Introduction to Reinforcement Learning with Function Approximation

Rich Sutton

Reinforcement Learning and Artificial Intelligence Laboratory
Alberta Centre for Machine Learning
Department of Computing Science
University of Alberta
Canada

(with thanks to David Silver and Michael Littman for some slides and ideas)
What is Reinforcement Learning?

- Agent-oriented learning—learning by interacting with an environment to achieve a goal
  - more **realistic** and **ambitious** than other kinds of machine learning
- Learning by trial and error, with only delayed evaluative feedback (reward)
  - the kind of machine learning most like natural learning
  - learning that can tell for itself when it is right or wrong
- The beginnings of a *science of mind* that is neither natural science nor applications technology
Lecture 1: Introduction to Reinforcement Learning

About RL

Many Faces of Reinforcement Learning

Computer Science

Economics

Mathematics

Engineering

Neuroscience

Machine Learning

Optimal Control

Reward System

Reinforcement Learning

Operations Research

Classical/Operant Conditioning

Bounded Rationality

David Silver 2015
Example: Hajime Kimura’s RL Robots

Before

Backward

After

New Robot, Same algorithm
The RL Interface

- Environment may be unknown, nonlinear, stochastic and complex
- Agent learns a policy mapping states to actions
  - Seeking to maximize its cumulative reward in the long run
Signature challenges of RL

- Evaluative feedback (reward)
- Sequentiality, delayed consequences
- Need for trial and error, to explore as well as exploit
- Non-stationarity
- The fleeting nature of time and online data
Some RL Successes

- Learned the world’s best player of Backgammon (Tesauro 1995)

- Learned acrobatic helicopter autopilots (Ng, Abbeel, Coates et al 2006+)

- Widely used in the placement and selection of advertisements and pages on the web (e.g., A-B tests)

- Used to make strategic decisions in Jeopardy! (IBM’s Watson 2011)

- Achieved human-level performance on Atari games from pixel-level visual input, in conjunction with deep learning (Google Deepmind 2015)

- In all these cases, performance was better than could be obtained by any other method, and was obtained without human instruction
Example: TD-Gammon

Start with a random Network
Play millions of games against itself
Learn a value function from this simulated experience

Six weeks later it’s the best player of backgammon in the world
Originally used expert handcrafted features, later repeated with raw board positions
RL + Deep Learning Performance on Atari Games

Space Invaders  Breakout  Enduro
RL + Deep Learning Performance on Atari Games

- Space Invaders
- Breakout
- Enduro
RL + Deep Learning, applied to Classic Atari Games

• Learned to play 49 games for the Atari 2600 game console, without labels or human input, from self-play and the score alone

mapping raw screen pixels

to predictions of final score for each of 18 joystick actions

• Learned to play better than all previous algorithms and at human level for more than half the games
Outline

- Introduction to RL successes and challenges
- The formal problem: Finite Markov decision processes
- Part I: Exact solution methods and core theoretical ideas
  - Semi-gradient methods
  - On-policy and off-policy methods
  - The deadly triad; how to evade or survive it
- Part II: Approximate solution methods
- Miscellany and closing remarks
You are the reinforcement learner!
(interactive demo)

Optimal policy (deterministic)

<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>2</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
</tr>
</tbody>
</table>

True model of the world
The Environment: 
A Finite Markov Decision Process (MDP)

- Discrete time \( t = 1, 2, 3, \ldots \)
- A finite set of states
- A finite set of actions
- A finite set of rewards
- Life is a trajectory:

\[ \ldots S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1}, R_{t+2}, S_{t+2}, \ldots \]

- With arbitrary Markov (stochastic, state-dependent) dynamics:

\[ p(r, s'|s, a) = \text{Prob} \left[ R_{t+1} = r, S_{t+1} = s' \mid S_t = s, A_t = a \right] \]
Policies

- Deterministic policy

\[ a = \pi(s) \]

- An agent following a policy

\[ A_t = \pi(S_t) \]

- Informally the agent’s goal is to choose each action so as to maximize the discounted sum of future rewards,

\[ R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots \]

- We are searching for a policy

<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>2</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
</tr>
</tbody>
</table>

The number of deterministic policies is *exponential* in the number of states.
**Action-value functions**

- An action-value function says how good it is to be in a state, take an action, and thereafter follow a policy:

\[
q_\pi(s, a) = \mathbb{E} \left[ R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots \bigg| S_t = s, A_t = a, A_{t+1:\infty} \sim \pi \right]
\]

---

**Action-value function for the optimal policy and \( \gamma = 0.9 \)**

<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>130.39</td>
</tr>
<tr>
<td>A</td>
<td>2</td>
<td>133.77</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>166.23</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
<td>146.23</td>
</tr>
</tbody>
</table>
Optimal policies

- A policy $\pi_*$ is optimal if it maximizes the action-value function:

$$q_{\pi_*}(s, a) = \max_{\pi} q_\pi(s, a) = q_*(s, a)$$

- Thus all optimal policies share the same optimal value function.

- Given the optimal value function, it is easy to act optimally:

$$\pi_*(s) = \arg \max_a q_*(s, a) \quad \text{“greedification”}$$

- We say that the optimal policy is greedy with respect to the optimal value function.

- There is always at least one deterministic optimal policy.
Part I

Exact Solution Methods
(tabular methods)
Q-learning, the simplest RL algorithm

1. Initialize an array \( Q(s, a) \) arbitrarily

2. Choose actions in any way, perhaps based on \( Q \), such that all actions are taken in all states (infinitely often in the limit)

3. On each time step, change one element of the array:

\[
\Delta Q(S_t, A_t) = \alpha \left( R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \right)
\]

4. If desired, reduce the step-size parameter \( \alpha \) over time

Theorem: For appropriate choice of 4, \( Q \) converges to \( q^* \), and its greedy policy to an optimal policy \( \pi^* \) (Watkins & Dayan 1992)

This is kind of amazing — learning long-term optimal behavior without any model of the environment, for arbitrary MDPs!
Demo

Off-policy learning gridworld
Policy improvement theorem

- Given the value function for *any policy* $\pi$:
  $$q_{\pi}(s, a) \text{ for all } s, a$$

- It can always be greedified to obtain a *better policy*:
  $$\pi'(s) = \arg\max_a q_{\pi}(s, a) \quad (\pi' \text{ is not unique})$$

- where better means:
  $$q_{\pi'}(s, a) \geq q_{\pi}(s, a) \text{ for all } s, a$$

- with equality only if both policies are optimal
The dance of policy and value (Policy Iteration)

Any policy evaluates to a unique value function (soon we will see how to learn it) which can be greedified to produce a better policy. That in turn evaluates to a value function which can in turn be greedified…

Each policy is strictly better than the previous, until eventually both are optimal.

There are no local optima.

The dance converges in a finite number of steps, usually very few.
The dance is very robust

to initial conditions

to delayed and asynchronous updating, as in parallel and distributed implementations

to incomplete evaluation and greedification
  • updating only some states but not others
  • updating only part of the way

to randomization and noise

in particular, it works if only a single state is updated at a time by a random amount that is only correct in expectation
The Explore/Exploit dilemma

- You can’t do the action that you think is best all the time
  - because you will miss out big—forever—if you are wrong
  - to find the real best action, you must explore them all…an infinite number of times!

- You also can’t explore all the time
  - because then you would never get any advantage of your learning

Thus you must both explore and exploit, but neither to excess. What is the right balance?

How did Q-learning escape the dilemma?
Q-learning, the simplest RL algorithm

1. Initialize an array \( Q(s, a) \) arbitrarily

2. Choose actions in any way, perhaps based on \( Q \), such that all actions are taken in all states (infinitely often in the limit)

3. On each time step, change one element of the array:

\[
\Delta Q(S_t, A_t) = \alpha \left( R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \right)
\]

4. If desired, reduce the step-size parameter \( \alpha \) over time
Bootstrapping

- The key idea underlying both dynamic programming (DP) and all temporal-difference (TD) learning

- Updating an estimate from an estimate, a guess from a guess

- Based on the Bellman expectation equation:

  \[
  q_{\pi}(s, a) = \mathbb{E}\left[ R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots \middle| S_t = s, A_t = a, A_{t+1:\infty} \sim \pi \right] \\
  = \mathbb{E}\left[ R_{t+1} + \gamma q_{\pi}(S_{t+1}, A_{t+1}) \middle| S_t = s, A_t = a, A_{t+1} \sim \pi \right]
  \]

- or the Bellman optimality equation:

  \[
  q^{*}(s, a) = \mathbb{E}\left[ R_{t+1} + \gamma \max_{a'} q^{*}(S_{t+1}, a') \middle| S_t = s, A_t = a \right]
  \]

Q-learning’s target for \(Q(S_t, A_t)\)
Q-learning is off-policy learning

- **Off-policy learning** is learning about the value of a policy other than the policy being used to generate the trajectory.

- Q-learning learns about the value of its deterministic greedy policy—which gradually become optimal—from data while behaving in a more exploratory manner.
  - thus **Q-learning is off-policy**
  - and this is essential to its strategy for escaping the explore/exploit dilemma.

- Some terminology
  - the **target policy** is the policy being learned about.
  - the **behavior policy** is the policy generating the trajectory data.
  - **on-policy learning** is when the two policies are the same.
Part II
Approximate Solution Methods
(function approximation)
So, RL finds optimal policies for arbitrary environments, if the value functions and policies can be exactly represented in tables.

But the real world is too large and complex for tables.

Will RL work with approximations?

Will RL work with function approximators?
Function approximation

- Represent the action-value function by a parameterized function approximator with parameter $\theta$

$$q(s, a, \theta) \approx q_*(s, a) \text{ or } \approx q_\pi(s, a)$$

- The approximator could be a deep neural network, with the weights being the parameter
  - or simply a linear weighting of features (the most pressing theoretical problems are all best addressed in this setting)

- Function approximation is a powerful concept, e.g., subsuming much of the problem of hidden state

- For large applications, it is important that all computations scale linearly with the number of parameters
Does Q-learning work with function approximation?

- Yes, there is a obvious generalization of Q-learning to function approximation (Watkins 1989)

- Often, it works well

- But there are counterexamples
  - simple examples where the parameters diverge to infinity
  - even for linear function approximation

- We could get by, but something is not right, there is something, probably many things, that we are not understanding
Consider the following objective function, based on the Bellman optimality equation:

$$\mathcal{L}(\theta) = \mathbb{E} \left[ \left( R_{t+1} + \gamma \max_a \hat{q}(S_{t+1}, a, \theta) - \hat{q}(S_t, A_t, \theta) \right)^2 \right]$$

The target here depends on the parameter, but if we ignore that dependence when taking the derivative, then we get a semi-gradient Q-learning update:

$$\Delta \theta_t = \alpha \left( R_{t+1} + \gamma \max_a \hat{q}(S_{t+1}, a, \theta_t) - \hat{q}(S_t, A_t, \theta_t) \right) \frac{\partial \hat{q}(S_t, A_t, \theta_t)}{\partial \theta_t}$$

- Consider instead an objective function based on the Bellman expectation equation:

  \[ L(\theta) = \mathbb{E} \left[ \left( R_{t+1} + \gamma \hat{q}(S_{t+1}, A_{t+1}, \theta) - \hat{q}(S_t, A_t, \theta) \right)^2 \right] \]

- Again the target depends on the parameter, and again we ignore that dependence when taking the derivative, this time to get the semi-gradient Sarsa update:

  \[ \Delta \theta_t = \alpha \left( R_{t+1} + \gamma \hat{q}(S_{t+1}, A_{t+1}, \theta_t) - \hat{q}(S_t, A_t, \theta_t) \right) \frac{\partial \hat{q}(S_t, A_t, \theta_t)}{\partial \theta_t} \]

- This is an on-policy algorithm: it approximates \( q_\pi \) not \( q_* \); thus \( \pi \) should be near greedy, typically it is \( \varepsilon \)-greedy.
Why is it called Sarsa?

- It is the only learning update that uses exactly these five things from the trajectory:

  $$\ldots S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1}, \ldots$$

- Sarsa is equivalent to the TD(0) algorithm (Sutton 1988) when applied to state-action pairs rather than to states

What is an $\epsilon$-greedy policy?

- An $\epsilon$-greedy policy is a stochastic policy that is usually greedy, but with small probability $\epsilon$ instead selects an action at random
As an on-policy method, Semi-gradient Sarsa has good convergence properties

- If the function approximator is linear, it is
  - guaranteed non-divergent for control, with bounded error (though may “chatter” –Gordon 1995)

- For general non-linear function approximation, there is one known counterexample, but it is very artificial and contrived

- On-policy methods typically perform better than off-policy methods, but find poorer policies
Cliff-walking example (on-policy vs off-policy)

\[ R = -1 \]

\[ R = -100 \]

\[ S \]

\[ T \]

\[ h \]

\[ e \]

\[ C \]

\[ l \]

\[ i \]

\[ f \]

\[ f \]

\[ G \]

safe path

optimal path

Example 3.9: Golf

To formulate playing a hole of golf as a reinforcement learning task, we count a penalty (negative reward) of \(-1\) for each stroke until we hit the ball into the hole. The state is the location of the ball. The value of a state is the negative of the number of strokes to the hole from that location. Our actions are how we aim and swing at the ball, of course, and which club we select. Let us take the former as given and consider just the choice of club, which we assume is either a putter or a driver. The upper part of Figure 3.6 shows a possible state-value function, \( v_{\text{put}}(s) \), for the policy that always uses the putter. The terminal state in-the-hole has a value of 0. From anywhere on the green we assume we can make a putt; these states have value 1. Off the green we cannot reach the hole by putting, and the value is greater. If we can reach the green from a state by putting, then that state must have value one less than the green's value, that is, 2. For simplicity, let us assume we can putt very precisely and deterministically, but with a limited range. This gives us the sharp contour line labeled 2 in the figure; all locations between that line and the green require exactly two strokes to complete the hole. Similarly, any location within putting range of the 2 contour line must have a value of 3, and so on to get all the contour lines shown in the figure. Putting doesn't get us out of sand traps, so they have a value of 1. Overall, it takes us six strokes to get from the tee to the hole by putting.
Cliff-walking example (on-policy vs off-policy)

Both algorithms are $\epsilon$-greedy

$\epsilon = 0.1$
Mountain Car Demo

- Linear function approximation
- Coarse-coded features of state (tile coding, CMAC)

Moore 1990, Sutton 1996
Acrobot Demo, Sarsa ($\lambda = 0.9$)
Episode 6
Acrobot Demo, Sarsa($\lambda=0.9$)
Episode 40+
RoboCup soccer keepaway

Stone, Sutton & Kuhlmann, 2005
Learned

Random

Hand-coded

Stone, Sutton & Kuhlmann, 2005
How is the state encoded?
In 13 continuous state variables

- 11 distances among the players, ball, and the center of the field
- 2 angles to takers along passing lanes
RoboCup Feature Vectors

Full soccer state

Sparse, coarse, tile coding

13 continuous state variables

Linear map $\theta$

Huge binary feature vector (about 400 1's and 40,000 0's)

action values
But let’s return to the bad news, the problem of instability with semi-gradient Q-learning
What causes the problem of instability?

- It has nothing to do with learning or sampling
  
  - Even dynamic programming, the classical solution method for known MDPs, suffers from divergence with function approximation

- It has nothing to do with exploration, greedification, or control
  
  - Even policy evaluation alone can diverge

- It has nothing to do with complex non-linear approximators
  
  - Even simple linear approximators can produce instability
The deadly triad

The risk of divergence arises whenever we combine three things:

1. Function approximation
   significantly generalizing from large numbers of examples

2. Bootstrapping
   learning value estimates from other value estimates,
   as in dynamic programming and temporal-difference learning

3. Off-policy learning  (Why is dynamic programming off-policy?)
   learning about a policy from data not due to that policy,
   as in Q-learning, where we learn about the greedy policy from data with a necessarily more exploratory policy

Any two without the third is ok
Can we do without bootstrapping?

- Bootstrapping is critical to the computational efficiency of DP.
- Bootstrapping is critical to the data efficiency of TD methods.
- On the other hand, bootstrapping introduces bias, which harms the asymptotic performance of approximate methods.
- The degree of bootstrapping can be finely controlled via the $\lambda$ parameter, from $\lambda=0$ (full bootstrapping) to $\lambda=1$ (no bootstrapping).
- For the naive loss:

$$\mathcal{L}(\theta) = \mathbb{E} \left[ (q_{\pi}(S_t, A_t) - \hat{q}(S_t, A_t, \theta))^2 \right]$$

semi-gradient Sarsa($\lambda$) converges to a fixpoint $\theta_{\text{Sarsa}}$ where

$$\mathcal{L}(\theta_{\text{Sarsa}}) \leq \frac{1 - \gamma \lambda}{1 - \gamma} \min_{\theta} \mathcal{L}(\theta)$$

Tsitsiklis & Van Roy 1997

$\Rightarrow \lambda=1$ is best!?
4 examples of the effect of bootstrapping suggest that $\lambda=1$ (no bootstrapping) is a very poor choice.

In all cases, lower is better.

Red points are the cases of no bootstrapping.
Other ways to survive the deadly triad

- Use high $\lambda$. Use good features.

- Recent results suggest that replay and more stable targets (e.g., Double Q-learning, van Hasselt 2010) may help, but it is too soon to be sure.

- Use least-squares methods like off-policy LSTD($\lambda$) (Yu 2010, Mahmood et al. 2015). Such methods (Bradtke & Barto 1996, Boyan 2000) easily survive the triad, but their computational costs scale with the square of the number of parameters.

- Try the new true-gradient RL methods (Gradient-TD and proximal-gradient-TD) developed by Maei (2011) and Mahadevan (2015) et al. These seem to me to be the best attempts to make TD methods with the robust convergence properties of stochastic gradient descent. Residual gradient methods (Baird 1999) are also true gradient methods, but optimize a poor objective, or can’t learn purely from data (double sampling). These and other methods based on the Bellman error/residual are not recommended.

- Try the even newer Emphatic-TD methods (Sutton, White & Mahmood 2015, Yu 2015). These semi-gradient methods attain stability through an extension of the early on-policy theorems.
Outline

- Introduction to RL successes and challenges
- The formal problem: Finite Markov decision processes
- Part I: Exact solution methods and core theoretical ideas
- Part II: Approximate solution methods
  - Semi-gradient methods
  - On-policy and off-policy methods
  - The deadly triad; how to evade or survive it
- Miscellany and closing remarks
The many dimensions of RL

- Problems
  - prediction vs control
  - MDPs vs Bandits (one state, non-sequential)

- Methods
  - Tabular vs function approximation
  - On-policy vs off-policy
  - Bootstrapping vs Monte Carlo (unified by eligibility traces)
  - Model-based vs model-free
  - Value-based vs policy-based

And yet there is an amazing unity and convergence of methods
Policy-gradient actor-critic methods

- Policy is explicitly represented with its own parameters independent of any value function.
- Policy parameters are updated by stochastic gradient ascent in a performance measure such as average reward per step.
- A state-value function (critic) is optional but can significantly reduce variance.
- Good convergence properties (on-policy).
Why approximate policies rather than values?

- In many problems, the policy is simpler than the value function
- In many problems, the optimal policy is stochastic
  - e.g., bluffing, POMDPs
- To enable smoother change in policies
- To avoid a search on every step (the max)
- To better relate to biology
We should never discount when optimizing approximate policies!

It breaks the definition of an optimal policy

With approximation, the optimal policy is no longer representable

There is no way to rank the remaining policies

Different policies will be best in different states

Instead, you must say what states you care about

Or else use average reward (which you should probably do anyway)
Model-based reinforcement learning

- Learn a model of the environment’s transition dynamics

\[ \hat{p}(r, s' | s, a) \approx p(r, s' | s, a) \]

- Use it to generate simulated trajectories
- Apply RL methods to the simulated trajectories, as if they had really happened, to learn an action-value function and policy
- Can be intermixed with direct RL
Model-based RL: GridWorld Example
Eligibility traces

- An elegant *unification* of bootstrapping and Monte Carlo (non-bootstrapping) methods

- A key algorithmic innovation that greatly *reduces computational complexity* in multi-step prediction learning; its most important advantages have nothing to do with bootstrapping or control

- Necessary to extend RL *beyond discrete time steps* that just happen to be nicely aligned with the world’s causal dynamics

- One of the *most algorithmically intense* topics in RL
  - interacts strongly with off-policy learning via *importance sampling*, and concomitant struggles to *reduce variance*
Temporal abstraction in RL

- Function approximation abstracts over state, but we need also to abstract over time.

- There are several approaches, including the framework of options—macro-actions of extended and variable duration that can nevertheless interoperate with primitive actions in DP planning methods and TD learning methods.

- The problems of temporal abstraction can be divided into three classes, increasing in difficulty:
  - representing temporal abstractions (e.g., by options)
  - learning temporal abstractions (e.g., by off-policy methods)
  - discovering temporal abstractions and selecting among them
Conclusion

- Reinforcement learning is a big topic, with a long history, an elegant theoretical core, novel algorithms, many open problems, and vast unexplored territories.

- RL can be viewed as a microcosm of the whole AI problem, including planning, acting, learning, perception, world modeling, even knowledge representation.

- Yet, even so, it can be reduced to small steps on each of which measurable progress can be made.

- RL fits well into the longest mega-trend in AI, that towards turning more of the work over to the machine.

- Realistic, ambitious, pragmatic.
Thank you for your attention

The RL&AI group at the Univ. of Alberta in 2011