2.193 Decision-Making in Large-Scale Systems March 8
MIT, Spring 2005 Scribe: Yunqing Ye

Lecture Note 10

1 Curse of dimensionality

In all the Dynamic Programming algorithms we discussed in previous lectures, we need to:

1. compute and store functions of the states

2. compute the greedy policy, i.e.

$$\begin{aligned} u(x) &= \mathrm{argmin_a g_a(x)} + \alpha \sum_{\mathrm{y}} \mathrm{P_a(x,y)J(y)} \quad \text{or} \\ u(x) &= \mathrm{argmin_a Q(x,a)} \end{aligned}$$

All of them are tedious if the state space is large enough. In some specific cases, we can develop compact ways of storing J^*, like in the following example:

Example 1: In an LQ system, we have

$$\begin{aligned} x_{k+1} &= Ax_k + Ba_k & (1) \\ g_a(x) &= x^T \nu x + a^T N a & (2) \end{aligned}$$

We can prove that the optimal value function $J^*(x)$ has the form of

$$J^*(x) = x^T Q x \tag{3}$$

And from Bellman's equation, we can prove that Q satisfies a certain algebraic Ricatti equation. So instead of storing $J^*(x)$ for any possible $x \in \mathcal{R}^n$, we only need to store Q, which requires only $\mathcal{O}(n^2)$ space.

But in general, $J^*(x)$ will not have an exact compact representation. Thus we can only store it as a table, and the total storage space requirement grows linearly in $|S|$. More specifically, if there are N state variables and each state variable can take on m different values, then $|S| = m^N$ which grows exponentially with N.

So when the state space is large enough, due to computational limitation exact DP algorithms may fail to produce the answer in tolerable time. Thus, approximate DP algorithms are developed to facilitate decision making in large scale systems.

2 Approximate Dynamic Programming

There are two places in exact DP algorithm, where we can introduce approximation. One is in the policy domain, the other is in the cost-to-go function domain. We are going to approximate the optimal cost-to-go or policy function using parametized functions.

2.1 Policy Approximation

We view a policy $u(x) : \mathcal{S} \rightarrow \mathcal{A}$ as a mapping from state space to action space. One approach to circumventing the curse of dimensionality is based on approximating this function using a parameterized function $u_\theta(x), \theta \in \mathcal{R}^p$. A simple example of parametrization is "threshold policy", where we approximate $u(x)$ by $u(x) = u_1$ if x is within some threshold and $u(x) = u_2$ otherwise. Take the average cost problem for example, by parametrization, we translate the original problem from

$$\min_u \lambda_u \tag{4}$$

to

$$min_\theta \lambda_{u_\theta} \tag{5}$$

This is a nonlinear programming problem, and our main task reduces to how to compute the gradient $\frac{\partial \lambda}{\partial \theta}$. We will study this kind of method in detail later on in the class.

2.2 Cost-to-go function Approximation

Similar as policy iteration, we consider approximating the optimal cost-to-go function $J^*(x)$ by searching over a class of parametric functions, i.e. $J^*(x) \approx \tilde{J}(x, r), r \in \mathcal{R}^p$. Note that in exact DP, we need to take care of $J^*(x)$ which has a dimension of $|\mathcal{S}$, but after approximation, we only need to consider a p-dimensional problem. A typical approximation scheme is the linear architecture:

$$J^*(x) \approx \tilde{J}(x, r) = \sum_{i=1}^{k} r_i \phi_i(x) \tag{6}$$

where $\phi_i(x), i = 1, 2, \ldots, k$ are pre-defined basis functions.

Interesting questions arising from the linear architecture are how to pick up a good series of basis functions and how to find a good r given the basis functions.

Value function approximation, at first sight, is quite similar as the statistical regression method. In the latter, we have N samples (x_i, y_i), where y_i is the noisy version of $J^*(x_i)$: $y_i = J*(x_i) + \omega_i$. We can approximate $J^*(x)$ using a linear architecture:

$$J^*(x) \approx \sum_{i=1}^{p} r_i \phi_i(x), \tag{7}$$

where ϕ_i are called *basis functions*, by minimizing the following mean square error:

$$\min_r \sum_{j=1}^{N} \left(y_j - \sum_{i=1}^{p} r_i \phi_i(x_j) \right)^2 \tag{8}$$

but in DP problem, there is no way for us to get the "sample points" $(x_i, J^*(x_i))$, thus making approximate DP problem more difficult than statistical regression.

3 Approximate Architecture

Two nontrivial tasks arise in approximating optimal cost-to-go function:

1. How to choose an approximate approximation architecture $\tilde{J}(\cdot, r)$?

2. How to choose an approximate parameter r?

We will deal with the first issue in this section and discuss the latter later on in the class.

3.1 linear Architecture

In the linear architecture, we approximate $J*(x)$ using a linear combination of a series of basis functions, i.e.

$$J^*(x) \approx \tilde{J}(x,r) = \sum_{i-1}^{p} r_i \phi_i(x) \tag{9}$$

A simple choice is to have ϕ_i being polynomial functions of the state, e.g.:

$$\tilde{J}(x,r) = r_0 + r_1^T x + x^T r_2 x + \dots \tag{10}$$

where the basis functions are $1, x, x^2, x^3, \dots$.

Another simple but useful example is state aggregation. i.e. we partition the state space into p separated clusters: $S_1, S_2, \dots, S_p$, $S_i \cap S_j = \varnothing$. This corresponds to setting $\phi_i(x)$ to be the indicator function on set S_i, i.e.

$$\phi_i(x) = \mathbf{1}_{x \in S_i} = \begin{cases} 1 & x \in S_i \\ 0 & x \notin S_i \end{cases} \tag{11}$$

and $\tilde{J}(x,r)$ will be

$$\tilde{J}(x,r) = \sum_{i=1}^{p} r_i \phi_i(x) = \begin{cases} r_i & x \in S_i \\ 0 & x \notin S_i \end{cases} \tag{12}$$

Another example is using heuristics. Suppose we have a "good" policy $\tilde{u}$ at hand. We can set $\phi_1(x) = J_{\tilde{u}}(x) + \omega$, where ω is some kind of noise.

3.2 Nonlinear Architecture

Nonlinear architecture represents the general case of $\tilde{j}(x,r)$, which has far richer context than linear architecture. We can view linear architecture as a degenerate case of nonlinear architecture.

Two interesting examples of nonlinear architectures are:

1. Radial basis function(RBF) networks:
   We express the basis function as
   $$\phi_i(x, x_i, \sigma_i) = \exp\left(-\frac{\|x - x_i\|}{\sigma_i}\right) \tag{13}$$
   where x_is are the position of the "weeds" and σ_is are the scaling factor. Figure 1 represents a typical radial basis function approximation.

2. Neuro-network:
   We first transform state vector $x \in \mathcal{R}^n$ into an input vector $y \in \mathcal{R}^m$ by doing
   $$y_j = \sum_{i=1}^{n} r_{ij} x_i \tag{14}$$

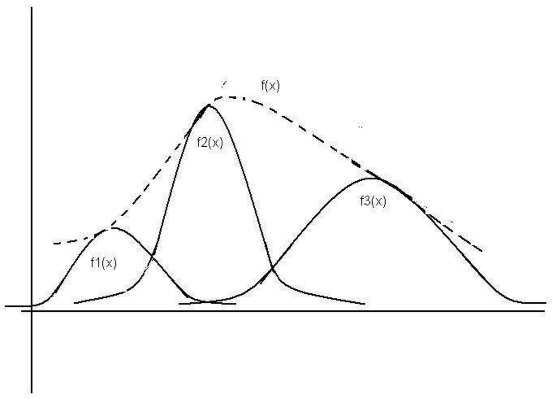

Figure 1: Radial basis function approximation

then we input y into the `sigmoidal layer`, which transforms y into another vector z by

$$z_i = f(y_i) \tag{15}$$

We restrict $f(\cdot)$, the `sigmoidal function` to be monotonically increasing, differentiable and bounded. The final output of the neural network is the weighed sum of z:

$$g(z) = \sum_i r_i z_i \tag{16}$$

It is not hard to see that a neural network represents a function $\tilde{J}(x, r)$, since x is the input, r is the set of weights used in the neural network and $g(z(x))$ is the value of $\tilde{J}(x, r)$. It is shown that neural network has universal approximation ability, but it is really difficult to find out the adequate set of weights r.

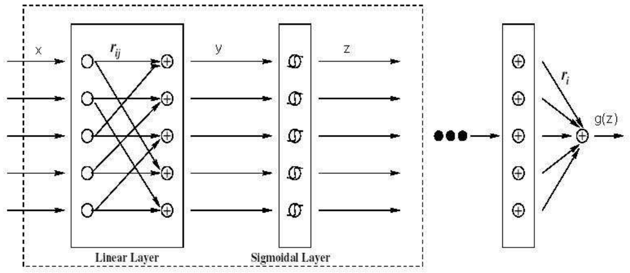

Figure 2: Neural Network