Reinforcement Learning and Optimal Control
A Selective Overview

Dimitri P. Bertsekas

Laboratory for Information and Decision Systems
Massachusetts Institute of Technology

2018 CDC
December 2018
Reinforcement Learning (RL): A Happy Union of AI and Decision/Control Ideas

- **AI/RL**: Learning through Experience, Simulation, Model-Free Methods, Feature-Based Representations, A*/Games/Heuristics

- **Decision/Control/DP**: Principle of Optimality, Markov Decision Problems, POMDP, Policy Iteration, Value Iteration

**Complementary Ideas**
Late 80s-Early 90s

**Historical highlights**
- Exact DP, optimal control (Bellman, Shannon, 1950s ...)
- First major 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 and Alphazero (DeepMind, 2016, 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)
AlphaZero was Trained Using Self-Generated Data

- The “current” player plays games that are used to “train” an “improved” player.
- At a given position, the “move probabilities” and the “value” of a position are approximated by a deep neural net (NN).
- Successive NNs are trained using self-generated data and a form of regression.
- A form of randomized policy improvement Monte-Carlo Tree Search (MCTS) generates move probabilities.
- AlphaZero bears similarity to earlier works, e.g., TD-Gammon (Tesauro, 1992), but is more complicated because of the MCTS and the deep NN.
- The success of AlphaZero is due to a skillful implementation/integration of known ideas, and 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**
- Deterministic $\longleftrightarrow$ Stochastic
- Combinatorial optimization $\longleftrightarrow$ Optimal control w/ infinite state/control spaces
- One decision maker $\longleftrightarrow$ 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 the sound ideas
Central Idea: Lookahead with an approximate cost

- Compute an approximation \( \tilde{J} \) to the optimal cost function \( J^* \)
- At current state, apply control that attains the minimum in

\[
\text{Current Stage Cost} + \tilde{J}(\text{Next State})
\]

Multistep lookahead extension

- At current state solve an \( \ell \)-step DP problem using terminal cost \( \tilde{J} \)
- Apply the first control in the optimal policy for the \( \ell \)-step problem

Example approaches to compute \( \tilde{J} \):

- Problem approximation: Use as \( \tilde{J} \) the optimal cost function of a simpler problem
- Rollout and model predictive control: Use a single policy iteration, with cost evaluated on-line by simulation or limited optimization
- Self-learning/approximate policy iteration (API): Use as \( \tilde{J} \) an approximation to the cost function of the final policy obtained through a policy iteration process
- Role of neural networks: “Learn” the cost functions of policies in the context of API; “learn” policies obtained by value space approximation
The purpose of this talk
To selectively review some of the methods, and bring out some of the AI-DP connections

References

- Quite a few Exact DP books (1950s-present starting with Bellman; my latest book “Abstract DP" came out earlier this year)
- Quite a few DP/Approximate DP/RL/Neural Nets books (1996-Present)
  - Bertsekas and Tsitsiklis, Neuro-Dynamic Programming, 1996
  - NEW DRAFT BOOK: Bertsekas, Reinforcement Learning and Optimal Control, 2019, on-line
- Many surveys on all aspects of the subject; Tesauro’s papers on computer backgammon, and Silver, et al., papers on AlphaZero
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.
System

\[ x_{k+1} = f_k(x_k, u_k, w_k), \quad k = 0, 1, \ldots, N - 1 \]

where \( x_k \): State, \( u_k \): Control, \( w_k \): Random disturbance

Cost function:

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

Perfect state information: \( u_k \) is applied with (exact) knowledge of \( x_k \)

Optimization over feedback policies \( \{\mu_0, \ldots, \mu_{N-1}\} \): Rules that specify the control \( \mu_k(x_k) \) to apply at each possible state \( x_k \) that can occur
Go backwards, $k = N - 1, \ldots, 0$, using

$$J_N(x_N) = g_N(x_N)$$

$$J_k(x_k) = \min_{u_k} E_{w_k} \left\{ g_k(x_k, u_k, w_k) + J_{k+1}(f_k(x_k, u_k, w_k)) \right\}$$

$J_k(x_k)$: Optimal cost-to-go starting from state $x_k$

Approximate DP is motivated by the ENORMOUS computational demands of exact DP

Approximation in value space: Use an approximate cost-to-go function $\tilde{J}_{k+1}$

$$\tilde{\mu}_k(x_k) \in \arg\min_{u_k} E_{w_k} \left\{ g_k(x_k, u_k, w_k) + \tilde{J}_{k+1}(f_k(x_k, u_k, w_k)) \right\}$$

There is also a multistep lookahead version

At state $x_k$ solve an $\ell$-step DP problem with terminal cost function approximation $\tilde{J}_{k+\ell}$. Use the first control in the optimal $\ell$-step sequence.
Approximation in Value Space Methods

One-step case at state $x_k$:

Approximate minimization

$$\min_{u_k} E \left\{ g_k(x_k, u_k, w_k) + \tilde{J}_{k+1}(x_{k+1}) \right\}$$

**Approximations:**
- Simplify $E\{\cdot\}$ (certainty equivalence)
- Adaptive simulation

**First Step**

**“Future”**

**Computation of $\tilde{J}_{k+1}$:**
- Problem approximation
- Rollout
- Model Predictive Control
- Parametric approximation
- Aggregation

Multistep case at state $x_k$:

DP minimization

$$\min_{u_k, \mu_{k+1}, \ldots, \mu_{k+\ell-1}} E \left\{ g_k(x_k, u_k, w_k) + \sum_{m=k+1}^{k+\ell-1} g_k(x_m, \mu_m(x_m), w_m) + \tilde{J}_{k+\ell}(x_{k+\ell}) \right\}$$

**First $\ell$ Steps**

**“Future”**

**Lookahead Minimization**

**Cost-to-go Approximation**
Problem Approximation: Simplify the Tail Problem and Solve it Exactly

Use as cost-to-go approximation $\tilde{J}_{k+1}$ the exact cost-to-go of a simpler problem

Many problem-dependent possibilities:

- **Probabilistic approximation**
  - Certainty equivalence: Replace stochastic quantities by deterministic ones (makes the lookahead minimization deterministic)
  - Approximate expected values by limited simulation
  - Partial versions of certainty equivalence

- **Enforced decomposition of coupled subsystems**
  - One-subsystem-at-a-time optimization
  - Constraint decomposition
  - Lagrangian relaxation

- **Aggregation**: Group states together and view the groups as aggregate states
  - Hard aggregation: $\tilde{J}_{k+1}$ is a piecewise constant approximation to $J_{k+1}$
  - Feature-based aggregation: The aggregate states are defined by “features” of the original states
  - Biased hard aggregation: $\tilde{J}_{k+1}$ is a piecewise constant local correction to some other approximation $\hat{J}_{k+1}$, e.g., one provided by a neural net
The base policy can be any suboptimal policy (obtained by another method)
- One-step or multistep lookahead; exact minimization or a “randomized form of lookahead” that involves “adaptive” simulation and Monte Carlo tree search
- With or without terminal cost approximation (obtained by another method)
- Some forms of model predictive control can be viewed as special cases (base policy is a short-term deterministic optimization)
- Important theoretical fact: With exact lookahead and no terminal cost approximation, the rollout policy improves over the base policy
Base policy was a backgammon player developed by a different RL method [TD(λ) trained with a neural network]; was also used for terminal cost approximation.

The best backgammon players are based on rollout ... but are too slow for real-time play (MC simulation takes too long).

AlphaGo has similar structure to backgammon.

The base policy and terminal cost approximation are obtained with a deep neural net. In AlphaZero the rollout-with-base-policy part was dropped (long lookahead suffices).
\[ \min_{u_k, \mu_{k+1}, \ldots, \mu_{k+\ell-1}} \mathbb{E} \left\{ g_k(x_k, u_k, w_k) + \sum_{m=k+1}^{k+\ell-1} g_k(x_m, \mu_m(x_m), w_m) + \tilde{J}_{k+\ell}(x_{k+\ell}) \right\} \]

\( \tilde{J}_k \) comes from a class of functions \( \tilde{J}_k(x_k, r_k) \), where \( r_k \) is a tunable parameter vector

**Feature-based architectures: The linear case**

State \( x_k \) → Feature Extraction Mapping → Feature Vector \( \phi_k(x_k) \) → Linear Mapping → Linear Cost Approximator \( r'_k \phi_k(x_k) \)
Training with Fitted Value Iteration

This is just DP with intermediate approximation at each step

- Start with $\tilde{J}_N = g_N$ and sequentially train going backwards, until $k = 0$
- Given $\tilde{J}_{k+1}$, we construct a number of samples $(x_k^s, \beta_k^s)$, $s = 1, \ldots, q$,
  $$\beta_k^s = \min_u E\left\{ g(x_k^s, u, w_k) + \tilde{J}_{k+1}(f_k(x_k^s, u, w_k), r_{k+1}) \right\}, \quad s = 1, \ldots, q$$
- We “train” $\tilde{J}_k$ on the set of samples $(x_k^s, \beta_k^s)$, $s = 1, \ldots, q$

Training by least squares/regression

- We minimize over $r_k$

$$\sum_{s=1}^{q} (\tilde{J}_k(x_k^s, r_k) - \beta_k^s)^2 + \gamma \|r_k - \bar{r}\|^2$$

where $\bar{r}$ is an initial guess for $r_k$ and $\gamma > 0$ is a regularization parameter
Neural Networks for Constructing Cost-to-Go Approximations $\tilde{J}_k$

**Major fact about neural networks**

They **automatically construct features** to be used in a linear architecture

- Neural nets are approximation architectures of the form

$$\tilde{J}(x, v, r) = \sum_{i=1}^{m} r_i \phi_i(x, v) = r' \phi(x, v)$$

involving two parameter vectors $r$ and $v$ with different roles

- View $\phi(x, v)$ as a feature vector
- View $r$ as a vector of linear weights for $\phi(x, v)$
- By training $v$ jointly with $r$, we obtain automatically generated features!

**Neural nets can be used in the fitted value iteration scheme**

Train the stage $k$ neural net (i.e., compute $\tilde{J}_k$) using a training set generated with the stage $k + 1$ neural net (which defines $\tilde{J}_{k+1}$)
State encoding (could be the identity, could include special features of the state)

Linear layer $Ay(x) + b$ [parameters to be determined: $v = (A, b)$]

Nonlinear layer produces $m$ outputs $\phi_i(x, v) = \sigma\left( (Ay(x) + b)_i \right)$, $i = 1, \ldots, m$

$\sigma$ is a scalar nonlinear differentiable function; several types have been used
(hyperbolic tangent, logistic, rectified linear unit)

Training problem is to use the training set $(x^s, \beta^s)$, $s = 1, \ldots, q$, for

$$\min_{v, r} \sum_{s=1}^{q} \left( \sum_{i=1}^{m} r_i \phi_i(x^s, v) - \beta^s \right)^2 + \text{(Regularization Term)}$$

Solved often with incremental gradient methods (known as backpropagation)

Universal approximation theorem: With sufficiently large number of parameters,
“arbitrarily" complex functions can be closely approximated
More complex NNs are formed by concatenation of multiple layers
- The outputs of each nonlinear layer become the inputs of the next linear layer
- A hierarchy of features
- Considerable success has been achieved in major contexts

Possible reasons for the success
- With more complex features, the number of parameters in the linear layers may be drastically decreased
- We may use matrices $A$ with a special structure that encodes special linear operations such as convolution
The $Q$-factor of a state-control pair $(x_k, u_k)$ at time $k$ is defined by

$$Q_k(x_k, u_k) = E \left\{ g_k(x_k, u_k, w_k) + J_{k+1}(x_{k+1}) \right\}$$

where $J_{k+1}$ is the optimal cost-to-go function for stage $k + 1$

Note that

$$J_k(x_k) = \min_{u \in U_k(x_k)} Q_k(x_k, u_k)$$

so the DP algorithm is written in terms of $Q_k$

$$Q_k(x_k, u_k) = E \left\{ g_k(x_k, u_k, w_k) + \min_{u \in U_{k+1}(x_{k+1})} Q_{k+1}(x_{k+1}, u) \right\}$$

We can approximate $Q$-factors instead of costs
Consider fitted value iteration of $Q$-factor parametric approximations\[\tilde{Q}_k(x_k, u_k, r_k) \approx E\left\{ g_k(x_k, u_k, w_k) + \min_{u \in U_{k+1}(x_{k+1})} \tilde{Q}_{k+1}(x_{k+1}, u, r_{k+1}) \right\}\]

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

We obtain $\tilde{Q}_k(x_k, u_k, r_k)$ by training with many pairs $((x^s_k, u^s_k), \beta_k^s)$, where $\beta_k^s$ is a sample of the approximate $Q$-factor of $(x^s_k, u^s_k)$. No need to compute $E\{\cdot\}$

No need for a model to obtain $\beta_k^s$. Sufficient to have a simulator that generates random samples of state-control-cost-next state\[((x_k, u_k), (g_k(x_k, u_k, w_k), x_{k+1}))\]

Having computed $r_k$, the one-step lookahead control is obtained on-line as\[\bar{\mu}_k(x_k) = \arg \min_{u \in U_k(x_k)} \tilde{Q}_k(x_k, u, r_k)\]

without the need of a model or expected value calculations

Also the on-line calculation of the control is simplified
A Few Remarks on Infinite Horizon Problems

- **Most popular setting**: Stationary finite-state system, stationary policies, discounting or termination state
- **Policy iteration (PI) method** generates a sequence of policies
  - The current policy $\mu$ is evaluated using a parametric architecture: $\tilde{J}_\mu(x, \bar{r})$
  - An "improved" policy $\bar{\mu}$ is obtained by one-step lookahead using $\tilde{J}_\mu(x, \bar{r})$
- The architecture is trained using simulation data with $\mu$
- Thus the system "observes itself" under $\mu$ and uses the data to "learn" the improved policy $\bar{\mu}$ - "self-learning"
- Exact PI converges to an optimal policy; approximate PI "converges" to within an "error zone" of the optimal, then oscillates
- TD-Gammon, AlphaGo, and AlphaZero, all use forms of approximate PI for training
A Few Topics we did not Cover in this Talk

- **Infinite horizon extensions**: Approximate value and policy iteration methods, error bounds, model-based and model-free methods
- **Temporal difference methods**: A class of methods for policy evaluation in infinite horizon problems with a rich theory, issues of variance-bias tradeoff
- **Sampling for exploration**, in the context of policy iteration
- **Monte Carlo tree search**, and related methods
- **Aggregation methods**, synergism with other approximate DP methods
- **Approximation in policy space**, actor-critic methods, policy gradient and cross-entropy methods
- **Special aspects of imperfect state information problems**, connections with traditional control schemes
- **Infinite spaces optimal control**, connections with aggregation schemes
- **Special aspects of deterministic problems**: Shortest paths and their use in approximate DP
- **A broad view of using simulation for large-scale computations**: Methods for large systems of equations and linear programs, connection to proximal algorithms
### Some words of caution
- There are challenging implementation issues in all approaches, and **no fool-proof methods**.
- Problem approximation and feature selection require **domain-specific knowledge**.
- Training algorithms are not as reliable as you might think by reading the literature.
- Approximate PI involves oscillations.
- 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!
Thank you!