CS229 Lecture notes

原作者：[Andrew Ng](http://cs229.stanford.edu/)（[吴恩达](http://open.163.com/movie/2008/1/M/C/M6SGF6VB4_M6SGHFBMC.html)）

翻译：[CycleUser](https://zhuanlan.zhihu.com/python-kivy)

Part XIII

# 强化学习（Reinforcement Learning）和控制（Control）

这一章我们就要学习强化学习（reinforcement learning）和适应性控制（adaptive control）了。在监督学习（supervised learning）中，我们已经见过的一些算法，输出的标签类 y 都是在训练集中已经存在的。这种情况下，对于每个输入特征 x，都有一个对应的标签作为明确的“正确答案（right answer）”。与之相反，在很多的连续判断（sequential decisions making）和控制（control）的问题中，很难提供这样的明确的显示监督（explicit supervision）给学习算法。例如，假设咱们制作了一个四条腿的机器人，然后要编程让它能走路，而我们并不知道怎么去采取“正确”的动作来进行四条腿的行走，所以就不能给他提供一个明确的监督学习算法来进行模仿。在强化学习（reinforcement learning）的框架下，我们就并不提供监督学习中那种具体的动作方法，而是只给出一个奖励函数（reward function），这个函数会告知学习程序（learning agent） 什么时候的动作是好的，什么时候的是不好的。在四腿机器人这个样例中，奖励函数会在机器人有进步的时候给出正面回馈，即奖励，而有退步或者摔倒的时候给出负面回馈，可以理解成惩罚。接下来随着时间的退役，学习算法就会解决如何选择正确动作以得到最大奖励。

强化学习（Reinforcement learning，下文中缩写为 RL）已经成功用于多种场景了，例如无人直升机的自主飞行，机器人用腿来运动，手机的网络选择，市场营销策略筛选，工厂控制，高效率的网页索引等等。我们对强化学习的探索，要先从马尔可夫决策过程（Markov decision processes，缩写为 MDP）开始，这个概念给出了强化学习问题的常见形式。

# 1 马尔可夫决策过程（Markov decision processes）

一个马尔可夫决策过程（Markov decision process）由一个元组（tuple）： (S, A, {Psa}, γ, R)组成，其中元素分别为：

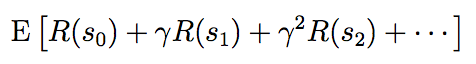
1. S 是一个状态集合（a set of states）。（例如，在无人直升机飞行的案例中，S 就可以是直升机所有的位置和方向的集合。）
2. A 是一个动作结合（a set of actions）。（例如，还以无人直升机为例，A 就可以是遥控器上面能够操作的所有动作方向。）
3. Psa 为状态转移概率（state transition probabilities）。对于每个状态 s ∈ S 和动作 a ∈ A， Psa 是在状态空间上的一个分布（a distribution over the state space）。后面会再详细讲解，不过简单来说， Psa 给出的是在状态 s 下进行一个动作 a 而要转移到的状态的分布。
4. γ ∈ [0, 1] 叫做折扣因子（discount factor）。
5. R : S × A → R 就是奖励函数（reward function）。（奖励函数也可以写成仅对状态 S 的函数，这样就可以写成 R : S → R。）

马尔可夫决策过程（MDP）的动力学（dynamics）过程如下所示：于某个起始状态 s0 启动，然后选择某个动作 a0 ∈ A 来执行 MDP 过程。根据所选的动作会有对应的结果，MDP 的状态则转移到某个后继状态（successor state），表示为 s1，根据 s1 ∼ Ps0a0 得到。然后再选择另外一个动作 a1，接下来又有对应这个动作的状态转移，状态则为 s2 ∼ Ps1a1。接下来在选择一个动作 a2，就这样进行下去。可以用下面的过程来作为表示：



通过序列中的所有状态 s0, s1, . . . 和对应的动作 a0, a1, . . .，就你能得到给出的总奖励值，即总收益函数（total payoff）为 R(s0,a0) + γR(s1,a1) + γ2R(s2,a2) + ···。如果把奖励函数只作为仅与状态相关的函数，那么这个值就简化成了 R(s0) + γR(s1) + γ2R(s2) + ···。多数情况下，我们都用后面这种仅为状态的函数这种形式，虽然扩展到对应状态-动作两个变量的函数 R(s,a) 也并不难。

强化学习的目标就是找到的一组动作，能使得总收益函数（total payoff）的期望值最大：



注意，在时间步长（timestep） t 上的奖励函数（reward）通过一个参数（factor）γt而进行了缩减（discounted）。因此，要使得期望最大化，就需要尽可能早积累符号为正的奖励（positive rewards），而尽量推迟负面奖励（negative rewards，即惩罚）的出现。在经济方面的应用中，其中的 R(·) 就是盈利金额（amount of money made），γ 也可以理解为利润率（interest rate）的表征，这样有自然的解释（natural interpretation），例如今天的一美元就比明天的一美元有更多价值。

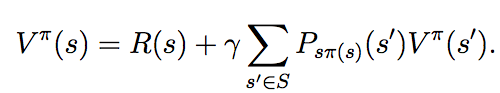
有一种策略（policy），是使用任意函数 π : S → A，从状态（states）到动作（actions）进行映射（mapping）。如果在状态 s，采取动作 a = π(s)，就可以说正在执行（executing）某种策略（policy） π。然后还可以针对策略函数（policy）π 来定义一个值函数（value function）：



Vπ(s) 就是从状态 s 开始，根据 π 给出的动作来积累的部分奖励函数（discounted rewards）的期望总和（expected sum）。1

1 实际上这里我们用 π 这个记号来表示，严格来说不太正确，因为 π 并不是一个随机变量，不过在文献里面这样表示很多，已经成了某种事实上的标准了。

给定一个固定的策略函数（policy） π，则对应的值函数 V π 满足贝尔曼等式（Bellman equations）：



This says that the expected sum of discounted rewards Vπ(s) for starting in s consists of two terms: First, the immediate reward R(s) that we get rightaway simply for starting in state s, and second, the expected sum of future discounted rewards. Examining the second term in more detail, we see that the summation term above can be rewritten Es′∼Psπ(s) [V π(s′)]. This is the expected sum of discounted rewards for starting in state s′, where s′ is distributed according Psπ(s), which is the distribution over where we will end up after taking the first action π(s) in the MDP from state s. Thus, the second term above gives the expected sum of discounted rewards obtained after the first step in the MDP.

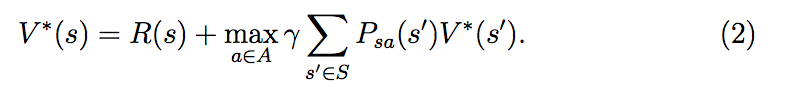
这也就意味着，从状态 s 开始的这个部分奖励（discounted rewards）的期望总和（expected sum） Vπ(s) 由两部分组成：首先是在状态 s 时候当时立即获得的奖励函数值 R(s)，也就是上面式子的第一项；另一个就是第二项，即后续的部分奖励函数值（discounted rewards）的期望总和（expected sum）。对第二项进行更深入的探索，就能发现这个求和项（summation term）可以写成 Es′∼Psπ(s) [V π(s′)] 的形式。这种形式也就是从状态 s′ 开始的这个部分奖励（discounted rewards）的期望总和（expected sum） Vπ(s′)，此处的 s′ 是根据 Psπ(s) 分布的，在 MDP 过程中从状态 s 采取第一个动作 π(s) 之后，确定了这个分布所在的空间。因此，上面的第二项实际上也就是给出了在 MDP 过程中第一步之后的部分奖励（discounted rewards）的期望总和（expected sum）。

贝尔曼等式（Bellman’s equations）可以有效地解出 Vπ。尤其是在一个有限状态的 MDP 过程中，即 (|S| < ∞)，我们可以把每个状态 s 对应的 V π (s) 的方程写出来。这样就得到了一系列的 |S | 个线性方程，有 |S | 个变量（也就是对应每个状态的未知的 Vπ(s) ），这些 Vπ(s) 都很容易解出来。

然后可以定义出**最优值函数（optimal value function）**



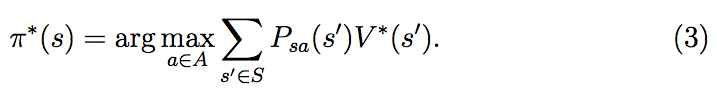
换一种说法，这个值也就是能用任意一种策略函数（policy）来获得的，最佳的可能部分奖励（discounted rewards）的期望总和（expected sum）。另外对于最优值函数（optimal value function），也有一个版本的贝尔曼等式（Bellman’s equations）：



上面这个等式中的第一项，还是跟之前一样的，还是即时奖励函数值。第二项是在采取了动作 a 之后的所有动作 a 的部分奖励（discounted rewards）的未来期望总和（expected future sum）的最大值。要确保理解这个等式，并且要明白为什么这个等式有意义。

（译者注：抱歉，这里的这个 discounted rewards 弄得我不知道怎么翻译才顺，意思表达得很狗，非常抱歉。）

另外还定义了一个策略函数（policy） π∗ : S → A，如下所示



注意，这里的 π∗(s) 给出的动作 a 给出的在上面等式（2）当中能够使 “max” 项取最大值。对于每个状态 s 和每个策略函数（policy）π，都有：



上面的第一个等式关系表明，对应策略函数（policy） π∗的值函数（value function）V π∗ 等于对于每个状态 s 的最优值函数 V ∗。右边的不等式则表明，π∗ 的值至少也等于任意其他策略函数的值。也就是说，上面在等式（3）当中定义的这个 π∗ 就是最佳策略函数（optimal policy）。

注意，这个 π∗ 有一个有趣的特性，它是所有状态 s 下的最佳策略。具体来讲，并不是说只是如果从某个状态 s 开始 MDP 过程，这个 π∗ 是对应这个状态的最佳策略，而如果从某个别的状态 s′ 开始就有其他的最佳策略。而是对于所有的状态 s，都是同样的一个策略函数 π∗ 能够使得等式（1）中的项目取得最大值。这也就意味着无论 MDP 过程的初始状态（initial state）如何，都可以使用同样的策略函数 π∗。

# 2 值迭代（Value iteration）和策略迭代（policy iteration）

We now describe two efficient algorithms for solving finite-state MDPs. For now, we will consider only MDPs with finite state and action spaces (|S| < ∞, |A| < ∞).

The first algorithm, value iteration, is as follows:

1. For each state s, initialize V (s) := 0.

2. Repeat until convergence {

For every state, update 

}

This algorithm can be thought of as repeatedly trying to update the estimated value function using Bellman Equations (2).

There are two possible ways of performing the updates in the inner loop of the algorithm. In the first, we can first compute the new values for V (s) for every state s, and then overwrite all the old values with the new values. This is called a synchronous update. In this case, the algorithm can be viewed as implementing a “Bellman backup operator” that takes a current estimate of the value function, and maps it to a new estimate. (See homework problem for details.) Alternatively, we can also perform asynchronous updates. Here, we would loop over the states (in some order), updating the values one at a time.

Under either synchronous or asynchronous updates, it can be shown that value iteration will cause V to converge to V ∗ . Having found V ∗ , we can then use Equation (3) to find the optimal policy.

Apart from value iteration, there is a second standard algorithm for finding an optimal policy for an MDP. The policy iteration algorithm proceeds as follows:

1. Initialize π randomly.

2. Repeat until convergence {

(a) Let V := V π.

(b) For each state s, let 

}

Thus, the inner-loop repeatedly computes the value function for the current policy, and then updates the policy using the current value function. (The policy π found in step (b) is also called the policy that is greedy with respect to V .) Note that step (a) can be done via solving Bellman’s equations as described earlier, which in the case of a fixed policy, is just a set of |S| linear equations in |S| variables.

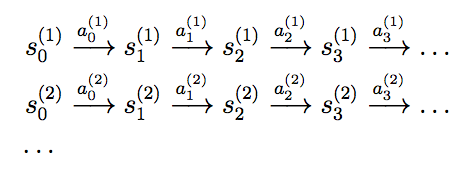
After at most a finite number of iterations of this algorithm, V will converge to V ∗, and π will converge to π∗.

Both value iteration and policy iteration are standard algorithms for solving MDPs, and there isn’t currently universal agreement over which algorithm is better. For small MDPs, policy iteration is often very fast and converges with very few iterations. However, for MDPs with large state spaces, solving for V π explicitly would involve solving a large system of linear equations, and could be difficult. In these problems, value iteration may be preferred. For this reason, in practice value iteration seems to be used more often than policy iteration.

# 3 Learning a model for an MDP

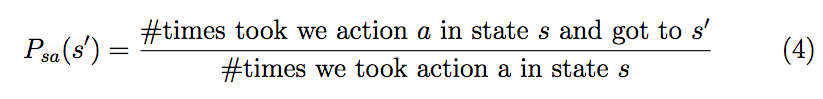
So far, we have discussed MDPs and algorithms for MDPs assuming that the state transition probabilities and rewards are known. In many realistic problems, we are not given state transition probabilities and rewards explicitly, but must instead estimate them from data. (Usually, S, A and γ are known.)

For example, suppose that, for the inverted pendulum problem (see problem set 4), we had a number of trials in the MDP, that proceeded as follows:



Here, si(j) is the state we were at time i of trial j, and ai (j) is the corresponding action that was taken from that state. In practice, each of the trials above might be run until the MDP terminates (such as if the pole falls over in the inverted pendulum problem), or it might be run for some large but finite number of timesteps.

Given this “experience” in the MDP consisting of a number of trials, we can then easily derive the maximum likelihood estimates for the state transition probabilities:



Or, if the ratio above is “0/0”—corresponding to the case of never having taken action a in state s before—the we might simply estimate Psa(s′) to be 1/|S|. (I.e., estimate Psa to be the uniform distribution over all states.)

Note that, if we gain more experience (observe more trials) in the MDP, there is an efficient way to update our estimated state transition probabilities using the new experience. Specifically, if we keep around the counts for both the numerator and denominator terms of (4), then as we observe more trials, we can simply keep accumulating those counts. Computing the ratio of these counts then given our estimate of Psa.

Using a similar procedure, if R is unknown, we can also pick our estimate of the expected immediate reward R(s) in state s to be the average reward observed in state s.

Having learned a model for the MDP, we can then use either value iteration or policy iteration to solve the MDP using the estimated transition probabilities and rewards. For example, putting together model learning and value iteration, here is one possible algorithm for learning in an MDP with unknown state transition probabilities:

1. Initialize π randomly.

2. Repeat {

(a) Execute π in the MDP for some number of trials.

(b) Using the accumulated experience in the MDP, update our estimates for Psa (and R, if applicable).

(c) Apply value iteration with the estimated state transition probabilities and rewards to get a new estimated value function V .

(d) Update π to be the greedy policy with respect to V .

}

We note that, for this particular algorithm, there is one simple optimization that can make it run much more quickly. Specifically, in the inner loop of the algorithm where we apply value iteration, if instead of initializing value iteration with V = 0, we initialize it with the solution found during the previous iteration of our algorithm, then that will provide value iteration with a much better initial starting point and make it converge more quickly.

# 4 Continuous state MDPs

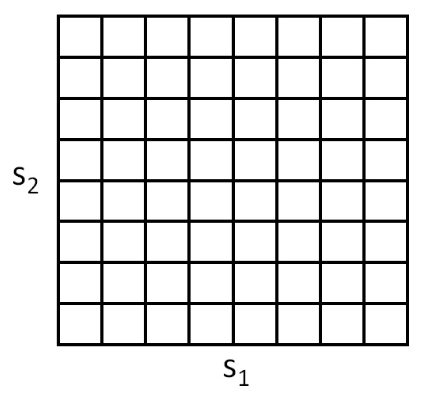
So far, we’ve focused our attention on MDPs with a finite number of states. We now discuss algorithms for MDPs that may have an infinite number of states. For example, for a car, we might represent the state as (x,y,θ,x ̇,y ̇,θ ̇), comprising its position (x, y); orientation θ; velocity in the x and y directions x ̇ and y ̇; and angular velocity θ ̇. Hence, S = R6 is an infinite set of states, because there is an infinite number of possible positions and orientations for the car.2 Similarly, the inverted pendulum you saw in PS4 has states , where θ is the angle of the pole. And, a helicopter flying in 3d space has states of the form, where here the roll φ, pitch θ, and yaw ψ angles specify the 3d orientation of the helicopter. In this section, we will consider settings where the state space is S = Rn, and describe ways for solving such MDPs.

2Technically, θ is an orientation and so the range of θ is better written θ ∈ [−π, π) than θ ∈ R; but for our purposes, this distinction is not important.

## 4.1 Discretization

Perhaps the simplest way to solve a continuous-state MDP is to discretize the state space, and then to use an algorithm like value iteration or policy iteration, as described previously.

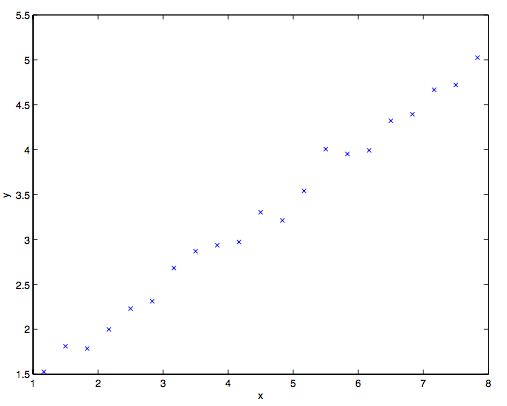
For example, if we have 2d states (s1,s2), we can use a grid to discretize the state space:



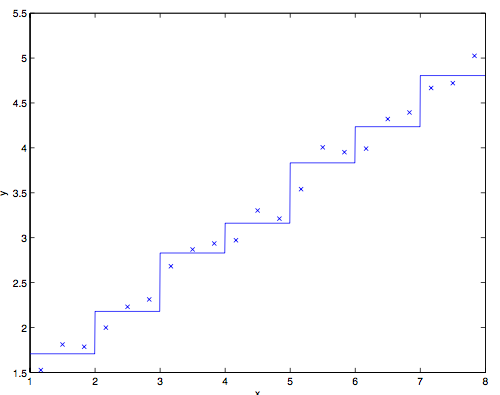
Here, each grid cell represents a separate discrete state s ̄. We can then approximate the continuous-state MDP via a discrete-state one (S ̄, A, {Ps ̄a}, γ, R), where S ̄ is the set of discrete states, {Ps ̄a} are our state transition probabilities over the discrete states, and so on. We can then use value iteration or policy iteration to solve for the V ∗(s ̄) and π∗(s ̄) in the discrete state MDP (S ̄, A, {Ps ̄a}, γ, R). When our actual system is in some continuous-valued state s ∈ S and we need to pick an action to execute, we compute the corresponding discretized state s ̄, and execute action π∗(s ̄).

This discretization approach can work well for many problems. However, there are two downsides. First, it uses a fairly naive representation for V ∗ (and π∗). Specifically, it assumes that the value function is takes a constant value over each of the discretization intervals (i.e., that the value function is piecewise constant in each of the gridcells).

To better understand the limitations of such a representation, consider a supervised learning problem of fitting a function to this dataset:



Clearly, linear regression would do fine on this problem. However, if we instead discretize the x-axis, and then use a representation that is piecewise constant in each of the discretization intervals, then our fit to the data would look like this:



This piecewise constant representation just isn’t a good representation for many smooth functions. It results in little smoothing over the inputs, and no generalization over the different grid cells. Using this sort of representation, we would also need a very fine discretization (very small grid cells) to get a good approximation.

A second downside of this representation is called the curse of dimensionality. Suppose S = Rn, and we discretize each of the n dimensions of the state into k values. Then the total number of discrete states we have is kn. This grows exponentially quickly in the dimension of the state space n, and thus does not scale well to large problems. For example, with a 10d state, if we discretize each state variable into 100 values, we would have 10010 = 1020 discrete states, which is far too many to represent even on a modern desktop computer.

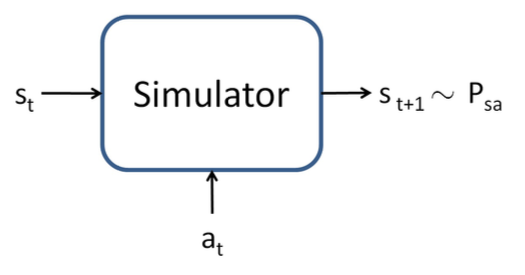
As a rule of thumb, discretization usually works extremely well for 1d and 2d problems (and has the advantage of being simple and quick to implement). Perhaps with a little bit of cleverness and some care in choosing the discretization method, it often works well for problems with up to 4d states. If you’re extremely clever, and somewhat lucky, you may even get it to work for some 6d problems. But it very rarely works for problems any higher dimensional than that.

## 4.2 Value function approximation

We now describe an alternative method for finding policies in continuous- state MDPs, in which we approximate V ∗ directly, without resorting to discretization. This approach, called value function approximation, has been successfully applied to many RL problems.

### 4.2.1 Using a model or simulator

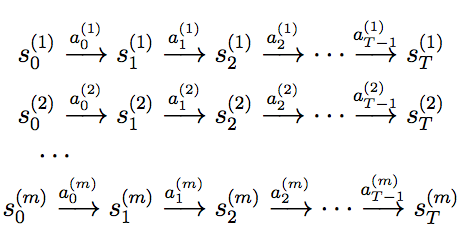
To develop a value function approximation algorithm, we will assume that we have a model, or simulator, for the MDP. Informally, a simulator is a black-box that takes as input any (continuous-valued) state st and action at, and outputs a next-state st+1 sampled according to the state transition probabilities Pstat :



There’re several ways that one can get such a model. One is to use physics simulation. For example, the simulator for the inverted pendulum in PS4 was obtained by using the laws of physics to calculate what position and orientation the cart/pole will be in at time t + 1, given the current state at time t and the action a taken, assuming that we know all the parameters of the system such as the length of the pole, the mass of the pole, and so on. Alternatively, one can also use an off-the-shelf physics simulation software package which takes as input a complete physical description of a mechanical system, the current state st and action at, and computes the state st+1 of the system a small fraction of a second into the future.3

3Open Dynamics Engine (http://www.ode.com) is one example of a free/open-source physics simulator that can be used to simulate systems like the inverted pendulum, and that has been a reasonably popular choice among RL researchers.

An alternative way to get a model is to learn one from data collected in the MDP. For example, suppose we execute m trials in which we repeatedly take actions in an MDP, each trial for T timesteps. This can be done picking actions at random, executing some specific policy, or via some other way of choosing actions. We would then observe m state sequences like the following:

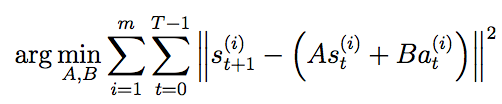


We can then apply a learning algorithm to predict st+1 as a function of st and at.

For example, one may choose to learn a linear model of the form



using an algorithm similar to linear regression. Here, the parameters of the model are the matrices A and B, and we can estimate them using the data collected from our m trials, by picking



(This corresponds to the maximum likelihood estimate of the parameters.) Having learned A and B, one option is to build a deterministic model, in which given an input st and at, the output st+1 is exactly determined. Specifically, we always compute st+1 according to Equation (5). Alternatively, we may also build a stochastic model, in which st+1 is a random function of the inputs, by modelling it as



where here εt is a noise term, usually modeled as εt ∼ N (0, Σ). (The covariance matrix Σ can also be estimated from data in a straightforward way.)

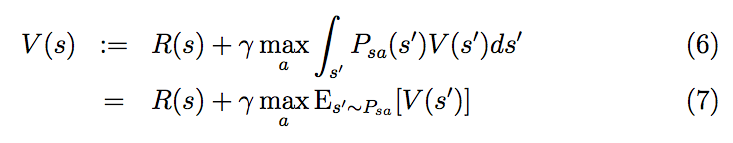
Here, we’ve written the next-state st+1 as a linear function of the current state and action; but of course, non-linear functions are also possible. Specifically, one can learn a model st+1 = Aφs(st) + Bφa(at), where φs and φa are some non-linear feature mappings of the states and actions. Alternatively, one can also use non-linear learning algorithms, such as locally weighted linear regression, to learn to estimate st+1 as a function of st and at. These approaches can also be used to build either deterministic or stochastic simulators of an MDP.

### 4.2.2 Fitted value iteration

We now describe the fitted value iteration algorithm for approximating the value function of a continuous state MDP. In the sequel, we will assume that the problem has a continuous state space S = Rn, but that the action space A is small and discrete.4

4In practice, most MDPs have much smaller action spaces than state spaces. E.g., a car has a 6d state space, and a 2d action space (steering and velocity controls); the inverted pendulum has a 4d state space, and a 1d action space; a helicopter has a 12d state space, and a 4d action space. So, discretizing ths set of actions is usually less of a problem than discretizing the state space would have been.

Recall that in value iteration, we would like to perform the update



(In Section 2, we had written the value iteration update with a summation rather than an integral over states; the new notation reflects that we are now working in continuous states rather than discrete states.) The main idea of fitted value iteration is that we are going to approximately carry out this step, over a finite sample of states s(1), . . . , s(m). Specifically, we will use a supervised learning algorithm—linear regression in our description below—to approximate the value function as a linear or non-linear function of the states:



Here, φ is some appropriate feature mapping of the states. For each state s in our finite sample of m states, fitted value iteration will first compute a quantity y(i), which will be our approximation to R(s)+γmaxaEs′∼Psa[V(s′)](the right hand side of Equation7). Then, it will apply a supervised learning algorithm to try to get V (s) close to R(s) + γ maxa Es′∼Psa [V (s′)] (or, in other words, to try to get V (s) close to

y(i)). In detail, the algorithm is as follows:

1. Randomly sample m states s(1), s(2), . . . s(m) ∈ S. 2. Initialize θ := 0.

3. Repeat {

For i = 1, . . . , m {

For each action a ∈ A {

Sample s′1, . . . , s′k ∼ Ps(i)a (using a model of the MDP).

Set 

// Hence, q(a) is an estimate of 

Set y(i) = maxa q(a).

// Hence, y(i) is an estimate of.

}

// In the original value iteration algorithm (over discrete states)

// we updated the value function according to V (s(i)) = y(i).

// In this algorithm, we want V (s(i)) ≈ y(i), which we’ll achieve

// using supervised learning (linear regression).

Set 

}

Above, we had written out fitted value iteration using linear regression as the algorithm to try to make V (s(i)) close to y(i). That step of the algorithm is completely analogous to a standard supervised learning (regression) problem in which we have a training set (x(1),y(1)),(x(2),y(2)),...,(x(m),y(m)), and want to learn a function mapping from x to y; the only difference is that here s plays the role of x. Even though our description above used linear regression, clearly other regression algorithms (such as locally weighted linear regression) can also be used.

Unlike value iteration over a discrete set of states, fitted value iteration cannot be proved to always to converge. However, in practice, it often does converge (or approximately converge), and works well for many problems. Note also that if we are using a deterministic simulator/model of the MDP, then fitted value iteration can be simplified by setting k = 1 in the algorithm. This is because the expectation in Equation (7) becomes an expectation over a deterministic distribution, and so a single example is sufficient to exactly compute that expectation. Otherwise, in the algorithm above, we had to draw k samples, and average to try to approximate that expectation (see the definition of q(a), in the algorithm pseudo-code).

Finally, fitted value iteration outputs V , which is an approximation to V∗. This implicitly defines our policy. Specifically, when our system is in some state s, and we need to choose an action, we would like to choose the action

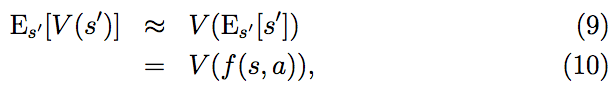


The process for computing/approximating this is similar to the inner-loop of fitted value iteration, where for each action, we sample s′1,...,s′k ∼ Psa to approximate the expectation. (And again, if the simulator is deterministic, we can set k = 1.)

In practice, there’re often other ways to approximate this step as well. For example, one very common case is if the simulator is of the form st+1 = f(st,at) + εt, where f is some determinstic function of the states (such as f(st,at) = Ast + Bat), and ε is zero-mean Gaussian noise. In this case, we can pick the action given by



In other words, here we are just setting εt = 0 (i.e., ignoring the noise in the simulator), and setting k = 1. Equivalently, this can be derived from Equation (8) using the approximation



where here the expection is over the random s′ ∼ Psa. So long as the noise terms εt are small, this will usually be a reasonable approximation.

However, for problems that don’t lend themselves to such approximations, having to sample k|A| states using the model, in order to approximate the expectation above, can be computationally expensive.