A Series of Lectures on Approximate Dynamic Programming

Dimitri P. Bertsekas

Laboratory for Information and Decision Systems
Massachusetts Institute of Technology

Lucca, Italy
June 2017

APPROXIMATE DYNAMIC PROGRAMMING I

Outline

Recall the Basic Problem Structure for DP

Discrete-time system

$$x_{k+1} = f_k(x_k, u_k, w_k), \qquad k = 0, 1, \ldots, N-1$$

- x_k: State
- u_k: Control from a constraint set $U_k(x_k)$
- w_k: Disturbance; random parameter with distribution $P(w_k \mid x_k, u_k)$

Optimization over Feedback Policies $\pi = \{\mu_0, \mu_1, \ldots, \mu_{N-1}\}$, with $u_k = \mu_k(x_k) \in U(x_k)$

Cost of a policy starting at initial state x_0:

$$J_\pi(x_0) = E\left\{ g_N(x_N) + \sum_{k=0}^{N-1} g_k\left(x_k, \mu_k(x_k), w_k\right) \right\}$$

Optimal cost function:

$$J^*(x_0) = \min_\pi J_\pi(x_0)$$

Recall the Exact DP Algorithm

Computes for all k and states x_k: $J_k(x_k)$, the opt. cost of tail problem that starts at x_k

Go backwards, $k = N-1, \ldots, 0$, using

$$J_N(x_N) = g_N(x_N)$$

$$J_k(x_k) = \min_{u_k \in U_k(x_k)} E\Big\{ g_k(x_k, u_k, w_k) + J_{k+1}\big(f_k(x_k, u_k, w_k)\big) \Big\}$$

Notes:

- $J_0(x_0) = J^*(x_0)$: Cost generated at the last step, is equal to the optimal cost
- Let $\mu_k^*(x_k)$ minimize in the right side above for each x_k and k. Then the policy $\pi^* = \{\mu_0^*, \ldots, \mu_{N-1}^*\}$ is optimal
- Potentially ENORMOUS computational requirements
- IF we knew J_{k+1}, the computation of the minimizing u_k would be much simpler

One-Step and Multistep Lookahead

One-Step Lookahead

- Replace J_{k+1} by an approximation $\tilde{J}_{k+1}$
- Apply $\bar{u}_k$ that attains the minimum in

$$\min_{u_k \in U_k(x_k)} E\Big\{ g_k(x_k, u_k, w_k) + \tilde{J}_{k+1}\big(f_k(x_k, u_k, w_k)\big) \Big\}$$

ℓ-Step Lookahead

- At state x_k solve the ℓ-step DP problem starting at x_k and using terminal cost $\tilde{J}_{k+\ell}$
- If $\overline{u}_k, \overline{\mu}_{k+1}, \ldots, \overline{\mu}_{k+\ell-1}$ is an optimal policy for the ℓ-step problem, apply the first control $\overline{u}_k$

Notes

- Other names used: Rolling or receding horizon control
- A key issue: How do we choose $\tilde{J}_{k+\ell}$?
- Another issue: How do we deal with the minimization and the computation of $E\{\cdot\}$
- Implementation issues; e.g., tradeoff between on-line vs off-line computation
- Performance issues; e.g., error bounds (we will not cover)

A Summary of Approximation Possibilities in Value Space

At State x_k

DP minimization

(Could be approximate)

First ℓ Steps

"Future"

$$\min_{u_k,\mu_{k+1},\ldots,\mu_{k+\ell-1}} E\left\{ g_k(x_k,u_k,w_k) + \sum_{m=k+1}^{k+\ell-1} g_k\big(x_m,\mu_m(x_m),w_m\big) + \tilde{J}_{k+\ell}(x_{k+\ell}) \right\}$$

Approximations:

Replace $E\{\cdot\}$ with nominal values
(certainty equivalent control)

Limited simulation
(Monte Carlo tree search)

Computation of $\tilde{J}_{k+\ell}$:

Simple choices

Parametric approximation

Tail problem approximation

Rollout

A First-Order Division of Lookahead Choices

Long lookahead ℓ and simple choice of $\tilde{J}_{k+\ell}$

- Some examples

$$\tilde{J}_{k+\ell}(x) \equiv 0 \qquad \text{(or a constant)}$$

$$\tilde{J}_{k+\ell}(x) = g_N(x)$$

For problems with a “goal state" use a simple penalty $\tilde{J}_{k+\ell}$

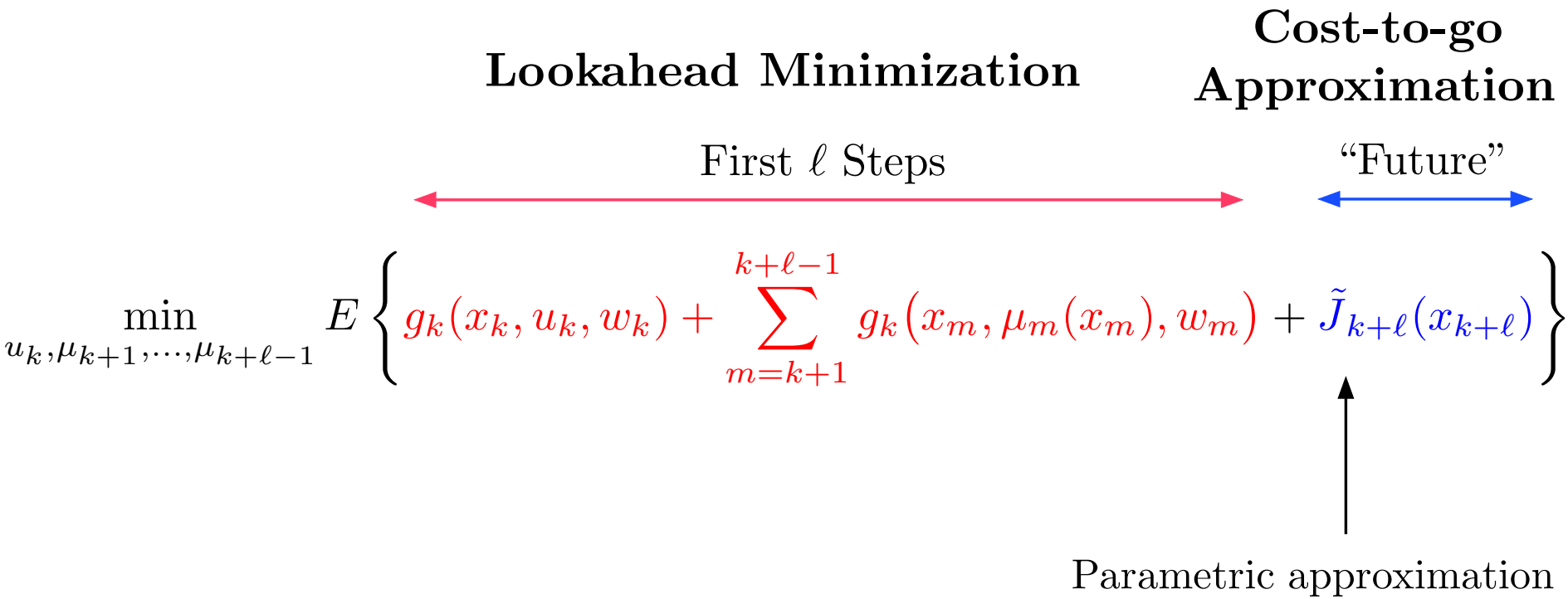
$$\tilde{J}_{k+\ell}(x) = \begin{cases} 0 & \text{if } x \text{ is a goal state} \\ >> 1 & \text{if } x \text{ is not a goal state} \end{cases}$$

- Long lookahead $\Longrightarrow$ A lot of DP computation
- Often must be done off-line

Short lookahead ℓ and sophisticated choice $\tilde{J}_{k+\ell} \approx J_{k+\ell}$

- The lookahead cost function approximates (to within a constant) the optimal cost-to-go produced by exact DP
- We will next describe a variety of off-line and on-line approximation approaches

Approximation in Value Space



Parametric Approximation: Approximation Architectures

- We approximate $J_k(x_k)$ with a function from an approximation architecture, i.e., a parametric class $\tilde{J}_k(x_k, r_k)$, where $r_k = (r_{1,k}, \ldots, r_{m,k})$ is a vector of "tunable" scalar weights
- We use $\tilde{J}_k$ in place of J_k (the optimal cost-to-go function) in a one-step or multistep lookahead scheme
- Role of r_k: By adjusting r_k we can change the "shape" of $\tilde{J}_k$ so that it is "close" to to the optimal J_k (at least within a constant)

Two key Issues

- The choice of the parametric class $\tilde{J}_k(x_k, r_k)$; there is a large variety
- The method for tuning/adjusting the weights ("training" the architecture)

Feature extraction

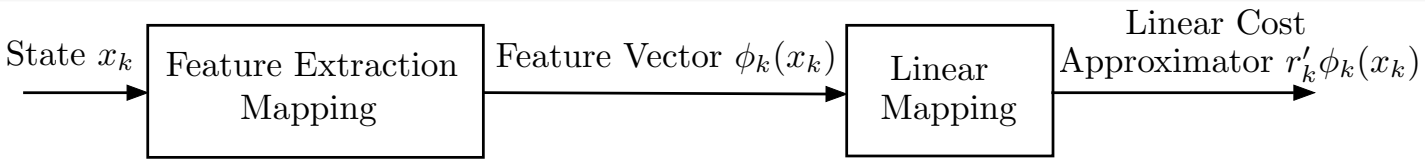
- A process that maps the state x_k into a vector $\phi_k(x_k) = \big(\phi_{1,k}(x_k), \ldots, \phi_{m,k}(x_k)\big)$, called the feature vector associated with x_k
- A feature-based cost approximator has the form

$$\tilde{J}_k(x_k, r_k) = \hat{J}_k\big(\phi_k(x_k), r_k\big)$$

where r_k is a parameter vector and $\hat{J}_k$ is some function, linear or nonlinear in r_k

- With a well-chosen feature vector $\phi_k(x_k)$, a good approximation to the cost-to-go is often provided by linearly weighting the features, i.e.,

$$\tilde{J}_k(x_k, r_k) = \hat{J}_k\big(\phi_k(x_k), r_k\big) = \sum_{i=1}^{m} r_{i,k}\phi_{i,k}(x_k) = r_k'\phi_k(x_k)$$

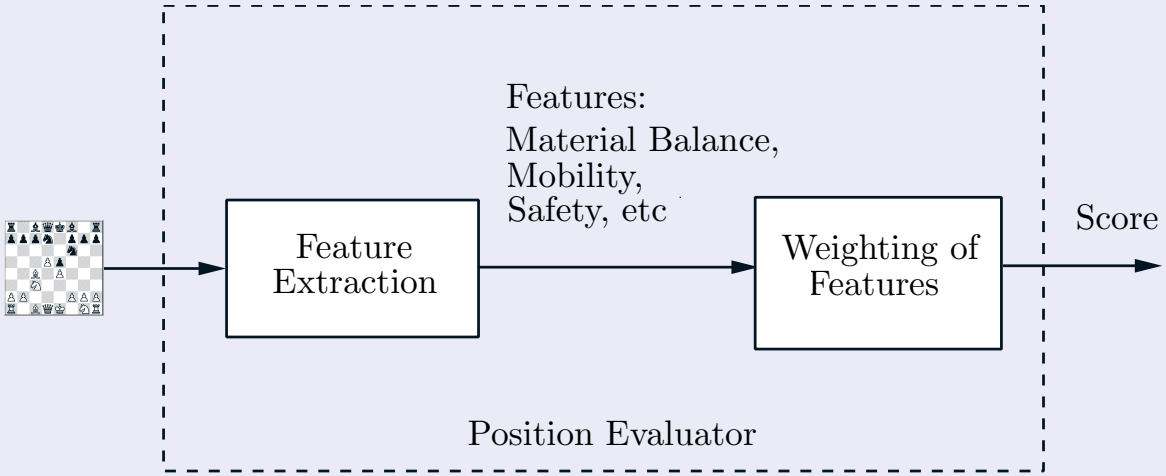


This can be viewed as approximation onto a subspace of basis functions of x_k defined by the features $\phi_{i,k}(x_k)$

Feature-Based Architectures

- Any generic basis functions, such as classes of polynomials, wavelets, radial basis functions, etc, can serve as features
- In some cases, problem-specific features can be “hand-crafted"

Computer chess example



- Think of state: board position; control: move choice
- Use a feature-based position evaluator assigning a score to each position
- Most chess programs use a linear architecture with “manual" choice of weights
- Some computer programs choose the weights by a least squares fit using lots of grandmaster play examples

An Example of Architecture Training: Sequential DP Approximation

A common way to train architectures $\tilde{J}_k(x_k, r_k)$ in the context of DP

- We start with $\tilde{J}_N = g_N$ and sequentially train going backwards, until $k = 1$
- Given a cost-to-go approximation $\tilde{J}_{k+1}$, we use one-step lookahead to construct a large number of state-cost pairs (x_k^s, β_k^s), $s = 1, \ldots, q$, where

$$\beta_k^s = \min_{u \in U_k(x_k^s)} E\Big\{ g(x_k^s, u, w_k) + \tilde{J}_{k+1}\big(f_k(x_k^s, u, w_k), r_{k+1}\big) \Big\}, \qquad s = 1, \ldots, q$$

- We "train" an architecture $\tilde{J}_k$ on the training set (x_k^s, β_k^s), $s = 1, \ldots, q$

Training by least squares/regression

- We minimize over r_k

$$\sum_{s=1}^{q} \big(\tilde{J}_k(x_k^s, r_k) - \beta^s\big)^2 + \gamma \|r_k - \bar{r}\|^2$$

  where $\bar{r}$ is an initial guess for r_k and $\gamma > 0$ is a regularization parameter
- Special algorithms called incremental gradient methods are typically used for this. They take advantage of the large sum structure of the cost function
- For a linear architecture the training problem is a linear least squares problem

Neural Networks for Constructing Cost-to-Go Approximations $\tilde{J}_k$

Neural nets can be used in the preceding sequential DP approximation scheme: Train the stage k neural net using a training set generated with the stage $k+1$ neural net

Two ways to view neural networks

- As nonlinear approximation architectures
- As linear architectures with automatically constructed features

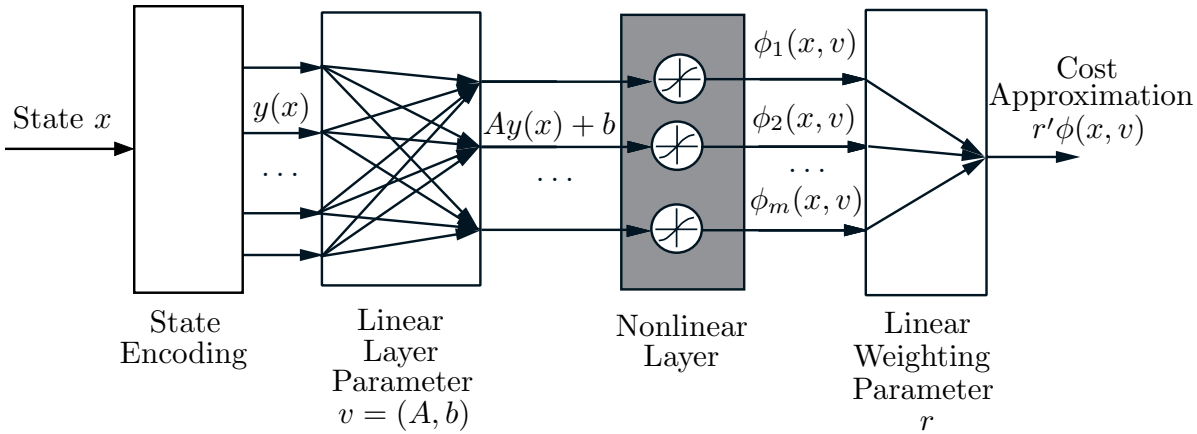
Focus at the typical stage k and drop the index k for convenience

- Neural nets are approximation architectures of the form

$$\tilde{J}(x, v, r) = \sum_{i=1}^{m} r_i \phi_i(x, v) = r' \phi(x, v)$$

  involving two parameter vectors r and v with different roles
- View $\phi(x, v)$ as a feature vector; view r as a vector of linear weighting parameters for $\phi(x, v)$
- By training v jointly with r, we obtain automatically generated features!

Neural Network with a Single Nonlinear Layer

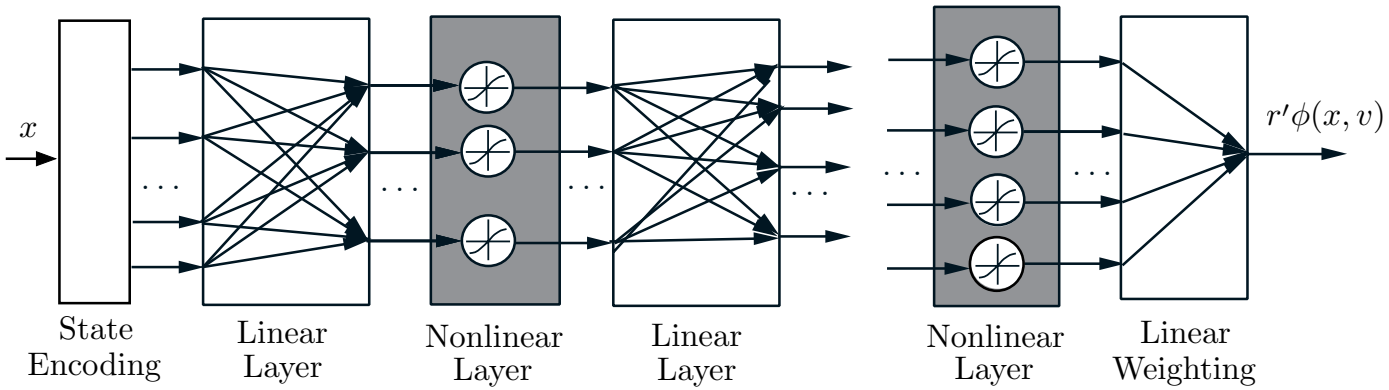


- State encoding (could be the identity, could include special features of the state)
- Linear layer $Ay(x) + b$ [parameters to be determined: $v = (A, b)$]
- Nonlinear layer produces m outputs $\phi_i(x, v) = \sigma\Big(\big(Ay(x) + b\big)_i\Big)$, $i = 1, \ldots, m$
- σ is a scalar nonlinear differentiable function; several types have been used (hyperbolic tangent, logistic, rectified linear unit)
- Training problem is to use the training set (x^s, β^s), $s = 1, \ldots, q$, for

$$\min_{A,b,r} \sum_{s=1}^{q} \left(\sum_{i=1}^{m} r_i \, \sigma\left(\big(Ay(x^s) + b\big)_i\right) - \beta^s \right)^2 + (\text{Regularization Term})$$

- Solved often with incremental gradient methods (known as backpropagation)
- Universal approximation theorem: With sufficiently large number of parameters, arbitrarily complex functions can be closely approximated

Deep Neural Networks



- More complex NNs are formed by concatenation of multiple layers
- The outputs of each nonlinear layer become the inputs of the next linear layer
- Considerable success has been achieved in major contexts

Possible reasons for the success

- The multilayer network provides a hierarchy of features (each set of features being a function of the preceding set of features) that can be exploited to specialize the role of some of the layers
- We may use matrices A with a special structure that encodes special linear operations such as convolution
- When such structures are used, the training problem may become easier, because the number of parameters in the linear layers is drastically decreased

Approximation in Q-Factor Space: Using a Simulator Instead of a Model

- The Q-factor of a state-control pair (x_k, u_k) at time k is defined by

$$Q_k(x_k, u_k) = E\Big\{ g_k(x_k, u_k, w_k) + J_{k+1}(x_{k+1}) \Big\}$$

  where J_{k+1} is the optimal cost-to-go function for stage $k+1$
- Note that $J_k(x_k) = \min_{u \in U_k(x_k)} Q_k(x_k, u_k)$; the DP algorithm is written in terms of Q_k
- Consider sequential DP approximation of Q-factor parametric approximations

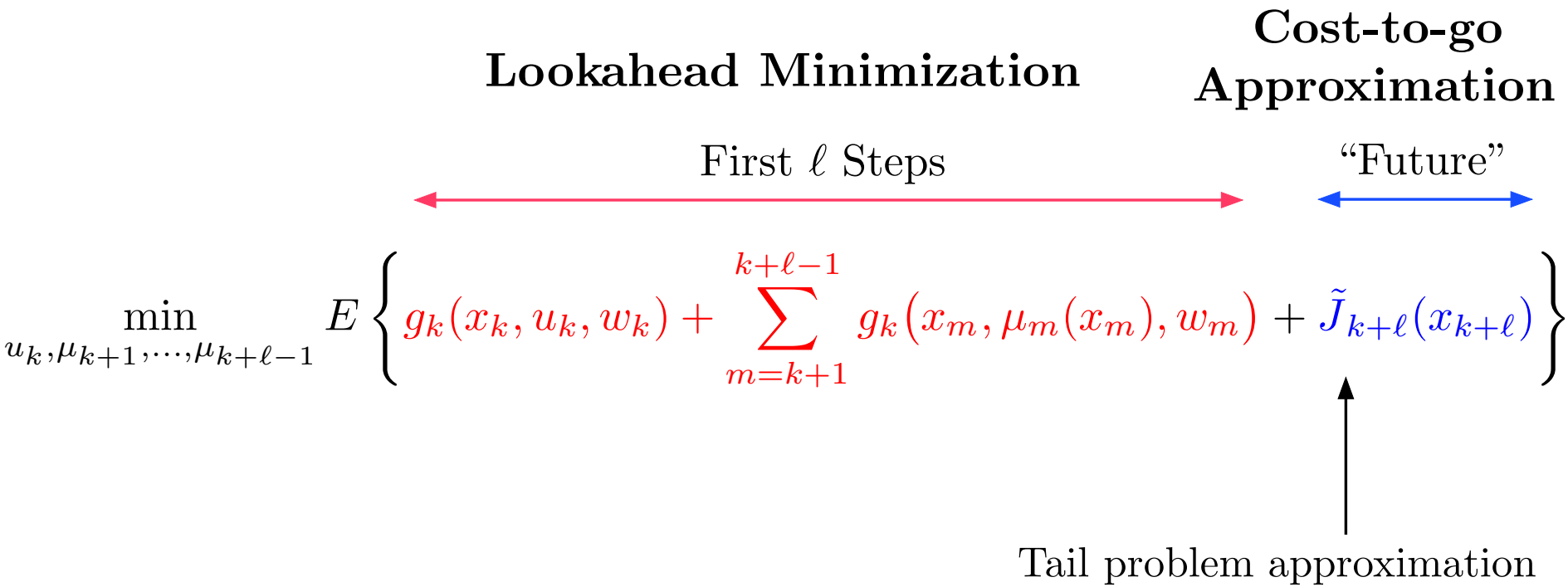
$$\tilde{Q}_k(x_k, u_k, r_k) = E\Big\{ g_k(x_k, u_k, w_k) + \min_{u \in U_{k+1}(x_{k+1})} \tilde{Q}_{k+1}(x_{k+1}, u, r_{k+1}) \Big\}$$

- We obtain $\tilde{Q}_k(x_k, u_k, r_k)$ by training with many pairs $\big((x_k^s, u_k^s), \beta_k^s\big)$, where β_k^s is a sample of the approximate Q-factor of (x_k^s, u_k^s). [No need to compute $E\{\cdot\}$]
- Note: No need for a model to obtain β_k^s. Sufficient to have a simulator that generates state-control-cost-next state samples $\big((x_k, u_k), (g_k(x_k, u_k, w_k), x_{k+1})\big)$
- Having computed r_k, the one-step lookahead control is obtained on-line as

$$\overline{\mu}_k(x_k) = \arg \min_{u \in U_k(x_k)} \tilde{Q}_k(x_k, u, r_k)$$

  without the need of a model or expected value calculations

Approximation in Value Space



Tail Problem Approximation Ideas

Obtain $\tilde{J}_{k+\ell}$ as the cost-to-go of a simplified problem
which is solved exactly or approximately

Enforced decomposition of interconnected subsystems

Applies to problems involving a collection I of interconnected subsystems, with each subsystem $i \in I$ applying control u_k^i at time k

- One-at-a time optimization: Obtain $\tilde{J}_{k+\ell}$ by optimizing one subsystem at a time, with controls of other subsystems fixed at nominal values
- Constraint relaxation: Artificially decouple subsystems by modifying the constraint set
- Lagrangean relaxation: Artificially decouple subsystems by using Lagrange multipliers (we will not cover)

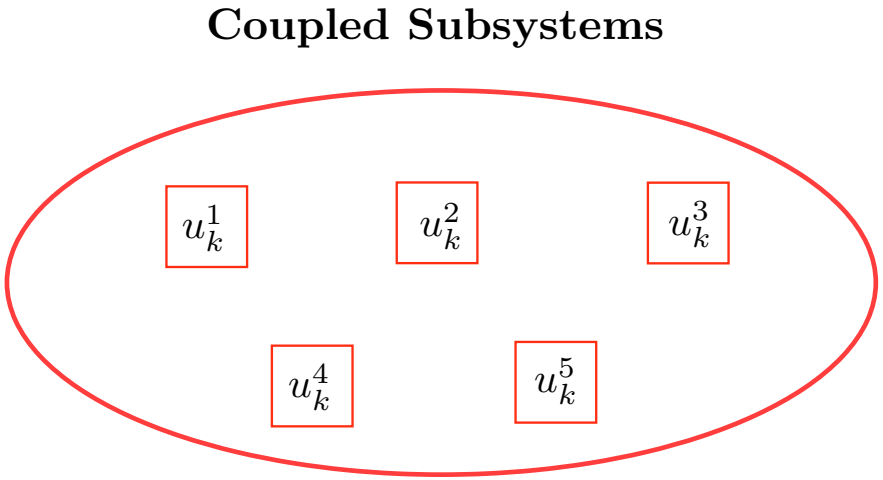
Probabilistic approximation

Simplify the probabilistic structure (e.g., replace random variables with deterministic)

Aggregation

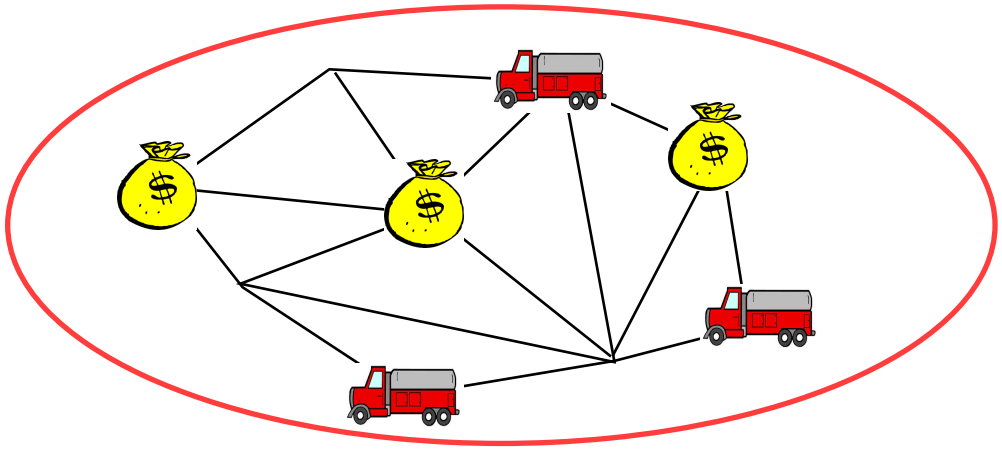
Reduce the size of the problem; e.g., by “combining" states into aggregate states

Enforced Decomposition: One Subsystem at a Time



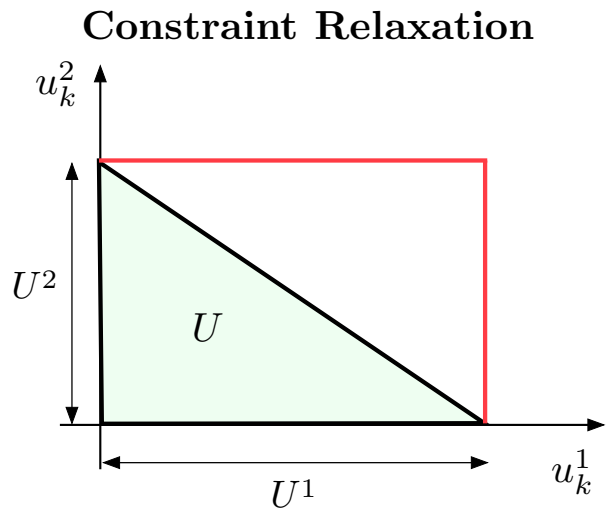
- Let $u_k = (u_k^1, \ldots, u_k^n)$, with u_k^i corresponding to the ith subsystem
- To compute cost-to-go approximation $\tilde{J}_k(x_k)$:
  - Start with subsystem 1, optimize over $(u_k^1, \ldots, u_{N-1}^1)$, with all future controls of other subsystems $i \neq 1$ held at nominal values $(\tilde{u}_k^i, \ldots, \tilde{u}_{N-1}^i)$
  - Fix the nominal values of subsystem 1 to the optimal sequence thus obtained
  - Repeat for all subsystems $i = 2, \ldots, n$ (with intermediate adjustment of the nominal control values)
- The scheme applies to both deterministic and stochastic problems

Example: Optimize the Routes of n Vehicles Through a Road Network



- Aim: Execute a number of tasks with given values
- The value of a task is collected only once; a finite horizon is assumed
- This is a very complex combinatorial problem
- The single vehicle problem is typically much simpler (e.g., can be solved exactly or with a high-quality heuristic)
- Do one-step-lookahead with (suboptimal) optimization of the tail subproblem one-vehicle-at-a-time. The nominal decisions of the other vehicles can be determined using some heuristic

Enforced Decomposition: Constraint Decoupling by Relaxation



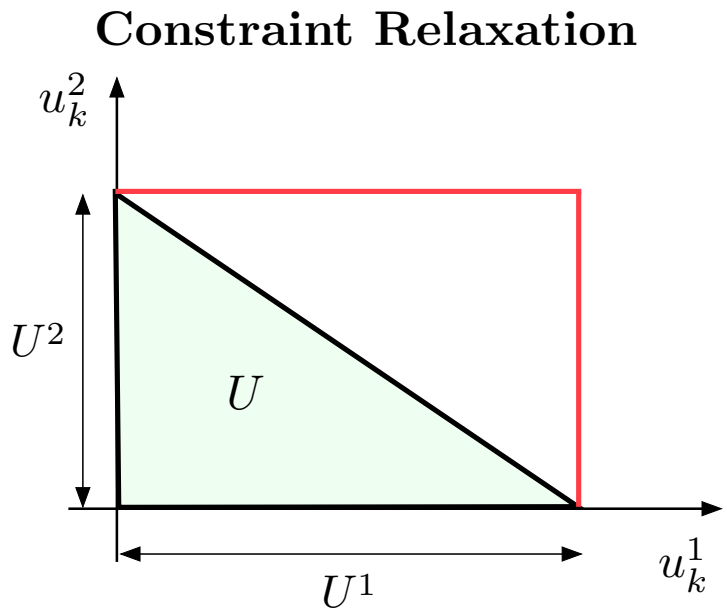
- Let $x_k = (x_k^1, \ldots, x_k^n)$, $u_k = (u_k^1, \ldots, u_k^n)$, $w_k = (w_k^1, \ldots, w_k^n)$, with (x_k^i, u_k^i, w_k^i) corresponding to the ith subsystem
- Assume that the only coupling between subsystems is the control constraint

$$(u_k^1, \ldots, u_k^n) \in U, \qquad \text{e.g., } u_k^i \in U^i,\ u_k^1 + \cdots + u_k^n \leq b_k$$

- Approximate U with a decomposed constraint $U^1 \times \ldots \times U^n$
- The problem decomposes into n decoupled subproblems. Let $\tilde{J}_k^i$ be the optimal cost to go functions for the ith decoupled subproblem (obtained by DP off-line)
- Use one-step lookahead with cost-to-go approximation

$$\tilde{J}_{k+1}(x_{k+1}) = \tilde{J}_{k+1}^1(x_{k+1}^1) + \cdots + \tilde{J}_{k+1}^n(x_{k+1}^n)$$

Example: Production Planning



A work center producing n product types

- x_k^i, u_k^i, w_k^i: the amounts stored, produced, and demanded of product i at time k
- State is the stock vector $x_k = (x_k^1, \ldots, x_k^n)$, where $x_{k+1}^i = x_k^i + u_k^i - w_k^i$
- U represents the (shared) production capacity of the work center
- In a more complex version (involving equipment failures), U depends on a random parameter α_k that changes according to a Markov chain

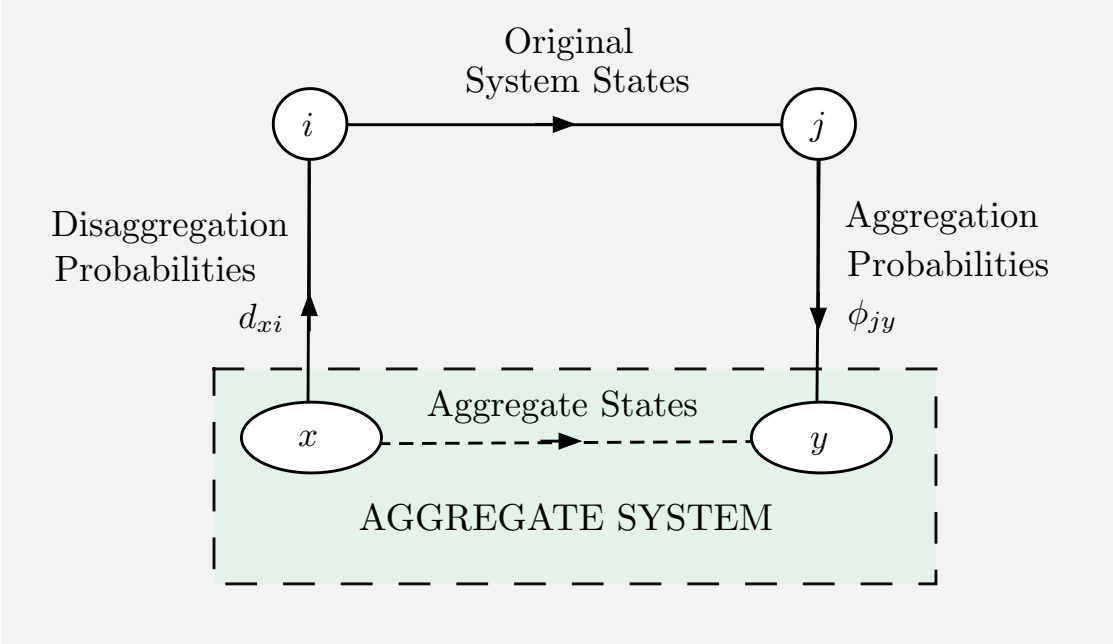
Probabilistic Approximation

Modify the probability distributions $P(w_k \mid x_k, w_k)$ to simplify the calculation of $\tilde{J}_{k+\ell}$ and/or the lookahead minimization

Certainty equivalent control ideas (inspired by LQG control)

- Replace uncertain quantities with deterministic nominal values
- The lookahead and tail problems are deterministic so they can be solved by DP or by special deterministic methods
- Use expected values or forecasts to determine nominal values; update policy when forecasts change (on-line replanning)
- A variant: Partial certainty equivalence. Fix only some uncertain quantities to nominal values
- A generalization: Approximate $E\{\cdot\}$ by limited simulation

Tail Problem Approximation by Aggregation



- Construct a “smaller" aggregate tail problem by introducing aggregate states
- Use the exact costs-to-go of the aggregate tail problem as approximate costs-to-go for the original

Aggregation examples:

- State discretization-intepolation schemes
- Grouping of states into subsets, which serve as aggregate states
- Feature-based discretization; aggregate problem operates in the space of features