Reinforcement Learning and Optimal Control

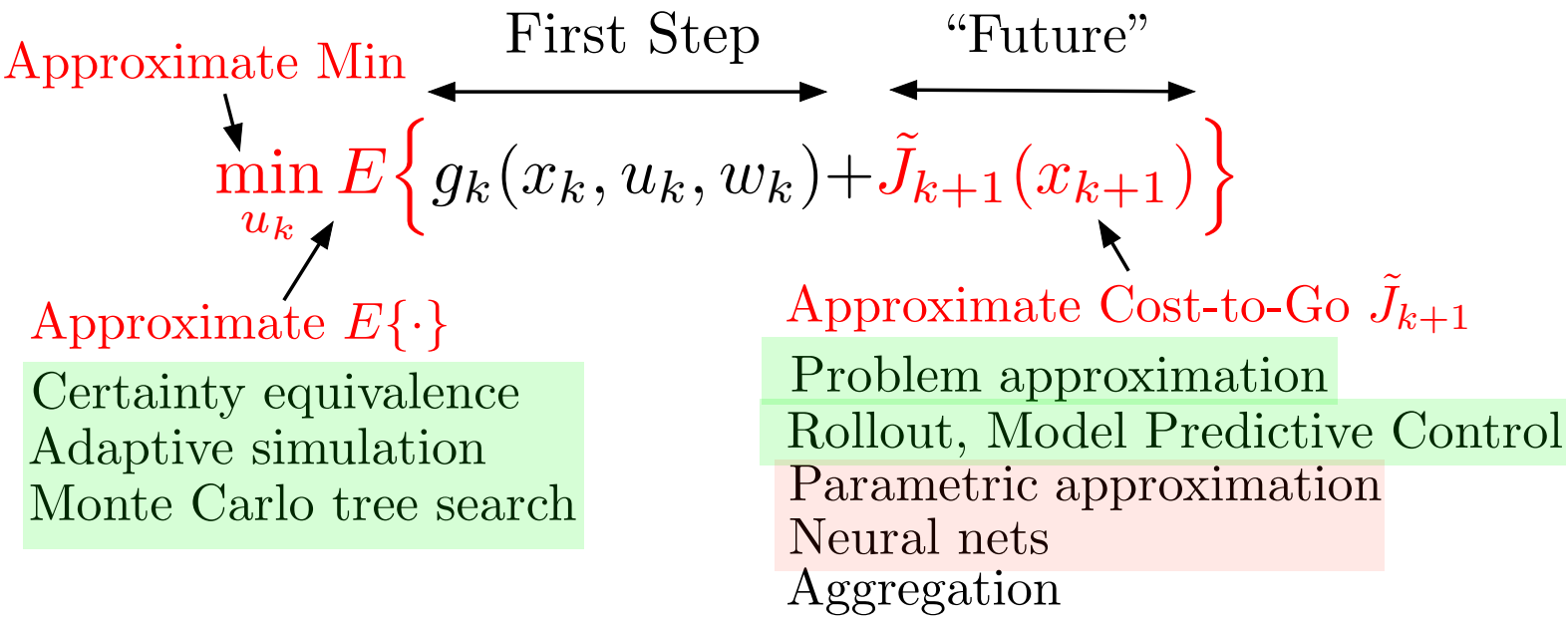
ASU, CSE 691, Winter 2020

Dimitri P. Bertsekas
dimitrib@mit.edu

Lecture 6

Outline

Recall the Approximation in Value Space Framework for Finite Horizon Problems

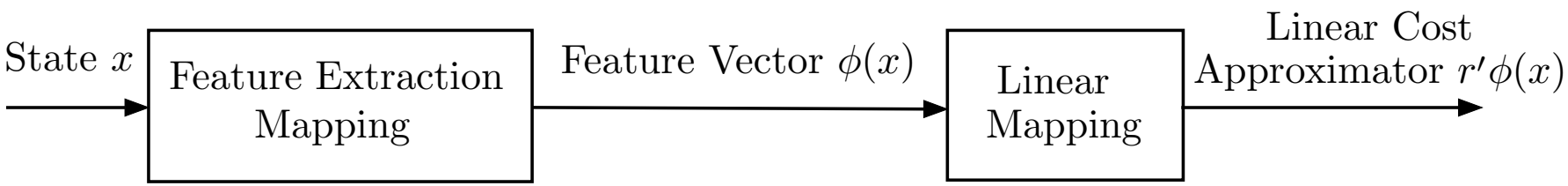


The starting point: A target cost function and an approximation architecture

- The architecture: A class of functions $\tilde{J}(x, r)$ that depend on x and a vector $r = (r_1, \ldots, r_m)$ of m "tunable" scalar parameters (or weights).
- Training: Use data to adjust r so that $\tilde{J}$ "matches" the target function, usually by some form of least squares fit.
- Architectures are feature-based if they depend on x via a feature vector $\phi(x)$,

$$\tilde{J}(x, r) = \hat{J}\big(\phi(x), r\big),$$

  where $\hat{J}$ is some function. Idea: Features capture dominant nonlinearities and can be problem-specific.
- Architectures $\tilde{J}(x, r)$ can be linear or nonlinear in r. Linear are much easier to train.
- A linear feature-based architecture: $\tilde{J}(x, r) = r'\phi(x) = \sum_{i=1}^{m} r_i \phi_i(x)$



Examples of Generic Feature-Based Architectures

- Piecewise constant and piecewise linear architectures: The features are constant or linear functions defined on “pieces" of the state space.
- Quadratic polynomial approximation: $\tilde{J}(x, r)$ is quadratic in the components $x^1, \ldots, x^n$ of x. Consider features

$$\phi_0(x) = 1, \qquad \phi_i(x) = x^i, \qquad \phi_{ij}(x) = x^i x^j, \quad i, j = 1, \ldots, n.$$

  A linear feature-based architecture, where r consists of weights r_0, r_i, and r_{ij}:

$$\tilde{J}(x, r) = r_0 + \sum_{i=1}^{n} r_i x^i + \sum_{i=1}^{n} \sum_{j=i}^{n} r_{ij} x^i x^j$$

- General polynomial architectures: Higher-degree polynomials in the components $x^1, \ldots, x^n$. Another possibility: Polynomials of features.
- Many other possibilities: Radial basis functions, data-dependent/kernel architectures, support vector machines, etc.
- Partial state observation problems (POMDP): Can be reformulated as problems of perfect state observation involving a belief state. Architectures involving features of the belief state (such as state estimates) are useful.

Examples of Domain-Specific Feature-Based Architectures

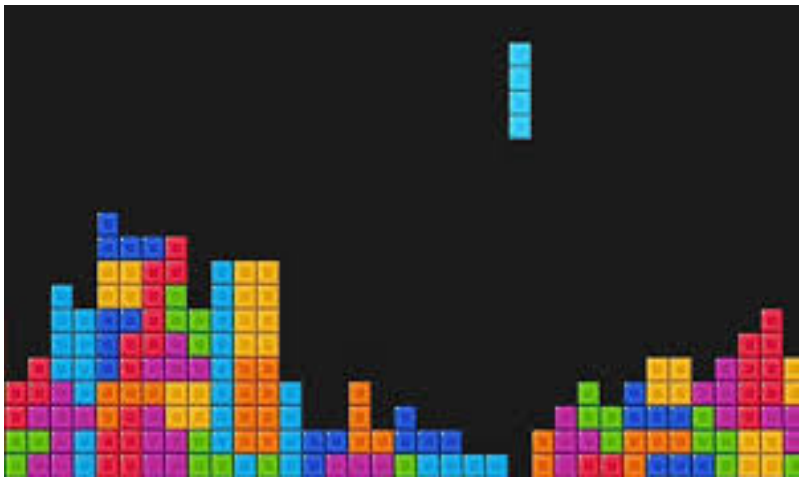
Features:
Material Balance,
Mobility,
Safety, etc

Feature Extraction

Weighting of Features

Win
"Probability"

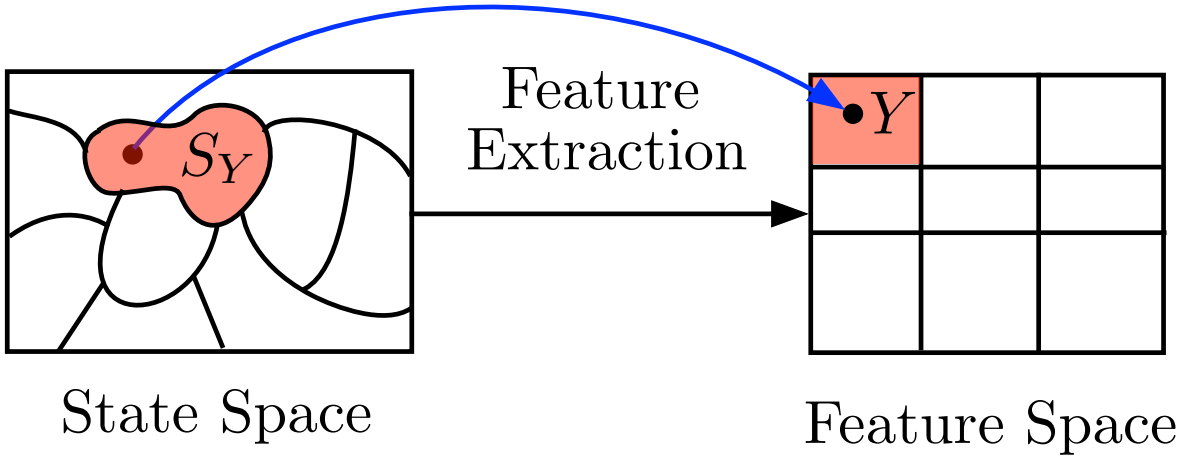
Chess



Tetris

Features such as column heights, column height differentials, number of “holes" etc

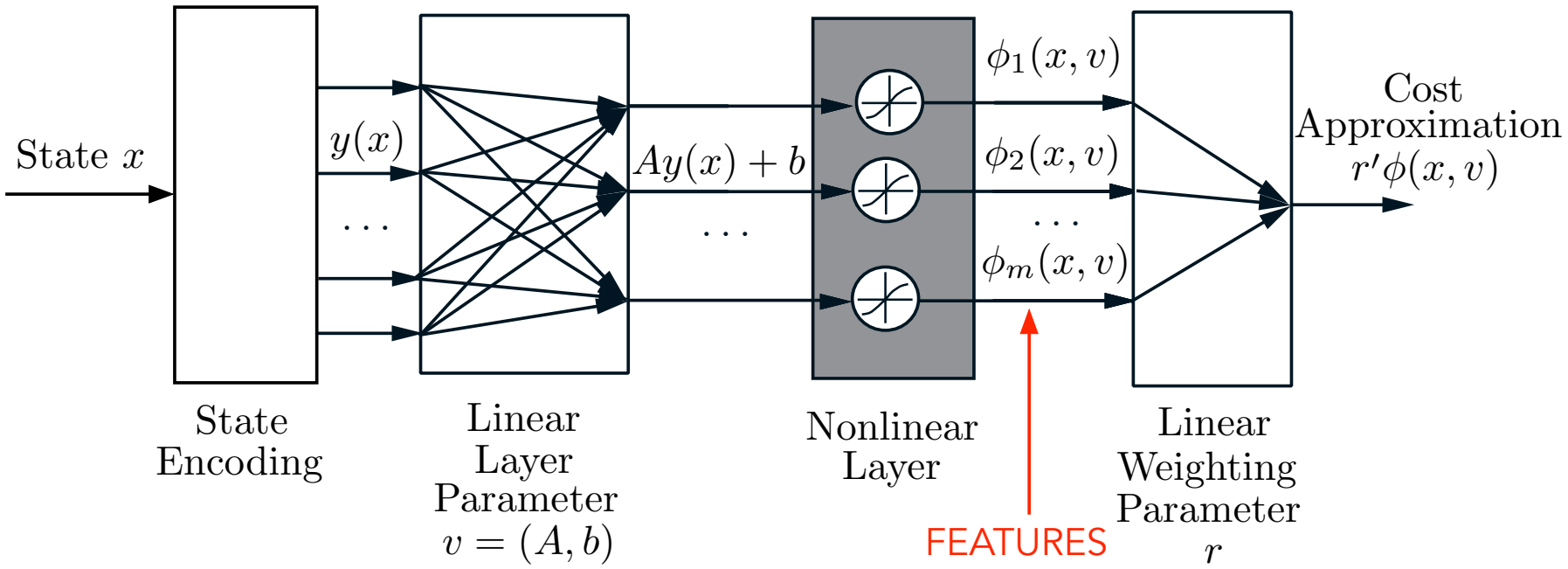
Feature-Based State Space Partitioning



A simple method to construct complex approximation architectures

- Use features to partition the state space into several subsets and construct a separate value and/or policy approximation in each subset.
- Example: Use a separate approximation architecture on each set of the partition.

Neural Nets: An Architecture that Automatically Constructs Features



Given a set of state-cost training pairs (x^s, β^s), $s = 1, \ldots, q$, the parameters of the neural network (A, b, r) are obtained by solving the training problem

$$\min_{A,b,r} \sum_{s=1}^{q} \left(\sum_{\ell=1}^{m} r_\ell \sigma\left(\left(Ay(x^s) + b \right)_\ell \right) - \beta^s \right)^2$$

- Special methods (also known as backpropagation or stochastic gradient descent) are typically used for training.
- Universal approximation property.

Training of Architectures

Least squares regression

- Collect a set of state-cost training pairs (x^s, β^s), $s = 1, \ldots, q$, where β^s is equal to the target cost $J(x^s)$ plus some "noise".
- r is determined by least squares fit, i.e., solving the problem

$$\min_r \sum_{s=1}^{q} \left(\tilde{J}(x^s, r) - \beta^s\right)^2$$

- Sometimes a quadratic regularization term $\gamma \|r\|^2$ is added to the least squares objective, to facilitate the minimization (among other reasons).

Training of linear feature-based architectures can be done exactly

- If $\tilde{J}(x, r) = r'\phi(x)$, where $\phi(x)$ is the m-dimensional feature vector, the training problem is quadratic and can be solved in closed form.
- The exact solution of the training problem is given by

$$\hat{r} = \left(\sum_{s=1}^{q} \phi(x^s)\phi(x^s)'\right)^{-1} \sum_{s=1}^{q} \phi(x^s)\beta^s$$

- This requires a lot of computation for a large m and data set; may not be best.

The main training issue

How to exploit the structure of the training problem

$$\min_r \sum_{s=1}^{q} \left(\tilde{J}(x^s, r) - \beta^s\right)^2$$

to solve it efficiently.

Key characteristics of the training problem

- Possibly nonconvex with many local minima, with horribly complicated graph (true when a neural net is used).
- Many terms in the least least squares sum; standard gradient and Newton-like methods are essentially inapplicable.
- Incremental iterative methods that operate on a single term $\left(\tilde{J}(x^s, r) - \beta^s\right)^2$ at each iteration have worked well enough (for many problems).

Incremental Gradient Methods

Generic sum of terms optimization problem

Minimize

$$f(y) = \sum_{i=1}^{m} f_i(y)$$

where each f_i is a differentiable scalar function of the n-dimensional vector y (this is the parameter vector in the context of parametric training).

The ordinary gradient method generates y^{k+1} from y^k according to

$$y^{k+1} = y^k - \gamma^k \nabla f(y^k) = y^k - \gamma^k \sum_{i=1}^{m} \nabla f_i(y^k)$$

where $\gamma^k > 0$ is a stepsize parameter.

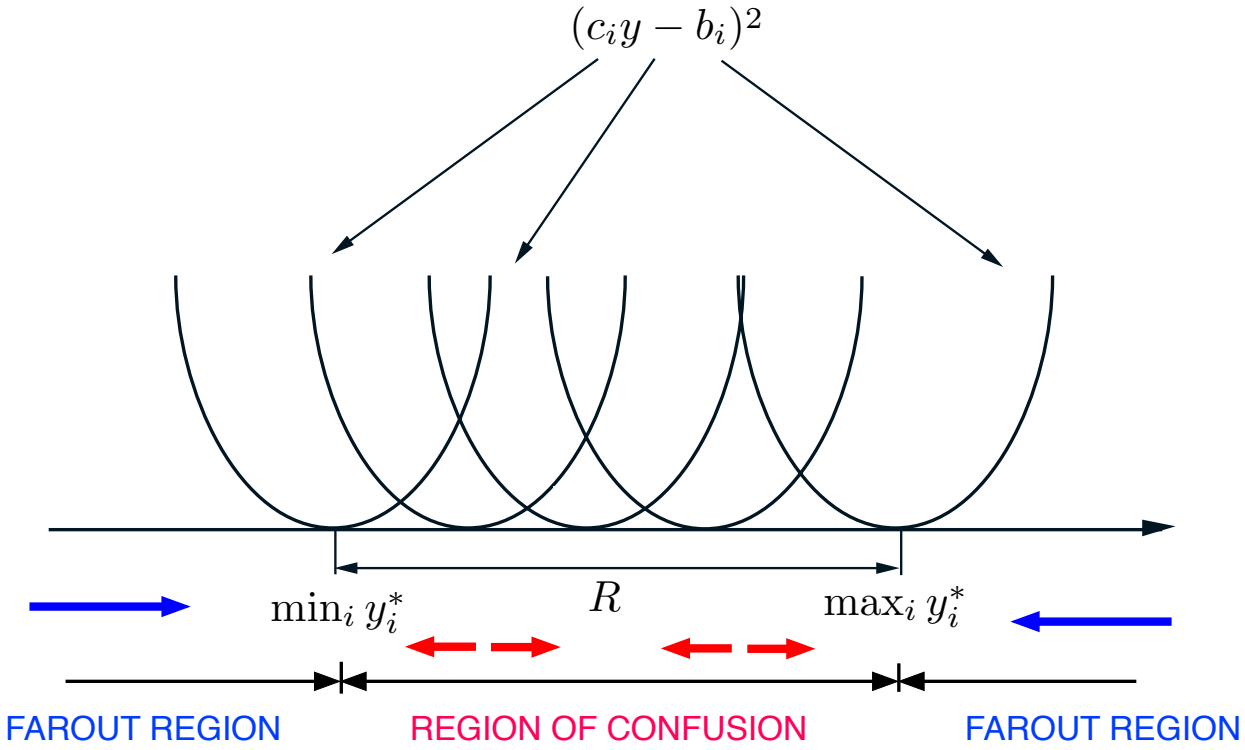
The incremental gradient counterpart

Choose an index i_k and iterate according to

$$y^{k+1} = y^k - \gamma^k \nabla f_{i_k}(y^k)$$

where $\gamma^k > 0$ is a stepsize parameter. Index selection can be orderly or randomly.

The Advantage of Incrementalism



Minimize $f(y) = \frac{1}{2}\sum_{i=1}^{m}(c_i y - b_i)^2$

Compare the ordinary and the incremental gradient methods in two cases

- When far from convergence: Incremental gradient is as fast as ordinary gradient with $1/m$ amount of work.
- When close to convergence: Incremental gradient gets confused and requires a diminishing stepsize for convergence.

Sequential DP Approximation - A Parametric Approximation at Every Stage (Also Called Fitted Value Iteration)

Start with $\tilde{J}_N = g_N$ and sequentially train going backwards, until $k = 0$

- Given a cost-to-go approximation $\tilde{J}_{k+1} \approx J^*_{k+1}$, 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), $s = 1, \ldots, q$, so that

$$\tilde{J}_k \approx J^*_k$$

Typical approach: Train by least squares/regression using a linear or nonlinear/neural net architecture

We minimize over r_k

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

An Alternative: Fitted Value Iteration Based on Q-Factors

- Consider sequential DP approximation of Q-factor parametric approximations

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

(Note a mathematical magic: The order of $E\{\cdot\}$ and min have been reversed.)

- 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 $E\{\cdot\}$.]
- Also: No need for a model to obtain β_k^s. Sufficient to have a simulator that generates state-control-cost-next state random samples

$$\big((x_k, u_k), (g_k(x_k, u_k, w_k), x_{k+1})\big)$$

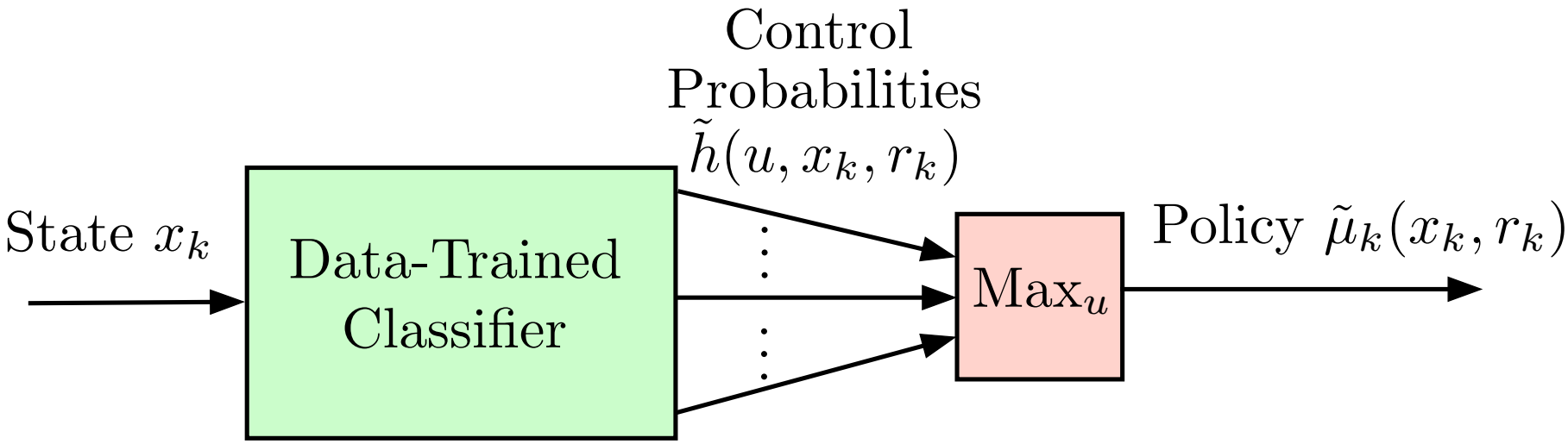
- Having computed r_k, the one-step lookahead control is obtained on-line as

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

without the need of a model or expected value calculations.

- Important advantage: The on-line calculation of the control is simplified.

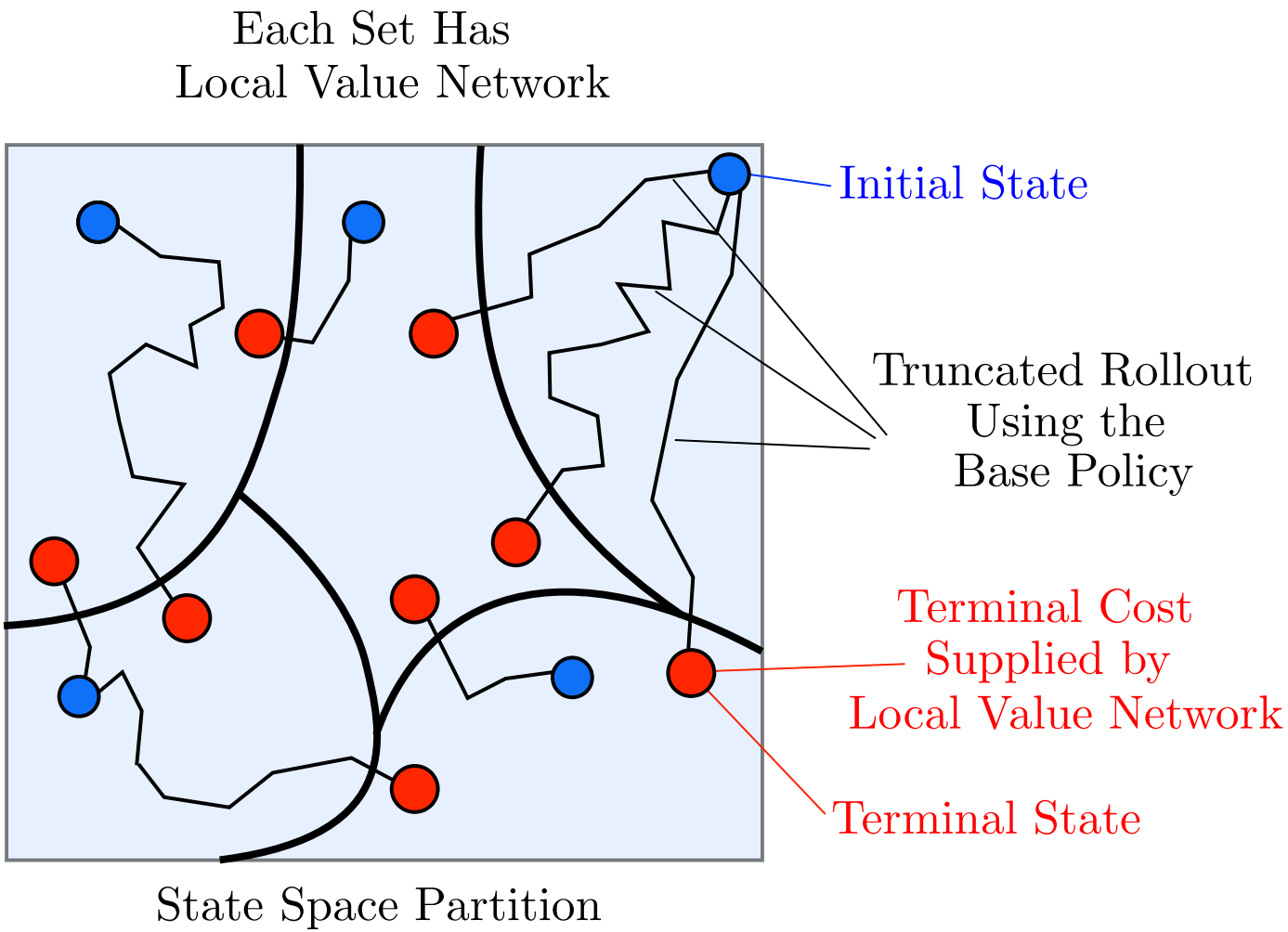
Parametric Approximation of a Given Policy - Finite Control Space



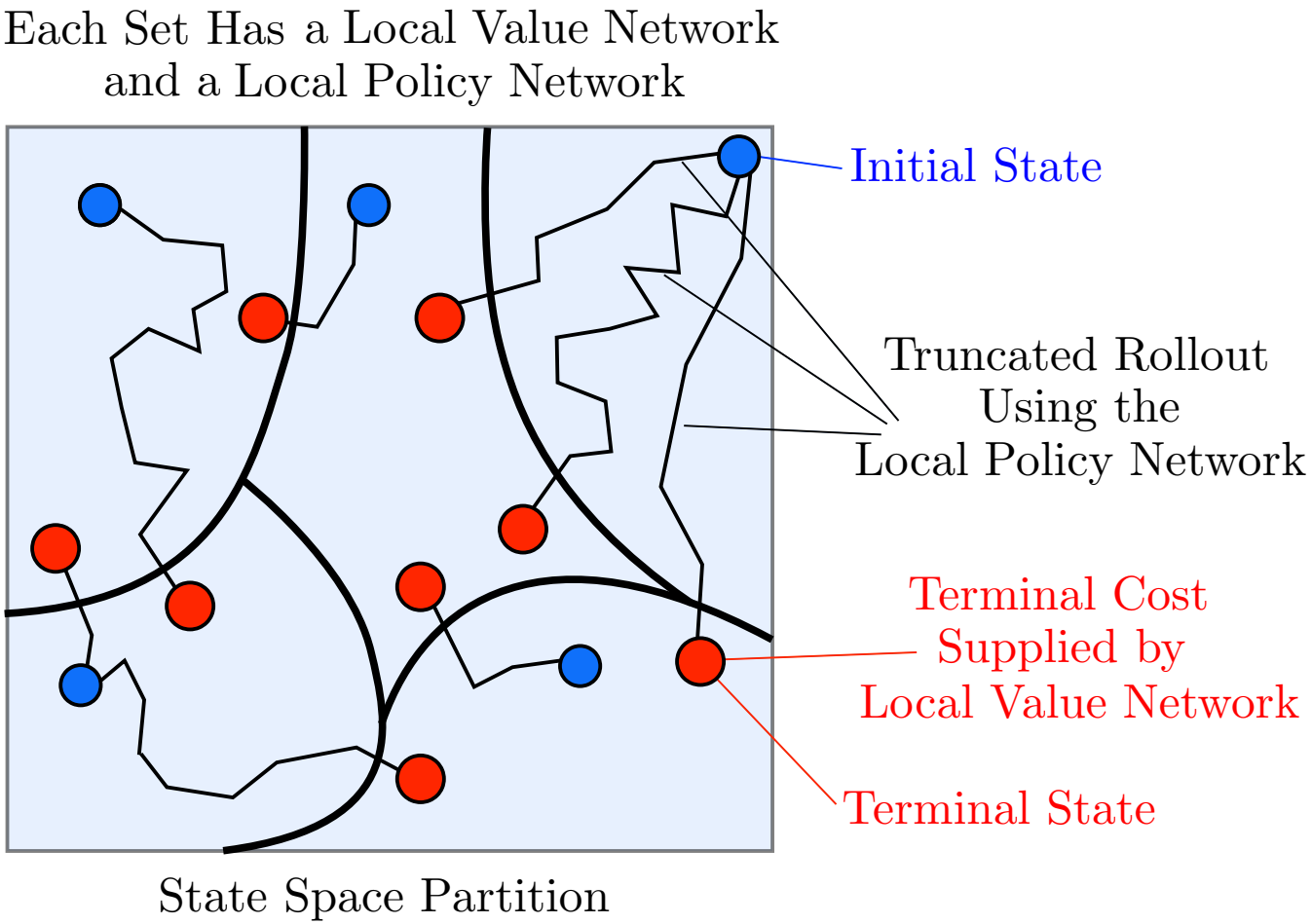
We can implement approximately a given policy with a data-trained classifier

- We collect a training set of many state-control pairs (x_k^s, u_k^s), $s = 1, \ldots, q$, using the policy (i.e., at x_k^s the policy applies u_k^s).
- The classifier generates for each state x_k the "probability" $\tilde{h}(u, x_k, r_k)$ of each control u being the correct one (i.e., the one generated by the given policy).
- The classifier outputs the control of max probability for each state.
- Thus a pattern classification/recognition method can be used to train the policy approximation.
- Neural nets are widely used for this.

Truncated Rollout with a Partitioned Architecture and a Value Network



Perpetual Rollout with a Partitioned Architecture
Multiple Value and Policy Networks



- Start with some base policy and a value network for each set.
- Obtain a policy and a value network for the truncated rollout policy. Repeat.
- Partitioning may be a good way to deal with adequate state space exploration.

About the Next Lecture

We will cover:

- Neural Network Discussion and Implementation Issues

PLEASE READ AS MUCH OF CHAPTER 3 AS YOU CAN

PLEASE DOWNLOAD THE LATEST VERSIONS FROM MY WEBSITE