Ten Key Ideas for Reinforcement Learning and Optimal Control

Dimitri P. Bertsekas

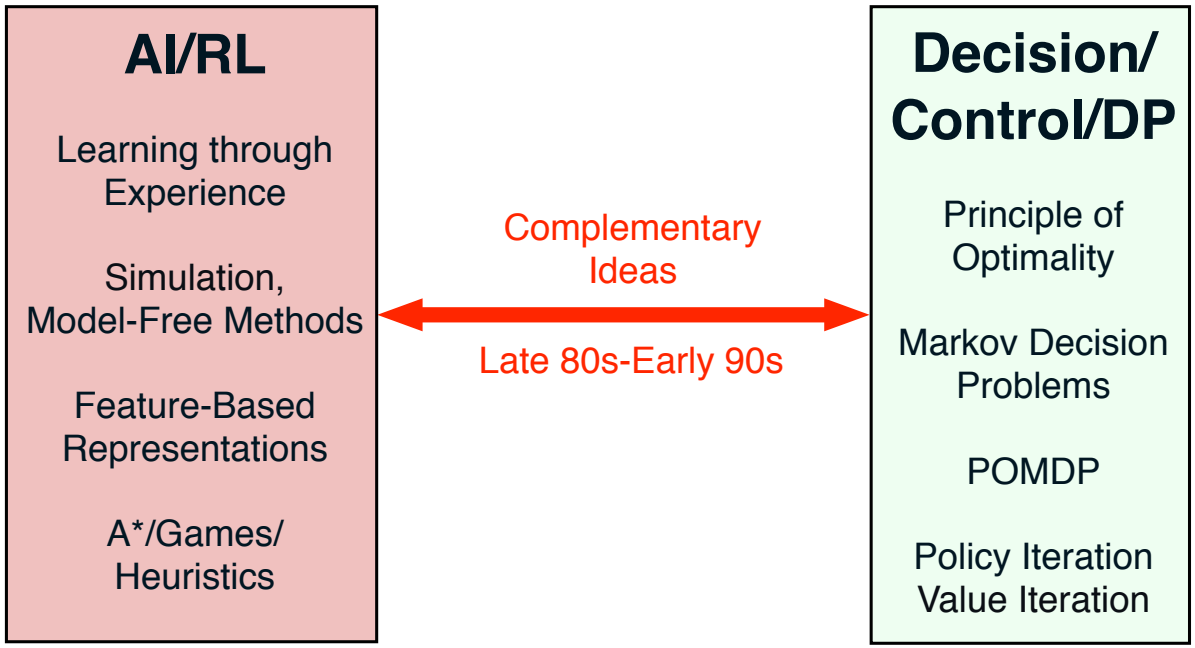
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology

and

School of Computing, Informatics, and Decision Systems Engineering
Arizona State University

August 2019
(Periodically Updated)

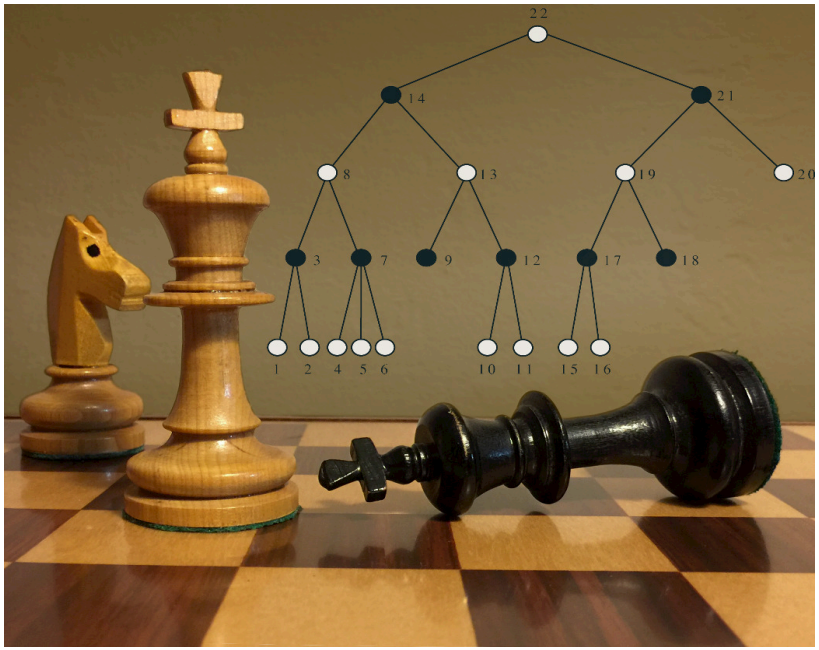
Reinforcement Learning (RL): A Happy Union of AI and Decision/Control/Dynamic Programming (DP) Ideas



Historical highlights

- Exact DP, optimal control (Bellman, Shannon, 1950s ...)
- First impressive successes: Backgammon programs (Tesauro, 1992, 1996)
- Algorithmic progress, analysis, applications, first books (mid 90s ...)
- Machine Learning, BIG Data, Robotics, Deep Neural Networks (mid 2000s ...)

AlphaGo (2016) and AlphaZero (2017)



AlphaZero

Plays much better than all chess programs

Plays different!

Learned from scratch ... with 4 hours of training!

Same algorithm learned multiple games (Go, Shogi)

Methodology:

- Simulation-based approximation to a form of the policy iteration method of DP
- Uses self-learning, i.e., self-generated data for policy evaluation, and Monte Carlo tree search for policy improvement

The success of AlphaZero is due to:

- A skillful implementation/integration of known ideas
- Awesome computational power

Approximate DP/RL Methodology is now Ambitious and Universal

Exact DP applies (in principle) to a very broad range of optimization problems

- From Deterministic to Stochastic
- From Combinatorial optimization to Optimal control w/ infinite state/control spaces
- From One decision maker to Two player games
- ... BUT is plagued by the curse of dimensionality and need for a math model

Approximate DP/RL overcomes the difficulties of exact DP by:

- Approximation (use neural nets and other architectures to reduce dimension)
- Simulation (use a computer model in place of a math model)

State of the art:

- Broadly applicable methodology: Can address broad range of challenging problems. Deterministic-stochastic-dynamic, discrete-continuous, games, etc
- There are no methods that are guaranteed to work for all or even most problems
- There are enough methods to try with a reasonable chance of success for most types of optimization problems
- Role of the theory: Guide the art, delineate sound ideas, filter out bad ideas

About this Talk

The purpose of this talk

- To selectively explain some of the key ideas of RL and its connections with DP.
- To provide a road map for further study (mostly from the perspective of DP).
- To provide a guide for reading my book (abbreviated RL-OC):
  - Bertsekas, "Reinforcement Learning and Optimal Control" Athena Scientific, 2019; see also the monograph "Rollout, Policy Iteration and Distributed RL" 2020, which deals with rollout, multiagent problems, and distributed asynchronous algorithms.
  - For slides and videolectures from 2019 and 2020 ASU courses, see my website.

References

- Quite a few Exact DP books (1950s-present starting with Bellman). My books:
  - My two-volume textbook "Dynamic Programming and Optimal Control" was updated in 2017.
  - My mathematically oriented research monograph “Stochastic Optimal Control" (with S. E. Shreve) came out in 1978.
  - My latest mathematically oriented research monograph “Abstract DP" came out in 2018.
- Quite a few Approximate DP/RL/Neural Nets books (1996-Present)
  - Bertsekas and Tsitsiklis, Neuro-Dynamic Programming, 1996
  - Sutton and Barto, 1998, Reinforcement Learning (new edition 2018)
- Many surveys on all aspects of the subject

Terminology in RL/AI and DP/Control (RL-OC, Section 1.4)

RL uses Max/Value, DP uses Min/Cost

- Reward of a stage = (Opposite of) Cost of a stage.
- State value = (Opposite of) State cost.
- Value (or state-value) function = (Opposite of) Cost function.

Controlled system terminology

- Agent = Decision maker or controller.
- Action = Control.
- Environment = Dynamic system.

Methods terminology

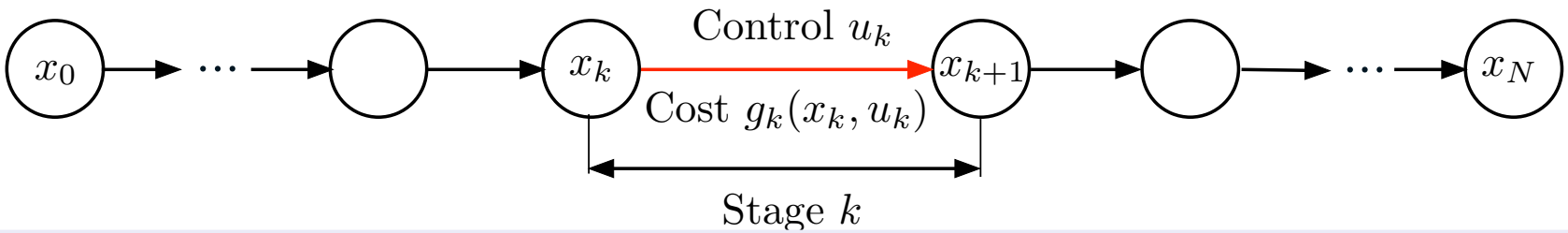
- Learning = Solving a DP-related problem using simulation.
- Self-learning (or self-play in the context of games) = Solving a DP problem using simulation-based policy iteration.
- Planning vs Learning distinction = Solving a DP problem with model-based vs model-free simulation.

AN OUTLINE OF THE SUBJECT - TEN KEY IDEAS

1. Principle of Optimality
2. Approximation in Value Space
3. Approximation in Policy Space
4. Model-Free Methods and Simulation
5. Policy Improvement, Rollout, and Self-Learning
6. Approximate Policy Improvement, Adaptive Simulation, and Q-Learning
7. Features, Approximation Architectures, and Deep Neural Nets
8. Incremental and Stochastic Gradient Optimization
9. Direct Policy Optimization: A More General Approach
10. Gradient and Random Search Methods for Direct Policy Optimization

1. PRINCIPLE OF OPTIMALITY
(RL-OC, Chapter 1)

Finite Horizon Deterministic Problem



- Discrete-time system:

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

where x_k: State, u_k: Control chosen from some constraint set $U_k(x_k)$

- Cost function:

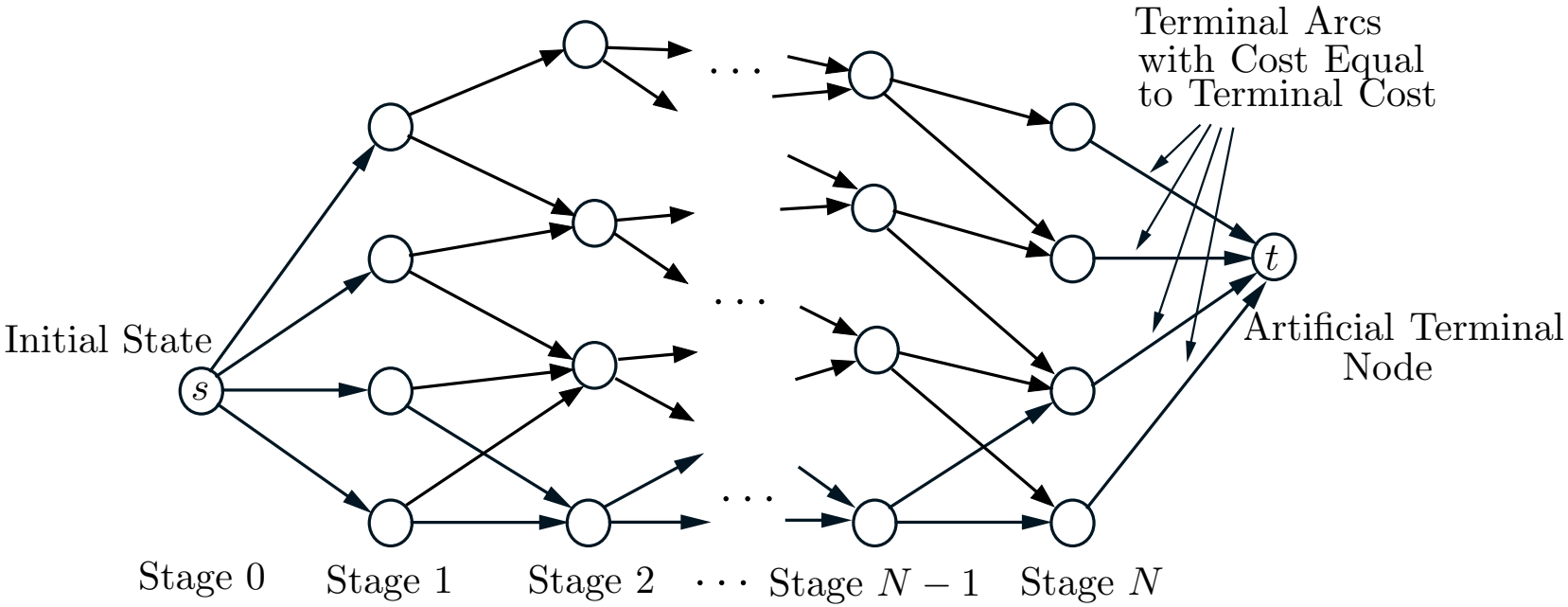
$$g_N(x_N) + \sum_{k=0}^{N-1} g_k(x_k, u_k)$$

- For given initial state x_0, minimize over control sequences $\{u_0, \ldots, u_{N-1}\}$

$$J(x_0; u_0, \ldots, u_{N-1}) = g_N(x_N) + \sum_{k=0}^{N-1} g_k(x_k, u_k)$$

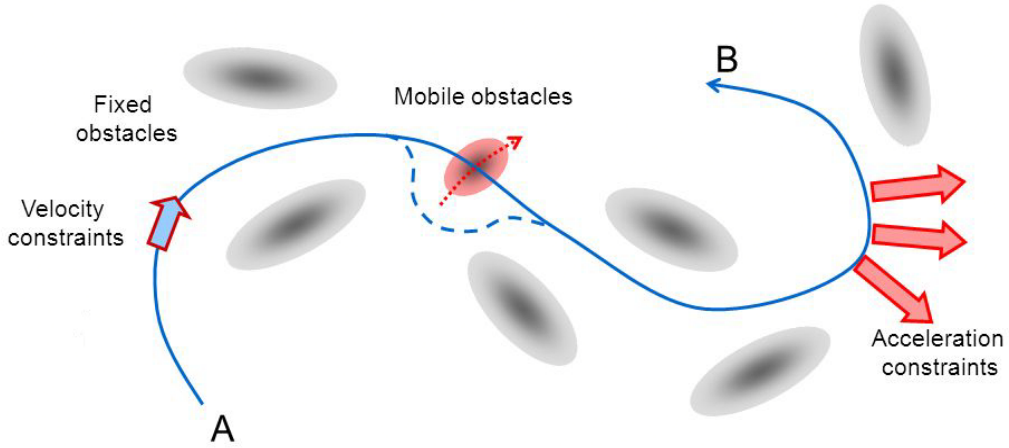
- Control sequences correspond to paths from start node to end node in the graph
- Optimal cost function $J^*(x_0) = \min_{\substack{u_k \in U_k(x_k) \\ k=0,\ldots,N-1}} J(x_0; u_0, \ldots, u_{N-1})$

Discrete-State Problems: Combinatorial/Shortest Path View



- Nodes correspond to states x_k
- Arcs correspond to state-control pairs (x_k, u_k)
- An arc (x_k, u_k) has start node x_k and end node $x_{k+1} = f_k(x_k, u_k)$
- A control sequence $\{u_0, \ldots, u_{N-1}\}$ corresponds to a sequence of arcs from the initial node s to the terminal node t.
- An arc (x_k, u_k) has a cost $g_k(x_k, u_k)$. The cost to optimize is the sum of the arc costs from s to t.
- The problem is equivalent to finding a minimum cost/shortest path from s to t.

Continuous-State Problems



PATH PLANNING

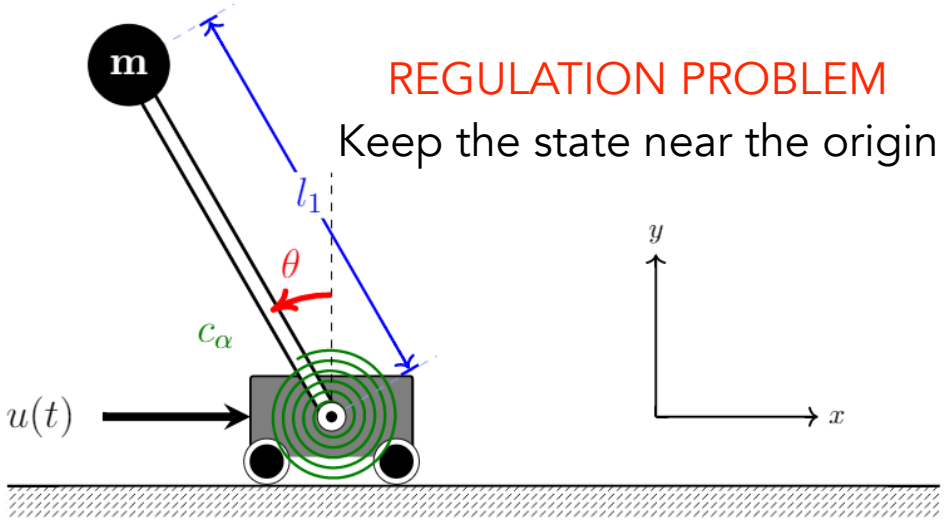
Keep state close to a trajectory

Motion equations

$$x_{k+1} = f_k(x_k, u_k)$$

Penalty for deviating from nominal trajectory

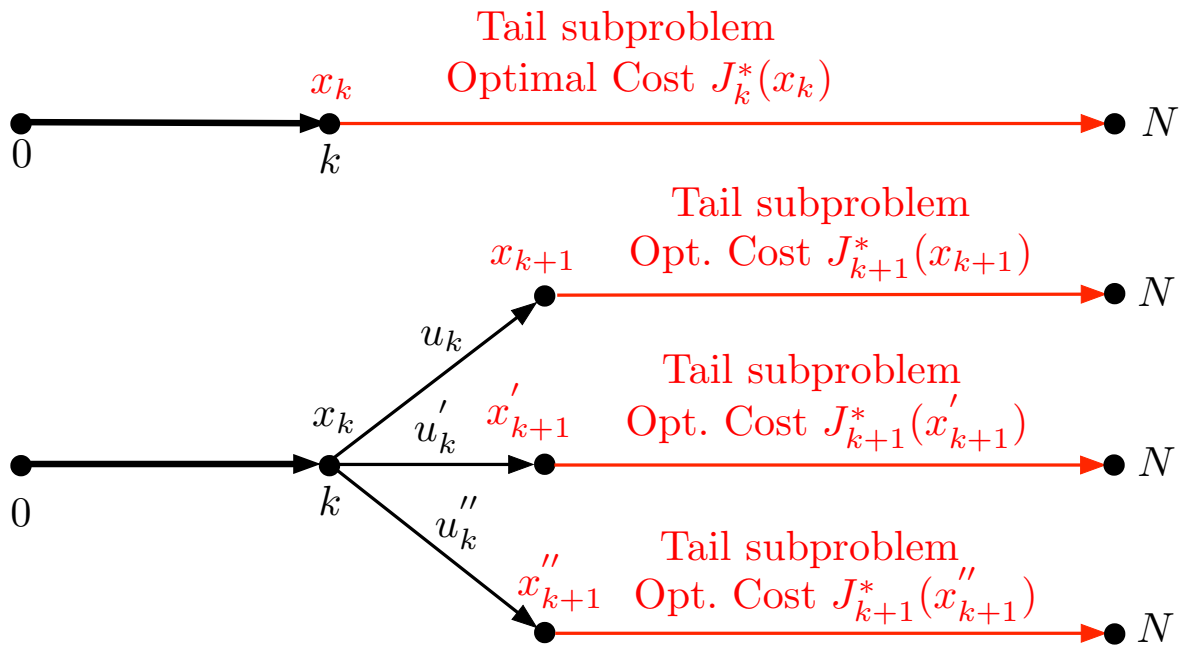
State and control constraints



REGULATION PROBLEM

Keep the state near the origin

Tail Subproblems and the Principle of Optimality



Consider solution of the tail subproblem that starts at x_k at time k and minimizes over $\{u_k, \ldots, u_{N-1}\}$ the "cost-to-go" from k to N,

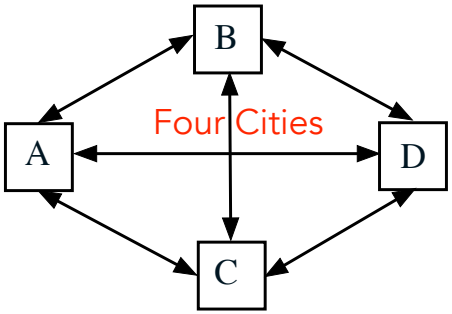
$$g_k(x_k, u_k) + \sum_{m=k+1}^{N-1} g_m(x_m, u_m) + g_N(x_N), \qquad \text{Optimal Cost-to-Go: } J_k^*(x_k)$$

To solve it, choose u_k to min (1st stage cost + Optimal tail problem cost) or

$$\min_{u_k \in U_k(x_k)} \Big[g_k(x_k, u_k) + J_{k+1}^*\big(f_k(x_k, u_k)\big) \Big]$$

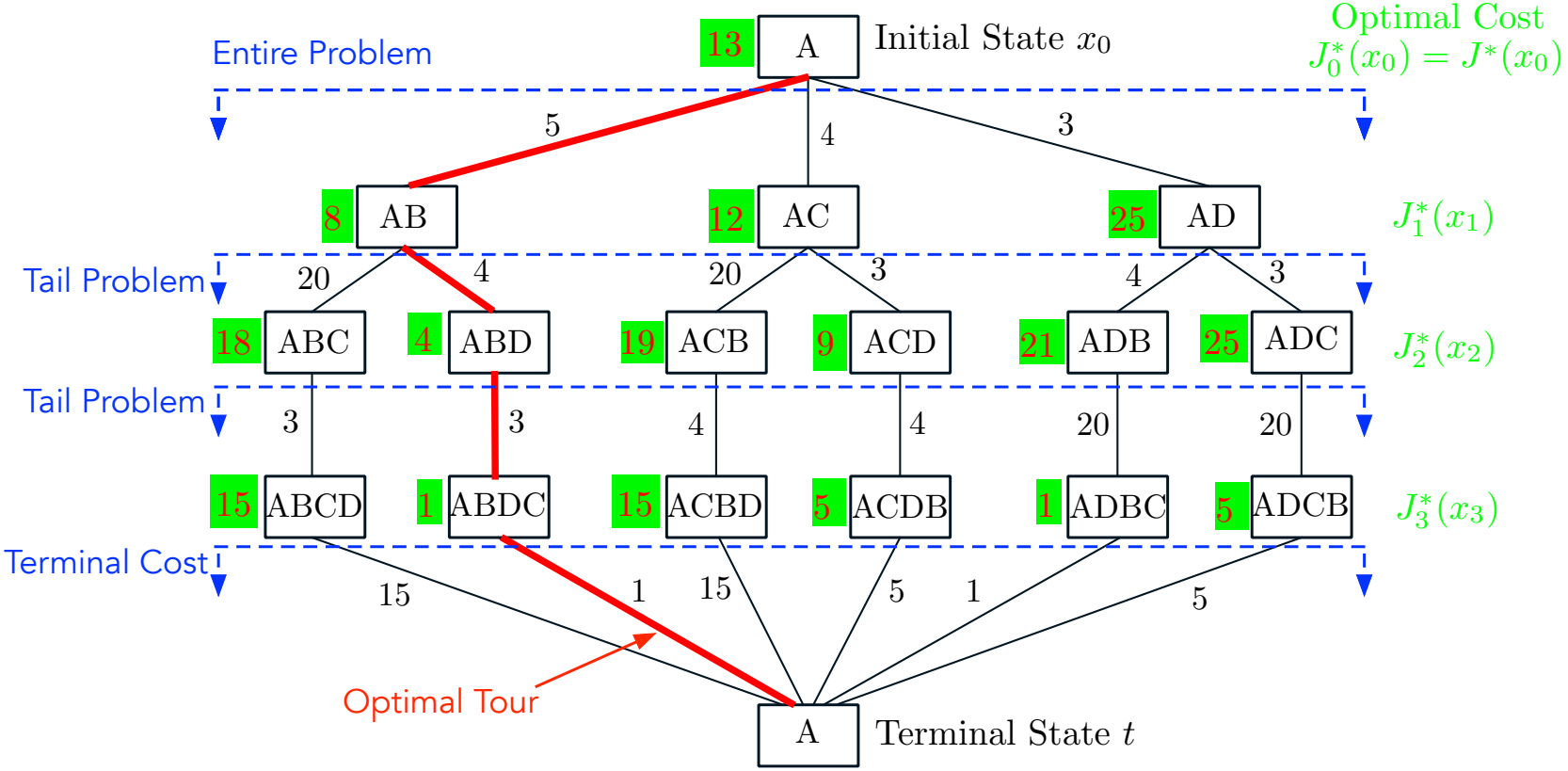
DP Algorithm Idea: Solve Progressively Longer Tail Subproblems

Traveling Salesman Example



Matrix of Intercity Travel Costs

	5	4	3
5		20	4
1	20		3
15	4	3	



DP Algorithm: Formal Statement

Go backward to compute the optimal costs $J_k^*(x_k)$ of the x_k-tail subproblems

Start with

$$J_N^*(x_N) = g_N(x_N), \qquad \text{for all } x_N,$$

and for $k = 0, \ldots, N-1$, let

$$J_k^*(x_k) = \min_{u_k \in U_k(x_k)} \Big[g_k(x_k, u_k) + J_{k+1}^*\big(f_k(x_k, u_k)\big) \Big], \qquad \text{for all } x_k.$$

The optimal cost $J^*(x_0)$ is obtained at the last step: $J_0^*(x_0) = J^*(x_0)$.

Go forward to construct optimal control sequence $\{u_0^*, \ldots, u_{N-1}^*\}$

Start with

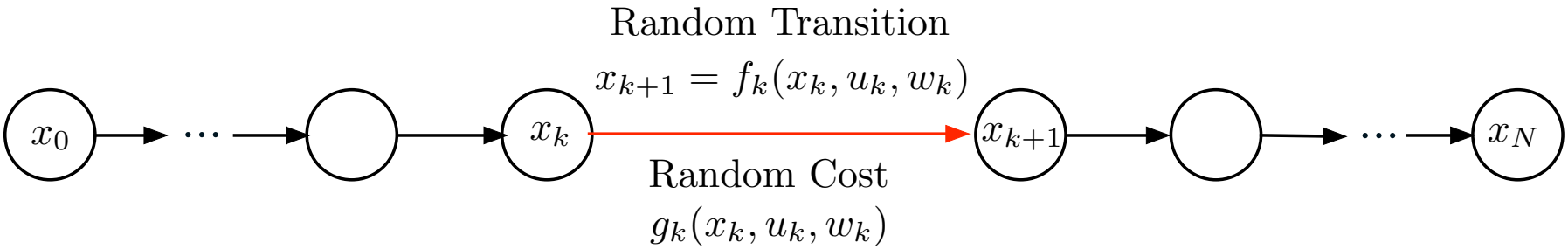
$$u_0^* \in \arg \min_{u_0 \in U_0(x_0)} \Big[g_0(x_0, u_0) + J_1^*\big(f_0(x_0, u_0)\big) \Big], \qquad x_1^* = f_0(x_0, u_0^*).$$

Sequentially, going forward, for $k = 1, 2, \ldots, N-1$, set

$$u_k^* \in \arg \min_{u_k \in U_k(x_k^*)} \Big[g_k(x_k^*, u_k) + J_{k+1}^*\big(f_k(x_k^*, u_k)\big) \Big], \qquad x_{k+1}^* = f_k(x_k^*, u_k^*).$$

Approximation idea: Replace J_k^* with an approximation $\tilde{J}_k$ (possibly a neural net).

Stochastic DP Problems



- Stochasticity in the form of a random "disturbance" w_k (e.g., physical noise, market uncertainties, demand for inventory, unpredictable breakdowns, etc)
- Cost function:

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

- Policies $\pi = \{\mu_0, \ldots, \mu_{N-1}\}$, where μ_k is a "closed-loop control law" or "feedback policy"/a function of x_k. Specifies control $u_k = \mu_k(x_k)$ to apply when at x_k.
- For given initial state x_0, minimize over all $\pi = \{\mu_0, \ldots, \mu_{N-1}\}$ the cost

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

- Optimal cost function $J^*(x_0) = \min_\pi J_\pi(x_0)$

The Stochastic DP Algorithm

Go backward to compute the optimal costs $J_k^*(x_k)$ of the x_k-tail subproblems

- Start with $J_N^*(x_N) = g_N(x_N)$, and for $k = 0, \ldots, N-1$, let

$$J_k^*(x_k) = \min_{u_k \in U_k(x_k)} E\Big\{ g_k(x_k, u_k, w_k) + J_{k+1}^*\big(f_k(x_k, u_k, w_k)\big) \Big\}, \qquad \text{for all } x_k.$$

- The optimal cost $J^*(x_0)$ is obtained at the last step: $J_0^*(x_0) = J^*(x_0)$.
- The optimal control function μ_k^* can be constructed simultaneously with J_k^*, and consists of the minimizing $u_k^* = \mu_k^*(x_k)$ above.

Go forward to implement the optimal policy, given $J_1^*, \ldots, J_{N-1}^*$

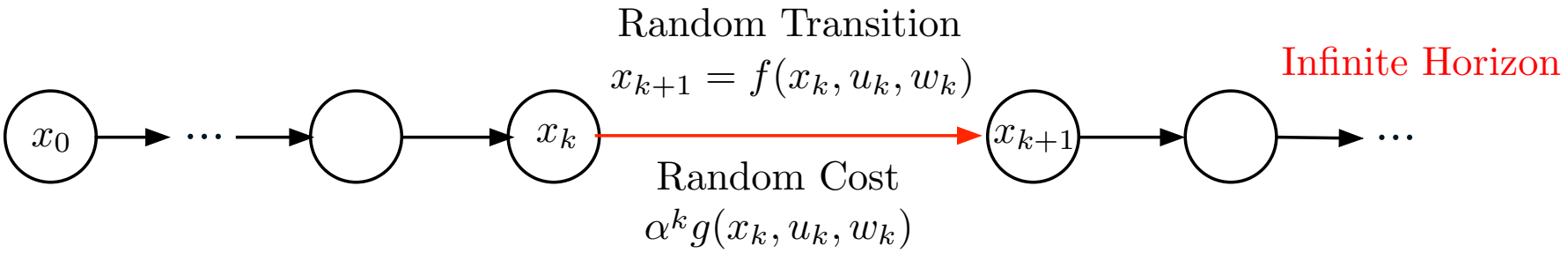
Sequentially, starting with x_0, observe x_k and apply

$$u_k^* \in \arg \min_{u_k \in U_k(x_k)} E\Big\{ g_k(x_k, u_k, w_k) + J_{k+1}^*\big(f_k(x_k, u_k, w_k)\big) \Big\}.$$

Issues: Need to compute J_{k+1}^* (possibly off-line), compute expectation for each u_k, minimize over all u_k

Approximation idea: Use $\tilde{J}_k$ in place of J_k^*, and approximate $E\{\cdot\}$ and $\min_{u_k}$.

Infinite Horizon Extension: α-Discounted Case ($0 < \alpha < 1$)



Infinite number of stages, and stationary system and cost

- System $x_{k+1} = f(x_k, u_k, w_k)$ with state, control, and random disturbance.
- Policies $\pi = \{\mu_0, \mu_1, \ldots\}$ with $\mu_k(x) \in U(x)$ for all x and k.
- Optimal cost function $J^*(x_0) = \min_\pi J_\pi(x_0)$ satisfies Bellman's equation

$$J^*(x) = \min_{u \in U(x)} E\Big\{ g(x,u,w) + \alpha J^*\big(f(x,u,w)\big) \Big\}$$

- Optimal policy: Applies at x the minimizing u above, regardless of stage k.
- When there are finitely many states, $i = 1, \ldots, n$, Bellman's equation is written in terms of the $i \to j$ transition probabilities $p_{ij}(u)$ as

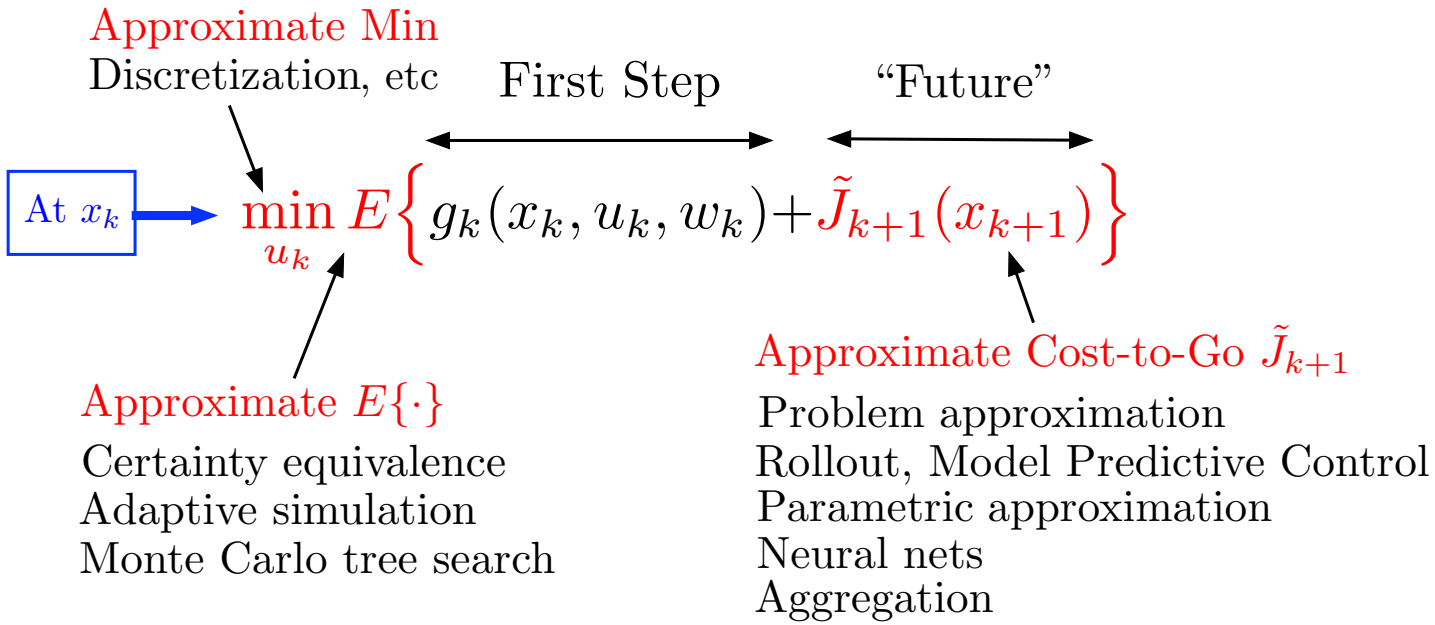
$$J^*(i) = \min_{u \in U(i)} \sum_{j=1}^{n} p_{ij}(u)\big(g(i,u,j) + \alpha J^*(j)\big)$$

- Approximation possibility: Use $\tilde{J}$ in place of J^*, and approximate $E\{\cdot\}$ and $\min_u$

2. APPROXIMATION IN VALUE SPACE (RL-OC, Sections 2.1-2.2)

Approximation in Value Space: One-Step Lookahead

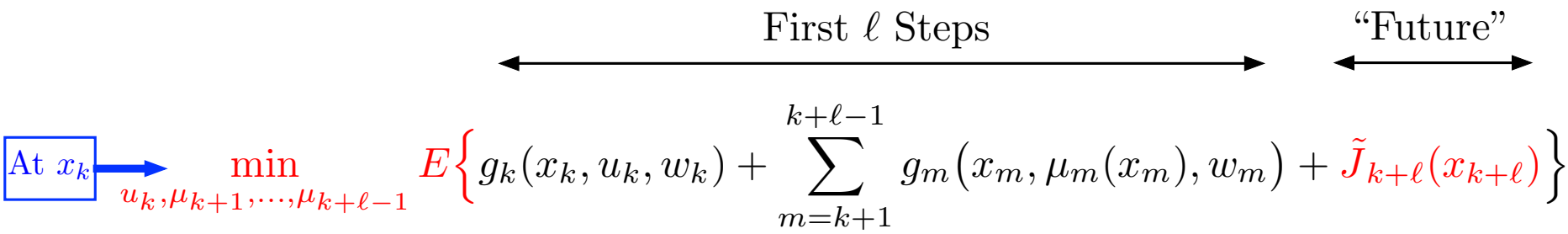
At state x_k, use $\tilde{J}_{k+1}$ (in place of J^*_{k+1}) to compute a (suboptimal) control



Three main issues; they can be addressed separately

- How to construct $\tilde{J}_k$ [an important example is parametric approximation $\tilde{J}_k(x_k, r_k)$ with parameter vector r_k, e.g., neural nets].
- How to simplify $E\{\cdot\}$ operation.
- How to simplify min operation.

Approximation in Value Space: Multistep Lookahead

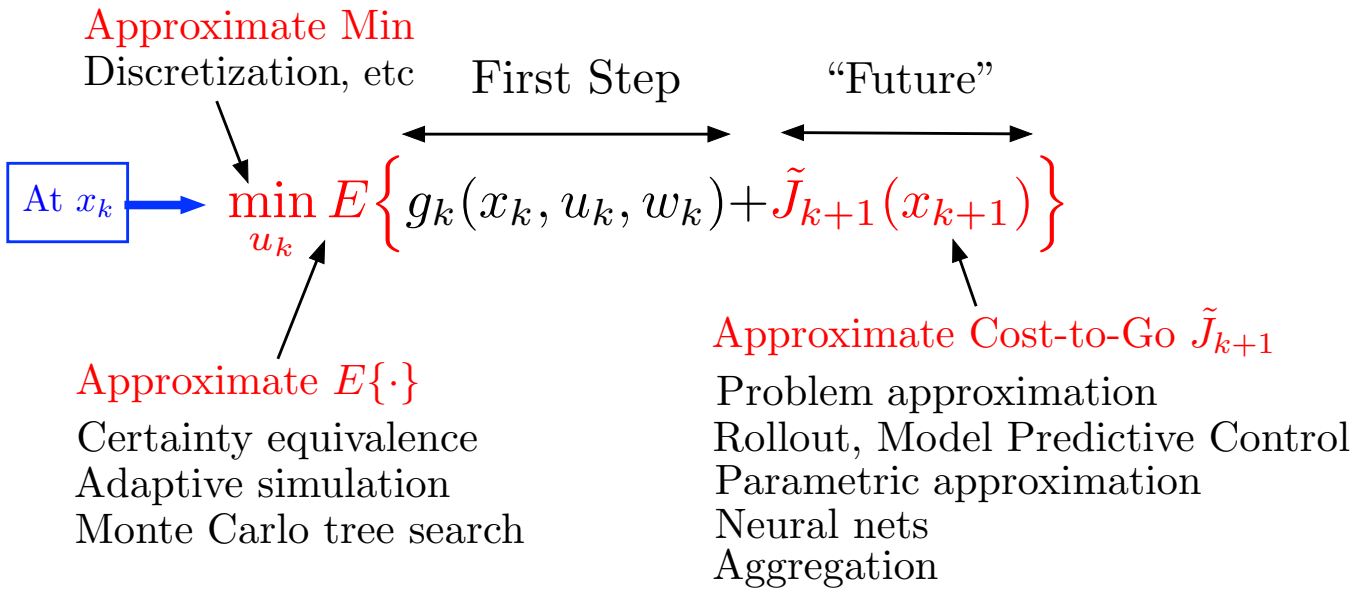


- At state x_k, solve an ℓ-stage version of the DP problem with x_k as the initial state and $\tilde{J}_{k+\ell}$ as the terminal cost function.
- Use the first control of the ℓ-stage policy thus obtained, and discard the others.

We can view ℓ-step lookahead as a special case of one-step lookahead:

The "effective" one-step lookahead function is the optimal cost function of an $(\ell-1)$-stage DP problem with terminal cost $\tilde{J}_{k+\ell}$.

On-Line and Off-Line Lookahead Implementations



- Off-line methods: All the functions $\tilde{J}_{k+1}$ are computed for every k, before the control process begins.
- Examples of off-line methods: Neural net and other parametric approximations.
- On-line methods: The values $\tilde{J}_{k+1}(x_{k+1})$ are computed only at the relevant next states x_{k+1}, and are used to compute the control to be applied at the N time steps.
- Examples of on-line methods: Rollout and model predictive control.
- On-line methods are well-suited for on-line replanning (but require more on-line computation).

Simplifying the Minimization of the Expected Value in Lookahead Schemes

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

The minimization can be done by:

- Brute force (after discretization if there are infinitely many controls).
- For deterministic problems and continuous controls, nonlinear programming.
- For stochastic problems and continuous controls, stochastic programming.
- When u has multiple components, multiagent/component-by-component minimization (see the book "Rollout, Policy Iteration and Distributed RL", 2020).

Possibilities to simplify the $E\{\cdot\}$:

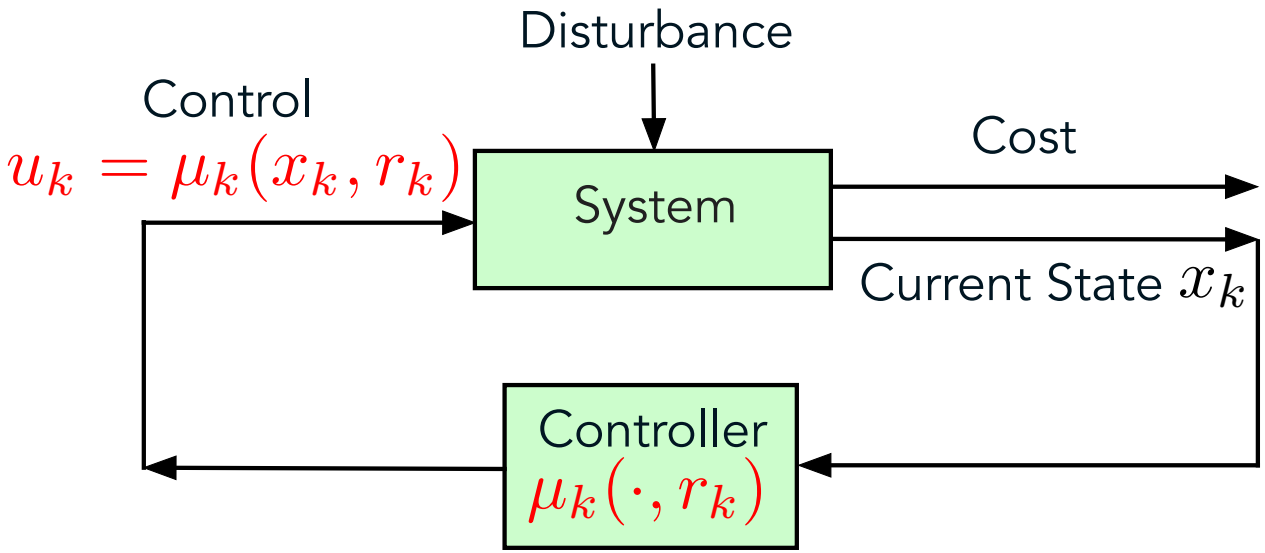
- Assumed certainty equivalence, i.e., choose a typical value $\tilde{w}_k$ of w_k, and use the control $\tilde{\mu}_k(x_k)$ that solves the deterministic problem

$$\min_{u_k \in U_k(x_k)} \Big[g_k(x_k, u_k, \tilde{w}_k) + \tilde{J}_{k+1}\big(f_k(x_k, u_k, \tilde{w}_k)\big) \Big]$$

  However, this may degrade performance significantly.
- Monte Carlo Tree Search (a form of lookahead based on adaptive simulation - simulate more accurately the controls that seem “most promising").

3. APPROXIMATION IN POLICY SPACE (RL-OC, Section 2.1)

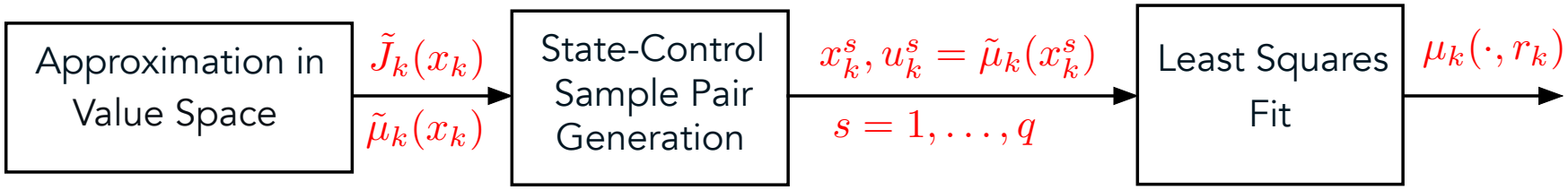
Approximation in Policy Space: The Major Alternative to Approximation in Value Space



- Idea: Select the policy by optimization over a suitably restricted class of policies.
- The restricted class is usually a parametric family of policies $\mu_k(x_k, r_k)$, $k = 0, \ldots, N-1$, of some form, where r_k is a parameter (e.g., a neural net).
- Important advantage once the parameters r_k are computed: The computation of controls during on-line operation of the system is often much easier; i.e.,

$$\text{At state } x_k \text{ apply } u_k = \mu_k(x_k, r_k)$$

Approximation in Policy Space on Top of Approximation in Value Space



- Compute approximate cost-to-go functions $\tilde{J}_{k+1}$, $k = 0, \ldots, N-1$.
- This defines a suboptimal policy $\tilde{\mu}_k$, $k = 0, \ldots, N-1$, through

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

- Generate a training set consisting of a large number q of sample pairs (x_k^s, u_k^s), $s = 1, \ldots, q$, where $u_k^s = \tilde{\mu}_k(x_k^s)$. Approximate $\tilde{\mu}_k$ using some form of parametric least squares fit.
- Example: Introduce a parametric family of policies $\mu_k(x_k, r_k)$, $k = 0, \ldots, N-1$, of some form, where r_k is a parameter vector. Then obtain r_k by

$$r_k \in \arg \min_r \sum_{s=1}^{q} \big\| u_k^s - \mu_k(x_k^s, r) \big\|^2$$

- Some other possibilities: Nearest neighbor optimization (use the control u_k^s of the sampled state x_k^s nearest to x_k), or interpolation.

4. MODEL-FREE METHODS AND SIMULATION (RL-OC, Section 2.1)

Model-Based Versus Model-Free Implementation for Stochastic Problems

Our use of the term "model-free": A method is called model-free if it involves calculations of expected values using Monte Carlo simulation.

Principal example

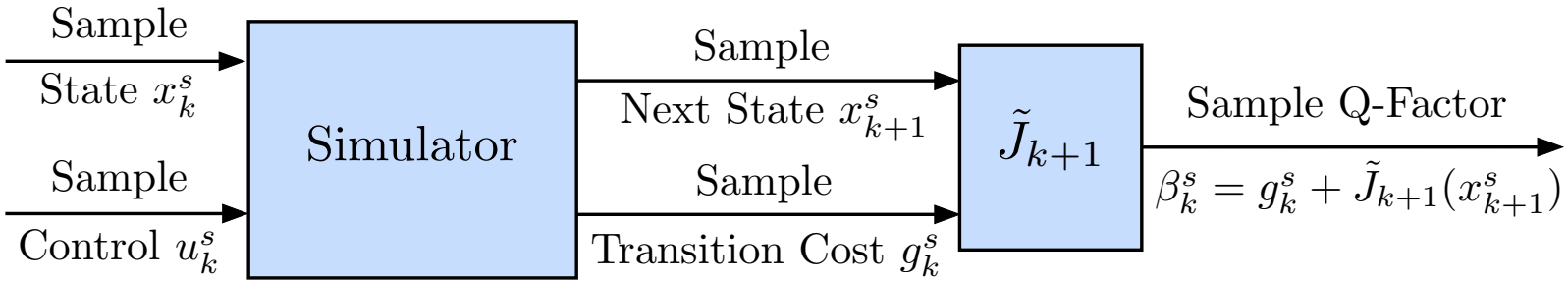
- Calculate $\tilde{J}_k$ in model-free fashion (rollout is a possibility to be discussed later).
- Implement model-free control minimization with one of three possible methods:
  - (1) On-line, at state x_k calculate by simulation the needed Q-factors for lookahead minimization

$$\tilde{Q}_k(x_k, u_k) = E\Big\{g_k(x_k, u_k, w_k) + \tilde{J}_{k+1}\big(f_k(x_k, u_k, w_k)\big)\Big\}$$

    and choose u_k that minimizes $\tilde{Q}_k(x_k, u_k)$
  - (2) Off-line "train" a parametric class of policies $\tilde{\mu}_k(x_k, r_k)$ by approximation in policy space on top of approximation in value space (see previous slide)
  - (3) Off-line "train" a parametric family of Q-factors $\tilde{Q}_k(x_k, u_k, r_k)$ and choose u_k that minimizes $\tilde{Q}_k(x_k, u_k, r_k)$ (see next slide)

Alternative: "Train" directly parametric Q-factors by Q-learning (RL-OC, Section 5.4)

Model-Free Q-Factor Parametric Approximation



- Use the simulator to collect a large number of "representative" samples of state-control-successor states-stage cost quadruplets $(x_k^s, u_k^s, x_{k+1}^s, g_k^s)$, and corresponding sample Q-factors

$$\beta_k^s = g_k^s + \tilde{J}_{k+1}(x_{k+1}^s), \qquad s = 1, \ldots, q$$

- Introduce a parametric family of Q-factors $\tilde{Q}_k(x_k, u_k, r_k)$.
- Determine the parameter vector $\bar{r}_k$ by the least-squares fit

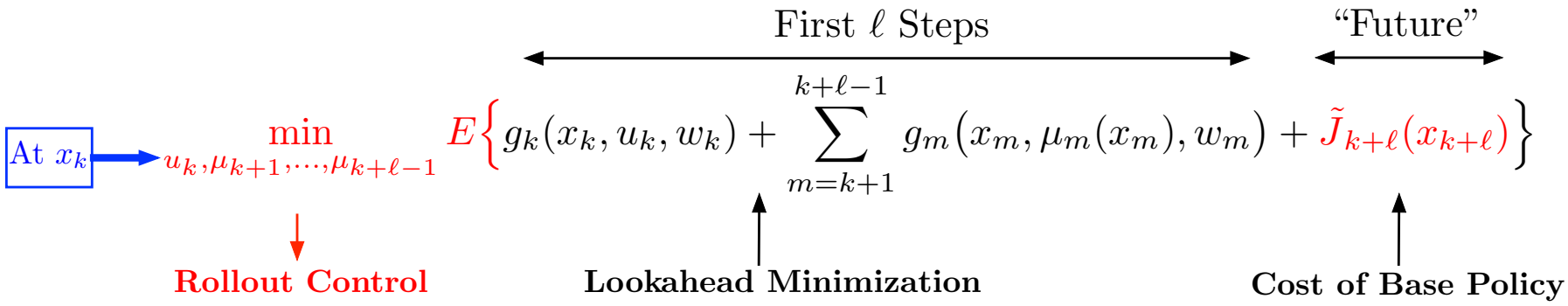
$$\bar{r}_k \in \arg\min_{r_k} \sum_{s=1}^{q} \left(\tilde{Q}_k(x_k^s, u_k^s, r_k) - \beta_k^s\right)^2$$

- Use the policy

$$\tilde{\mu}_k(x_k) \in \arg\min_{u_k \in U_k(x_k)} \tilde{Q}_k(x_k, u_k, \bar{r}_k)$$

5. POLICY IMPROVEMENT, ROLLOUT, SELF-LEARNING (RL-OC, Section 2.4)

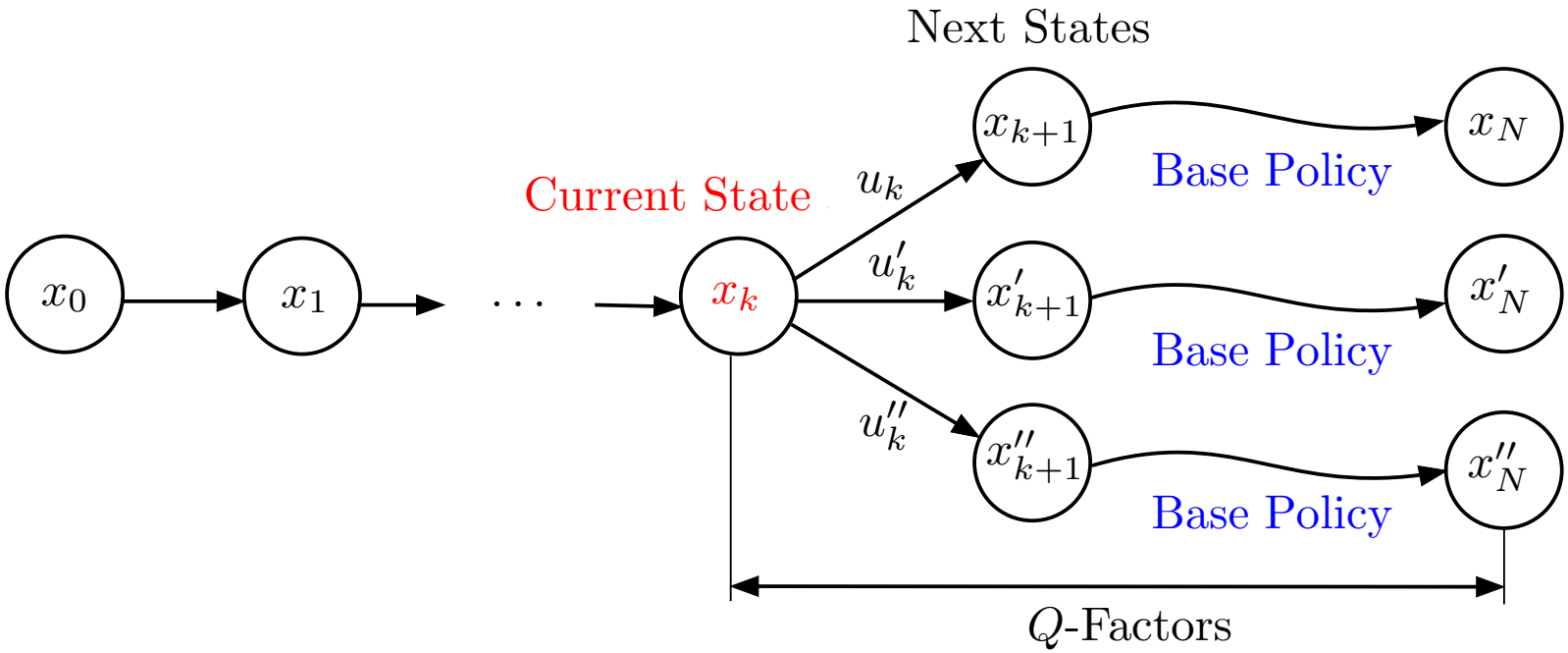
Main Rollout Idea: Start with a Base Policy, Get a Better Policy



Use the cost of the base/suboptimal policy at the end of ℓ-step lookahead

- Assume a base policy is available and can be simulated.
- The control $\tilde{\mu}_k(x_k)$ of the lookahead policy, can be computed at any x_k. It defines the rollout policy.
- The rollout policy performs better than the base policy. (Intuition: Using optimization in the first ℓ steps instead of using the base policy should work better.)
- In practice rollout performs well, is very reliable, is very simple to implement, can be model-free (particularly in the case $\ell = 1$).
- Rollout in its "standard" form involves simulation and can be implemented on-line.
- The simulation can be prohibitively expensive (so further approximations may be needed); particularly for stochastic problems and multistep lookahead.

Deterministic Rollout ($\ell = 1$) - Avoids Expensive Simulation

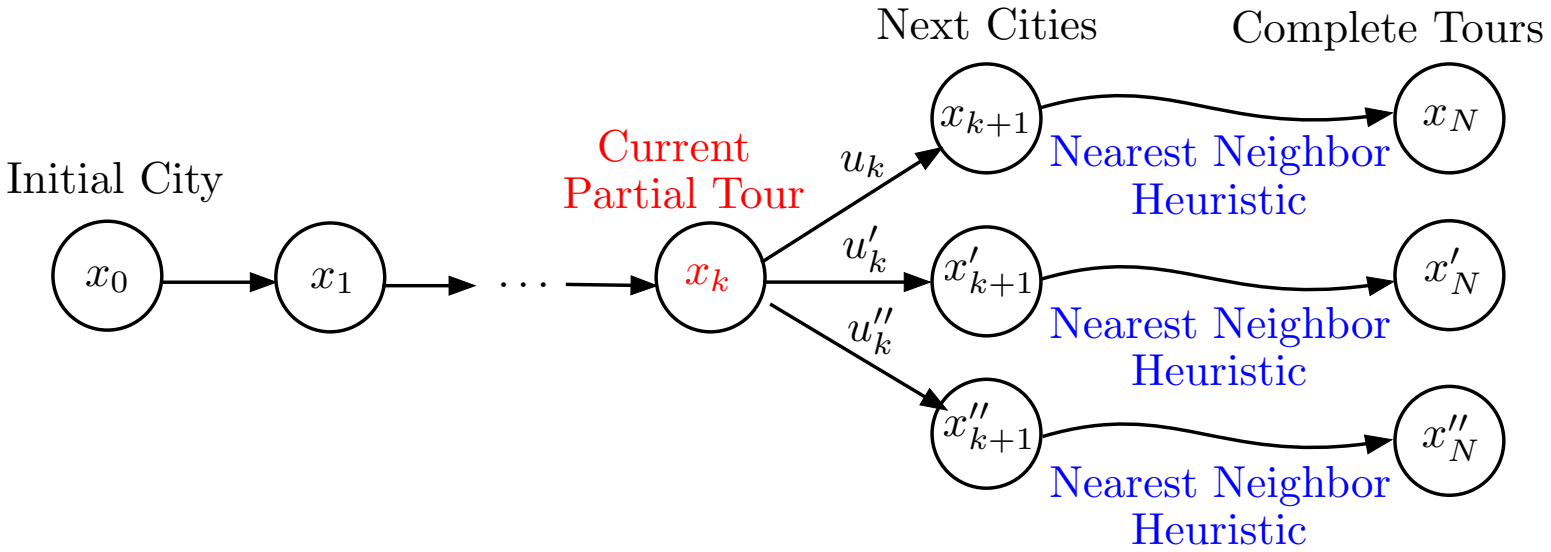


- At state x_k, for every pair (x_k, u_k), $u_k \in U_k(x_k)$, we generate a Q-factor

$$\tilde{Q}_k(x_k, u_k) = g_k(x_k, u_k) + H_{k+1}\big(f_k(x_k, u_k)\big)$$

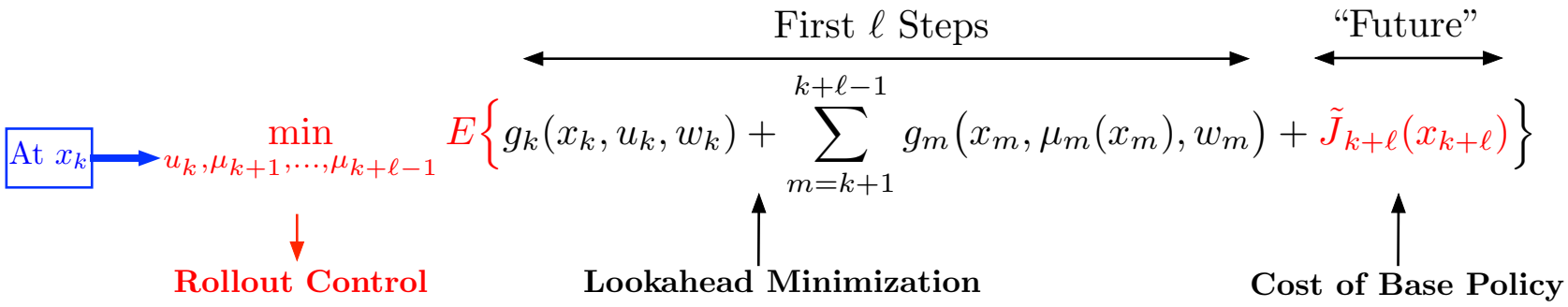
  using the base policy [$H_{k+1}(x_{k+1})$ is the base policy cost starting from x_{k+1}].
- We select the control u_k with minimal Q-factor.
- We move to the next state x_{k+1}, and continue.
- Multistep lookahead versions (length of lookahead limited by the branching factor of the lookahead tree).

Traveling Salesman Example of Rollout with a Greedy Heuristic



- N cities $c = 0, \ldots, N-1$; each pair of distinct cities c, c', has traversal cost $g(c, c')$.
- Find a minimum cost tour that visits each city once and returns to the initial city.
- Recall that it can be viewed as a shortest path/deterministic DP problem. States are the partial tours, i.e., the sequences of ordered collections of distinct cities.
- Nearest neighbor heuristic; sequentially extends the partial tour with the closest next city.
- Rollout algorithm: Start at some city; given a partial tour $\{c_0, \ldots, c_k\}$ of distinct cities, select as next city c_{k+1} the one that yielded the minimum cost tour under the nearest neighbor heuristic.

Stochastic Rollout: Learning (Cost Improvement) Through Self-Observation (Simulation)



- Start with a base policy $\pi = \{\mu_0, \ldots, \mu_{N-1}\}$.
- Let the rollout policy be $\tilde{\pi} = \{\tilde{\mu}_0, \ldots, \tilde{\mu}_{N-1}\}$. Then cost improvement is obtained

$$J_{k,\tilde{\pi}}(x_k) \leq J_{k,\pi}(x_k), \qquad \text{for all } x_k \text{ and } k.$$

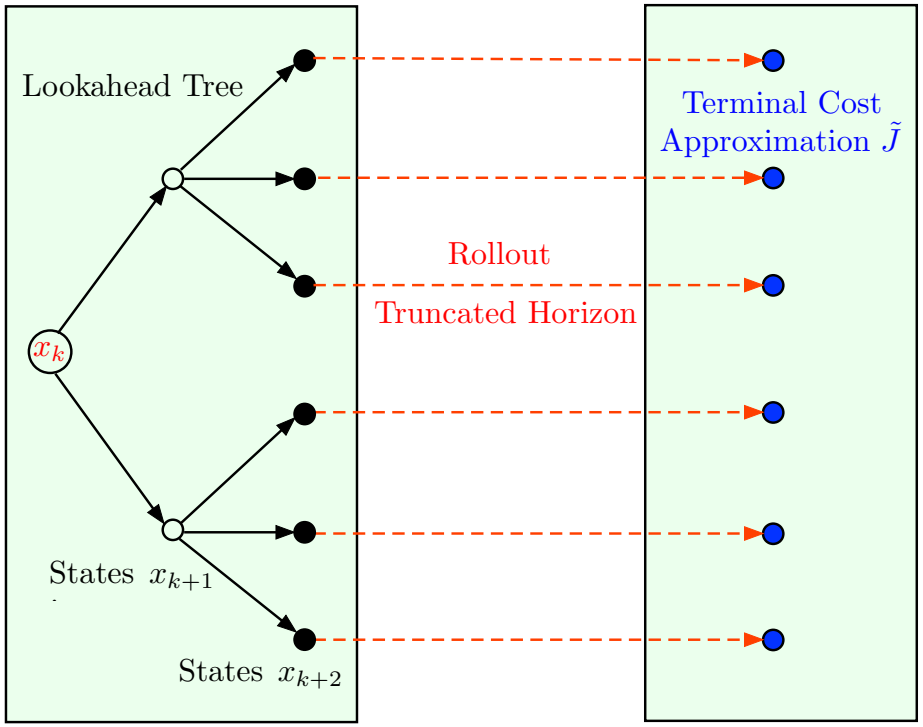
- This fundamental property carries over to policy iteration, which can be viewed as perpetual rollout:

$$\text{Start Policy} \implies \text{Rollout Policy} \implies \text{Rollout of Rollout Policy} \implies \cdots$$

- Approximate policy iteration (or self-learning): Use of simulation, and approximation in policy and/or value space, to learn sequentially improved policies.
- Many variants: Actor only, critic only, actor-critic, Q-learning methods.

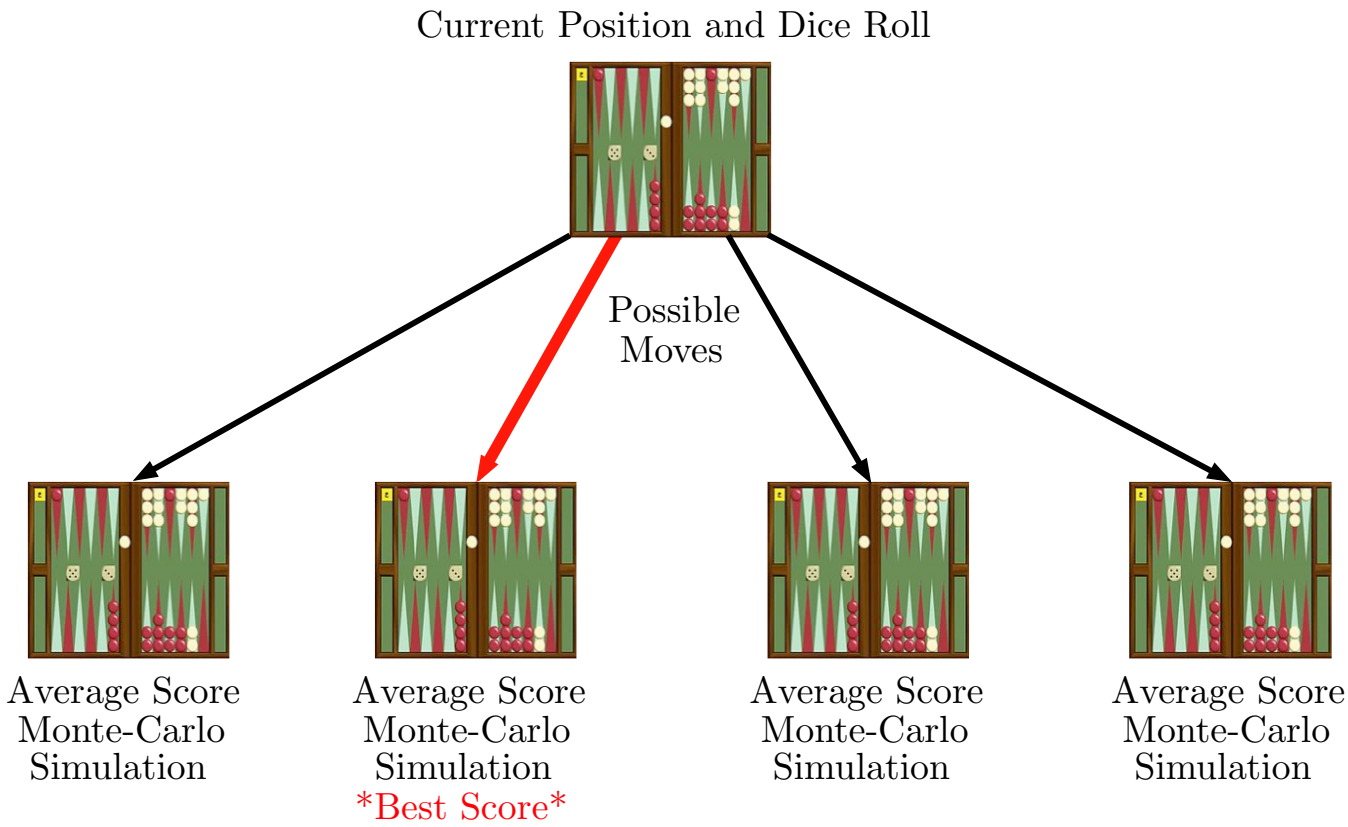
6. APPROXIMATE POLICY IMPROVEMENT, ADAPTIVE SIMULATION, Q-LEARNING
(RL-OC, Section 2.4)

Multistep Truncated Rollout with Cost Approximation



- Truncation of the rollout with cost approximation at the end saves computation.
- Cost improvement property holds approximately (error bounds in RL-OC, Section 5.1).
- Typically, longer lookahead leads to improved performance. However, long lookahead is costly.
- Experimentation with length of rollout and terminal cost function approximation are recommended.

Backgammon Example (Tesauro 1996)



- Truncated rollout with cost function approximation provided by TD-Gammon (earlier program involving a neural network trained by a form of policy iteration).
- Plays better than TD-Gammon, and better than any human.
- Too slow for real-time play (without parallel hardware), due to excessive simulation.

Monte Carlo Tree Search - Motivation

We assumed equal effort for evaluation of Q-factors of all controls at a state x_k

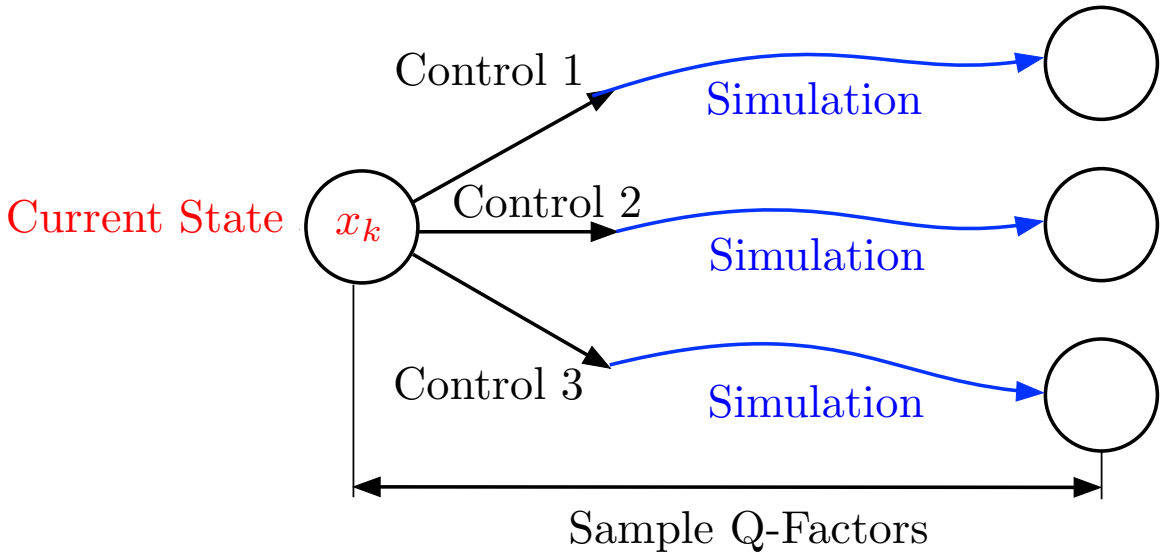
Drawbacks:

- Some of the controls u_k may be clearly inferior to others, and may not be worth as much sampling effort.
- Some of the controls u_k that appear to be promising, may be worth exploring better through multistep lookahead.

Monte Carlo tree search (MCTS) is a “randomized" form of lookahead

- MCTS aims to trade off computational economy with a hopefully small risk of degradation in performance.
- It involves adaptive simulation (simulation effort adapted to the perceived quality of different controls). Methods that use only few samples are called “optimistic".
- Aims to balance exploitation (extra simulation effort on controls that look promising) and exploration (adequate exploration of the potential of all controls).

Implementation of MCTS



Find a control $\tilde{u}_k$ that minimizes the approximate Q-factor

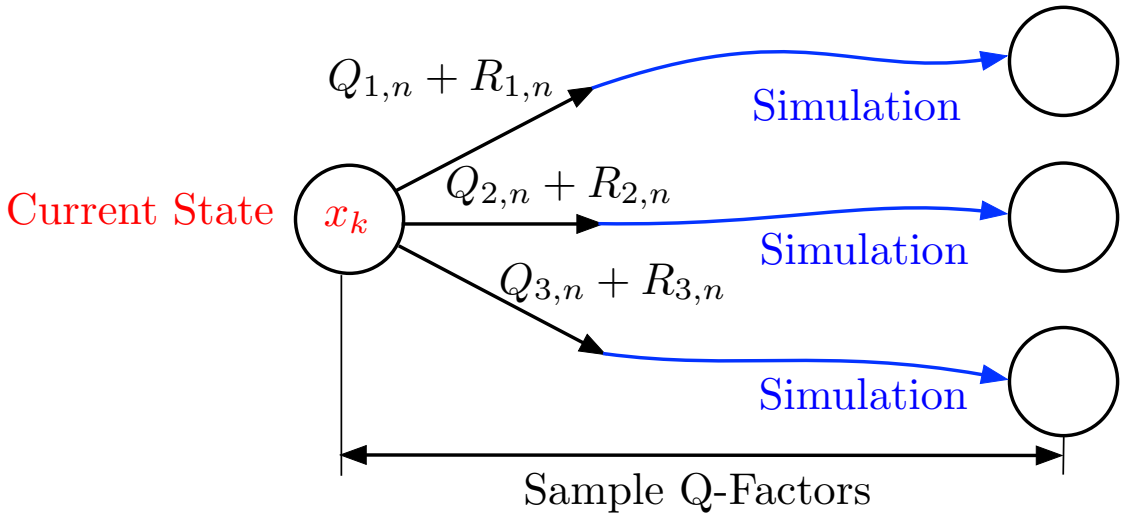
$$\tilde{Q}_k(x_k, u_k) = E\Big\{g_k(x_k, u, w_k) + \tilde{J}_{k+1}\big(f_k(x_k, u, w_k)\big)\Big\}$$

over $u_k \in U_k(x_k)$, by averaging samples of $\tilde{Q}_k(x_k, u_k)$.

Assume that $U_k(x_k)$ contains m elements, denoted $1, \ldots, m$

- After the nth sampling period we have $Q_{i,n}$, the empirical mean of the Q-factor of control i (total sample value divided by total number of samples).
- How do we use the estimates $Q_{i,n}$ to select the control to sample next?

Implementation of MCTS Based on Statistical Tests



A good sampling policy balances exploitation (sample controls that seem most promising, i.e., a small $Q_{i,n}$) and exploration (sample controls with small sample count).

- A popular strategy: Sample next the control i that minimizes the sum $Q_{i,n} + R_{i,n}$ where $R_{i,n}$ is an exploration index.
- $R_{i,n}$ is based on a confidence interval formula and depends on the sample count s_i of control i (which comes from analysis of multiarmed bandit problems).
- The UCB rule (upper confidence bound) sets $R_{i,n} = -c\sqrt{\log n / s_i}$, where c is a positive constant, selected empirically (values $c \approx \sqrt{2}$ are suggested, assuming that $Q_{i,n}$ is normalized to take values in the range $[-1, 0]$).
- MCTS with UCB rule has been extended to multistep lookahead.

Optimistic Policy Iteration with Q-Factor Approximation - Infinite Horizon

- In “optimistic" policy iteration, the policy is changed frequently (few samples are used for rollout/policy evaluation between policy updates).
- Implementation of policy changes with Q-factors: Introduce a parametric approximation $\tilde{Q}(x, u, r)$, and iterate on r.
- Each value of r defines a policy, which generates controls. We update r in a “direction of policy improvement" after a few samples.

SARSA/DQN: At iteration t, we have r^t, x^t, and we have chosen a control u^t

- We simulate the next transition to x^{t+1} and the transition cost g^t.
- We generate u^{t+1} with the minimization $u^{t+1} \in \arg\min_{u \in U(x^{t+1})} \tilde{Q}(x^{t+1}, u, r^t)$. [In some schemes, u^{t+1} is chosen with a small probability to be a different element of $U(x^{t+1})$ to enhance exploration (this is called off-policy exploration).]
- We update r^t in a gradient direction, aimed at solving a training/regression problem, via

$$r^{t+1} = r^t - \gamma^t \nabla_r \tilde{Q}(x^t, u^t, r^t) q^t,$$

where γ^t is a positive stepsize, and the scalar

$$q^t = \tilde{Q}(x^t, u^t, r^t) - \alpha \tilde{Q}(x^{t+1}, u^{t+1}, r^t) - g^t$$

is referred to as the temporal difference at iteration t.

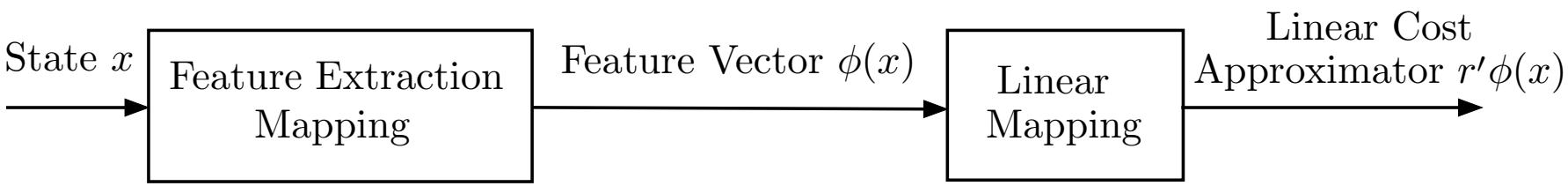
7. APPROXIMATION ARCHITECTURES, FEATURES, AND DEEP NEURAL NETS (RL-OC, Chapter 3)

The starting point: An approximation architecture and a target function

- 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: Adjust r to change $\tilde{J}$ and "match" the target function, usually by some form of least squares fit of the architecture to data relating to the target function.
- 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

- Quadratic polynomial approximation: $\tilde{J}(x, r)$ is quadratic in the components x^i 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.
- In aggregation (RL-OC, Chapter 6) we use piecewise constant and piecewise linear architectures.
- 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 (see RL-OC and the references there). Architectures involving features of the belief state (such as state estimates) are useful.

Examples of Domain-Specific Feature-Based Architectures

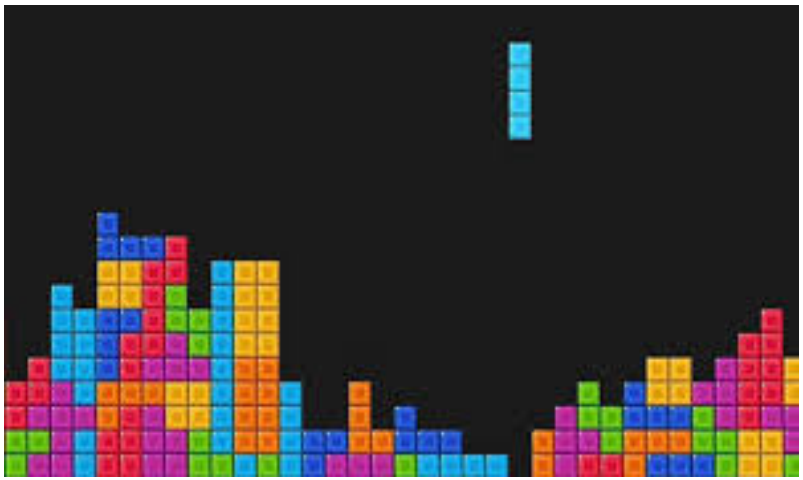
Features:
Material Balance,
Mobility,
Safety, etc

Feature Extraction

Weighting of Features

Win
"Probability"

Chess



Tetris

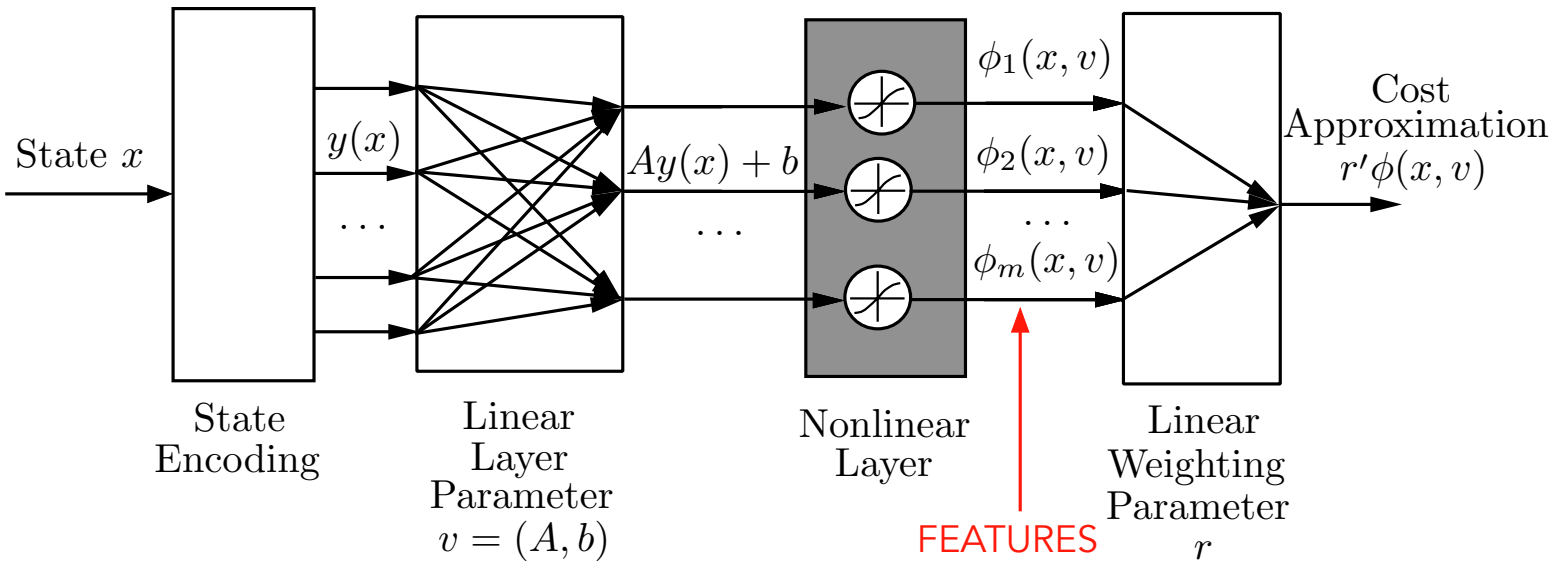
Features such as column heights, column height differentials, number of “holes" etc

Underparametrized and Overparametrized Architectures

If Number of Features > (or <) Number of Data Points, we say that the architecture is overparametrized (or underparametrized), respectively.

- Overparametrized architectures can be trained exactly (essentially with zero training error), i.e., they fit exactly/interpolate the data (this is also known as “overfitting").
- In this case, the training problem has many globally optimal solutions with equal (essentially zero) optimal cost. Some of these solutions are better than others in modeling the target function.
- Underparametrized training is subject to significant generalization error (good performance on the training set, bad model of the target function, i.e., much worse performance on more general data, outside the training set).
- To deal with this, a regularization term is added to underparametrized least squares training objectives (to essentially “fool" the optimization away from “overfitting").
- One of the exciting recent discoveries is that overparametrized architectures do not have nearly as much generalization error difficulty, assuming proper (but not very tricky) implementation, to find a good solution out of the many candidates.
- Current speculation is that this is the main reason for the success of deep neural nets (a primary example of overparametrized architecture).

Neural Nets: An Architecture that Automatically Constructs Features

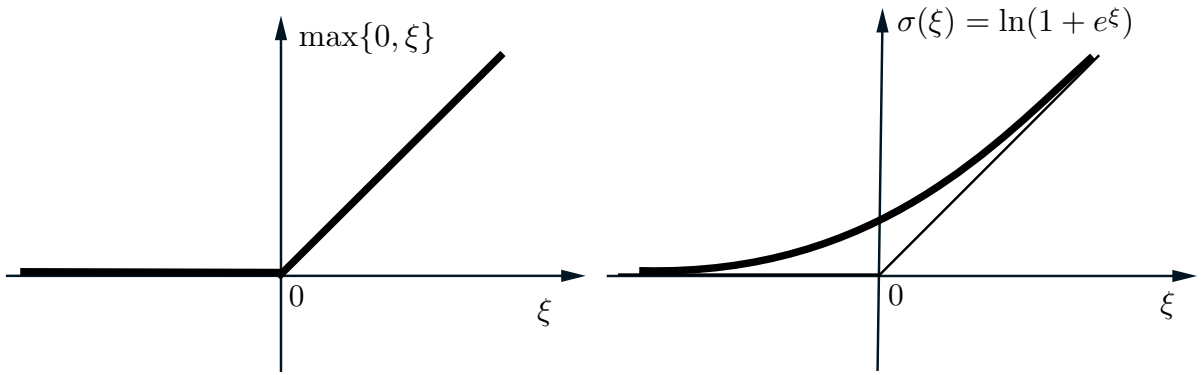


Given a set of state-cost training pairs (x^s, β^s), $s = 1, \ldots, q$, the parameters of the neural network (A, b, r) are obtained by solving the training problem

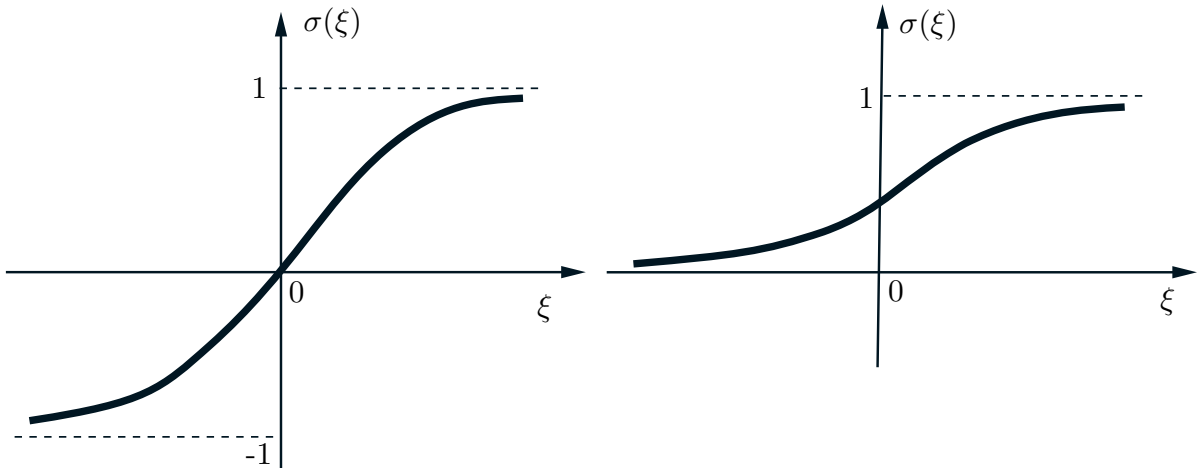
$$\min_{A,b,r} \sum_{s=1}^{q} \left(\sum_{\ell=1}^{m} r_\ell \sigma\left(\left(Ay(x^s) + b \right)_\ell \right) - \beta^s \right)^2$$

- Universal approximation property.
- An "incremental" gradient method (back-propagation) is typically used for training.
- The state encoding $y(x)$ may include problem-dependent features, thought to be "informative"; this may be important for success in practice.

Rectifier and Sigmoidal Nonlinearities

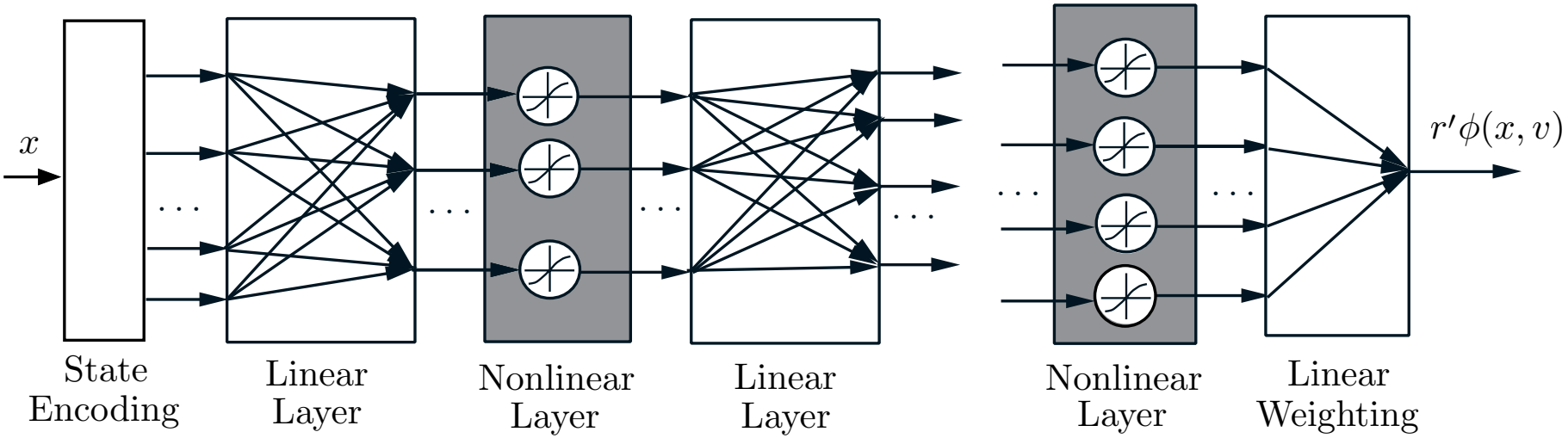


The rectified linear unit $\sigma(\xi) = \ln(1 + e^{\xi})$. It is the rectifier function $\max\{0, \xi\}$ with its corner "smoothed out"



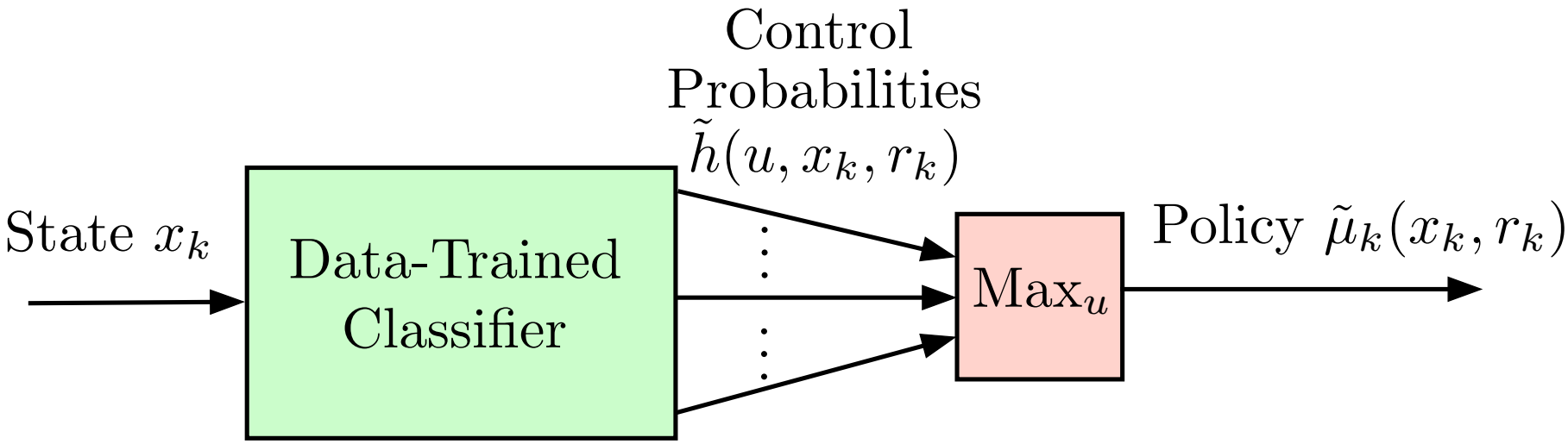
Sigmoidal units: The hyperbolic tangent function $\sigma(\xi) = \tanh(\xi) = \frac{e^{\xi} - e^{-\xi}}{e^{\xi} + e^{-\xi}}$ is on the left. The logistic function $\sigma(\xi) = \frac{1}{1+e^{-\xi}}$ is on the right.

Deep Neural Networks



- A deep NN is a multilayer network that provides a hierarchy of features (each set of features being a function of the preceding set of features).
- 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.
- The large number of deep NN weights (overparametrization) may contribute to their ability to produce good predictors of the target function.
- Deep NNs have been found more effective than shallow NNs in some important problem settings.
- There is a lot of research and speculation regarding their success.

Parametric Approximation of Policies (RL-OC, Section 3.5)



A given policy can be implemented approximately 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.

8. INCREMENTAL AND STOCHASTIC GRADIENT OPTIMIZATION (RL-OC, Sections 3.1.2, 3.1.3)

Incremental Gradient Methods

Generic training problem: Minimize a sum of cost function components (each corresponding to some data)

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^{t+1} from y^t according to

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

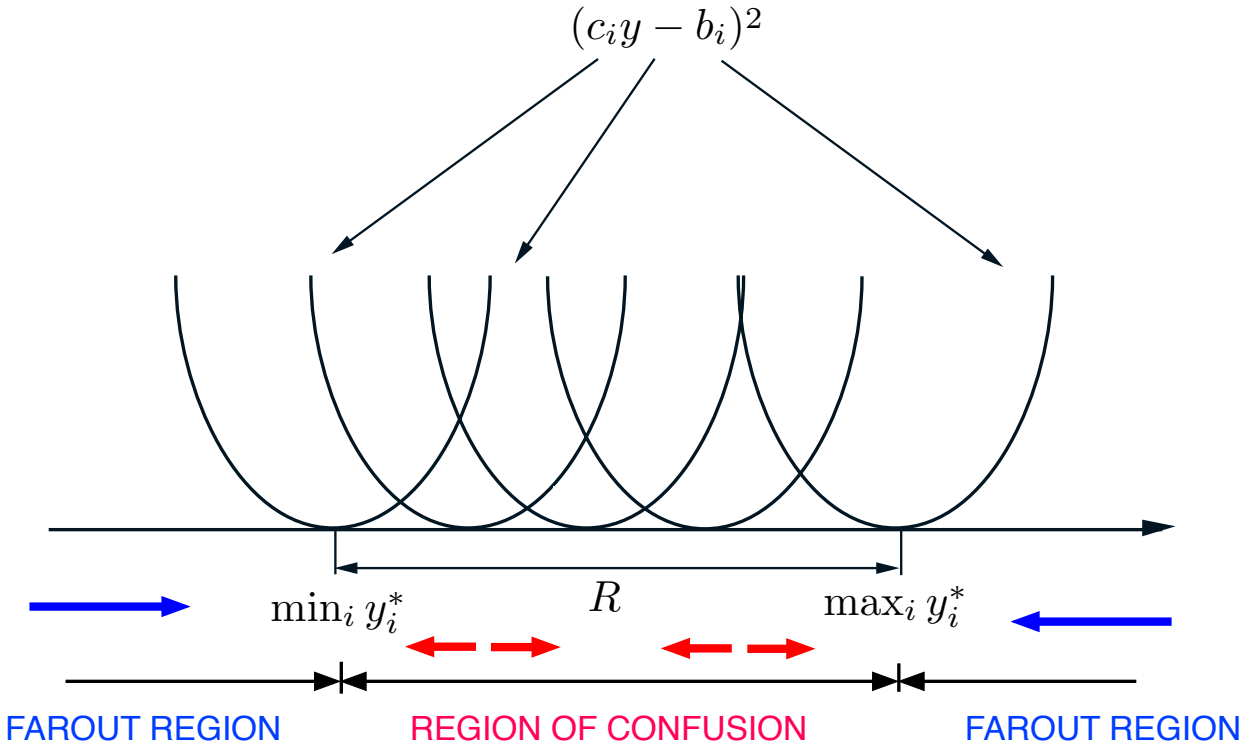
where $\gamma^t > 0$ is a stepsize parameter.

The incremental gradient counterpart

Choose an index i_t and iterate according to:

$$y^{t+1} = y^t - \gamma^t \nabla f_{i_t}(y^t)$$

The Advantage of Incrementalism



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

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.

The incremental aggregated method aims at acceleration

- Evaluates gradient of a single term at each iteration.
- Uses previously calculated gradients as if they were up to date

$$y^{t+1} = y^t - \gamma^t \sum_{\ell=0}^{m-1} \nabla f_{i_{t-\ell}}(y^{t-\ell})$$

- Has theoretical and empirical support, and it is often preferable.

Stochastic gradient method (also called stochastic gradient descent or SGD)

- Applies to minimization of $f(y) = E\{F(y,w)\}$ where w is a random variable
- Has the form

$$y^{t+1} = y^t - \gamma^t \nabla_y F(y^t, w^t)$$

  where w^t is a sample of w and $\nabla_y F$ denotes gradient of F with respect to y.
- The incremental gradient method with random index selection is the same as SGD [convert the sum $\sum_{i=1}^m f_i(y)$ to an expected value, where i is random with uniform distribution].

Implementation Issues of Incremental Methods - Alternative Methods

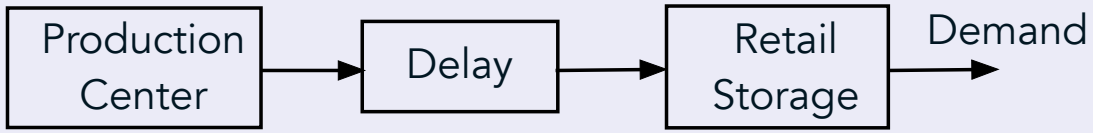
- How to pick the stepsize γ^t (usually $\gamma^t = \frac{\gamma}{t+1}$ or similar).
- How to deal (if at all) with region of confusion issues (detect being in the region of confusion and reduce the stepsize).
- How to select the order of terms to iterate (cyclic, random, other).
- Dealing with issues of underparametrization and overparametrization.
- Diagonal scaling (a different stepsize for each component of y).
- Alternative methods (more ambitious): Incremental Newton method, diagonalized Newton Method, extended Kalman filter (see RL-OC, Section 3.1.3, or the author's Nonlinear Programming book).
- Issues of parallel and asynchronous computation in special (partitioned) types of architectures.
- For extensive discussions see the author's book "Nonlinear Programming," 3rd edition, Athena Scientific, 2016, and the references given there.

9. DIRECT POLICY OPTIMIZATION: A MORE GENERAL APPROACH (RL-OC, Section 5.7.1)

General Framework for Approximation in Policy Space

- Parametrize stationary policies with a parameter vector r; denote them by $\tilde{\mu}(r)$, with a component $\tilde{\mu}(x, r)$ for each state x. Each r defines a policy.
- Parametrization may be problem-specific, or may involve a neural network.
- The idea is to optimize some measure of performance with respect to r.
- Approximation in policy space applies more broadly; relies less on DP structure.

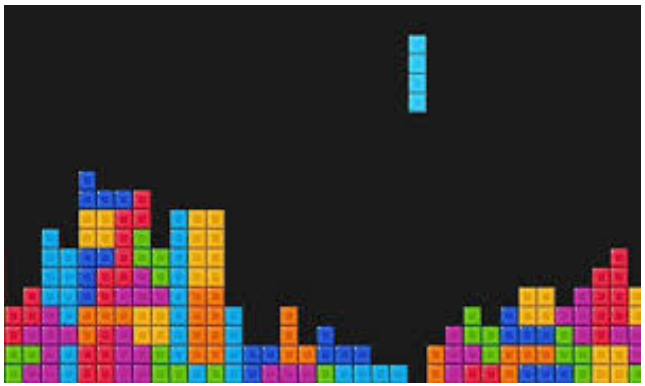
An example of problem-specific/natural parametrization: Supply chains, inventory control



- Retail center places orders to the production center, depending on current stock; there may be orders in transit; demand and delays can be stochastic.
- State is (current stock, orders in transit, etc). Can be formulated by DP but can be very difficult to solve exactly.
- Intuitively, a near-optimal policy is of the form: When the retail inventory goes below level r_1, order an amount r_2. Optimize over the parameter vector $r = (r_1, r_2)$.
- Extensions to a network of production/retail centers, multiple products, etc.

Another Example: Policy Parametrization Through Value Parametrization

- Suppose $\tilde{J}(x, r)$ is a given cost function parametric approximation.
- $\tilde{J}$ may be a linear feature-based architecture that is natural for the given problem.
- Define $$\tilde{\mu}(x, r) \in \arg \min_{u \in U(x)} E\Big\{ g(x, u, x) + \tilde{J}\big(f(x, u, w), r\big) \Big\}$$
- This is useful when we know a good parametrization in value space, but we want to use a method that works well in policy space, and results in an easily implementable policy.



Tetris example: There are good linear parametrizations through features. Great success has been achieved by indirect approximation in policy space.

Policy Approximation on Top of Value Approximation

- Compute approximate cost-to-go function $\tilde{J}$ using an approximation in value space scheme.
- This defines the corresponding suboptimal policy $\hat{\mu}$ through one-step lookahead,

$$\hat{\mu}(x,r) \in \arg\min_{u \in U(x)} E\Big\{ g(x,u,w) + \tilde{J}\big(f(x,u,w),r\big) \Big\}$$

  or a multistep lookahead version.
- Approximate $\hat{\mu}$ using a training set consisting of a large number q of sample pairs (x^s, u^s), $s = 1, \ldots, q$, where $u^s = \hat{\mu}(x^s)$.
- In particular, introduce a parametric family of policies $\tilde{\mu}(x,r)$. Then obtain r by

$$\min_r \sum_{s=1}^{q} \big\| u^s - \tilde{\mu}(x^s,r) \big\|^2.$$

Learning from a Software or Human Expert

- Suppose we have a software or human expert that can choose a “good" or “near-optimal" control u at any state x.
- We use the expert to generate a sample set of representative state-control pairs (x^s, u^s), $s = 1, \ldots, q$.
- We introduce a parametric family of policies $\tilde{\mu}(x, r)$. Then obtain r by

$$\min_r \sum_{s=1}^{q} \left\| u^s - \tilde{\mu}(x^s, r) \right\|^2.$$

- This approach is known as expert supervised training.
- It has been used (in various forms) in backgammon and in chess.
- It can be used, among others, for initialization of other methods (such as rollout) with a reasonably good policy.

Unconventional Information Structures

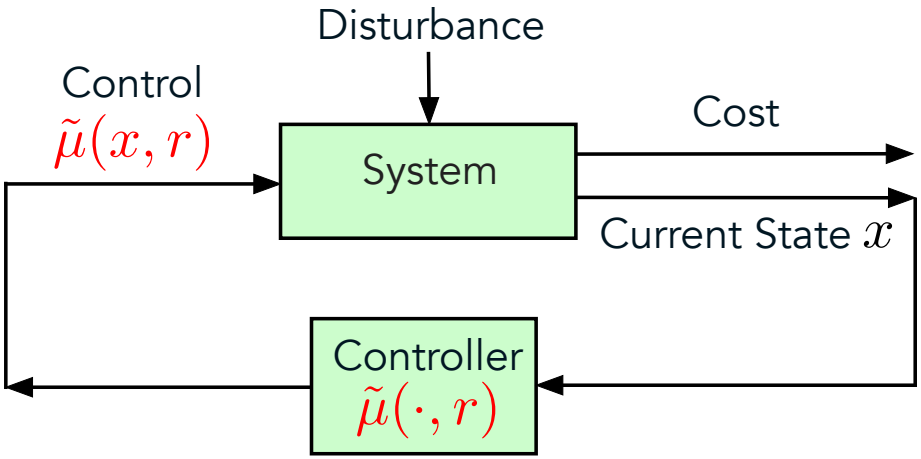
- Approximation in value space is based on a DP formulation, so the controller has access to the exact state (or a belief state in case of partial state information).
- In some contexts this may not be true. There is a DP-like structure, but no full state or belief state is available.
- Example 1: The controller “forgets" information, e.g., has “limited memory".
- Example 2: Some control components may be chosen on the basis of different information that others.

Example: Multiagent systems with local agent information

- Suppose decision making and information gathering is distributed among multiple autonomous agents.
- Each agent’s action depends only on his/her local information.
- Agents may be receiving delayed information from other agents.
- Then conventional DP and much of the approximation in value space methodology breaks down.
- Approximation in policy space is still applicable.

10. GRADIENT AND RANDOM SEARCH METHODS FOR DIRECT POLICY OPTIMIZATION (RL-OC, Section 5.7.1-5.7.2)

Optimization/Training Framework



Training by Cost Optimization

- Each r defines a stationary policy $\tilde{\mu}(r)$, with components $\tilde{\mu}(x,r)$.
- Determine r through the minimization

$$\min_r J_{\tilde{\mu}(r)}(x_0)$$

where $J_{\tilde{\mu}(r)}(x_0)$ is the cost of the policy $\tilde{\mu}(r)$ starting from initial state x_0.

- More generally, determine r through the minimization

$$\min_r E\{J_{\tilde{\mu}(r)}(x_0)\}$$

where the $E\{\cdot\}$ is with respect to a suitable probability distribution of x_0.

Policy Gradient Method for Stochastic Problems

Consider the generic optimization problem $\min_{z \in Z} F(z)$

We take an unusual step: Convert this problem to the stochastic optimization problem

$$\min_{p \in \mathcal{P}_Z} E_p\{F(z)\}$$

where

- z is viewed as a random variable.
- $\mathcal{P}_Z$ is the set of probability distributions over Z.
- p denotes the generic distribution in $\mathcal{P}_Z$.
- $E_p\{\cdot\}$ denotes expected value with respect to p.

How does this relate to our DP problems?

- For this framework to apply to a stochastic DP context, we must enlarge the set of policies to include randomized policies, mapping a state x into a probability distribution over the set of controls $U(x)$.
- Note that in our DP problems, optimization over randomized policies gives the same results as optimization over ordinary/nonrandomized policies.
- In the DP context, z is the state-control trajectory: $z = \{x_0, u_0, x_1, u_1, \ldots\}$.

Parametrization of the probability distributions

- We restrict attention to a parametrized subset $\tilde{\mathcal{P}}_Z \subset \mathcal{P}_Z$ of probability distributions $p(z; r)$, where r is a continuous parameter.
- In other words, we approximate the problem $\min_{z \in Z} F(z)$ with the restricted problem

$$\min_r E_{p(z;r)}\{F(z)\}$$

- We use a gradient method for solving this problem:

$$r^{t+1} = r^t - \gamma^t \nabla \Big(E_{p(z;r^t)}\{F(z)\} \Big)$$

- Key fact: There is a useful formula for the gradient, which involves the gradient with respect to r of the natural logarithm $\log\big(p(z; r^t)\big)$.

Log-Likelihood Policy Gradient Method

The Gradient Formula: Assuming that $p(z; r^t)$ is a discrete distribution,

$$\nabla\Big(E_{p(z;r^t)}\{F(z)\}\Big) = \nabla\left(\sum_{z\in Z} p(z;r^t)F(z)\right)$$

$$= \sum_{z\in Z} \nabla p(z;r^t)F(z)$$

$$= \sum_{z\in Z} p(z;r^t)\frac{\nabla p(z;r^t)}{p(z;r^t)}F(z)$$

$$= E_{p(z;r^t)}\left\{\nabla\Big(\log\big(p(z;r^t)\big)\Big)F(z)\right\}$$

Sample-Based Gradient Method for Parametric Approximation of $\min_{z\in Z} F(z)$

- At r^t obtain a sample z^t according to the distribution $p(z; r^t)$.
- Compute the sample gradient $\nabla\Big(\log\big(p(z^t; r^t)\big)\Big)F(z^t)$.
- Use it to iterate according to

$$r^{t+1} = r^t - \gamma^t \nabla\Big(\log\big(p(z^t; r^t)\big)\Big)F(z^t)$$

Policy Gradient Method - Finite-State Discounted Problem

- Denote by z the infinite horizon state-control trajectory: $z = \{i_0, u_0, i_1, u_1, \ldots\}$.
- We consider a parametrization of randomized policies $p(u \mid i; r)$ with parameter r, i.e., at state i, u is generated according to a distribution $p(u \mid i; r)$ over $U(i)$.
- Example: Soft-min parametrization using a Q-factor parametrization $\tilde{Q}(i, u, r)$,

$$p(u \mid i; r) = \frac{e^{-\beta \tilde{Q}(i,u,r)}}{\sum_{v \in U(i)} e^{-\beta \tilde{Q}(i,v,r)}}$$

- Then for a given r, the state-control trajectory z is a random trajectory with probability distribution denoted $p(z; r)$.
- The cost corresponding to the trajectory z is $F(z) = \sum_{m=0}^{\infty} \alpha^m g(i_m, u_m, i_{m+1})$, and the problem is to minimize $E_{p(z;r)}\{F(z)\}$, over r.
- The gradient needed in the gradient iteration

$$r^{t+1} = r^t - \gamma^t \nabla\Big(\log\big(p(z^t; r^t)\big)\Big) F(z^t)$$

is given by

$$\nabla\Big(\log\big(p(z^t; r^t)\big)\Big) = \sum_{m=0}^{\infty} \log\big(p_{i_m i_{m+1}}(u_m)\big) + \sum_{m=0}^{\infty} \nabla\Big(\log\big(p(u_m \mid i_m; r^t)\big)\Big)$$

Training by Random Search

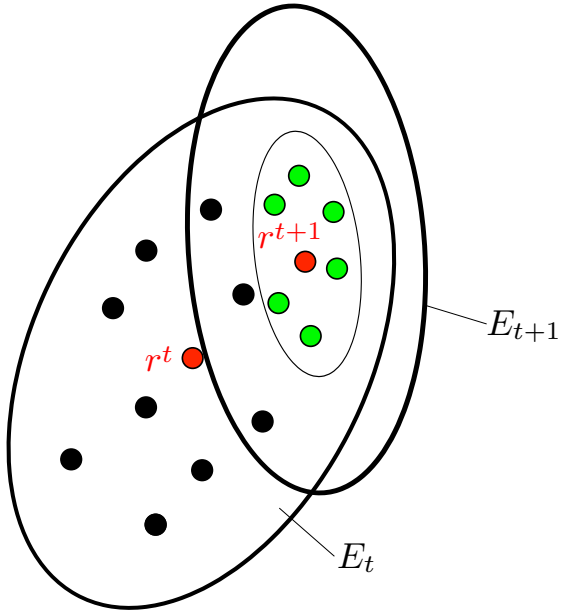
Random search methods apply to the general minimization $\min_{r \in R} F(r)$

- They generate a parameter sequence $\{r^t\}$ aiming for cost reduction.
- Given r^t, points are chosen in some random fashion in a neighborhood of r^t, and some new point r^{t+1} is chosen within this neighborhood.
- In theory they have good convergence properties. In practice they can be slow.
- They are not affected as much by local minima (as for example gradient-type methods).
- They don't require a differentiable cost function, and they apply to discrete as well as continuous minimization.
- There are many methods and variations thereoff.

Some examples

- Evolutionary programming.
- Tabu search.
- Simulated annealing.
- Cross entropy method.

Cross-Entropy Method - A Sketch



- At the current iterate r^k, construct an ellipsoid E_k centered at r^k.
- Generate a number of random samples within E_k. “Accept" a subset of the samples that have “low" cost.
- Let r^{k+1} be the sample “mean" of the accepted samples.
- Construct a sample “covariance" matrix of the accepted samples, form the new ellipsoid E_{k+1} using this matrix, and continue.
- Limited convergence rate guarantees. Success depends on domain-specific insight and the skilled use of implementation heuristics.
- Simple and well-suited for parallel computation. Resembles a “gradient method".

FINAL REMARKS

A List of Some Important Topics we Did Not Cover Fully (or at all)

- Infinite horizon extensions: Approximate value and policy iteration, error bounds, stochastic shortest path and average cost problems (RL-OC, Sections 5.1-5.3)
- Optimistic (few sample) and multistep variants of self-learning and policy iteration (RL-OC, Sections 5.3-5.4)
- Sampling for exploration, in the context of policy evaluation (RL-OC, Section 5.3.4)
- Temporal difference methods: A class of methods for approximate policy evaluation in infinite horizon problems with a rich theory (RL-OC, Section 5.5)
- Q-learning, with and without approximations (RL-OC, Section 5.4)
- Many variants of policy approximation methods: actor-critic methods, policy gradient and random search methods (RL-OC, Section 5.7)
- Approximate linear programming: Start with a linear program corresponding to a discounted infinite horizon problem. Use its optimal solution for approximation in value space (RL-OC, Section 5.6).
- Aggregation: Construct a “smaller" aggregate problem using “aggregate states". Use its optimal cost function for approximation in value space (RL-OC, Chapter 6)
- Special aspects of deterministic problems: Shortest paths and their use in approximate DP (RL-OC, Sections 1.3.1, 2.4.1)
- Special aspects of partial observation problems (POMDP): (RL-OC, Sections 1.3.6, 3.1.1, 5.7.3)
- Multiagent rollout and policy iteration: Reduces dramatically the cost for lookahead minimization (see the book "Rollout, Policy Iteration and Distributed RL", 2020)

Concluding Remarks

Some words of caution about RL

- There are challenging implementation issues in all approaches, and no fool-proof methods
- Good features are helpful, but finding them requires domain-specific knowledge (or else use a NN and a potentially long training process)
- Training algorithms are not as reliable as you might think by reading the literature
- Approximate policy iteration/self-learning involves exploration challenges (i.e., "representative" training sets)
- Recognizing success or failure can be difficult!
- The RL successes in game contexts are spectacular, but they have benefited from perfectly known and stable models and small number of controls (per state)
- Problems with partial state observation remain a big challenge

On machine learning (Steven Strogatz, NY Times Article, Dec. 2018)

"What is frustrating about machine learning is that the algorithms can't articulate what they're thinking. We don't know why they work, so we don't know if they can be trusted ... As human beings, we want more than answers. We want insight. This is going to be a source of tension in our interactions with computers from now on."

On the Positive Side

- Massive computational power together with distributed computation are a source of hope.
- Silver lining: We can begin to address practical optimization problems of unimaginable difficulty!
- There is an exciting journey ahead!

Thank you!