Policy iteration based on a learned transition model

Vivek Ramavajjala and Charles Elkan

Department of Computer Science & Engineering
University of California, San Diego
La Jolla, CA 92092

Abstract. This paper investigates a reinforcement learning method that combines learning a model of the environment with least-squares policy iteration (LSPI). The LSPI algorithm learns a linear approximation of the optimal state-action value function; the idea studied here is to let this value function depend on a learned estimate of the expected next state instead of directly on the current state and action. This approach makes it easier to define useful basis functions, and hence to learn a useful linear approximation of the value function. Experiments show that the new algorithm, called NSPI for next-state policy iteration, performs well on two standard benchmarks, the well-known mountain car and inverted pendulum swing-up tasks. More importantly, the NSPI algorithm performs well, and better than a specialized recent method, on a resource management task known as the day-ahead wind commitment problem. This latter task has action and state spaces that are high-dimensional and continuous.

1 Introduction

> "...the state of a system is often summarized by certain features or basis functions that capture the state's salient properties. The selection of suitable features is often the *condicio sine qua non* for practical success. The choice of features is almost always influenced by sound engineering understanding of the problem domain, but automating this process would be a major step ahead." (Tsitsiklis, 2010) [17].

This paper takes a step forward towards the goal of automating the process of choosing useful basis functions in reinforcement learning.

The objective of a reinforcement learning algorithm is to acquire a policy for choosing actions that can control an agent to desired goal states. A value function is a function that estimates the long-term reward, as opposed to the immediate reward, of a given state, or of a given state combined with a given action. A Q function is a value function that maps each state-action pair to a real number that measures the long-term utility of taking that action in that state. Approaches that learn value functions approximately, and in particular methods that learn Q functions approximately, have been used to solve reinforcement learning problems successfully. While other functional forms have also been used, linear approximations of Q functions are popular because of convergence guarantees, ease of implementation, and low computational complexity [8]. A linear

approximator represents the value of a state-action pair as a weighted sum

$$Q(s,a) = \sum_{j=1}^{m} \phi_j(s,a) v_j = \phi(s,a) \cdot v$$

of m predetermined basis functions where v is a real-valued vector of length m. Several approaches to learn v, notably approximate policy iteration [8], have been investigated intensively but the choice of the basis functions themselves has been studied less. In this paper we suggest a process to define useful basis functions, and we present empirical results that demonstrate the effectiveness of the approach.

The rest of this paper first reviews Markov decision processes (Section 2), and then describes the proposed process for defining useful basis functions (Section 3). The specific algorithm that we suggest, called NSPI, is presented in Section 4, which also discusses related research briefly. Then Section 5 presents experimental results on standard benchmarks, and Section 6 shows the effectiveness of the approach on a high-dimensional application.

2 Markov decision processes

The context of our work is reinforcement learning (RL) from historical data, which is often called batch RL. Although the majority of research in recent decades has concerned RL with interactive exploration of the environment, non-interactive RL tasks were in fact the applications that motivated the original invention of Markov decision processes (MDPs) [6].

An MDP is the formalization of the scenario underlying an RL task. Precisely, an MDP is a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$ where $\mathcal{S}$ is the state space, $\mathcal{A}$ is the action space, $\mathcal{P}$ is the transition model with $\mathcal{P}(s,a,s')$ being the probability of making a transition from s to s' on taking action a, $\mathcal{R}$ is the reward model with $\mathcal{R}(s,a,s')$ being the immediate reward associated with transitioning from s to s' on taking action a, and $\gamma \leq 1$ is the discount factor. We assume that the MDP has an infinite horizon and that future rewards are discounted exponentially with the discount factor γ. For problems with a goal or sink state, the discount factor may be equal to 1, while in problems without any goal or sink state, the discount factor must be strictly less than one. The expected reward for taking an action a in a state s is

$$R(s,a) = \int_{s'} \mathcal{R}(s,a,s') \mathcal{P}(s,a,s') ds'.$$

A stationary policy is a mapping $\pi : \mathcal{S} \rightarrow \Omega(\mathcal{A})$ where $\Omega(\mathcal{A})$ is the space of probability distributions over the action space; $\pi(a;s)$ is the probability of choosing action a in state s. A deterministic stationary policy is a mapping $\pi : \mathcal{S} \rightarrow \mathcal{A}$, with $\pi(s)$ being the action the agent takes in state s. For a stationary deterministic policy π, the Q function $Q^\pi(s,a)$ that represents the utility of taking action a in state s can be expressed as

$$Q^\pi(s,a) = R(s,a) + \gamma \int_{s'} Q^\pi(s', \pi(s')) \mathcal{P}(s,a,s') ds'.$$

Given any Q function $Q(s,a)$, the corresponding policy is $\pi(s) = \operatorname{argmax}_a Q(s,a)$.

All problems considered in this paper are Markov decision processes where the state, action, reward and next state can be observed in every transition. The state and action spaces $\mathcal{S}$ and $\mathcal{A}$ are real-valued and may be high-dimensional. The transition model $\mathcal{P}$ and the reward model $\mathcal{R}$ are assumed to be unknown.

3 Defining useful basis functions

An optimal policy, and hence an optimal Q function, specifies an action that moves the agent from the current state to the best possible next state (which may be the same as the current state). The long-term value of a state in an MDP does not depend on the state from which it was reached, or on the action taken to reach it. Thus, the next state (which is the net effect of the action taken in the current state) may be more informative than the current state and the action taken, as the argument of a Q function. Given that an approximate Q function is a weighted combination of basis functions, this observation leads to our primary intuition for defining useful basis functions: these should take as sole input an estimate of the next state that results from the action taken in the current state.

Concretely, the proposal is to write

$$Q(s,a) = Q(\hat{s}) = \sum_{j=1}^{m} \phi_j(\hat{s}) v_j$$

where $\hat{s} = f(s,a)$ for some function f is an estimate of the next state reached from the current state s by taking action a. In general, the next state is not determined deterministically by s and a, so it is more precise to say that $\hat{s}$ is an estimate of the expected next state. In this paper we assume that states are real-valued vectors, so expectations are well-defined, subject to some minor technical conditions. The transition model that describes the next state is the component $\mathcal{P}$ of the Markov decision process. The essence of reinforcement learning is that $\mathcal{P}$ is not known, but an approximation of it can be estimated from sampled data.

Having the next state as input leads to the secondary intuition for defining useful basis functions: these should measure aspects of the next state that are correlated with its long-term value. This intuition is consistent with most previous work, so arguably it is not novel. However, it seems easier to put into practice when it is applied to the estimated expected next state, rather than to the current state and action. The idea can be made more precise by considering two different varieties of task (again this distinction is not novel). The first variety consists of tasks where there are specific terminal states. For these tasks, each basis function should describe a small neighborhood of states. Then, the trained coefficient of each basis function can indicate the proximity of the neighborhood to good and/or bad ending states. The second variety consists of tasks where there are no defined terminal states, but each state generates a varying reward. For these tasks, each basis function should describe a relatively separable aspect of the following state. Then, the trained coefficients of the basis functions can collectively indicate the goodness of the expected next state. As a special case, basis functions

can be components of the vector $\hat{s}$ that represents the expected next state, plus other functions that can be computed from this vector.

The first variety of tasks includes the well-known mountain car and inverted pendulum benchmarks, where a small region of the state space is the goal region. These two tasks are investigated experimentally in Section 5 below, using the primary and secondary intuitions just described. The second variety of tasks includes resource management scenarios, where stocks and flows must be controlled in order to maximize profits and minimize costs [13]. In these scenarios, aspects of the estimated state can be the levels of various resources, their prices, environmental conditions such as weather, and so on. A task of this nature is solved in detail in Section 6.

4 The NSPI algorithm

This section describes a concrete algorithm that puts into practice the ideas of the previous section. As mentioned, the primary idea is that a Q function should be represented as $Q(s, a) = Q(\hat{s})$ where $\hat{s}$ is an estimate of the expectation of the next state. In our approach, $\hat{s}$ is a linear function of s and a. This function is used only to predict the immediate next state and not an extended path, so typically it only needs to predict transitions to states in a small neighborhood of the current state. Even domains exhibiting a high degree of nonlinearity, such as the inverted pendulum domain described below, have local transitions that are approximately linear functions of a representation of the current state and action. Of course, local linearity does not imply global linearity, and in some domains the true transition function is discontinuous.

For a batch reinforcement learning task, the training data consist of n quadruples of the form (s_i, a_i, r_i, s'_i) where s_i is a state, a_i is the action taken in that state, and r_i and s'_i are the reward and next state that were observed to ensue. The linear estimated transition model is simply the matrix T that is the least squares solution of the overdetermined system of n linear equations each one of the form

$$[s_i \ a_i]T = s'_i.$$

Then, for any state s and action a, the approximate Q function is

$$Q(s, a) = Q(\hat{s}) = \sum_{j=1}^{m} \phi_j([s \ a]T)v_j$$

where the weights v_j are learned in a second stage, separately from learning T.

When appropriate, the linear transition model can depend on a nonlinear representation of the state. For example, if one component of the state is a measured angle, then having the sine and cosine of the angle in the state representation can give additional information. If the re-representations of the state and action spaces are denoted by the functions f_s and f_a respectively, then the transition matrix T is the least squares solution of the n linear equations

$$[f_s(s_i) \ f_a(a_i)]T = s'_i$$

Algorithm 1 NSPI (next-state policy iteration)

```
// Input: training samples D = {(s_i, a_i, r_i, s'_i)}_{i=1}^n
// φ: basis functions φ_1 to φ_m
// γ: discount factor
// ε: stopping criterion
// Output: weight vector v' representing the learned policy

Stage 1: Solve [f_s(s_i) f_a(a_i)]T = s'_i |_{i=1}^n for T
Stage 2: // LSPI
v' ← 0
repeat
    v ← v'
    A ← 0                         // m × m matrix
    b ← 0                         // m × 1 column vector
    for each (s, a, r, s') ∈ D do
        s_next = [f_s(s) f_a(a)]T              // estimate the next state for s
        a* = argmax_{a'} φ([f_s(s') f_a(a')]T) · v
        s'_next = [f_s(s') f_a(a*)]T           // estimate the next state for s'
        A ← A + φ(s_next)(φ(s_next) − γφ(s'_next))^T
        b ← b + φ(s_next)r
    end for
    Solve Av' = b for v'
until ||v − v'|| < ε
```

and the Q function for a given state s and action a is $Q(s,a) = \phi(\hat{s}) \cdot v$ where $\hat{s} = [f_s(s)\ f_a(a)]T$.

After a linear transition model T is trained, the second stage of our approach is to learn a Q function using the next-state basis function representation $\phi(\hat{s})$. This paper uses the least squares policy iteration (LSPI) method for the second stage [8]. LSPI has the advantage of not having to select parameters such as the learning rate or the number of steps to iterate over. The full proposed algorithm using next-state basis functions is shown in Algorithm 1.

Regularization can be used for learning both the transition matrix T and the weights v. For the experiments described below, regularization is not needed for learning the transition matrix, because the number n of training samples is much larger than the number of features, which is the length of the concatenated vectors $[f_s(s)\ f_a(a)]$. For learning the weights v, regularization is used by adding a scaled identity matrix λI to the matrix A, where λ is a small positive value. Empirically, regularization in learning v helps stabilize the estimates of the weights, and reduces the number of steps required to reach a stable estimate.

A practical concern with Q functions is the argmax operation used to find the optimal action. For small discrete action spaces, as in Section 5, finding an optimal action is easy, but searching in a continuous action space requires an optimization algorithm of some sort. This can be computationally expensive, especially if the space of feasible

actions is constrained. Choosing the optimal action given s can be formulated as

$$a^* = \underset{a}{\operatorname{argmax}} \sum_{j=1}^{m} \phi_j([f_s(s)\ f_a(a)]T)v_j \text{ such that } x \leq Ca \leq y$$

where the matrix C and the lower and upper bound vectors x and y specify constraints on allowed action vectors. If the basis functions ϕ_j and the action representation function f_a are linear, then linear programming can find the optimal a quickly even with constraints. Otherwise, an alternative optimization approach such as an interior point method can be used, as is the case in Section 6 below.

Algorithm 1 is novel as far as we know, but of course it is not unprecedented. The fundamental aspect that makes reinforcement learning be a process of learning from data, and different from solving the MDP directly, is that the transition model $\mathcal{P}$ is unknown. RL methods are called model-based if they learn $\mathcal{P}$ explicitly in some way, and model-free if they do not [12]. (Some model-based algorithms learn to predict the immediate reward as well as the next state.) Standard least-squares policy iteration is a model-free method, while NSPI is model-based. Given a limited quantity of training samples $\{(s_i, a_i, r_i, s'_i)\}_{i=1}^{n}$, learning both a model and a value function can maximize the amount of useful information extracted from the training data. One reason is that in many domains the transition model is close to linear, even though the optimal value function is not. Another reason is that in many domains the long-term value is a simpler function of the following state than it is directly of the current state s and an action a.

There are several RL methods that are similar in some ways to Algorithm 1. The earliest method to learn a Q function and a transition model at the same time may have been the Dyna-Q method [15]. A more recent paper that combines model-based and model-free learning is [7]. Its algorithm called AMBI is interactive, which is likely a major reason why it needs about 100 episodes to learn consistently a good policy for the mountain car task, compared to about 40 episodes for the NSPI algorithm. Other related work is [9], whose algorithm has a first stage that uses information collected about the state space to define so-called proto-value functions as basis functions, and a second stage that uses LSPI to determine a good policy based on these. Another line of research whose aim is similar to the aim here investigates methods to adapt the parameters of basis functions of a known form [11].

5 Benchmark results

This section describes experiments applying the proposed NSPI algorithm to the mountain car problem and to the inverted pendulum swing-up problem, which are two standard benchmarks for reinforcement learning. The following section then describes applying the method to a large-scale inventory management problem that concerns managing a wind energy farm to maximize profits from sale of electricity. The first two problems require the policy to control the agent to a goal state, while the last requires the agent to maximize long-term reward. For each task, we briefly describe the problem domain, define the basis functions, explain procedures for training and testing policies, and compare the performance of learned policies with existing results.

5.1 Mountain car

The mountain car task requires controlling a car up a hill when its own power is insufficient to climb the hill. The car must back up on an opposite hill to gain momentum, and use that momentum to accelerate to the goal position [16]. The state is a tuple (x, v) where $x \in [-1.2, 0.6]$ is the position of the car and $v \in [-0.07, 0.07]$ is the velocity of the car. The goal states are those with $x \geq 0.5$. There is an inelastic wall at the left end of the domain, i.e., if the car reaches $x = -1.2$, it is stopped and given a velocity of 0. In each state, the action is either to accelerate forward, cruise, or accelerate backwards, i.e., $a \in \{-0.001, 0, 0.001\}$. The effect of gravity at a position x is given by $-0.0025 \cos(3x)$. After each transition, the position and velocity are bounded to stay within the specified domain. Thus, the transition from a state (x_t, v_t) to the next state for a given action a is

$$
\begin{aligned}
x_{t+1} &= \max\{\min\{x_t + v_t, 0.6\}, -1.2\} \\
v_{t+1} &= \max\{\min\{v_t + a - 0.0025 \cos(3x_t), 0.07\}, -0.07\}.
\end{aligned}
$$

The reward at each step is $r_t = 0$ if $x_t \geq 0.5$ and $r_t = -1$ otherwise. Given n training samples $\{s_i, a_i, r_i, s'_i\}_{i=1}^{n}$, the transition matrix T is the least squares solution of the overdetermined linear system of n equations

$$[s_i \; a_i]T = s'_i.$$

Directly following the intuition suggested in Section 3 above, the basis functions are 25 Gaussian radial functions whose centers are distributed evenly over the state space. Approximately, the learned weight for each basis function indicates how good that neighborhood of the state space is. Each Gaussian has a fixed width proportionate to the distance between two neighboring centers. The basis functions are thus

$$\phi_m(\hat{s}) = \exp\left(-\frac{1}{2}(\hat{s} - c_m)\Sigma^{-1}(\hat{s} - c_m)^T\right)$$

with $c_m \in \{-1.2, -1, -0.8, \ldots, 0.6\} \times \{-0.07, -0.054, \ldots, 0.054, 0.07\}$ and

$$\Sigma = \begin{bmatrix} 0.036 & 0 \\ 0 & 0.0006 \end{bmatrix}.$$

Samples for training were collected using random trials, i.e., starting the car with a randomly chosen position and velocity and following a policy that selects actions at random. The training trials were restricted to at most 60 steps. For testing, each policy was given 100 randomly chosen starting positions and each test trial was followed for at most 500 steps. A policy that controlled the car to the goal state for all 100 test trials was considered to be successful. This experiment was repeated 10 times for each training set size. The performance for each training set size is reported as the average over the 10 experiments. All policies were learned using $\gamma = 0.95$ with strength of regularization $\lambda = 0.4$. The strength of regularization was chosen empirically to ensure that the LSPI stage of the NSPI algorithm arrives at a stable estimate of the weights.

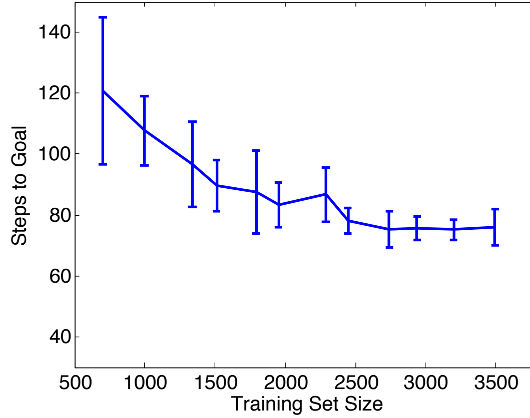


Fig. 1: Results for the mountain car benchmark: dependence of steps-to-goal on the training set size.

The variation of the performance with the training set size is shown in Fig. 1. The first successful policy was learned with a training set having only 300 samples (roughly 5 random trials), though the policy was not optimal. With training set sizes of 2400 samples or more, the NSPI algorithm consistently learns a policy that controls the car to the goal state in at most 76 steps averaged across all test starting states, with a standard deviation of 3 to 6 steps. The best learned policy controls the car to the goal in an average of 68 steps, learned on a training set of 2200 samples. Some of the average steps-to-goal reported previously are 104 [10], 70 to 80 [14], and 63 [18]. In light of these comparisons, we can conclude that a close-to-optimal policy is learned consistently with about 40 training trials.

5.2 Inverted pendulum

The inverted pendulum task involves a pendulum attached to a cart, where the agent can only apply horizontal forces to the cart. The goal is to swing the pendulum from its stable equilibrium position (the pendulum pointing downwards) to the unstable equilibrium position where the pendulum points upwards. A state is a tuple $(\theta, \dot{\theta})$ where θ is the angle measured from the unstable equilibrium position and $\dot{\theta}$ is the angular velocity. Both dimensions are unbounded. The available actions are a positive force $+10N$, a negative force $-10N$, or no force to be applied to the cart.

The dynamics for the inverted pendulum problem are given by [19] as

$$\ddot{\theta} = \frac{g\sin(\theta) - \alpha ml(\dot{\theta})^2 - \alpha\cos(\theta)u}{4l/3 - \alpha ml\cos^2(\theta)}$$

where the action $u \in \{-10, 0, 10\}$, the constant $\alpha = 1/(M+m)$, and the one-step transition function is $\theta \leftarrow \theta + \tau\dot{\theta}$ and $\dot{\theta} \leftarrow \dot{\theta} + \tau\ddot{\theta}$. Here, $M = 1.0, m = 0.1, l = 0.5$

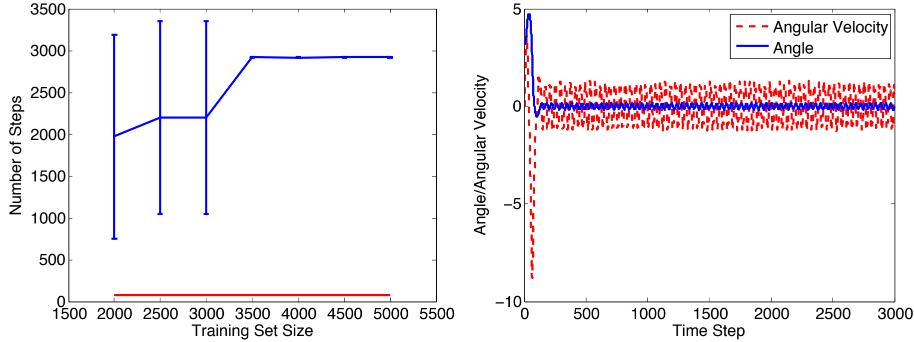


Fig. 2: Results for the inverted pendulum. In the left panel, the red line shows the number of steps needed to reach the region $(-\pi/12, \pi/12)$, and the blue line shows the number of steps for which the pendulum stayed in the region $(-\pi/12, \pi/12)$. In the right panel, the blue line shows the angle of the pendulum measured from upright, starting at π and stabilizing around 0, while the red line indicates the angular velocity.

are the masses of the cart and the pole and the length of the pole, while $\tau = 0.02$ is the time step duration, and g is the gravitational constant. The agent receives a penalty of -1 for each step that the pendulum is below the horizontal line, and a reward of 0 for each step the pendulum is above the horizontal line, i.e. $r(\theta, \dot{\theta}) = 0$ if $\cos(\theta) > 0$; $r(\theta, \dot{\theta}) = -1$ otherwise. An implementation of the above system dynamics is available online [5].

Given n training samples $\{s_i, a_i, r_i, s_i'\}_{i=1}^n$, the transition matrix T is the least squares solution of the linear system of n equations

$$[f_s(s_i)\ f_a(a_i)]T = s_i'$$

where

$$\begin{aligned} f_s(s) = [&\theta, \dot{\theta}, \sin(\theta), \cos(\theta), \sin(\theta)^2, \cos(\theta)^2, \sin(\theta)\cos(\theta), \\ &\dot{\theta}\sin(\theta), \dot{\theta}\cos(\theta), \dot{\theta}\sin(\theta)^2, \dot{\theta}\cos(\theta)^2, \dot{\theta}\sin(\theta)\cos(\theta), \\ &\dot{\theta}^2\sin(\theta), \dot{\theta}^2\cos(\theta), \dot{\theta}^2\sin(\theta)^2, \dot{\theta}^2\cos(\theta)^2, \dot{\theta}^2\sin(\theta)\cos(\theta)] \\ f_a(a) = [&a\cos(\theta), a\sin(\theta)]. \end{aligned}$$

The function f_s is a nonlinear representation that maps the state to a higher-dimensional space. Though it appears complex, the mapping f_s is easy to define: it consists of the terms of the polynomial $(1 + \sin(\theta) + \cos(\theta))^2(1 + \dot{\theta} + \dot{\theta}^2)$ without the constant term. Representing the state using such a mapping captures the dependence of the transition model on higher order terms of $\dot{\theta}, \sin(\theta)$ and $\cos(\theta)$. Adopting this mapping does not require prior knowledge of the real transition model. Similarly, the function f_a maps the action to a higher-dimensional space, simply separating the action into the component directions along and perpendicular to the pendulum.

Again following the intuition suggested in Section 3 above, the basis functions consist of 25 Gaussians evenly distributed in $\{-\pi, 0, \pi\} \times \{-3, 0, 3\}$. Each Gaussian has

width proportional to the distance between two neighboring centers. Note that although the Gaussians are centered inside the grid bounded by $\{-\pi, \pi\} \times \{-3, 3\}$, the values of angle and angular velocity may go outside this grid. The basis functions are thus

$$\phi_m(\hat{s}) = \exp\left(-\frac{1}{2}(\hat{s} - c_m)\Sigma^{-1}(\hat{s} - c_m)^T\right)$$

where

$$\hat{s} = [f_s(s)\ f_a(a)]T$$

$$\Sigma = \begin{bmatrix} \pi/5 & 0 \\ 0 & 3/5 \end{bmatrix}$$

$$c_m \in \{-\pi, \pi/2, 0, \pi/2, \pi\} \times \{-3, -1.5, 0, 1.5, 3\}$$

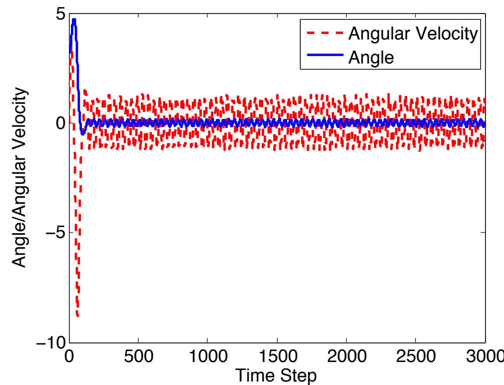
and the approximate Q function is

$$Q(\hat{s}) = \sum_{m=1}^{25} \phi_m(\hat{s}) v_m.$$

Samples for training were collected by randomly sampling the state-action space. Each sample (s, a, r, s') was generated by randomly selecting a state from $[-\pi, \pi] \times [-3, 3]$ and an action from {-10, 0, 10}. The next state and the reward were calculated following the system dynamics. For testing, each test trial began with the pendulum at the lowest position and the policy was followed for at most 3000 steps. The performance metrics tracked were the time taken to reach the goal region $(-\pi/12, \pi/12)$, and the time the pendulum stayed in this region. The experiment was repeated 10 times for each training set size. The performance for each training set size is reported as the average over the 10 experiments. All policies were learned using $\gamma = 0.95$ with strength of regularization $\lambda = 1$. The strength of regularization was chosen empirically to ensure the algorithm arrives at a stable estimate of the weights.

The variation of performance with training set size is shown in Fig. 2. A policy that controls the pendulum to the goal is learned with as few as 1500 samples. With a training set size of 3500 or more samples, the learned policies consistently control the pendulum to the goal and keep it in the upright position for an indefinite duration.

As a further test, a training dataset of 4000 samples was generated as described earlier. The policy learned from this dataset was allowed to choose actions from the set {-10, -8, -2, ..., 0, 2, ..., 10} during testing. Though the policy was trained on only the actions {-10, 0, 10}, it was still able to balance the pendulum using the new set of actions (Fig. 3). Further, it chose the new actions for 1,279 steps out of a 3000 step trial, preferring the lower magnitude actions when the pendulum was closer to the goal state. This indicates that the learned Q function correctly estimates the greater utility of lower magnitude actions closer to the goal, even though those actions were absent during training. This is because the basis functions depend only on the predicted next state, and prediction of next states is sufficiently accurate even for the new actions. For comparison, in the framework used by Lagoudakis and Parr [8] and by other researchers for the inverted pendulum task, each of the three actions {-10, 0, 10} has its own set of basis functions. Because of the explicit dependence on discrete actions as arguments, each new action requires additional basis functions in that framework.



(a)

Fig. 3: Pendulum trajectory using the extended set of actions: the pendulum is still balanced, despite the policy being learned for a different set of actions.

6 Management of a wind farm

The day-ahead wind commitment problem is a multistage stochastic optimization problem described recently [4]. The agent is a wind energy producer with a certain storage capacity for electricity. Energy markets are based around bids for future production, so producers must commit a certain amount of electricity 24 hours ahead in the day-ahead market. At the end of each day, the agent knows the hourly wind speed for the day and the hourly prices per unit of electricity for the next day.

The agent must commit to providing a certain amount of electricity for each hour of the next day. For each unit of electricity committed, it receives revenue equal to the price per unit of electricity at that hour. If the agent is unable to provide the amount committed, it must make up the difference by buying from the spot market at twice the per unit price at that hour. If it generates more electricity than promised, it may store the excess generated electricity. Storage is free but limited. If the storage capacity has been reached, then the excess electricity that cannot be stored must be dumped at the cost of $5 per unit dumped. Units are megawatt hours (MWh) and a typical price per unit is $50, so dumping is not free but not highly expensive either. The goal of the agent is to commit energy each hour of the next day in a way that maximizes profit. Note that the scenario assumes that operating costs are fixed, as are capital costs, so they are not part of the problem definition.

We applied NSPI to the same weather and pricing data used by [4]. Hourly wind speeds were obtained from the North American Land Data Assimilation Survey for the eight years from January 1, 1998 to December 31, 2005 for three locations: the outer banks of North Carolina (33.9375N, 77.9375W), the lake shore near Cleveland, Ohio (41.8125N, 81.5625W), and the ocean shore near Point Judith, Rhode Island (41.3125N, 71.4375W). Day-ahead hourly prices were obtained from the PJM market for the New Jersey area for the period from January 1, 2002 to December 31, 2009.

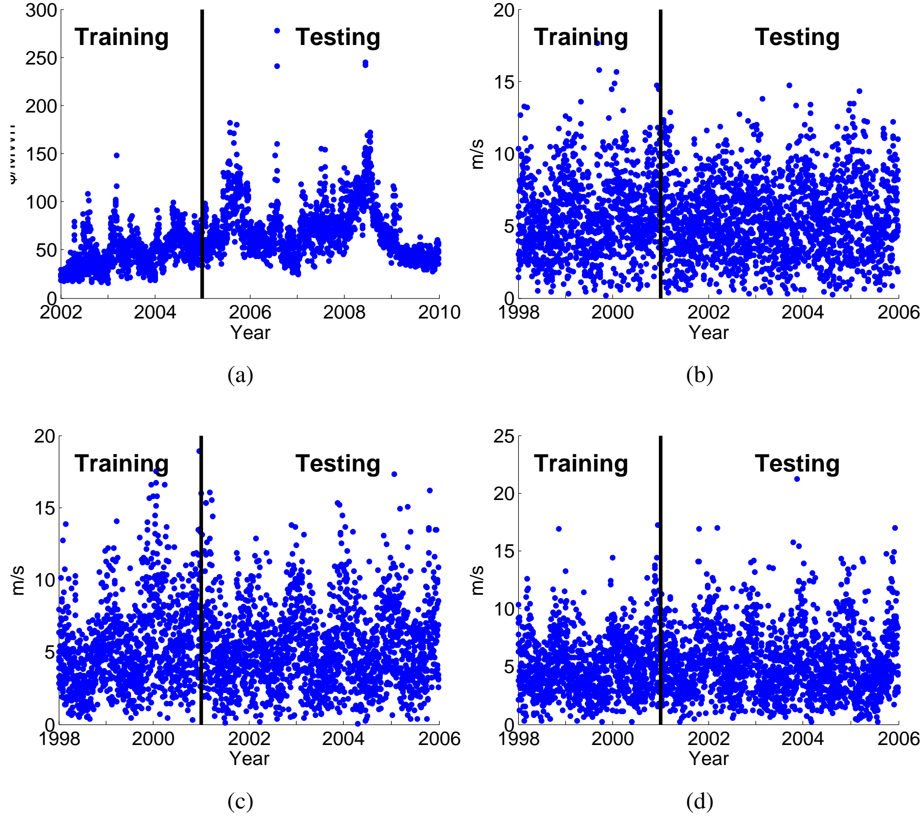


Fig. 4: (a) Price per MWh of electricity from 2002 to 2009, (b) wind speed for North Carolina, (c) wind speed for Rhode Island, (d) wind speed for Ohio.

The sequences of prices and wind speeds for the three locations over time are shown in Fig. 4. Each point indicates the wind speed or the price at hour 12 of each day.

For any given day, the action vector is a 24-dimensional vector $(a_1, ..., a_{24})$, where a_h is the amount of electricity promised to be supplied at hour h of the following day. The state at the end of a given day is the vector $(w_1, ..., w_{24}, s_{24}, p_1, ..., p_{24})$ where w_h is the wind speed at hour h of the day, s_{24} is the electricity in storage at the end of the day, and p_h is the per-unit bid price of electricity at hour h of the next day.

Since the storage level varies within a day, we define s_h to be the storage level at the end of hour h for $h = 1$ to $h = 24$. We further define $L_h = L(w_h)$ to be the power generated from the wind speed w_h. To compute L_h we use a numerical approximation of the power generation curve for an industry standard General Electric 1.5MW SL wind turbine. The approximation was graciously provided by Lauren Hannah via email.

For each hour, the excess available electricity compared to the promised supply is $e_h = s_{h-1} + L(w_h) - a_h$. The storage level at the end of each hour is $s_h =$

$\max\{0, \min\{e_h, M\}\}$ where M is the maximum storage capacity. The units purchased on the spot market during each hour are $b_h = \max\{0, -e_h\}$ and the units dumped during each hour are $d_h = \max\{0, e_h - M\}$. The daily revenue is thus

$$R = \sum_{h=1}^{24} p_h a_h - 2p_h b_h - 5d_h.$$

6.1 Applying NSPI

The first stage of applying NSPI is to learn a transition model that can estimate the expected next state $\hat{s}$ based on any current state s and any selected action a. Here, the learned transition model consists of two matrices T_w and T_p that predict the wind velocities and prices seen in the next state as a function of the current state. Note that these speeds and prices are independent of the current action, and of each other. The matrices T_w and T_p are the least squares solutions of the linear systems

$$(w_1, w_2, ..., w_{24}, y_1, y_2, y_3, y_4)\, T_w = (w'_1, w'_2, ..., w'_{24})$$
$$(p_1, p_2, ..., p_{24}, y_1, y_2, y_3, y_4)\, T_p = (p'_1, p'_2, ..., p'_{24})$$

where the prime notation $'$ refers to the following day in the training data. The four y_i are binary indicator variables that describe which season the current day is in. These indicators are informative because wind velocity and electricity prices have seasonal patterns. The two matrices T_w and T_p can be combined into a single transition matrix

$$T = \begin{bmatrix} T_w & 0 \\ 0 & T_p \end{bmatrix}.$$

The transition model must also predict the storage component of $\hat{s}$, i.e., the estimated storage level $\hat{s}_{24}$ at the end of the next day. This is calculated using the equations above from the action vector and the wind speeds predicted using T_w.

The second stage of applying NSPI is to learn an approximate Q function

$$Q(s,a) = Q(\hat{s}) = \sum_{j=1}^{m} \phi_j(\hat{s}) v_j.$$

We define $m = 26$ basis functions as follows. Let d be the day described by $\hat{s}$. The first basis function is the anticipated revenue $\hat{R}$ achieved during day d, the second basis function is the anticipated storage level $\hat{s}_{24}$ at the end of day d, and the remaining basis functions are the 24 estimated prices for day $d+1$. The trained weights v_j of these basis functions capture the estimated long-term value of the estimated state $\hat{s}$. Formally, $\phi(\hat{s}) = (\hat{R}, \hat{s}_{24}, \hat{p}_1, ..., \hat{p}_{24})$ where $\hat{R}$ and $\hat{s}_{24}$ are estimates that are computed deterministically from $(\hat{w}_1, ..., \hat{w}_{24})$ and $(\hat{p}_1, ..., \hat{p}_{24})$ using the equations above.

During testing, given a state s the recommended action is computed by maximizing $Q(\hat{s})$ using an interior point algorithm [1]. Specifically,

$$a^* = \operatorname*{argmax}_a Q(s,a) = \operatorname*{argmax}_a \sum_{j=1}^{m} \phi_j([s\ a]T) v_j.$$

Table 1: Average annual reward in $1,000 for fixed storage sizes for ADPS and NSPI.

Site	Storage capacity	ADPS	NSPI
NC	7.5MWh	114.86	149.80
	15MWh	163.51	184.70
	30MWh	205.38	208.95
OH	7.5MWh	90.53	123.32
	15MWh	131.83	155.67
	30MWh	171.82	181.09
RI	7.5MWh	107.60	138.56
	15MWh	155.00	173.23
	30MWh	200.83	197.75

This maximization is computationally efficient because the 24-dimensional action vector is continuous. With discrete actions, searching the action space would be much more expensive.

6.2 Experiments

Training and test data are identical to those used previously [4]. Each location is an independent agent. The first three years of data (01/01/1998 to 12/31/2000 for wind, 01/01/2002 to 12/31/2004 for prices) are used to create the training samples. The policy for generating training samples is persistent [4]: at the end of each day, commit for the next day as much electricity as was generated (sold, stored, or dumped) on the current day. Based on this commitment, the actual next state and reward are calculated. using the transition equations above. The training policy is followed for each day of the first 3 years, resulting in 1095 samples from the 1096 days. All learning uses a discount factor $\gamma = 0.9$. This numerical value is chosen somewhat arbitrarily, based on the assumption that weather and prices are predictable at most a few days in advance, so decisions need not take into account the far distant future. The strength of regularization was chosen empirically to be $\lambda = 20$.

For testing, each wind farm begins with no stored electricity and is followed over the last 5 years of data. For each wind farm, different policies are separately learned and tested for storage capacities of 7.5MWh, 15MWh and 30MWh. Thus nine policies are learned and evaluated: three for each wind farm, for three different storage capacities.

Table 1 compares the policies learned using NSPI with the policies learned using the Approximate Dynamic Programming for Storage (ADPS) algorithm [4]. The metric used for comparisons is the annual revenue averaged over the five year test period. The paper [4] presents several results obtained using different parameters for the ADPS method. For comparison purposes we use the best performances obtained via ADPS for each case, i.e., each row in the table may use different parameters for ADPS to achieve the best performance. NSPI outperforms ADPS in all but the last case. The biggest improvement is for the smaller storage sizes, for which the task is intrinsically more difficult because a large storage capacity can rarely be used fully. The performance improvement is encouraging considering that ADPS is a method designed specifically for solving storage problems while NSPI is a general method for learning Q functions.

To estimate the transition model, [4] employ a Dirichlet process mixture model, which is considerably more complex than the linear model used here. Another advantage of the approach here is that it treats storage level as a continuous variable, while the ADPS method discretizes the possible storage levels.

The weather and pricing data used here, and in previous research, are from different time periods. This fact implies that wind speeds and prices are statistically independent. In general, these can be correlated. In this case, additional information in the form of tomorrow's prices would be available for predicting tomorrow's wind speeds. We can take advantage of this additional information simply by not restricting the transition matrix T to be block-diagonal. Similarly, if tomorrow's prices are correlated with today's actions by the agent (because other market participants observe these actions and react to them), this can be taken into account by including the action vector when learning T. This will be necessary if the agent produces a significant share of the electricity supply of the region, so that it has market power and cannot be assumed to be merely a price-taker.

7 Discussion

This paper proposes a simple but useful approach to the definition of basis functions for use in a reinforcement learning method. The approach is driven by the observation that the goal of an optimal policy is to choose the best next state. The primary idea is to write $Q(s, a) = Q(\hat{s})$ where $\hat{s}$ is an estimate of the expected next state reached by taking action a in the current state s. The second idea is that basis functions should capture how good the estimated expected next state $\hat{s}$ is. For goal-oriented tasks, the goodness of $\hat{s}$ can be captured by basis functions that represent local neighborhoods, because the coefficient of each basis function can then indicate the proximity of the neighborhood to good and bad terminal states. For tasks such as inventory management, each basis function can describe a certain component or aspect of the following state, such as the level or availability of a particular resource.

The intuitions just described are made concrete in the NSPI algorithm, which is the well-known least squares policy iteration method with the Q function changed to depend on the expected next state $\hat{s}$ instead of on s and a directly. Experimental results demonstrate the effectiveness of NSPI in three domains, including one with high-dimensional continuous state and action spaces.

One issue worth exploring with NSPI is using a more complex but possibly more accurate transition model. The experiments above use a linear transition model, but Gaussian processes, Dirichlet process mixture models, and others have been used effectively in prior work. It is possible that NSPI may perform better with a more accurate transition models, especially in domains where next-state dynamics are highly nonlinear, such as robot navigation with obstacles, including the well-known so-called puddle world. Another avenue for future work is to use methods other than LSPI after the transition model has been learned. In particular, the process suggested here for making the Q function depend on estimated next states could be combined with the fitted Q iteration algorithm, and related methods [3, 2].

References

1. R. H. Byrd, J. C. Gilbert, and J. Nocedal. A trust region method based on interior point techniques for nonlinear programming. *Mathematical Programming*, 89:149–185, 1996.
2. C. Elkan. Reinforcement learning with a bilinear Q function. In *Proceedings of the Ninth European Workshop on Reinforcement Learning (EWRL)*, number 7188 in Lecture Notes in Computer Science, pages 25–47. Springer Verlag, 2011.
3. D. Ernst, P. Geurts, and L. Wehenkel. Tree-based batch mode reinforcement learning. *Journal of Machine Learning Research*, 6(1):503–556, 2005.
4. L. Hannah and D. B. Dunson. Approximate dynamic programming for storage problems. In *Proceedings of the 28th International Conference on Machine Learning (ICML)*, pages 337–344, 2011.
5. A. Hesami. Matlab implementation of inverted pendulum. Available at `http://webdocs.cs.ualberta.ca/~sutton/pole.zip`.
6. R. A. Howard. Comments on the origin and application of Markov decision processes. *Management Science*, 14(7):503–507, 1968.
7. N. Jong and P. Stone. Model-based function approximation in reinforcement learning. In *Proceedings of the Sixth International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 658–665. ACM, 2007.
8. M. G. Lagoudakis, R. Parr, and L. Bartlett. Least-squares policy iteration. *Journal of Machine Learning Research*, 4:1107–1149, 2003.
9. S. Mahadevan and M. Maggioni. Proto-value functions: A Laplacian framework for learning representation and control in Markov decision processes. *Journal of Machine Learning Research*, pages 2169–2231, 2007.
10. F. S. Melo and M. Lopes. Fitted natural actor-critic: A new algorithm for continuous state-action MDPs. In *Proceedings of the European Conference on Machine Learning (ECML)*, volume 5212 of *Lecture Notes in Computer Science*, pages 66–81. Springer, 2008.
11. I. Menache, S. Mannor, and N. Shimkin. Basis function adaptation in temporal difference reinforcement learning. *Annals of Operations Research*, 134(1):215–238, 2005.
12. R. Parr, L. Li, G. Taylor, C. Painter-Wakefield, and M. Littman. An analysis of linear models, linear value-function approximation, and feature selection for reinforcement learning. In *Proceedings of the 25th International Conference on Machine Learning (ICML)*, pages 752–759, 2008.
13. W. B. Powell. Merging AI and OR to solve high-dimensional stochastic optimization problems using approximate dynamic programming. *INFORMS Journal on Computing*, 22(1):2–17, 2010.
14. W. D. Smart and L. P. Kaelbling. Practical reinforcement learning in continuous spaces. In *Proceedings of the International Conference on Machine Learning (ICML)*, pages 903–910. Morgan Kaufmann, 2000.
15. R. S. Sutton. Reinforcement learning architectures for animats. In *Proceedings of the international workshop on the simulation of adaptive behavior: From animals to animats*, pages 288–296. MIT Press, 1991.
16. R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. Cambridge University Press, 1998.
17. J. N. Tsitsiklis. Commentary—perspectives on stochastic optimization over time. *INFORMS Journal on Computing*, 22(1):18–19, Jan. 2010.
18. V. Uc Cetina. Multilayer perceptrons with radial basis functions as value functions in reinforcement learning. In *Proceedings of the 16th European Symposium on Artificial Neural Networks (ESANN)*, pages 161–166, 2008.
19. H. O. Wang, K. Tanaka, and M. F. Griffin. An approach to fuzzy control of nonlinear systems: stability and design issues. *IEEE Transactions on Fuzzy Systems*, 4(1):14 –23, 1996.