

Reinforcement learning II

So far, we have considered reinforcement learning using the framework of discrete MDPs. In particular, we assumed a finite state space. We now discuss ways to handle problems where the state space is continuous. Specifically, we discuss two general methods for solving continuous MDPs. The first, based on discretization, will approximate the continuous value function in terms of a value function defined on a discrete set of points drawn from the continuous space. The second, called fitted value iteration, will compute an approximation to the value function using an iterative learning algorithm. Fitted value iteration will allow us to scale MDP algorithms to much larger numbers of dimensions than would be possible using discretization.

1 Discretization

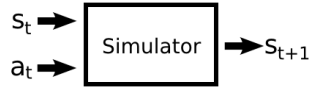
Suppose you want to apply reinforcement learning to driving a car. Let's say we model the state of the car as the vector

$$s_t = \begin{bmatrix} x_t \\ y_t \\ \theta_t \end{bmatrix} \in \mathbb{R}^3,$$

where x_t and y_t give the location of the car at time t , and θ gives its orientation. Assume for now that we have a finite set of actions A (e.g. turn left, step on the breaks, etc.).¹ We suppose we have a simulator which takes a state-action pair (s_t, a_t) for time t and returns a state s_{t+1} for time $t + 1$. In

¹In some problems, we can have continuous actions also. For instance, when driving, we can control how much to turn left, how strongly to step on the brakes, and so on. But for most problems, A has only a small number of degrees of freedom, and so it's not hard to discretize. For instance, we might realistically model the state of a car as a vector

other words, it will sample s_{t+1} from the distribution $P_{s_t a_t}$. We treat the simulator as a black box which takes a state-action pair and returns a successor state:



In some cases, we have a good model of the physics, and therefore we can determine the transition probabilities $P_{s_t a_t}$ through physical simulation. Often, however, it's hard to construct a good physical model a priori. In these cases, the simulator itself has to be learned.

Sometimes, the simulator is **deterministic**, in that it will always return the same state s_{t+1} for a particular state-action pair (s_t, a_t) . Sometimes, the simulator is **stochastic**, where s_{t+1} is a random function of s_t and a_t .

Say we have a continuous state space S . One way to apply the techniques from the previous set of notes is to **discretize** S to obtain a discrete set of states $\bar{S} = \{\bar{s}^{(1)}, \bar{s}^{(2)}, \dots, \bar{s}^{(n)}\}$. For instance, we might choose to break up the continuous state space S into boxes and let \bar{S} have one discrete state $\bar{s}^{(i)}$ for each of these boxes. We will refer to this method as **simple grid discretization**. This is not necessarily the best approach for many problems, but it is simple to implement, and is often good enough. In these notes, s will always denote a state in the original continuous state, and \bar{s} will denote one of the discrete states.

After discretizing our continuous space, we typically need to estimate the transition probabilities $P_{\bar{s}a}(\bar{s}')$. We do this by taking a lot of samples and estimating $P_{\bar{s}a}(\bar{s}')$ as the proportion of times we landed in discrete state \bar{s}' after taking action a in some continuous state associated with discrete state \bar{s} . More formally, we use the algorithm shown in Figure 1.

We also need to estimate the reward function for our discretized MDP. This can be done analogously to how we estimate the transition probabilities. Specifically, we take a large number of samples from S , and then for each discrete state \bar{s} , we take $R(\bar{s})$ to be the mean value of $R(s)$ for all of our samples s which were associated with \bar{s} .

$s \in \mathbb{R}^6 = (x, y, \theta, \dot{x}, \dot{y}, \dot{\theta})$ giving the location and orientation, and the rate of change of each. The set of actions, however, can be given as a vector in \mathbb{R}^2 . Therefore, we will focus our attention on dealing with continuous state spaces, and assume the set of actions is discrete.

For $i = 1$ to k ,

 Sample s_t randomly from the continuous state space S .

 For each action $a \in A$,

 “Try” action a from state s_t . In other words, let s_{t+1} be the state returned by the simulator when given state s_t and action a_t .

 Find the discrete states \bar{s}_t and \bar{s}_{t+1} associated with the continuous states s_t and s_{t+1} .

Estimate

$$P_{\bar{s}_t a}(\bar{s}_{t+1}) = \frac{\# \text{ times action } a \text{ in state } \bar{s}_t \text{ caused a transition to state } \bar{s}_{t+1}}{\# \text{ times we tried action } a \text{ in state } \bar{s}_t}.$$

Figure 1: An algorithm for estimating the transition probabilities in a discretized space.

This process gives us the complete specification of a discrete MDP, which we can then solve using the techniques such as value iteration or policy iteration. This will give us the optimal value function $V^*(\bar{s})$ and the optimal policy $\pi^*(\bar{s})$ for the discrete problem. How do we use this to choose a policy for the original continuous problem? If we want to choose an action a_t for a given continuous state s_t , we can map s_t to the corresponding discrete state \bar{s}_t and choose the action $a_t = \pi^*(\bar{s}_t)$.

Formulating a problem as a continuous MDP has several advantages over the motion planning algorithms presented earlier in this course. Specifically, it can handle cases that deterministic motion planning methods can't, including *nonholonomic motion* and *stochasticity*. As a consequence of its being able to handle nonholonomic problems, it can also account for the **dynamics** of the problem. In other words, it can account for the rates of change of the different variables. For instance, in the notes on motion planning, we represented the configuration of a helicopter as a vector in \mathbb{R}^6 , with three dimensions for the location of the helicopter and three dimensions for its orientation. In the MDP framework, we can represent the state as a vector in \mathbb{R}^{12} which also includes the rate of change of each of these variables. This means a state where the helicopter is flying steadily in the air will be treated as distinct from one where it is plummeting towards the ground.

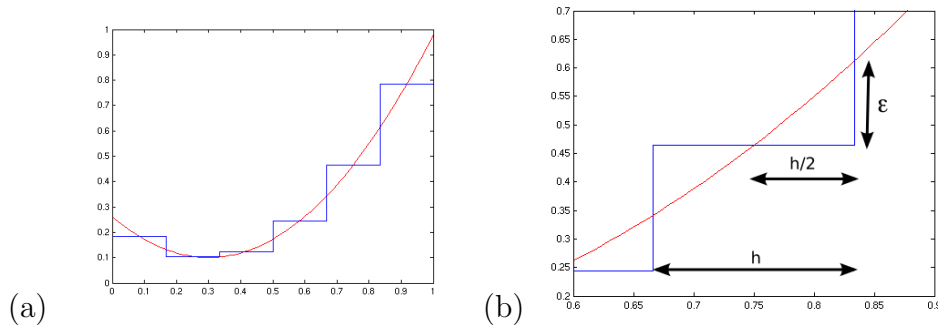


Figure 2: (a) An example of simple grid discretization applied to a real-valued function on the interval $[0, 1]$. (b) A close-up with the edge width h and error ε labeled.

2 Linear interpolation

Previously, we introduced a simple algorithm for discretizing a continuous state space. We supposed we had a function which would assign each continuous state s to some discrete state \bar{s} . The value we assign to s would simply be the value of \bar{s} . For the moment, let's restrict ourselves to one-dimensional state spaces, and consider the more general problem of discretizing some real-valued function $f : \mathbb{R} \mapsto \mathbb{R}$ on the interval $[0, 1]$. What does simple discretization look like when applied to this problem? We would partition $[0, 1]$ into some discrete set of intervals. We then approximate the value of f in these intervals as being constant, to get an approximation \bar{f} that is **piecewise constant**. An example is shown in Figure 2a. In the sequel, we will use h to denote the width of each of the intervals. (E.g., if you discretize $[0, 1]$ into n discrete intervals, then we would have $h = 1/n$.)

How good is this approximation? Clearly, this will depend how finely we discretize the input domain. I.e., it will depend on how small h is. We are specifically interested in the problem of how well \bar{f} approximates f as $h \rightarrow 0$ (i.e., in the limit of finer and finer discretizations). Figure 2b shows a close-up view of f and its approximation \bar{f} . If the function f has approximately gradient m in this region, then it is possible to show that the maximum error of the approximation is given by

$$\varepsilon \approx \frac{h}{2}m = O(mh).$$

This big- O notation is in the limit of $h \rightarrow 0$. The error, therefore, decreases

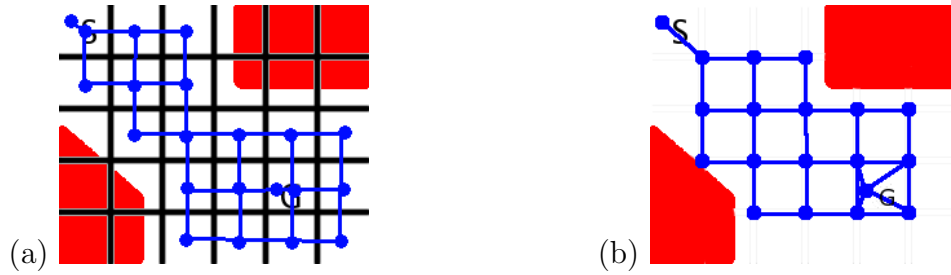


Figure 3: An example of the difference between the two kinds of grid discretization, using a configuration space from our motion planning lecture. (a) The kind of discretization where the space is divided into “boxes.” (b) The kind of discretization where we lay down a lattice of grid points in the state space. This is the kind of discretization we will be using.

linearly with the density at which the points are sampled.² If you want your answer to be 10 times as accurate, you need to sample 10 times as many points. This problem gets worse when we add more dimensions; in three dimensions, sampling 10 times more finely requires 10^3 times as many samples. Hence, we would need to choose 1000 times as many points to get one more significant digit accuracy.

Fortunately, we can do better than this with relatively little additional computational cost. Rather than approximate f with a piecewise constant function, we will use a **piecewise linear** approximation. To explain this, we will need to take a different view of discretization. Previously, we talked about discretization in terms of splitting up the state space into a number of “boxes.” We then associated each of the “boxes” with a value. This is illustrated in Figure 3a. However, we will now take a different view of discretization in which we lay down a lattice of grid points in the state space. We will then associate each of the **lattice points** with a value. This is shown in Figure 3b. Sometimes, the “boxes” view leads to much more natural algorithms, and sometimes the “lattice” view does. If you’re ever choosing some discretization for a problem, remember this picture, and make sure to choose the most appropriate one for your problem.

In detail, let’s again consider the case of discretizing a function f over

²You are probably used to big-O notation where $n \rightarrow \infty$. Here, the big-O notation represents the limit as $h \rightarrow 0$. More formally, $g_1(h) = O(g_2(h))$ if for any constant $c > 0$, there exists some constant d such that $g_1(h) \leq cg_2(h)$ whenever $0 < h < d$.

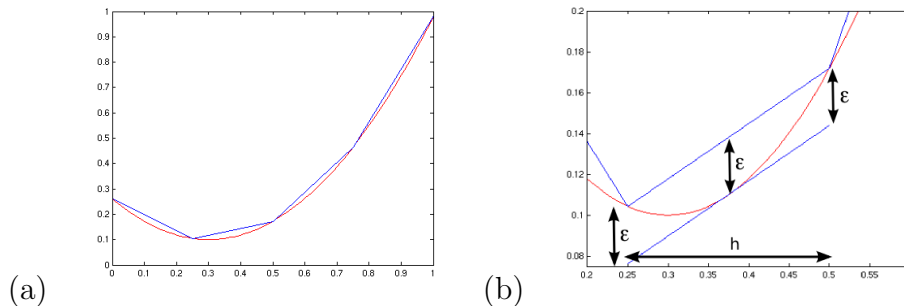


Figure 4: (a) An example of linear interpolation applied to a real-valued function on the interval $[0, 1]$. (b) A close-up with the edge width h and error ε labeled.

$[0, 1]$. Suppose we choose $n + 1$ lattice points, at locations

$$\bar{x}^{(0)} = \frac{0}{n}, \bar{x}^{(1)} = \frac{1}{n}, \dots, \bar{x}^{(n)} = \frac{n}{n}.$$

If we are trying to approximate $f(x)$ for some $x \in \mathbb{R}$, let $x^{(-)}$ denote the nearest lattice point to the left of x , and $x^{(+)}$ the nearest one to the right of x . We then **interpolate** between the values $f(x^{(-)})$ and $f(x^{(+)})$. More formally, let $\alpha = \frac{x - x^{(-)}}{x^{(+)} - x^{(-)}}$ denote the fraction of the space between $x^{(-)}$ and $x^{(+)}$ which lies to the left of x . Then we approximate

$$\bar{f}(x) = (1 - \alpha)f(x^{(-)}) + \alpha f(x^{(+)}).$$

This process is known as **linear interpolation**, and it is illustrated in Figure 4a. Visually, it certainly looks like a better approximation than our piecewise constant one.

How accurate is linear interpolation? As before, we check how fast ε decreases as $h \rightarrow 0$. It turns out that the error decreases quadratically with h , i.e., $\varepsilon = O(h^2)$. The precise statement of this result is somewhat technical, but it suffices to say linear interpolation gives significantly better results than piecewise constant approximations. Put the other way, with linear interpolation, the number of grid points we need is roughly the *square root* of what we would need with piecewise constant approximations.

2.1 Value iteration updates

Still limiting ourselves to the 1-dimensional case, we're going to show how to apply value iteration when we use linear interpolation on our discretized

points.

Suppose our state space S is an interval in \mathbb{R} , and we have discretized it into a grid of n points $\bar{s}^{(1)}, \dots, \bar{s}^{(n)}$. We will explicitly determine the value function $V(\bar{s})$ at the grid points, and then compute $V(s)$ with linear interpolation everywhere else. If s is between $\bar{s}^{(i)}$ and $\bar{s}^{(i+1)}$, and $\alpha = \frac{s - \bar{s}^{(i)}}{\bar{s}^{(i+1)} - \bar{s}^{(i)}}$, we define

$$V(s) = (1 - \alpha)V(\bar{s}^{(i)}) + \alpha V(\bar{s}^{(i+1)}).$$

We will approximately solve for $V(\bar{s}^{(i)})$ for each lattice point $\bar{s}^{(i)}$ using value iteration. Assume for simplicity that our simulator is deterministic, i.e., there is some function $\delta : S \times A \mapsto S$ which gives us the resulting state s' whenever we feed a state-action pair (s, a) into the simulator. In this case, we do the following:

Initialize $V(\bar{s}^{(i)}) = 0$ for all grid points $\bar{s}^{(i)}$.

Repeat until convergence:

For each grid point $\bar{s}^{(i)}$:

For each action $a \in A$:

Let $s'_a = \delta(\bar{s}^{(i)}, a)$.

Compute $V(s'_a)$ via linear interpolation.

Set $V(\bar{s}^{(i)}) := R(\bar{s}^{(i)}) + \gamma \max_a V(s'_a)$.

As with ordinary value iteration, this will converge, giving us the (approximate) optimal policy V^* . We can choose our policy as:

$$\pi^*(s) = \arg \max_a V^*(\delta(s, a)).$$

Essentially, we are using our simulator to do one-step lookahead to see which action takes us to the state with the highest value.

We just assumed the simulator was deterministic. If the simulator is stochastic, then rather than choosing a single point $s' = \delta(\bar{s}^{(i)}, a)$, we sample k successor states, and use the average value of these successor states in the update rule. Finally, to choose an action a_t for a given continuous state s_t , we run the simulator k times for each action a to get k points s'_{a1}, \dots, s'_{ak} . We choose:

$$\pi^*(s) = \arg \max_a \frac{1}{k} \sum_{i=1}^k V^*(s'_{ai}).$$

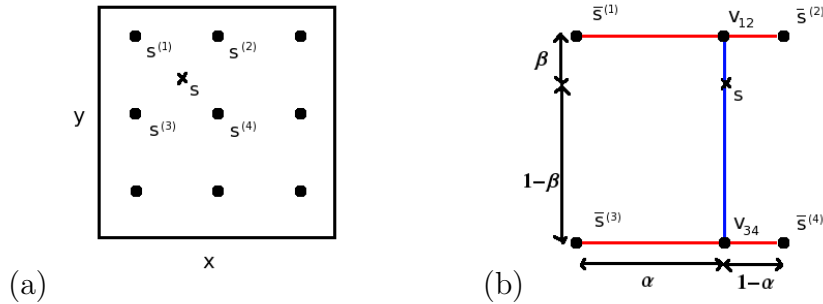


Figure 5: (a) An example of a discretized MDP. We are trying to estimate the value of state s based on the values assigned to our lattice points $\bar{s}^{(1)}, \dots, \bar{s}^{(4)}$. (b) How to apply bilinear interpolation to this MDP. First, the value function is interpolated between $\bar{s}^{(1)}$ and $\bar{s}^{(2)}$ to get V_{12} , and between $\bar{s}^{(3)}$ and $\bar{s}^{(4)}$ to get V_{34} . Then we interpolate between V_{12} and V_{34} to get $V(s)$.

3 Multilinear interpolation

We now describe the generalization of linear interpolation to higher dimensional state spaces. Suppose now that we have a two-dimensional continuous state space S , which we have discretized into a grid as shown in Figure 5a. Assume we somehow have a value function V defined on all of the grid points. How do we compute $V(s)$? Notice first that we can't simply use piecewise linear interpolation. The values of V at our four grid points $\bar{s}^{(1)}, \dots, \bar{s}^{(4)}$ have to be specified with four real numbers. On the other hand, if we tried to predict $V(s)$ with a linear function, e.g.

$$V(s) = \theta_0 + \theta_1 s_1 + \theta_2 s_2,$$

we only have three degrees of freedom to specify V . Hence, we can't always come up with a linear function which matches V at each of the four grid points $\bar{s}^{(1)}, \dots, \bar{s}^{(4)}$.

Instead, we will use **bilinear interpolation**, as demonstrated in Figures 5b and 6. First, we interpolate linearly between $\bar{s}^{(1)}$ and $\bar{s}^{(2)}$ to get V_{12} . Similarly, we compute V_{34} by interpolating $\bar{s}^{(3)}$ and $\bar{s}^{(4)}$. Finally, we interpolate between V_{12} and V_{34} to get $V(s)$.

It might seem funny to interpolate in the x direction before interpolating in the y direction. Does this give preference to one coordinate over the other? Actually, it turns out that we get the same result no matter which coordinate we interpolate first. We can show this by taking our definition of bilinear

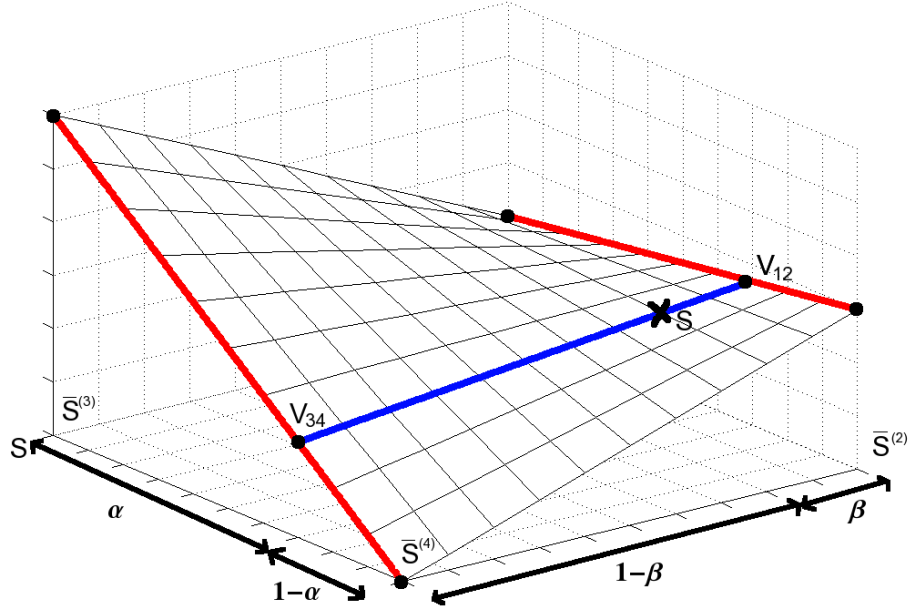


Figure 6: An example of bilinear interpolation of the value function between four lattice points $\bar{s}^{(1)}, \dots, \bar{s}^{(4)}$.

interpolation and expanding out the polynomial in terms of α and β :

$$\begin{aligned}
 V(s) &= (1 - \beta) \left((1 - \alpha)V(\bar{s}^{(1)}) + \alpha V(\bar{s}^{(2)}) \right) \\
 &\quad + \beta \left((1 - \alpha)V(\bar{s}^{(3)}) + \alpha V(\bar{s}^{(4)}) \right) \\
 &= (1 - \alpha)(1 - \beta)V(\bar{s}^{(1)}) + \alpha(1 - \beta)V(\bar{s}^{(2)}) \\
 &\quad + (1 - \alpha)\beta V(\bar{s}^{(3)}) + \alpha\beta V(\bar{s}^{(4)}). \tag{1}
 \end{aligned}$$

In other words, the weights of the points to the left of s are proportional to $1 - \alpha$, while the weights of the points to the right of s are proportional to α . The same is true for the vertical direction. Clearly, if we had begun by interpolating vertically rather than horizontally, we would have arrived at the same formula.

Let's stop to think briefly about what it means for a function to be bilinear. Our function $V(s)$, as computed by the polynomial (1), is linear in each coordinate of s taken individually. In other words, as we move s directly north, our value function changes linearly. However, $V(s)$ is *not* linear in s . As s moves from one corner $\bar{s}^{(3)}$ to the opposite corner $\bar{s}^{(2)}$, $V(s)$ clearly

changes nonlinearly. A function which is linear in each of its inputs taken separately is called bilinear.

Our example was in two dimensions, but if we apply the exact same technique in higher dimensional spaces, we get **multilinear interpolation**.

4 Fitted value iteration

Unfortunately, both of the methods we described above (simple grid discretization and multilinear interpolation) suffer from exponential growth in the problem size as the number of dimensions is increased. (We saw exactly the same problem with the discretization techniques we introduced in our motion planning lecture.) In reinforcement learning, this exponential growth is often referred to as the **curse of dimensionality**. Simple grid discretization works well in 3 dimensions, is sometimes OK in 4 dimensions, and occasionally works in 6 dimensions with much luck and effort. With multilinear interpolation, we can buy ourselves a couple extra dimensions; it sometimes works in 6 dimensions, but only rarely in 7 or 8.

But in our supervised learning lectures, we used algorithms which performed well for hundreds or thousands of dimensions. Linear regression was able to handle large numbers of dimensions because its hypotheses were restricted to be linear functions of the inputs. This suggests learning a value function which is a linear function of some predefined set of features. More formally, suppose we have a **feature map** $\phi : S \mapsto \mathbb{R}^n$, which associates with each state s a **feature vector** $\phi(s) \in \mathbb{R}^n$. For instance, if our state space is one dimensional, and we want to approximate our value function as a cubic polynomial, we might use the feature vector

$$\phi(s) = \begin{bmatrix} s \\ s^2 \\ s^3 \end{bmatrix}.$$

Or, if our state space is n -dimensional, we might simply choose as our feature map the identity function $\phi(s) = s$.

We will use the **value function approximation**, and approximate the value function V as a linear function of the feature vector $\phi(s)$, i.e.

$$V(s) = V_\theta(s) = \theta^T \phi(s) = \sum_{i=1}^n \theta_i \phi_i(s).$$

Note that this is completely analogous to linear regression, where our hypotheses h_θ were linear functions of the input variables x_i :

$$h_\theta(x) = \theta^T x = \sum_{i=1}^n \theta_i x_i.$$

We can learn the weights θ using another variant of value iteration called **fitted value iteration**. If our simulator is deterministic, the algorithm is as shown in Figure 7.

Eventually, we will find a linear approximation V_θ to the optimal value function. Unlike the previous algorithms we presented, fitted value iteration is not guaranteed to converge, but in practice, it will usually converge or approximately converge. As with multilinear interpolation, we can then find the optimal policy with

$$\pi_\theta^*(s) = \arg \max_a V_\theta^*(\delta(s, a)).$$

If the simulator is stochastic, then when setting $y^{(i)}$, we must estimate the expected value of the successor state by sampling successor states from the simulator and taking the average. Given a continuous state s , for each action a , we run the simulator k times for each action a to get k points s'_{a1}, \dots, s'_{ak} . We choose:

$$\pi^*(s) = \arg \max_a \frac{1}{k} \sum_{i=1}^k V^*(s'_{ai}).$$

Initialize $\theta = 0$.

Sample a set of states $\bar{s}^{(1)}, \bar{s}^{(2)}, \dots, \bar{s}^{(m)}$ randomly from S .

Repeat until convergence:

For $i = 1$ to m :

Set $x^{(i)} = \phi(\bar{s}^{(i)})$.

Set

$$\begin{aligned} y^{(i)} &= R(\bar{s}^{(i)}) + \max_a V(s_a^{(i)}) \\ &= R(\bar{s}^{(i)}) + \max_a \theta^T \phi(s_a^{(i)}), \end{aligned}$$

where $s_a^{(i)} = \delta(\bar{s}^{(i)}, a)$.

Choose

$$\begin{aligned} \theta &:= \arg \min_{\theta} \frac{1}{2} \sum_{i=1}^m (V_{\theta}(\bar{s}^{(i)}) - y^{(i)})^2 \\ &= \arg \min_{\theta} \frac{1}{2} \sum_{i=1}^m (\theta^T x^{(i)} - y^{(i)})^2 \end{aligned}$$

Figure 7: Fitted value iteration algorithm.