Chapter 18

Fitting Cost Approximation Architectures

[1]The dynamic programming methods discussed in the first part of the course and in chapter 14 require an explicit model of the stage costs and transition probabilities of the system. Many systems of interest however are too complicated to make the development of such a mathematical model an attractive or even feasible approach (think of a complex robot, a financial market, a large communication network, a transportation network, etc.). It is often the case however that we can simulate these systems, or observe their response to controls in real-time. One can of course try to construct and estimate the model parameters using the simulator, and then apply the classical DP techniques, but this is not the only possible approach. For example, the Q-learning method discussed in chapter 16 never builds an explicit model of the system and does not include such a preliminary estimation phase. In this chapter we discuss methods that use simulation to estimate the cost of a *given fixed policy*, without explicitly estimating the transition probabilities, and deal with the problem of large and complex state spaces by using an approximation architecture. These methods can be used to compare different policies, or to search for an optimal policy in the context of an approximate policy iteration scheme.

18.1 Simulation-Based Approximate Policy Iteration

Recall that the approximate policy iteration method (see section 7.2) follows the PI scheme, but uses an approximation of the cost of the current policy μ in the policy improvement step instead of the exact cost (and potentially an approximate policy improvement step as well). Each policy improvement step corresponds to the computation of a one-step lookahead policy, or approximate rollout policy, as described in section 14.2. There we already mentioned that the cost approximation could be obtained from an approximation architecture using a simulator. A simulation-based implementation of the approximate

[1]This version: November 8 2009

policy iteration algorithm follows the cycle below, starting from some initial policy μ:

- Use a *simulator* and a *cost approximation algorithm* to obtain an approximation $\tilde{J}(\cdot; r)$ of the cost of J_μ.

- A *decision generator* uses the values of $\tilde{J}(\cdot; r)$ to generate a new policy $\bar{\mu}$ according to an (possibly approximate) policy improvement step

$$\bar{\mu}(x) \in \arg \min_{u \in \mathsf{U}(x)} \{c(x,u) + E_w[\tilde{J}(f(x,u,w); r)|x]\}, \forall x \in \mathsf{X}. \tag{18.1}$$

  Then repeat, replacing μ by $\bar{\mu}$ in the first step.

The two steps above are combined in the sense that given a current approximation $\tilde{J}(\cdot; r)$ of the cost of μ, $\bar{\mu}$ is simultaneously generated according to (18.1) and driving the simulator to estimate the next approximation $\tilde{J}(\cdot; \bar{r})$ of $J_{\bar{\mu}}$. At some point, we are satisfied with the approximation, and we replace $\tilde{J}(\cdot; r)$ by $\tilde{J}(\cdot; \bar{r})$ in (18.1).

Optimistic Policy Iteration

In optimistic versions of (approximate) policy iteration, we replace the value function $\tilde{J}(\cdot; r)$ by $\tilde{J}(\cdot; \bar{r})$ much sooner, even if that means an innacurate approximation of $J_{\bar{\mu}}$. That is, we don't wait until convergence of policy evaluation. Such algorithms, used often in practice especially in the artificial intelligence literature, are not very well understood and their behavior can be complex. See [BT96].

Approximate Policy Iteration using Q-Factors

We can work with the Q-factors instead of the value function. Assuming that we have an approximation of the Q-factors $Q_\mu(i,u)$ of the current policy

$$\tilde{Q}(x,u;r) \approx Q_\mu(x,u) = c(x,\mu(x)) + E_w[J_\mu(f(x,\mu(x),w)|x],$$

we compute the next policy according to the policy improvement step

$$\bar{\mu}(x) \in \arg \min_{u \in \mathsf{U}(x)} \tilde{Q}(x,u;r).$$

Any method used to construct the cost function approximations $\tilde{J}(x; r)$ for a policy μ can be used to compute the Q-factor approximations $\tilde{Q}(x,u;r)$ for Q_μ. Just apply the method to the Markov chain with states the pairs of state-actions (x,u) and transition probabilities from (x,u) to (x',u') equal to $p_{xx'}(u)$ if $u' = \mu(x)$, and 0 otherwise.

State Exploration Issues

A difficulty with simulation-based policy evaluation is that we need to generate cost samples using a policy μ, under which certain states might be unlikely to occur (think for example of a deterministic problem!). With few cost samples generated for these states, the cost estimation is then typically highly inaccurate in the corresponding part of the state space. This can in turn cause serious errors in the calculation of the improved policy $\bar{\mu}$. In order to guarantee adequate coverage of the state space, we can restart the simulation from a rich enough set of initial states, or occasionally deviate from the policy μ and introduce extra randomly selected controls. An issue with the introduction of these random actions is that applying directly the methods below then estimate the cost of a different policy. See [Ber07b, section 6.3.7] on how to correct this problem.

18.2 Direct Approximation Methods for Policy Evaluation

[2]In order to able to implement the simulation based PI scheme of section 18.1, or more simply a one-step lookahead policy, the only missing element at this point is a cost approximation algorithm, which adjusts the parameters (weights) of the approximation architecture $J(\cdot; r)$ to obtain a good estimate of the cost J_μ of the policy μ currently used. We will mostly discuss linear approximation architectures as first encountered in section 14.2 i.e.,

$$J(x; r) = \langle r, \phi(x) \rangle = \sum_{k=1}^{m} r_k \phi_k(x),$$

or for finite-state problems

$$J = \Phi r, \quad \text{with } \Phi = \Big[\phi_1 \Big| \dots \Big| \phi_m \Big].$$

Hence a state x is characterized by a restricted set of m numbers $\phi_i(x), i = 1 \dots, m$, called features. The function $\phi : \mathsf{X} \to \mathbb{R}^m$ maps the state space into a finite dimensional space. For a finite state space of size n, we typically take m much smaller than n and hope (or know by other means) that a small number of these functions ϕ_i will indeed be enough to approximate J_μ reasonably well.

Remark. There are other potentially interesting approximation architectures, such as *neural networks* (NN), described for example in [BT96, Sut98], which were particularly popular at the time when approximate DP and reinforcement learning encountered their first successes (see e.g. the backgammon player of Tesauro developped in the early 1990's, which played at the level of the best human players). The issue with nonlinear approximation architecture is that

[2]According to [Ber07a], there methods are "less popular" than the ones presented in section 18.3. I suppose this means that their performance is not as good somehow.

the optimization of the weights becomes in general more difficult and can get stuck in local minima. Moreover, there seems to be no clear theoretical advantage of using them instead of linear architectures (note that the functions ϕ_i are already supposed to capture the nonlinearities of J_μ). The goal of developing architectures that perform well for all problems is most likely to be impossible to achieve. A particular architecture will perform well on some problems, and not as well in other problems with completely different characteristics. I think that it is more interesting to find good approximation architectures for restricted classes of problems, e.g. queueing network problems, or chess games, or Tetris, etc. I repeat that the point is not to have a sequence of functions that will converge to the policy cost as $m \to \infty$ (you can pretty much always do this, e.g. by taking an orthonormal basis if your function J_μ leaves in a nice Hilbert space), it is to encode J_μ in as few coefficients r_i as possible for better performance with limited computational and memory capabilities. We encounter the same problem for example in many signal processing or statistic applications. For example, it was found that it is more efficient to use wavelet bases to encode images instead of Fourier bases. However, if you know that your signal is a sinusoid, there is no reason to use wavelets... The point is that it is probably necessary to have a good insight into your particular application in order to develop an efficient approximation architecture (think of say computer chess players).

The most straightforward way to find an approximation $J_r := J(\cdot; r)$ that best matches J_μ, referred as the *direct approximation* method, is to solve

$$\min_r \; f(r) := \|J_\mu - J_r\|, \tag{18.2}$$

where $\|\cdot\|$ is some norm, or for finite-state problems

$$\min_r \; \|J_\mu - \Phi r\|. \tag{18.3}$$

If $\|\cdot\|$ is a weighted Euclidean norm, i.e.

$$\|x\|_{2,W} = \sqrt{x^T W x}, \; W \succ 0,$$

then (18.3) is a least-square problem. Of course, the problem here is that J_μ is not available. It can only be estimated through simulations, which provide only noisy samples of the true function. Now recall that although a close form exist for the standard least-square approximation,

$$\begin{aligned} x^* \in \arg\min_x \|y - Ax\|_{2,W} &\iff x^* \in \arg\min_x \; \frac{1}{2}(y - Ax)^T W (y - Ax) \\ &\iff (A^T W A) x^* = A^T W y, \\ &\iff x^* = (A^T W A)^{-1} A^T W y \text{ (if } A \text{ has indep. cols.)}, \end{aligned}$$

in practice you might want instead to solve large scale problems directly using a modern (quadratic) optimization solver[3]. Here we are in this set-up except

[3]other norms can then be considered for which a closed form solution does not exist, such as ℓ_1

that y is composed of noisy measurements (of some of the cost entries $J_\mu(i)$), as in the classical linear regression problem in statistics. In our applications, more simulation samples become progressively available, and simulation noise enters the computation of the gradient of f in (18.2). Hence the optimization over the parameter r typically leads to some form of stochastic gradient descent, see section 15.2.

So let us suppose that we wish to solve problem (18.3) for a finite state problem and a given policy μ (μ would be the policy $\bar{\mu}$ of section 18.1). Recall that for a finite-state space $\mathsf{X} = \{1, \ldots, n\}$, J_μ is a vector $[J_\mu(1), \ldots, J_\mu(n)]^T$ of size n, and we cannot expect to obtain values of J_μ for all of these states in practice. Instead, we evaluate $J_\mu(i)$ for a few representative states $i \in \tilde{\mathsf{X}}$, for example using simulations. If the dynamics of the system are subject to noise, we cannot obtain the value $J_\mu(i)$ exactly, which is by definition the expected value of the cost of μ starting from state i. Instead we can obtain using repeated simulations several noisy versions of $J_\mu(i)$, denoted $g_s(i), s = 1, \ldots, K(i)$, if $K(i)$ simulations are performed starting from state i. Then we solve the least-squares problem

$$\min_r \sum_{i \in \tilde{\mathsf{X}}} \sum_{s=1}^{K(i)} [J(i;r) - g_s(i)]^2 = \min_r \|\tilde{J}_r - g\|^2 \qquad (18.4)$$

Here $\tilde{J}_r$ is just the vector with coordinate $\tilde{J}(i;r)$ repeated $K(i)$ times, for $i \in \tilde{\mathsf{X}}$. We can solve (18.4) using gradient descent type methods.

Let us start in state i_0 at time $k = 0$, and consider a portion $i_0, i_1, \ldots, i_N$ with N transitions of a simulated trajectory using the stationary policy μ, also called a *batch*. Note that some states might be repeated in that simulation. Then we can view each of the tail costs

$$g_k(i_k) = \sum_{t=k}^{N-1} \alpha^{t-k} c(i_t, \mu(i_t), i_{t+1}), k = 0, \ldots, N-1, \qquad (18.5)$$

as a cost sample, one per state $i_0, \ldots, i_{N-1}$ along the trajectory, which can be used as $g_s(i_k)$ in (18.4). Although the quality of these cost estimates decreases as we approach N, this technique allows us to extract more information out of each trajectory simulation, compared to throwing away the tail costs (18.5) for $k \geq 1$.

Next we want to use a gradient descent method to minimize (18.4). Recall that such a method to find a minimum

$$\min_r F(r),$$

in the standard set-up where F is deterministic is an iterative method which updates r following

$$r_{l+1} = r_l - \gamma_l \nabla F(r_l),$$

where γ_k is a sequence of positive, usually decreasing, step-sizes. Then r_k converges to a critical point of F, and to a global minimum if F is convex. You can also replace the gradient ∇F by a subgradient if F is nonsmooth. So let us follow this method in order to minimize

$$\min_r \sum_{k=0}^{N-1} \frac{1}{2} \left(\tilde{J}(i_k; r) - g_k(i_k) \right)^2 . \tag{18.6}$$

That is, the representative states $\tilde{\mathsf{X}}$ are chosen to be the successive states along the simulated trajectory. This leads us to the iterations

$$r_{l+1} := r_l - \gamma_l \sum_{k=0}^{N-1} \nabla_r \tilde{J}(i_k; r_l) \left(\tilde{J}(i_k; r_l) - g_k(i_k) \right), \tag{18.7}$$

where ∇_r denotes the gradient with respect to r. The update (18.7) is performed after simulating the whole portion $(i_0, \ldots, i_N)$ of the trajectory and recording the tail costs $g_k(i_k)$. This method is called a *batch gradient method*. Note that we need to be able to evaluate the gradient of $r \mapsto \tilde{J}(x; r)$, which is particularly simple for linear approximation architectures, see below.

Now in the standard gradient method, after processing this batch, the update (18.7) would be repeated using the same batch until convergence of r is obtained and we get a solution to (18.6). In that case however, it would be necessary to use a large value of N in order to obtain sufficiently many cost samples for a large number of representative states. However, increasing N makes the computation of the update (18.7) computationally more difficult. Hence in a more general method, we can keep N relatively small and use different batches after one or more iterations (18.7), in order to increase the set of states considered (i.e. sufficiently "explore" the state space). The length N of the different batches may also vary, and the batches may overlap. Overall, the least-square optimization provides better approximations for the states that arise more frequently in the batches used. Under some reasonable assumptions, we typically expect convergence of r_k to a minimum (local if the architecture used is not linear and the resulting optimization problem non-convex). For this, the step size is often required to be gradually reduced, and a popular choice is to take γ_l proportional to $1/l$ while processing the l^{th} batch, although some experimentation with different step sizes is usually necessary. In any case, the rate of convergence is often very slow and depends on the choice of r_0, the number of states, the dynamics of the underlying Markov chain, simulation errors, the stepsize choice, etc.

A variant of the gradient method, called *incremental*, is suitable for use with very long batches, including the possibility of a single very long simulated trajectory, viewed as a single batch. In general, this method is more flexible and has a somewhat better performance than the batch method, so it can be adopted by default. It does not wait for N transitions to happen before updating r according to (18.7), but performs an update after each transition.

Rewrite (18.7), by interchanging the sums in the second term, as

$$\begin{aligned}
&r_{l+1} \\
&= r_l - \gamma_l \sum_{k=0}^{N-1} \nabla_r \tilde{J}(i_k; r_l) \left(\tilde{J}(i_k; r_l) - \sum_{t=k}^{N-1} \alpha^{t-k} c(i_t, \mu(i_t), i_{t+1}) \right) \qquad (18.8) \\
&= r_l - \sum_{k=0}^{N-1} \gamma_l \nabla_r \tilde{J}(i_k; r_l)\, \tilde{J}(i_k; r_l) + \gamma_l \sum_{k=0}^{N-1} \sum_{t=k}^{N-1} \alpha^{t-k}\, \nabla_r \tilde{J}(i_k; r_l)\, c(i_t, \mu(i_t), i_{t+1}) \\
&= r_l - \sum_{k=0}^{N-1} \gamma_l \nabla_r \tilde{J}(i_k; r_l)\, \tilde{J}(i_k; r_l) + \gamma_l \sum_{t=0}^{N-1} \left(\sum_{k=0}^{t} \alpha^{t-k}\, \nabla_r \tilde{J}(i_k; r_l) \right) c(i_t, \mu(i_t), i_{t+1}) \\
&= r_l - \sum_{t=0}^{N-1} \gamma_l \left(\nabla_r \tilde{J}(i_t; r_l)\, \tilde{J}(i_t; r_l) - \left(\sum_{k=0}^{t} \alpha^{t-k}\, \nabla_r \tilde{J}(i_k; r_l) \right) c(i_t, \mu(i_t), i_{t+1}) \right). \qquad (18.9)
\end{aligned}$$

Now the incremental gradient algorithm is based on the last expression, but changes r during the processing of a batch. Namely, after each transition (i_k, i_{k+1}):

1. We evaluate the gradient $\nabla_r \tilde{J}(i_k; r_k)$ at the current value of r_k of r.
2. We update r as in (18.10)

$$r_{k+1} = r_k - \gamma_k \left(\nabla_r \tilde{J}(i_k; r_k)\, \tilde{J}(i_k; r_k) - \left(\sum_{t=0}^{k} \alpha^{k-t}\, \nabla_r \tilde{J}(i_t; r_k) \right) c(i_k, \mu(i_k), i_{k+1}) \right). \qquad (18.10)$$

Hence after N iteration the only difference between (18.10) and (18.9) is that r was continuously updated during the processing of the batch in the incremental version. Note that the gradients in (18.10) are computed at the most recent value of r whereas in the batch method (18.9) they are evaluated at the value of r fixed at the beginning of the batch. Note also that in the sum (18.10), it might make more sense to use $\nabla_r \tilde{J}(i_t; r_t)$ instead of $\nabla_r \tilde{J}(i_t; r_k)$ in order to avoid recalculating the gradients and obtain a recursive algorithm. We had the same issue when discussing Q-learning, and this will be done in the next paragraph. Here however I'm following the apparent convention in [Ber07a, p.337]. Finally, from the form of the update (18.10) it is clear that the batch length N is irrelevant and in particular the updates can be performed along a single long trajectory. A state i_k is essentially weighted in the least squares method proportionally to the frequency of its occurrence in the simulations, reflecting the larger confidence in the cost value estimate for this state. However, as the trajectory grows, the computation of the sum in (18.10) becomes difficult.

Implementation Using Temporal Differences

We can rewrite the batch and incremental gradient updates in a cleaner way using the *temporal differences* (TD), defined as

$$q_{l,k} = \tilde{J}(i_k; r_l) - \alpha \tilde{J}(i_{k+1}; r_l) - c(i_k, \mu(i_k), i_{k+1}),\ k = 0, \ldots, N-2, \tag{18.11}$$

$$q_{l,N-1} = \tilde{J}(i_{N-1}; r_l) - c(i_{N-1}, \mu(i_{N-1}), i_N). \tag{18.12}$$

Next remark that the expression in parenthesis in (18.8) can be rewritten

$$q_{l,k} + \alpha q_{l,k+1} + \ldots + \alpha^{N-1-k} q_{l,N-1} = \sum_{t=k}^{N-1} \alpha^{t-k} q_{l,t}.$$

Hence we can rewrite (18.8), again using a simple interchange of sums

$$r_{l+1} = r_l - \gamma_l \sum_{t=0}^{N-1} q_{l,t} \left(\sum_{k=0}^{t} \alpha^{t-k} \nabla_r \tilde{J}(i_k; r_l) \right).$$

Now this batch iteration, instead of being performed once at the end of the batch, can be performed in steps with $r_{l+1}^0 = r_l$ and after observing the transition (i_t, i_{t+1}) computing

$$r_{l+1}^{t+1} := r_l^t - \gamma_l q_{l,t} \left(\sum_{k=0}^{t} \alpha^{t-k} \nabla_r \tilde{J}(i_k; r_l) \right). \tag{18.13}$$

Letting

$$z_{l,t} = \sum_{k=0}^{t} \alpha^{t-k} \nabla_r \tilde{J}(i_k; r_l)$$

the expression in parenthesis in (18.13) can be updated as

$$z_{l,t+1} = \alpha z_{l,t} + \nabla_r \tilde{J}(i_t + 1; r_l).$$

We then recover $r_{l+1} = r_{l+1}^N$ after N iterations.

Note that in (18.13) the temporal differences $q_{l,t}$ and the gradients are evaluated at $r = r_l$. If instead we evaluate them at a more recent value of r, to get the iterations

$$r_{k+1} = r_k - \gamma_k q_{k,k} \left(\sum_{t=0}^{k} \alpha^{k-t} \nabla_r \tilde{J}(i_t; r_t) \right), \tag{18.14}$$

we obtain a type of incremental gradient method[4]. Here $q_{k,k}$ is evaluated at r_k. Note that in addition to updating r, we can also modify the stepsize γ_k at

[4]this does not seem to be exactly equivalent to (18.10), even if the convention $\nabla_r \tilde{J}(i_t; r_t)$ is used there as described in the previous paragraph. Here the terms in successive TDs do not quite cancel, because we get terms of the form $\alpha J(i_{k+1}; r_k)$ and $\alpha J(i_{k+1}; r_{k+1})$ with different values of r.

each transition. The algorithm (18.14), starting from some vector r_0, is called TD(1). A variant is TD(λ), which uses a parameter $\lambda \in [0,1]$ in the iteration

$$r_{k+1} = r_k - \gamma_k q_k \left(\sum_{t=0}^{k} (\alpha\lambda)^{k-t} \nabla_r \tilde{J}(i_t; r_t) \right), \tag{18.15}$$

where the only modification with respect to TD(1) consists in replacing α by $\alpha\lambda$. For $\lambda = 0$, we obtain TD(0), which has the simple form

$$r_{k+1} = r_k - \gamma_k q_{k,k} \nabla_r \tilde{J}(i_k; r_k). \tag{18.16}$$

The main advantage of using $\lambda < 1$ is that the sum (18.15) generally has smaller variance than for $\lambda = 1$, hence it improves the convergence properties of the method. However, we loose the property of convergence to the optimum parameter r^* and the quality of approximation deteriorates as λ is reduced towards 0.

In fact the only case for which there exists a convergence analysis for $\lambda < 1$ is for the important case of a linear approximation architecture

$$\tilde{J}(i; r) = \phi(i)^T r,\ i = 1, \ldots, n.$$

Then $\nabla_r \tilde{J}(i_t; r_t) = \phi(i_t)$, and the TD($\lambda$) algorithm becomes

$$r_{k+1} = r_k - \gamma_k q_{k,k} \left(\sum_{t=0}^{k} (\alpha\lambda)^{k-t} \phi(i_t) \right), \tag{18.17}$$

with the temporal differences given by

$$q_{k,k} = \phi(i_k)^T r_k - \alpha\phi(i_{k+1})^T r_k - c(i_k, \mu(i_k), i_{k+1}).$$

Again, a more convenient recursion is obtained if we introduce the vector

$$z_k = \sum_{t=0}^{k} (\alpha\lambda)^{k-t} \nabla_r \tilde{J}(i_t; r_t) = \sum_{t=0}^{k} (\alpha\lambda)^{k-t} \phi(i_t),$$

which evolves according to

$$z_k = \alpha\lambda z_{k-1} + \nabla_r \tilde{J}(i_k; r_k) = \alpha\lambda z_{k-1} + \phi(i_k).$$

Then TD(λ) takes the simple form

$$r_{k+1} = r_k - \gamma_k z_k q_{k,k}. \tag{18.18}$$

Finally we can combine the TD methods with optimistic variants of policy iteration. There we replace μ by the approximate rollout policy μ after only a few simulation samples have been processed. In the extreme case, we change the policy after each transition. That is, after transition (i_k, i_{k+1}), set the parameter r to

$$r_{k+1} = r_k - \gamma_k q_{k,k} \sum_{t=0}^{k} (\alpha\lambda)^{k-t} \nabla_r \tilde{J}(i_t; r_t),$$

and generate the next transition (i_{k+1}, i_{k+2}) by simulation using the control

$$\bar{\mu}(i_{k+1}) \in \arg \min_{u \in \mathsf{U}(i_{k+1})} \sum_{j=1}^{n} p_{i_{k+1}j}(u)(c(i_{k+1}, u, j) + \alpha \tilde{J}(j; r_{k+1}).$$