Rollout, Policy Iteration, and Distributed Reinforcement Learning

Current Course at ASU
(Research monograph to appear; partial draft at my website)

Dimitri P. Bertsekas

February 2020
Outline

1. Approximate Policy Iteration
2. Approximate Policy Iteration with Value and Policy Networks
3. Multiagent Rollout - Simplifying and Parallelizing the One-Step Lookahead
4. Multiprocessor Parallelization
AlphaGo (2016) and AlphaZero (2017)

**AlphaZero**

Plays much better than all chess programs

Plays different!

Alphazero has discovered a new way to play!

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

Same algorithm learned multiple games (Go, Shogi)

The AlphaZero methodology is based on several ideas:

- The fundamental DP idea of policy iteration/improvement.
- Approximations with value and policy neural net approximations.
- Massive parallel computation.
- Lookahead approximations: Monte Carlo Tree Search.

We will aim to:

Develop a methodology that relates to AlphaZero, but applies far more generally.
Recall the $\alpha$-Discounted Markovian Decision Problem

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

Policies $\mu$ that map states to controls, with $\mu(x) \in U(x)$ for all $x$ and $k$.

Cost of stage $k$: $\alpha^k g(x_k, \mu(x_k), w_k)$; $\alpha \in (0, 1]$ is the discount factor.

Cost of policy $\mu$

$$J_\mu(x_0) = \lim_{N \to \infty} E_{w_k} \left\{ \sum_{k=0}^{N-1} \alpha^k g(x_k, \mu(x_k), w_k) \right\}$$

Optimal cost function $J^*(x_0) = \min_{\mu} J_\mu(x_0)$.

Optimality condition: Minimize the RHS of Bellman’s equation

$$\mu^*(x) \in \arg \min_{u \in U(x)} E \left\{ g(x, u, w) + J^*(f(x, u, w)) \right\}$$
Policy Iteration Algorithm

Fundamental policy improvement property

\[ J_{\tilde{\mu}}(x) \leq J_\mu(x), \text{ for all } x \]

There are many variants of policy iteration

Optimistic, multistep, Q-learning versions, etc.

OUR FOCUS: APPROXIMATE VERSIONS
Approximate Policy Iteration

Policy improvement property holds approximately

Methodological issues to deal with for challenging large-scale problems

- **Theoretical issues**: Error bounds, convergence guarantees, sampling efficiency, etc.
- **Implementation choices**: What to approximate, how to sample, how to train, on-line vs off-line, model-free vs model-based, etc.
- **No guarantee of success**: We just try different schemes based on theoretical understanding, intuition, experience ... hopefully something will work.

**OUR FOCUS**: DISTRIBUTED (ASYNCHRONOUS) COMPUTATION
We will focus on two types of distributed computation schemes

- **Multiagent parallelization**: Deal with large control spaces, e.g., controls with multiple components
  \[ u = (u^1, \ldots, u^m) \]

- **Multiprocessor parallelization**: Deal with large state spaces through partitioning, and distributed training of multiple value and policy networks (one per set of the state space partition).

References

- Distributed asynchronous value iteration papers (DPB, 1982-83), Parallel and Distributed Computation book (DPB and Tsitsiklis, 1989).
- Distributed asynchronous policy iteration and Q-learning papers (Williams and Baird, 1993, DPB and Yu, 2010-14).
- Multiagent rollout paper (DPB, 2019).
- Partitioned rollout and policy iteration for POMDP paper (Bhattacharya, Badyal, Wheeler, Gil, DPB, 2020).
Approximation in Value Space: From Values $\tilde{J}(x)$ to a Policy $\tilde{\mu}(x)$

At state $x$, use $\tilde{J}$ (in place of $J^*$) in Bellman’s Eq. to obtain a control $\tilde{u} = \tilde{\mu}(x)$.

THE THREE APPROXIMATIONS:

- How to construct $\tilde{J}$.
- How to simplify $E\{\cdot\}$ operation.
- How to simplify min operation.

Each of the three approximations can be designed almost independently of the others, leading to a large variety of methods.
Optimization and training over a parametric family of policies $\mu(x, r)$, where $r$ is a parameter (e.g., a neural net).
From Value Approx. $\tilde{J}(x)$ to Policy $\mu(x, r)$

$x$ is classified as type $u$ $\iff$ at state $x$ we apply control $u$

Training the rollout policy as a classifier:

- We generate a training set of sample pairs $(x^s, u^s)$, $s = 1, \ldots, q$, by one-step lookahead, i.e.,

$$u^s \in \arg\min_{u \in U(x)} E\left\{ g(x^s, u, w) + \alpha \tilde{J}(f(x^s, u, w)) \right\}$$

- Approximate the one-step lookahead policy using the training set.
- Example: Introduce a parametric family of policies $\mu(x, r)$ of some form (e.g., a neural net). Then estimate $r$ by least squares fit

$$\min_r \sum_{s=1}^{q} \| u^s - \mu(x^s, r) \|^2$$
From Policy $\mu$ to Value Approx. $\tilde{J}$

Policy $\mu$ defines a cost approximation $\tilde{J} \approx J_\mu$ through truncated simulation

How to approximate $J_\mu(x)$?
- For deterministic problems: Run $\mu$ from $x$ once and accumulate stage costs.
- For stochastic problems: Run $\mu$ from $x$ many times and Monte Carlo average.
- Use truncation: Simulate $\mu$ for a limited number of stages, and neglect the costs of the remaining stages or add some cost approximation at the end to compensate.
**Policy improvement property:** In the idealized case (no approximations),

\[ J_{\tilde{\mu}}(x) \leq J_{\mu}(x), \quad \text{for all } x \]

- With approximations, policy improvement is approximate (within an error bound).
- **There are many variants of this scheme:** Optimistic policy iteration, Q-learning, temporal differences, etc.
- Most RL algorithms, including Alphazero, use variants of the above scheme.
- Some variants are highly optimistic, i.e., use very little data between value updates and policy updates.

**HOW DO WE USE PARALLELIZATION 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.

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 $u^i$. We will consider schemes that reduce the computation dramatically (to $n \cdot m$).

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

WE WILL FOCUS ON THE LAST TWO
A Spiders-and-Fly Example (or Search-and-Rescue)

15 spiders move along 4 directions (≤ 1 unit) w. perfect observation; fly moves randomly

Objective is to catch the fly in minimum time.
One-step lookahead and rollout are impossible: ≈ 5^{15} Q-factors.
We reformulate one-step lookahead but maintain the cost improvement property:
- Spiders move one-at-a-time with knowledge of other spiders’ and fly’s positions.
- The control is broken down into a sequence of 15 spider moves (5 \cdot 15 = 75 Q-factors).
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

\[
J^1(x_k, u_k^1), J^2(x_k, u_k^1, u_k^2), \ldots, J^{m-1}(x_k, u_k^1, \ldots, u_k^{m-1})
\]

- Multiagent (one-component-at-a-time) rollout is just standard rollout for the reformulated problem.

- The increase in size of the state space does not adversely affect rollout.

- The cost improvement property is maintained.

- Complexity reduction: The one-step lookahead branching factor is reduced from \( n^m \) to \( nm \), where \( n \) is the number of possible choices for each component \( u_k^i \).
A Single Step of Policy Iteration

Time to catch the flies

- Base policy (each spider follows the shortest path): 85
- Rollout (all spiders move at once, 625 move choices): 34
- Rollout (spiders move one at a time, 20 move choices): 34
Multiagent Parallelization and Coordination Issues

- One-at-a-time rollout and all-at-once rollout produce different rollout policies. One may be better than the other.
- **Exact policy iteration issues.** One-at-a-time rollout used repeatedly (as in policy iteration) may stop short of the optimal.
- **We speculate that in approximate policy iteration, one-at-a-time rollout will often perform about as well as all-at-once rollout.**
- **We can try to induce agent parallelization and asynchronism:** Divide agents in “weakly coupled groups” ... Require little or no coordination among groups.
Several interesting theoretical and algorithmic issues remain to be resolved

- How do we form groups? Use feature-based groupings?
- Frequency of communication?
- Aggregated coordination between groups?
- Distributed info processing?
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

Each Set Has a Local Value Network and a Local Policy Network

1. Start with some base policy and a value network for each set.
2. Obtain a policy and a value network for the truncated rollout policy. Repeat.
3. Partitioning may be a good way to deal with adequate state space exploration.
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.
Concluding Remarks on Distributed RL

- RL is a VERY computationally intensive methodology.
- Parallel 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.
- Rollout has close connections to model predictive control.
- Rollout has many applications to discrete/combinatorial optimization problems.
- There are many interesting analytical and implementation challenges.
Thank you!