Reinforcement Learning (Lecture 2)

Ron Parr
CPS 270

RL Highlights

• Everybody likes to learn from experience
• Use ML techniques to generalize from relatively small amounts of experience
• Some notable successes:
  – Backgammon
  – Flying a helicopter upside down
• Sutton’s seminal RL paper is 42nd most cited paper in computer science (Citeeseer 10/05)

Comparison w/Other Kinds of Learning

• Learning often viewed as:
  – Classification (supervised), or
  – Model learning (unsupervised)
• RL is between these (delayed signal)
• What the last thing that happens before an accident? 😱

Overview

• Review of value determination
• Motivation for RL
• Algorithms for RL
  – Overview
  – TD
  – Q-learning
  – Approximation

Recall Our Game Show

Start $100

1 correct $1,000
2 correct $10,000
2 correct $100,000

Optimal Policy w/o Cheating

V=$3,750
V=$4,166
V=$5,555
V=$11.1K

V=9/10
V=3/4
V=1/2
V=1/10

$0 $0 $0 $0

$100 $1,100 $11,100 $111,100

$100 $1,100 $11,100

$100 $1,100 $11,100

$100 $1,100 $11,100

$0 $0 $0 $0
**Cheat until you win policy**

\[
V = \begin{cases} 
\$3.7K & V = \$4.1K \\
\$90.5K & V = \$5.6K \\
\$90.6K & V = \$11.1K \\
\$90.9K & V = \$82.4K \\
\end{cases}
\]

\[
\text{w/o cheat} \quad \text{w/ cheat}
\]

\[
\frac{9}{10} \quad \frac{3}{4} \quad \frac{1}{2} \quad \frac{1}{10}
\]

\[
\$-1000 \quad \$111,100
\]

**Solving for Values**

\[
V = \gamma P V + R
\]

For moderate numbers of states we can solve this system exactly:

\[
V = (I - \gamma P)^{-1} R
\]

Guaranteed invertible because \( \gamma P \) has spectral radius < 1

---

**Iteratively Solving for Values**

\[
V = \gamma P V + R
\]

For larger numbers of states we can solve this system indirectly:

\[
V^{i+1} = \gamma P V^{i} + R
\]

Guaranteed convergent because \( \gamma P \) has spectral radius < 1 for \( \gamma < 1 \)

Convergence not guaranteed for \( \gamma = 1 \)

---

**Iterative Policy Evaluation**

\[
\frac{9}{10} \quad \frac{3}{4} \quad \frac{1}{2} \quad \frac{1}{10}
\]

\[
\$-1000 \quad \$111,100
\]

---

**Iterations Contd.**

<table>
<thead>
<tr>
<th>Iterations</th>
<th>Value</th>
<th>Value</th>
<th>Value</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>i=0</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>i=1</td>
<td>-100.00</td>
<td>-250.00</td>
<td>-500.00</td>
<td>10210.00</td>
</tr>
<tr>
<td>i=2</td>
<td>-335.00</td>
<td>-650.00</td>
<td>4555.00</td>
<td>10120.00</td>
</tr>
<tr>
<td>i=3</td>
<td>-718.50</td>
<td>3082.50</td>
<td>4392.50</td>
<td>9908.50</td>
</tr>
<tr>
<td>i=4</td>
<td>2602.40</td>
<td>2864.75</td>
<td>4095.00</td>
<td>9563.35</td>
</tr>
<tr>
<td>i=5</td>
<td>2738.52</td>
<td>3471.85</td>
<td>5582.88</td>
<td>12552.16</td>
</tr>
<tr>
<td>i=20</td>
<td>15697.49</td>
<td>16688.07</td>
<td>18396.47</td>
<td>23621.43</td>
</tr>
<tr>
<td>i=100</td>
<td>56740.99</td>
<td>57190.86</td>
<td>58074.31</td>
<td>60999.20</td>
</tr>
<tr>
<td>i=200</td>
<td>74658.96</td>
<td>74872.93</td>
<td>75999.39</td>
<td>77318.76</td>
</tr>
<tr>
<td>i=1000</td>
<td>82469.80</td>
<td>82580.93</td>
<td>82951.31</td>
<td>84432.82</td>
</tr>
<tr>
<td>i=10000</td>
<td>82470.37</td>
<td>82581.48</td>
<td>82951.85</td>
<td>84433.33</td>
</tr>
</tbody>
</table>

Note: Slow convergence b/c \( \gamma = 1 \)

---

**Overview**

- Review of value determination
- Motivation for RL
- Algorithms for RL
  - Overview
  - TD
  - Q-learning
  - Approximation
Why We Need RL

• Where do we get transition probabilities?

• How do we store them?
  • Big problems have big models
  • Model size is quadratic in state space size

• Where do we get the reward function?

RL Framework

• Learn by “trial and error”

• No assumptions about model

• No assumptions about reward function

• Assumes:
  – True state is known at all times
  – Immediate reward is known
  – Discount is known

RL Schema

• Act

• Perceive results

• Update something

• Repeat

RL for Our Game Show

• Problem: Don’t know prob of answering correctly

• Solution:
  – Buy the home version of the game
  – Practice on the home game to refine our strategy
  – Deploy strategy when we play the real game

Model Learning Approach

• Learn model, solve

• How to learn a model:
  – Take action a in state s, observe s’
  – Take action a in state s, n times
  – Observe s’ m times
  – \( P(s'|s,a) = \frac{m}{n} \)
  – Fill in transition matrix for each action
  – Compute avg. reward for each state

• Solve learned model as an MDP

Limitations of Model Learning

• Partitions learning, solution into two phases

• Model may be large (hard to visit every state lots of times)
  – Note: Can’t completely get around this problem...

• Model storage is expensive

• Model manipulation is expensive
Overview

- Review of value determination
- Motivation for RL
- Algorithms for RL
  - Overview
  - TD
  - Q-learning
  - Approximation

Temporal Difference Learning

- One of the first RL algorithms
- Learn the value of a fixed policy (no optimization; just prediction)
- Recall iterative value determination:

\[ V^{t+1}(s) = R(s, \pi(s)) + \gamma \sum_{s'} P(s'|s, \pi(s))V^t(s') \]

Problem: We don’t know this.

First Idea: Monte Carlo Sampling

- Assume that we have a black box:

\[ S \rightarrow * \rightarrow S' \]

- Count the number of times we see each \( s' \)
  - Estimate \( P(s'|s) \) for each \( s' \)
  - Essentially learns a mini-model for state \( s \)
  - Can think of as numerical integration
- Problem: The world doesn’t work this way

Next Idea

- Remember Value Determination:

\[ V^{t+1}(s) = R(s, \pi(s)) + \gamma \sum_{s'} P(s'|s, \pi(s))V^t(s') \]
- Compute an update as if the observed \( s' \) and \( r \) were the only possible outcomes:

\[ V_{\text{temp}}(s) = r + \gamma V^t(s') \]
- Make a small update in this direction:

\[ V^{t+1}(s) = (1 - \alpha)V^t(s) + \alpha V_{\text{temp}}(s) \]

\[ 0 < \alpha \leq 1 \]

Idea: Value Function Soup

Suppose: \( \alpha = 0.1 \)

Upon observing \( s' \):
- Discard 10% of soup
- Refill with \( V_{\text{temp}}(s) \)
- Stir
- Repeat

\[ V^{t+1}(s) = (1 - \alpha)V^t(s) + \alpha V_{\text{temp}}(s) \]

Example: Home Version of Game

Suppose we guess: \( V(s_3) = 15K \)
We play and get the question wrong

\[ V_{\text{temp}} = 0 \]
\[ V(s_3) = (1 - \alpha)15K + \alpha0 \]

\[
\begin{array}{c}
\$0 \\
\$100 \\
\$1,100 \\
\$11,100 \\
\$111,100 \\
\end{array}
\]
Convergence?

- Why doesn’t this oscillate?
  - e.g. consider some low probability s’ with a very high (or low) reward value

  - This could still cause a big jump in \( V(s) \)

Convergence Intuitions

- Need heavy machinery from stochastic process theory to prove convergence
- Main ideas:
  - Iterative value determination converges
  - Updates approximate value determination
  - Samples approximate expectation

\[
V_{\text{int}}(s) = R(s, \pi(s)) + \gamma \sum_{s'} P(s'| s, \pi(s)) V'(s')
\]

Ensuring Convergence

- Rewards have bounded variance
  - \( 0 \leq \gamma < 1 \)
  - Every state visited infinitely often
  - Learning rate decays so that:

\[
\sum a_i(s) = \infty
\]

\[
\sum a_i'(s) < \infty
\]

These conditions are jointly sufficient to ensure convergence in the limit with probability 1.

How Strong is This?

- Bounded variance of rewards: easy
- Discount: standard
- Visiting every state infinitely often: Hmmm…
- Learning rate: Often leads to slow learning
- Convergence in the limit: Weak
  - Hard to say anything stronger w/o knowing the mixing rate of the process
  - Mixing rate can be low; hard to know a priori

Using TD for Control

- Recall value iteration:

\[
V_{\text{int}}(s) = \max_a R(s, a) + \gamma \sum_{s'} P(s'| s, a) V'(s')
\]

- Why not pick the maximizing \( a \) and then do:

\[
V_{\text{int}}(s) = (1 - \alpha) V'(s') + \alpha V'(s')
\]

- \( s' \) is the observed next state after taking action \( a \)

Problems

- Pick the best action w/o model?
  - Must visit every state infinitely often
    - What if a good policy doesn’t do this?

- Learning is done “on policy”
  - Taking random actions to make sure that all states are visited will cause problems
### Q-Learning Overview

- Want to maintain good properties of TD
- Learns good policies and optimal value function, not just the value of a fixed policy
- Simple modification to TD that learns the optimal policy regardless of how you act! (mostly)

### Q-learning

- Recall value iteration:
  \[ V^{(i)}(s) = \max_a R(s,a) + \gamma \sum_{s'} P(s'|s,a)V^{(i)}(s') \]
- Can split this into two functions:
  \[ Q^{(i)}(s,a) = R(s,a) + \gamma \sum_{s'} P(s'|s,a)V^{(i)}(s') \]
  \[ V^{(i)}(s) = \max_a Q^{(i)}(s,a) \]

### Q-learning Properties

- Converges under same conditions as TD
- Still must visit every state infinitely often
- Separates policy you are currently following from value function learning:
  \[ Q^{\text{imp}}(s,a) = r + \gamma \max_{a'} Q^{(i)}(s',a') \]
  \[ Q^{(i)}(s,a) = (1-\alpha)Q^{(i)}(s,a) + \alpha Q^{\text{imp}}(s,a) \]

### Value Function Representation

- Fundamental problem remains unsolved:
  - TDQ learning solves model-learning problem, but
  - Large models still have large value functions
  - Too expensive to store these functions
  - Impossible to visit every state in large models
- Function approximation
  - Use machine learning methods to generalize
  - Avoid the need to visit every state

### Function Approximation

- General problem: Learn function \( f(s) \)
  - Linear regression
  - Perceptron (single layer neural network)
  - Neural networks
- Idea: Approximate \( f(s) \) with \( g(s,\theta) \)
  - \( g \) is some easily computable function of \( s \) and \( \theta \)
  - Try to find \( \theta \) that minimizes the error in \( g \)
### Linear Regression
- Define a set of basis functions (vectors) \( h_1(s), h_2(s), \ldots h_k(s) \)
- Approximate f with a weighted combination of these \( g(s) = \sum \theta h_j(s) \)
- Example: Space of quadratic functions: \( h_1(s) = 1, h_2(s) = s, h_3(s) = s^2 \)
- Orthogonal projection minimizes SSE

### Updates with Approximation
- Recall regular TD update:
  \[
  V(s, t+1) = (1 - \alpha) V(s, t) + \alpha V^{\text{temp}}(s)
  \]
- With function approximation:
  \[
  V(s) = \sum \theta h_j(s)
  \]
- Update:
  \[
  \theta_{i+1} = \theta_i + \alpha (V^{\text{temp}} - V(s)) h_j(s)
  \]

### For linear value functions
- Gradient is trivial:
  \[
  V(s, \theta) = \sum \theta h_j(s)
  \]
- Update is trivial:
  \[
  \theta_{i+1} = \theta_i + \alpha (V^{\text{temp}} - V(s, \theta)) h_j(s)
  \]

### Neural Networks
- \( s \) = input into neural network
- \( w \) = weights of neural network
- \( g(s, \theta) \) = output of network
- Try to minimize
  \[
  E = \sum (f(s) - g(s, \theta))^2
  \]
- Compute gradient of error WRT weights
  \[
  \frac{\partial E}{\partial \theta}
  \]
- Adjust to minimize error

### Combining NNs with TD
- Recall TD:
  \[
  V^{\text{temp}}(s) = R(s) + \gamma V^{\text{temp}}(s')
  \]
  \[
  V(s, t+1) = (1 - \alpha) V(s, t) + \alpha V^{\text{temp}}(s)
  \]
- Compute error function:
  \[
  E = (V(s, w) - V^{\text{temp}}(s, \theta))^2
  \]
- Update:
  \[
  \theta_{i+1} = \theta_i - \alpha \frac{\partial E}{\partial \theta}
  \]
  \[
  \theta_{i+1} = \theta_i + 2 \alpha [V^{\text{temp}}(s, \theta) - V(s, \theta)] \frac{\partial V(s, \theta)}{\partial \theta}
  \]

### Gradient-based Updates
- \[
  \theta_{i+1} = \theta_i - \alpha \frac{\partial E}{\partial \theta}
  \]
- \[
  \theta_{i+1} = \theta_i + 2 \alpha [V^{\text{temp}}(s, \theta) - V(s, \theta)] \frac{\partial V(s, \theta)}{\partial \theta}
  \]
- Equivalent to one step of backprop with \( V^{\text{temp}} \) as target
- Constant factor absorbed into learning rate
- Table-updates are a special case
- Perceptron, linear regression are special cases
Properties of approximate RL

- Table-updates are a special case
- Can be combined with Q-learning
- Convergence not guaranteed
  - Function approximators typically converge to local optimum
  - Convergence \textbf{NOT} guaranteed when combined with RL
    - Chasing a moving target
    - Errors can compound
  - Success requires very well chosen features

Other Approaches

- TD, Q-learning approximate value iteration
- Typically use parameterized \( V \)
- Can also approximate policy iteration
  - Parameterized space of policies
  - Estimate values from samples
  - Update policy parameters to improve performance

How’d They Do That???

- Helicopter (Ng et al.)
  - Approximate policy iteration
  - Constrained policy space
  - Trained on a simulator
- Backgammon (Tesaurco)
  - Predecessor: Neuro-Gammon
  - Generalize RL to alternating move games (already done by Samuel)
  - Neural network value function approximation
  - Used TD
    - Model was known
    - Action value was large
    - Exploration/Exploitation evaluation?
  - Carefully selected inputs to neural network
  - About 1.5 million games played against self

Conclusions

- Reinforcement learning solves an MDP
- Converges for exact value function representation
- Can be combined with approximation methods
- Good results require good features

Swept under the rug...

- Difficulty of finding good features
- Partial observability
- Exploration vs. Exploitation