6.231 DYNAMIC PROGRAMMING

LECTURE 24

LECTURE OUTLINE

- Additional methods for approximate DP
- Q-Learning
- Aggregation
- Linear programming with function approximation
- Gradient-based approximation in policy space

Q-LEARNING I

- To implement an optimal policy, what we need are the Q-factors defined for each pair (i, u) by

$$Q(i,u) = \sum_j p_{ij}(u)\big(g(i,u,j) + J^*(j)\big)$$

- Bellman's equation is $J^*(j) = \min_{u' \in U(j)} Q(j,u')$, so the Q-factors solve the system of equations

$$Q(i,u) = \sum_j p_{ij}(u)\Big(g(i,u,j) + \min_{u' \in U(j)} Q(j,u')\Big), \ \forall\, (i,u)$$

- One possibility is to solve this system iteratively by a form of value iteration

$$Q(i,u) := (1-\gamma)Q(i,u) + \gamma \sum_j p_{ij}(u)\Big(g(i,u,j) + \min_{u' \in U(j)} Q(j,u')\Big),$$

where γ is a stepsize parameter with $\gamma \in (0,1]$, that may change from one iteration to the next.

Q-LEARNING II

- The $Q\text{-}learning\ method$ is an approximate version of this iteration, whereby the expected value is replaced by a single sample, i.e.,

$$Q(i,u) := Q(i,u) + \gamma\Big(g(i,u,j) + \min_{u' \in U(j)} Q(j,u') - Q(i,u)\Big)$$

- Here j and $g(i,u,j)$ are generated from the pair (i,u) by simulation, i.e., according to the transition probabilities $p_{ij}(u)$.

- Thus Q-learning can be viewed as a combination of value iteration and simulation.

- Convergence of the method to the (optimal) Q factors can be shown under some reasonable (but quite technical) assumptions.

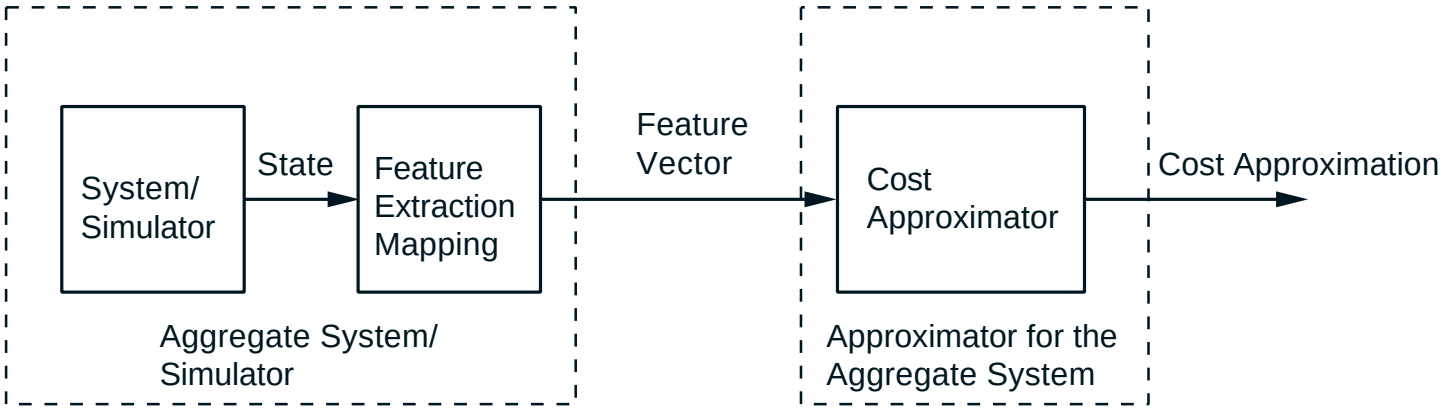
- Strong connections with the theory of stochastic iterative algorithms (such as stochastic gradient methods).

- Challenging analysis, limited practicality (only for a small number of states).

AGGREGATION I

- Another major idea in approximate DP is to approximate the cost-to-go function of the problem with the cost-to-go function of a simpler problem.
- The main idea of aggregation approach:
  - Lump many states together into a few "aggregate" states
  - View the aggregate states as the states of an "aggregate" system
  - Formulate and solve (optimally) the "aggregate" problem by any kind of value or policy iteration method (including simulation-based methods, such as Q-learning)
  - Use the optimal cost of the aggregate problem for a piecewise-constant approximation of the optimal cost of the original problem (all states that belong to the same aggregate state are restricted to have the same cost, the optimal cost of the aggregate state)
- The aggregate problem could also be solved approximately.

AGGREGATION II



- Main steps to define the aggregate system
- Form the aggregate states by partitioning the original state space (features can be used for this).
  - Each aggregate state is a subset S of states of the original system
  - Each state of the original system belongs to a unique aggregate state
- Define the dynamics of the aggregate system

$$\text{Current aggregate state } S \;\mapsto\; \text{New aggregate state } S'$$

Example: If the current aggregate state is S, generate a "typical" state i within S in some probabilistic way, then generate j according to the p_{ij}, then declare S' to be the aggregate state to which j belongs.

SOFT AGGREGATION

- A more general approach is to specify that each original system state j "belongs to each aggregate state k with prescribed probability π_{jk}." Then find the costs $\tilde{J}_k$ of the aggregate states by solving an aggregate problem, and approximate the cost of an original system state j by $\sum_k \pi_{jk} \tilde{J}_k$.

- Define the dynamics of the aggregate system as follows: from the current aggregate state, generate a "typical" state i in some probabilistic way, then generate j according to the p_{ij}, then generate probabilistically the next aggregate state k according to probabilities π_{jk}.

- A variant of this approach when the aggregate states are themselves states of the original system (so aggregation here represents a coarse discretization of the original state space).
  - Define the dynamics of the aggregate system as follows: from the current aggregate state, generate a next state j according to the original system transition probabilities, then generate the next aggregate state k according to probabilities π_{jk}.

APPROXIMATE LINEAR PROGRAMMING

- Approximate J^* using a linear architecture

$$\tilde{J} = \Phi r$$

where $r = (r_1, \ldots, r_s)$ is a weight vector, and Φ is an $n \times s$ feature matrix.

- Use $\tilde{J}$ in place of J^* in the linear programming approach, i.e., compute r by solving

$$\begin{aligned} &\text{maximize} \quad c' \Phi r \\ &\text{subject to} \quad \Phi r \leq g_\mu + \alpha P_\mu \Phi r, \quad \forall\, \mu \end{aligned}$$

where c is a vector with positive components.

- This is a linear program with s variables but an enormous number of constraints (one constraint for each state-control pair).

- Special large-scale linear programming methods (cutting plane or column generation methods) may be used for such problems.

- Approximations using only a "sampled" subset of state-control pairs are possible (see the papers by de Farias and Van Roy).

APPROXIMATION IN POLICY SPACE I

• Consider an average cost problem, where the problem data are parameterized by a vector r, i.e., a cost vector $g(r)$, transition probability matrix $P(r)$. Let $\lambda(r)$ be the (scalar) average cost per stage, satisfying Bellman's equation

$$\lambda(r)e + v(r) = g(r) + P(r)v(r)$$

• Consider minimizing $\lambda(r)$ over r (here the data dependence on control is encoded in the parameterization). We can try to solve the problem by nonlinear programming/gradient descent methods.

• **Important fact:** If $\Delta\lambda$ is the change in λ due to a small change Δr from a given r, we have

$$\Delta\lambda \cdot e = \overline{p}'(\Delta g + \Delta P v),$$

where $\overline{p}$ is the steady-state probability distribution/vector corresponding to $P(r)$, and all the quantities above are evaluated at r:

$$\Delta\lambda = \lambda(r + \Delta r) - \lambda(r),$$

$$\Delta g = g(r+\Delta r) - g(r), \qquad \Delta P = P(r+\Delta r) - P(r)$$

APPROXIMATION IN POLICY SPACE II

• **Proof of the gradient formula:** We have, by "differentiating" Bellman's equation,

$$\Delta\lambda(r)\cdot e+\Delta v(r) = \Delta g(r)+\Delta P(r)v(r)+P(r)\Delta v(r)$$

By left-multiplying with $\overline{p}'$,

$$\overline{p}'\Delta\lambda(r)\cdot e+\overline{p}'\Delta v(r) = \overline{p}'\Big(\Delta g(r)+\Delta P(r)v(r)\Big)+\overline{p}'P(r)\Delta v(r)$$

Since $\overline{p}'\Delta\lambda(r)\cdot e = \Delta\lambda(r)e$ and $\overline{p}' = \overline{p}'P(r)$, this equation simplifies to

$$\Delta\lambda\cdot e = \overline{p}'(\Delta g + \Delta P v)$$

• Since we don't know $\overline{p}$, we cannot implement a gradient-like method for minimizing $\lambda(r)$. An alternative is to use "sampled gradients", i.e., generate a simulation trajectory $(i_0, i_1, \ldots)$, and change r once in a while, in the direction of a simulation-based estimate of $\overline{p}'(\Delta g + \Delta P v)$.

• There is much recent research on this subject, see e.g., the work of Marbach and Tsitsiklis, and Konda and Tsitsiklis.