix will be sparse and will contain a lot of noise and useless information. The length of tests must be at least as long as the length of core tests but it also strongly dependent on the selected game. For example, in StandTiger, the maximum length of a test is 1 and the maximum length of history is two, since in StandTiger, the outcome of opening a door is determined by what the agent listens which is at least one step before. Therefore, to overcome the di culty, the agent should remember two steps of history at least. This intuition was validated experimentally, where using two-steps histories provided better results.

Algorithm 4 Filling in Test-History Matrix

```
Input: a collection of trajectories L,
       a test dictionary testDict, a history dictionary histDict
Output: A Test-History matrix P(T, H)
initialize P(T, H)
TPSR: P(T, H) = P(T, H)
CPSR: P(T, H) =  T,H
for each trajectory l 2 Z do
  ζao = list()
  for each action-observation unit u in l do
    ζao.add(u)
    for i 2 (0:len(ζao)) do
      t = ζao[i + 1:len(ζao)]
      tid = testDict[t]
      h = actObs[0:i]
      hid = histDict[h]
      TPSR: P(T, H)[tid, hid] = P(T, H)[tid, hid] + 1
      CPSR:  T,H =  T,H + φ(t)  φ(h)
    end for
  end for
end for
return P(T, H)
```

In Algorithm 4, Every time a new unit u (action-observation or action-observation-reward) is added to the current sequence, the model retrieves all possible tests t under

the constraint of maximum test length, I_{max_t}, in the current sequence, that ends with the given unit u. The part of the current sequence present before a retrieved test is known as the prior sequence of that test. The prior sequence thus acts as the history for the particular test. Every PSR model has a maximum length of history, I_{max_h} and a maximum length of test, I_{max_t} that it can work with. In case the prior sequence of a test is longer than I_{max_h}, its beginning is truncated to make its length equal to I_{max_h}. TPSR builds up a *jTj jHj* matrix where each row indexes a test and each column indexes a history while the CPSR model projects each test and history into a low dimensional vector and the Test-History matrix, $_{T,H}$ is the sum of outer products of all legal tests and legal histories.

Algorithm 5 Computing aoMats

Input: a collections of trajectories L
a test dictionary testDict, a history dictionary histDict
Output: the set of M_{ao}, a, o *2* A O
Initialize a set of matrices M_{ao}, a, o *2* A O
TPSR:M_{ao} = B_{ao}
CPSR:M_{ao} = C_{ao}
for each trajectory l *2* L **do**
  ζ_{ao} = list()
  for each action-observation unit u in l **do**
    ζ_{ao}.add(u)
    for i *2* (0:len(ζ_{ao})) **do**
      h = ζ_{ao}[0:i-1]
      h_{id} = histDict[h]
      t = ζ_{ao}[i+1:len(ζ_{ao})]
      t_{id} = testDict[t]
      ao = ζ_{ao}[i]
      TPSR: B_{ao} = B_{ao} + U^T[:, t_{id}] $(U^T P(T,H))^y$[h_{id}, :]
      CPSR: C_{ao} = C_{ao} + $U^T \phi(t_{id}) \phi^T(h_{id})(U^T b$ $_{T,H})^y$
    end for
  end for
end for

According to Eq. 3.29, each aoMats M_{ao} is computed using Algorithm 5. The process is similar to Algorithm 4 except that the sequence is equal to h + ao + t instead of h + t.

In order to obtain a good quality of predictions in TPSR model, choosing a proper SVD-dimension is important. SVD-dimension is the dimension of predictive vectors. If the SVD-dimension is too large, the predictive vectors would contain too much overlapped information. If SVD-dimension is too small, the predictive vectors would lose key information. Because it is also considered as the number of core tests, it is not greater than the number of states. In CPSR model, beisde SVD-dimension, the proj-dimension is a important parameter. It denotes the length of representations of each test and history. Finding a proper proj-dimension is a key step of obtaining good PSR models.

In this chapter, we presented the implementation details of how to construct the di erent PSR models as well as how to employ distributional learning algorithm on the di erent PSR models. As we saw earlier, due to high complexity of the TPSR and CPSR models, some hyper-parameters have considerably in uence on their representation and therefore must be tuned carefully. In the next chapter, we will discuss the performance of all the PSR models.

Chapter 7

Result Analysis

In this chapter, we show the performances of those games based on di erent PSR models with di erent learning algorithms. We rstly look at the relationship between the quality of predictions and the performances of avgValue of a \game" in rst ve games. Using a set of events that are manually de ned to check the accuracy of predictions, \predLoss" of TPSR and CPSR models at each data-expansion epoch is calculated by Eq. 5.1. Then, we compare the performances of training the agent between \ xed model" and \un xed model" by Fitted Q learning algorithm. \ xed model" means that the PSR model has nished training and will thus remain xed in the further epochs. \un xed model" on the other hand is a model that is still being trained on the next epochs. Next, we see if the DRL algorithm can promote the performances on top of PSR models. Besides, in the section 4.2.2, we mention that PSR models can be extended to include rewards. In the

	Tiger95	NiceEnv	Shuttle	Maze	StandTiger	PacMan
maxQIteration	30	30	30	30	40	100
epsilon	0.5	0.3	0.3	0.3	0.5	0.3
samplingBatch	1000 games	2000 games	2000 games	1000 games	2000 games	5000 games
initialSampling	5000 games	3000 games	3000 games	5000 games	20000 games	5000 games
evaluation	2000 games	1000 games	1000 games	1000 games	1000 games	500 games
γ	0.95	0.95	0.95	0.95	0.95	0.95
v_{max}	100	600	100	50	600	200
v_{min}	-100	-2000	-100	-100	-2000	-150
numAtoms	11	11	11	11	11	30

Table 7.1: Learning algorithm Setting for each games

	Tiger95	NiceEnv	Shuttle	Maze	StandTiger	PacMan
svdDim	2	5	7	11	6	5, 20(rewards)
projDim	250	150	150	150	250	250
maxTestLen	1	2	1	1	1	1
maxHistLen	1	2	2	2	2	2
RandomInit	true	false	false	true	true	false

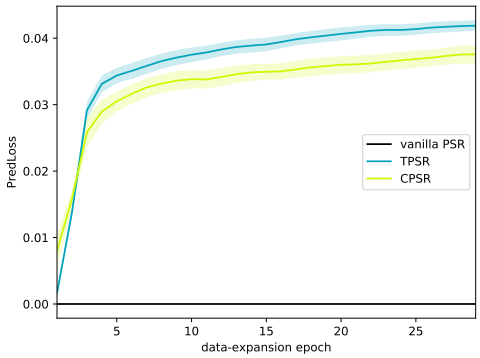
Table 7.2: PSR model Setting for each game

last, we can check if PSR models with rewards would be better than original PSR model and how the performances of training an agent on data generated from PSR models. In order to check the generality of DRL with PSR models, we train an agent by DRL algorithm on PSR models in PacMan environment.

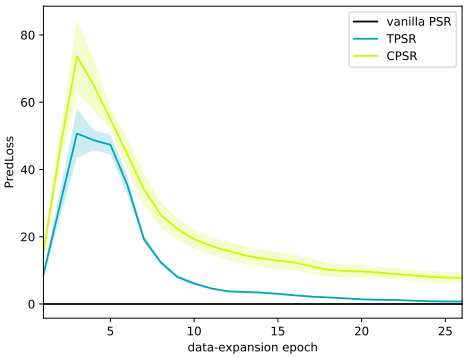
The \data-expansion epoch" denote a due time to sample a new batch of data based on epsilon-greedy method. Initially, the agent takes random policy. Table 7.1 shows the di erent hyper-parameters on di erent games and Table 7.2 shows the di erent hyper-parameters on di erent PSR models.

7.1 Effect of model quality

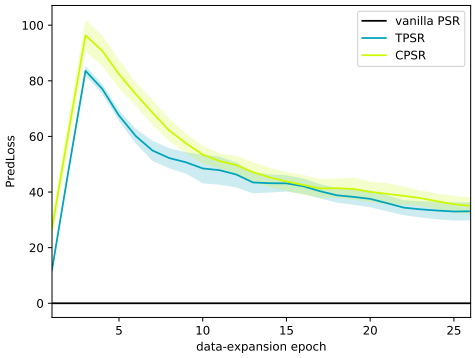
In the rst epoch, the TPSR and CPSR models are dynamically built using data generated by the random policy. In the subsequent epochs, the models are reconstructed based on all previous data as well as the new batch of data. The new batch of data is generated using the latest policy with the \epsilon-greedy sampling" strategy. The shift from the random policy to the epsilon-greedy policy introduces a bias in the probability distribution over the events, as the agent begins sampling \higher value" actions substantially more than \lower value" actions. The frequency of \lower value" actions thus becomes quite low in the data, and when these frequencies are normalized to calculate the likelihood over events, a substantial error is introduced in the likelihood estimates. This is re ected in Figure 7.1 where CPSR and TPSR models have higher \predLoss" than PSR. \predLoss" is calculated using Eq. 5.1.



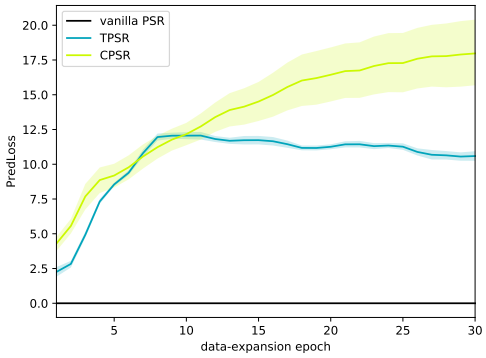
7.1.a: Tiger95



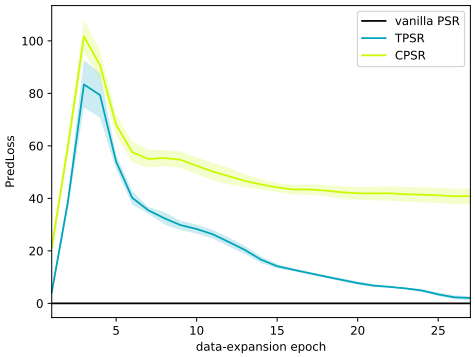
7.1.b: Shuttle



7.1.c: Maze



7.1.d: NiceEnv



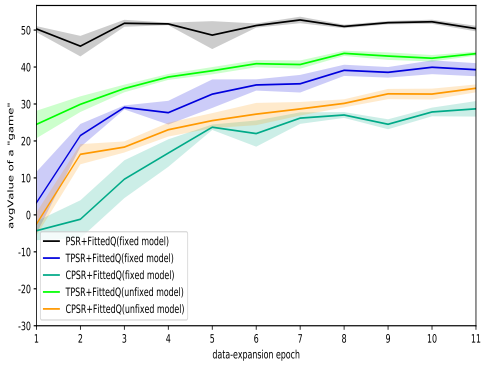
7.1.e: StandTiger

Figure 7.1: \predLoss" of di erent PSR model at each data-expansion epoch

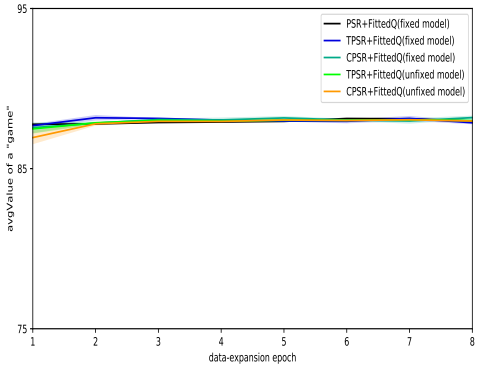
The high \predLoss" of CPSR and TPSR during training is because small batches of biased data (data contains larger frequency of \higher value" actions) in every epoch a ects these two models while vanilla PSR is directly constructed using the POMDP model of the same environment. Due to consciously sampling, the probability distribution that the vanilla PSR model learns from POMDP model does not remain the same as the probability distribution that TPSR model and CPSR model learn from the biased data. In other words, the world that the vanilla PSR model understands on the basis of prior information from POMDP model is not the same as the world that TPSR model and CPSR model learn from the sampling data. Thus, the \predLoss" increases when the agent starts to sample, using epsilon-greedy method. This phenomenon is however, not a problem, since at the end of training, when TPSR and CPSR have been exposed to large amount of data, they are able to ful ll their responsibility of being a good representation of the environment.

Compared with CPSR model, TPSR builds a better probability distribution of the events in general as it uses the Test-History matrix directly without losing information on compression. This can be noted by observing the \predLoss" which is lower for TPSR model in all experiments except Tiger95 where the di erence between TPSR and CPSR models is very small. With more data, the performance of TPSR and CPSR will become comparable.

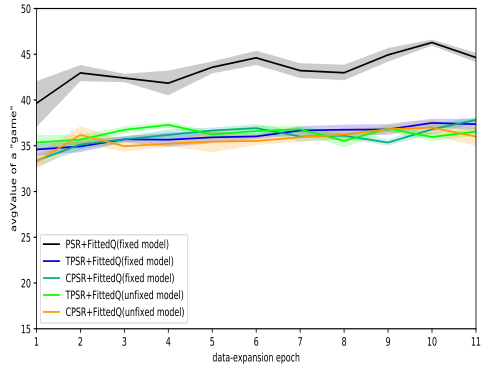
In Figure 7.2, it can be seen that the di erent PSR models have large in uence in the StandTiger game where xed PSR models perform better than un xed PSR models. This in uence of xed vs un xed models is not substantial in the other games. In these other games, as expected, TPSR performs better than CPSR.



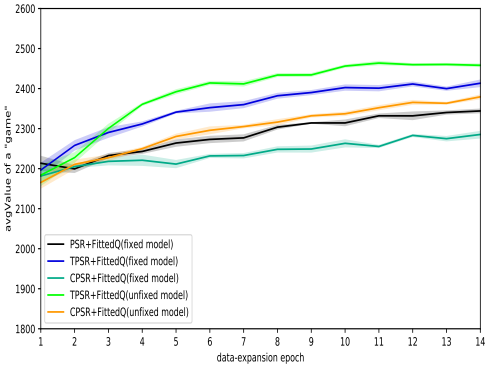
7.2.a: Tiger95



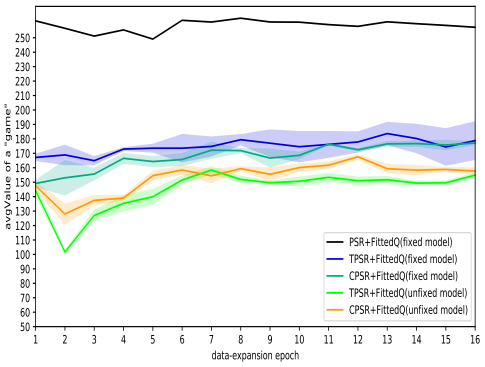
7.2.b: Shuttle



7.2.c: Maze

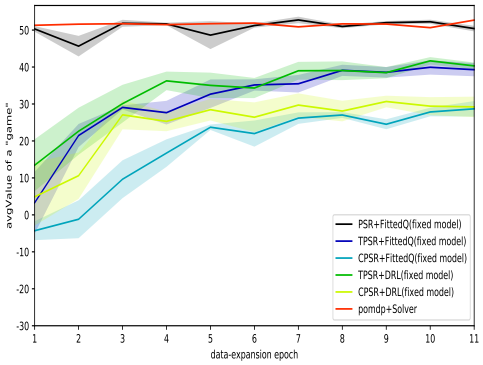


7.2.d: NiceEnv

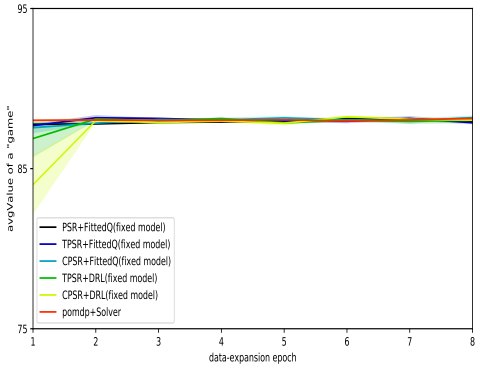


7.2.e: StandTiger

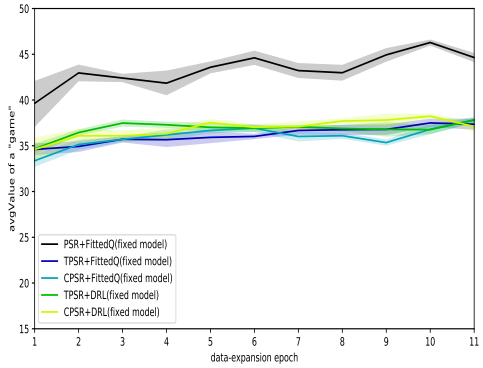
Figure 7.2: Performances of \ xed" and \un xed" PSR models at each data-expansion epoch



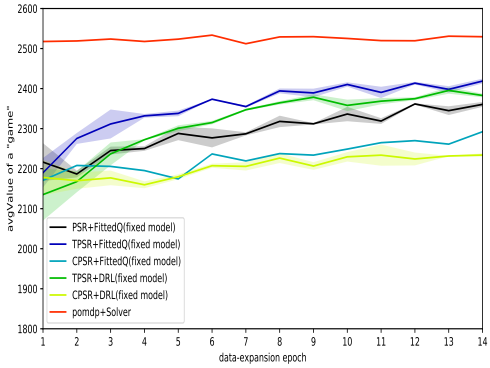
7.3.a: Tiger95



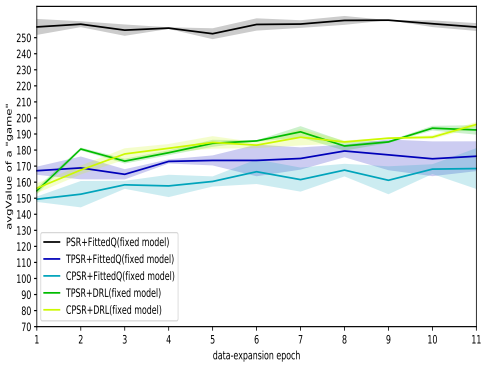
7.3.b: Shuttle



7.3.c: Maze



7.3.d: NiceEnv

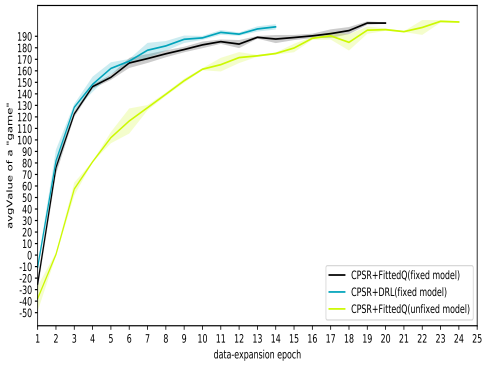


7.3.e: StandTiger

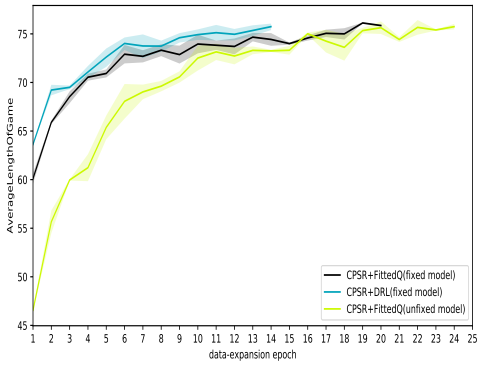
Figure 7.3: Comparison DRL and Fitted Q algorithm on \ xed" PSR model

7.2 Effect of the different learning algorithm

An advanced POMDP solver, which combines multiple POMDP algorithms is imported for comparison with the PSR model. It can be seen in Figure 7.3 that the POMDP solver is able to perform better than the PSR model only in niceEnv. Furthermore, in environments such as Maze and StandTiger, the POMDP solver struggles to even nd the solution. Thus, it is hard to apply in such little complex environments. Figure 7.3 mainly illustrates how well the DRL and Fitted Q algorithms are on di erent PSR models. Except in niceEnv, DRL learning algorithm are no worse than Fitted Q learning algorithm. It is possible that those ve partially observable environments cannot unleash the true power of DRL. We further compare those two learning algorithms on PacMan environment. PacMan is a complicated game with large space of observations and TPSR model is unavailable in this environment. In Figure 7.4, the performance of DRL with CPSR (xed model) converges faster to upper bound of the performance, which means distributional learning algorithm is more advantages in such dynamic environment which has a large randomness on nal outcomes. Besides, the comparison between CPSR + FittedQ (un x model) and CPSR + FittedQ(xed model) demonstrates that training a learning algorithm (here, FittedQ) using a completely trained CPSR model gives better performance than learning the environmental representations and action expectations simultaneous. The \ xed model" at here is not the CPSR model at the rst epoch. It is a model that keeps rebuilding until 23 epoch. Since PacMan is a complicated environment, modelling such environments are much harder than constructing PSR models in previous POMDP games. On the other hand, if the environment space is small, the agent is able to explore every possible situations by random policy, while in PacMan, to explore every space of the PacMan Maze, only using data generated by random policy is impossible and thus in order to modelling the PacMan environment, CPSR model has to use the data that is consciously sampled.



7.4.a: Average Value of one \game"

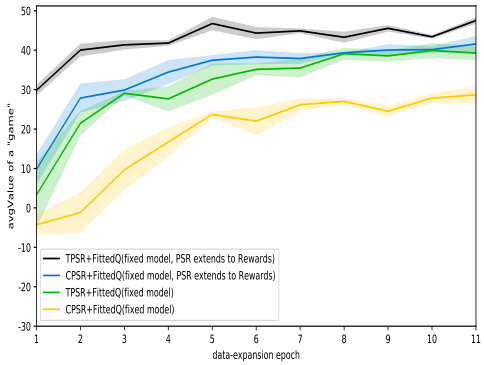


7.4.b: Average steps of one \game"

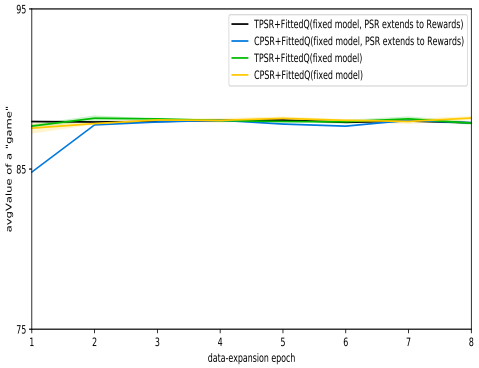
Figure 7.4: Comparison Fitted Q and DRL in PacMan

7.3 Influence of including rewards in the PSR model

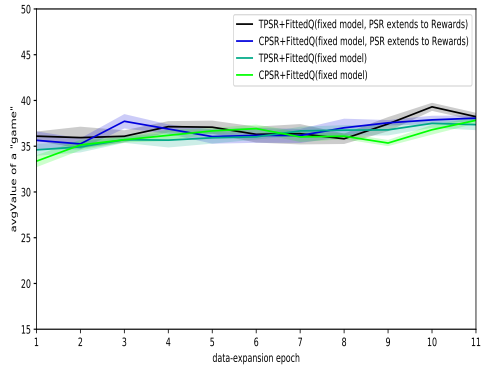
As shown in Figure 7.5, including rewards in PSR models is bene cial. In niceEnv, using a PSR model which is built using observations and rewards considerably promotes the performance because the rewards are strongly related with the state and can help the agent to avoid being cheated by observations. In Tiger95, the performance of the agent trained on PSR model with rewards is better and the performance also converges quickly. Boost in agent's performance in NiceEnv and Tiger95 is because in these environments, the observations are unable to indicate the \goodness" of a state. The agent needs additional information in the form of rewards to be able to make this distinction. On the other hand, no boost in performance is seen in Maze and Shuttle as observations in these environments are designed to indicate how good a state is. For instance, in Maze, when the agent reaches the correct destination it receives the observation \good" and when it reached the wrong destination it receives the observation \wrong". On the other hand, in Shuttle, when the agent approaches the LRV port, it receives the observation \See that we are docked in LRV" that tells the agent that the agent has arrived correctly at



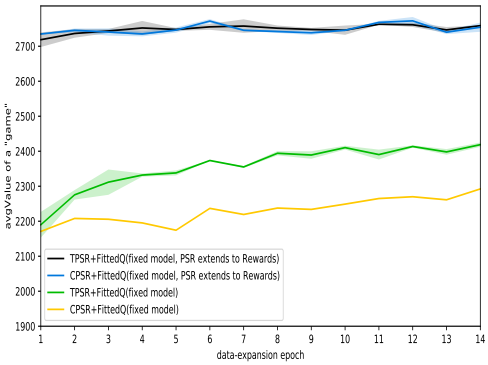
7.5.a: Tiger95



7.5.b: Shuttle

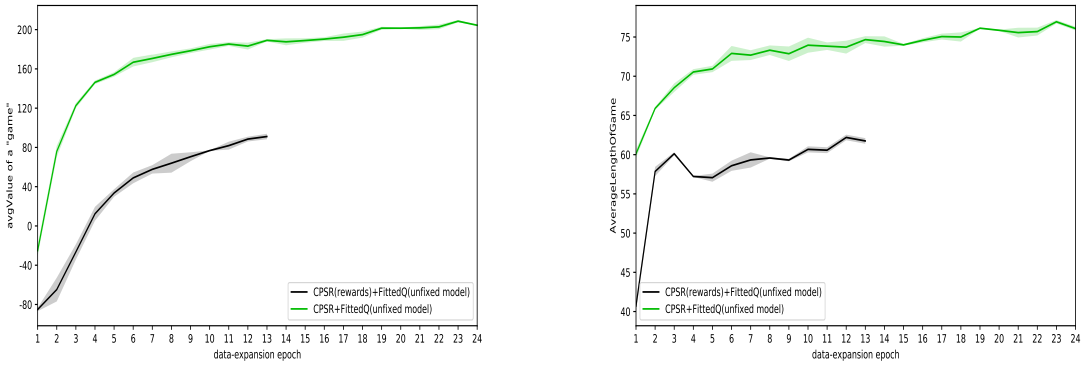


7.5.c: Maze



7.5.d: NiceEnv

Figure 7.5: In uence of including rewards in the PSR model



7.6.a: avgValue of a \game"

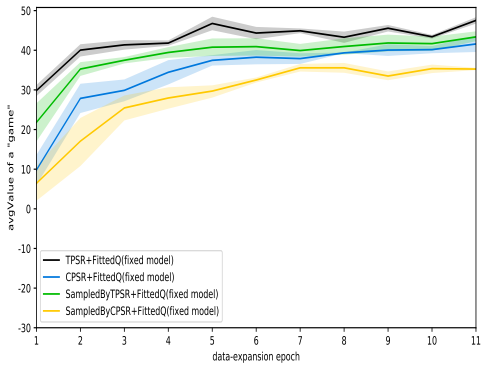
7.6.b: avgLength of a \game"

Figure 7.6: E ective of PSR model with rewards in PacMan

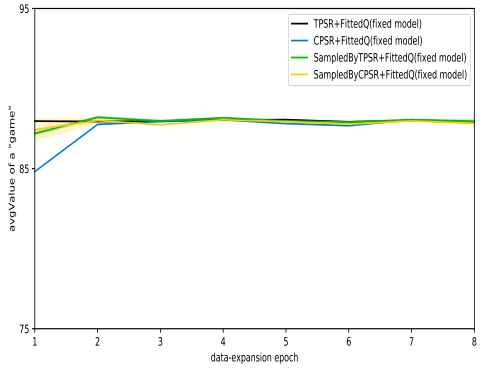
the required port. Those observations contain the same information as the rewards and thus there is no extra information being exposed even if PSR models consider rewards.

However, training PSR models with reward does have some side-e ects. For instance, introducing extra information into PSR models exponentially increases the complexity of construction. The main cause is that the number of M_{aos} matrices is exponentially increasing. For instance gure 7.6 shows that training the CPSR model with rewards in PacMan is very hard. Because of this bad quality, the performance of PSR models with rewards increases very slowly. This e ect however is not seen in Tiger95 and NiceEnv games because of their highly simplistic natures.

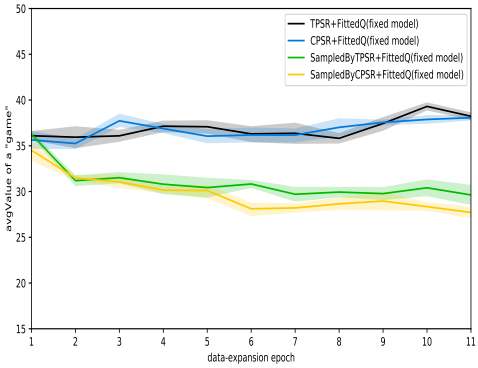
If interacting on the real environment is costly, using PSR model to sample training data is also e ective. This method needs a batch of data rstly to build up a PSR model that includes rewards. Then, the agent collects data by interacting with the PSR model. From Figure 7.7, the performances of training an agent by interacting with PSR models are lower than the performances of training an agent on real environment. But the di erence of the performance is acceptable. It shows that if having a good quality of PSR model that includes rewards, the agent can learn a good policy purely on PSR



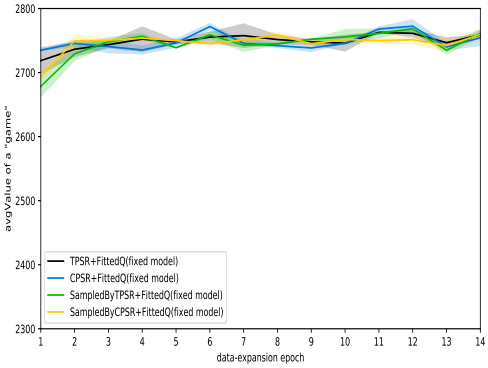
7.7.a: Tiger95



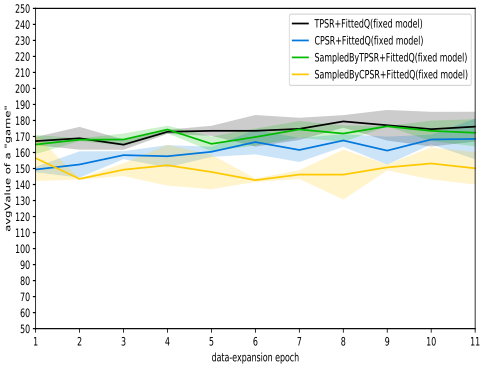
7.7.b: Shuttle



7.7.c: Maze



7.7.d: NiceEnv



7.7.e: StandTiger

Figure 7.7: E ective of training agent by interacting with PSR model

models.

7.4 Conclusion

In this chapter, we discuss three important e ects on the performance of the agent which are; \E ect of model quality", \E ect of the di erent learning algorithms", and \In- uence of including rewards in the PSR model". In section 7.1, we saw that the TPSR models are generally able to create a better representation of the environment than CPSR models. The CPSR models however, are more suitable for partially observable environments where there are large number of observations. In section 7.2, we compared PSR models with pomdp-solver and observed that the PSR models are able to give answers to all environments while pomdp-solver is unable to provide answers in StandTiger and Maze. In dynamic and uncertain environment such as pacman, DRL algorithm enables faster learning as opposed to Fitted-Q algorithm. In section 7.3, we saw that although the introduction of rewards into PSR models enable better representation of the \states", the time complexity for the construction increases exponentially. There thus exists a trade-o between better state representation and time complexity. Furthermore, if we already have a well-trained PSR model with rewards, we can directly train agents on it instead of making the agent re-learn through interactions with the environment; with acceptable performances.

In conclusion, we have made four observations; rst, the compressed PSR (CPSR) model generally creates the weakest representations but has the best performance in large and complex environments; second, the addition of rewards in PSR models improves representation but hurts the time complexity; third, a well-trained PSR model with rewards can take the role of the real environments with which agents can interact and learn; fourth, DRL algorithm is highly suitable in dynamic and uncertain environments.

Chapter 8

Conclusion and Future Work

Currently, we have tested the performance of di erent PSR models with tted Q learning algorithm and distributional learning algorithm.

Amongst vanilla PSR, TPSR and CPSR, vanilla PSR model has the best quality of predictions while CPSR model has relatively the worst quality of predictions. The bad quality of PSR model causes poor performances, which is apparently in large domain. However, the performance gap between vanilla PSR model and CPSR model is not very large if we choose a proper setting of CPSR model. The performance of learning based on CPSR is acceptable even though it is not the best. More importantly, CPSR has overcome the bottleneck of working in an environment with a large number of observations, and thus has a huge advantage in more complex environments.

Even though the performance of TPSR model is better than the performance of CPSR model, it has a large time complexity as it computes SVD decomposition of a high dimension matrix. Therefore unlike CPSR model, TPSR model cannot handle complicated environments. This has been proved empirically in the very complicated game of PacMan where TPSR model does not work while CPSR is able to handle it. However, due to the very high complexity of PacMan, constructing a CPSR model on it is very hard. This is especially true when CPSR model is built using rewards as rewards further add to the complexity.

The performance of distributional learning algorithm and tted Q learning algorithm are not very di erent in simple partially observable games. However, in PacMan, distributional learning algorithm converges faster than tted Q algorithm and has a better performance in environments with large randomness and uncertainty.

References

[1] Douglas Aberdeen et al. Policy-gradient algorithms for partially observable markov decision processes. 2003.

[2] Masanao Aoki. Optimal control of partially observable markovian systems. *Journal of The Franklin Institute*, 280(5):367{386, 1965.

[3] Karl J Astrom. Optimal control of markov processes with incomplete state information. *Journal of mathematical analysis and applications*, 10(1):174{205, 1965.

[4] Leemon C Baird III and Andrew W Moore. Gradient descent for general reinforcement learning. In *Advances in neural information processing systems*, pages 968{974, 1999.

[5] Gabriel Barth-Maron, Matthew W Ho man, David Budden, Will Dabney, Dan Horgan, Alistair Muldal, Nicolas Heess, and Timothy Lillicrap. Distributed distributional deterministic policy gradients. *arXiv preprint arXiv:1804.08617*, 2018.

[6] Marc G Bellemare, Will Dabney, and Remi Munos. A distributional perspective on reinforcement learning. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 449{458. JMLR. org, 2017.

[7] Byron Boots, Sajid M Siddiqi, and Geo rey J Gordon. Closing the learning-planning loop with predictive state representations. *The International Journal of Robotics Research*, 30(7):954{966, 2011.

[8] Leon Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177{186. Springer, 2010.

[9] Michael Bowling, Peter McCracken, Michael James, James Neufeld, and Dana Wilkinson. Learning predictive state representations using non-blind policies. In *Proceedings of the 23rd international conference on Machine learning*, pages 129{ 136. ACM, 2006.

[10] Anthony Cassandra, Michael L Littman, and Nevin L Zhang. Incremental pruning: A simple, fast, exact method for partially observable markov decision processes. In *Proceedings of the Thirteenth conference on Uncertainty in arti cial intelligence*, pages 54{61. Morgan Kaufmann Publishers Inc., 1997.

[11] Anthony R Cassandra, Leslie Pack Kaelbling, and Michael L Littman. Acting optimally in partially observable stochastic domains. In *Aaai*, volume 94, pages 1023{ 1028, 1994.

[12] Lonnie Chrisman. Reinforcement learning with perceptual aliasing: The perceptual distinctions approach. In *AAAI*, volume 1992, pages 183{188. Citeseer, 1992.

[13] Will Dabney, Mark Rowland, Marc G Bellemare, and Remi Munos. Distributional reinforcement learning with quantile regression. In *Thirty-Second AAAI Conference on Arti cial Intelligence*, 2018.

[14] Sanjoy Dasgupta and Anupam Gupta. An elementary proof of the johnson-lindenstrauss lemma. *International Computer Science Institute, Technical Report*, 22(1):1{5, 1999.

[15] Sanjoy Dasgupta and Anupam Gupta. An elementary proof of a theorem of johnson and lindenstrauss. *Random Structures & Algorithms*, 22(1):60{65, 2003.

[16] EB Dynkin. Controlled random sequences. *Theory of Probability & Its Applications*, 10(1):1{14, 1965.

[17] William Hamilton, Mahdi Milani Fard, and Joelle Pineau. E cient learning and planning with compressed predictive states. *The Journal of Machine Learning Research*, 15(1):3395{3439, 2014.

[18] E Hansen. Markov decision processes with observation costs. Technical report, Technical Report UM-CS-1997-001, 1997.

[19] Hado V Hasselt. Double q-learning. In *Advances in Neural Information Processing Systems*, pages 2613{2621, 2010.

[20] Masoumeh T Izadi and Doina Precup. Using rewards for belief state updates in partially observable markov decision processes. In *European Conference on Machine Learning*, pages 593{600. Springer, 2005.

[21] Michael R James and Satinder Singh. Learning and discovery of predictive state representations in dynamical systems with reset. In *Proceedings of the twenty- rst international conference on Machine learning*, page 53. ACM, 2004.

[22] Michael R James, Satinder Singh, and Michael L Littman. Planning with predictive state representations. In *2004 International Conference on Machine Learning and Applications, 2004. Proceedings.*, pages 304{311. IEEE, 2004.

[23] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. Reinforcement learning: A survey. *Journal of arti cial intelligence research*, 4:237{285, 1996.

[24] Sham M Kakade. A natural policy gradient. In *Advances in neural information processing systems*, pages 1531{1538, 2002.

[25] Michael L Littman. A tutorial on partially observable markov decision processes. *Journal of Mathematical Psychology*, 53(3):119{125, 2009.

[26] Michael L Littman and Richard S Sutton. Predictive representations of state. In *Advances in neural information processing systems*, pages 1555{1561, 2002.

[27] R McCallum. Reinforcement learning with selective perception and hidden state. 1997.

[28] Daniel Nikovski. *State-aggregation algorithms for learning probabilistic models for robot control*. PhD thesis, Carnegie Mellon University, The Robotics Institute, 2002.

[29] Loren K Platzman. State-estimation of partially-observed markov chains: Decomposition, convergence, and component identi cation. Technical report, MASSACHUSETTS INST OF TECH CAMBRIDGE ELECTRONIC SYSTEMS LAB, 1977.

[30] Martin L Puterman. Markov decision processes. *Handbooks in operations research and management science*, 2:331{434, 1990.

[31] Detlef Rhenius et al. Incomplete information in markovian decision models. *The Annals of Statistics*, 2(6):1327{1334, 1974.

[32] Ulrich Rieder. Bayesian dynamic programming. *Advances in Applied Probability*, 7(2):330{348, 1975.

[33] Matthew Rosencrantz, Geo Gordon, and Sebastian Thrun. Learning low dimensional predictive representations. In *Proceedings of the twenty- rst international conference on Machine learning*, page 88. ACM, 2004.

[34] Mark Rowland, Marc G Bellemare, Will Dabney, Remi Munos, and Yee Whye Teh. An analysis of categorical distributional reinforcement learning. *arXiv preprint arXiv:1802.08163*, 2018.

[35] Matthew R Rudary and Satinder P Singh. A nonlinear predictive state representation. In *Advances in neural information processing systems*, pages 855{862, 2004.

[36] Mats Rudemo. State estimation for partially observed markov chains. *Journal of Mathematical Analysis and Applications*, 44(3):581{611, 1973.

[37] Yoshikazu Sawaragi and Tsuneo Yoshikawa. Discrete-time markovian decision processes with incomplete state observation. *The Annals of Mathematical Statistics*, 41(1):78{86, 1970.

[38] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International conference on machine learning*, pages 1889{1897, 2015.

[39] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[40] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. 2014.

[41] Satinder P Singh, Michael L Littman, Nicholas K Jong, David Pardoe, and Peter Stone. Learning predictive state representations. In *Proceedings of the 20th International Conference on Machine Learning (ICML-03)*, pages 712{719, 2003.

[42] Richard D Smallwood and Edward J Sondik. The optimal control of partially observable markov processes over a nite horizon. *Operations research*, 21(5):1071{1088, 1973.

[43] Edward J Sondik. The optimal control of partially observable markov processes over the in nite horizon: Discounted costs. *Operations research*, 26(2):282{304, 1978.

[44] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[45] Richard S Sutton, David A McAllester, Satinder P Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Advances in neural information processing systems*, pages 1057{1063, 2000.

[46] Csaba Szepesvari and Michael L Littman. A uni ed analysis of value-function-based reinforcement-learning algorithms. *Neural computation*, 11(8):2017{2060, 1999.

[47] SS Vallender. Calculation of the wasserstein distance between probability distributions on the line. *Theory of Probability & Its Applications*, 18(4):784{786, 1974.

[48] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Thirtieth AAAI conference on arti cial intelligence*, 2016.

[49] Joel Veness, Kee Siong Ng, Marcus Hutter, William Uther, and David Silver. A monte-carlo aixi approximation. *Journal of Arti cial Intelligence Research*, 40:95{142, 2011.

[50] Abraham Wald. *Sequential analysis*. Courier Corporation, 1973.

[51] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279{292, 1992.