Topics in Reinforcement Learning: Lessons from AlphaZero for (Sub)Optimal Control and Discrete Optimization

Arizona State University
Course CSE 691, Spring 2022

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

Dimitri P. Bertsekas
dbertsek@asu.edu

Lecture 10
Approximate Policy Iteration and Q-Learning: Centralized and Distributed Implementation Aspects

General Remarks on Approximate Policy Iteration and Variations

- For large state space problems, policy iteration can only be implemented approximately
- It requires approximation architectures (a value and/or a policy network) to go from one iteration to the next
- It is thus an off-line training method
- It produces a sequence of policies. The last policy can be used for on line play directly
- Alternatively (and much more effectively) it can produce a policy and/or a cost function approximation that can be used for on line play through an approximation in value space/truncated rollout scheme.
- This is the approach used in AlphaZero and TD-Gammon (relation to Newton's method)
- Variants of the method include:
  - Optimistic (use very approximate policy evaluation/neural network training - just a few samples and gradient iterations)
  - Q-learning versions (train Q-factors)
  - Distributed and multiagent versions

Outline

Main Results - Discounted Problems

Infinite horizon discounted problems: States i, controls $u \in U(i)$, transition probs $p_{ij}(u)$, cost per stage $g(i,u,j)$, discount factor $\alpha < 1$

Bellman's equation for optimal cost J^* and policy cost J_μ

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

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

Value iteration convergence for optimal cost and policy cost

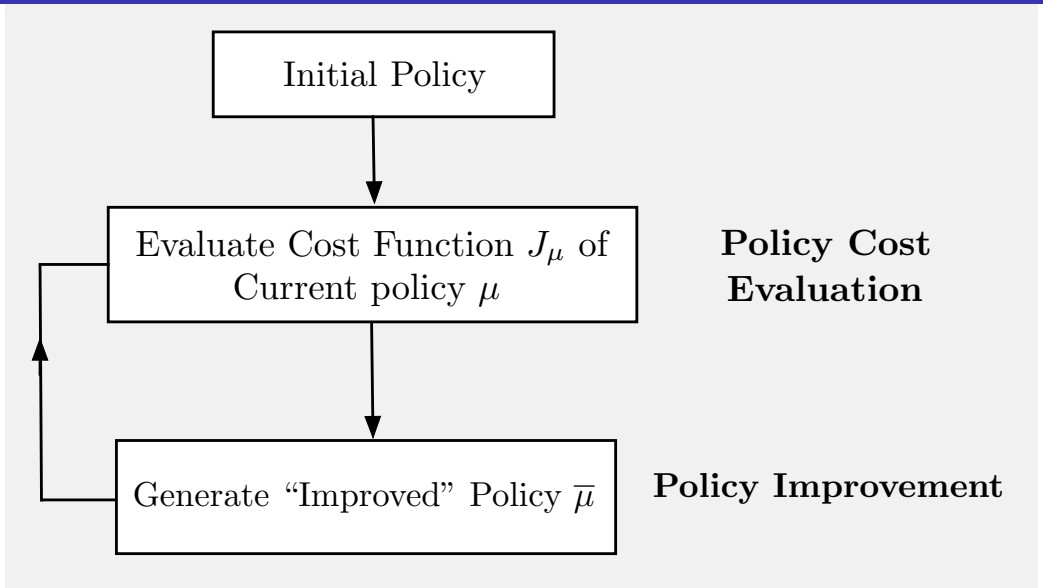
$$J_{k+1}(i) = \min_{u \in U(i)} \sum_{j=1}^{n} p_{ij}(u)\big(g(i,u,j) + \alpha J_k(j)\big), \qquad J_k \to J^*$$

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

Optimality condition

μ is optimal if and only if it attains the min in Bellman's equation

Policy Iteration (PI) Algorithm: Generates a Sequence of Policies $\{\mu^k\}$



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

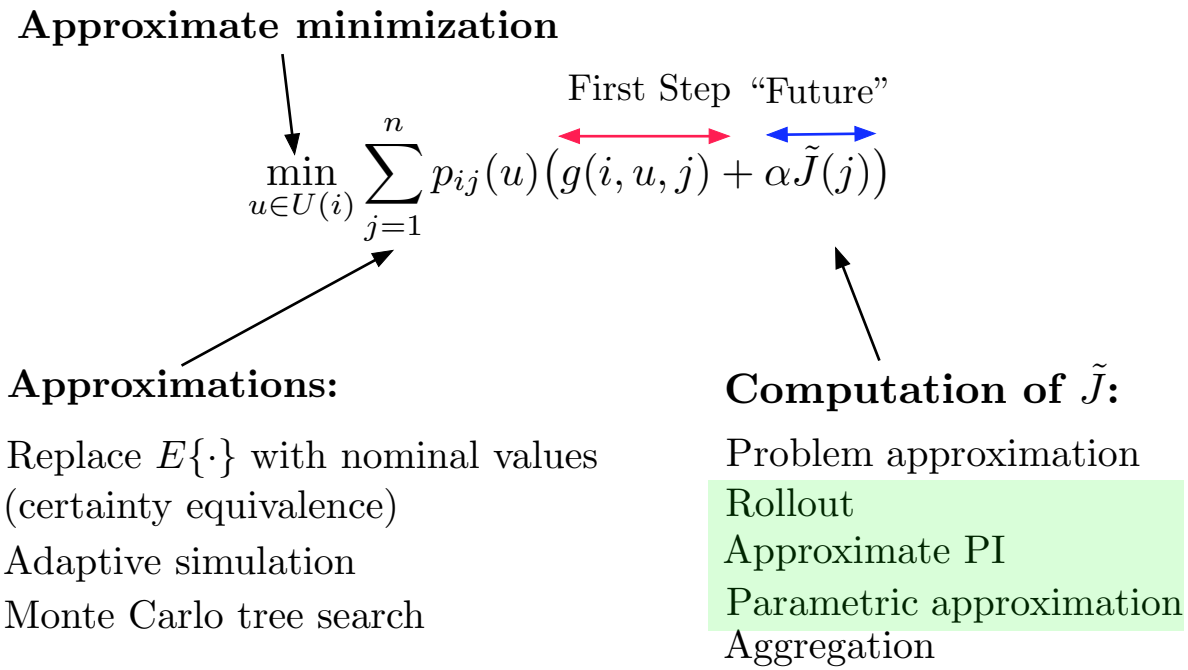
$$J_{\mu^k}(i) = \sum_{j=1}^{n} p_{ij}\big(\mu^k(i)\big)\Big(g\big(i, \mu^k(i), j\big) + \alpha J_{\mu^k}(j)\Big), \quad i = 1, \ldots, n$$

- 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 PI: Like standard PI, but policy evaluation uses a finite number of VI.

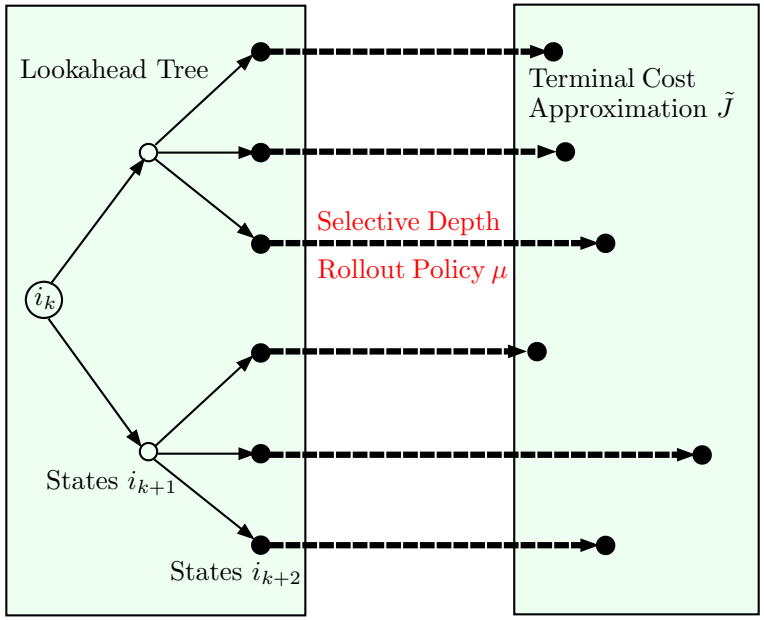
Approximation in Value Space for Infinite Horizon Problems



We focus on rollout, and particularly on approximate PI schemes, which operate as follows:

- Several policies $\mu^0, \mu^1, \ldots, \mu^m$ are generated, starting with an initial policy μ^0.
- Each policy μ^k is evaluated approximately, with a cost function $\tilde{J}_{\mu^k}$, often with the use of a parametric approximation/neural network approach.
- The next policy μ^{k+1} is generated by policy improvement based on $\tilde{J}_{\mu^k}$.
- The approximate evaluation $\tilde{J}_{\mu^m}$ of the last policy in the sequence is used as the lookahead approximation $\tilde{J}$ in a one-step or multistep lookahead minimization.

Rollout and Truncated Rollout

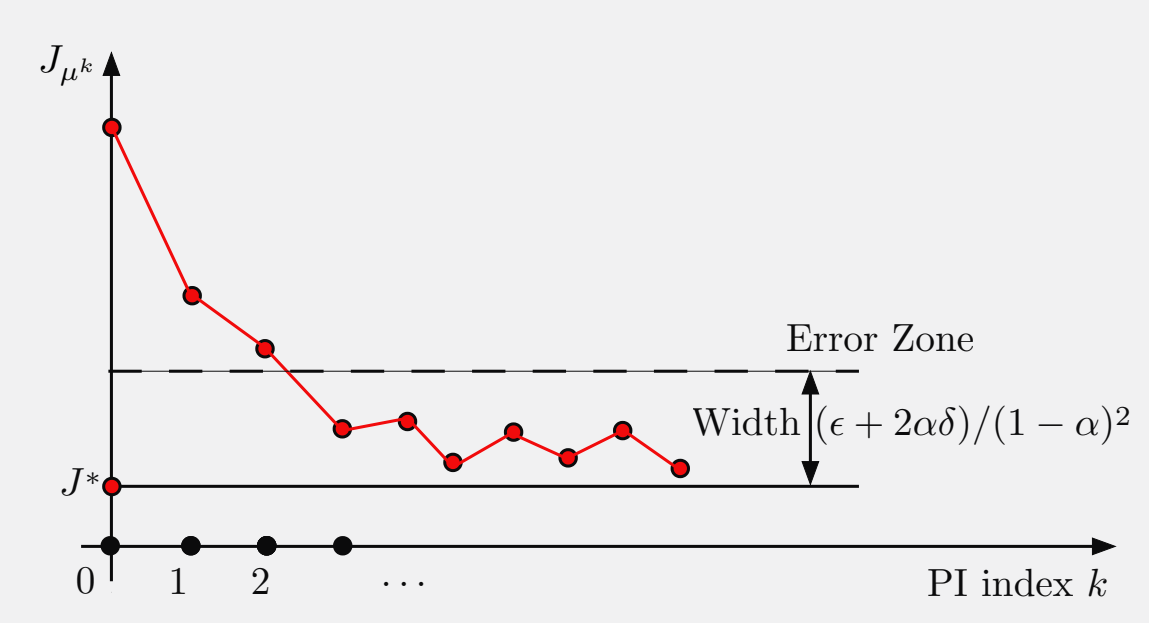


ℓ-step lookahead, truncated rollout, terminal cost approximation

- ℓ-step lookahead, then rollout with policy μ for a limited number of steps, and finally a terminal cost approximation $\tilde{J}$.
- Without terminal approximation, this is a single PI combined with multistep lookahead.
- With a terminal approximation, this is a single optimistic PI combined with multistep lookahead.

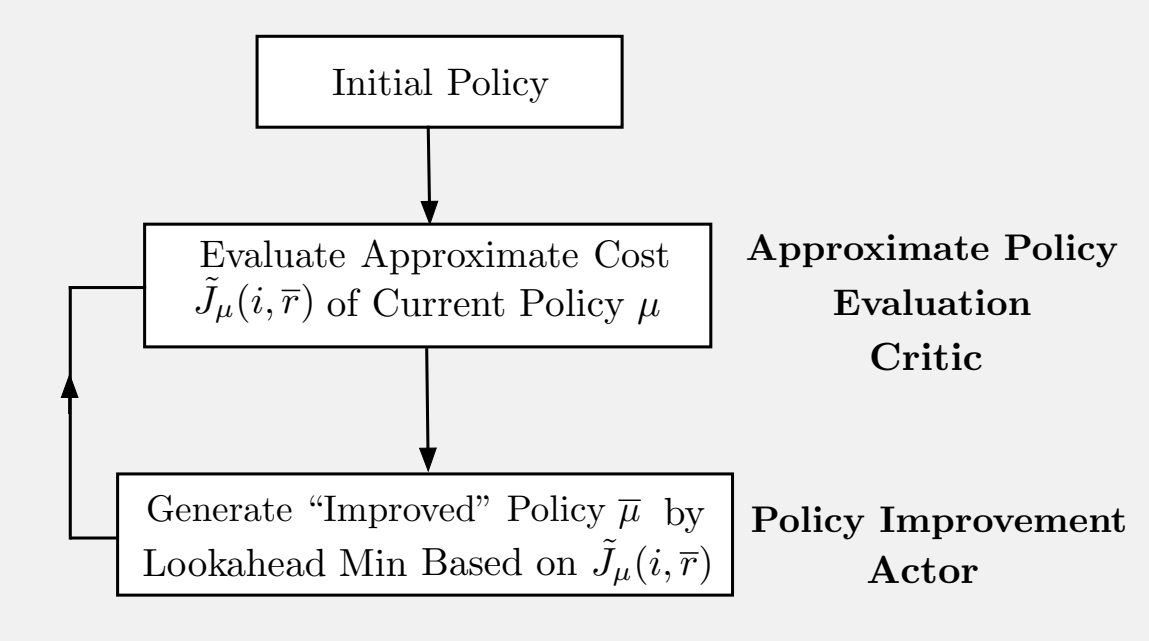
Performance bounds: They improve as ℓ increases and $\tilde{J} \approx J^*$ (within a constant shift).

Approximate (Nonoptimistic) Policy Iteration - Performance Bound



Typical Behavior: Oscillations within an error zone

- "Size" of the zone depends on the "approximation quality" of policy evaluation (δ) and policy improvement (ϵ).
- When the generated policies converge, the performance bound is better.



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

- Examples: A linear featured-based architecture or a neural net.
- Approximate policy evaluation/training: 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 a trajectory that starts at i^s, using μ for some number N of stages, accumulating the corresponding discounted costs, and adding a terminal cost approximation $\alpha^N \hat{J}(i_N)$.

Training, Exploration, and Other Issues

- The training problem

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

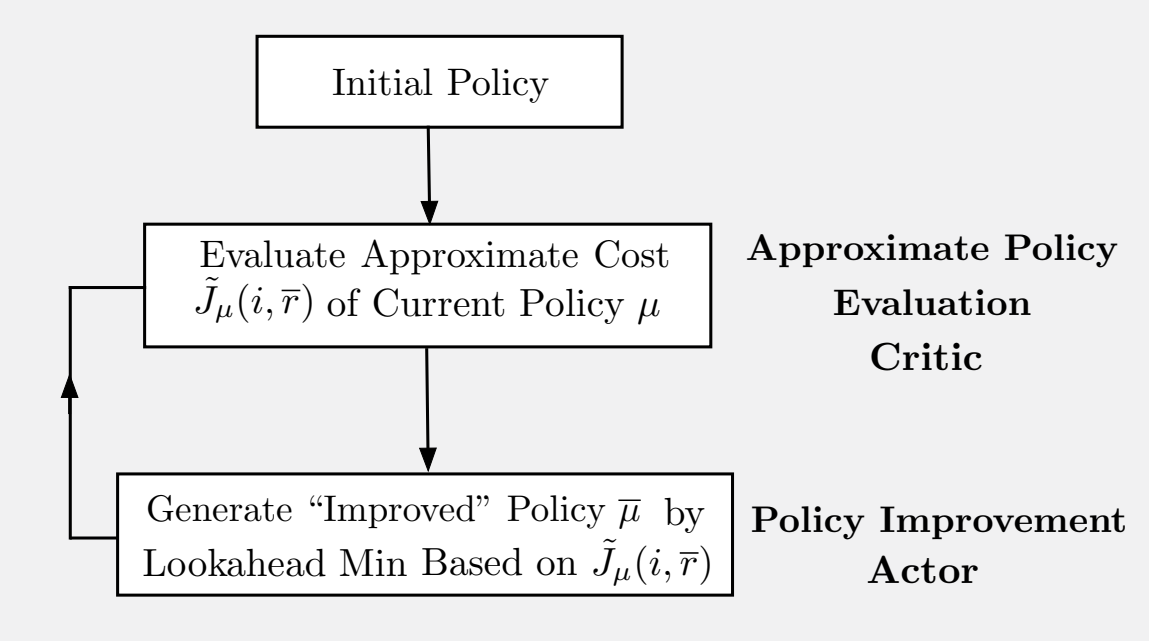
  is well-suited for incremental gradient:

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

  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."
- Cost shaping: Replace $g(i, u, j)$ with $\hat{g}(i, u, j) = g(i, u, j) + \alpha V(j) - V(i)$, to approximate $J_\mu - V$ rather than J_μ. Suboptimal policies depend on V, and V can capture much of the "nonlinearity" in J_μ. Allows the use of enhanced approximations.

A Working Break: Think About Exploration in Approximate PI

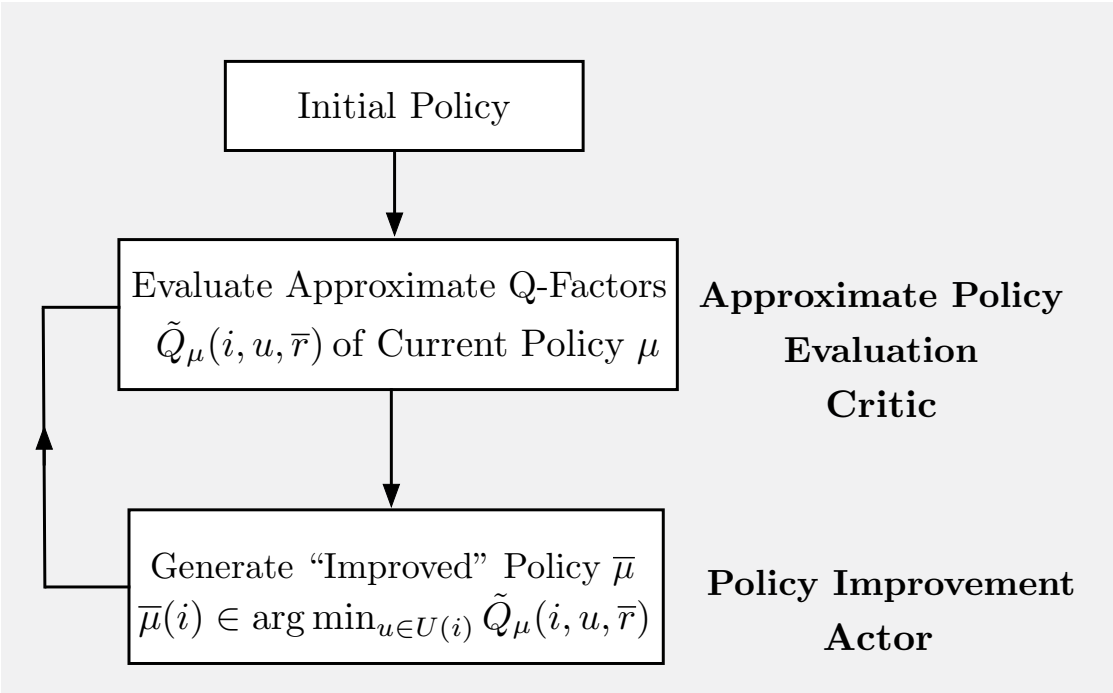


How would you introduce both exploration and trajectory reuse into policy evaluation?

- What kind of schemes would diversify the cost samples of a given policy μ?
- How would they work for deterministic problems?
- How would they work if we estimate Q-factors?

Answer: Many starting states, short trajectories, terminal cost approximation, use of an "off-policy".

Approximate PI Schemes with Q-Factors



Introduce a parametric architecture $\tilde{Q}_\mu(i,u,r)$ for Q-factor evaluation

- Approximate policy evaluation/training: Generate training triplets (i^s, u^s, β^s), where β^s is a sample Q-factor corresponding to (i^s, u^s). Use least squares/regression:

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

- β^s is generated by simulating a trajectory that starts at (i^s, u^s), using μ for some number N of stages, accumulating the corresponding discounted costs, and adding a terminal cost approximation $\alpha^N \hat{J}(i_N)$.

Approximate PI with Approximation in Policy Space on Top of Approximation in Value Space

Trajectory Reuse and Exploration Issues

- Trajectory reuse is more problematic in Q-factor evaluation than in cost evaluation; each trajectory generates state-control pairs of the special form $\big(i, \mu(i)\big)$ at every stage after the first, so pairs (i, u) with $u \neq \mu(i)$ are not adequately explored.
- For this reason, it is necessary to make an effort to include in the samples a rich enough set of trajectories that start at pairs (i, u) with $u \neq \mu(i)$.
- An alternative approach: First compute in model-free fashion a cost function approximation $\tilde{J}_\mu(j, \bar{r})$, and then use a second sampling process and regression to approximate further the (already approximate) Q-factor

$$\sum_{j=1}^{n} p_{ij}(u)\big(g(i,u,j) + \alpha \tilde{J}_\mu(j, \bar{r})\big),$$

  with some $\tilde{Q}_\mu(i, u, \bar{r})$ possibly obtained with a policy approximation architecture.
- This is model-free approximate PI that is based on approximation in policy space on top of approximation in value space. It is more complex, but allows trajectory reuse and thus deals better with the exploration issue.

Q-Learning with Lookup Table Representation

Recall the VI Algorithm for Q-Factors $Q_{k+1} = FQ_k$ where F is the operator

$$(FQ)(i,u) = \sum_{j=1}^{n} p_{ij}(u)\left(g(i,u,j) + \alpha \min_{v \in U(j)} Q(j,v)\right), \qquad \text{for all } (i,u)$$

F is a contraction with modulus α, so VI converges to Q^*.

Q-Learning is a simulation-based VI algorithm for Q-factors, based on "asynchronous DP" ideas [iterate on a single pair (i,u) at a time]

- An infinitely long sequence of state-control pairs $\{(i^k, u^k)\}$ is generated according to some (essentially arbitrary) probabilistic mechanism.
- For each pair (i^k, u^k), a state j^k is generated according to the probabilities $p_{i^k j}(u^k)$.
- Then only the Q-factor of (i^k, u^k) is updated using a stepsize $\gamma^k \in (0,1]$; all other Q-factors are left unchanged:

$$Q_{k+1}(i,u) = (1-\gamma^k)Q_k(i,u) + \gamma^k (F_k Q_k)(i,u), \qquad \text{for all } (i,u),$$

where $(F_k Q_k)(i,u) = Q_k(i,u)$ if $(i,u) \neq (i^k, u^k)$, and

$$(F_k Q_k)(i^k, u^k) = g(i^k, u^k, j^k) + \alpha \min_{v \in U(j^k)} Q_k(j^k, v) \qquad \text{if } (i,u) = (i^k, u^k)$$

- To guarantee convergence some technical conditions are needed, e.g., $\gamma^k \to 0$.

Optimistic Policy Iteration Methods with Q-Factor Approximation

Introduce a linear parametric architecture $\tilde{Q}(i,u,r) = \phi(i,u)'r$, and iterate on r. Each value of r defines a policy, which generates controls. As r is iterated on the policy changes.

SARSA: At iteration k, we have r^k, i^k, and we have chosen a control u^k

- We simulate the next transition (i^k, i^{k+1}) using the transition probabilities $p_{i^k j}(u^k)$.
- We generate u^{k+1} with the minimization $u^{k+1} \in \arg\min_{u \in U(i^{k+1})} \tilde{Q}(i^{k+1}, u, r^k)$ [In some schemes, u^{k+1} is chosen with a small probability to be a different element of $U(i^{k+1})$ to enhance exploration.]
- We update the parameter vector via

$$r^{k+1} = r^k - \gamma^k \phi(i^k, u^k) q_k,$$

where γ^k is a positive stepsize, and q_k is given by

$$q_k = \phi(i^k, u^k)' r^k - \alpha \phi(i^{k+1}, u^{k+1})' r^k - g(i^k, u^k, i^{k+1})$$

- The vector $\phi(i^k, u^k) q_k$ can be interpreted as an approximate gradient direction, and q_k is referred to as a temporal difference.

A Projection View of Approximate Policy Evaluation

- Approximation of solution of Bellman's equation $J_\mu = T_\mu J_\mu$ with a parametric architecture amounts to replacing J_μ with a vector in

$$\mathcal{M} = \Big\{ \big(\tilde{J}(1,r), \ldots, \tilde{J}(n,r)\big) \;\Big|\; \text{all parameter vectors } r \Big\}$$

- A common approach uses projection onto $\mathcal{M}$:

$$\Pi(J) \in \arg\min_{V \in \mathcal{M}} \|J - V\|^2$$

where

$$\|J\|^2 = \sum_{i=1}^{n} \xi_i \big(J(i)\big)^2,$$

where $J(i)$ are the components of J, and ξ_i are some positive weights.

Three general approaches for approximation of J_μ using projection

- Project J_μ onto $\mathcal{M}$ to obtain $\Pi(J_\mu)$, which is used in place of J_μ.
- Start with some approximation $\hat{J}$ of J_μ, perform N VIs to obtain $T_\mu^N \hat{J}$, and project onto $\mathcal{M}$ to obtain $\Pi(T_\mu^N \hat{J})$. We then use $\Pi(T_\mu^N \hat{J})$ in place of J_μ.
- Solve a projected equation $J_\mu = \Pi(T_\mu J_\mu)$, and use the solution in place of J_μ.

Approximate Projection by Monte-Carlo Simulation

- We focus on the case where the manifold $\mathcal{M}$ is a subspace $\mathcal{M} = \{\Phi r \mid r \in \Re^m\}$ where Φ is an $n \times m$ matrix with rows denoted by $\phi(i)'$, $i = 1, \ldots, n$.
- The projection $\Pi(J)$ is of the form Φr^*, where

$$r^* \in \arg\min_{r \in \Re^m} \|\Phi r - J\|_\xi^2 = \arg\min_{r \in \Re^m} \sum_{i=1}^n \xi_i \big(\phi(i)' r - J(i)\big)^2$$

- This minimization can be done in closed form,

$$r^* = \left(\sum_{i=1}^n \xi_i \phi(i)\phi(i)'\right)^{-1} \sum_{i=1}^n \xi_i \phi(i) J(i)$$

View the two terms as expectations and approximate them by MC simulation

- Generate samples i^s, $s = 1, \ldots, q$, according to ξ, and form the estimates

$$\frac{1}{q}\sum_{s=1}^q \phi(i^s)\phi(i^s)' \approx \sum_{i=1}^n \xi_i \phi(i)\phi(i)', \qquad \frac{1}{q}\sum_{t=1}^q \phi(i^s)\beta^s \approx \sum_{i=1}^n \xi_i \phi(i) J(i)$$

  where β^s is a sample of $J(i^s)$ plus a "zero mean noise" term $n(i^s)$ (see the text).
- Estimate r^* by $\overline{r} = \left(\sum_{t=1}^q \phi(i^s)\phi(i^s)'\right)^{-1} \sum_{t=1}^q \phi(i^s)\beta^s$

Connection with Least Squares

The solution of the simulation-based approximate projection

$$\overline{r} = \left(\sum_{t=1}^{q} \phi(i^s)\phi(i^s)' \right)^{-1} \sum_{t=1}^{q} \phi(i^s)\beta^s$$

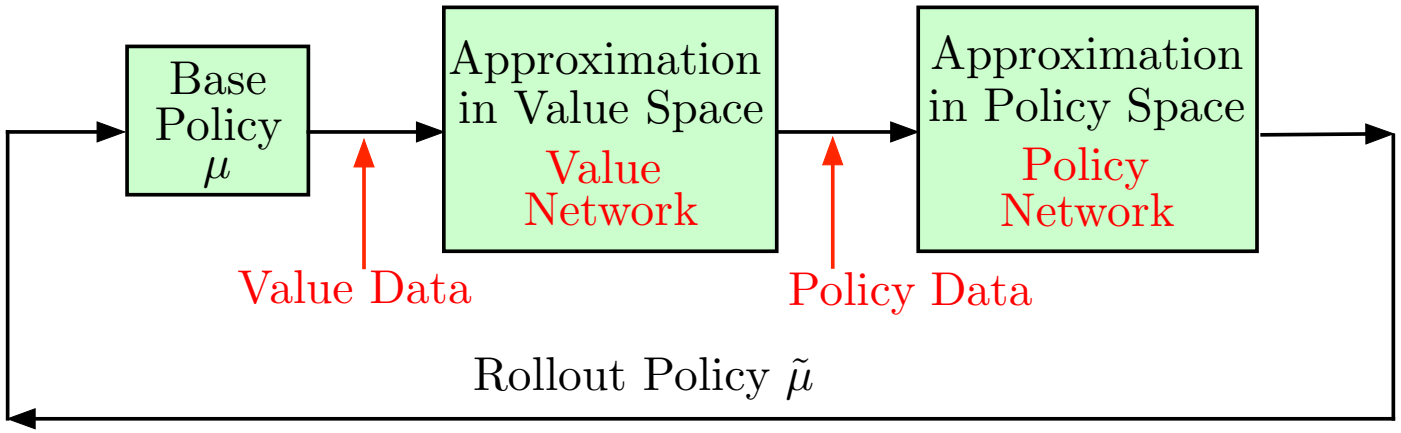
is also obtained by the least squares minimization

$$\overline{r} \in \arg\min_{r \in \Re^m} \sum_{s=1}^{q} \left(\phi(i^s)'r - \beta^s \right)^2$$

Thus simulation-based projection can be implemented in two equivalent ways

- Replacing expected values in the exact projection formula by simulation-based estimates.
- Replacing the exact least squares/projection problem with a simulation-based least squares approximation.
- It is not necessary that the simulation produces independent samples.
- It is sufficient that the long term empirical frequencies by which the indices i appear in the simulation sequence are consistent with the probabilities ξ_i.
- We do not need the probabilities ξ_i (the simulation determines them implicitly).

Rollout and Approximate Policy Iteration: Consider the Computations



Lots of computation needed for off-line training and on-line play

- Collection of training data may require lots of simulation/computation
- The training algorithm (e.g., gradient method) may be slow
- Exploring adequately a large state space is an issue
- On-line play requires minimization and truncated rollout, possibly under tight time constraints

HOW DO WE USE PARALELLIZATION IN ROLLOUT AND APPROXIMATE PI?

Four Possible Types of Parallelization

Q-factor parallelization: At the current state x, one-step lookahead/rollout does a separate Q-factor calculation for each control $u \in U(x)$. These calculations are decoupled and can be executed in parallel.

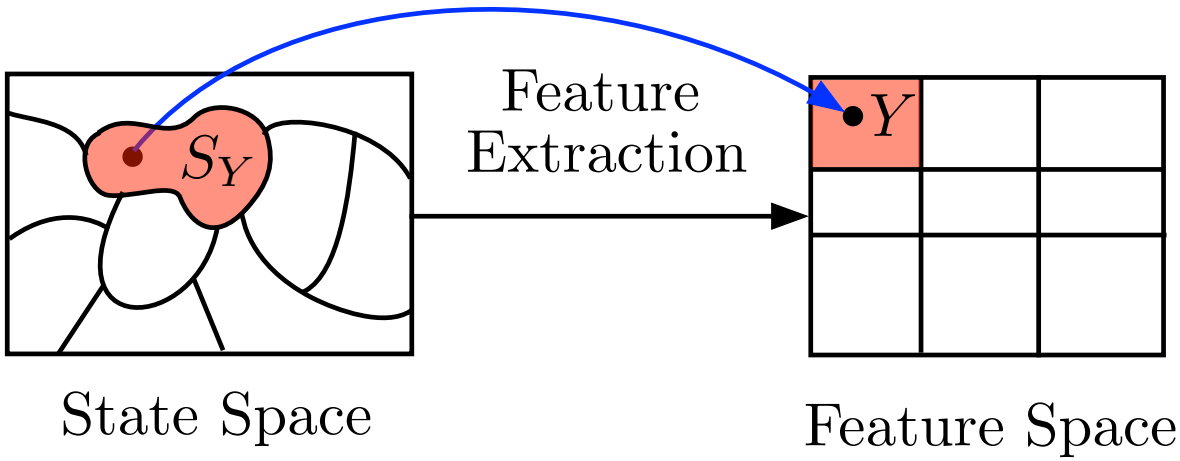
Monte Carlo parallelization: Each of the Q-factor calculations involves a Monte Carlo simulation when the problem is stochastic. Monte Carlo simulation can be parallelized.

Multiprocessor parallelization: Use a state space partition, and execute separate (but coupled) value and policy approximations on each subset in parallel.

Multiagent parallelization: When the control has m components, $u = (u^1, \ldots, u^m)$ the lookahead minimization at x involves the computation of as many as n^m Q-factors, where n is the max number of possible values of u^i. Multiagent (possibly autonomous) schemes can reduce the computation dramatically.

WE WILL FOCUS ON THE LAST TWO

Multiprocessor Parallelization: State Space Partitioning



Partition the state space into several subsets and construct a separate policy and value approximation in each subset.

- Use features to generate the partition.
- How do we implement truncated rollout and policy iteration with partitioning?

Distributed Asynchronous Policy Iteration (Williams and Baird, 1993, Bertsekas and Yu, 2010)

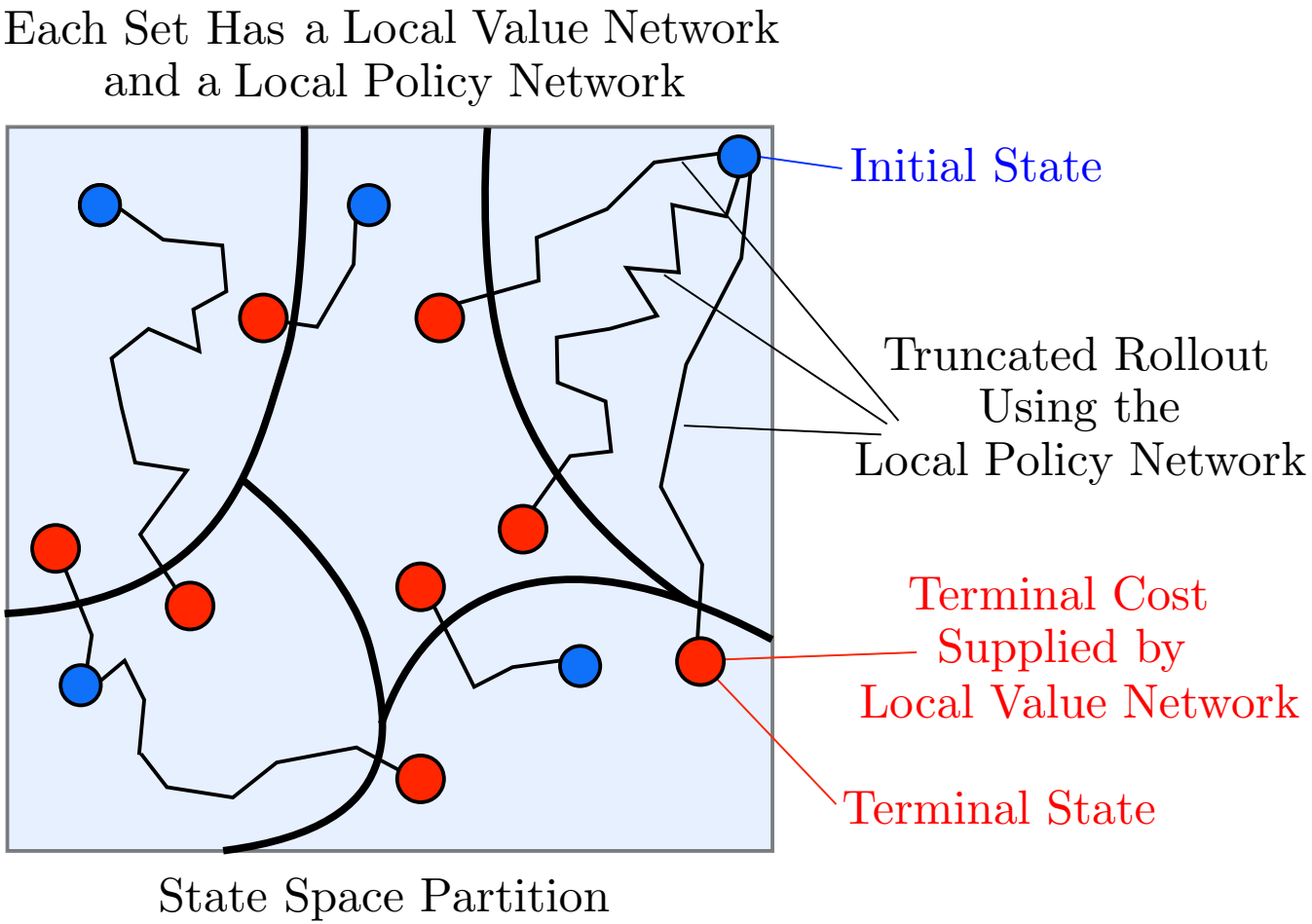
An old and fairly obvious training idea:

- Assign one processor to each subset of the partition.
- Each processor uses a local value and a local policy approximation, and maintains asynchronous communication to other processors.
- Update values locally on each subset (policy evaluation by value iteration).
- Update policies locally on each subset (policy improvement, possibly using multiagent parallelization).
- Communicate asynchronously local values and policies to other processors.

However:

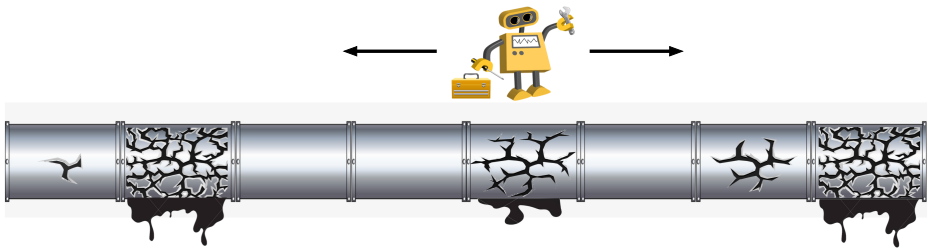
- The obvious algorithm fails (for the lookup table representation case - a counterexample by Williams and Baird, 1993).
- The DPB-HJY algorithm, 2010, corrects this difficulty and proves convergence (assuming a lookup table representation for policies and cost functions).
- Admits extension to neural net approximations (some error bounds available).

Approximate Policy Iteration with Local Value and Policy Networks

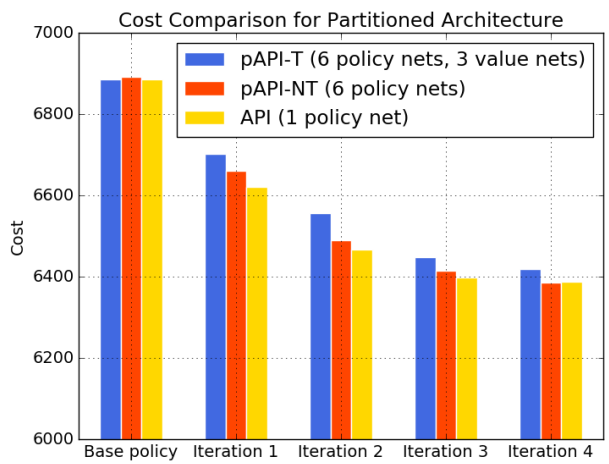


- Start with some base policy and a value network for each set.
- Obtain a policy and a value network for the truncated rollout policy. Repeat.
- Partitioning may be a good way to deal with adequate state space exploration.

Distributed RL for POMDP (Bhatacharya, Badyal, Wheeler, Gil, Bertsekas Paper, 2020)



- 20 potentially damaged locations along a pipeline.
- Damage of each location is imperfectly known; evolves according to a Markov chain (5 levels of damage). Number of states: $\approx 10^{15}$
- Repair robot moves left or right, visits and repairs locations. May want to give preference to “urgent" repairs.
- Belief space partitioning with 6 policy networks and 3 value networks.



Parallelization of Agent-by-Agent Policy Improvement

Simplified minimization (one-agent-at-a-time in a given order) reduces dramatically the cost of policy improvement, but it is an inherently serial computation. Each agent needs the rollout control of the preceding agents in the order.

How can we parallelize it?

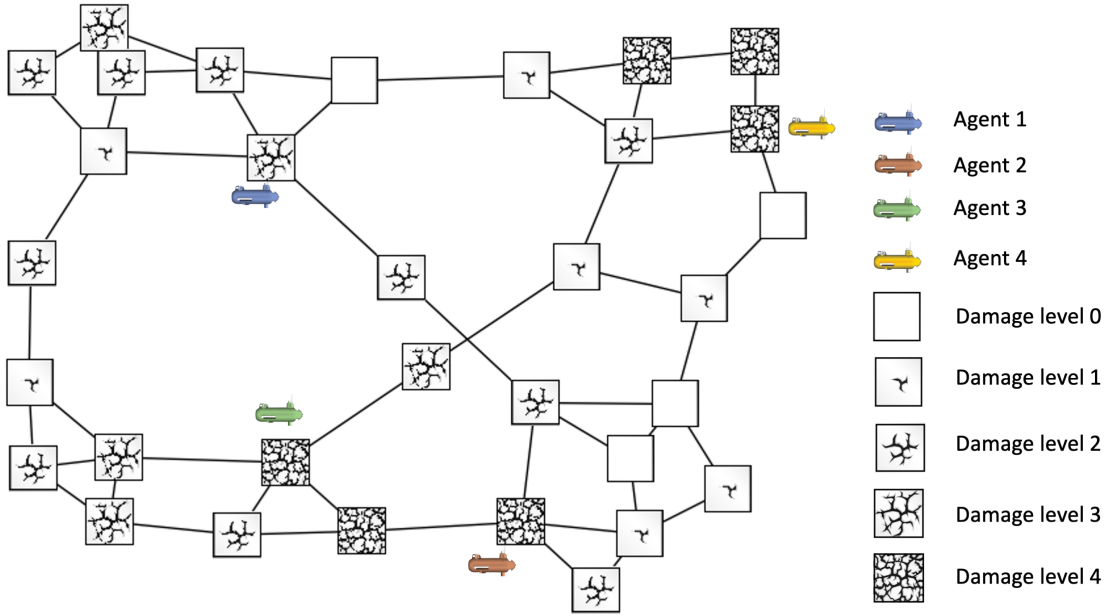
Precomputed signaling

- Obstacle to parallelization/agent autonomy: To compute the kth agent rollout control we need the rollout controls of the preceding agents $i < k$
- Signaling remedy: Use precomputed substitute "guesses" $\widehat{\mu}_i(x)$ in place of the preceding rollout controls $\tilde{\mu}_i(x)$

Signaling possibilities

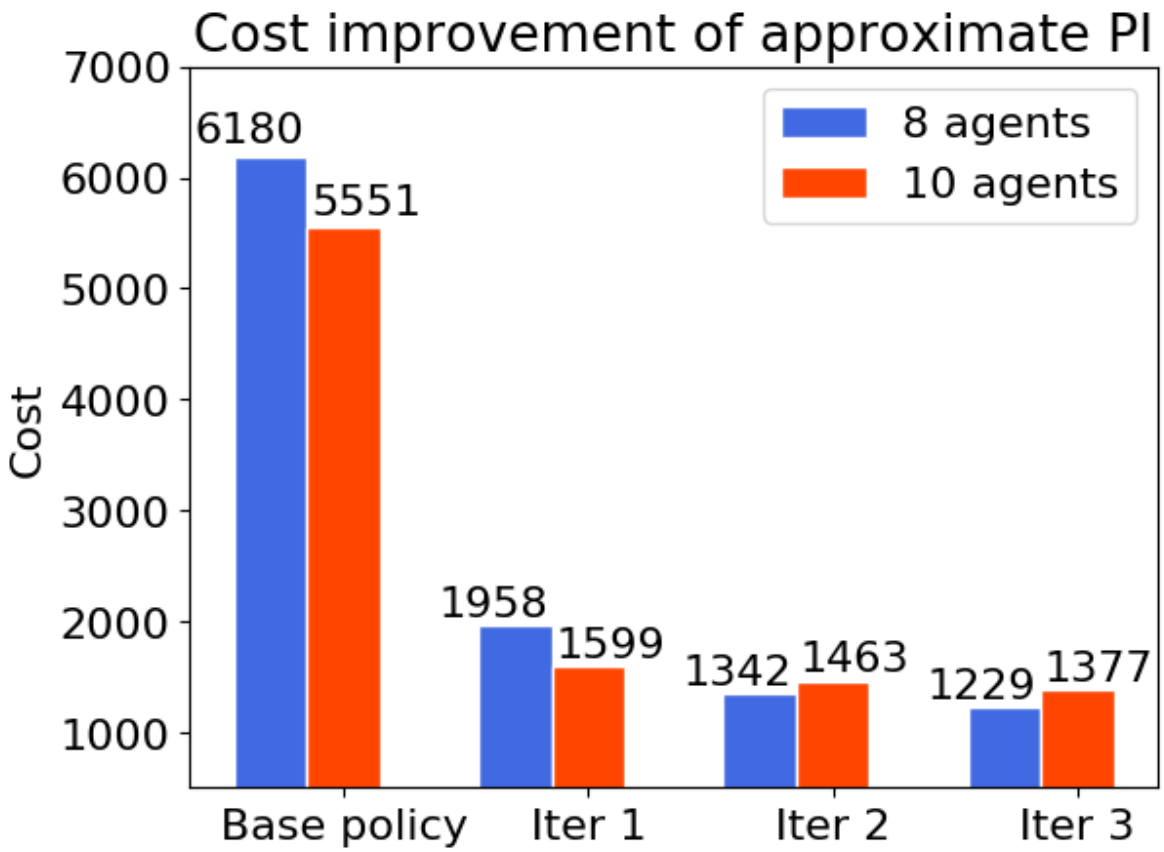
- Use the base policy controls for signaling $\widehat{\mu}_i(x) = \mu_i(x)$, $i = 1, \ldots, k-1$ (this may work poorly)
- Use a neural net representation of the rollout policy controls for signaling $\widehat{\mu}_i(x) \approx \tilde{\mu}_i(x)$, $i = 1, \ldots, k-1$ (this requires off-line computation)
- Other, problem-specific possibilities

Multirobot Repair of a Network of Damaged Sites (Bhatacharya, Kailas, Badyal, Gil, Bertsekas Paper, 2021)



- Damage level of each site is unknown, except when inspected. It deteriorates according to a known Markov chain unless the site is repaired
- Control choice of each robot: Inspect and repair (which takes one unit time), or inspect and move to a neighboring site
- State of the system: The set of robot locations, plus the belief state of the site damages (the joint probability distribution of the damage levels of the sites)
- Stage cost at each unrepaired site: Depends on the level of its damage
- A POMDP with $\approx 10^{30}$ states and $\approx 10^{7}$ controls

Approximate Policy Iteration with Policy Nets (Bhatacharya, Kailas, Badyal, Gil, Bertsekas Paper, 2021)



- Recall that a policy network must be used to represent a policy generated by PI
- As a result the PI training must be done off-line
- Typical performance: Large cost improvement at first few iterations, which tails off and ends up in an oscillation as the number of generated policies increases

Concluding Remarks on Distributed RL

- RL is a VERY computationally intensive methodology.
- Distributed asynchronous computation is an obvious answer.
- It is important to identify methods that are amenable to distributed computation.
- One-time rollout with a base policy, multiagent parallelization, and/or local value and policy networks is well-suited. Often easy to implement, typically reliable.
- Repeated rollout (i.e, approximate policy iteration) with partitioned architecture and multiagent parallelization, and/or local value and policy networks is well-suited, but is more complicated and more ambitious.
- Multiagent rollout parallelization has many applications to discrete/combinatorial optimization problems.
- There are many interesting analytical and implementation challenges.

About the Next Lecture

Note:

The performance shown in the graphs of the two robot repair problems corresponds to the off-line trained policies. The performance of the corresponding on line play/rollout algorithm using the last policy obtained from off-line policy iteration is much better.

We will cover additional methods:

- The linear programming approach
- Approximation in policy space
- Policy gradient methods
- Random search methods

As preparation:

Review videolecture 11 of the 2021 ASU course