Online Relative Entropy Policy Search using Reproducing Kernel Hilbert Space Embeddings

Zhitang Chen
Noah's Ark Lab, Huawei

Pascal Poupart
University of Waterloo, Canada

Yanhui Geng
Noah's Ark Lab, Huawei

Abstract

Kernel methods have been successfully applied to reinforcement learning problems to address some challenges such as high dimensional and continuous states, value function approximation and state transition probability modeling. In this paper, we develop an online policy search algorithm based on a recent state-of-the-art algorithm REPS-RKHS that uses conditional kernel embeddings. Our online algorithm inherits the advantages of REPS-RKHS, including the ability to learn non-parametric control policies for infinite horizon continuous MDPs with high-dimensional sensory representations. Different from the original REPS-RKHS algorithm which is based on batch learning, the proposed online algorithm updates the model in an online fashion and thus is able to capture and respond to rapid changes in the system dynamics. In addition, the online update operation takes constant time (i.e., independent of the sample size n), which is much more efficient computationally and allows the policy to be continuously revised. Experiments on different domains are conducted and results show that our online algorithm outperforms the original algorithm.

1 Introduction

Reinforcement learning algorithms have been successfully applied to solve many real problems where the state-action space $\mathcal{S} \times \mathcal{A}$ is finite and discrete, or continuous and low-dimensional . However, high dimensional continuous state-action spaces pose a challenge for the estimation of the transition function in model-based approaches and the value function or Q-function in model free techniques.

Appearing in Proceedings of the 19^{th} International Conference on Artificial Intelligence and Statistics (AISTATS) 2016, Cadiz, Spain. JMLR: W&CP volume 51.

Reproducing Kernel Hilbert Space (RKHS) embeddings, as a recent advance (Gretton et al., 2012; Song et al., 2009, 2013) in kernel methods, ease the estimation of value functions and transition functions possible in high dimensional spaces. Some recent works (Grünewälder et al., 2012; Nishiyama et al., 2012) applied the RKHS embedding method to value functions and update the policy greedily with respect to the learnt value function. The advantage of using kernel methods to approximate the value function is that we do not need to define the feature mapping. The value can be easily calculated by kernel tricks (Smola and Schölkopf, 1998). However, as reported in (Van Hoof et al., 2015), these methods update the policy greedily by choosing actions that maximize the value function, which turn out to be unstable during the learning process.

To improve stability, Peters et al. (2010) proposed Relative Entropy Policy Search, which bounds the search for a new policy at each iteration to a neighbourhood of the previous policy. By kernelizing the REPS algorithm, Van Hoof et al. (2015) proposed REPS-RKHS, which inherits the advantage of REPS in terms of stability while employing RKHS embeddings to better approximate the value function with respect to the transition function. The combination of REPS and RKHS yields a *more stable, effective learning progress* for non-parametric reinforcement learning (Van Hoof et al., 2015). In addition, REPS-RKHS avoids using the hand-crafted features and thus is more general and flexible.

However, shortcomings of REPS-RKHS are also very obvious. One of the biggest challenges is the computational complexity of REPS-RKHS. The computational complexity of training a model is $\mathcal{O}(n^3)$, where n is the number of samples in the batch. In order to learn a good policy, in most cases, we need a large number of samples, which leads to long training times for REPS-RKHS. The algorithm periodically updates the policy when a certain amount of data becomes available from the control trajectory, which means that before the policy converges to the optimal one, the algorithm must train multiple times. Another challenge is about the model parameters and the kernel hyperparameter optimization, which could be very important, but very difficult especially for the kernel hyperpa-

rameters. A bad hyperparameter optimization can lead to little policy improvement and even worse policies.

In this paper, we develop an online version of REPS-RKHS, which not only inherits the advantages of REPS-RKHS (e.g., nonparametric policy), but also updates the policy in an online fashion based on the most recent sample $(\mathbf{s}, \mathbf{a}, \mathbf{s}', R)$. The basic idea of our approach is to use the Reduced Rank Approximation to approximate the model based on REPS-RKHS. Based on the reduced rank model, we can easily develop an online learning algorithm that uses the most recent sample to update the model. Compared to the original REPS-RKHS algorithm, our proposed algorithm can detect and respond to the rapidly changing dynamics of the system or the environment since the most recent samples can be used to update the model; while the original REPS-RKHS algorithm has to wait until sufficiently many samples are available to learn the new model in a batch way. Consequently, our proposed online algorithm is far more computationally efficient with $\mathcal{O}(m^2)$ computational complexity compared to REPS-RKHS, which has a computational complexity of $\mathcal{O}(n^3)$, where $m \ll n$. In order to verify the effectiveness of our proposed online algorithm, we conduct a series of experiments on different domains. Experimental results show that our proposed online algorithm converges to the optimal policy faster than the original REPS-RKHS algorithm. Furthermore, our algorithm is much more efficient than REPS-RKHS in terms of computation time.

The rest of the paper is structured as follows. We review some background about Relative Entropy Policy Search (REPS) and Reproducing Kernel Hilbert Space (RKHS) in Section 2. In Section 3, we introduce the Reduced Rank Approximation method to approximate the original REPS-RKHS algorithm. Then we explain how to incorporate the information from the most recent sample to update the model and the policy in an online fashion. In Section 4, we report some experiments with different domains including a toy Markov Decision Process (MDP), the mountain car problem as well as the under-powered pendulum control problem. In Section 5, we draw conclusions based on our algorithmic contributions and experimental results.

2 Preliminaries: REPS and RKHS embeddings

In this section, we review the REPS-RKHS algorithm (Van Hoof et al., 2015). We first review the Relative Entropy Policy Search framework (Peters et al., 2010) where a policy maximizing the expected reward with constraints to avoid deviating too much from the previous one is derived. Next, the REPS-RKHS method is described. It uses kernel embeddings to calculate the expected value function with respect to the transition function. The advantages and the disadvantages of REPS-RKHS are discussed at the end of this section.

2.1 Relative Entropy Policy Search

We use the following notation, which is common in the reinforcement learning literature.

- $\mathbf{s} \in \mathcal{S} = \mathbb{R}^{d_s}$: the state variable of a Markov Decision Process (MDP) in a d_s dimensional state space,
- $\mathbf{a} \in \mathcal{A} = \mathbb{R}^{d_a}$: the action variable of a MDP in a d_a dimensional action space,
- $\pi(\mathbf{a}|\mathbf{s})$: a stochastic policy encoding a distribution over actions $\mathbf{a}$ for each state $\mathbf{s}$,
- $\mu_\pi(\mathbf{s})$: the stationary state distribution (assuming it exists) under policy π,
- $R^{\mathbf{a}}_{\mathbf{s}}$: the reward received after executing action $\mathbf{a}$ when the system is in state $\mathbf{s}$,
- $\mathcal{P}^{\mathbf{a}}_{\mathbf{ss}'}$: the probability of transitioning from state $\mathbf{s}$ to state $\mathbf{s}'$ after executing action $\mathbf{a}$.

The REPS framework gradually improves a policy by repeatedly solving the following constrained optimization problem:

$$\begin{aligned}
&\max_{\pi,\mu_\pi} J(\pi) = \max_{\pi,\mu_\pi} \iint_{\mathcal{S}\times\mathcal{A}} \pi(\mathbf{a}|\mathbf{s})\mu_\pi(\mathbf{s})\mathcal{R}^{\mathbf{a}}_{\mathbf{s}} d\mathbf{a} d\mathbf{s}, \\
&\text{s.t.} \iint_{\mathcal{S}\times\mathcal{A}} \pi(\mathbf{s}|\mathbf{a})\mu_\pi(\mathbf{s}) d\mathbf{s} d\mathbf{a} = 1, \\
&\iint_{\mathcal{S}\times\mathcal{A}} \mathcal{P}^{\mathbf{a}}_{\mathbf{ss}'}\pi(\mathbf{a}|\mathbf{s})\mu_\pi(\mathbf{s}) d\mathbf{a} d\mathbf{s} = \mu_\pi(\mathbf{s}'), \\
&KL(\pi(\mathbf{a}|\mathbf{s})\mu_\pi(\mathbf{s})||q(\mathbf{s},\mathbf{a})) \leq \epsilon,
\end{aligned} \tag{1}$$

The solution to this optimization problem is the best policy (measured by expected value) within a neighbourhood of the current policy in order to avoid unstable learning. The neighbourhood is defined by bounding the Kullback-Leibler divergence between $p(\mathbf{s},\mathbf{a}) = \pi(\mathbf{a}|\mathbf{s})\mu_\pi(\mathbf{s})$ and $q(\mathbf{s},\mathbf{a})$, where $q(\mathbf{s},\mathbf{a})$ is the joint state-action distribution under the previous policy.

Since we cannot explore the whole state-action space $\mathcal{S} \times \mathcal{A}$, we have to approximate the expected reward by the mean of the rewards in the samples $\{(\mathbf{s}_1, \mathbf{a}_1, \mathbf{s}'_1, R_1), \cdots, (\mathbf{s}_n, \mathbf{a}_n, \mathbf{s}'_n, R_n)\}$. It is shown in (Peters et al., 2010) that the optimal solution can be computed by multiplying the joint state-action distribution $q(\mathbf{s},\mathbf{a})$ of the previous policy by the exponential of the Bellman error:

$$\begin{aligned}
&\pi(\mathbf{a}|\mathbf{s})\mu_\pi(\mathbf{s}) \propto q(\mathbf{s},\mathbf{a}) \exp\left(\frac{\delta(\mathbf{s},\mathbf{a},V)}{\eta}\right) \\
&\text{where } \delta(\mathbf{s},\mathbf{a},V) = \mathcal{R}^{\mathbf{a}}_{\mathbf{s}} + \mathbb{E}_{\mathbf{s}'}\left[V(\mathbf{s}')|\mathbf{s},\mathbf{a}\right] - V(\mathbf{s})
\end{aligned} \tag{2}$$

Here $V(\mathbf{s})$ and η denote Lagrangian multipliers, and δ denotes the Bellman error. The Lagrange multipliers are obtained by minimizing the following dual optimization problem:

$$\begin{aligned} &V^*, \eta^* = \arg\min_{V,\eta} g(\eta, V) \\ &\text{s.t. } g(\eta, V) = \eta\epsilon + \eta \log\left(\sum_{i=1}^{n} \frac{1}{n} \exp\left(\frac{\delta(\mathbf{s}_i, \mathbf{a}_i, V)}{\eta}\right)\right) \\ &\text{where } (\mathbf{s}_i, \mathbf{a}_i) \sim q(\mathbf{s}, \mathbf{a}) \end{aligned} \tag{3}$$

It is very challenging to calculate the Bellman error δ since it depends on V and the transition probabilities $\mathcal{P}^{\mathbf{a}}_{\mathbf{ss}'}$. Note also that V is a function of $\mathbf{s}$ for which an analytical form is very difficult to obtain when solving optimization problem (3). Furthermore, $\mathcal{P}^{\mathbf{a}}_{\mathbf{ss}'}$ is usually unknown and very challenging to estimate from samples especially for high dimensional state-action spaces $\mathcal{S} \times \mathcal{A}$. Hence, the estimation of the expected value $\mathbb{E}_{\mathbf{s}\sim p(\mathbf{s}'|\mathbf{s},\mathbf{a})}[V(\mathbf{s}'|\mathbf{s}, \mathbf{a})]$ is also challenging since high dimensional integration is involved, even when $\mathcal{P}^{\mathbf{a}}_{\mathbf{ss}'}$ is known. Realizing these difficulties, Van Hoof et al. (2015) proposed to use Reproducing Kernel Hilbert Space (RKHS) embeddings to avoid parametric assumptions while allowing the accuracy to improve gracefully with the amount of data.

2.2 Relative Entropy Policy Search by Reproducing Kernel Hilbert Space Embeddings

Note that in the REPS framework, we need to estimate $\delta(\mathbf{s}, \mathbf{a}, V)$ which requires us to estimate V and $\mathcal{P}^{\mathbf{a}}_{\mathbf{ss}'}$ from data, which is very challenging. Van Hoof et al. (2015) proposed to estimate these quantities in a Reproducing Kernel Hilbert Space (RKHS) by using kernel embeddings.

First, assume that $\mathcal{H}$ is a Hilbert space that is rich enough to include V. We can then represent V by a linear combination of a set of basis functions in $\mathcal{H}$:

$$V(\cdot) = \sum_{\tilde{\mathbf{s}}_i \in \tilde{S}} \alpha_i k(\tilde{\mathbf{s}}_i, \cdot) \tag{4}$$

To calculate $\mathbb{E}_{\mathbf{s}'}[V(\mathbf{s}')|\mathbf{s}, \mathbf{a}]$, REPS-RKHS embeds the conditional $\mathcal{P}^{\mathbf{a}}_{\mathbf{ss}'}$ in the RKHS $\mathcal{H}$:

$$\mathcal{P}^{\mathbf{a}}_{\mathbf{ss}'} \mapsto \mu_{S'|\mathbf{s},\mathbf{a}} = \mathbb{E}_{\mathbf{s}'}[\phi(\mathbf{s}')|\mathbf{s}, \mathbf{a}], \tag{5}$$

where $\phi(\mathbf{s}') = k(\mathbf{s}', \cdot)$ is the feature vector of $\mathbf{s}'$. The advantage of using (conditional) kernel embeddings is that the expected value of any function with respect to a (conditional) probability density can be quickly calculated by the dot product between the embedded vector representing that function and the (conditional) kernel embedding (Song et al., 2009). Consequently, we obtain:

$$\mathbb{E}_{\mathbf{s}'}[V(\mathbf{s}')|\mathbf{s}, \mathbf{a}] = \left\langle V(\cdot), \boldsymbol{\mu}_{S'|\mathbf{s},\mathbf{a}} \right\rangle_{\mathcal{H}}$$

Empirical estimations of $\hat{\boldsymbol{\mu}}_{S'|\mathbf{s},\mathbf{a}}$ and $\hat{\mathbb{E}}_{\mathbf{s}'}[V(\mathbf{s}')|\mathbf{s}, \mathbf{a}]$ can be found as follows:

$$\hat{\boldsymbol{\mu}}_{S'|s,a} = \sum_{i=1}^{n} \beta_i(\mathbf{s}, \mathbf{a})\phi(\mathbf{s}'_i) \tag{6}$$

where $\beta(\mathbf{s}_i, \mathbf{a}_i) = (\mathbf{K}_{\mathbf{sa}} + \lambda\mathbf{I})^{-1}\mathbf{k}_{\mathbf{sa}}(\mathbf{s}_i, \mathbf{a}_i)$.

$$\mathbb{E}_{\mathbf{s}'}[V(\mathbf{s}')|\mathbf{s}, \mathbf{a}] - V(\mathbf{s}) = \boldsymbol{\alpha}^T\tilde{\mathbf{K}}_{\mathbf{s}}\boldsymbol{\beta}(\mathbf{s}, \mathbf{a}) - \boldsymbol{\alpha}^T\mathbf{k}_{\mathbf{s}}(\mathbf{s}) \tag{7}$$

In this case, the value function V can be estimated by

$$\begin{aligned} &V^* = \boldsymbol{\alpha}^{*T}\mathbf{k}_{\mathbf{s}}(\cdot), \\ &\boldsymbol{\alpha}^*, \eta^* = \arg\min_{\boldsymbol{\alpha},\eta} \eta\epsilon + \eta \log\left(\sum_{i=1}^{n} \frac{1}{n} \exp(\frac{\delta(\mathbf{s}_i, \mathbf{a}_i, \boldsymbol{\alpha})}{\eta})\right), \\ &\delta(\mathbf{s}, \mathbf{a}, \boldsymbol{\alpha}) = \mathcal{R}^{\boldsymbol{\alpha}}_{\mathbf{s}} + \boldsymbol{\alpha}^T\left(\tilde{\mathbf{K}}_{\mathbf{s}}\boldsymbol{\beta}(\mathbf{s}, \mathbf{a}) - \mathbf{k}_{\mathbf{s}}(s)\right), \end{aligned} \tag{8}$$

where $[\mathbf{K}_{\mathbf{sa}}]_{ij} = k_s(\mathbf{s}_i, \mathbf{s}_j)k_a(\mathbf{a}_i, \mathbf{a}_j)$, $[\mathbf{k}_{\mathbf{sa}}((s), \mathbf{a})]_i = k_s(\mathbf{s}_i, \mathbf{s})k_a(\mathbf{a}_i, \mathbf{a})$, $\left[\tilde{\mathbf{K}}_{\mathbf{s}}\right]_{ij} = k_s(\tilde{\mathbf{s}}_i, \mathbf{s}'_j)$ and $[\mathbf{k}_{\mathbf{s}}(\mathbf{s})]_i = k_s(\tilde{\mathbf{s}}_i, \mathbf{s})$. The parameters obtained by solving the optimization problem in (8) can be inserted in (2) to obtain the optimal sample based policy $\pi(\mathbf{a}|\mathbf{s})$. However, since the state-action space is continuous, the sample-based policy needs to be generalized to the nearby data points. Van Hoof et al. (2015) used cost sensitive Gaussian Processes (Kober et al., 2011) to obtain the generalized policy $\tilde{\pi}(\mathbf{a}|\mathbf{s})$ for those state-action pairs that did not appear in the past trajectory as follows:

$$\begin{aligned} &\tilde{\pi}(\mathbf{a}|\mathbf{s}) = \mathcal{N}(\mu(\mathbf{s}), \sigma^2(\mathbf{s})), \mu(\mathbf{s}) = \mathbf{k}_{\mathbf{s}}(\mathbf{s})^T(\mathbf{K}_{\mathbf{s}} + \lambda\mathbf{D})^{-1}\mathbf{A}, \\ &\sigma^2(\mathbf{s}) = k + \lambda - \mathbf{k}_{\mathbf{s}}(\mathbf{s})^T(\mathbf{K}_{\mathbf{s}} + \lambda\mathbf{D})^{-1}\mathbf{k}_{\mathbf{s}}(\mathbf{s}), \end{aligned} \tag{9}$$

where $k = k(\mathbf{s}, \mathbf{s})$, $\mathbf{k}_{\mathbf{s}}(\mathbf{s}) = \phi(\mathbf{s})^T\boldsymbol{\Phi}$, $\mathbf{K}_{\mathbf{s}} = \boldsymbol{\Phi}^T\boldsymbol{\Phi}$, $\mathbf{A} = [\mathbf{a}_1, \cdots, \mathbf{a}_n]^T$, $\mathbf{D}$ is a diagonal matrix with $\mathbf{D}_{ii} = (w_i/\max_i w_i)^{-1}$ where $w_i = \exp(\delta(\mathbf{s}_i, \mathbf{a}_i)/\eta^*)$, and λ is a regularization parameter. The REPS-RKHS algorithm is summarized in Algorithm 1.

3 Online REPS-RKHS

The original REPS-RKHS algorithm has been shown to outperform several other state-of-the-art algorithms such as sample based model, feature based REPS (Daniel et al., 2013; Peters et al., 2010), approximate value iteration (Grünewälder et al., 2012) and non-parametric approximate linear programming (Pazis and Parr, 2011). However, one of the drawbacks of the original algorithm is that we need to learn the model in a batch way, i.e., we control the system under the previous policy π^n and collect the resulting trajectory $\{(\mathbf{s}_0, \mathbf{a}_0, \mathbf{s}'_0, R_0), (\mathbf{s}_1, \mathbf{a}_1, \mathbf{s}'_1, R_1), \cdots, (\mathbf{s}_n, \mathbf{a}_n, \mathbf{s}'_n, R_n)\}$.

Algorithm 1 REPS with RKHS embeddings (Van Hoof et al., 2015))

1: **for** $i = 1, \cdots,$ MaxIteration **do**
2: generate roll-outs according to $\tilde{\pi}_{i-1}$
3: minimize kernel-based dual:

$$\eta^*, \boldsymbol{\alpha}^* \leftarrow \text{argmin} g(\eta, \boldsymbol{\alpha}) \tag{10}$$

4: calculate kernel embedding strengths:

$$\beta_j \leftarrow (\mathbf{K_{sa}} + \lambda \mathbf{I})^{-1} \mathbf{k_{sa}}(\mathbf{s}_j, \mathbf{a}_j) \tag{11}$$

5: calculate kernel-based Bellman errors:

$$\delta_j \leftarrow R_j + \boldsymbol{\alpha}{*}^T \left(\tilde{\mathbf{K}}_s \beta_j - \mathbf{k}_s(\mathbf{s}_j) \right) \tag{12}$$

6: calculate the sample weights:

$$w_j \leftarrow \exp(\delta_j / \eta^*) \tag{13}$$

7: fit a generalizing non-parametric policy:

$$\tilde{\pi}_i(\mathbf{a}|\mathbf{s}) = \mathcal{N}(\mu(\mathbf{s}; \mathbf{w}), \sigma^2(\mathbf{s}; \mathbf{w})) \tag{14}$$

8: **end for**

Based on the samples collected, we learn the model and obtain the new policy π^{n+1}. The above procedure is repeated until the policy converges. However, we prefer the policy to be updated online, i.e. whenever there is a new sample $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, R_i)$, we would like to use it to update the policy. We develop the following online learning algorithm. First, we use a Reduced Rank Approximation to reduce the computational complexity of the original algorithm from $\mathcal{O}(n^3)$ to $\mathcal{O}(m^2 n)$, where $m \ll n$. Based on the Reduced Rank Approximation, we further develop an online version of REPS-RKHS that uses the most recent sample $(\mathbf{s}, \mathbf{a}, \mathbf{s}', R)$ to update the model such that the algorithm will converge to the optimal policy faster than the original one.

3.1 Reduced Rank Approximation

First of all, we focus on approximating the matrix $\mathbf{K_{sa}}$. We propose to use the Reduced Rank Approximation (Rasmussen, 2006). Suppose we have a set of samples $S = \{(\mathbf{s}_0, \mathbf{a}_0, \mathbf{s}'_0, R_0), (\mathbf{s}_1, \mathbf{a}_1, \mathbf{s}'_1, R_1), \cdots, (\mathbf{s}_n, \mathbf{a}_n, \mathbf{s}'_n, R_n)\}$. We divide S into $\tilde{S}$ and $\bar{S}$ such that $\tilde{S} \bigcup \bar{S} = S$ and $\tilde{S} \bigcap \bar{S} = \emptyset$. Accordingly, we divide A into $\tilde{A}$ and $\bar{A}$, where $\tilde{A} \bigcup \bar{A} = A$ and $\tilde{A} \bigcap \bar{A} = \emptyset$. Denote $\mathbf{K}^{nn}_{\mathbf{sa}} = k(S,S)k(A,A)$, $\mathbf{K}^{mm}_{\mathbf{sa}} = k(\tilde{S},\tilde{S})k(\tilde{A},\tilde{A})$ and $\mathbf{K}^{nm}_{\mathbf{sa}} = k(S,\tilde{S})k(A,\tilde{A})$. We have

$$\mathbf{K}^{nn}_{\mathbf{sa}} = \left[\begin{array}{c:c} \mathbf{K}^{mm}_{\mathbf{sa}} & \mathbf{K}^{m(n-m)}_{\mathbf{sa}} \\ \hdashline \mathbf{K}^{(n-m)m}_{\mathbf{sa}} & \mathbf{K}^{(n-m)(n-m)}_{\mathbf{sa}} \end{array} \right] \tag{15}$$

We use the Reduced Rank Approximation method to approximate $\mathbf{K_{sa}}$ by the following equation:

$$\hat{\mathbf{K}}^{nn}_{\mathbf{sa}} \approx \mathbf{K}^{nm}_{\mathbf{sa}} (\mathbf{K}^{mm}_{\mathbf{sa}})^{-1} \mathbf{K}^{mn}_{\mathbf{sa}} \tag{16}$$

3.2 Online Update

The original REPS-RKHS algorithm updates the policy by batch learning, i.e. the agent is controlled under one policy $\pi^{(k-1)T}$ for a period of T steps during which it collects a batch of data $S_k = \{(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, R_i) | (k-1)T \leq i < kT\}$. Then, it uses S_k to learn a new policy π^{kT}. There are two important drawbacks with batch learning. First, the algorithm has to wait until a sufficient amount of data has been collected and thus the algorithm might not be able to capture the dynamics of the system (environment) rapidly enough, especially in some non-stationary environment. Second, as the original algorithm belongs to kernel methods, the computational complexity is $\mathcal{O}(n^3)$, which is high and thus the original algorithm might not be applicable to tasks that require realtime operations.

In this section, we develop the Online REPS-RKHS algorithm based on the reduced rank REPS-RKHS model. The most significant advantage of the online algorithm over the original one is that our algorithm can update the model online. To be more specific, suppose we learn the model M^n based on n samples and we have the policy $\pi^n(\mathbf{a}|\mathbf{s})$. The agent is controlled under policy π^n and then we have the new tuple $\mathbf{s}_n, \mathbf{a}_n, \mathbf{s}'_n, R_n$. By incorporating the new information, we update the policy from π^n to π^{n+1} accordingly.

Suppose there are n samples, we learn the following model:

$$\begin{aligned} \boldsymbol{\beta}^n_j &\leftarrow (\hat{\mathbf{K}}^{nn}_{\mathbf{sa}} + \lambda \mathbf{I})^{-1} \mathbf{k_{sa}}(\mathbf{s}_j, \mathbf{a}_j) \\ \delta^n_j &\leftarrow R^n_j + \boldsymbol{\alpha}^{*T} (\tilde{\mathbf{K}}^n_{\mathbf{s}} \boldsymbol{\beta}^n_j - \mathbf{k_s}(\mathbf{s}_j)) \\ w^n_j &\leftarrow \exp(\delta^n_j / \eta^*), \forall j \in [0, n] \end{aligned} \tag{17}$$

When the $(n+1)^{th}$ sample arrives, we need to calculate

$$\boldsymbol{\beta}^{n+1}_{n+1}, \delta^{n+1}_{n+1}, w^{n+1}_{n+1},$$

which are the new information we need to update the current model and policy. One way to obtain those statistics is to augment the original training data with the new sample and learn the model based on the augmented data. However, this approach is extremely computational demanding and requires a large amount of memory when n becomes large. To tackle this problem, we use the reduced rank approximation, for which the computational complexity and the storage complexity only depends on m, where $m \ll n$ is the cardinality of the subset of regressors. The approximation process sarts with an LU decomposition:

$$\mathbf{K}^{mm}_{\mathbf{sa}} = \mathbf{L}\mathbf{L}^T \text{ and } (\mathbf{K}^{mm}_{\mathbf{sa}})^{-1} = \mathbf{L}^{-T}\mathbf{L}^{-1} \tag{18}$$

Let $\mathbf{Q}_n = \mathbf{K}^{nm}_{\mathbf{sa}} \mathbf{L}^{-T}$. We approximate the matrix $(\mathbf{K}^{nn}_{\mathbf{sa}} +$

$\lambda \mathbf{I}_n)^{-1}$, which is a key part to compute $\boldsymbol{\beta}$ (see Algorithm 1), as follows:

$$\begin{aligned}&(\hat{\mathbf{K}}_{\mathbf{sa}}^{nn} + \lambda \mathbf{I}_n)^{-1} \\ &\approx \lambda^{-1}\mathbf{I}_n - \lambda^{-1}\mathbf{Q}_n(\lambda \mathbf{I}_m + \mathbf{Q}_n^T\mathbf{Q}_n)^{-1}\mathbf{Q}_n^T\end{aligned}$$

Let $\mathbf{\Pi}_n = (\mathbf{Q}_n^T\mathbf{Q}_n + \lambda \mathbf{I}_m)^{-1}$. We obtain

$$(\hat{\mathbf{K}}_{\mathbf{sa}} + \lambda \mathbf{I}_n)^{-1} = \lambda^{-1}\mathbf{I}_n - \lambda^{-1}\mathbf{Q}_n\mathbf{\Pi}_n\mathbf{Q}_n^T.$$

Denote by

$$\mathbf{k}_{\mathbf{sa}}^{n+1} = \begin{bmatrix} k((\tilde{\mathbf{s}}_1, \tilde{\mathbf{a}}_1), (\mathbf{s}_{n+1}, \mathbf{a}_{n+1})) \\ \vdots \\ k((\tilde{\mathbf{s}}_m, \tilde{\mathbf{a}}_m), (\mathbf{s}_{n+1}, \mathbf{a}_{n+1})) \end{bmatrix} \tag{19}$$

$$\tilde{\mathbf{k}}_{\mathbf{s}}^{n+1} = \begin{bmatrix} k(\tilde{\mathbf{s}}_1, \mathbf{s}'_{n+1}) \\ \vdots \\ k(\tilde{\mathbf{s}}_m, \mathbf{s}'_{n+1}) \end{bmatrix} \quad \mathbf{k}_{\mathbf{s}}^{n+1} = \begin{bmatrix} k(\tilde{\mathbf{s}}_1, \mathbf{s}_{n+1}) \\ \vdots \\ k(\tilde{\mathbf{s}}_m, \mathbf{s}_{n+1}) \end{bmatrix} \tag{20}$$

the new information from the new sample $(\mathbf{s}_{n+1}, \mathbf{a}_{n+1}, \mathbf{s}'_{n+1}, R_{n+1})$ when the $(n+1)^{th}$ sample arrives. We use the new information to update the model as follows:

$$\begin{aligned}&\mathbf{\Pi}_{n+1} = (\lambda \mathbf{I}_m + \mathbf{Q}_{n+1}^T\mathbf{Q}_{n+1})^{-1} \\ &= (\lambda \mathbf{I}_m + \mathbf{Q}_n^T\mathbf{Q}_n + \mathbf{L}^{-1}\mathbf{k}_{\mathbf{sa}}^{n+1}(\mathbf{k}_{\mathbf{sa}}^{n+1})^T\mathbf{L}^{-T})^{-1}\end{aligned} \tag{21}$$

where

$$\mathbf{Q}_{n+1} = \begin{bmatrix} \mathbf{K}_{\mathbf{sa}}^{nm} \\ (\mathbf{k}_{\mathbf{sa}}^{n+1})^T \end{bmatrix} \mathbf{L}^{-T} = \begin{bmatrix} \mathbf{Q}_n \\ (\mathbf{k}_{\mathbf{sa}}^{n+1})^T\mathbf{L}^{-T} \end{bmatrix}$$

According to the Woodbury identity, we have

$$\mathbf{\Pi}_{n+1} = \mathbf{\Pi}_n - \frac{\mathbf{\Pi}_n\mathbf{L}^{-1}\mathbf{k}_{\mathbf{sa}}^{n+1}(\mathbf{k}_{\mathbf{sa}}^{n+1})^T\mathbf{L}^{-T}\mathbf{\Pi}_n}{1 + (\mathbf{k}_{\mathbf{sa}}^{n+1})^T\mathbf{L}^{-T}\mathbf{\Pi}_n\mathbf{L}^{-1}(\mathbf{k}_{\mathbf{sa}}^{n+1})} \tag{22}$$

Let $\mathbf{q}_{n+1} = \mathbf{L}^{-1}\mathbf{k}_{\mathbf{sa}}^{n+1}$ and substitute $\mathbf{q}_{n+1}$ in the aove equation to obtain:

$$\mathbf{\Pi}_{n+1} = \mathbf{\Pi}_n - \frac{\mathbf{\Pi}_n\mathbf{q}_{n+1}\mathbf{q}_{n+1}^T\mathbf{\Pi}_n}{1 + \mathbf{q}_{n+1}^T\mathbf{\Pi}_n\mathbf{q}_{n+1}} \tag{23}$$

We can see that the key matrix to compute $\boldsymbol{\beta}^{n+1}$ can be approximated by

$$\begin{aligned}&(\hat{\mathbf{K}}_{\mathbf{sa}}^{n+1} + \lambda \mathbf{I}_{n+1})^{-1} \approx \lambda^{-1}\mathbf{I}_{n+1} - \lambda^{-1}\mathbf{Q}_{n+1}\mathbf{\Pi}_{n+1}\mathbf{Q}_{n+1}^T \\ &= \lambda^{-1}\begin{bmatrix} \mathbf{I}_n & \\ & 1 \end{bmatrix} - \lambda^{-1}\begin{bmatrix} \mathbf{Q}_n \\ \mathbf{q}_{n+1}^T \end{bmatrix}\mathbf{\Pi}_{n+1}\begin{bmatrix} \mathbf{Q}_n^T & \mathbf{q}_{n+1} \end{bmatrix}\end{aligned}$$

Note that

$$\boldsymbol{\beta}_{n+1}^{n+1} = (\mathbf{K}_{\mathbf{sa}}^{n+1} + \lambda \mathbf{I}_{n+1})^{-1}\mathbf{k}_{\mathbf{sa}}^{n+1}(\mathbf{s}_{n+1}, \mathbf{a}_{n+1}) \tag{24}$$

To compute $\mathbf{k}_{\mathbf{sa}}^{n+1}(\mathbf{s}_{n+1}, \mathbf{a}_{n+1})$, we need to store all previous samples, which is unrealistic if the system keeps evolving. To tackle this problem, we use the following approximation:

$$\begin{aligned}\mathbf{k}_{\mathbf{sa}}^{n+1}(\mathbf{s}_{n+1}, \mathbf{a}_{n+1}) &\approx \mathbf{K}_{\mathbf{sa}}^{nm}(\mathbf{K}_{\mathbf{sa}}^{mm})^{-1}\mathbf{k}_{\mathbf{sa}}^{n+1} \\ &= \mathbf{Q}_n\mathbf{q}_{n+1}.\end{aligned} \tag{25}$$

After some algebraic manipulations, we arrive at

$$\boldsymbol{\beta}_{n+1}^{n+1} = \begin{bmatrix} \lambda^{-1}\mathbf{Q}_n(\frac{\lambda + \mathbf{q}_{n+1}^T\mathbf{q}_{n+1}}{1 + \mathbf{q}_{n+1}^T\mathbf{\Pi}_n\mathbf{q}_{n+1}}\mathbf{\Pi}_n - \mathbf{\Pi}_{n+1})\mathbf{q}_{n+1} \\ \lambda^{-1}\frac{1 - \mathbf{q}_{n+1}^T\mathbf{q}_{n+1}}{1 + \mathbf{q}_{n+1}^T\mathbf{\Pi}\mathbf{q}_{n+1}} + \frac{(\mathbf{q}_{n+1}^T\mathbf{\Pi}_n\mathbf{q}_{n+1})^2}{1 + \mathbf{q}^T\mathbf{\Pi}_n\mathbf{q}_{n+1}} \end{bmatrix} \tag{26}$$

and

$$\delta_{n+1}^{n+1} = R_{n+1} + \boldsymbol{\alpha}^{*T}(\tilde{\mathbf{K}}_{\mathbf{s}}^{n+1}\boldsymbol{\beta}_{n+1}^{n+1} - \mathbf{k}_{\mathbf{s}}(\mathbf{s}_{n+1})), \tag{27}$$

where $\tilde{\mathbf{K}}_{\mathbf{s}}^{n+1} = \left[\tilde{\mathbf{K}}_{\mathbf{s}}^{n}, \tilde{\mathbf{k}}_{\mathbf{s}}^{n+1}\right]$. Let $\mathbf{H}_n = \tilde{\mathbf{K}}_{\mathbf{s}}^{n}\mathbf{Q}_n$, which always has dimensions $m \times m$. After computing $\boldsymbol{\beta}_{n+1}^{n+1}$ and δ_{n+1}^{n+1}, we can easily obtain:

$$w_{n+1}^{n+1} = \exp(\delta_{n+1}^{n+1}/\eta^*), \tag{28}$$

which is the regularizer for the new information from the new sample when we integrate it to update our model.

3.3 Approximating the non-parametric policy generation

The original model proposed by (Van Hoof et al., 2015) cannot integrate the new information easily. In order to make the model online, we develop the following Gaussian Process based on the subset of regressors method (Rasmussen, 2006) to derive the generalized non-parametric policy:

$$\begin{aligned}&\tilde{\pi}(\mathbf{a}|\mathbf{s}) = \mathcal{N}(\mu_n(\mathbf{s}), \sigma_n^2(\mathbf{s})), \\ &\mu_n(\mathbf{s}) = \mathbf{k}_{\mathbf{s}}(\mathbf{s})^T(\mathbf{K}_{\mathbf{s}}^{mn}\mathbf{D}_n\mathbf{K}_{\mathbf{s}}^{nm} + \lambda\mathbf{K}_{\mathbf{s}}^{mm})^{-1}\mathbf{K}_{\mathbf{s}}^{mn}\mathbf{A}_n, \\ &\sigma^2(\mathbf{s}) = \lambda\mathbf{k}_{\mathbf{s}}(\mathbf{s})^T(\mathbf{K}_{\mathbf{s}}^{mn}\mathbf{D}_n\mathbf{K}_{\mathbf{s}}^{nm} + \lambda{\mathbf{K}_{\mathbf{s}}}^{mm})^{-1}\mathbf{k}_{\mathbf{s}}(\mathbf{s})\end{aligned} \tag{29}$$

where $\mathbf{k}_{\mathbf{s}}(\mathbf{s}) = [k(\tilde{\mathbf{s}}_1, \mathbf{s}), \cdots, k(\tilde{\mathbf{s}}_m, \mathbf{s})]$, $\mathbf{K}_{\mathbf{s}}^{mn} = \mathbf{K}_{\mathbf{s}}^{nmT} = k(\tilde{S}, S)$, $\mathbf{K}_{\mathbf{s}}^{mm} = k(\tilde{S}, \tilde{S})$, $[\mathbf{D}_n]_{ii} = w_i/C = \exp(\delta_i/\eta^*)/C$ and C is a normalization constant. We have

$$\mathbf{D}_{n+1} = \left[\begin{array}{c|c} \mathbf{D}_n & \mathbf{0} \\ \hline \mathbf{0} & \exp(\delta_{n+1}^{n+1}/\eta^*) \end{array}\right] \tag{30}$$

where $\delta_{n+1}^{n+1} = R_{n+1} + \boldsymbol{\alpha}^{*T}(\tilde{\mathbf{K}}_{\mathbf{s}}^{n+1}\boldsymbol{\beta}_{n+1}^{n+1} - \mathbf{k}_{\mathbf{s}}(\mathbf{s}_{n+1}))$

Denote by $\mathbf{\Xi}_n = ({\mathbf{K}_{\mathbf{s}}}^{mn}\mathbf{D}_n{\mathbf{K}_{\mathbf{s}}}^{nm} + \lambda{\mathbf{K}_{\mathbf{s}}}^{mm})^{-1}$, $\mathbf{Y}_n = {\mathbf{K}_{\mathbf{s}}}^{mn}\mathbf{A}_n$, and $\mathbf{M}_n = \mathbf{\Xi}_n\mathbf{Y}_n$ where $\mathbf{M}_n$ is our model

learnt from n samples. When the new sample is available, we update the model according to the following formulas:

$$\boldsymbol{\Xi}_{n+1} = \boldsymbol{\Xi}_n - \frac{\exp(\delta_{n+1}^{n+1})/\eta^*}{C} \frac{\boldsymbol{\Xi}_n \mathbf{k_s}^{n+1} (\mathbf{k_s}^{n+1})^T \boldsymbol{\Xi}_n}{1+\exp(\delta_{n+1}^{n+1}/\eta^*)/C (\mathbf{k_s}^{n+1})^T \boldsymbol{\Xi}_n \mathbf{k_s}^{n+1}} \tag{31}$$

$$\mathbf{Y}_{n+1} = \mathbf{K_s}^{m,n+1} \mathbf{A}_{n+1} = \mathbf{Y}_n + \mathbf{k_s}^{n+1} \mathbf{a}_{n+1}^T \tag{32}$$

The new model is updated by

$$\mathbf{M}_{n+1} = \boldsymbol{\Xi}_{n+1} \mathbf{Y}_{n+1} \tag{33}$$

Note that although we keep adding rows or columns which are corresponding to the new sample, the actual dimensionality of our model remains unchanged, which is m. Algorithm 2 summarizes the online REPS-RKHS algorithm proposed in this paper.

Algorithm 2 Online REPS with RKHS embeddings

1: Initially, use Random Policy to control the agent and collect n samples. Use Algorithm 1 to learn

$$\eta^*, \boldsymbol{\alpha}^*, \boldsymbol{\beta}_j^n, \delta_j^n, w_j^n, \forall j \in [0, n] \tag{34}$$

and calculate $\mathbf{K}_{\mathbf{sa}}^{n,m}$, $\mathbf{K}_{\mathbf{sa}}^{m,m}$, $\boldsymbol{\Pi}_n$, $\boldsymbol{H}_n$, $\mathbf{D}_n$, $\boldsymbol{\Xi}_n$,$\boldsymbol{Y}_n$, $\mathbf{M}_n$ and π^n.
2: Let $k = n$
3: **while** termination condition is not true **do**
4: Use π^k to control the agent.
5: Collect the new tuple $(\mathbf{s}_{k+1}, \mathbf{a}_{k+1}, \mathbf{s}'_{k+1}, R_{k+1})$.
6: Update $\boldsymbol{\Pi}_{k+1}$, $\boldsymbol{\Xi}_{k+1}$, $\boldsymbol{Y}_{k+1}$ and $\boldsymbol{M}_{k+1}$ by Eqs $(23) \sim (33)$.
7: Update policy π^{k+1}.
8: $k \leftarrow k + 1$.
9: **end while**

Note that the resulting Online-RKHS-REPS behaves similarly to Recursive Least Square which has been shown in (Johnstone et al., 1982) to converge exponentially fast provided that the measurement vector sequence is persistently exciting and an exponential forgetting factor is used. Interested readers can refer to (Johnstone et al., 1982) for more details.

4 Experiments

In order to demonstrate the effectiveness of our proposed online method, we conduct experiments on three different MDPs; a synthetic toy task following (Lever and Stafford, 2015), Mountain Car (Sutton and Barto, 1998) and Underpowered Pendulum (Van Hoof et al., 2015). We compare the performance of the online algorithm with the original REPS-RKHS algorithm in terms of the average reward and the computational time. In the following experiments, we use the Matern class kernels (Rasmussen, 2006) for the state and action variables, where a Matern class kernel with $v = 1.5$ is defined as $k_{v=3/2}(r) = (1 + \frac{\sqrt{3}r}{l}) \exp(-\frac{\sqrt{3}r}{l})$.

4.1 Toy MDPs

The toy benchmark is a Markov chain on interval $\mathcal{S} \in [-4, 4]$ where $\mathcal{A} \in [-1, 1]$ and $r(s, a) = e^{-|s-3|}$. The dynamics can be described by $s' = s + a + \epsilon$ where ϵ is Gaussian noise with standard deviation $\sigma = 0.001$.

We reset the agent to $s = 0$ if it leaves the interval. We compare Online REPS-RKHS with the original Batch REPS-RKHS. For each algorithm, we use a random policy as the initial policy for 9 roll-outs where each roll-out has 40 steps. For REPS-RKHS-Online, we use the trajectory consisting of 360 samples to build the reduced rank approximated model. The model is updated whenever a new (s, a, s', r) is available. For REPS-RKHS-Batch, we update the model every 3 roll-outs. We calculate the average reward for each roll-out and the results are shown in Figure 1.

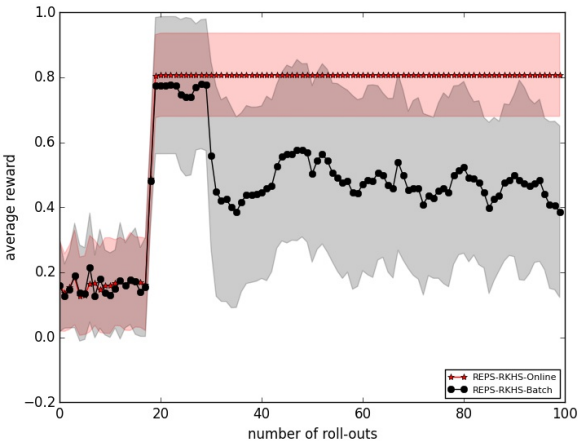

Figure 1: Toy MDP Problem

Based on Figure 1, we can see that REPS-RKHS-Online finds a better policy than REPS-RKHS-Batch in terms of the average reward. in the early roll-outs, REPS-RKHS-Batch finds a good policy; however, in the later roll-outs REPS-RKHS-Batch fails to stick to the good policy and the later policies become worse than the early policies. A possible explanation is that REPS-RKHS-Batch updates the policy by batch learning based on recent samples generated by recent policies where the state-action space is not explored as much as the initial random policy and consequently, the new policy might not perform as well as the first policy learnt from samples generated by the initial random policy. Another possible cause could be that when the REPS-RKHS-Batch algorithm updates the policy, hyper-parameter optimization is performed, but converges to some bad values. In contrast, for REPS-RKHS-Online, the policy is learnt from samples generated from the initial random policy and then is updated by new samples. In this case, the first batch of samples explore the state-action

space extensively and thus the REPS-RKHS-Online suffers less from local optima.

In order to show that REPS-RKHS-Online is computationally more efficient than the REPS-RKHS-Batch algorithm, we also report the running time of both algorithms for 100 roll-outs, where each roll-out has 20 steps. For REPS-RKHS-Online, we use $m = n$ and $m = n/2$ samples to form the subset of regressors $\tilde{S}$. For REPS-RKHS-Batch, we update the policy every 10 roll-outs. The running times are given in Figure 2.

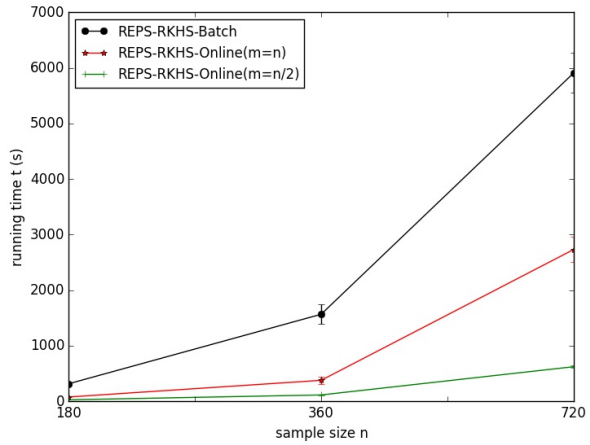

Figure 2: Comparison of running time

Based on Fig. 2, we can see that the running time of REPS-RKHS-Online is much less than that of REPS-RKHS-Batch, even when $m = n$ because REPS-RKHS-Online does not need to train the model every 10 roll-outs where the hyper-parameter optimization is very time consuming. When $m \ll n$, the speed up is much more significant, and thus the REPS-RKHS-Online is more scalable and appropriate for solving real problems that are time sensitive.

4.2 Mountain Car

The second benchmark is the Mountain Car problem (Lever and Stafford, 2015; Sutton and Barto, 1998). A car located at the bottom of a valley is to be controlled by the agent and the objective is to drive passed the target position of a hill. However, the car is underpowered and it cannot reach the target position by directly climbing up the hill. It has to climb the opposite hill in order to use gravity to boost its acceleration as it goes down and then up towards the goal. States $s = (x, v)$ are defined by position and velocity, $\mathcal{S} = (-1.2, 0.7) \times (-0.07, 0.07)$, $\mathcal{A} = [-1, 1]$ and $r(s, a) = e^{-8(x-0.6)^2}$ and $s_0 = (-0.5, 0)$. The dynamics are $x' = x + v + \epsilon_1, v' = v + 0.001a - 0.0025\cos(3x) + \epsilon_2/10$, where ϵ_1, ϵ_2 are Gaussian random variables with standard deviation 0.02. We compare Online REPS-RKHS with the original Batch REPS-RKHS. For each algorithm, we use a random policy as the initial policy for 9 roll-outs, and each roll-out consists of 40 steps. For REPS-RKHS-Online, we use a trajectory consisting of 360 samples to build the reduced rank approximated model. The model is updated whenever a new (s, a, s', r) is available. For REPS-RKHS-Batch, we update the model every 3 iterations. We calculate the average reward for each roll-out and the results are shown in Figure 3.

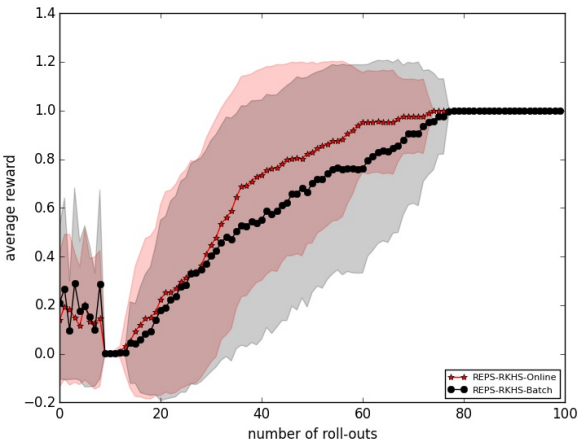

Figure 3: Mountain Car Problem

Based on Figure 3, we can see that both REPS-RKHS-Online and REPS-RKHS-Batch can find the optimal policy and drive the mountain-car to the target position. From the curves of average rewards, we can see that the mountain-car controlled by REPS-RKHS-Online moves towards the target position faster than the mountain car controlled by REPS-RKHS-Batch. The experimental results demonstrate that with online learning, the policy adapts quickly as the online algorithm uses every new sample to update the model.

4.3 Low-Dimensional Swing-Up Experiment

In this experiment, we simulate a pendulum with length $l = 1m$ and mass $m = 1kg$ distributed evenly along its length. We apply a torque a at the pivot. The dynamic of the pendulum is modeled by $\ddot{\theta} = (glm\sin\theta + a - k\dot{\theta})/(ml^2)$, where $k = 0.05Ns$ is a friction coefficient and $g = 9.81$ is the gravitational constant. The control frequency is $20Hz$. The maximum torque is $20Nm$. We also apply additive noise to the controls. The reward function is $r(s, a) = -10\theta^2 - 0.1\dot{\theta}^2 - 10^{-3}a^2$, where θ is mapped to $[-0.5\pi, 1.5\pi)$ to prefer swing-up counter-clockwise. We compare Online REPS-RKHS algorithm with the original Batch REPS-RKHS. For each algorithm, we use a random policy as the initial policy for 18 roll-outs, and each roll-out consists of 40 steps. For REPS-RKHS-Online, we use a trajectory consisting of 720 samples to build the reduced rank approximated model. The model is updated whenever

a new (s, a, s', r) is available. For REPS-RKHS-Batch, we update the model every 3 roll-outs.

The algorithms are given the angle θ and the angular velocity $\dot{\theta}$ directly, which define the state vector $\mathbf{s} = [\theta, \dot{\theta}]^T$. We calculate the average reward for each roll-out and the results are shown in Figure 4.

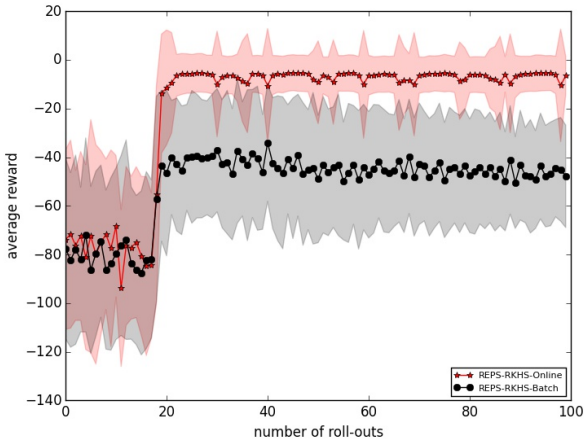

Figure 4: Low Dimensional Swing-Up

Based on Figure 4, we can see that a given small sample size (720), REPS-RKHS-Online performs better than REPS-RKHS-Batch since REPS-RKHS-Online controls the pendulum to the upstanding position within 100 roll-outs. REPS-RKHS-Batch improves the policy, but fails to find an optimal one within 100 roll-outs. Usually, a much larger sample size should be provided to REPS-RKHS-Batch to help it learn a good policy. REPS-RKHS-Online generally requires less samples and less iterations than REPS-RKHS-Batch to converge to an optimal solution.

4.4 High-Dimensional Swing-up Experiment

Following Van Hoof et al. (2015), we also conduct the swing-up experiment with high dimensional input. An underpowered pendulum is controlled, but only the raw image of the current position of the pendulum is given to the algorithm instead of the direct input of the angle and the angular velocity. The experimental setting is the same as the low dimensional swing-up experiment except for the input. The experimental results are shown in Figure 5.

Based on Figure 5, we can see that although it converges to the optimal policy slower than with the direct input of angle and angular velocity, REPS-RKHS-Online successfully controls the pendulum to the upstanding position based on the raw pixel input of an image capturing the instantaneous position of the pendulum, which illustrates the power of kernel embeddings to handle high dimensional input.

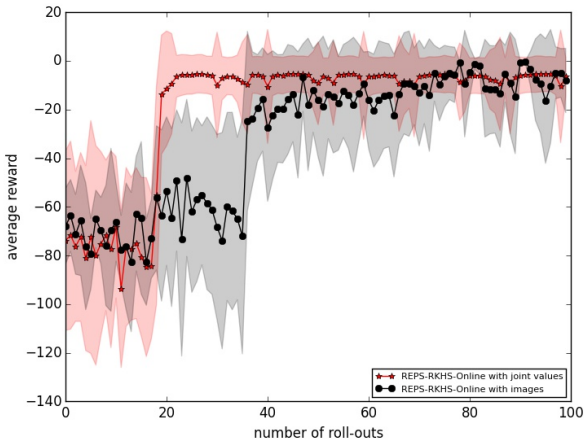

Figure 5: High Dimensional Swing-Up

5 Conclusion

In this paper, we proposed an Online Relative Entropy Policy Search using Reproducing Kernel Hilbert Space Embeddings based on the state-of-the-art REPS-RKHS algorithm. The online algorithm incorporates the most recent information, i.e., the state-action-state-reward tuple to update the model such that the algorithm is able to capture and respond to the rapidly changing dynamics of the system (environment). Furthermore, the online REPS-RKHS algorithm updates the model in an online fashion instead of batch learning and thus the computational complexity is reduced significantly, making the online algorithm much more suitable for many real world problems that require real time processing such as network control. Based on the experimental results, we find that the online algorithm performs better than the original batch learning algorithm in terms of converging to good policies faster. The computation time is also much less than the original batch algorithm. The theoretical contribution and the empiricall results show that our proposed online REPS-RKHS algorithm is a very promising method to address some real problems with high dimensional input and unknown transition functions.

However, our proposed algorithm has some limitations. First, it depends on the optimization of the model parameters and the hyper-parameters of the kernels. Second, the algorithm requires the selection of a subset of regressors for value function approximation and reduced rank approximation, which is fixed after the selection. Developing online algorithms which can replace the subset of regressors such that the model becomes more adaptive to the environment remains open and challenging. In future work, we will continue to investigate these issues.

References

Christian Daniel, Gerhard Neumann, Oliver Kroemer, and Jan Peters. Learning sequential motor tasks. In *Robotics and Automation (ICRA), 2013 IEEE International Conference on*, pages 2626–2632. IEEE, 2013.

Arthur Gretton, Karsten M Borgwardt, Malte J Rasch, Bernhard Schölkopf, and Alexander Smola. A kernel two-sample test. *The Journal of Machine Learning Research*, 13(1):723–773, 2012.

Steffen Grünewälder, Guy Lever, Luca Baldassarre, Massi Pontil, and Arthur Gretton. Modelling transition dynamics in mdps with rkhs embeddings. In *Proceedings of the 29th International Conference on Machine Learning, ICML 2012*, volume 1, pages 535–542, 2012.

Richard M Johnstone, C Johnson, Robert R Bitmead, and BDO Anderso. Exponential convergence of recursive least squares with exponential forgetting factor. In *Decision and Control, 1982 21st IEEE Conference on*, pages 994–997. IEEE, 1982.

Jens Kober, Erhan Oztop, and Jan Peters. Reinforcement learning to adjust robot movements to new situations. In *IJCAI Proceedings-International Joint Conference on Artificial Intelligence*, volume 22, page 2650, 2011.

Guy Lever and Ronnie Stafford. Modelling policies in mdps in reproducing kernel hilbert space. In *Proceedings of the Eighteenth International Conference on Artificial Intelligence and Statistics*, pages 590–598, 2015.

Y Nishiyama, A Boularias, A Gretton, and K Fukumizu. Hilbert space embeddings of pomdps. In *Conference on Uncertainty in Artificial Intelligence (UAI 2012)*, 2012.

Jason Pazis and Ronald Parr. Non-parametric approximate linear programming for mdps. In *AAAI*, 2011.

Jan Peters, Katharina Mülling, and Yasemin Altun. Relative entropy policy search. In *AAAI*, 2010.

Carl Edward Rasmussen. Gaussian processes for machine learning. 2006.

Alex J Smola and Bernhard Schölkopf. *Learning with kernels*. Citeseer, 1998.

Le Song, Jonathan Huang, Alex Smola, and Kenji Fukumizu. Hilbert space embeddings of conditional distributions with applications to dynamical systems. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 961–968. ACM, 2009.

Le Song, Kenji Fukumizu, and Arthur Gretton. Kernel embeddings of conditional distributions: A unified kernel framework for nonparametric inference in graphical models. *Signal Processing Magazine, IEEE*, 30(4):98–111, 2013.

Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.

Herke Van Hoof, Jan Peters, and Gerhard Neumann. Learning of non-parametric control policies with high-dimensional state features. In *Proceedings of the Eighteenth International Conference on Artificial Intelligence and Statistics*, pages 995–1003, 2015.