Napping for Functional Representation of Policy

Qing Da, Yang Yu, Zhi-Hua Zhou
National Key Laboratory for Novel Software Technology
Nanjing University, Nanjing 210023, China
{daq,yuy,zhouzh}@lamda.nju.edu.cn

ABSTRACT

Reinforcement learning aims at learning a policy from interactions with the environment to maximize the long-term reward. In practice, we commonly expect that the policy can be a nonlinear mapping from the state features to the candidate actions, and thus has the ability to fit complex decision situations. Functional representation, by which a function is represented as a combination of basis functions, is a powerful tool for learning non-linear functions, and has been used in policy learning (e.g., the non-parametric policy gradient (NPPG) method). Despite the many unique advantages of functional representation, it has a practical defect that a functional represented policy involves a lot of basis functions, and consequently the policy learning algorithm will be costed a lot of time in calculating the many constituting basis functions. This defect will badly hamper the functional representation from being practically applicable in reinforcement learning tasks, as the complex policies are to be continually evaluated. In this work, we proposed the *napping* mechanism to improve the efficiency of using the functional representation, which periodically simplifies the generated function by a simple approximation model along with the learning process. We integrated the napping mechanism into the NPPG algorithm, and carried out empirical studies. Experiment results showed that the NPPG with napping can not only drastically improve the training and predicting speed from the original NPPG, but also improve the performance significantly.

Categories and Subject Descriptors

I.2.6 [**Artificial Intelligence**]: Learning

General Terms

Algorithms

Keywords

functional representation, reinforcement learning, policy gradient

1. INTRODUCTION

A reinforcement learning agent receives rewards after a sequence of actions, and aims to learn the best immediate decisions to maximize the long-term rewards [28]. Various approaches have been developed for reinforcement learning problems, among which policy learning has shown to be a family of effective methods. A highlighted advantage of policy learning methods is the immunity to the policy degradation problem of value function learning methods [4], which estimate a value function for state-action pairs and derive the policy by greedily following the action that is associated with the largest value [6, 31]. Therefore, policy learning has attracted increasing attention and have been successfully applied in many domains [37, 30, 1], especially in robotics [22, 24, 19].

In policy learning, one of the foremost issues is to determine the representation of the policy. The linear representation is the simplest form, which maps the action as a linear combination of the state features, but is incapable to represent nonlinear mappings, while in almost all practical situations the nonlinearity is essential for a good policy. Approaches of learning nonlinear policies have been investigated, including the methods that shift the feature space using kernel mapping (e.g. [11]), the methods that employ parameterized nonlinear models (e.g. [32]), etc. However, it is still quite tricky to select the kernel function, or a proper parameterized nonlinear model, and moreover, since the selections are commonly done ahead of the learning process, it is hard to be improved along with the learning.

Meanwhile, the functional representation, by which a function is represented as a combination of basis functions, can represent nonlinear functions quite naturally. It has been applied in supervised learning and led to the state-of-the-art learning algorithms, including the famous AdaBoost [15] algorithm, the Gradient Boosting [16] algorithm, and many variant boosting algorithms. It has also been applied in reinforcement learning, e.g. the non-parametric policy gradients (NPPG) [18] method. Using the functional representation in reinforcement learning can have many unique advantages: functional representation can be quite powerful in representing nonlinear functions, thus avoids approximating a nonlinear optimal policy by a linear representation [29]; with a flexible base learner, functional gradient method adaptively generates a nonlinear policy, thus alleviates the difficulty of feature engineering or parameterized model selection; moreover, well-established machine learning approaches are readily useable to induce the basis functions, thus a strong generalization performance can be expected.

These advantages, however, come with a practical defect that learning a functional represented policy involves training and accumulating a lot of basis functions, all of which have to be invoked in every calculation of the policy output. Since the policy has to be repeatedly evaluated during both training and prediction stages of the reinforcement learning, learning functional representation policy suffers from a very large time cost for calculating every constituting basis functions, which hampers its applications in real-world reinforcement learning tasks. Moreover, in ensemble learning liter-

Appears in: *Alessio Lomuscio, Paul Scerri, Ana Bazzan, and Michael Huhns (eds.), Proceedings of the 13th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2014), May 5-9, 2014, Paris, France.*

atures, it has been proven that using too many models degrades the generalization ability [15, 20, 40]. We conjecture this is also true for our situation.

In this work, we propose a *napping* mechanism to reduce the time cost of using functional represented policy in reinforcement learning. The idea is to replace the learned function by a simple approximation function periodically along with the learning process. For a given policy formed by a set of models, an approximation model is obtained by mimicking the input-output behavior of the policy. To achieve this goal, two questions need to be addressed, i.e., how to do the mimicking and what kind of behaviors should the model mimic.

To the first question, we employ the point-wise approximation, which have been shown doable in decision-rule extraction literatures [13, 39, 34]. The behavior of the policy is exposed by observing its behavior on a collection of probing instances. A model is then trained on the probing instances with the observed behavior. As for the source of the probing instances, collecting fresh instance by sampling the policy would bring a high sampling and time cost, while keeping the historical instances would cost a large storage problem. Thus we propose to apply the reservoir sampling method [35] to keep only a small amount of historical instances. To the second problem, we investigate two approaches, mimicking the state-action value function and mimicking the final actions directly.

We implement the napping in the NPPG algorithm [18], which is a policy gradient approach in functional policy space, and conduct empirical studies on three domains to verify the effectiveness and efficiency of the napping mechanism. The experiment results confirm that NPPG with napping has a much smaller training and predicting time cost. Moreover, it is surprisingly observed that the napping that mimics the actions but not the state-action value function leads to a superior performance to the original NPPG algorithm.

The rest of this paper starts with an introduction of the background. Then the proposed approach is presented, which is followed by the empirical studies. The paper ends with a section of discussion and conclusion.

2. BACKGROUND

The reinforcement learning (RL) tasks are commonly studied based on the formal Markov decision process (MDP). An MDP is defined by the tuple (S, A, P, R), where S is the space of states, A is the space of actions, $P(s_j|s_i, a)$ is the transition probability denoting the probability of transitioning from state i to state j under action a, and $R(s, a)$ returns the immediate reward for the pair (s, a). An agent acts in such environment from an initial state following cycles of observing the state, making its decision that leads it to another state and receives an immediate reward. The core of an agent is its decision-making strategy, i.e., the policy. A policy is usually defined as $\pi(s, a) : S \times A \to \mathbb{R}$, which outputs the probability of choosing the action a under the state s. The goal of the agent is to learn a policy to maximize its total reward, through interacting with the environment in which the agent can only observe sampling outcomes of the transition probability P and the reward function R without explicitly awareness of their formulation.

One traditional branch of RL are value function based approaches, which aim to learn a value function of states or state-action pairs that approximates the Bellman optimality [5]. With a learned value function, the policy is simply formed by greedily choosing the action which leads to the highest value among all available actions. Such methods include Q-Learning [36] and SARSA [26] for finite state problems, and function approximation methods for continuous state problems [3, 27, 10, 14]. A limitation of the value function based approaches is that a better approximated value function does not necessarily lead to a better policy, which is described as the policy degradation problem [4].

2.1 Policy Gradient

A later developed branch are policy gradient approaches, which directly optimize a parameterized control policy to maximize the expected total reward of the policy in a fixed policy class via gradient ascent method [23]. Policy gradient approaches share many advantages, such as they avoid the policy degradation problem in value function based approaches, allow domain knowledge guided policy parameter customization, and can naturally handle large continuous state space and action space.

We take finite state and action spaces for example. In policy gradient approaches, the Gibbs policy is usually employed [30, 2]

$$\pi_\theta(s, a) = \frac{e^{\Psi(s,a)}}{\sum_{b \in A} e^{\Psi(s,b)}}, \tag{1}$$

where $\Psi(s, a)$ is the potential function that is often represented in a linear form $\Psi(s, a) = \theta^T \phi(s, a)$ with θ the parameters of π_θ and $\phi(s, a)$ the feature extraction function. In most cases, ϕ needs to be designed carefully by domain experts and is one of the keys to the success of policy gradient methods.

The goal of policy gradient methods is to maximize the per step long-term total expected reward of a policy π_θ, i.e., maximizing

$$\rho(\pi_\theta) = \lim_{T \to \infty} \frac{1}{T} E\{r_1 + \ldots + r_T | \pi_\theta\}.$$

The gradient of $\rho(\pi_\theta)$ with respect to the parameters θ can be derived as [30]

$$\frac{\partial \rho}{\partial \theta} = \sum_{s \in S} d^{\pi_\theta}(s) \sum_{a \in A} \frac{\partial \pi_\theta(s, a)}{\partial \theta} Q^{\pi_\theta}(s, a), \tag{2}$$

where Q^{π_θ} is the state-action value function and $d^{\pi_\theta}(s)$ is probability of s in the stationary distribution following policy π_θ. The environment is commonly assumed to be ergodic so that the stationary distribution exists and independent of the initial state.

Starting from a random initialization of θ_0, the parameter vector θ is optimized following the canonical gradient ascent method. At the t-th iteration, the current obtained policy $\pi_{\theta_{t-1}}$ is applied to explore the environment and the episode's data $\{(s, a)_i\}_{i=1}^n$ are collected to calculate the gradient

$$\nabla\theta_t = \sum_{s \in S'} \sum_{a \in A} \frac{\partial \pi_{\theta_{t-1}}(s, a)}{\partial \theta} Q^{\pi_{\theta_{t-1}}}(s, a). \tag{3}$$

Here S' is the set of collected states from episode's data, so the probability term is taken away since the sum of states from S' naturally induces such distribution. Then the policy parameter vector is updated as $\theta_t = \theta_{t-1} + \eta_t \nabla\theta_t$, where η_t is the step size. Usually the Q^π is also unknown, and can be estimated by either the Monte Carlo method as in REINFORCE [37] or the function approximation method as in [30].

2.2 NPPG

Contrasting to the policy gradient in a parametric policy space, the NPPG algorithm proposed in [18] searches for an optimal policy in a functional policy space. In traditional machine learning field, functional gradient approaches, i.e., boosting algorithms [15, 16, 12], which employ well-established learning techniques to train base leaner and produce adaptively nonlinear model alleviating the

difficult feature engineering problem, have been shown to be extremely powerful. Policy gradient in function space, as a combination of policy gradient and functional gradient, inherits both of their advantages, i.e., powerful and less feature engineering dependent.

NPPG employs the same form of Eq.(1) but uses a non-parametric potential function Ψ. Starting from a constant function Ψ_0 that in fact induces a random policy, Ψ is updated by functional gradient. At the t-th iteration, the current policy is applied to perform the task and collect episode's data to estimate the functional gradient

$$\frac{\partial \rho}{\partial \Psi}(\cdot) = \sum_{s \in S} d^{\pi}(s) \sum_{a \in A} \underbrace{\frac{\partial \pi(s,a)}{\partial \Psi}(\cdot) Q^{\pi}(s,a)}_{f_t(s,a)}, \tag{4}$$

where $\frac{\partial \pi(s,a)}{\partial \Psi}(\cdot) = \pi(s,a)(1-\pi(s,a))$ for the input (s,a). Note that the functional gradient itself is also a function. A regression model h_t is then trained to approximate the functional gradient on the collected data $\{(s,a,f_t(s,a))_i\}_{i=1}^n$ treating (s,a) as the input and $f_t(s,a)$ as the target variable. The potential function is updated as $\Psi_t = \Psi_{t-1} + \eta_t h_t$. After T iterations, we obtain the final potential function $\Psi_T = \Psi_0 + \sum_{t=1}^T \eta_t h_t$, and thus the final policy as

$$\pi_T(s,a) = \frac{e^{\Psi_0(s,a) + \sum_{t=1}^T \eta_t h_t(s,a)}}{\sum_b e^{\Psi_0(s,b) + \sum_{t=1}^T \eta_t h_t(s,b)}} \tag{5}$$

Despite many advantages of using the functional representation, it is obvious that NPPG trains and combines T models into the policy as can be observed in Eq.(5), which results $O(T|A|)$ evaluations every time a policy decision is calculated, therefore it is quite time consuming. Moreover, in ensemble learning literatures, it has been proven that using too many models degrades the generalization ability [15, 20, 40], which may also be true in our situation.

3. NAPPING MECHANISM

Our idea of reducing the training and predicting time of an agent with functional represented policy is to, instead of sleeplessly training and combining more models into the potential function as in the original NPPG method, let the agent take a nap periodically. During each nap, the agent trains a simple model to approximate the complex potential function accumulated so far. After that, the agent resumes the policy gradient procedure.

This idea is implemented in Algorithm 1. It is almost the same as the original NPPG method except the lines from 7 to 11. The ϵ-greedy strategy is applied here for exploration, i.e., with a probability of ϵ, a random action is chosen, and with the rest probability, a deterministic action of π_{t-1} is chosen, as in line 3. Once the number of models exceeds a pre-defined number nap, the agent takes a nap, which consists of three steps. Firstly we sample a collection of probing instances in line 8 (step (a)). Then an approximation model is built by mimicking the output of the potential function Ψ_t on the probing instances (step (b)), where ℓ is some loss function defining the quality of the mimicking. Finally, f is used to replace the original potential function (step (c)), where a transformation of f may be needed. By such napping strategy, we can keep the potential function to involve a constant number of models, thus accelerate the calculation of the policy.

It's important to note that the napping mechanism does not arbitrarily simplify the policy, but aims at reducing the unnecessary complexity of the potential function. There are at least two reasons why there can be an unnecessary complexity. First, it has been well recognized by the multiple classifier systems community that too many models will degrade the performance. Second, the NPPG method improves the combined models progressively, thus the importance of the beginning models fade out at later iterations, which makes the combined model has some redundancy that can be removed.

In the following three subsections, we will discuss on how to implement these three steps in detail.

Algorithm 1 NPPG with Napping

Input:
T: Number of iterations
ϵ: Probability for ϵ-greedy
nap: Napping interval
$\{\eta_t\}_{t=1}^T$: Step lengths
Output:
π: The learned policy
1: $\Psi_0(s,a) = 1, \forall (s,a) \in S \times A$
2: **for** $t = 1$ to T **do**
3: Collect episodes E_t by following π_{t-1} where $\pi_{t-1} = e^{\Psi_{t-1}(s,a)} / \sum_b e^{\Psi_{t-1}(s,b)}$ with ϵ-greedy
4: Generate functional gradient examples D_t from E_t as $\{(s,a,\frac{\partial \pi_{t-1}}{\partial \Phi}(s,a) Q^{\pi_{t-1}}(s,a))_i\}_{i=1}^n$
5: Train a regression model h_t from D_t
6: $\Psi_t = \Psi_{t-1} + \eta_t h_t$
7: **if** $t \mod nap = 0$ **then** {take a nap}
8: (a) Let D_p be a collection of probing instances
9: (b) Train an approximation model f by $f = \text{argmin}_g \sum_{\boldsymbol{x} \in D_p} \ell(\Psi_t(\boldsymbol{x}), g(\boldsymbol{x}))$
10: (c) Replace Ψ_t with a proper form of f
11: **end if**
12: **end for**
13: **return** $\pi = e^{\Psi_T(s,a)} / \sum_b e^{\Psi_T(s,b)}$

3.1 Collecting Probing Instances

Ideally, the probing instances are the state-action pairs sampled from the stationary distribution of the current policy. However, when the napping starts, we do not have any instance drawn from that distribution. Sampling fresh instances by executing the current policy is of a large extra cost. Therefore, we turn to use some historically visited instances as a rough alternate. Nevertheless, it is also costly to store every historical instance for drawing a sample. We then employ the *reservoir sampling* [35] to perform an online-sampling from all visited state-action pairs.

Reservoir sampling, summarized in Algorithm 2, is a randomized algorithm for sampling K examples from a streaming data set with unknown size. For sampling K instances, it only needs a buffer of size K. It accepts all instances as long as the buffer is not full, and otherwise accepts the i-th instance with probability K/i and the accepted instance then randomly replaces an instance in the buffer.

Algorithm 2 runs as a daemon thread that watches the agent. Once an instance of state-action pair is observed, it goes through its procedure to determine if the instance should be stored. Whenever it is asked to output a collection of sampled instances, it simply returns its current buffer.

3.2 Training an Approximation Model

To build an approximation model of a potential function Ψ, our first idea is to train a model by mimicking the value output of Ψ over the state-action pairs in the probing instances. The mimicking is done by minimizing the least square error in Eq.(6). One can

Algorithm 2 Reservoir Sampling

Input:
K: Number of examples to sample
Output:
D_p: The buffer
Let D_p be an empty multi-set
$i = 1$
for every observed state-action pair $\boldsymbol{x}$ **do**
  if $|D_p| < K$ **then**
    Put $\boldsymbol{x}$ into D_p
  else if toss a coin with head-up probability $\frac{K}{i}$, and gets a head **then**
    Choose q from $\{1, \ldots, K\}$ randomly
    Let $D_p(q)$ be replaced by $\boldsymbol{x}$
  end if
  $i = i + 1$
end for

employ any state-of-the-art regression algorithm for this task. The learned model is denoted as f_v.

$$f_v = \operatorname*{argmin}_f \sum_{(s,a)\in D_p} \left(\Psi_t(s,a) - f(s,a)\right)^2 \tag{6}$$

Similar idea of mimicking the state-action value function is widely used in reinforcement learning for function approximation. It has been disclosed that this could lead to the policy degradation problem, since the regression algorithm only focuses on reducing the least square error but ignores the order among actions. A model with a smaller least square error may however have a worse disorder among actions.

Therefore, our second idea is to directly mimic the action output implied by Ψ. This idea can be formulated as a weighted classification problem as in Eq.(7), where we only use the state s but not the action in the probing instances. The learned model is denoted as f_a, which inputs a state and outputs an action.

$$f_a = \operatorname*{argmin}_f \sum_{s\in D_p} \sum_{a\in A} \pi_t(s,a) I(f(s) \neq a), \tag{7}$$

where $\pi(s,a)$ is the probability value of the policy derived from Ψ, and I is the indicator function that returns 1 if its inner expression is true and 0 otherwise.

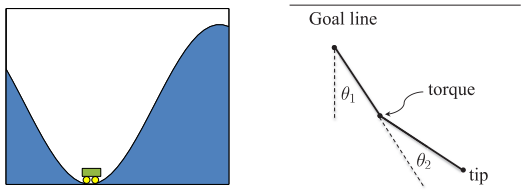
In order to train a classifier according to Eq.7, one needs to further construct a training data set from the probing instances by replicating the one state $|A|$ times each associated with a label as the action and weight as the probability derived from Ψ. We thus choose a more efficient formulation as in Eq.8, where only the optimal action needs to be considered.

$$f_a = \operatorname*{argmin}_f \sum_{s\in D_p} I(f(s) \neq \operatorname*{argmax}_a \Psi_t(s,a)). \tag{8}$$

By the new formulation, we only need a training data set that is of the same size as the probing instances.

3.3 Replacing the Potential Function

After obtaining the approximation model, it will be used to replace the potential function. Then the future policy gradient ascent step will continue on the base of the approximation model. It is possible that the approximation model output a value that is not in the same scale as the original potential function. This is particular true for the mimicking target is the policy action instead of the state-action value function.



(a) Mountain Car (b) Acrobot

Figure 1: Illustrations of the Mountain Car and Acrobot domains

Noticing that even for the classification model f_a, there are easy ways to output its internal preference of choosing an arbitrary action. We thus denote $p(a \mid s, f)$ as the probability or normalized preference of choosing action a by the model f, and we then replace the potential function by the value of the probability/preference multiplied by a constant C, i.e.,

$$\Psi_t(s,a) = C \cdot p(a \mid s, f), \tag{9}$$

where the constant is used to keep the new potential function outputs in a similar scale with the original potential function.

When the napping ends with the potential function replaced by either f_v or f_a, the number of models is reduced from $O(t)$ to $O(1)$. The following training and predicting invoke less calculations of models, thus can be much faster than the original method.

4. EXPERIMENTS

We empirically verify the napping mechanism by investigating three questions sequentially:

1. On efficiency: does the napping mechanism effectively reduce the time cost of the original NPPG?
2. On efficacy: does the napping mechanism affect the convergence performance of the original NPPG?
3. How do the parameters, i.e., the napping interval nap as well as the sampling size K, effect the performance?

4.1 Domains

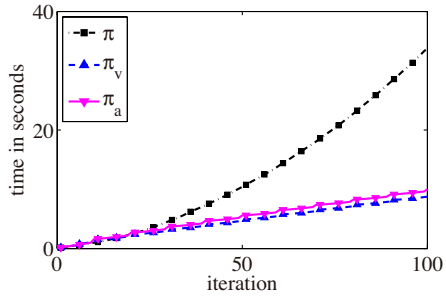
The experiments are conducted on three well known continuous domains, Corridor World [18], Mountain Car and Acrobot [28]. For all the three domains, the agent starts from an random initial state and receives a reward 0 after reaching the goal and -1 otherwise.

4.1.1 Corridor World

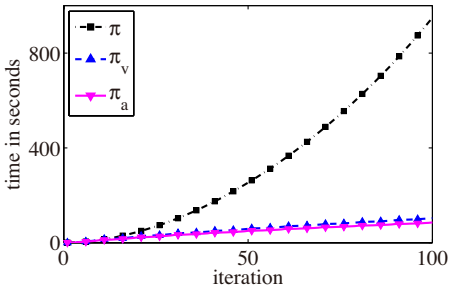
For this domain we consider to navigate an agent from any random position $x_0 \in [4, 6]$ in a one dimensional corridor $[0, 10]$ to one of the exists at both ends (0 and 10). At each time, the agent can go either *left* ($a = -1$) or *right* (a=1), added by an gaussian noise with 0 mean and variance of σ, i.e., $x_t = x_{t-1} + a \cdot L + \mathcal{N}(0, \sigma^2)$. In our setting, L and σ are both set to 0.2

4.1.2 Mountain Car

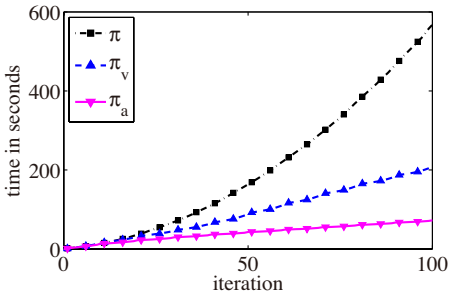
In Mountain Car task, an under-powered car must drive up a steep hill as show in Figure 1 (a). The states of the agent are two continuous variables, the horizontal position x and the velocity $\dot{x}$, which are restricted to the ranges $[-1.2, 0.6]$ and $[-0.07, 0.07]$ respectively. At each time, the agent needs to select one of three actions: driving left ($a = -1$), driving right ($a = 1$) and not to use the engine at all ($a = 0$). The velocity is updated by $\dot{x}_t =$



(a) Corridor World

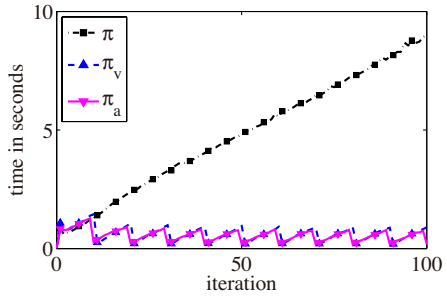


(b) Mountain Car

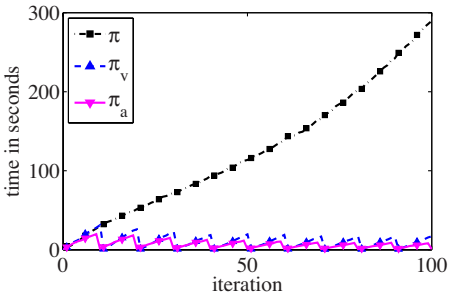


(c) Acrobot

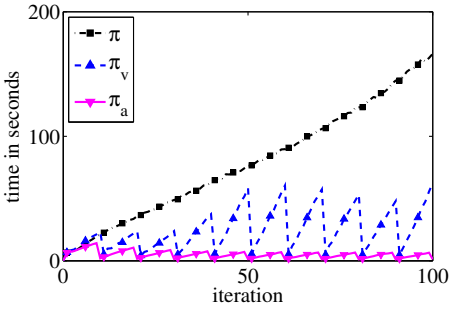
Figure 2: Comparisons of total training time.



(a) Corridor World



(b) Mountain Car



(c) Acrobot

Figure 3: Comparisons of prediction time.

$\dot{x}_{t-1} + 0.001a - 0.0025\cos(3x_{t-1})$, where the last term is due to the effect of gravity. The position is then added by $\dot{x}_t$. The goal of the agent is to reach the right mountain top, i.e., $x > 0.5$.

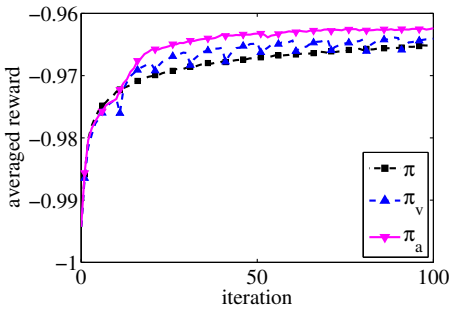
4.1.3 Acrobot

Acrobot is a two-link, underactuated robot roughly analogous to a gymnast swinging on a high bar (Figure 1 (b)). The first joint cannot exert torque, but the second joint can. The system has four continuous state variables: two joint positions θ_1, θ_2 and two joint velocities $\dot{\theta}_1$, $\dot{\theta}_2$ and three actions correspond to torque to the joint between the first and second link of $-1, 0, 1$ respectively. The detail of the dynamics can be found in [28]. The goal of the agent is to let the tip above the goal line.

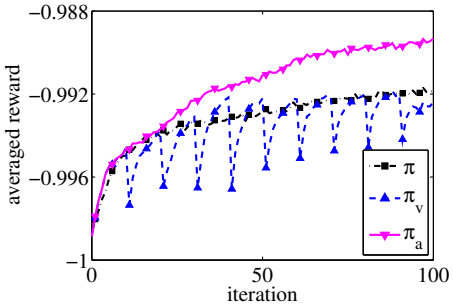
4.2 Experiment Configurations

There are only two parameters in the NPPG method, the step size η_t and the ϵ-greedy probability. For all the experiments, ϵ is 0.1 and $\eta_t = \alpha/\sqrt{t}$. α are 0.1, 0.1, 0.008 for the three domains respectively. We find that the performance of our napping method is not very sensitive to C and a simple setting (C=1) is good enough for almost all cases except that a smaller value ($C = 0.05$) is more proper for the napping method with action mimicking approximation on the Corridor World domain. We utilize the later parts of every trajectory as an approximation of the stationary distribution. The regression learner employed in the NPPG method as well as in our napping method with state-action value mimicking approximation is a bagging [7] with 10 regression decision trees [9]. The classifier used in our napping method with action mimicking approximation is a random forest [8] with 10 random trees. We use the implementation of above models in WEKA [38] in our experiments. The codes of our implementation can be found from `http://cs.nju.edu.cn/yuy`.

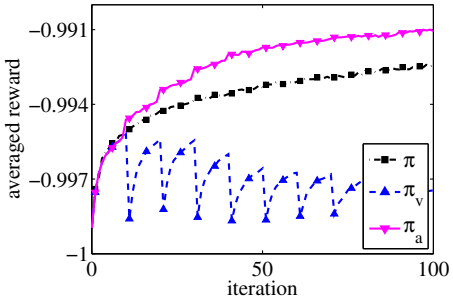
We denote the policy learned by original NPPG method, the napping version with approximation by mimicking the state-action value, and the napping version with approximation by mimicking the action as π, π_v and π_a respectively. We apply the three method on the above three domains for 100 iterations. For the Corridor World, 10 episodes are collected each iteration, while for the rest two the number is 20. We test the policies obtained at each iterations on 500 trials, and the test is independent of the training process.



(a) Corridor World



(b) Mountain Car



(c) Acrobot

Figure 4: Averaged reward of the π, π_v and π_a at each iteration

4.3 Results

To answer the first question, we compare the total training time as well as the prediction time of π, π_v and π_a in Figure 2 and Figure 3, respectively. The napping policies are all with a napping interval 10 and sufficient sampling examples ($K = 5000$) for approximation. Figure 2 shows the accumulated training time for each iteration of the three policies on the three domains. It is clear that, even with an extra time for model approximation every 10 iterations, the napping policies use much less time for training than the original policy. Figure 3 (b) shows prediction time of the policies obtained at each iteration on 500 trials of task. It can be observed that, the prediction time of napping policies are limited to some constant even with more iterations as expected, since the complexity of the policy drops after every napping, while that of the original one grows with a near-linear trend.

To answer the second question, we show the averaged reward of the π, π_v and π_a at each iteration in Figure 4. It is can be shown that, after every nap, the performance of π_v degrades significantly, and then grows at a faster rate than that of π at the same point. For the Corridor World, π_v gets comparable performance with π; while for the other two domains, its performance become unstable, i.e., the performance degradation caused by approximation cannot get enough compensations brought by faster growing rate later, so the overall performance of π_v is below π. At the same time, the policies napped by action-mimicking based approximation keep competitive performances for all the domains, and with the increasing number of iterations, π_a even outperforms the original policy π. An interesting result is that, different from π_v, π_a never degrades the performance when the nap happens. Moreover, it even always gets a small improvement after every nap for Acrobot problem. It's worth noting that, for π_a at the 100th iteration which is represented by a single random forest with 10 trees (it takes a nap at every 10 iteration), outperforms the original π with 100 Baggings of 10 trees. We also test the performance in a stochastic version of the Mountain Car domain, as suggested by a reviewer, where the velocity is updated by $\dot{x}_t = \dot{x}_{t-1} + 0.001(a + \varepsilon) - 0.0025\cos(3x_{t-1})$ with ε being the noise signal uniformly sampled from $[-0.05, 0.05]$. In the stochastic environment, the absolute performance of all policies are higher than in the deterministic environment, while the relative performance among the policies keeps just similar: the curve of π_a is above that of π, the curve of π_a is below that of π, while the performance of π_a drops after every nap.

These results show that, napping with action-mimicking based approximation is better and more robust than that of mimicking state-action value. To further understand this phenomenon, we test the two approximated models for their disagreement with the original policy. For the comparison, we use two state-of-the-art regression methods, i.e., bagging with regression tree (BRT) and Gradient Boosting, and three stat-of-the-art classification methods, i.e., C4.5, random forest (RF) and AdaBoost to guarantee that the result will not be biased by the power of different learning models. The disagreement is defined as the averaged difference of actions selected by approximated models and the original policy on another set of independent instance, which in fact is equivalent to the classification error of the approximated models on a test data labeled by original policy. The Mean Disagreement Error (MDE) of all the models are shown in Table 1. The Relative Mean Square Error (RMSE) for regression models are also listed in the table. It can be seen that, even though the regression models do well for the regression job, i.e., the relative mean square errors are very small, but the decisions made by the regression model are quite different from the original policy; on the other hand, the approximation of mimicking action perform well in keeping consistent with the original policy. Moreover, we compare the policies before and after the napping with the optimal policy for their disagreement on action selection, for the two approximation approaches. The results are listed in Table 2. It shows that, compared with the approximation of mimicking actions, the policy after napping by mimicking values get higher disagreement not only with the original policy, but also with the optimal policy.

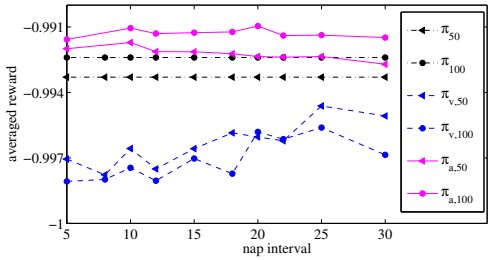
Finally, we study how the performance changes with different napping settings, i.e., the napping interval nap as well as the sampling size K. Average reward of policies obtained at 50th/100th iteration of π, π_v and π_a with different parameters on Acrobot are illustrated in Figure 5, similar results are obtain on other domains and they are omitted due to page limitation. π_v always perform worse than π under any situations, while π_a only gets worse performance than π when the sampling size is too small. Besides, π_a is not very sensitive to the napping interval as in Figure 5 (a). Figure 5 (b) shows that, with more examples for approximation, the performance of π_a always gets better, but one do not need too many examples since the policies with 500 examples for approximation seem to be almost as good as the original policy in this case.

Table 1: Comparison the approximation quality to the original policy (on Mountain Car)

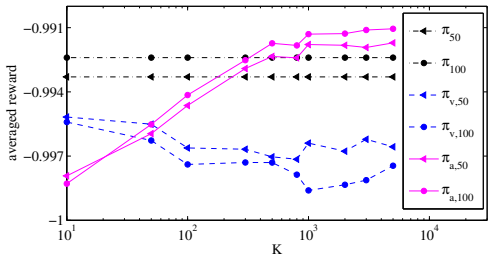
time	mimic state-action value				mimic action		
	BRT		GB		C4.5	RF	AdaBoost
	RMSE	MDE	RMSE	MDE	MDE	MDE	MDE
1st nap	7.0e-11±1.1e-11	0.52±0.02	1.1e-10±3.0e-11	0.60±0.10	0.35±0.02	0.29±0.02	0.30±0.02
3rd nap	3.5e-11±7.3e-12	0.48±0.04	6.9e-11±1.7e-11	0.53±0.05	0.30±0.02	0.23±0.02	0.23±0.02
5th nap	4.1e-11±9.9e-12	0.23±0.02	8.4e-11±1.7e-11	0.33±0.20	0.19±0.02	0.15±0.02	0.15±0.01
7th nap	2.9e-11±6.9e-12	0.43±0.03	5.9e-11±1.5e-11	0.41±0.05	0.15±0.02	0.12±0.02	0.12±0.02
9th nap	3.1e-11±4.7e-12	0.39±0.02	5.6e-11±1.4e-11	0.43±0.10	0.14±0.01	0.11±0.01	0.11±0.01

Table 2: Disagreement with the (near) optimal policy (on Mountain Car).

time	mimic state-action value			mimic action			
	before nap	after nap		before nap	after nap		
		BRT	GB		C4.5	RF	AdaBoost
1st nap	0.549±0.002	0.579±0.014	0.615±0.017	0.422±0.002	0.381±0.009	0.421±0.003	0.410±0.007
3rd nap	0.681±0.002	0.679±0.020	0.685±0.023	0.273±0.002	0.263±0.003	0.273±0.002	0.273±0.002
5th nap	0.534±0.004	0.726±0.009	0.720±0.023	0.188±0.001	0.184±0.006	0.188±0.001	0.188±0.001
7th nap	0.534±0.003	0.692±0.016	0.658±0.065	0.150±0.001	0.147±0.002	0.149±0.001	0.150±0.001
9th nap	0.635±0.001	0.563±0.024	0.531±0.014	0.100±0.001	0.098±0.002	0.099±0.001	0.100±0.001



(a) With nap



(b) With K

Figure 5: Performance of napping policies with different parameters on Acrobot

5. CONCLUSION

In this paper, to make the functional representation more practically usable in reinforcement learning, we proposed the napping mechanism to reduce its complexity of both time and space. By this mechanism, a simple function is trained to approximate the learned function periodically along with the learning process. We implemented the napping in the NPPG method, which is a policy gradient approach. We incorporated reservoir sampling in the implementation and studied two ways for the approximation for napping: mimicking the state-action values and mimicking the policy actions. Experiments on three well-studied domains verified the efficiency as well as the efficacy of the napping mechanism. Moreover, we found that the approximation mimicking actions are more suitable in our cases. The work verified the possibility that we can keep the policies learned by functional policy gradient methods with a constant number of complexity, and at the same time improving the convergence performance. We also noticed that, in partially observable Markov decision process (POMDP) literatures, searching an optimal policy in a bounded space can lead to a better performance [21, 25, 17]. In the future it is interesting to apply the NPPG with napping approach in POMDP domains. Besides the least square regression and the classification loss, we will study more losses for approximation, such as the ranking loss [33] that could focus on correctly rank the actions for a state.

Acknowledgements

This work was supported by the National Science Foundation of China (61375061, 61321491, 61223003) and the Jiangsu Science Foundation (BK2012303).

6. REFERENCES

[1] D. A. Aberdeen. *Policy-gradient algorithms for partially observable Markov decision processes*. PhD thesis, Australian National University, 2003.

[2] J. A. Bagnell. *Learning decisions: Robustness, uncertainty, and approximation*. PhD thesis, Oregon State University, 2004.

[3] L. Baird. Residual algorithms: Reinforcement learning with function approximation. In *Proceedings of the 12th International Conference on Machine Learning (ICML)*, pages 30–37, Tahoe City, CA, 1995.

[4] P. L. Bartlett and J. Baxter. Infinite-horizon policy-gradient estimation. *Journal of Artificial Intelligence Research*, 15:319–350, 2001.

[5] R. Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ, 1957.

[6] J. Boyan and A. W. Moore. Generalization in reinforcement learning: Safely approximating the value function. In D. S. Touretzky, M. Mozer, and M. E. Hasselmo, editors, *Advances in Neural Information Processing Systems 8 (NIPS)*, pages 369–376. MIT Press, 1996.

[7] L. Breiman. Bagging predictors. *Machine learning*, 24(2):123–140, 1996.

[8] L. Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[9] L. Breiman, J. Friedman, and C. J. Stone. *Classification and Regression Trees*. Chapman and Hall/CRC, 1984.

[10] L. Busoniu, R. Babuska, B. De Schutter, and D. Ernst. *Reinforcement learning and dynamic programming using function approximators*. CRC Press, Boca Raton, FL, 2010.

[11] S. S. D. Ormoneit. Kernel-based reinforcement learning. *Machine Learning*, 49(2-3):161–178, 2002.

[12] T. G. Dietterich, A. Ashenfelter, and Y. Bulatov. Training conditional random fields via gradient tree boosting. In *Proceedings of the 21st International Conference on Machine Learning (ICML)*, Banff, Canada, 2004.

[13] P. Domingos. Knowledge discovery via multiple models. *Intelligent Data Analysis*, 2(1-4):187–202, 1998.

[14] K. Doya. Reinforcement learning in continuous time and space. *Neural computation*, 12(1):219–245, 2000.

[15] Y. Freund and R. E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences*, 55(1):119–139, 1997.

[16] J. H. Friedman. Greedy function approximation: A gradient boosting machine. *Annals of Statistics*, 29(5):1189–1232, 2001.

[17] M. Grześ, P. Poupart, and J. Hoey. Isomorph-free branch and bound search for finite state controllers. In *Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI)*, pages 2282–2290, Beijing, China, 2013.

[18] K. Kersting and K. Driessens. Non-parametric policy gradients: A unified treatment of propositional and relational domains. In *Proceedings of the 25th International Conference on Machine Learning (ICML)*, pages 456–463, Helsinki, Finland, 2008.

[19] J. Kober and J. Peters. Policy search for motor primitives in robotics. *Machine Learning*, 84(1):171–203, 2011.

[20] D. D. Margineantu and T. G. Dietterich. Pruning adaptive boosting. In *Proceedings of the 14th International Conference on Machine Learning (ICML)*, pages 211–218, Nashville, TN, 1997.

[21] N. Meuleau, K.-E. Kim, L. P. Kaelbling, and A. R. Cassandra. Solving POMDPs by searching the space of finite policies. In *Proceedings of the 15th Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 417–426, Stockholm, Sweden, 1999.

[22] N. Mitsunaga, C. Smith, T. Kanda, H. Ishiguro, and N. Hagita. Robot behavior adaptation for human-robot interaction based on policy gradient reinforcement learning. In *Proceedings of IEEE International Conference on Intelligent Robots and Systems (IROS)*, pages 218–225, Edmonton, Canada, 2005.

[23] J. Peters. Policy gradient methods. *Scholarpedia*, 5(10):3698, 2010.

[24] J. Peters and S. Schaal. Natural actor-critic. *Neurocomputing*, 71(7):1180–1190, 2008.

[25] P. Poupart and C. Boutilier. Bounded finite state controllers. In S. Thrun, L. K. Saul, and B. Schölkopf, editors, *Advances in Neural Information Processing Systems 16 (NIPS)*. MIT Press, 2004.

[26] G. Rummery and M. Niranjan. On-line Q-learning using connectionist systems. Technical report, Engineering Department, University of Cambridge, 1994.

[27] W. D. Smart and L. P. Kaelbling. Practical reinforcement learning in continuous spaces. In *Proceedings of the 17th International Conference on Machine Learning (ICML)*, pages 903–910, Stanford, CA, 2000.

[28] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*. MIT Press, Cambridge, MA, 1998.

[29] R. S. Sutton, H. R. Maei, D. Precup, S. Bhatnagar, D. Silver, C. Szepesvári, and E. Wiewiora. Fast gradient-descent methods for temporal-difference learning with linear function approximation. In *Proceedings of the 26th International Conference on Machine Learning (ICML)*, pages 993–1000, Montreal, Canada, 2009.

[30] R. S. Sutton, D. McAllester, S. Singh, and Y. Mansour. Policy gradient methods for reinforcement learning with function approximation. In S. A. Solla, T. K. Leen, and K.-R. Müller, editors, *Advances in Neural Information Processing Systems 12 (NIPS)*, pages 1057–1063. MIT Press, 2000.

[31] C. Szepesvári and M. L. Littman. A unified analysis of value-function-based reinforcement-learning algorithms. *Neural computation*, 11(8):2017–2060, 1999.

[32] G. Tesauro. TD-Gammon, a self-teaching backgammon program achieves master-level play. *Neural Computation*, 6:215–219, 1994.

[33] A. Trotman. Learning to rank. *Information Retrieval*, 8(3):359–381, 2005.

[34] A. Van Assche and H. Blockeel. Seeing the forest through the trees: Learning a comprehensible model from an ensemble. In *Proceedings of the 18th European Conference on Machine Learning (ECML)*, pages 418–429, Warsaw, Poland, 2007.

[35] J. S. Vitter. Random sampling with a reservoir. *ACM Transactions on Mathematical Software*, 11(1):37–57, 1985.

[36] C. H. Watkins. *Learning from Delayed Rewards*. PhD thesis, King's College, University of Cambridge, 1989.

[37] R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3):229–256, 1992.

[38] I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, 2005.

[39] Z.-H. Zhou and Y. Jiang. NeC4.5: Neural ensemble based C4.5. *IEEE Transactions on Knowledge and Data Engineering*, 16(6):770–773, 2004.

[40] Z.-H. Zhou, J. Wu, and W. Tang. Ensembling neural networks: Many could be better than all. *Artificial Intelligence*, 137(1-2):239–263, 2002.