Topics in Reinforcement Learning: Rollout and Approximate Policy Iteration

ASU, CSE 691, Spring 2021

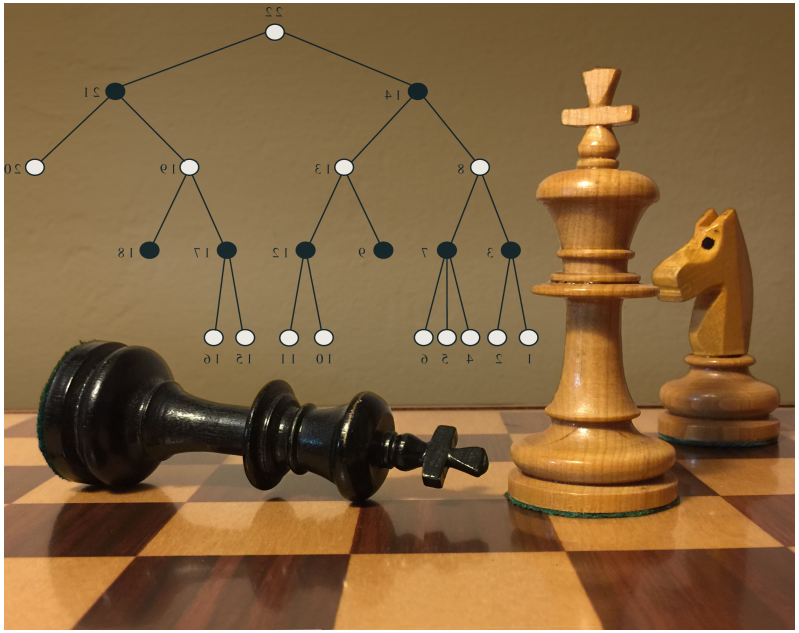
Links to Class Notes, Videolectures, and Slides at
http://web.mit.edu/dimitrib/www/RLbook.html

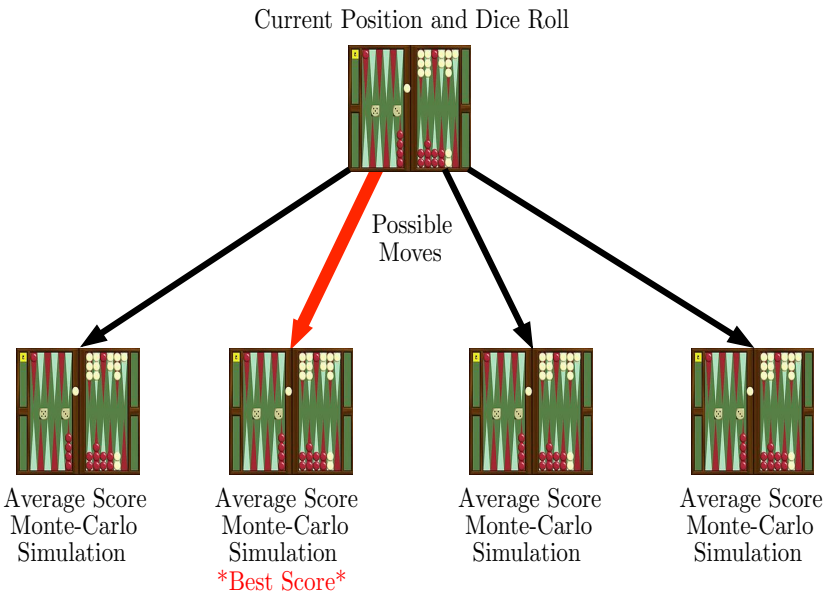
Dimitri P. Bertsekas
dbertsek@asu.edu

Lecture 13
Overview of the Entire Course

Outline

Chess and Backgammon - Off-Line Training and On-Line Play





These programs use two distinct types of algorithms:

- Off-line training (learning process): Precomputes value and policy approximations (or other useful quantities)
- On-line play (actual play): Multistep lookahead, rollout (and the precomputed value and/or policy approximations)

Strong Connections to Dynamic Programming

Principal ideas:

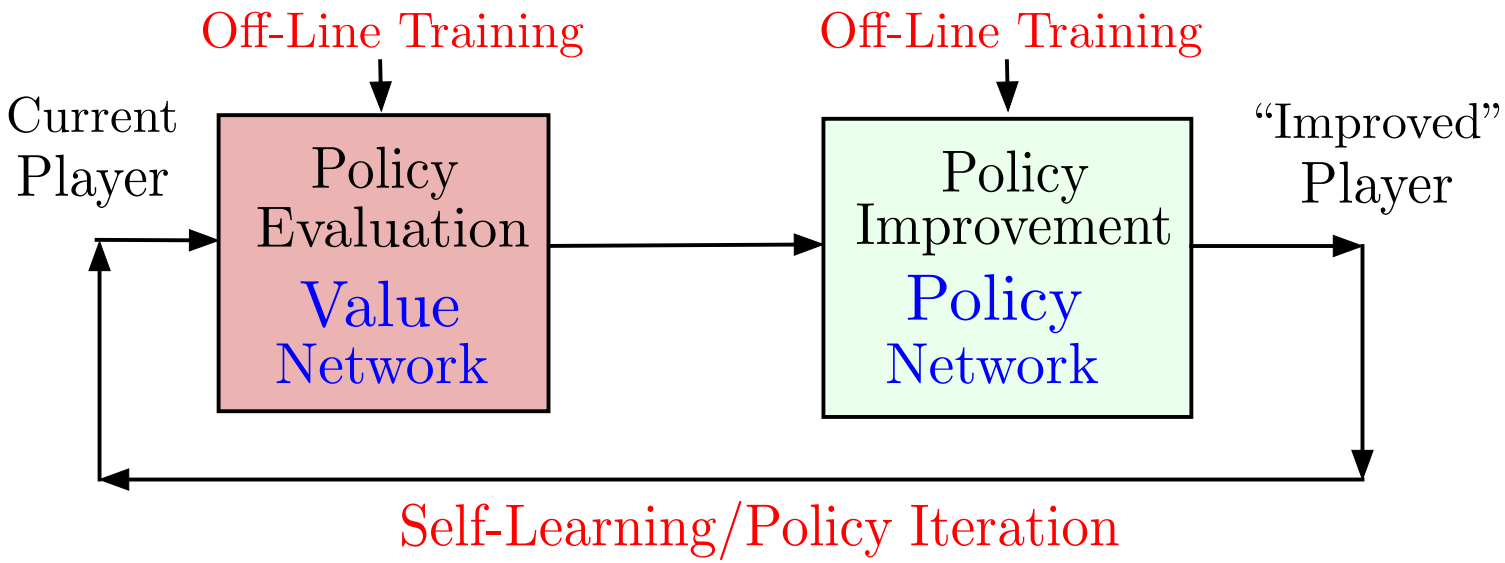
- Rollout: Given a policy, compute a new policy by lookahead minimization
- Policy iteration: Repeated rollout
- Policy improvement principle: Each new policy performs better than the preceding one (fundamental property in policy iteration)
- Use approximations to deal with the curse of dimensionality
- Approximation in value and policy space: For example with neural networks or problem approximation

OUR FOCUS AND POINT OF VIEW IN THIS COURSE

We develop, extend, and unify this methodology, so it applies far more generally to:

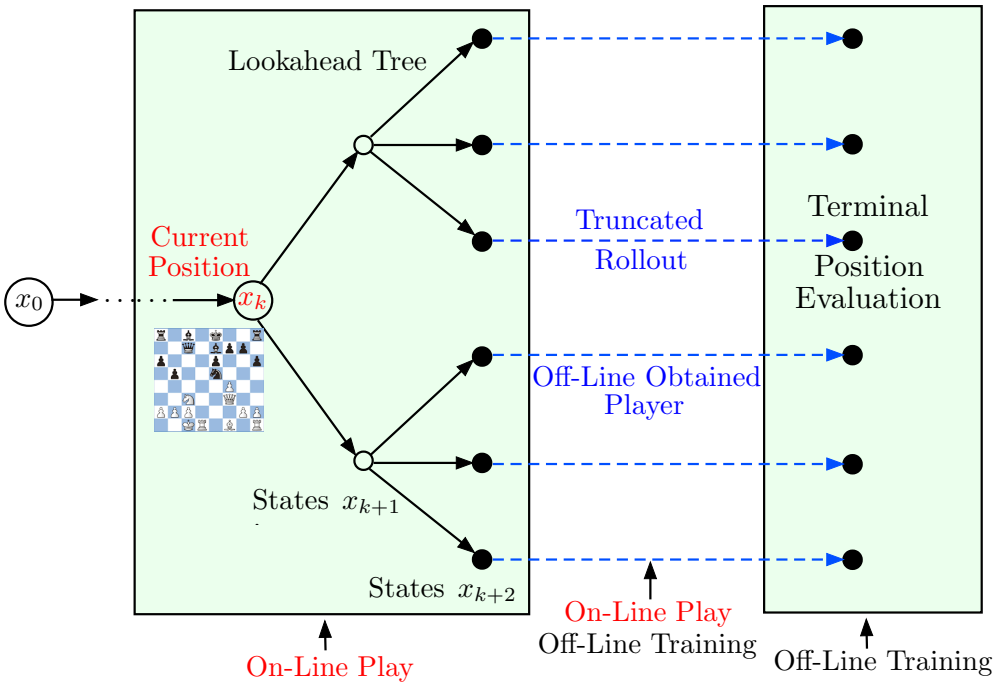
- Dynamic Programming and Operations Research applications
- Optimal Control, including model predictive control, robotics, and planning
- Multiagent problems
- Challenging/intractable Combinatorial Optimization (integer programming) problems
- Decision and control in a Changing Environment (adaptive control)

AlphaZero/AlphaGo/TD-Gammon Off-Line Training by Policy Iteration Using Self-Generated Data



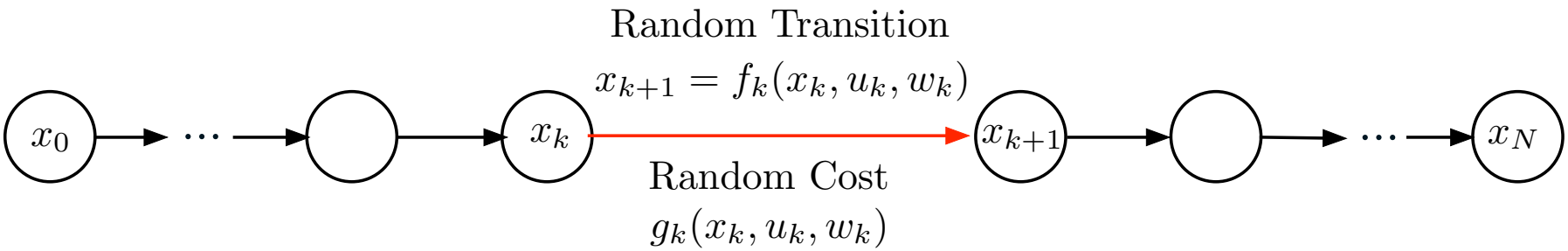
- The "current" player plays games that are used to "train" an "improved" player
- At a given position, the "value" of a position and the "move probabilities" (the player) are approximated by neural nets
- Successive neural nets are trained using self-generated data and a form of regression
- TD-Gammon (for the game of backgammon - Tesauro 1992) has similar structure, but uses only a value network (which also serves as substitute to a policy network via lookahead minimization)

AlphaZero/AlphaGo/TD-Gammon On-Line Play: Multistep Lookahead, Rollout, and Terminal Cost Approximation



- Off-line training yields a “value network" and a “policy network" that provide a position evaluation function and a default/base policy to play
- On-line play improves the default/base policy by:
  - Searching forward for several moves
  - Simulating the base policy for some more moves - approximating the effect of future moves by using the terminal position evaluation
- Backgammon programs by Tesauro (1992, 1996) use only an off-line trained value network

Stochastic Finite Horizon DP Problems



- System $x_{k+1} = f_k(x_k, u_k, w_k)$ with state x_k, control u_k, and random "disturbance" w_k
- 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

Produces the optimal costs $J_k^*(x_k)$ of the tail subproblems that start at x_k

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: $J_0^*(x_0) = J^*(x_0)$.

On-line implementation of the optimal policy, given $J_1^*, \ldots, J_N^*$

Sequentially, going forward, for $k = 0, 1, \ldots, N-1$, 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 know J_{k+1}^*, compute $E\{\cdot\}$ for each u_k, minimize over all u_k

Approximation in value space: Use $\tilde{J}_k$ in place of J_k^*; approximate $E\{\cdot\}$ and $\min_{u_k}$ (the three approximations)

Note the division between precomputation/learning (off-line training to obtain J_k^*) and real-time implementation (on-line play to obtain u_k^*)

Finite horizon opt. costs –> Infinite horizon opt. cost: Consider the k-stages problem

Let $J_k(x)$ = k-stages optimal cost starting from x. Then

$$J^*(x) = \lim_{k\to\infty} J_k(x), \qquad \text{for all states } x \quad (??)$$

J^* satisfies Bellman's equation:

$$J^*(x) = \min_{u\in U(x)} E\Big\{g(x,u,w) + \alpha J^*\big(f(x,u,w)\big)\Big\}, \qquad \text{for all states } x \quad (??)$$

Optimality condition: Let $\mu^*(x)$ attain the min in the Bellman equation for all x

The policy $\{\mu^*, \mu^*, \ldots\}$ is optimal (??)

The three theorems hold for the finite-state problems that we have considered (discounted and SSP)

Value iteration (VI): Generates finite horizon opt. cost function sequence $\{V_k\}$

$$V_k(x) = \min_{u \in U(x)} E\Big\{g(x,u,w) + \alpha V_{k-1}\big(f(x,u,w)\big)\Big\}, \qquad V_0 \text{ is "arbitrary" (??)}$$

Policy Iteration (PI): Generates sequences of policies $\{\mu^k\}$ and their cost functions $\{J_{\mu^k}\}$; μ^0 is "arbitrary"

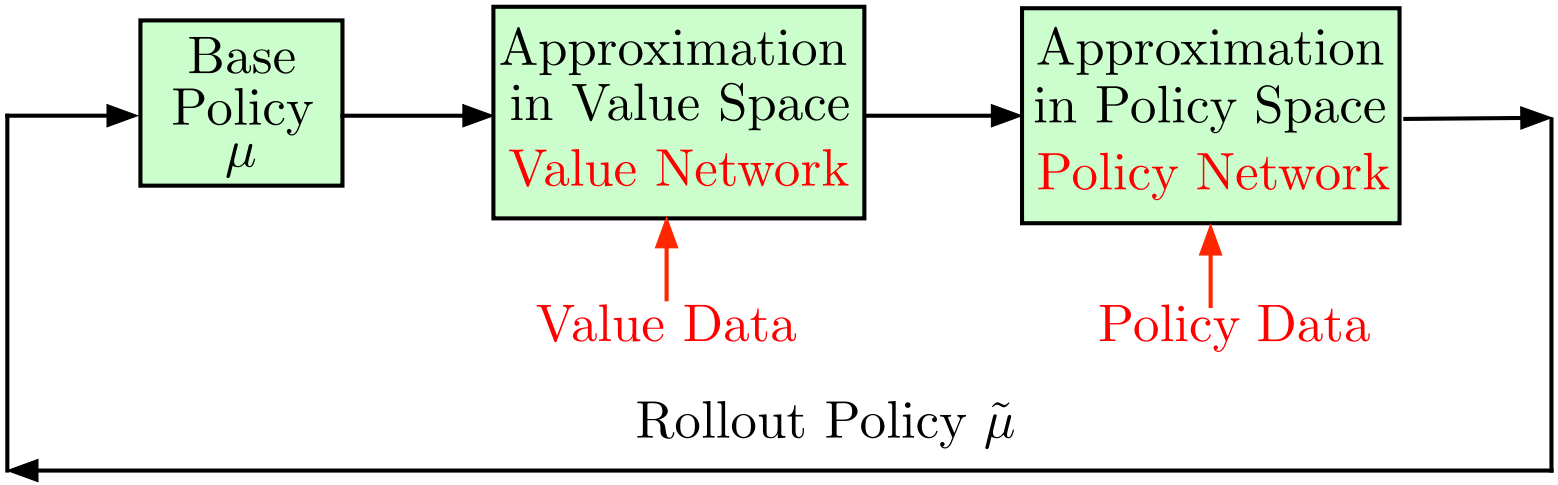
The typical iteration starts with a policy μ and generates a new policy $\tilde{\mu}$ in two steps:

- Policy evaluation step, which computes the cost function J_μ
- Policy improvement step, which computes the improved policy $\tilde{\mu}$ using the one-step lookahead minimization

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

- PI can be viewed as Newton's method for solving Bellman's Eq.
- Several PI variants: Optimistic, simplified, multiagent, Q-learning versions
- Rollout is just the first iteration of PI (a single Newton iteration - from J_{μ^0} it produces μ^1 and J_{μ^1})

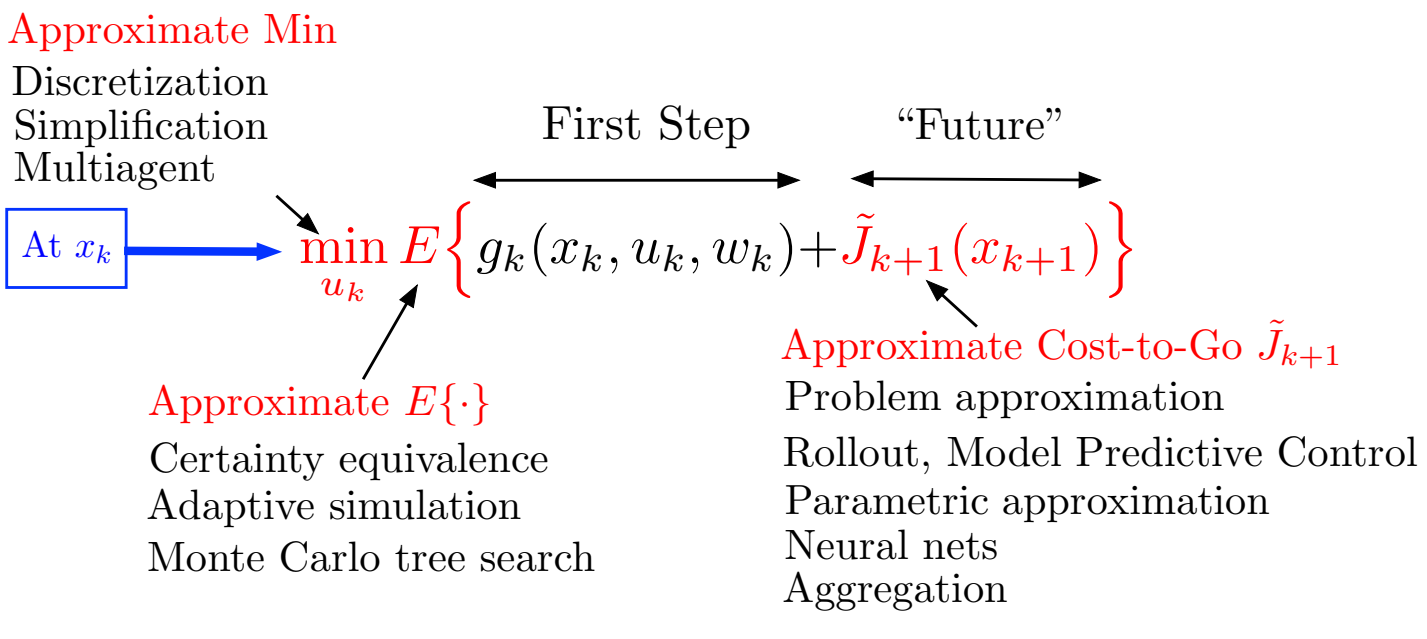
Approximate Policy Iteration and Rollout



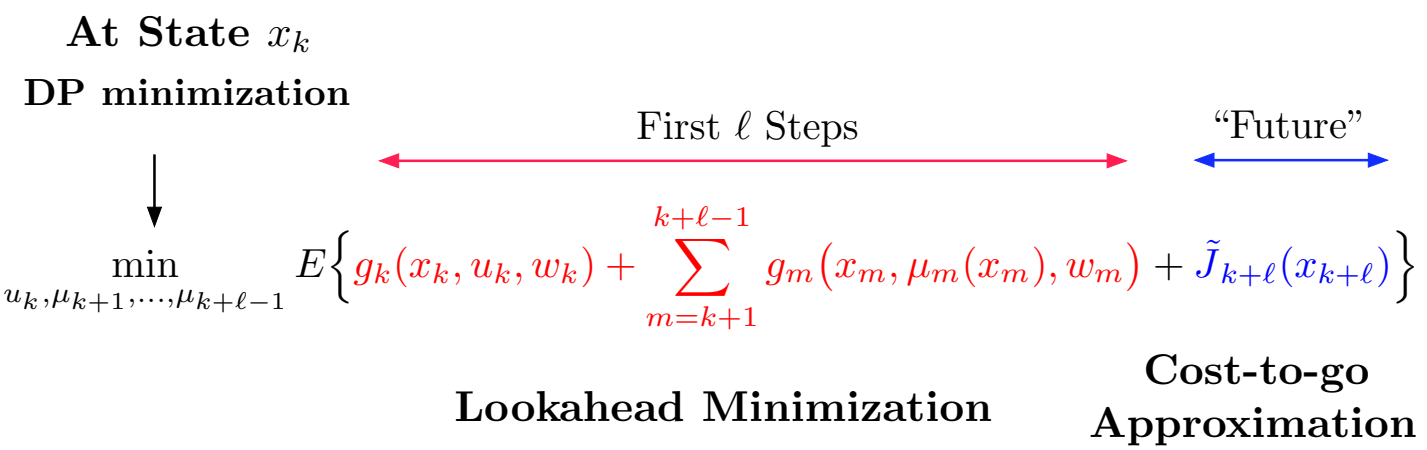
Important facts:

- PI can be implemented approximately, with a value and (perhaps) a policy network. This is off-line training
- Rollout is just the first iteration of PI
- Rollout typically improves substantially the base policy; finds the optimal policy if started sufficiently close (cf., the power of Newton's method)

Approximation in Value Space (On-Line Play After Off-Line Training)

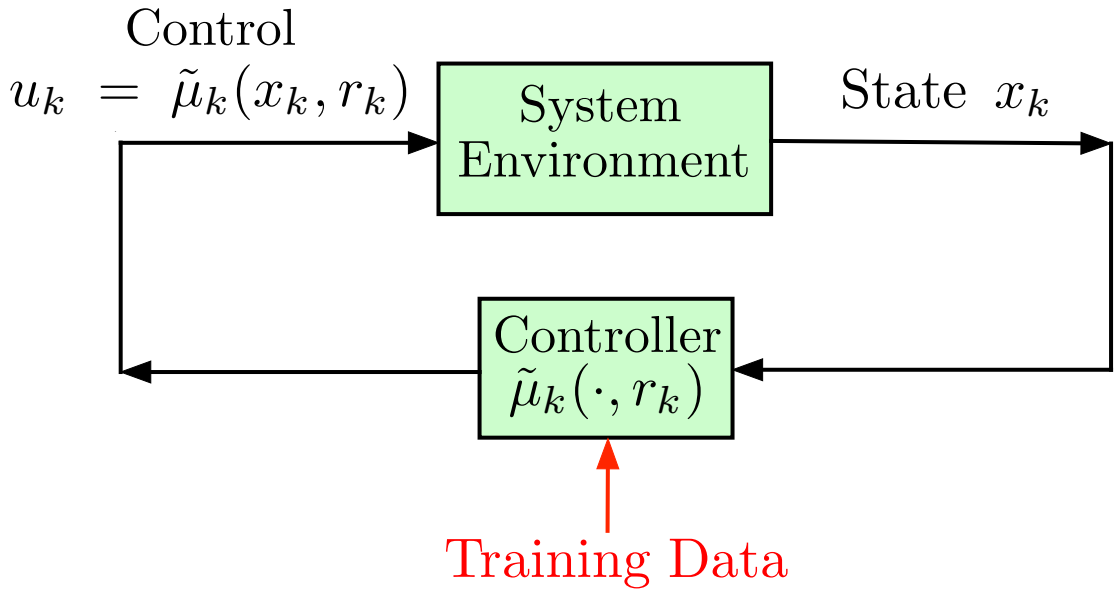


ONE-STEP LOOKAHEAD



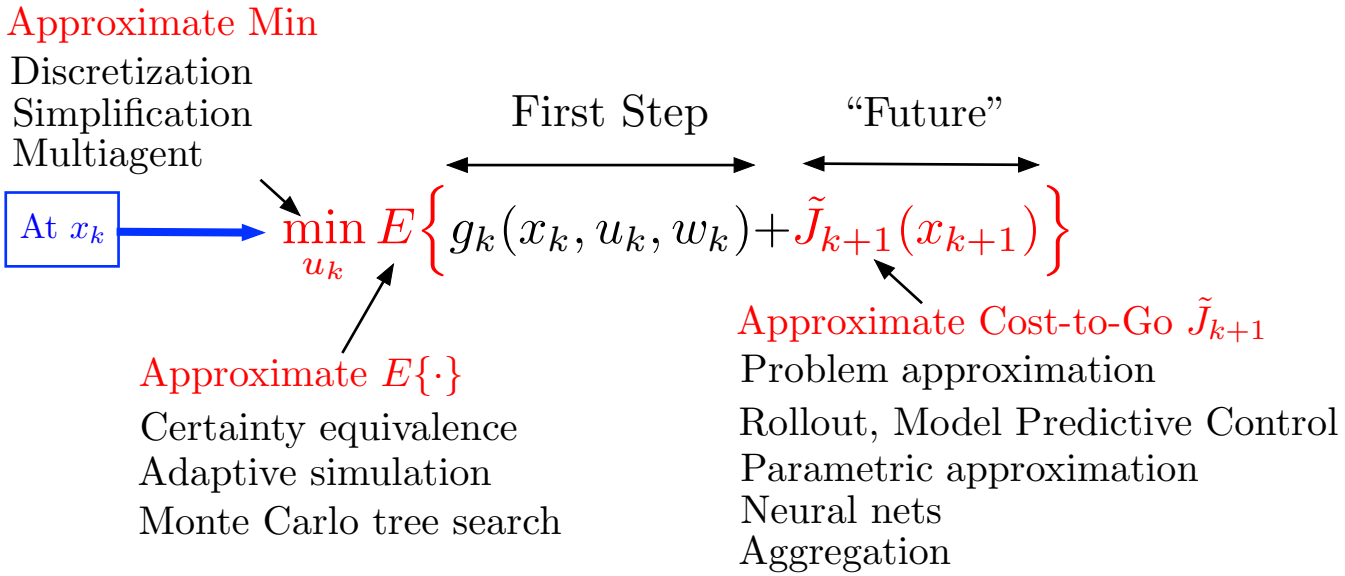
MULTISTEP LOOKAHEAD

Approximation in Policy Space (Strictly Off-Line Training)



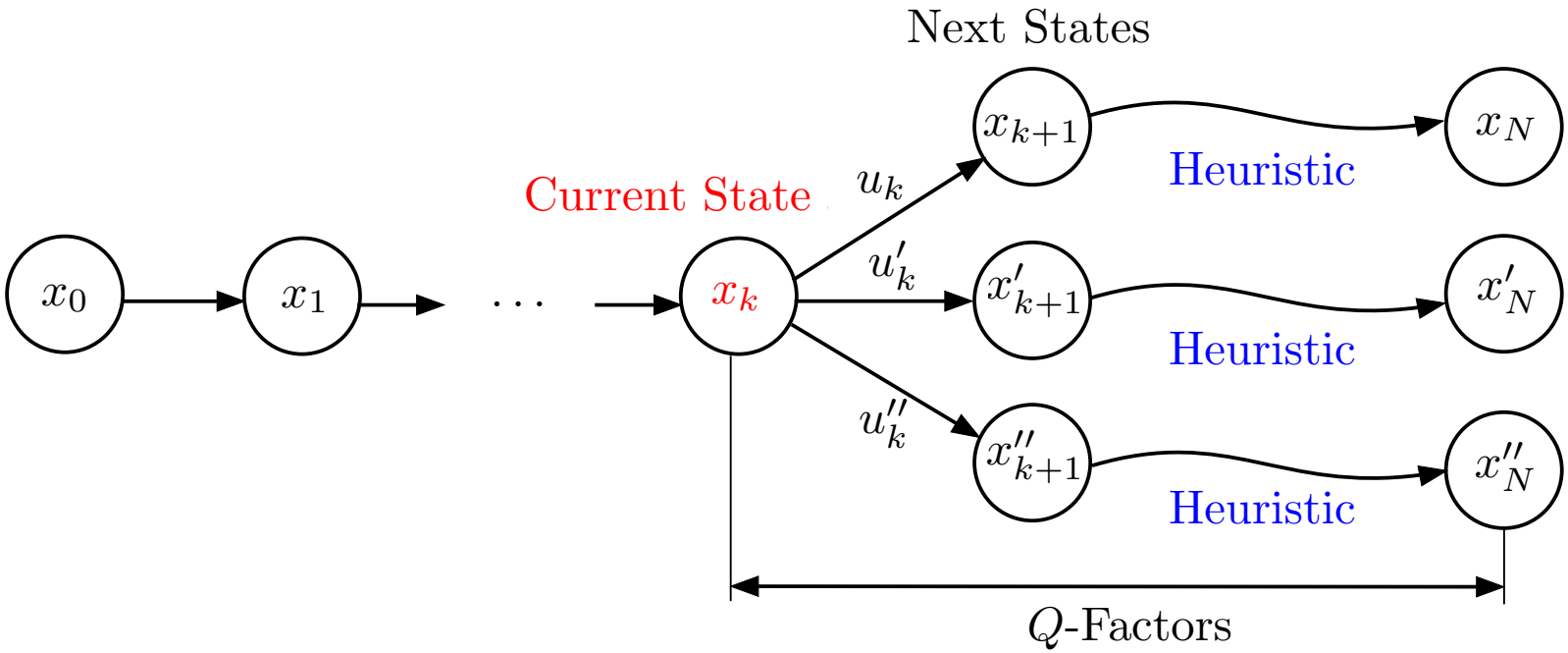
- Idea: Select the policy by optimization over a suitably restricted class of policies
- The restricted class is usually a parametric family of policies $\tilde{\mu}_k(x_k, r_k)$, $k = 0, \ldots, N-1$, of some form, where r_k is a parameter (e.g., a neural net)
- Leverages classification, gradient, and random search methodologies for training
- Important advantage once the parameters r_k are computed: The on-line computation of controls is often much faster ... at state x_k apply $u_k = \tilde{\mu}_k(x_k, r_k)$
- Important disadvantage: It does not allow for on-line replanning
- By contrast, value space approximation is well-suited for on-line replanning (even with off-line computation of $\tilde{J}_{k+1}$).

On-Line and Off-Line Implementations of Value Space Approximations



- Off-line methods (partly): All the functions $\tilde{J}_{k+1}$ are computed for every k, before the control process begins.
- Example of off-line methods: Neural network and other parametric approximations; also aggregation.
- On-line methods (mostly): 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 mostly on-line methods: Rollout and model predictive control.

General Structure of Deterministic Rollout

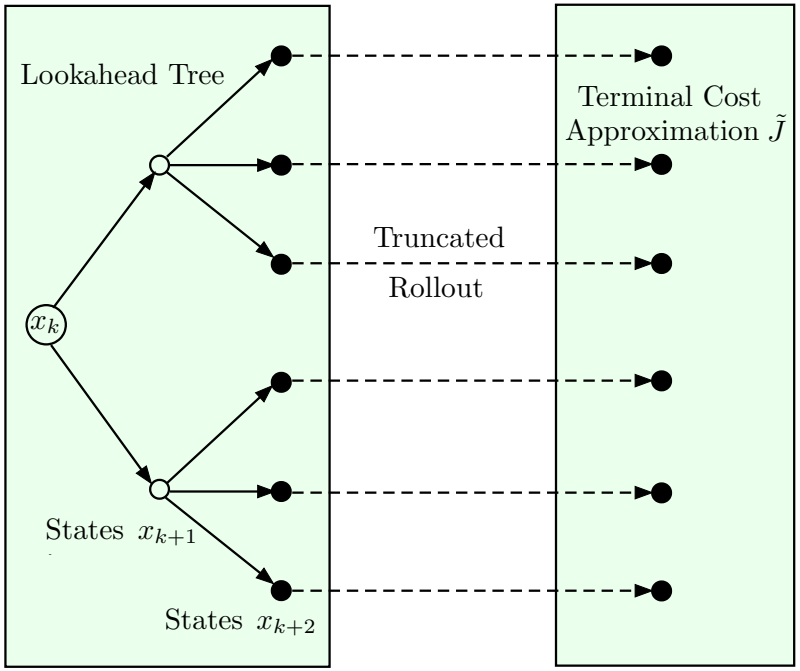


- At state x_k, for every pair (x_k, u_k), $u_k \in U_k(x_k)$, we generate a Q-factor

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

  using the base heuristic [$H_{k+1}(x_{k+1})$ is the heuristic 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 is limited by the branching factor of the lookahead tree).

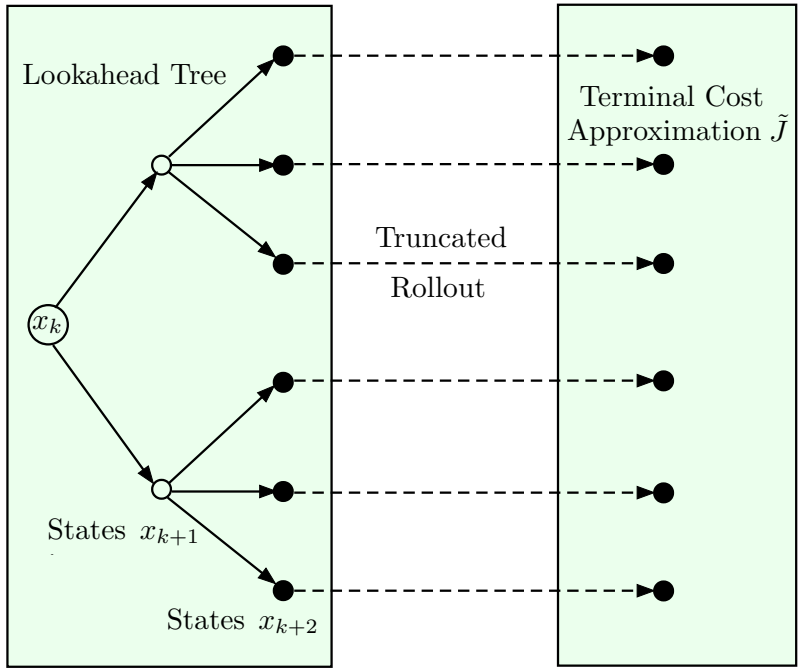
Multistep Lookahead-Truncated Rollout-Terminal Cost Approximation



Motivation and properties:

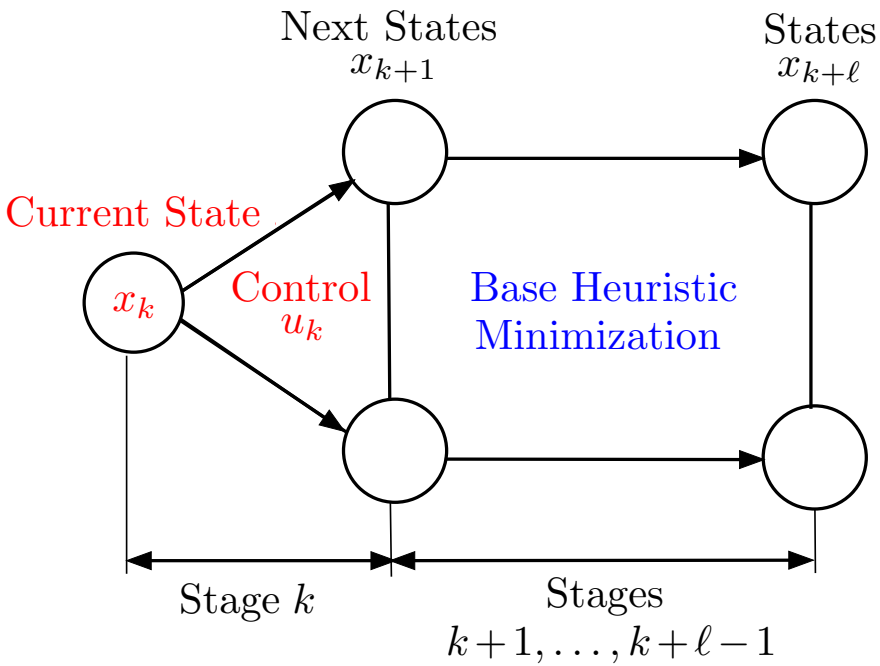
- Long rollout is costly. Also, it is not necessarily true that increasing the length of the rollout leads to improved performance.
- Terminal cost approximation allows combinations with off-line training.
- We can prove cost improvement, assuming various sequential consistency and/or sequential improvement conditions, as well as modifications (fortified rollout).
- Rollout is the most reliable and most easily implementable RL algorithm. Still some trial and error experimentation is recommended for its implementation.

Rollout Variations



- Fortified for deterministic problems; guarantees cost improvement.
- Simplified (i.e., one-step or multistep simplified minimization).
- Constrained for deterministic problems.
- Rollout with an expert; no need to know the cost function.
- With continuous control space; MPC is an important example.
- Multiagent rollout; for problems with multicomponent controls.
- Combinations with MCTS; for stochastic problems, to save simulation time.
- Combinations with approximations in value and policy space.

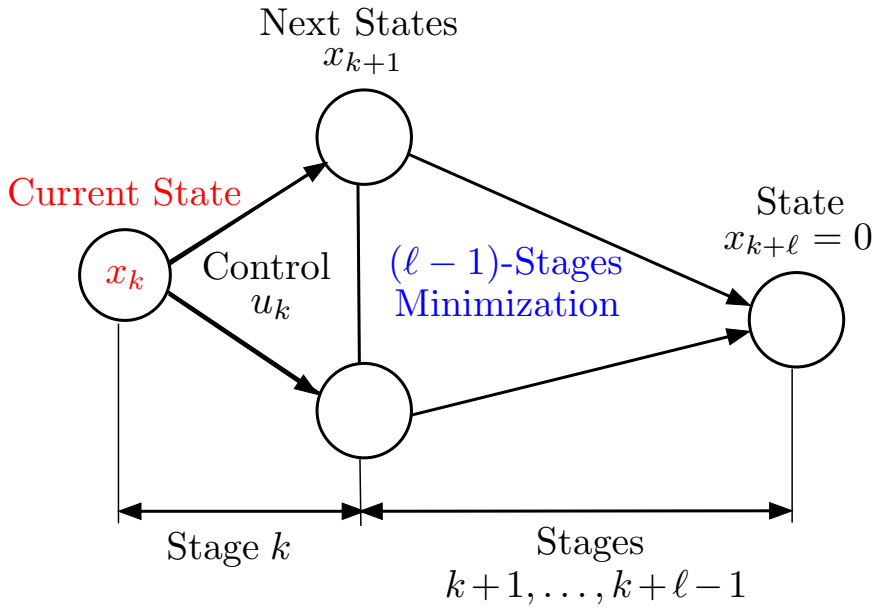
On-Line Rollout for Deterministic Infinite-Spaces Problems



When the control space is infinite, rollout needs a different implementation

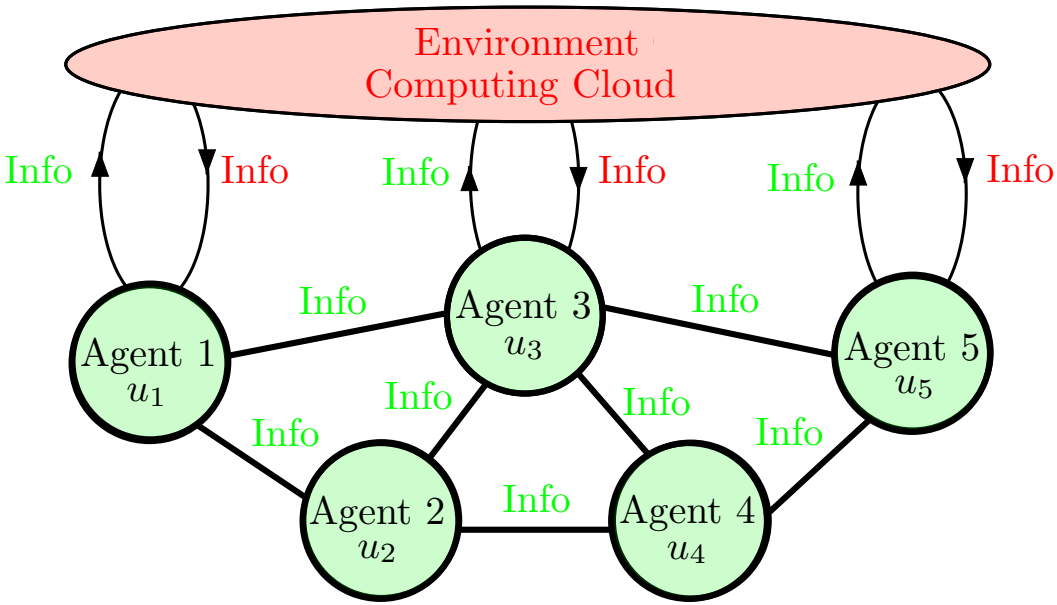
- One possibility is discretization of $U_k(x_k)$; but then excessive number of Q-factors.
- The major alternative is to use optimization heuristics.
- Seemlessly combine the kth stage minimization and the optimization heuristic into a single ℓ-stage deterministic optimization.
- Can solve it by nonlinear programming/optimal control methods (e.g., quadratic programming, gradient-based).
- This is the idea underlying model predictive control (MPC).

Model Predictive Control for Deterministic Control-to-the-Origin Problems



- System: $x_{k+1} = f_k(x_k, u_k)$.
- Cost per stage: $g_k(x_k, u_k) \geq 0$, the origin 0 is cost-free and absorbing.
- State and control constraints: $x_k \in X_k$, $u_k \in U_k(x_k)$ for all k.
- At x_k solve an ℓ-step lookahead version of the problem, requiring $x_{k+\ell} = 0$ while satisfying the state and control constraints.
- If $\{\tilde{u}_k, \ldots, \tilde{u}_{k+\ell-1}\}$ is the control sequence so obtained, apply $\tilde{u}_k$.
- Variants: Terminal cost approximation, treatment of constraints, simplified versions.

Multiagent Problems - A Very Old (1960s) and Well-Researched Field

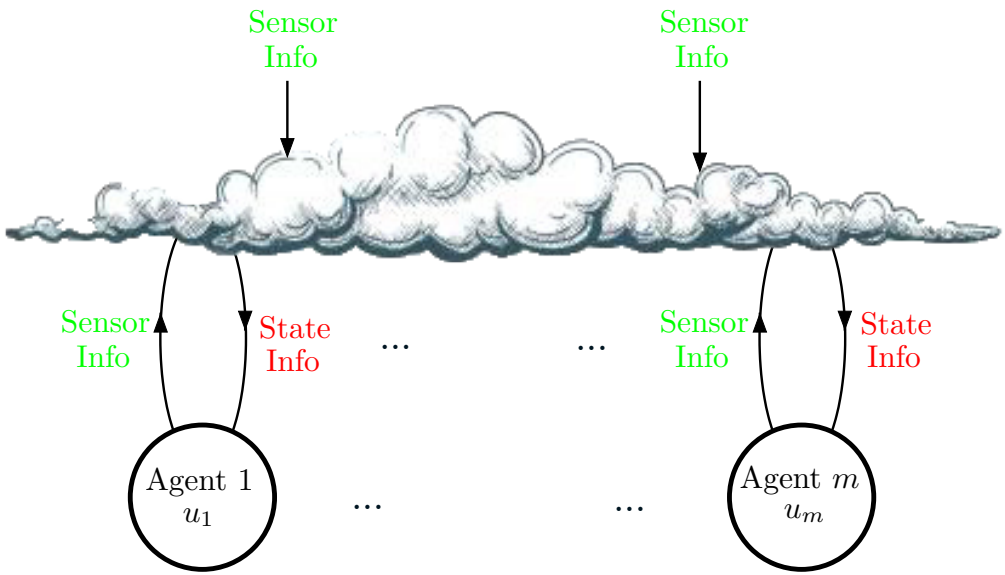


- Multiple agents collecting and sharing information selectively with each other and with an environment/computing cloud
- Agent i applies decision u_i sequentially in discrete time based on info received

The major mathematical distinction between structures

- The classical information pattern: Agents are fully cooperative, fully sharing and never forgetting information. Can be treated by Dynamic Programming (DP)
- The nonclassical information pattern: Agents are partially sharing information, and may be antagonistic. HARD because it cannot be treated by DP

Our Starting Point: A Classical Information Pattern



The agents have exact state info, and choose their controls as functions of the state

Model: Stochastic DP (finite or infinite horizon) with state x and control u

- Decision/control has m components $u = (u_1, \ldots, u_m)$ corresponding to m "agents"
- "Agents" is just a metaphor - the important math structure is $u = (u_1, \ldots, u_m)$
- We apply approximate DP/rollout ideas, aiming at faster computation in order to:
  - Deal with the exponential size of the search/control space by one-agent-at-a-time computations
  - Use signaling to estimate info that we don't know
  - Signaling is precomputed (using neural nets or other estimators)

Multiagent Rollout/Policy Improvement When $u = (u_1, \ldots, u_m)$

To simplify notation, consider infinite horizon setting. The standard rollout operation is

$$\big(\tilde{\mu}_1(x), \ldots, \tilde{\mu}_m(x)\big) \in \arg \min_{(u_1,\ldots,u_m)} E\Big\{g(x, u_1, \ldots, u_m, w) + \alpha J_\mu\big(f(x, u_1, \ldots, u_m, w)\big)\Big\};$$

the search space is exponential in m (μ is the base policy, seq. consistency holds)

Multiagent rollout (a form of simplified rollout; implies cost improvement)

Perform a sequence of m successive minimizations, one-agent-at-a-time

$$\tilde{\mu}_1(x) \in \arg \min_{u_1} E\Big\{g(x, u_1, \mu_2(x), \ldots, \mu_m(x), w) + \alpha J_\mu\big(f(x, u_1, \mu_2(x), \ldots, \mu_m(x), w)\big)\Big\}$$

$$\tilde{\mu}_2(x) \in \arg \min_{u_2} E\Big\{g(x, \tilde{\mu}_1(x), u_2, \mu_3(x) \ldots, \mu_m(x), w) + \alpha J_\mu\big(f(x, \tilde{\mu}_1(x), u_2, \mu_3(x), \ldots, \mu_m(x), w)\big)\Big\}$$

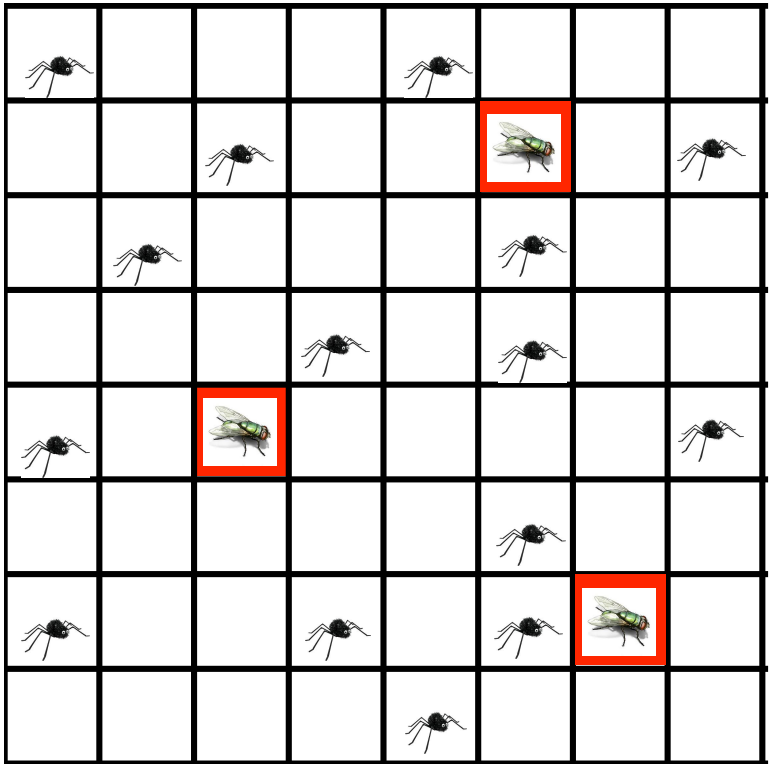
$$\ldots \qquad \ldots \qquad \ldots \qquad \ldots$$

$$\tilde{\mu}_m(x) \in \arg \min_{u_m} E\Big\{g(x, \tilde{\mu}_1(x), \tilde{\mu}_2(x), \ldots, \tilde{\mu}_{m-1}(x), u_m, w) + \alpha J_\mu\big(f(x, \tilde{\mu}_1(x), \tilde{\mu}_2(x), \ldots, \tilde{\mu}_{m-1}(x), u_m$$

- Has a search space with size that is linear in m; ENORMOUS SPEEDUP!

Survey reference: Bertsekas, D., "Multiagent Reinforcement Learning: Rollout and Policy Iteration," IEEE/CAA J. of Aut. Sinica, 2021 (and earlier papers quoted there).

Spiders-and-Flies Example
(e.g., Delivery, Maintenance, Search-and-Rescue, Firefighting)

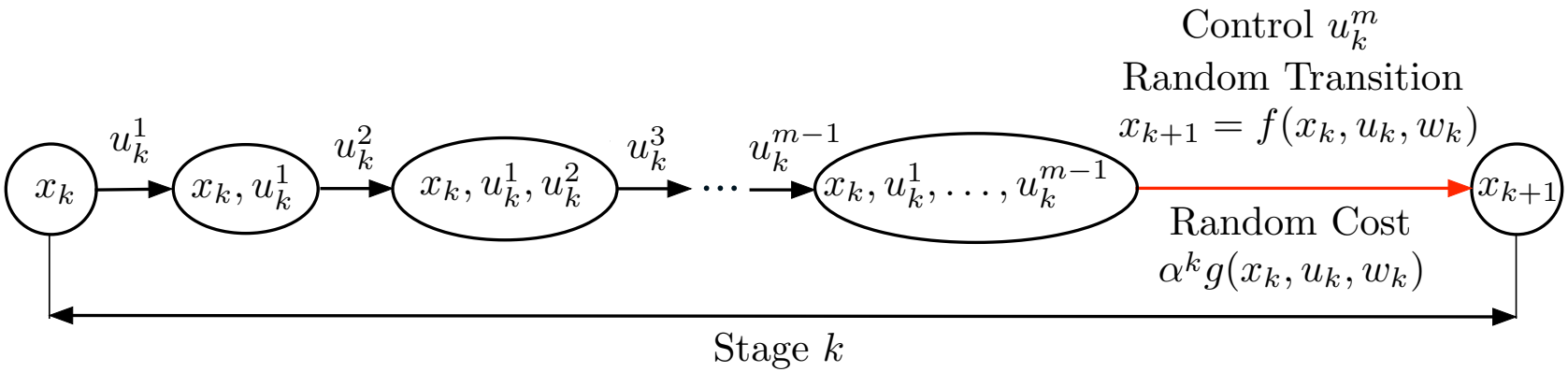


15 spiders move in 4 directions with perfect vision

3 blind flies move randomly

- Objective is to catch the flies in minimum time
- At each stage we must select one out of $\approx 5^{15}$ joint move choices
- Multiagent rollout reduces this to $5 \cdot 15 = 75$ choices (while maintaining cost improvement); applies a sequence of one-spider-at-a-time moves
- With “precomputed signaling/coordination", the 15 spiders will choose moves in parallel (extra speedup factor of up to 15)

Justification of Cost Improvement through Reformulation: Trading off Control and State Complexity (NDP Book, 1996)



An equivalent reformulation - "Unfolding" the control action

- The control space is simplified at the expense of $m-1$ additional layers of states, and corresponding $m-1$ cost functions

$$J^1(x, u_1), J^2(x, u_1, u_2), \ldots, J^{m-1}(x, u_1, \ldots, u_{m-1})$$

- Multiagent rollout is just standard rollout for the reformulated problem
- The increase in size of the state space does not adversely affect rollout (only one state per stage is looked at during on-line play)
- Complexity reduction: The one-step lookahead branching factor is reduced from n^m to $n \cdot m$, where n is the number of possible choices for each component u_i

Approximation Architectures

- 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).
- We adjust r to change $\tilde{J}$ and "match" the cost function approximated.
- Training the architecture: The algorithm to choose r (typically use data/regression).
- Architectures are linear or nonlinear, depending on whether $\tilde{J}(x, r)$ is linear or nonlinear in r.
- Architectures are feature-based if they depend on x via a feature vector $\phi(x)$,

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

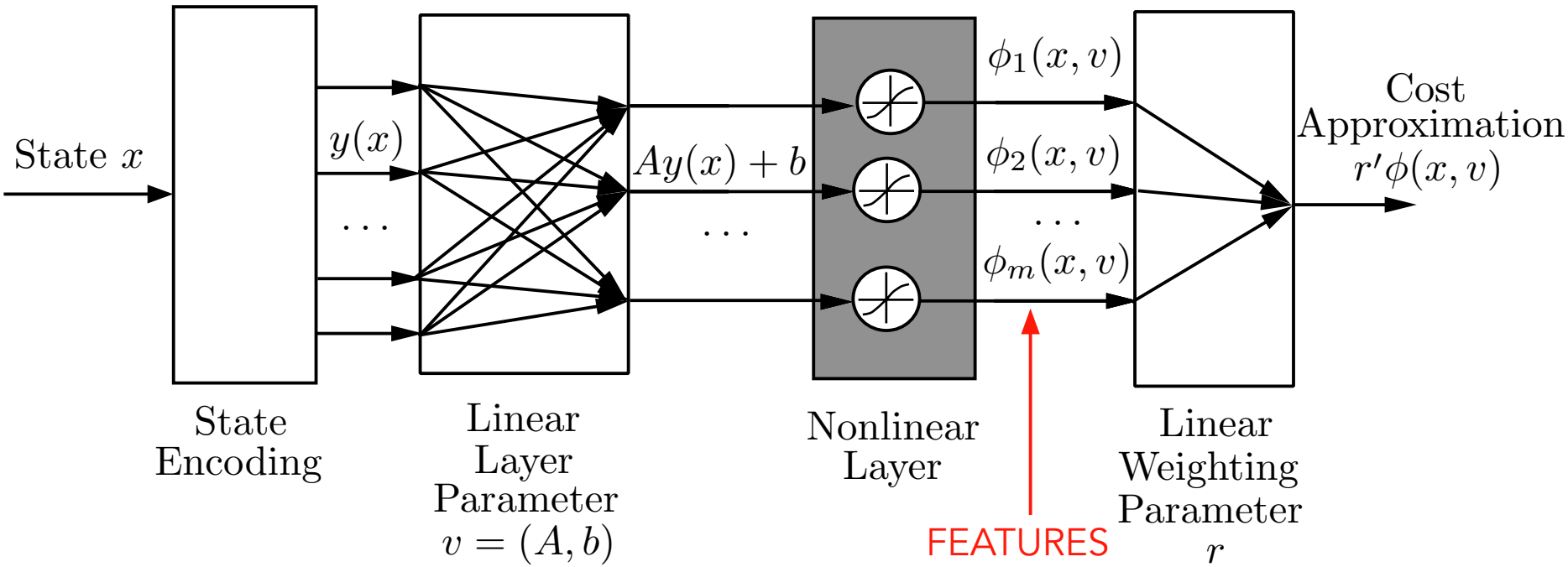
where $\hat{J}$ is some function. Idea: Features capture dominant nonlinearities.

- A linear feature-based architecture:

$$\tilde{J}(x, r) = \sum_{\ell=1}^{m} r_\ell \phi_\ell(x),$$

where r_ℓ and $\phi_\ell(x)$ are the ℓth components of r and $\phi(x)$.

Neural Nets: An Architecture that does not Require Knowledge of Features



- Can be used when problem-specific handcrafted features and linear feature-based architectures are inadequate.
- Tricky training issues by incremental gradient (backpropagation) methods.
- Deep neural nets have proved useful in important contexts.
- There are other nonlinear architectures (e.g., radial basis functions) that we have not covered.

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 $\tilde{J}_{k+1}$, 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), $s = 1, \ldots, q$.

Typical approach: Train by least squares/regression and possibly using a neural net

We minimize over r_k

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

(plus a regularization term).

Sequential Q-Factor Approximation

- Consider sequential DP approximation of Q-factor parametric approximations

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

- Note: $E\{\min(\ldots)\}$ can be sampled; $\min(E\{\ldots\})$ cannot be sampled.
- 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 $E\{\cdot\}$.]
- 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.

Convergence of VI

Given any initial conditions $J_0(1), \ldots, J_0(n)$, the sequence $\{J_k(i)\}$ generated by VI

$$J_{k+1}(i) = \min_{u \in U(i)} \sum_{j=1}^{n} p_{ij}(u)\big(g(i,u,j) + \alpha J_k(j)\big), \qquad i = 1, \ldots, n,$$

converges to $J^*(i)$ for each i.

Bellman's equation

The optimal cost function $J^* = \big(J^*(1), \ldots, J^*(n)\big)$ satisfies the equation

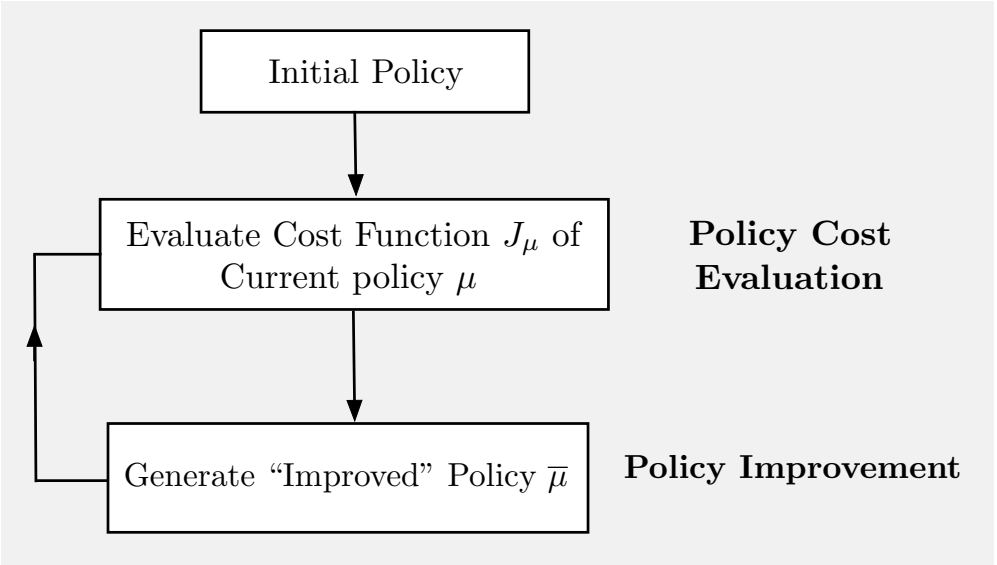
$$J^*(i) = \min_{u \in U(i)} \sum_{j=1}^{n} p_{ij}(u)\big(g(i,u,j) + \alpha J^*(j)\big), \qquad i = 1, \ldots, n,$$

and is the unique solution of this equation.

Optimality condition

A stationary policy μ is optimal if and only if for every state i, $\mu(i)$ attains the minimum in the Bellman equation.

Policy Iteration (PI) Algorithm



Given the current policy μ^k, a PI consists of two phases:

- Policy evaluation computes $J_{\mu^k}(i)$, $i = 1, \ldots, n$, as the solution of the (linear) Bellman equation system

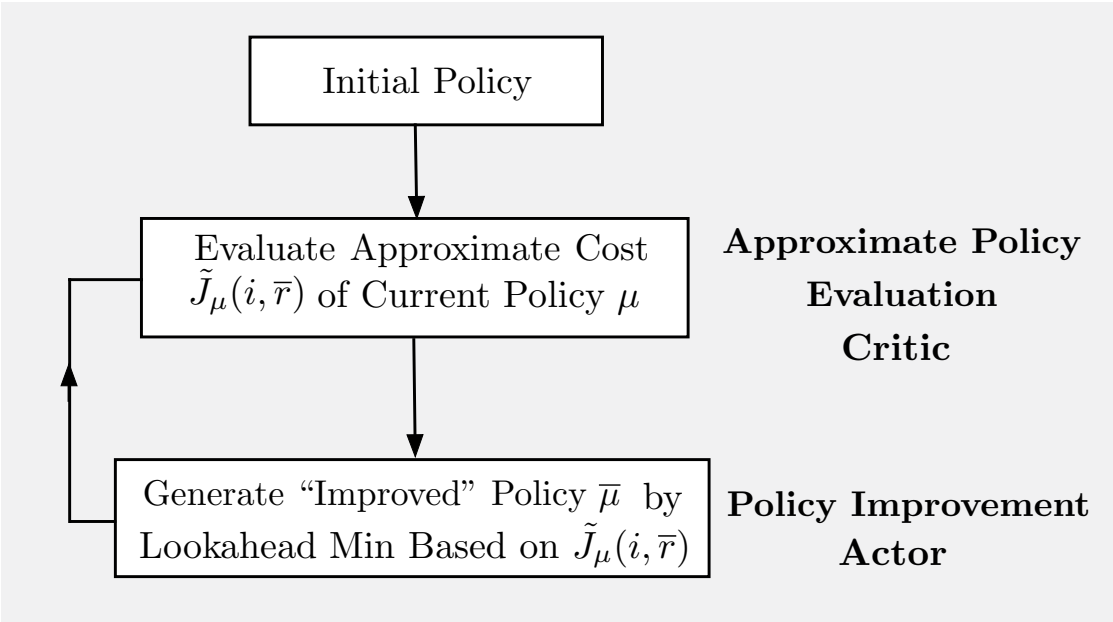
$$J_{\mu^k}(i) = \sum_{j=1}^{n} p_{ij}\big(\mu^k(i)\big)\Big(g\big(i, \mu^k(i), j\big) + \alpha J_{\mu^k}(j)\Big), \quad i = 1, \ldots, n$$

- Policy improvement then computes a new policy μ^{k+1} as

$$\mu^{k+1}(i) \in \arg\min_{u \in U(i)} \sum_{j=1}^{n} p_{ij}(u)\big(g(i, u, j) + \alpha J_{\mu^k}(j)\big), \quad i = 1, \ldots, n$$

- Optimistic and multistep lookahead versions.

Parametric Approximation and Actor-Critic Schemes



Introduce a differentiable parametric architecture $\tilde{J}_\mu(i,r)$ for policy evaluation

- Example architectures: A linear featured-based or a neural net.
- Example of approximate policy evaluation: Generate state-cost pairs (i^s, β^s), where β^s is a sample cost corresponding to i^s. Use least squares/regression:

$$\overline{r} \in \arg\min_r \sum_{s=1}^{q} \left(\tilde{J}_\mu(i^s, r) - \beta^s\right)^2$$

- β^s is generated by simulating an N-step trajectory starting at i^s, using μ, and adding a terminal cost approximation $\alpha^N \hat{J}(i_N)$.
- Alternative approximate policy evaluation methods: TD(λ), LSTD(λ), LSPE(λ)

- The training problem

$$\overline{r} \in \arg\min_{r} \sum_{s=1}^{q} \left(\tilde{J}_{\mu}(i^s, r) - \beta^s\right)^2$$

  is well-suited for incremental gradient:

$$r^{k+1} = r^k - \gamma^k \nabla \tilde{J}_{\mu}(i^{s_k}, r^k)\left(\tilde{J}_{\mu}(i^{s_k}, r^k) - \beta^{s_k}\right)$$

  where (i^{s_k}, β^{s_k}) is the state-cost sample pair that is used at the kth iteration.
- Trajectory reuse: Given a long trajectory $(i_0, i_1, \ldots, i_N)$, we can obtain cost samples for all the states $i_0, i_1, i_2, \ldots$, by using the tail portions of the trajectory.
- Exploration: When evaluating μ with trajectory reuse, we generate many cost samples that start from states frequently visited by μ. Then the cost of underrepresented states may be estimated inaccurately, causing potentially serious errors in the calculation of the improved policy $\overline{\mu}$.
- Bias-variance tradeoff: As the trajectory length N increases, the cost samples β^s become more accurate but also more "noisy."
- Error bounds quantify qualitative behavior; e.g., convergence to within an "error zone."

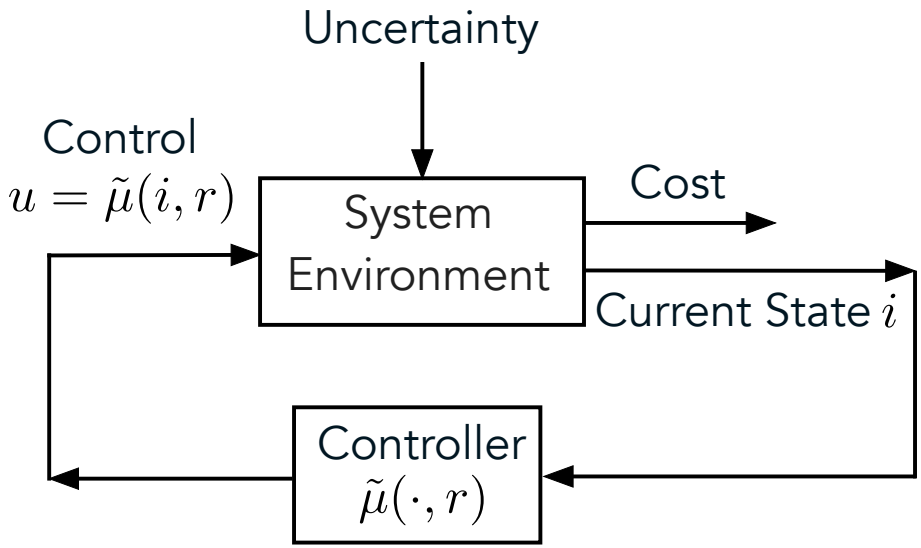
General Framework for Approximation in Policy Space

- Parametrize stationary policies with a parameter vector r; denote them by $\tilde{\mu}(r)$, with components $\tilde{\mu}(i, r)$, $i = 1, \ldots, n$. Each r defines a policy.
- The parametrization may be problem-specific, or feature-based, or may involve a neural network.
- The idea is to optimize some measure of performance with respect to r.

Five contexts where approximation in policy space is either essential or is helpful

- Problems with natural policy parametrizations (like supply chain problems)
- Problems with natural value parametrizations, where a good policy training method works well (like the tetris problem).
- Approximation in policy space on top of approximation in value space.
- Learning from a software or human expert.
- Unconventional information structures, e.g., multiagent systems with local information (not shared with other agents) - Conventional DP breaks down.

Approximation in Policy Space by Optimization-Based Training



Training by Cost Optimization

- Each r defines a stationary policy $\tilde{\mu}(r)$, with components $\tilde{\mu}(i, r)$, $i = 1, \ldots, n$.
- Determine r through the minimization

$$\min_r J_{\tilde{\mu}(r)}(i_0)$$

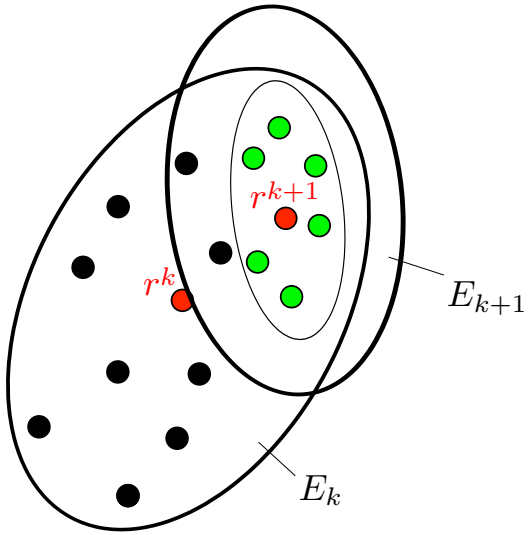
where $J_{\tilde{\mu}(r)}(i_0)$ is the cost of the policy $\tilde{\mu}(r)$ starting from initial state i_0.

- More generally, determine r through the minimization

$$\min_r E\{J_{\tilde{\mu}(r)}(i_0)\}$$

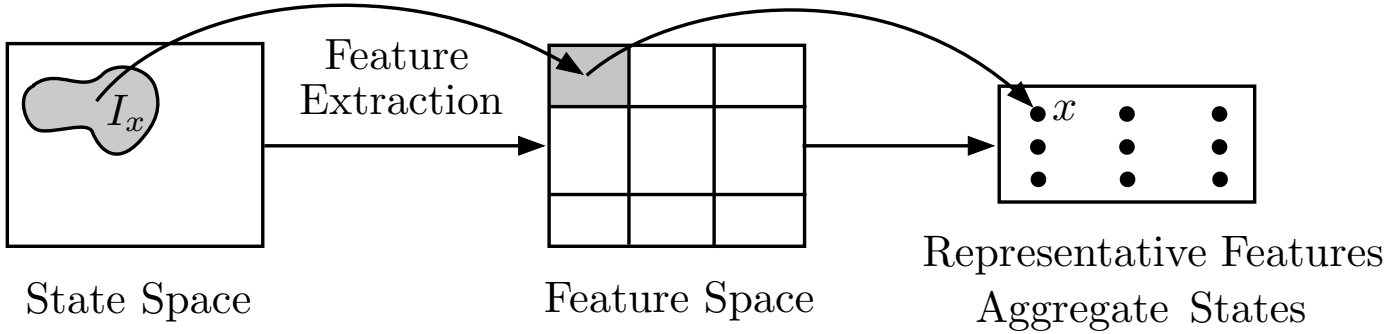
where the $E\{\cdot\}$ is with respect to a suitable probability distribution of i_0.

Training by Random Search - Cross-Entropy Method

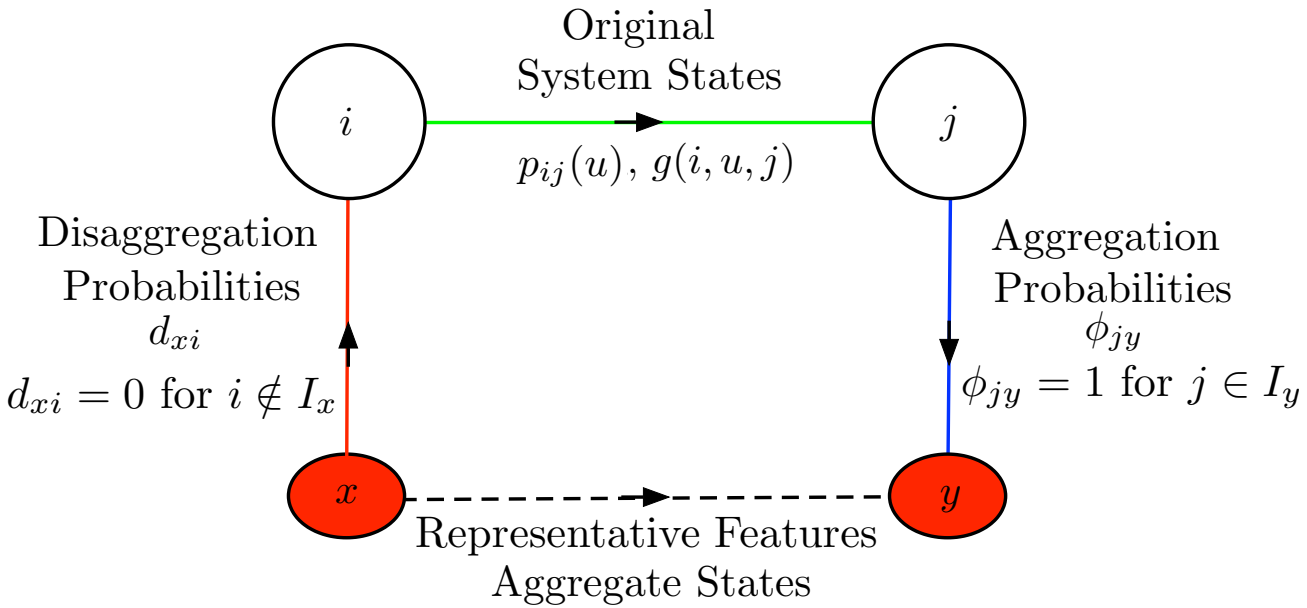


- At the current iterate r^k, construct an ellipsoid E_k centered at r^k.
- Generate a number of random samples within E_k. “Accept" a subset of the samples that have “low" cost.
- Let r^{k+1} be the sample “mean" of the accepted samples.
- Construct a sample “covariance" matrix of the accepted samples, 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". Naturally model-free.

Feature-Based Aggregation Framework



Representative feature formation



Transition diagram for the aggregate problem

Some of the Major Points Relating to Aggregation

- It aims to approximate J^*, not J_μ of some policy μ.
- It can yield an arbitrarily close approximation to J^*, with sufficient number of aggregate states.
- Distinction between representative features schemes and their simpler special case, representative states schemes.
- Simulation-based VI and PI methods for solving the aggregate problem.
- Spatio-temporal aggregation: Solve a simpler aggregate problem involving “compression" in space and time.

Concluding Remarks

Some words of caution

- There are challenging implementation issues in all approaches, and no fool-proof methods.
- Problem approximation and hand-crafted feature selection require domain-specific knowledge.
- Training algorithms are not as reliable as you might think by reading the literature.
- Approximate PI involves oscillations and faces challenging exploration issues.
- Recognizing success or failure can be a challenge!
- 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 the positive side

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

Some Words of Relevance

Some old quotes ...

- The book of the universe is written in the language of mathematics. Gallileo
- Learning without thought is labor lost; thought without learning is perilous. Confucius
  (In the language of Confucius' day: learning $\approx$ obtaining knowledge; thought $\approx$ ideas on how to do things)
- Many arts have been discovered through practice, empirically; for experience makes our life proceed deliberately, but inexperience unpredictably. Plato
- White cat or black cat it is a good cat if it catches mice. Deng Xiaoping

... and some more recent ones

- Machine learning is the new electricity. Andrew Ng
  (Electricity changed how the world operated. It upended transportation, manufacturing, agriculture and health care. AI is poised to have a similar impact.)
- Machine learning is the new alchemy. Ali Rahimi and Ben Recht
  (We do not know why some algorithms work and others don't, nor do we have rigorous criteria for choosing one architecture over another ...)

Thank you and good luck!