

CHAPTER 6

POLICIES

Perhaps one of the most widely used and poorly understood terms in dynamic programming is *policy*. A simple definition of a policy is:

Definition 6.0.1 A **policy** is a rule (or function) that determines a decision given the available information in state S_t .

The problem with the concept of a policy is that it refers to *any* method for determining an action given a state, and as a result it covers a wide range of algorithmic strategies, each suited to different problems with different computational requirements. Given the vast diversity of problems that fall under the umbrella of dynamic programming, it is important to have a strong command of the types of policies that can be used, and how to identify the best type of policy for a particular decision.

In this chapter, we review a range of policies, grouped under four broad categories:

Myopic policies - These are the most elementary policies. They optimize costs/rewards now, but do not explicitly use forecasted information or any direct representation of decisions in the future. However, they may use tunable parameters to produce good behaviors over time.

Lookahead policies - These policies make decisions now by explicitly optimizing over some horizon by combining some approximation of future information, with some approximation of future actions.

Policy function approximations - These are functions which directly return an action given a state, without resorting to any form of imbedded optimization, and without directly using any forecast of future information.

Value function approximations - These policies, often referred to as *greedy* policies, depend on an approximation of the value of being in a future state as a result of a decision made now. The impact of a decision now on the future is captured purely through a value function that depends on the state that results from a decision now.

These four broad categories span a wide range of strategies for making decisions. Further complicating the design of clean categories is the ability to form hybrids combined of strategies from two or even three of these categories. However, we feel that these four fundamental categories offer a good starting point. We note that the term “value function approximation” is widely used in approximate dynamic programming, while “policy function approximation” is a relatively new phrase, although the idea behind policy function approximations is quite old. We use this phrase to emphasize the symmetry between approximating the value of being in a state, and approximating the action we should take given a state. Recognizing this symmetry will help to synthesize what is currently a disparate literature on solution strategies.

Before beginning our discussion, it is useful to briefly discuss the nature of these four strategies. Myopic policies are the simplest, since they make no explicit attempt to capture the impact of a decision now on the future. Lookahead policies capture the effect of decisions now by explicitly optimizing in the future (perhaps in an approximate way) using some approximation of information that is not known now. A difficulty is that lookahead strategies can be computationally expensive. For this reason, considerable energy has been devoted to developing approximations. The most popular strategy involves developing an approximation $\bar{V}(s)$ of the value around the pre- or post-decision state. If we use the more convenient form of approximating the value function around the post-decision state, we can make decisions by solving

$$a_t = \arg \max_a (C(S_t, a) + \bar{V}(S^{M,a}(S_t, a))). \quad (6.1)$$

If we have a vector-valued decision x_t with feasible region \mathcal{X}_t , we would solve

$$x_t = \arg \max_{x \in \mathcal{X}_t} (C(S_t, x) + \bar{V}(S^{M,x}(S_t, x))). \quad (6.2)$$

Equations (6.1) and (6.2) both represent policies. For example, we can write

$$A^\pi(S_t) = \arg \max_a (C(S_t, a) + \bar{V}(S^{M,a}(S_t, a))). \quad (6.3)$$

When we characterize a policy by a value function approximation, π refers to a specific approximation in a set Π which captures the family of value function approximations. For example, we might use $\bar{V}(s) = \theta_0 + \theta_1 s + \theta_2 s^2$. In this case, we might write the space of policies as Π^{VFA-LR} to represent value function approximation-based policies using linear regression. $\pi \in \Pi^{VFA-LR}$ would represent a particular setting of the vector $(\theta_0, \theta_1, \theta_2)$.

Remark: Some authors like to write the decision function in the form

$$A_t^\pi(S_t) \in \arg \max_a (C(S_t, a) + \bar{V}(S^{M,a}(S_t, a))).$$

The “ \in ” captures the fact that there may be more than one optimal solution, which means the $\arg \max_a(\cdot)$ is really a set. If this is the case, then we need some rule for choosing which member of the set we are going to use. Most algorithms make this choice at random (typically, we find a best solution, and then replace it only when we find a solution that is even better, rather than just as good). In rare cases, we do care. However, if we use $A_t^\pi(S_t) \in \arg \max$ then we have to introduce additional logic to solve this problem. We assume that the “ $\arg \max$ ” operator includes whatever logic we are going to use to solve this problem (since this is typically what happens in practice).

Alternatively, we may try to directly estimate a decision function $A^\pi(S_t)$ (or $X^\pi(S_t)$), which does not have the imbedded \max_a (or \max_x) within the function. We can think of $\bar{V}(S_t^a)$ (where $S_t^a = S^{M,a}(S_t, a)$) as an approximation of the value of being in (post-decision) state S_t^a (while following some policy in the future). At the same time, we can think of $A^\pi(S_t)$ as an approximation of the decision we should make given we are in (pre-decision) state S_t . $\bar{V}(s)$ is almost always a real-valued function. Since we have defined a to always refer to discrete actions, $A^\pi(s)$ is a discrete function, while $X^\pi(s)$ may be a scalar continuous function, or a vector-valued function (discrete or continuous). For example, we might use $X^\pi(s) = \theta_0 + \theta_1 s + \theta_2 s^2$ to compute a decision x directly from a continuous state s .

Whether we are trying to estimate a value function approximation $\bar{V}(s)$, or a decision function $A^\pi(s)$ (or $X^\pi(s)$), it is useful to identify three categories of approximation strategies:

Lookup tables - Also referred to as tabular functions, lookup tables mean that we have a discrete value $\bar{V}(s)$ (or action $A^\pi(s)$) for each discrete state s .

Parametric representations - These are explicit, analytic functions for $\bar{V}(s)$ or $A^\pi(s)$ which generally involve a vector of parameters that we typically represent by θ .

Nonparametric representations - Nonparametric representations offer a more general way of representing functions, but at a price of greater complexity.

We have a separate category for lookup tables because they represent a special case. First, they are the foundation of the field of Markov decision processes on which the original theory is based. Also, lookup tables can be represented as a parametric model (with one parameter per state), but they share important characteristics of nonparametric models (since both focus on the local behavior of functions).

Some authors in the community take the position (sometimes implicitly) that approximate dynamic programming is equivalent to finding value function approximations. While designing policies based on value function approximations arguably remains one of the most powerful tools in the ADP toolbox, it is virtually impossible to create boundaries between a policy based on a value function approximation, and a policy based on direct search, which is effectively equivalent to maximizing a value function, even if we are not directly trying to estimate the value function. Lookahead policies are little more than an alternative method for approximating the value of being in a state, and these are often used as heuristics within other algorithmic strategies.

In the remainder of this chapter, we illustrate the four major categories of policies. For the case of value function and policy function approximations, we draw on the three major ways of representing functions. However, we defer to chapter 8 an in-depth presentation of methods for approximating value functions. Chapter 7 discusses methods for finding policy function approximations in greater detail.

6.1 MYOPIC POLICIES

The most elementary class of policy does not explicitly use any forecasted information, or any attempt to model decisions that might be implemented in the future. In its most basic form, a myopic policy is given by

$$A^{Myopic}(S_t) = \arg \max_a C(S_t, a). \quad (6.4)$$

Myopic policies are actually widely used in resource allocation problems. Consider the dynamic assignment problem introduced in section 2.2.10, where we are trying to assign resources (such as medical technicians) to tasks (equipment that needs to be installed). Let \mathcal{I} be the set of technicians, and let r_i be the attributes of the i^{th} technician (location, skills, time away from home). Let \mathcal{J}_t be the tasks that need to be handled at time t , and let b_j be the attributes of task j , which can include the time window in which it has to be served and the reward earned from serving the task. Finally let $x_{tij} = 1$ if we assign technician i to task j at time t , and let c_{tij} be the contribution of assigning technician i to task j at time t , which captures the revenue we receive from serving the task minus the cost of assigning a particular technician to the task. We might solve our problem at time t using

$$x_t = \arg \max_{x \in \mathcal{X}_t} \sum_{i \in \mathcal{I}} \sum_{j \in \mathcal{J}_t} c_{tij} x_{tij},$$

where the feasible region \mathcal{X}_t is defined by

$$\begin{aligned} \sum_{j \in \mathcal{J}_t} x_{tij} &\leq 1, \quad \forall i \in \mathcal{I}, \\ \sum_{i \in \mathcal{I}} x_{tij} &\leq 1, \quad \forall j \in \mathcal{J}_t, \\ x_{tij} &\geq 0. \end{aligned}$$

Note that we have no difficulty handling a high-dimensional decision vector x_t , since it requires only that we solve a simple linear program. The decision x_t ignores its effect on the future, hence its label as a myopic policy.

6.2 LOOKAHEAD POLICIES

Lookahead policies make a decision now by solving an approximation of the problem over some horizon. These policies are distinguished from value function and policy function approximations by the explicit representation of both future information and future decisions. In this section we review tree search, rollout heuristics and rolling horizon procedures, all of which are widely used in engineering practice.

6.2.1 Tree search

Tree search is a brute force strategy that enumerates all possible actions and all possible outcomes over some horizon, which may be fairly short. As a rule, this can only be used for fairly short horizons, and even then only when the number of actions and outcomes is not too large. Tree search is identical to solving decision trees, as we did in section 2.2.1.

If a full tree search can be performed for some horizon T , we would use the results to pick the action to take right now. The calculations are exact, subject to the single limitation that we are only solving the problem over T time periods rather than the full (and possibly infinite) horizon.

6.2.2 Sparse sampling tree search

If the action space is not too large, we may limit the explosion due to the number of outcomes by replacing the enumeration of all outcomes with a Monte Carlo sample of outcomes. It is possible to actually bound the error from the optimal solution if enough outcomes are generated, but in practice this quickly produces the same explosion in the size of the outcome tree. Instead, this is best viewed as a potential heuristic which provides a mechanism for limiting the number of outcomes in the tree.

6.2.3 Roll-out heuristics

For problems where the number of actions per state is fairly large, we may replace the explicit enumeration of the entire tree with some sort of heuristic policy to evaluate what might happen after we reach a state. Imagine that we are trying to evaluate if we should take action a_0 which takes us to location A . Instead of enumerating the tree out of location A , we might follow some simple policy (perhaps a myopic policy) starting at A and extending a reasonable number of time periods in the future. The value of this trajectory, then, is used purely to develop an estimate of the value of being in location A . This process is repeated for each action a_0 so that we get a rough estimate of the downstream value of each of these actions. From this, we choose the best value for a_0 given these rough estimates. The idea is illustrated in figure 6.1, assuming that we are using a myopic policy to evaluate future trajectories.

The advantage of this strategy is that it is simple and fast. It provides a better approximation than using a pure myopic policy since in this case, we are using a myopic policy to provide a rough estimate of the value of being in a state. This can work quite well, but it can work very poorly. The performance is very situation-dependent.

It is also possible to choose a decision using the current value function approximation, as in

$$\bar{a} = \arg \max_{a \in \mathcal{A}} (C(s, a) + \bar{V}^{n-1}(S^M(s, a))).$$

We note that we are most dependent on this logic in the early iterations when the estimates \bar{V}^n may be quite poor, but some estimate may be better than nothing.

In approximate dynamic programming (and in particular the reinforcement learning community), it is common practice to introduce an artificial discount factor. This factor is given its own name λ , reflecting the common (and occasionally annoying) habit of the reinforcement learning community to name algorithms after the notation used. If we introduce this factor, step 3c would be replaced with

$$\hat{v} = C(s, \bar{a}) + \gamma \lambda \bar{V}^n(s').$$

In this setting, γ is viewed as a fixed model parameter, while λ is a tunable algorithmic parameter. Its use in this setting reflects the fact that we are trying to estimate the value of a state s' by following some heuristic policy from s' onward. For example, imagine that

Procedure MyopicRollOut(a)

Step 0. Initialize: Given initial state s , increment tree depth $m = m + 1$.

Step 1. Find the best myopic decision using

$$\bar{a} = \arg \max_{a \in \mathcal{A}} C(s, a).$$

Step 2. If $m < M$, do:

Step 3a. Sample the random information W given state s .

Step 3b. Compute $s' = S^M(s, \bar{a}, W)$

Step 3c. Compute

$$\bar{V}^m(s') = \text{MyopicRollOut}(s').$$

Step 3d. Find the value of the decision

$$\hat{v} = C(s, \bar{a}) + \gamma \bar{V}^m(s').$$

Else:

$$\hat{v} = C(s, \bar{a}).$$

Endif

Step 4. Approximate the value of the state using:

$$\text{MyopicRollOut} = \hat{v}.$$

Figure 6.1 Approximating the value of being in a state using a myopic policy.

we are using this idea to play a game, and we find that our heuristic policy results in a loss. Should this loss be reflected in a poor value for state s' (because it led to a loss), or is it possible that state s' is actually a good state to be in, but the loss resulted from poor decisions made when we tried to search forward from s' ? Since we do not know the answer to this question, we discount values further into the future so that a negative outcome (or positive outcome) does not have too large of an impact on the results.

6.2.4 Rolling horizon procedures

Typically, the reason we cannot solve our original stochastic optimization problem to optimality is the usual explosion of problem size when we try to solve a stochastic optimization problem over an infinite or even a long horizon. As a result, a natural approximation strategy is to try to solve the problem over a shorter horizon. This is precisely what we were doing in section 6.2.1 using tree search, but that method was limited to discrete (and typically small) action spaces.

Imagine that we are at time t (in state S_t), and that we can solve the problem optimally over the horizon from t to $t+H$, for a sufficiently small horizon H . We can then implement the decision a_t (or x_t if we are dealing with vectors), after which we simulate our way to a single state S_{t+1} . We then repeat the process by optimizing over the interval $t+1$ to $t+H+1$. In this sense, we are “rolling” the horizon one time period forward. This method also goes under the names receding horizon procedure (common in operations research), or model predictive control (common in the engineering control theory literature).

Our hope is that we can solve the problem over a sufficiently long horizon that the decision a_t (or x_t) “here and now” is a good one. The problem is that if we want to solve

the full stochastic optimization problem, we might be limited to extremely short horizons. For this reason, we typically have to resort to approximations even to solve shorter-horizon problems. These strategies can be divided based on whether we are using a deterministic forecast of the future, or a stochastic one. A distinguishing feature of these methods is that we can handle vector-valued problems.

Deterministic forecasts A popular method in practice for building a policy for stochastic, dynamic problems is to use a point forecast of future exogenous information to create a deterministic model over a H -period horizon. In fact, the term rolling horizon procedure (or model predictive control) is often interpreted to specifically refer to the use of deterministic forecasts. We illustrate the idea using a simple example drawn from energy systems analysis. We switch to vector-valued decisions x , since a major feature of deterministic forecasts is that it makes it possible to use standard math programming solvers which scale to large problems.

Consider the problem of managing how much energy we should store in a battery to help power a building which receives energy from the electric power grid (but at a random price) or solar panels (but with random production due to cloud cover). We assume that we can always purchase power from the grid, but the price may be quite high.

Let

- R_t = The amount of energy stored in the battery at time t ,
- p_t = The price of electricity purchased at time t from the grid,
- h_t = The energy production from the solar panel at time t ,
- D_t = The demand for electrical power in the building at time t .

The state of our system is given by $S_t = (R_t, p_t, h_t, D_t)$. The system is controlled using

- x_t^{gb} = The amount of energy stored in the battery from the grid at price p_t at time t ,
- x_t^{sb} = The amount of energy stored in the battery from the solar panels at time t ,
- x_t^{sd} = The amount of energy directed from the solar panels to serve demand at time t ,
- x_t^{bd} = The amount of energy drawn from the battery to serve demand at time t ,
- x_t^{gd} = The amount of energy drawn from the grid to serve demand at time t .

We let $x_t = (x_t^{gb}, x_t^{sb}, x_t^{sd}, x_t^{bd})$ be the decision vector. The cost to meet demand at time t is given by

$$C(S_t, x_t) = p_t(x_t^{gb} + x_t^{gd}).$$

The challenge with deciding what to do right now is that we have to think about demands, prices and solar production in the future. Demand and solar production tend to follow a daily pattern, although demand rises early in the morning more quickly than solar production, and can remain high in the evening after solar production has disappeared. Prices tend to be highest in the middle of the afternoon, and as a result we try to have energy stored in the battery to reduce our demand for expensive electricity at this time.

We can make the decision x_t by optimizing over the horizon from t to $t + H$. While $p_{t'}$, $h_{t'}$ and $D_{t'}$, for $t' > t$, are all random variables, we are going to replace them with forecasts $\bar{p}_{tt'}$, $\bar{h}_{tt'}$ and $\bar{D}_{tt'}$, all made with information available at time t . Since these are deterministic, we can formulate the following deterministic optimization problem:

$$\min_{x_{tt}, \dots, x_{t,t+H}} \sum_{t'=t}^{t+H} \bar{p}_{tt'} (x_{tt'}^{gb} + x_{tt'}^{gd}), \quad (6.5)$$

subject to

$$R_{t'+1} = R_{t'} + x_{tt'}^{gb} + x_{tt'}^{sb} - x_{tt'}^{bd}, \quad (6.6)$$

$$x_{tt'}^{bd} \leq R_{t'}, \quad (6.7)$$

$$x_{tt'}^{gb}, x_{tt'}^{sb}, x_{tt'}^{bd}, x_{tt'}^{sd} \geq 0. \quad (6.8)$$

We note that we are letting $x_{tt'}$ be a type of forecast of what we think we will do at time t' when we solve the optimization problem at time t . It is useful to think of this as a forecast of a decision. We project decisions over this horizon because we need to know what we would do in the future in order to know what we should do right now.

The optimization problem (6.5) - (6.8) is a deterministic linear program. We can solve this using, say, 1 minute increments over the next 12 hours (720 time periods) without difficulty. However, we are not interested in the values of x_{t+1}, \dots, x_{t+H} . We are only interested in x_t , which we implement at time t . As we advance the clock from t to $t + 1$, we are likely to find that the random variables have not evolved precisely according to the forecast, giving us a problem starting at time $t + 1$ (extending through $t + H + 1$) that is slightly different than what we thought would be the case.

Our deterministic model offers several advantages. First, we have no difficulty handling the property that x_t is a continuous vector. Second, the model easily handles the highly nonstationary nature of this problem, with daily cycles in demand, prices and solar production. Third, if a weather forecast tells us that the solar production will be less than normal six hours from now, we have no difficulty taking this into account. Thus, knowledge about the future does not complicate the problem.

At the same time, the inability of the model to handle uncertainty in the future introduces significant weaknesses. One problem is that we would not feel that we need to store energy in the battery in the event that solar production *might* be lower than we expect. Second, we may wish to store electricity in the battery during periods where electricity prices are lower than normal, something that would be ignored in a forecast of future prices, since we would not forecast stochastic variations from the mean.

Deterministic rolling horizon policies are widely used in operations research, where a deterministic approximation provides value. The real value of this strategy is that it opens the door for using commercial solvers that can handle vector-valued decisions. These policies are rarely seen in the classical examples of reinforcement learning which focus on small action spaces, but which also focus on problems where a deterministic approximation of the future would not produce an interesting model.

Stochastic forecasts The tree search and roll-out heuristics were both methods that captured, in different ways, uncertainty in future exogenous events. In fact, tree search is, technically speaking, a rolling horizon procedure. However, neither of these methods can handle vector-valued decisions, something that we had no difficulty handling if we were willing to assume we knew future events were known deterministically.

We can extend our deterministic, rolling horizon procedure to handle uncertainty in an approximate way. The techniques have evolved under the umbrella of a field known as *stochastic programming*, which describes the mathematics of introducing uncertainty in math programs. Introducing uncertainty in multistage optimization problems in the presence of vector-valued decisions is intrinsically difficult, and not surprisingly there does not exist a computationally tractable algorithm to provide an exact solution to this problem. However, some practical approximations have evolved. We illustrate the simplest strategy here.

We start by dividing the problem into two *stages*. The first stage is time t , representing our “here and now” decision. We assume we know the state S_t at time t . The second stage starts at time $t + 1$, at which point we assume we know all the information that will arrive from $t + 1$ through $t + H$. The approximation here is that the decision x_{t+1} is allowed to “see” the entire future.

Even this fairly strong approximation is not enough to allow us to solve this problem, since we still cannot compute the expectation over all possible realizations of future information exactly. Instead, we resort to Monte Carlo sampling. Let ω be a sample path representing a sample realization of all the random variables $(p_{t'}, h_{t'}, D_{t'})_{t' > t}$ from $t + 1$ until $t + H$. We use Monte Carlo methods to generate a finite sample Ω of potential outcomes. If we fix ω , then this is like solving the deterministic optimization problem above, but instead of using forecasts $\bar{p}_{t'}$, $\bar{h}_{t'}$ and $\bar{D}_{t'}$, we use sample realizations $p_{t'}(\omega)$, $h_{t'}(\omega)$ and $D_{t'}(\omega)$. Let $p(\omega)$ be the probability that ω happens, where this might be as simple as $p(\omega) = 1/|\Omega|$. We further note that $p_{tt}(\omega) = p_{tt}$ for all ω (the same is true for the other random variables). We are going to create decision variables $x_{tt}(\omega), \dots, x_{t,t+H}(\omega)$ for all $\omega \in \Omega$. We are going to allow $x_{t't'}(\omega)$ to depend on ω for $t' > t$, but then we are going to require that $x_{tt}(\omega)$ be the same for all ω .

We now solve the optimization problem

$$\min_{x_{tt}(\omega), \dots, x_{t,t+H}(\omega), \omega \in \Omega} \bar{p}_{tt}(x_{tt}^{gb} + \bar{p}_{tt}x_{tt}^{gd}) + \sum_{\omega \in \Omega} p(\omega) \sum_{t'=t+1}^{t+H} \bar{p}_{t't'}(\omega)(x_{t't'}^{gb}(\omega) + x_{t't'}^{gd}(\omega)),$$

subject to the constraints

$$R_{t'+1}(\omega) = R_{t'}(\omega) + x_{t't'}^{gb}(\omega) + x_{t't'}^{sb}(\omega)x_{t't'}^{bd}(\omega), \quad (6.9)$$

$$x_{t't'}^{bd}(\omega) \leq R_{t'}(\omega), \quad (6.10)$$

$$x_{t't'}^{gb}(\omega), x_{t't'}^{sb}(\omega), x_{t't'}^{bd}(\omega), x_{t't'}^{sd}(\omega) \geq 0. \quad (6.11)$$

If we only imposed the constraints (6.9)-(6.11), then we would have a different solution $x_{tt}(\omega)$ for each scenario, which means we are allowing our decision now to see into the future. To prevent this, we impose *nonanticipativity* constraints

$$x_{tt}(\omega) - \bar{x}_{tt} = 0. \quad (6.12)$$

Equation (6.12) means that we have to choose one decision \bar{x}_{tt} regardless of the outcome ω , which means we are not allowed to see into the future. However, we are allowing $x_{t,t+1}(\omega)$ to depend on ω , which specifies not only what happens between t and t' , but also what happens for the entire future. This means that at time $t + 1$, we are allowing our decision to see future energy prices, future solar output and future demands. However, this is only for the purpose of approximating the problem so that we can make a decision at time t , which is not allowed to see into the future.

This formulation allows us to make a decision at time t while modeling the stochastic variability in future time periods. In the stochastic programming community, the outcomes ω are often referred to as *scenarios*. If we model 20 scenarios, then our optimization problem becomes roughly 20 times larger, so the introduction of uncertainty in the future comes at a significant computational cost. However, this formulation allows us to capture the variability in future outcomes, which can be a significant advantage over using simple forecasts. Furthermore, while the problem is certainly much larger, it is still a deterministic optimization problem that can be solved with standard commercial solvers.

There are numerous applications where there is a desire to break the second stage into additional stages, so we do not have to make the approximation that x_{t+1} sees the entire future. However, if we have $|\Omega|$ outcomes in each stage, then if we have K stages, we would be modeling $|\Omega|^K$ scenario paths. Needless to say, this grows quickly with K . Modelers overcome this by reducing the number of scenarios in the later stages, but it is hard to assess the errors that are being introduced.

6.2.4.1 Rolling horizon with discounting A common objective function in dynamic programming uses discounted costs, whether over a finite or infinite horizon. If $C(S_t, x_t)$ is the contribution earned from implementing decision x_t when we are in state S_t , our objective function might be written

$$\max_{\pi} \mathbb{E} \sum_{t=0}^{\infty} \gamma^t C(S_t, X^{\pi}(S_t))$$

where γ is a discount factor that captures, typically, the time value of money. If we choose to use a rolling horizon policy (with deterministic forecasts), our policy might be written

$$X^{\pi}(S_t) = \arg \max_{x_t, \dots, x_{t+H}} \sum_{t'=t}^{t+H} \gamma^{t'-t} C(S_{t'}, x_{t'}).$$

Let's consider what a discount factor might look like if it only captures the time value of money. Imagine that we are solving an operational problem where each time step is one hour (there are many applications where time steps are in minutes or even seconds). If we assume a time value of money of 10 percent per year, then we would use $\gamma = 0.999989$ for our hourly discount factor. If we use a horizon of, say, 100 hours, then we might as well use $\gamma = 1$. Not surprisingly, it is common to introduce an artificial discount factor to reflect the fact that decisions made at $t' = t + 10$ should not carry the same weight as the decision x_t that we are going to implement right now. After all, $x_{t'}$ for $t' > t$ is only a forecast of a decision that might be implemented.

In our presentation of rollout heuristics, we introduced an artificial discount factor λ to reduce the influence of poor decisions backward through time. In our rolling horizon model, we are actually making optimal decisions, but only for a deterministic approximation (or perhaps an approximation of a stochastic model). When we use introduce this new discount factor, our rolling horizon policy would be written

$$X^{\pi}(S_t) = \arg \max_{x_t, \dots, x_{t+H}} \sum_{t'=t}^{t+H} \gamma^{t'-t} \lambda^{t'-t} C(S_{t'}, x_{t'}).$$

In this setting, γ plays the role of the original discount factor (which may be equal to 1.0, especially for finite horizon problems), while λ is a tunable parameter. An obvious motivation for the use of λ in a rolling horizon setting is the recognition that we are solving a deterministic approximation over a horizon in which events are uncertain.

6.2.5 Discussion

Lookahead strategies have long been a popular method for designing policies for stochastic, dynamic problems. In each case, note that we use both a model of exogenous information, as well as some sort of method for making decisions in the future. Since optimizing over the future, even for a shorter horizon, quickly becomes computationally intractable, different strategies are used for simplifying the outcomes of future information, and simplifying how decisions might be made in the future.

Even with these approximations, lookahead procedures can be computationally demanding. It is largely as a result of a desire to find faster algorithms that researchers have developed methods based on value function approximations and policy function approximations.

6.3 POLICY FUNCTION APPROXIMATIONS

It is often the case that we have a very good idea of how to make a decision, and we can design a function (which is to say a policy) that returns a decision which captures the structure of the problem. For example:

■ EXAMPLE 6.1

A policeman would like to give tickets to maximize the revenue from the citations he writes. Stopping a car requires about 15 minutes to write up the citation, and the fines on violations within 10 miles per hour of the speed limit are fairly small. Violations of 20 miles per hour over the speed limit are significant, but relatively few drivers fall in this range. The policeman can formulate the problem as a dynamic program, but it is clear that the best policy will be to choose a speed, say \bar{s} , above which he writes out a citation. The problem is choosing \bar{s} .

■ EXAMPLE 6.2

A utility wants to maximize the profits earned by storing energy in a battery when prices are lowest during the day, and releasing the energy when prices are highest. There is a fairly regular daily pattern to prices. The optimal policy can be found by solving a dynamic program, but it is fairly apparent that the policy is to charge the battery at one time during the day, and discharge it at another. The problem is identifying these times.

Policies come in many forms, but all are functions that return a decision given an action. In this section, we are limiting ourselves to functions that return a policy directly from the state, without solving an imbedded optimization problem. As with value function approximations, which return an estimate of the value of being in a state, policy functions return an action given a state. Policy function approximations come in three basic styles:

Lookup tables - When we are in a discrete state s , the function returns a discrete action a directly from a matrix.

Parametric representations - These are explicit, analytic functions which are designed by the analyst. Similar to the process of choosing basis functions for value function approximations, designing a parametric representation of a policy is an art form.

Nonparametric representations - Nonparametric statistics offer the potential of producing very general behaviors without requiring the specification of basis functions.

Since a policy is *any* function that returns an action given a state, it is important to recognize when designing an algorithm if we are using a) a rolling horizon procedure, b) a value function approximation or c) a parametric representation of a policy. Readers need to understand that when a reference is made to “policy search” and “policy gradient methods,” this usually means that a parametric representation of a policy is being used.

We briefly describe each of these classes below.

6.3.1 Lookup table policies

We have already seen lookup table representations of policies in chapter 3. By a lookup table, we specifically refer to a policy where every state has its own action (similar to having a distinct value of being in each state). This means we have one parameter (an action) for each state. We exclude from this class any policies that can be parameterized by a smaller number of parameters.

It is fairly rare that we would use a lookup table representation in an algorithmic setting, since this is the least compact form. However, lookup tables are relatively common in practice, since they are easy to understand. The Transportation Safety Administration (TSA) has specific rules that determine when and how a passenger should be searched. Call-in centers use specific rules to govern how a call should be routed. A utility will use rules to govern how power should be released to the grid to avoid a system failure. Lookup tables are easy to understand, and easy to enforce. But in practice, they can be very hard to optimize.

6.3.2 Parametric policy models

Without question the most widely used form of policy function approximation is simple, analytic models parameterized by a low dimensional vector. Some examples include:

- We are holding a stock, and would like to sell it when it goes over a price θ .
- In an inventory policy, we will order new product when the inventory S_t falls below θ_1 . When this happens, we place an order $a_t = \theta_2 - S_t$, which means we “order up to” θ_2 .
- We might choose to set the output x_t from a water reservoir, as a function of the state (the level of the water) S_t of the reservoir, as a linear function of the form $x_t = \theta_0 + \theta_1 S_t$. Or we might desire a nonlinear relationship with the water level, and use a basis function $\phi(S_t)$ to produce a policy $x_t = \theta_0 + \theta_1 \phi(S_t)$. These are known in the control literature as *affine policies* (literally, linear in the parameters).

Designing parametric policy function approximations is an art form. The science arises in determining the parameter vector θ .

To choose θ , assume that we have a parametric policy $A^\pi(S_t|\theta)$ (or $X^\pi(S_t|\theta)$), where we express the explicit dependence of the policy on θ . We would then approach the problem

of finding the best value of θ as a stochastic optimization problem, written in the form

$$\max_{\theta} F(\theta) = \mathbb{E} \sum_{t=0}^T \gamma^t C(S_t, A^{\pi}(S_t|\theta)).$$

Here, we write \max_{θ} whereas before we would write $\max_{\pi \in \Pi}$ to express the fact that we are optimizing over a class of policies Π that captures the parametric structure. $\max_{\pi \in \Pi}$ is more generic, but in this setting, it is equivalent to \max_{θ} .

The major challenge we face is that we cannot compute $F(\theta)$ in any compact form, primarily because we cannot compute the expectation. Instead, we have to depend on Monte Carlo samples. Fortunately, there is an entire field known as stochastic search (see Spall (2003) for an excellent overview) to help us with this process. We describe these algorithms in more detail in chapter 7.

There is another strategy we can use to fit a parametric policy model. Let $\bar{V}^{n-1}(s)$ be our current value function approximation at state s . Assume that we are currently at a state S^n and we use our value function approximation to compute an action a^n using

$$a^n = \arg \max_a \left(C(S^n, a) + \bar{V}^{n-1}(S^{M,a}(S^n, a)) \right).$$

We can now view a^n as an “observation” of the action a^n corresponding to state S^n . When we fit value function approximations, we used the observation \hat{v}^n of the value of being in state S^n to fit our value function approximation. With policies, we can use a^n as the observation corresponding to state S^n to fit a function that “predicts” the action we should be taking when in state S^n .

6.3.3 Nonparametric policy models

The strengths and weaknesses of using parametric models for policy function approximations is the same as we encountered when using them for approximating value functions. Not surprisingly, there has been some interest in using nonparametric models for policies. This has been most widely used in the engineering control literature, where neural networks have been popular. We describe neural networks in chapter 8.

The basic idea of a nonparametric policy is to use nonparametric statistics, such as neural networks, to fit a function that predicts the action a when we are in state s . Just as we described above with a parametric policy model, we can compute a^n using a value function approximation $\bar{V}(s)$ when we are in state $s = S^n$. We then use the pairs of responses a^n and independent variables (covariates) S^n to fit a neural network.

6.4 VALUE FUNCTION APPROXIMATIONS

The most powerful and visible method for solving complex dynamic programs involves replacing the value function $V_t(S_t)$ with an approximation of some form. In fact, for many, approximate dynamic programming is viewed as being equivalent to replacing the value function with an approximation as a way of avoiding “the” curse of dimensionality. If we are approximating the value around the pre-decision state, a value function approximation means that we are making decisions using equations of the form

$$a_t = \arg \max_a \left(C(S_t, a) + \gamma \mathbb{E} \left\{ \bar{V}(S^M(S_t, a, W_{t+1})) | S_t \right\} \right). \quad (6.13)$$

In chapter 4, we discussed the challenge of computing the expectation, and argued that we can approximate the value around the post-decision state S_t^a , giving us

$$a_t = \arg \max_a (C(S_t, a) + \bar{V}(S^{M,a}(S_t, a))). \quad (6.14)$$

For our presentation here, we are not concerned with whether we are approximating around the pre- or post-decision state. Instead, we simply want to consider different types of approximation strategies. There are three broad classes of approximation strategies that we can use:

- 1) Lookup tables - If we are in discrete state s , we have a stored value $\bar{V}(s)$ that gives an estimate of the value of being in state s .
- 2) Parametric models - These are analytic functions $\bar{V}(s|\theta)$ parameterized by a vector θ whose dimensionality is much smaller than the number of states.
- 3) Nonparametric models - These models draw on the field of nonparametric statistics, and which avoid the problem of designing analytic functions.

We describe each of these briefly below, but defer a more complete presentation to chapter 8.

6.4.1 Lookup tables

Lookup tables are the most basic way of representing a function. This strategy is only useful for discrete states, and typically assumes a *flat representation* where the states are numbered $s \in \{1, 2, \dots, |\mathcal{S}|\}$, where \mathcal{S} is the original set of states, where each state might consist of a vector. Using a flat representation, we have one parameter, $\bar{V}(s)$, for each value of s (we refer to the value of being in each state as a parameter, since it is a number that we have to estimate). As we saw in chapter 4, if \hat{v}^n is a Monte Carlo estimate of the value of being in a state S^n , then we can update the value of being in state S^n using

$$\bar{V}^n(S^n) = (1 - \alpha)\bar{V}^{n-1}(S^n) + \alpha\hat{v}^n.$$

Thus, we only learn about $\bar{V}^n(S^n)$ by visiting state S^n . If the state space is large, then this means that we have a very large number of parameters to estimate.

The power of lookup tables is that, if we have a discrete state space, then in principle we can eventually approximate $V(S)$ with a high level of precision (if we visit all the states often enough). The down side is that visiting state s tells us nothing about the value of being in state $s' \neq s$.

6.4.2 Parametric models

One of the simplest ways to avoid the curse of dimensionality is to replace value functions using a lookup table with some sort of regression model. We start by identifying what we feel are important features. If we are in state S , a feature is simply a function $\phi_f(S)$, $f \in \mathcal{F}$ that draws information from S . We would say that \mathcal{F} (or more properly $(\phi_f(S))_{f \in \mathcal{F}}$) is our set of features. In the language of approximate dynamic programming the functions $\phi_f(S)$ are referred to as *basis functions*.

Creating features is an art form, and depends on the problem. Some examples are as follows:

- We are trying to manage blood inventories, where R_{ti} is the number of units of blood type $i \in (1, 2, \dots, 8)$ at time t . The vector R_t is our state variable. Since R_t is a vector, the number of possible values of R_t may be extremely large, making the challenge of approximating $V(R)$ quite difficult. We might create a set of features $\phi_i^1(R) = R_i$, $\phi_i^2(R) = R_i^2$, $\phi_{ij}^3(R) = R_i R_j$.
- Perhaps we wish to design a computer to play tic-tac-toe. We might design features such as: a) the number of X's in corner points; b) 0/1 to indicate if we have the center point; c) the number of side points (excluding corners) that we have; d) the number of rows and columns where we have X's in at least two elements.

Once we have a set of features, we might specify a value function approximation using

$$\bar{V}(S) = \sum_{f \in \mathcal{F}} \theta_f \phi_f(S), \quad (6.15)$$

where θ is a vector of regression parameters to be estimated. Finding θ is the science within the art form. When we use an approximation of the form given in equation (6.15), we say that the function has a *linear architecture* or is a linear model, or simply linear. Statisticians will often describe (6.15) as a linear model, but care has to be used, because the basis functions may be nonlinear functions of the state. Just because an approximation is a linear model does not mean that the function is linear in the state variables.

If we are working with discrete states, it can be analytically convenient to create a matrix Φ with element $\phi_f(s)$ in row s , column f . If \bar{V} is our approximate value function, expressed as a column vector with an element for every state, then we can write the approximation using matrix algebra as

$$\bar{V} = \Phi\theta.$$

There are, of course, problems where a linear architecture will not work. For example, perhaps our value function is expressing a probability that we will win at backgammon, and we have designed a set of features $\phi_f(S)$ that capture important elements of the board, where S is the precise state of the board. We propose that we can create a utility function $U(S|\theta) = \sum_f \theta_f \phi_f(S)$ that provides a measure of the quality of our position. However, we do not observe utility, but we do observe if we eventually win or lose a game. If $V = 1$ if we win, we might write the probability of this using

$$P(V = 1|\theta) = \frac{e^{U(S|\theta)}}{1 + e^{U(S|\theta)}}. \quad (6.16)$$

We would describe this model as one with a *nonlinear architecture*, or nonlinear in the parameters. Our challenge is estimating θ from observations of whether we win or lose the game.

Parametric models are exceptionally powerful when they work. Their real value is that we can estimate a parameter vector θ using a relatively small number of observations. For example, we may feel that we can do a good job if we have 10 observations per feature. Thus, with a very small number of observations, we obtain a model that offers the potential of approximating an entire function, even if S is multidimensional and continuous. Needless to say, this idea has attracted considerable interest. In fact, many people even equate “approximate dynamic programming” with “approximating value functions using basis functions.”

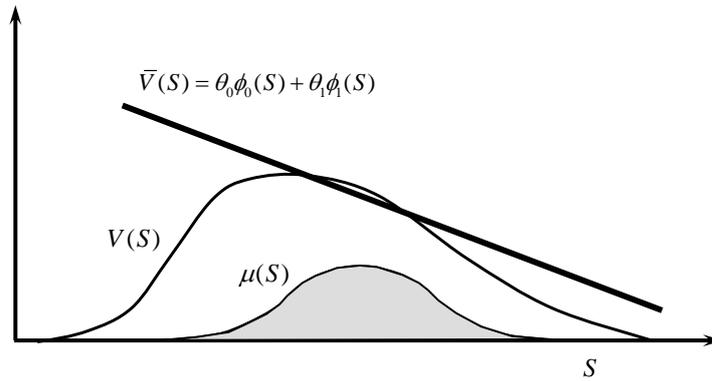


Figure 6.2 Illustration of linear (in the state) value function approximation of a nonlinear value function ($\bar{V}(S)$), given the density $\mu(S)$ for visiting states.

The downside of parametric models is that we may not have the right basis functions to produce a good approximation. As an example, consider the problem of approximating the value of storing natural gas in a storage facility, shown in figure 6.2. Here, $V(S)$ is the true value function as a function of the state S which gives the amount of gas in storage, and $\mu(S)$ is the probability distribution describing the frequency with which we visit different states. Now assume that we have chosen basis functions where $\phi_0(S) = 1$ (the constant term) and $\phi_1(S) = S$ (linear in the state variable). With this choice, we have to fit a value function that is nonlinear in S , with a function that is linear in S . The figure depicts the best linear approximation given $\mu(S)$, but obviously this would change quite a bit if the distribution describing what states we visited changed.

6.4.3 Nonparametric models

Nonparametric models can be viewed as a hybrid of lookup tables and parametric models, but in some key respects are actually closer to lookup tables. There is a wide range of nonparametric models, but they all share the fundamental feature of using simple models (typically constants or linear models) to represent small regions of a function. They avoid the need to specify a specific structure such as the linear architecture in (6.15) or a nonlinear architecture such as that illustrated in (6.16). At the same time, most nonparametric models offer the additional feature of providing very accurate approximations of very general functions, as long as we have enough observations.

There are a variety of nonparametric methods, and the serious reader should consult a reference such as Hastie et al. (2009). The central idea is to estimate a function by using a weighted estimate of local observations of the function. Assume we wish an estimate of the value function $\bar{V}(s)$ at a particular state (called the query state) s . We have made a series of observations $\hat{v}^1, \hat{v}^2, \dots, \hat{v}^n$ at states S^1, S^2, \dots, S^n . We can form an estimate $\bar{V}(s)$ using

$$\bar{V}(s) = \frac{\sum_{i=1}^n K_h(s, s^i) \hat{v}^i}{\sum_{i=1}^n K_h(s, s^i)}, \quad (6.17)$$

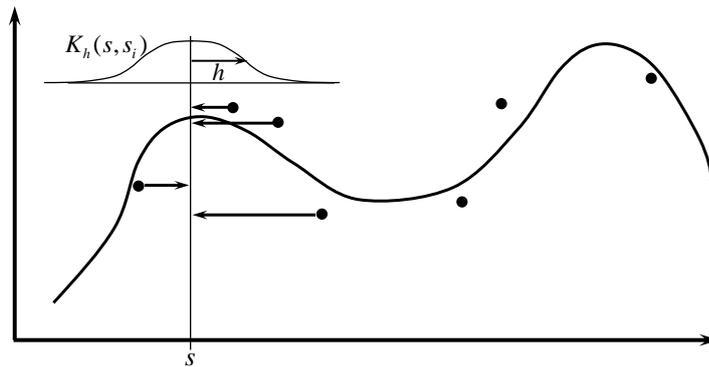


Figure 6.3 Illustration of kernel with bandwidth h being used to estimate the value function at a query state s .

where $K_h(s, s^i)$ is a weighting function parameterized by a *bandwidth* h . A common choice of weighting function is the Gaussian density, given by

$$K_h(s, s^i) = \exp\left(-\frac{(s - s^i)^2}{h}\right).$$

Here, the bandwidth h serves the role of the standard deviation. This form of weighting is often referred to as *radial basis functions* in the approximate dynamic programming literature, which has the unfortunate effect of placing it in the same family of approximation strategies as linear regression. However, parametric and nonparametric approximation strategies are fundamentally different, with very different implications in terms of the design of algorithms.

Nonparametric methods offer significant flexibility. Unlike parametric models, where we are finding the parameters of a fixed functional form, nonparametric methods do not have a formal model, since the data itself forms the model. Any observation of the function at a query state s requires summing over all prior observations, which introduces additional computational requirements. For approximate dynamic programming, we encounter the same issue we saw in chapter 4 of exploration. With lookup tables, we have to visit a particular state in order to estimate the value of a state. With kernel regression, we relax this requirement, but we still have to visit points close to a state in order to estimate the value of a state.

6.5 HYBRID STRATEGIES

The concept of (possibly tunable) myopic policies, lookahead policies, policies based on value function approximations, and policy function approximation represent the core tools in the arsenal for approximating solving dynamic programs. Given the richness of applications, it perhaps should not be surprising that we often turn to mixtures of these strategies.

Myopic policies with tunable parameters

Myopic policies are so effective for some problem classes that all we want to do is to make them a little smarter, which is to say make decisions which do a better job capturing the impact of a decision now on the future. We can create a family of myopic policies by introducing tunable parameters which might help achieve good behaviors over time. Imagine that task $j \in \mathcal{J}_t$ needs to be handled before time τ_j . If a task is not handled within the time window, it vanishes from the system and we lose the reward that we might have earned. Now define a modified reward given by

$$c_{tij}^\pi(\theta) = c_{ij} + \theta_0 e^{-\theta_1(\tau_j - t)}. \quad (6.18)$$

This modified contribution increases the reward (assuming that θ_0 and θ_1 are both positive) for covering a task as it gets closer to its due date τ_j . The reward is controlled by the parameter vector $\theta = (\theta_0, \theta_1)$. We can now write our myopic policy in the form

$$X^\pi(S_t) = \arg \max_{x \in \mathcal{X}_t} \sum_{i \in \mathcal{I}} \sum_{j \in \mathcal{J}_t} c_{tij}^\pi(\theta) x_{tij}. \quad (6.19)$$

The policy π is characterized by the function in (6.19), which includes the choice of the vector θ . We note that by tuning θ , we might be able to get our myopic policy to make decisions that produce better decisions in the future. But the policy does not explicitly use any type of forecast of future information or future decisions.

Rolling horizon procedures with value function approximations

Deterministic rolling horizon procedures offer the advantage that we can solve them optimally, and if we have vector-valued decisions, we can use commercial solvers. Limitations of this approach are a) they require that we use a deterministic view of the future and b) they can be computationally expensive to solve (pushing us to use shorter horizons). By contrast, a major limitation of value function approximations is that we may not be able to capture the complex interactions that are taking place within our optimization of the future.

An obvious strategy is to combine the two approaches. For low-dimensional action spaces, we can use tree search or a roll-out heuristics for H periods, and then use a value function approximation. If we are using a rolling horizon procedure for vector-valued decisions, we might solve

$$X^\pi(S_t) = \arg \max_{x_t, \dots, x_{t+H}} \sum_{t'=t}^{t+H-1} \gamma^{t'-t} C(S_{t'}, x_{t'}) + \gamma^H \bar{V}_{t+H}(S_{t+H}),$$

where S_{t+H} is determined by X_{t+H} . In this setting, $\bar{V}_{t+H}(S_{t+H})$ would have to be some convenient analytical form (linear, piecewise linear, nonlinear) in order to be used in an appropriate solver. If we use the algorithmic discount factor λ , the policy would look like

$$X^\pi(S_t) = \arg \max_{x_t, \dots, x_{t+H}} \sum_{t'=t}^{t+H-1} \gamma^{t'-t} \lambda^{t'-t} C(S_{t'}, x_{t'}) + \gamma^H \lambda^H \bar{V}_{t+H}(S_{t+H}).$$

Rolling horizon procedures with tunable policies

A rolling horizon procedure using a deterministic forecast is, of course, vulnerable to the use of a point forecast of the future. For example, in the energy application we used in

section 6.2.4 to introduce rolling horizon procedures, we might use a deterministic forecast of prices, energy from the solar panels and demand. A problem with this is that the model will optimize assuming these forecasts are known perfectly, potentially leaving no buffer in the battery to handle surges in demand or drops in solar energy.

This limitation will not be solved by introducing value function approximations at the end of the horizon. It is possible, however, to perturb our deterministic model to make the solution more robust with respect to uncertainty. For example, we might reduce the amount of energy we expect to receive from the solar panel, or inflate the demand. As we step forward in time, if we have overestimated demand or underestimated the energy from the solar panel, we can put the excess energy in the battery. Assume we factor demand by $\beta^D > 1$, and factor solar energy output by $\beta^S < 1$. Now, we have a rolling horizon policy parameterized by (β^D, β^S) . We can tune these parameters just as we would tune the parameters of a policy function approximation.

Just as designing a policy function approximation is an art, designing these adjustments to a rolling horizon policy (or any other lookahead strategy) is also an art. Imagine optimizing the movement of a robot arm to take into consideration variations in the mechanics. We can optimize over a horizon ignoring these variations, but we might plan a trajectory that puts the arm too close to a barrier. We can modify our rolling horizon procedure by introducing a margin of error (the arm cannot be closer than β to a barrier), which becomes a tunable parameter.

Rollout heuristics with policy function approximation

Rollout heuristics can be particularly effective when we have access to a reasonable policy to simulate decisions that we might take if an action a takes us to a state s' . This heuristic might exist in the form of a policy function approximation $X^\pi(s|\theta)$ parameterized by a vector θ . We can tune θ , recognizing that we are still choosing a decision based on the one period contribution $C(s, a)$ plus the approximate value of being in state $s' = S^M(s, a, W)$.

Tree search with rollout heuristic and a lookup table policy

A surprisingly powerful heuristic algorithm that has received considerable success in the context of designing computer algorithms to play games uses a limited tree search, which is then augmented by a rollout heuristic assisted by a user-defined lookup table policy. For example, a computer might evaluate all the options for a chess game for the next four moves, at which point the tree grows explosively. After four moves, the algorithm might resort to a rollout heuristic, assisted by rules derived from thousands of chess games. These rules are encapsulated in an aggregated form of lookup table policy that guides the search for a number of additional moves into the future.

Value function approximation with lookup table or policy function approximation

Assume we are given a policy $\bar{A}(S_t)$, which might be in the form of a lookup table or a parameterized policy function approximation. This policy might reflect the experience of a domain expert, or it might be derived from a large database of past decisions. For example, we might have access to the decisions of people playing online poker, or it might be the historical patterns of a company. We can think of $\bar{A}(S_t)$ as the decision of the domain expert or the decision made in the field. If the action is continuous, we could incorporate

it into our decision function using

$$A^\pi(S_t) = \arg \max_a (C(S_t, a) + \bar{V}(S^{M,a}(S_t, a)) + \beta(\bar{A}(S_t) - a)^2).$$

The term $\beta(\bar{A}(S_t) - a)^2$ can be viewed as a penalty for choosing actions that deviate from the external domain expert. β controls how important this term is. We note that this penalty term can be set up to handle decisions at some level of aggregation.

6.6 RANDOMIZED POLICIES

In section 3.10.5, we introduced the idea of randomized policies, and showed that under certain conditions a deterministic policy (one where the state S and a decision rule uniquely determine an action) will always do at least as well or better than a policy that chooses an action partly at random. In the setting of approximate dynamic programming, randomized policies take on a new role. We may, for example, wish to ensure that we are testing all actions (and reachable states) infinitely often. This arises because in ADP, we need to *explore* states and actions to learn how well they do.

Common notation in approximate dynamic programming for representing randomized policies is to let $\pi(a|s)$ be the probability of choosing action a given we are in state s . This is known as notational overloading, since the policy in this case is both the rule for choosing an action, and the probability that the rule produces for choosing the action. We separate these roles by allowing $\pi(a|s)$ to be the probability of choosing action a if we are in state s (or, more often, the parameters of the distribution that determines this probability), and let $A^\pi(s)$ be the decision function, which might sample an action a at random using the probability distribution $\pi(a|s)$.

There are different reasons in approximate dynamic programming for using randomized policies. Some examples include

- We often have to strike a balance between exploring states and actions, and choosing actions that appear to be best (exploiting). The exploration vs. exploitation issue is addressed in depth in chapter 12.
- In chapter 9 (section 9.4) discusses the need for policies where the probability of selection an action be strictly positive.
- In chapter 10 (section 10.7), we introduce a class of algorithms which require that the policy be parameterized so that the probability of choosing an action is differentiable in a parameter, which means that it also needs to be continuous.

Possibly the most widely used policy that mixes exploration and exploitation is called ϵ -greedy. In this policy, we choose an action $a \in \mathcal{A}$ at random with probability ϵ , and with probability $1 - \epsilon$ we choose the action

$$a^n = \arg \max_{a \in \mathcal{A}} (C(S^n, a) + \gamma \mathbb{E} \bar{V}^{n-1}(S^M(S^n, a, W^{n+1}))),$$

which is to say we exploit our estimate of $\bar{V}^{n-1}(s')$. This policy guarantees that we will try all actions (and therefore all reachable states) infinitely often, which helps with convergence proofs. Of course, this policy may be of limited value for problems where \mathcal{A} is large.

One limitation of ϵ -greedy is that it keeps exploring at a constant rate. As the algorithm progresses, we may wish to spend more time evaluating actions we think are best, rather

than exploring actions at random. A strategy that addresses this is to use a declining exploration probability, such as

$$\epsilon^n = 1/n.$$

This strategy not only reduces the exploration probability, but it does so at a rate that is sufficiently slow (in theory) that it ensures that we will still try all actions infinitely often. In practice, it is likely to be better to use a probability such as $1/n^\beta$ where $\beta \in (0.5, 1]$.

The ϵ -greedy policy exhibits a property known in the literature as “greedy in the limit with infinite exploration,” or GLIE. Such a property is important in convergence proofs. A limitation of ϵ -greedy is that when we explore, we choose actions at random without regard to their estimated value. In applications with larger action spaces, this can mean that we are spending a lot of time evaluating actions that are quite poor. An alternative is to compute a probability of choosing an action based on its estimated value. The most popular of these rules uses

$$P(a|s) = \frac{e^{\beta \bar{Q}^n(s,a)}}{\sum_{a' \in \mathcal{A}} e^{\beta \bar{Q}^n(s,a')}},$$

where $\bar{Q}^n(s, a)$ is an estimate of the value of being in state s and taking action a . β is a tunable parameter, where $\beta = 0$ produces a pure exploration policy, while as $\beta \rightarrow \infty$, the policy becomes greedy (choosing the action that appears to be best).

This policy is known under different names such as Boltzmann exploration, Gibbs sampling and a soft-max policy. Boltzmann exploration is also known as a “restricted rank-based randomized” policy (or RRR policy). The term “soft-max” refers to the fact that it is maximizing over the actions based on the factors $Q(s, a)$, but produces an outcome where actions other than the best have a positive probability of being chosen.

6.7 HOW TO CHOOSE A POLICY?

Given the choice of policies, the question naturally arises, how do we choose which policy is best for a particular problem? Not surprisingly, it depends on both the characteristics of the problem and constraints on computation time and the complexity of the algorithm. Below we summarize different types of problems, and provide a sample of a policy that appears to be well suited to the application, largely based on our own experiences with real problems.

A myopic policy

A business jet company has to assign pilots to aircraft to serve customers. In making decisions about which jet and which pilot to use, it might make sense to think about whether a certain type of jet is better suited given the destination of the customer (are there other customers nearby who prefer that same type of jet)? We might also want to think about whether we want to use a younger pilot who needs more experience. However, the here-and-now costs of finding the pilot and aircraft which can serve the customer at least cost dominate what *might* happen in the future. Also, the problem of finding the best pilot and aircraft, across a fleet of hundreds of aircraft and thousands of pilots, produces a fairly difficult integer programming problem. For this situation, a myopic policy works extremely well given the financial objectives and algorithmic constraints.

A lookahead policy - tree search

You are trying to program a computer to operate a robot that is described by several dozen different parameters describing the location, velocity and acceleration of both the robot and its arms. The number of actions is not too large, but choosing an action requires considering not only what might happen next (exogenous events) but also decisions we might make in the future. The state space is very large with relatively little structure, which makes it very hard to approximate the value of being in a state. Instead, it is much easier to simply step into the future, enumerating future potential decisions with an approximation (e.g. with Monte Carlo sampling) of random events.

A lookahead policy - deterministic rolling horizon procedures

Now assume that we are managing an energy storage device for a building that has to balance energy from the grid (where an advance commitment has been made the previous day), an hourly forecast of energy usage (which is fairly predictable), and energy generation from a set of solar panels on the roof. If the demand of the building exceeds its commitment from the grid, it can turn to the solar panels and battery, using a diesel generator as a backup. With a perfect forecast of the day, it is possible to formulate the decision of how much to store and when, when to use energy in the battery, and when to use the diesel generator as a single, deterministic mixed-integer program. Of course, this ignores the uncertainty in both demand and the sun, but most of the quantities can be predicted with some degree of accuracy. The optimization model makes it possible to balance dynamics of hourly demand and supply to produce an energy plan over the course of the day.

A lookahead policy - stochastic rolling horizon procedures

A utility has to operate a series of water reservoirs to meet demands for electricity, while also observing a host of physical and legal constraints that govern how much water is released and when. For example, during dry periods, there are laws that specify that the discomfort of low water flows has to be shared. Of course, reservoirs have maximum and minimum limits on the amount of water they can hold. There can be tremendous uncertainty about the rainfall over the course of a season, and as a result the utility would like to make decisions now while accounting for the different possible outcomes. Stochastic programming enumerates all the decisions over the entire year, for each possible scenario (while ensuring that only one decision is made now). The stochastic program ensures that all constraints will be satisfied for each scenario.

Policy function approximations

A utility would like to know the value of a battery that can store electricity when prices are low and release them when prices are high. The price process is highly volatile, with a modest daily cycle. The utility needs a simple policy that is easy to implement in software. The utility chose a policy where we fix two prices, and store when prices are below the lower level and release when prices are above the higher level. This requires optimizing these two price points. A different policy might involve storing at a certain time of day, and releasing at another time of day, to capture the daily cycle.

Value function approximations - transportation

A truckload carrier needs to determine the best driver to assign to a load, which may take several days to complete. The carrier has to think about whether it wants to accept the load (if the load goes to a particular city, will there be too many drivers in that city?). It also has to decide if it wants to use a particular driver, since the driver eventually needs to get home, and the load may take it to a destination from which it is difficult for the driver to eventually get home. Loads are highly dynamic, so the carrier will not know with any precision which loads will be available when the driver arrives. What has worked very well is to estimate the value of having a particular type of driver in a particular location. With this value, the carrier has a relatively simple optimization problem to determine which driver to assign to each load.

Value function approximations - energy

There is considerable interest in the interaction of energy from wind and hydroelectric storage. Hydroelectric power is fairly easy to manipulate, making it an attractive source of energy that can be paired with energy from wind. However, this requires modeling the problem in hourly increments (to capture the variation in wind) over an entire year (to capture seasonal variations in rainfall and the water being held in the reservoir). It is also critical to capture the uncertainty in both the wind and rainfall. The problem is relatively simple, and decisions can be made using a value function approximation that captures only the value of storing a certain amount of water in the reservoir. However, these value functions have to depend on the time of year. Deterministic approximations provide highly distorted results, and simple rules require parameters that depend on the time (hour) of year. Optimizing a simple policy requires tuning thousands of parameters. Value function approximations make it relatively easy to obtain time-dependent decisions that capture uncertainty.

Discussion

These examples raise a series of questions that should be asked when choosing the structure of a policy:

- Will a myopic policy solve your problem? If not, is it at least a good starting point?
- Does the problem have structure that suggests a simple and natural decision rule? If there is an “obvious” policy (e.g. replenish inventory when it gets too low), then more sophisticated algorithms based on value function approximations are likely to struggle. Exploiting structure always helps.
- Is the problem fairly stationary, or highly nonstationary? Nonstationary problems (e.g. responding to hourly demand or daily water levels) mean that you need a policy that depends on time. Rolling horizon problems can work well if the level of uncertainty is low relative to the predictable variability. It is hard to produce policy function approximations where the parameters vary by time period.
- If you think about approximating the value of being in a state, does this appear to be a relatively simple problem? If the value function is going to be very complex, it will be hard to approximate, making value function approximations hard to use. But if it is not too complex, value function approximations may be a very effective strategy.

Unless you are pursuing an algorithm as an intellectual exercise, it is best to focus on your problem and choose the method that is best suited to the application. For more complex problems, be prepared to use a hybrid strategy. For example, rolling horizon procedures may be combined with adjustments that depend on tunable parameters (a form of policy function approximation). You might use a tree search combined with a simple value function approximation to help reduce the size of the tree.

6.8 BIBLIOGRAPHIC NOTES

The goal of this chapter is to organize the diverse policies that have been suggested in the ADP and RL communities into a more compact framework. In the process, we are challenging commonly held assumptions, for example, that “approximate dynamic programming” always means that we are approximating value functions, even if this is one of the most popular strategies. Lookahead policies, and policy function approximations are effective strategies for certain problem classes.

Section 6.2 - Lookahead policies have been widely used in engineering practice in operations research under the name of rolling horizon procedure, and in computer science as a receding horizon procedure, and in engineering under the name model predictive control (Camacho & Bordons (2004)). Decision-trees are similarly a widely used strategy for which there are many references. Roll-out heuristics were introduced by Wu (1997) and Bertsekas & Castanon (1999). Stochastic programming, which combines uncertainty with multi-dimensional decision vectors, is reviewed in Birge & Louveaux (1997) and Kall & Wallace (1994), among others. Secomandi (2008) studies the effect of reoptimization on rolling horizon procedures as they adapt to new information.

Section 6.3 - While there are over 1,000 papers which refer to “value function approximation” in the literature (as of this writing), there were only a few dozen papers using “policy function approximation.” However, this is a term that we feel deserves more widespread use as it highlights the symmetry between the two strategies.

Section 6.4 - Making decisions which depend on an approximation of the value of being in a state has defined approximation dynamic programming since Bellman & Kalaba (1959).

PROBLEMS

6.1 Following is a list of how decisions are made in specific situations. For each, classify the decision function in terms of which of the four fundamental classes of policies are being used. If a policy function approximation or value function approximation is used, identify which functional class is being used:

- If the temperature is below 40 degrees F when I wake up, I put on a winter coat. If it is above 40 but less than 55, I will wear a light jacket. Above 55, I do not wear any jacket.
- When I get in my car, I use the navigation system to compute the path I should use to get to my destination.

- To determine which coal plants, natural gas plants and nuclear power plants to use tomorrow, a grid operator solves an integer program that plans over the next 24 hours which generators should be turned on or off, and when. This plan is then used to notify the plants who will be in operation tomorrow.
- A chess player makes a move based on her prior experience of the probability of winning from a particular board position.
- A stock broker is watching a stock rise from \$22 per share up to \$36 per share. After hitting \$36, the broker decides to hold on to the stock for a few more days because of the feeling that the stock might still go up.
- A utility has to plan water flows from one reservoir to the next, while ensuring that a host of legal restrictions will be satisfied. The problem can be formulated as a linear program which enforces these constraints. The utility uses a forecast of rainfalls over the next 12 months to determine what it should do right now.
- The utility now decides to capture uncertainties in the rainfall by modeling 20 different scenarios of what the rainfall might be on a month-by-month basis over the next year.
- A mutual fund has to decide how much cash to keep on hand. The mutual fund uses the rule of keeping enough cash to cover total redemptions over the last 5 days.
- A company is planning sales of TVs over the Christmas season. It produces a projection of the demand on a week by week basis, but does not want to end the season with zero inventories, so the company adds a function that provides positive value for up to 20 TVs.
- A wind farm has to make commitments of how much energy it can provide tomorrow. The wind farm creates a forecast, including an estimate of the expected amount of wind and the standard deviation of the error. The operator then makes an energy commitment so that there is an 80 percent probability that he will be able to make the commitment.

6.2 Earlier we considered the problem of assigning a resource i to a task j . If the task is not covered at time t , we hold it in the hopes that we can complete it in the future. We would like to give tasks that have been delayed more higher priority, so instead of just maximizing the contribution c_{ij} , we add in a bonus that increases with how long the task has been delayed, giving us the modified contribution

$$c_{ij}^{\pi}(\theta) = c_{ij} + \theta_0 e^{-\theta_1(\tau_j - t)}.$$

Now imagine using this contribution function, but optimizing over a time horizon T using forecasts of tasks that might arrive in the future. Would solving this problem, using $c_{ij}^{\pi}(\theta)$ as the contribution for covering task j using resource i at time t , give you the behavior that you want?