Learning Multi-Step Predictive State Representations

Anonymous Submission

Abstract

Recent years have seen the development of efficient and provably correct spectral algorithms for learning models of partially observable environments arising in many applications. But despite the high hopes raised by this new class of algorithms, their practical impact is still below expectations. One reason for this is the difficulty in adapting spectral methods to exploit structural constraints about different target environments which can be known beforehand. A natural structure intrinsic to many dynamical systems is a multi-resolution behaviour where interesting phenomena occur at different time scales during the evolution of the system. In this paper we introduce the multi-step predictive state representation (M-PSR) and an associated learning algorithm that finds and leverages frequent patterns of observations at multiple scales in dynamical systems with discrete observations. We perform experiments on robot exploration tasks in a wide variety of environments and conclude that the use of M-PSR improves over the classical PSR for varying amounts of data, environment sizes, and number of observations symbols.

Introduction

Learning models of partially observable dynamical systems is very important in practice — several alternatives...

General algorithms are not designed to exploit frequen patterns/structure in sequences of observations to speed-up learning — but in practice with large observations and highly structured environments this might be necessary to achieve decent results

We propose a new model of predictive state representation for environments with discrete observations: the multi-step PSR (M-PSR)

We show how the standard spectral learning for PSR extends to M-PSR

Then we present a data-driven algorithm for selecting a particular M-PSR from data sampled from a structured partially observable environment

We evaluate the performance of our algorithms in an extensive collection of synthetic environments and conclude that...

The Multi-Step PSR

A linear *predictive state representation* (PSR) for an autonomous dynamical system with discrete observations is a tuple $\mathcal{A} = \langle \Sigma, \boldsymbol{\alpha}_\epsilon, \boldsymbol{\alpha}_\infty, \{\mathbf{A}_\sigma\}_{\sigma\in\Sigma}\rangle$ where: Σ is a finite set of possible observations, $\boldsymbol{\alpha}_\epsilon, \boldsymbol{\alpha}_\infty \in \mathbb{R}^n$ are vectors of initial and final weights, and $\mathbf{A}_\sigma \in \mathbb{R}^{n\times n}$ are the transition operators associated with each possible observation. The dimension n is the number of states of $\mathcal{A}$. Formally, a PSR is a *weighted finite automata* (WFA) (**?**) computing a function given by the probability distribution of sequences of observations in a partially observable dynamical system with finite state. The function $f_\mathcal{A} : \Sigma^\star \to \mathbb{R}$ computed by $\mathcal{A}$ is given by

$$f_\mathcal{A}(x) = f_\mathcal{A}(x_1 \cdots x_t) = \boldsymbol{\alpha}_\epsilon^\top \mathbf{A}_{x_1} \cdots \mathbf{A}_{x_t} \boldsymbol{\alpha}_\infty = \boldsymbol{\alpha}_\epsilon^\top \mathbf{A}_x \boldsymbol{\alpha}_\infty \ .$$

The value of $f_\mathcal{A}(x)$ is interpreted as the probability that the system produces the sequence of observations $x = x_1 \cdots x_t$ starting from the initial state specified by $\boldsymbol{\alpha}_\epsilon$. Depending on the model, this can be a probability that the system generates x and *stops*, or the probability that the system generates x and *continues*. Given a partial history u of the evolution of the system – i.e. a prefix in string terminology – the state of a PSR can be updated from $\boldsymbol{\alpha}_\epsilon$ to $\boldsymbol{\alpha}_u = \boldsymbol{\alpha}_\epsilon \mathbf{A}_u$. This allows for conditional queries about the possible observations that will follow u. For example, the probability of observing the string v given that we have already observed u will be written as

$$f_{\mathcal{A},u}(v) = \frac{\boldsymbol{\alpha}_u \mathbf{A}_v \boldsymbol{\alpha}_\infty}{\nu_\mathcal{A}(u)} \ ,$$

where $\nu_\mathcal{A}(u)$ is a normalising constant to obtain a proper conditional distribution whose expression depends on the actual semantics of the PSR (stop v.s. continuation probabilities).

To define our model for multi-step PSR we basically augment a PSR with two extra objects: a set of *multi-step observations* $\Sigma' \subset \Sigma^+$ containing non-empty strings formed by basic observations, and a *coding function* $\kappa : \Sigma^\star \to \Sigma'^\star$ that given a string of basic observations produces an equivalent string composed using multi-step observations. The choice of Σ' and κ can be quite application-dependent, in order to reflect the particular patterns arising from different environments. However, we assume this objects satisfy a basic set of requirements for the sake of simplicity and to avoid degenerate situations:

1. The set Σ' must contain all symbols in Σ; i.e. $\Sigma \subseteq \Sigma'$
2. The function κ satisfies $\partial(\kappa(x)) = x$ for all $x \in \Sigma^\star$, where $\partial : \Sigma'^\star \to \Sigma^\star$ is the *decoding morphism* between free monoids given by $\partial(z) = z \in \Sigma^\star$ for all $z \in \Sigma'$. Note this implies that $\kappa(\epsilon) = \epsilon$, $\kappa(\sigma) = \sigma$ for all $\sigma \in \Sigma$, and κ is injective.

Using these definitions, a *multi-step PSR* (M-PSR) is a tuple $\mathcal{A}' = \langle \Sigma, \Sigma', \kappa, \boldsymbol{\alpha}_\epsilon, \boldsymbol{\alpha}_\infty, \{\mathbf{A}_\sigma\}_{\sigma\in\Sigma'}\rangle$ containing a PSR with observations in Σ', together with the basic observations Σ and the corresponding coding function κ. In addition to the function $f_{\mathcal{A}'} : \Sigma'^\star \to \mathbb{R}$ that $\mathcal{A}'$ by being a PSR over Σ', it also computes another function $f'_{\mathcal{A}'} : \Sigma^\star \to \mathbb{R}$ given by $f'_{\mathcal{A}'}(x) = f_{\mathcal{A}'}(\kappa(x))$. In many cases we will abuse our notation and write $f_{\mathcal{A}'}$ for $f'_{\mathcal{A}'}$ when there is no risk of confusion.

Examples of M-PSRs

We now describe several examples of M-PSR, and put special emphasis on models that will be used in our experiments.

Base M-PSR A PSR with a single observation $\Sigma = \{\sigma\}$ can be used to measure the time – i.e. number of discrete time-steps – until a certain event happens (**?**). In this case, a natural approach to build an M-PSR for timing models is to build a set of multi-step observations containing sequences of a's whose lengths are powers of a fixed base. That is, given an integer $b > 0$, we build the set of multi-step observations as $\Sigma' = \{\sigma, \sigma^b, \sigma^{b^2}, \ldots, \sigma^{b^K}\}$ for some positive K. A natural choice of coding map in this case is the one that represents any length $t \geq 0$ as a number in base b, with the difference that the largest power b that is allowed is b^K. This corresponds to writing (in a unique way) $t = t_0 b^0 + t_1 b^1 + t_2 b^2 + \cdots + t_K b^K$, where $0 \leq t_k \leq b-1$ for $0 \leq k \leq K$, and $t_K \geq 0$. With this decomposition we obtain the coding map $\kappa(\sigma^t) = (\sigma^{b^K})^{t_K}(\sigma^{b^{K-1}})^{t_{K-1}} \cdots (\sigma^b)^{t_1}(\sigma)^{t_0}$. Note that we choose to write powers of longer multi-step observations first, followed by powers of shorter multi-step observations. For further reference, we will call this model the Base M-PSR.

Lucas: changed K-1 to K

For the multiple observation case one can also use a Base M-PSR. For example, if there are two observations $\Sigma = \{\sigma_1, \sigma_2\}$, we can take $\Sigma' = \{\sigma_1, \sigma_2, \sigma_1^b, \sigma_2^b, \sigma_1^{b^2}, \sigma_2^{b^2}, ..., \sigma_1^{b^K}, \sigma_2^{b^K}\}$. This can of course be extended for any finite number of observations. For the encoding map κ we first split the string into sequences of consecutive repeated symbols and then use the same encoding as for timing. For example: $\kappa(\sigma_1^5\sigma_2^3) = (\sigma_1^2)^2(\sigma_1)(\sigma_2^2)(\sigma_2)$ when using $b = 2$ and $K = 1$.

Tree M-PSR Another example for the multiple observation case is what we call the Tree M-PSR. In a Tree M-PSR, we set $\Sigma' = \{x \in \Sigma^\star, |x| \leq L\}$. Here L is a parameter of choice, but note that $|\Sigma'| = O(|\Sigma|^L)$, thus in practice L must remain small if we want the M-PSR to be representable using a small number of parameters. For the decoding map κ, we first split a string x as $x = u_1 u_2 \cdots u_m u_f$, where $|u_i| = L$ for $1 \leq i \leq n$ and $|u_f| = |x| - (m \cdot L) < L$. With this we set $\kappa(x) = (u_1)(u_2)\cdots(u_m)u_f$, where we write u_f to denote that this part of the string is represented directly using symbols from Σ.

Borja: LUCAS: I see an inconsistency here. If you add to Σ' strings of length $1, 2, \ldots, L$, then once you're done using the symbols of length L in κ you should start using symbols of length $L-1$, and so on. But the text seems to suggest that you jump from symbols of length L to symbols of length 1, in which case it'd be better to choose $\Sigma' = \Sigma \cup \Sigma^L$ directly. Note I changed some of your notation when making edits.

Lucas: I don't see where the inconsistency is. Maybe i'm missing something but u_f is just the string remainder $|u_f|$ ¡ L, so we don't jump from L to 1. Ex: L = 3 "abcab" –¿ ("abc")("ab")

Data-Driven M-PSR Although the constructions above have some parameters that can be tuned depending on the particular application, they are not directly dependent on the observations produced by the target environment. In our experiments section, we find that performance of M-PSRs depend heavily on how Σ' reflects the observations in from the target environment. Thus, we develop an algorithm for choosing Σ' in a data-driven way. In addition, we provide a generic coding function κ that can be applied to any M-PSR. This encoding delivers good experimental predictive performance and computationally inexpensive probability queries. Together, these yield another type of M-PSR, which we call the Data-Driven M-PSR, which is the output of the learning algorithm described in the next section.

Lucas: Split one long sentence into two shorter ones

Learning Algorithm for M-PSR

In this section, we describe a learning algorithm for M-PSR which combines the standard spectral method for PSR (Boots, Siddiqi, and Gordon 2011) with a greedy algorithm for extracting an extended set of symbols Σ' containing frequent patterns of observations and which minimises a coding cost for a generic choice of coding function κ.

Spectral Learning Algorithm

We start by extending the spectral learning algorithm to M-PSR under the assumption that κ and Σ' are known. In this case, the learning procedure only need to recover the operators $\mathbf{A}_\sigma$ for all $\sigma \in \Sigma'$, and the initial and final weights $\boldsymbol{\alpha}_\epsilon, \boldsymbol{\alpha}_\infty$. We start by recalling some basic notation about Hankel matrices, and then proceed to describe the learning algorithm. For the purpose of describing the learning algorithm we shall start by assuming that the function $f : \Sigma^\star \to \mathbb{R}$ associated with the target M-PSR can be evaluated at every string. Note that this does not necessarily imply that we know the M-PSR, since the values of f correspond to probabilities of observations that can be effectively estimated from data.

Given $f : \Sigma^\star \to \mathbb{R}$ we will use its *Hankel matrix* representation $\mathbf{H}_f \in \mathbb{R}^{\Sigma^\star \times \Sigma^\star}$, which is an infinite matrix whose rows and columns are indexed by strings in $\Sigma^\star$ and whose entries are given by $\mathbf{H}_f(u, v) = f(uv)$. To efficiently work

with this matrix we shall only consider finite sub-blocks indexed by a finite set of prefixes $\mathcal{P} \subset \Sigma^\star$ and suffixes $\mathcal{S} \subset \Sigma^\star$. Both $\mathcal{P}$ and $\mathcal{S}$ are input parameters given to the algorithm, see (**?**) for a discussion of how to choose them in practice. The pair $\mathcal{B} = (\mathcal{P}, \mathcal{S})$ is sometimes called a basis, and it determines a sub-block $\mathbf{H}_\mathcal{B} \in \mathbb{R}^{\mathcal{P} \times \mathcal{S}}$ of $\mathbf{H}_f$ with entries given by $\mathbf{H}_\mathcal{B}(u, v) = \mathbf{H}_f(u, v)$ for all $u \in \mathcal{P}$ and $v \in \mathcal{S}$. For a fixed basis, we also consider the vectors $\mathbf{h}_\mathcal{S} \in \mathbb{R}^\mathcal{S}$ with entries given by $\mathbf{h}_\mathcal{S}(v) = \mathbf{H}_f(\epsilon, v)$ for every $v \in \mathcal{S}$, and $\mathbf{h}_\mathcal{P} \in \mathbb{R}^\mathcal{P}$ with $\mathbf{h}_\mathcal{P}(u) = \mathbf{H}_f(u, \epsilon)$.

Note that so far the definitions above have only used Σ. In order to recover operators $\mathbf{A}_\sigma$ for all $\sigma \in \Sigma'$ we will need to consider multi-step shifts of the finite Hankel matrix $\mathbf{H}_\mathcal{B}$. In particular, given $\sigma \in \Sigma'$ we define the sub-block $\mathbf{H}_\sigma \in \mathbb{R}^{\mathcal{P} \times \mathcal{S}}$ whose entries are given by $\mathbf{H}_\sigma(u, v) = f(u \, \sigma \, v)$. Note that this can be interpreted as either using the lift $f(\kappa(u) \, \sigma \, \kappa(v))$ or the decoding $f(u \, \partial(\sigma) \, v)$, but the actual value in the matrix $\mathbf{H}_\sigma$ will be the same.

Now we can give the details about the learning algorithm. Suppose the basis $\mathcal{B}$ and the desired number of states n are given. Start by collecting a set of sampled trajectories and use them to estimate the matrices $\mathbf{H}_\mathcal{B}, \mathbf{H}_\sigma \in \mathbb{R}^{\mathcal{P} \times \mathcal{S}}$ and vectors $\mathbf{h}_\mathcal{P} \in \mathbb{R}^\mathcal{P}$, $\mathbf{h}_\mathcal{S} \in \mathbb{R}^\mathcal{S}$. Then take the truncated SVD $\mathbf{U}_n \mathbf{D}_n \mathbf{V}_n^\top$ of $\mathbf{H}_\mathcal{B}$, where $\mathbf{D}_n \in \mathbb{R}^{n \times n}$ contains the first n singular values of $\mathbf{H}_\mathcal{B}$, and $\mathbf{U}_n \in \mathbb{R}^{\mathcal{P} \times n}$ and $\mathbf{V}_n \in \mathbb{R}^{\mathcal{S} \times n}$ contain the first left and right singular vectors respectively. Finally, compute the transition operators of the M-PSR as $\mathbf{A}_\sigma = \mathbf{D}_n^{-1} \mathbf{U}_n^\top \mathbf{H}_\sigma \mathbf{V}_n$ for all $\sigma \in \Sigma'$, and the initial and final weights as $\boldsymbol{\alpha}_\epsilon^\top = \mathbf{h}_\mathcal{S}^\top \mathbf{V}_n$ and $\boldsymbol{\alpha}_\infty = \mathbf{D}_n^{-1} \mathbf{U}_n^\top \mathbf{h}_\mathcal{P}$. This algorithm yields an M-PSR with n states. It was proved in (Boots, Siddiqi, and Gordon 2011) that this algorithm is statistically consistent for PSR (under a mild condition on $\mathcal{B}$) and the same guarantees extend to M-PSR.

A Generic Coding Function

Given Σ and Σ', a generic coding function $\kappa : \Sigma^\star \to \Sigma'^\star$ can be obtained by minimising the coding length $|\kappa(x)|$ of every string $x \in \Sigma^\star$. More formally, we consider the coding κ given by

$$\kappa(x) = \operatorname*{argmin}_{y \in \Sigma'^\star, \, \partial(y) = x} |y| \ .$$

Note that for the single observation case this is equivalent to the coin change problem. This has advantage of minimising the number of operators $\mathbf{A}_\sigma$ that will need to be multiplied to compute the value of the M-PSR on a string x. At the same time, operators expressing long transition sequences capture intermediate contributions of weak states throughout the transition sequence. This is true even if one chooses to learn a smaller model which excludes these weak states. Thus, with this choice of κ, or for any κ which compactly expresses transition sequences, one should be able to get much better performance with reduced models.

Lucas: Changed explanation of performance increase from fewer matrices. Let me know what you think. I don't think noise is the right term here because even for very large amounts of data (basically the true model), M-PSRs do far better than PSRs. I think it has more to do with the projection intuition.

The optimization problem above can be solved by dynamic programming. To do so, one inductively computes the optimal string encoding for the prefix $x_1 \cdots x_i$ for all $1 \leq i \leq |x|$. This can be obtained by minimising over all $\sigma \in \Sigma'$ which terminate at the index i of x. We provide the full pseudo-code for this encoding function in the appendix.

Greedy Selection of Multi-Step Observations

Here we present a greedy algorithm that selects which multi-step transition sequences need to be added to Σ' given a collection of observed trajectories. Having a Σ' which reflects the type of observations produced the target system will promote short encodings when coupled with the coding function described above. In practice, this greedy algorithm will pick substrings appearing in the training data which are long, frequent, and diverse. From an intuitive standpoint, one can view structure in observation sequences as relating to the level of entropy in the distribution over multi-step observations produced by the system.

Here we provide a general description of how the algorithm works, and a detailed pseudo-code implementation is presented in the supplementary material. The algorithm starts by finding all possible substrings in $\Sigma^\star$ that appear in the training dataset. As a preprocessing step, and for computational reasons, this is constrained to contain only the the M most frequent substrings, where M is a parameter chosen by the user. Here the frequency is measured by number of observed trajectories that contain a given substring.

The construction of Σ' is initialised by Σ and a new multi-step symbol is added at each phase. A phase starts by evaluating each substrings in terms of how much they would reduce the number of transition operators used to encode the whole training set using κ if added to Σ'. The algorithm then adds to Σ' a multi-step symbol corresponding to the best substring, i.e. the one that would reduce the most the whole coding cost. More formally, at the ith iteration of the algorithm the following is computed:

$$\operatorname*{argmin}_{u \in \mathrm{sub}_M} \sum_{x \in \mathrm{train}} |\kappa_{\Sigma'_i \cup \{u\}}(x)| \ ,$$

where Σ'_i is the set of multi-step observations at the beginning of phase i and we use $\kappa_{\Sigma'}$ to denote the encoding function with respect to a given set of multi-step observations for clarity. The algorithm terminates after Σ' reaches a predetermined size.

Borja: TODO: Introduce the train and sub_M notation used above before, in the text

Experiments

In this section, we assess the performance of PSRs and different kinds of Multi-PSRs. We do so over many different configurations of parameters, the most relevant ones being the model size, the number of observations used, and the type of environment. For all the plots, the x-axis is model size of the PSR/M-PSRs and the y-axis is an error measurement of the learned PSR/M-PSRs.

Obtaining Observation Sequences

In all the experiments, an agent is positioned in a starting location and stochastically navigates the environment based on a transition function $\delta : S \times S \rightarrow [0, 1]$. Whenever a transition occurs an observation symbol is produced. When the agent exits the labyrinth, we say the trajectory is finished, and we record the concatenation of the symbols produced. We call this concatenation the observation sequence for that trajectory.

Learning Implementation: Timing v.s Multiple Symbols

For the timing case, we construct our empirical hankel matrix by taking P,S = $\{a^i, \forall i <= n\}$. The parameter n depends on the application. For Double Loop environments we set $n = 150$, while for the pacman labyrinth $n = 600$. For these choices of n, we verify that as the amount of data gets large the learned PSR with the true model size becomes increasingly close to the true model. For Base M-PSR, we set $b = 2, K = 9$, so that the longest string in Σ' is a^{256}.

For multiple observations a slightly more complex approach is required to choose P and S. For prefixes P, we select the k most frequent prefixes from our observation set. For suffixes S, we take all suffixes that occur from P. We also require prefix completeness. That is, if p' is a prefix of p $\in P$, then p' $\in P$. This heuristic for constructing empirical hankel matrices was given in previous work by [] and it showed that (). For the Base M-PSR, we take K=8 and B=2 for both symbols $\Sigma = \{g, b\}$, g for green and b for blue. For the Tree M-PSR we set $L = 7$ for a total of 128 operators, a far larger allowance than the other M-PSRs.

Lucas: Borja's heuristics for choosing P,S need citation and explanation

Measuring Performance

To measure the performance of a PSR/M-PSR we use the following norm:

$$||f - \hat{f}|| = \sqrt{\sum_{x \in observations} (f(x) - \hat{f(x)})^2}$$

We use this norm because of a bound presented by [AUTHORS], which states that (). Here the function f denotes the true probability distribution over observations and the function $\hat{f}$ denotes the function associated with the learned M-PSR/PSR. In our environments, the function f is obtainable directly as we have access to the underlying HMMs.

Since the set of observations $\Sigma^\star$ is infinite, we compute approximations to this error norm, by fixing a set of strings T and summing over T. For the timing case, we take $T = \{a^i, \forall i <= C\}$ with C=600. Importantly C large enough such that $\sum_{i=0}^{C} f(a^i) > 0.99$. For the multiple observation case, we take all possible strings producible from the prefixes and suffixes in our dataset. That is, for the multiple observation case $T = \{ps, \forall p \in P, \forall s \in S\}$

Lucas: Need citation for bound on norm2 error

Double Loop Timing

For timing, we start by considering a double loop environment. The lengths of the loops correspond to the number of states in the loop. A trajectory begins with the agent starting at the intersection of the two loops. Here, the agent has a 50% probability of entering either loop. At intermediate states in the loops the agent moves to the next state in the loop with probability 1-P or remains in its current state with probability P. Here, P represents the self-transition probability for internal states. Exit states are located halfway between each loop. Agents leaving the environment at exit states with 50% probability. This means that if loop lengths are $l_1 = 64$ and $l_2 = 16$, we observations will come in the form of $n_1 * l_1 + n_2 * l_2 + (1 - \alpha) * l1/2 + \alpha * l2/2$, with $\alpha = 1$ representing an exit through loop1 and $\alpha = 0$ an exit through loop2.

Number of Trajectories

Here, we vary the number of observations used in our dataset. PSRs/M-PSRs learned in Figure 1 use 100 observation sequences, while those in Figure 2 use 10000. In both cases the M-PSRs outperform the standard PSR for reduced model sizes.

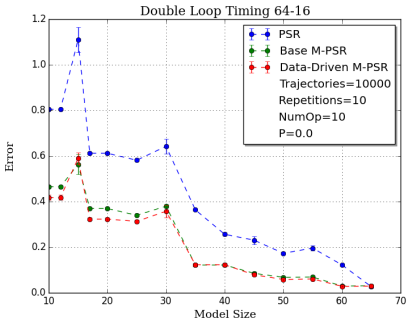

Figure 1: Low Data Double Loop 64-16

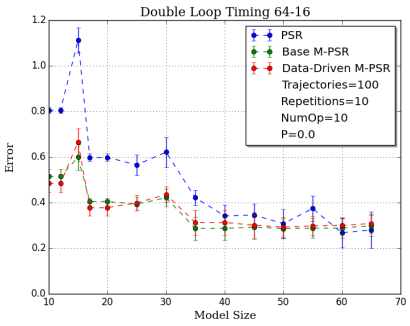

Figure 2: High Data Double Loop 64-16

Noise: P

Next, we vary the self-transition probability P to simulate noise in an environment. Figure 3 is a 64-16 double loop

with P=0.2, and Figure 1 is a 64-16 double loop with P=0. We find that the noisy loops are more compressible, that is one can achieve better performance for low model sizes, but the performance becomes worse as the model size attains the environments true size. Nevertheless, M-PSRs still significantly outperform the standard PSR for reduced model sizes.

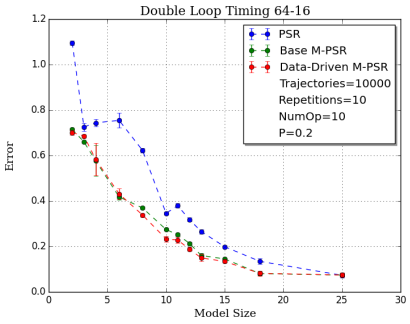

Figure 3: Noisy Double Loop 64-16

Loop Lengths

In Figure 4, we plot the results of a 47-27 labyrinth, where observations will not be as compactly expressed from the Base M-PSR. Again, M-PSRs outperform the standard PSR for reduced model sizes. In addition we see that the Data-Driven M-PSR does better than the Base M-PSR.

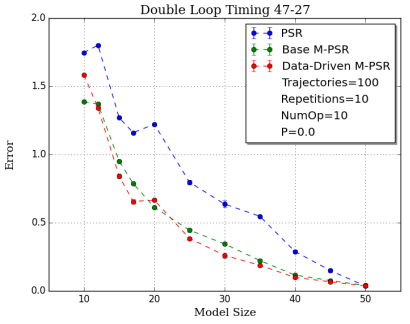

Figure 4: High Data Double Loop 47-27

Choice of K for Base M-PSRs

In Figure 5, we plot performance of Base M-PSRs with different values of L. We note that larger values of K have better performance up to K=7.

Varying NumOps

In Figure 6, we look at how the varying the number of multi-step transition operators affects performance for the 64-16 double loop. In this environment, a higher number of operators improves performance up until about 20 operators. Here the most important operators seem to be: $\{a, a^8, a^{24}, a^{32}, a^{72}\}$ which are closely tied to the environment's structure.

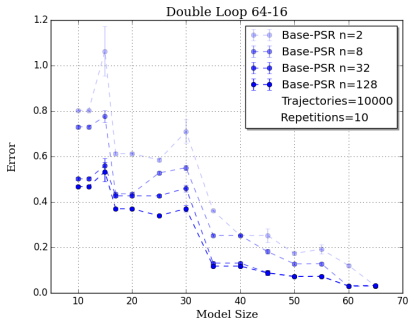

Figure 5: Double Loop 64-16

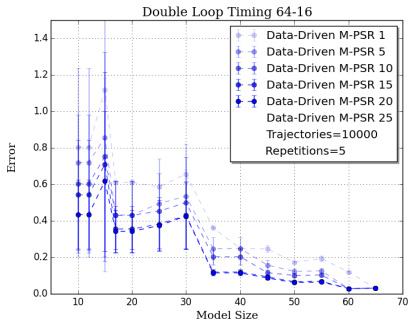

Figure 6: Varying NumOps

Large Labyrinth Timing

We proceed to work with a labyrinth environment similar to a Pacman game. A graphical representation of this labyrinth is available in the appendix. Transitions to new states occur with equal probability. The weight $w(u, v)$ between states u and v corresponds to the number of time steps from u to v. We add an additional parameter sF: stretch factor, which scales all of the weights in the graph.

Number of Trajectories

In Figure 7,8 we vary the number of observations used for learning. M-PSRs outperform the traditional PSR regardless of the amount of data. Secondly, as expected, the performance of all models is worse for less data.

Stretch Factor: sF

In Figures 7,9 and 10 we vary the stretch factor parameter with a fixed dataset. We find that a higher values of sF allow for increased improvement of the M-PSR relative to the performance of the standard PSR.

Multiple Observations: Coloured Loops

We now move to the multiple observation case. We construct a Double Loop environment where loop1 is green with $l_1 = 27$ and loop2 is blue with $l_2 = 17$. We fix the length of observations to be:

$$TrajectoryLength := (l_1 + l_2) * 3$$

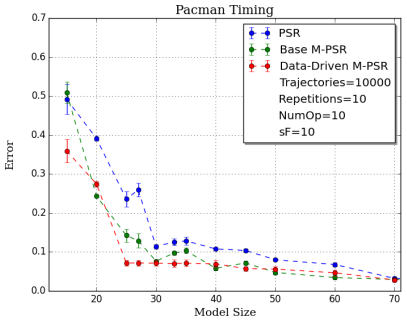

Figure 7: High Data Pacman Labyrinth

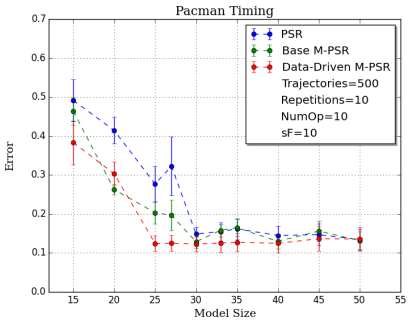

Figure 8: Low Data Pacman Labyrinth

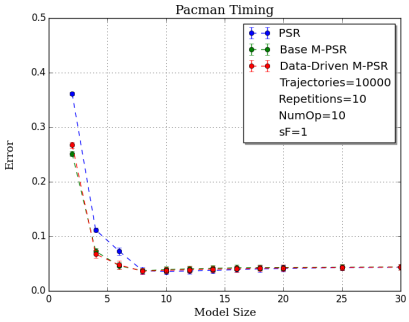

Figure 9: Stretch Factor: 1

We build empirical estimates for the hankel matrix as follows:

$$f(x) = \frac{count([s \in Obs, s = x])}{count([s \in Obs, |s| \geq x])}$$

This means that the PSRs will compute the probability of x occurring as a prefix.

Number of Trajectories

As for the timing case, we vary the amount of data to learn PSRS/M-PSRs in Figures 11 and 12. Once again we find M-PSRs perform far better, especially the Data-Driven M-PSR. This makes sense as when complexity in observations

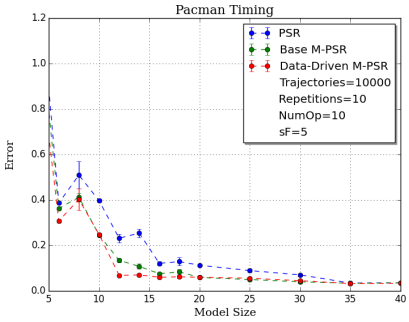

Figure 10: Stretch Factor: 5

increases only custom M-PSRs will express transitions compactly.

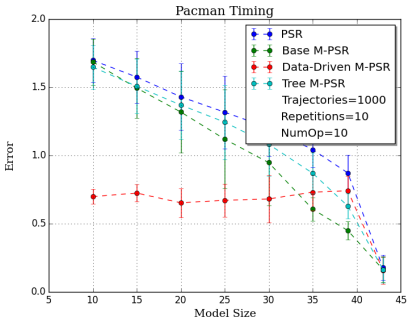

Figure 11: High Data Colored Loops 27-17

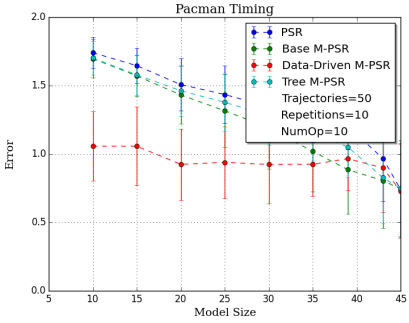

Figure 12: Low Data Colored Loops 27-17

Varying NumOps

In Figure 13, we vary the number of multi-step transition operators learned. Here the important operators seem to be $\{g, b, g^{27}, b^{17}\}$ which reflects the structure of the environment.

Discussion

In all the experiments conducted one aspect dominates: M-PSRs offer significantly better predictive performance for

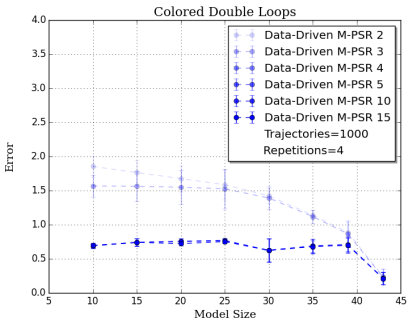

Figure 13: Varying NumOps

reduced model sizes than PSRs. In addition, Data-Driven PSRs offer improvement over generic M-PSRs by learning transition operators specific to the environment. Although not covered in experiments M-PSRs also offer a computational advantage for queries as they use far fewer matrices. This can be of large significance in applications where queries are done online such as for planning.

Lucas: The above is roughly what we had discussed for the discussion, should reword and add more if we have room

Acknowledgments

Funding and friends...

Appendix

Q: A query string (a string for which one wishes to determine the probability of).

bestE: A map from indices i of Q to the optimal encoding of Q[:i].

minE: A map from indices i of Q to $|bestE[i]|$

opEnd: A map from indices i of Q to the set of strings in Σ': $\{x \in \Sigma' s.tQ[i-|x|:i] == x\}$

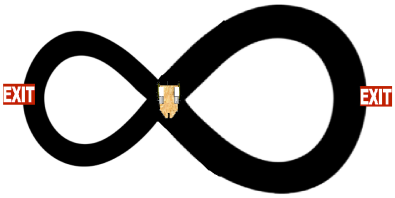

Figure 14: Double Loop Environment

References

Boots, B.; Siddiqi, S.; and Gordon, G. 2011. Closing the learning planning loop with predictive state representations. *International Journal of Robotic Research*.

Algorithm 1 Base Selection Algorithm

procedure BASE SELECTION
  $\Sigma' \leftarrow \{s, s \in \sum\}$
  $Subs \leftarrow$ {k frequent $s \in subObs$}
  $prevBestE \leftarrow null$
  for each obs in Obs **do**
    $prevBestEncoding[obs] \leftarrow |obs|$
  end for
  $i \leftarrow 0$
  while $i < numOperators$ **do**
    $bestOp \leftarrow null$
    $bestImp \leftarrow null$
    for each s $\in Subs$ **do**
      $c \leftarrow 0$
      for each obs in Obs **do**
        $c \leftarrow c + DPEncode(obs) - prevBestE(obs)$
      end for
      if $c > bestImp$ **then**
        $bestOp \leftarrow observation$
        $bestImp \leftarrow c$
      end if
    end for
    $\Sigma' \leftarrow \Sigma' \cup bestOp$
    for each obs in Obs **do**
      $prevBestE \leftarrow DPEncode(obs, \Sigma')$
    end for
    $i \leftarrow i + 1$
  end while
  return Σ'
end procedure

Algorithm 2 Encoding Algorithm

procedure DPENCODE
  $bestE[] \leftarrow String[|Q|+1]$
  $minE[] \leftarrow Int[|Q|+1]$
  $opEnd[] \leftarrow String[|Q|+1][]$
  $bestEnd[0] = Q[0]$
  $minE[0] = 0$
  for i in $[1, |Q|]$ **do**
    $opEnd[i] \leftarrow \{s \in \Sigma', Q[i-|s|:i] == s\}$
  end for
  for i in $[1, |Q|]$ **do**
    $bestOp \leftarrow null$
    $m \leftarrow null$
    for $s \in opEnd[i]$ **do**
      $tempInt \leftarrow minE[i-|s|]+1$
      if $m == null$ or $tempInt < m$ **then**
        $m \leftarrow temp$
        $bestOp \leftarrow s$
      end if
    end for
    $minE[i+1] \leftarrow m$
    $bestE[i+1] \leftarrow bestE[i-|bestOp|] + bestOp$
  end for
  return $bestE[|Q|]$
end procedure

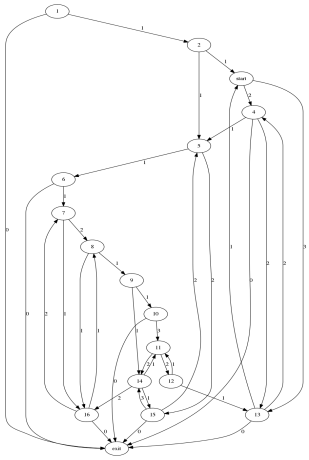

Figure 15: Graph of Pacman Labyrinth