Lecture Note 10

1 Value Function Approximation

DP problems are centered around the cost-to-go function J^* or the Q-factor Q^*. In certain problems, such as linear-quadratic-Gaussian systems, J^* exhibits some structure which allows for its compact representation:

Example 1 *In LQG system, we have*

$$\begin{aligned} x_{k+1} &= Ax_k + Bu_k + Cw_k, \quad x \in \Re^n \\ g(x,u) &= x'Dx + u'Eu, \end{aligned}$$

where x_k represents the system state, u_k represents the control action, and w_k is a Gaussian noise. It can be shown that the optimal policy is of the form

$$u_k = L_k x_k$$

and the optimal cost-to-go function is of the form

$$J^*(x) = x'Rx + S, \quad R \in \Re^{n\times m}, S \in \Re$$

where R is a symmetric matrix. It follows that, if there are n state variables (i.e., $x_k \in \Re^n$), storing J^* requires storing $n(n+1)/2+1$ real numbers, corresponding to the matrix R and the scalar S. The computational time and storage space required is quadratic in the number of state variables.

In general, we are not as lucky as in the LQG system case, and exact representation of J^* requires that it be stored as a lookup table, with one value per state. Therefore, the space is proportional to the size of the state space, which grows exponentially with the number of state variables. This problem, known as the *curse of dimensionality*, makes dynamic programming intractable in face of most problems of practical scale.

Example 2 *Consider the game of Tetris, represented in Fig. 1. As seen in previous lectures, this game may be represented as an MDP, and a possible choice of state is the pair (B,P), in which $B \in \Re^{n\times m}$ represents the board configuration and P represents the current falling piece. More specifically, we have $b(i,j)=1$, if position (i,j) of the board is filled, and $b(i,j)=0$ otherwise.*

If there are p different types of pieces, and the board has dimension $n\times m$, the number of states is on the order of $p\times 2^{n\times m}$, which grows exponentially with n and m.

Since exact solution of large-scale of MDPs is intractable, we consider approximate solutions instead. There are two directions for approximations which are directly based on dynamic programming principles:

(1) **Approximation in the policy space**

Suppose that we would like to find a policy minimizing the average cost in an MDP. We can pose this

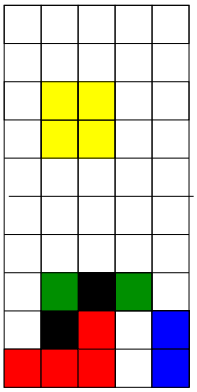

Figure 1: A tetris game

as a deterministic optimization problem, in the following way. Denote by $\lambda(u)$ the average cost of policy u. Then our problem corresponds to

$$\min_{u \in \mathcal{U}} \lambda(u), \tag{1}$$

where $\mathcal{U}$ is the set of all possible policies. In principle, we could solve (1) by enumerating all policies and choosing the one with the smallest value of $\lambda(u)$; however, note that the number of policies is exponential in the number of states — we have $|\mathcal{Y}| = |A|^{|\mathcal{S}|}$; if there is no special structure to $\mathcal{U}$, this problem requires even more computational time than solving Bellman's equation for the cost-to-go function. A possible approach to approximating the solution is to transform problem (1) by considering only a tractable subset of all policies:

$$\min_{u \in F} \lambda(u)$$

where F is a subset of the policy space. If F has some appropriate format, e.g., we consider policies that are parameterized by a continuous variable, we may be able to solve this problem without having to enumerate all policies in the set, but by using some standard optimization method such as gradient descent. Methods based on this idea are called approximations in the policy space, and will be studied later on in this class.

(2) **Cost-to-go Function Approximation**

Another approach to approximating the dynamic programming solution is to approximate the cost-to-go function. The underlying idea for cost-to-go function approximation is that J^* has some structure that allows for approximate compact representation

$$J^*(x) \approx \tilde{J}(x, r), \qquad \text{for some parameter } r \in \Re^P.$$

If we restrict ourselves to approximations of this form, the problem of computing and storing J^* is reduced to computing and storing the parameter r, which requires considerably less space. Some examples of approximation architectures $\tilde{J}$ that may be considered are as follows:

Example 3

$$\begin{array}{ll} \tilde{J}(x,r) = \cos(x^T r) & \textit{nonlinear in } r \\ \left.\begin{array}{ll} \tilde{J}(x,r) & = r_0 + r_1^T x \\ \tilde{J}(x,r) & = r_0 + r_1^t \phi(x) \end{array}\right\} & \textit{linear in } r \end{array}$$

In the next few lectures, we will focus on cost-to-go function approximation. Note that there are two important preconditions to the development of an effective approximation. First, we need to choose a parameterization $\tilde{J}$ that can closely approximate the desired cost-to-go function. In this respect, a suitable choice requires some practical experience or theoretical analysis that provides rough information on the shape of the function to be approximated. "Regularities" associated with the function, for example, can guide the choice of representation. Designing an approximation architecture is a problem-specific task and it is not the main focus of this paper; however, we provide some general guidelines and illustration via case studies involving queueing problems. Second, given a parameterization for the cost-to-go function approximation, we need an efficient algorithm that computes appropriate parameter values.

We will start by describing usual choices for approximation architectures.

2 Approximation Architectures

2.1 Neural Networks

A common choice for an approximation architecture are neural networks. Fig. **??** represents a neural network. The underlying idea is as follows: we first convert the original state x into a vector $\bar{x} \in \Re^n$, for some n. This vector is used as the input to a *linear layer* of the neural network, which maps the input to a vector $y \in \Re^m$, for some m, such that $y_j = \sum_{i=1}^{n} r_{ij}\bar{x}_i$. The vector y is then used as the input to a *sigmoidal layer*, which outputs a vector $z \in \Re^m$ with the property that $z_i = f(y_i)$, and $f(.)$ is a *sigmoidal function*. A sigmoidal function is any function with the following properties:

1. monotonically increasing
2. differentiable
3. bounded

Fig. 3 represents a typical sigmoidal function.

The combination of a linear and a sigmoidal layer is called a *perceptron*, and a neural network consists of a chain of one or more perceptrons (i.e., the output of a sigmoidal layer can be redirected to another sigmoidal layer, and so on). Finally, the output of the neural network consists of a weighted sum of the output z of the final layer:

$$g(z) = \sum_i r_i z_i.$$

It is easy to see that a neural network represents a function $\tilde{J}(x,r)$, where x is the input and r is the set of weights in each of the perceptrons. Recall that we are interested in representing a function $J^*(x)$ as $\tilde{J}(x,r)$, for some set of weights r. Part of the appeal of neural networks is that they can be efficiently trained to fit a function $f(x)$ based on samples $(x_i, f(x_i))$ via an algorithm known as backpropagation. Moreover, they are flexible enough to adequately represent a wide class of functions — indeed, it can be shown given a class

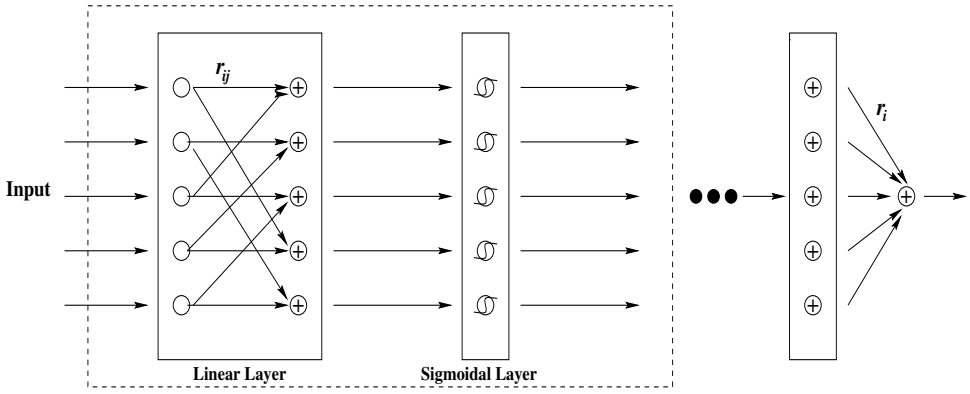

Figure 2: A neural network

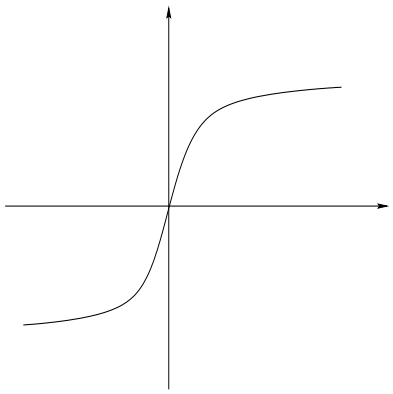

Figure 3: A sigmoidal function

of functions on some bounded and closed set, if functions are "uniformly smooth," we can get error $O(\frac{1}{n})$ with n sigmoidal functions. (Barron 1990). Note, however, that in order to obtain a good approximation, an adequate set of weights r must be found. Backpropagation, which is simply a gradient descent algorithm, is able to find a local optimum among all set of weights, but finding the global optimum may be a difficult problem.

2.2 State Space Partitioning

Another common choice for approximation architecture is based on partitioning of the state space. The underlying idea is that "similar" states may be grouped together. For instance, in an MDP involving continuous state variables (e.g., $\mathcal{S} = \Re^2$), one may consider partitioning the state space by using a grid (e.g., divide $\Re^2$ in squares). The simplest case would be to use a uniform grid, and assume that the cost-to-go function remains constant in each of the partitions. Alternatively, one may use functions that are linear in each of the partitions, or splines, and so on. One may also consider other kinds of partition beyond uniform grids — representing the partitioning of the state space as a tree or using adaptive methods for choosing the partitions, for instance.

2.3 Features

A special case of state space partitioning consists of mapping states to *features*, and considering approximations of the cost-to-go function that are functions of the features. The hope is that the *feature* would capture aspects of the state that are relevant for the decision-making process and discard irrelevant details, thus providing a more compact representation. At the same time, one would also hope that, with an appropriate choice of features, the mapping from features to the (approximate) cost-to-go function would be smoother than that from the original state to the cost-to-go function, thereby allowing for successful approximation with architectures that are suitable for smooth mappings (e.g., polynomials). This process is represented below.

$$\text{State } x \longrightarrow \text{ features } f(x) \longrightarrow \tilde{J}(f(x), r).$$

$$J^*(x) \approx \bar{J}\left(f(x)\right) \text{ such that } f(x) \to \bar{J} \text{ is smooth.}$$

Example 4 *Consider the tetris game. What features we should choose?*

1. $|h(i) - h(i+1)|$ *(height)*
2. *how many holes*
3. *max* $h(i)$