Reinforcement Learning

CSE 473

Ever Feel Like Pavlov's Poor Dog?

Review: MDPs

$S$ = set of states set ($|S| = n$)

$A$ = set of actions ($|A| = m$)

$Pr$ = transition function $Pr(s,a,s')$

represented by set of $mn \times n$ stochastic matrices (factored into DBNs)

each defines a distribution over $S \times S$

$R(s) = $ bounded, real-valued reward fun

represented by an n-vector

Value Iteration

- Assign arbitrary values to each state (or use an admissible heuristic).

- Iterate over all states

  Improving value funct via Bellman Backups

  $V_{t+1}(s) = R(s) + \max_{a \in A} \{c(a) + \gamma \sum_{s'} \Pr(s'|a,s) V_t(s')\}$

- Stop the iteration when converges ($V_t$ approaches $V^*$ as $t \to \infty$)

Policy evaluation

- Given a policy $\Pi:S \to A$, find value of each state using this policy.

  $V^\Pi(s) = R(s) + c(\Pi(s)) + \gamma \sum_{s'} \Pr(s'|\Pi(s),s) V^\Pi(s')$

- This is a system of linear equations involving $|S|$ variables.

Bellman Backup

Policy iteration

- Start with any policy ($\Pi_0$).

- Iterate

  Policy evaluation: For each state find $V^\Pi(s)$.

  Policy improvement: For each state $s$, find action $a^*$ that maximizes $Q^\Pi(a,s)$.

  If $Q^\Pi(a^*,s) > V^\Pi(s)$ let $\Pi_{i+1}(s) = a^*$

  else let $\Pi_{i+1}(s) = \Pi_i(s)$

- Stop when $\Pi_{i+1} = \Pi_i$

- Converges faster than value iteration but policy evaluation step is more expensive.
Modified Policy iteration

- Instead of evaluating the actual value of policy by
  Solving system of linear equations, ...
- Approximate it:
  Value iteration with fixed policy.

Excuse Me...

- MDPs are great, IF...
  We know the state transition function \( P(s,a,s') \)
  We know the reward function \( R(s) \)
- But what if we don’t?
  Like the dog
  Like when we were babies...

Walking Demo

How is learning to act possible when...

- Actions have non-deterministic effects
  Which are initially unknown
- Rewards / punishments are infrequent
  Often at the end of long sequences of actions
- Learner must decide what actions to take
- World is large and complex

Naïve Approach

1. Act Randomly for a while
   (Or systematically explore all possible actions)
2. Learn
   - Transition function
   - Reward function
3. Use value iteration, policy iteration, ...

RL Techniques

- Temporal-difference learning
  Learns a utility function on states
  - Treats the difference between expected / actual
    reward as an error signal, that is propagated
    backward in time
- Exploration functions
  Balance exploration / exploitation
- Function approximation
  Compress a large state space into a small one
  Linear function approximation, neural nets, ...
  Generalization
Example:

- Suppose given policy
- Want to determine how good it is

```
1 2 3 4
1 1 1 1 1
2 1 1 1 1
3 1 1 1 1
4 1 1 1 1
```

Objective: Value Function

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.705</td>
<td>0.665</td>
<td>0.611</td>
<td>0.388</td>
</tr>
<tr>
<td>2</td>
<td>0.762</td>
<td>0.660</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0.812</td>
<td>0.868</td>
<td>0.918</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Just Like Policy Evaluation

- Except...?

Approach 1

- Direct estimation
  Estimate $U(s)$ as average total reward of epochs containing $s$ (calculating from $s$ to end of epoch)
- Pros / Cons?

  Requires huge amount of data
  doesn't exploit Bellman constraints!

  Expected utility of a state = its own reward + expected utility of successors

Passive RL

- Given policy $\pi$, estimate $U_\pi(s)$
- Not given transition matrix, nor reward function!
- Epochs: training sequences

```
(1,1)Æ(1,2)Æ(2,1)Æ(2,2)Æ(3,2)Æ(3,3)Æ(2,2)Æ(2,1)Æ(1,1)Æ(1,2)Æ(1,3)Æ(2,3)Æ(3,3)Æ(3,2)
```

Approach 2

Adaptive Dynamic Programming

- Requires fully observable environment
- Estimate transition function $M$ from training data
- Solve Bellman eqn w/ modified policy iteration

```
U^* = R(s) + \gamma \sum_{s'} M_{ss'} U^*(s')
```

Pros / Cons:

- Requires complete observations
  Don't usually need value of all states
Approach 3

• Temporal Difference Learning
  Do backups on a per-action basis
  Don't try to estimate entire transition function!
  For each transition from s to s', update:

\[
\alpha = \gamma = \frac{U^\pi(s)}{U^\pi(s) + \gamma R(s) + \gamma U^\pi(s') - U^\pi(s)}
\]

Notes

• Once U is learned, updates become 0:
  \[0 = \alpha(R(s) + \gamma U^\pi(s')) - U^\pi(s)\] when \[U^\pi(s) = R(s) + \gamma U^\pi(s')\]

• Similar to ADP
  Adjusts state to 'agree' with observed successor
  \* Not all possible successors
  Doesn't require M, model of transition function
  Intermediate approach: use M to generate "Pseudo experience"

Notes II

• "TD(0)"
  One step lookahead
  \[U^\pi(s) \leftarrow U^\pi(s) + \alpha(R(s) + \gamma U^\pi(s') - U^\pi(s))\]
  Can do 2 step, 3 step...

TD(\lambda)

• Or, ... take it to the limit!
• Compute weighted average of all future states
  \[U^\pi(s) \leftarrow U^\pi(s) + \alpha(R(s) + \gamma U^\pi(s_{next}) - U^\pi(s))\]
  becomes
  \[U^\pi(s) \leftarrow U^\pi(s) + \alpha(R(s) + \gamma(1-\lambda)\sum_{t=0}^{\infty} \lambda^t U^\pi(s_{next}) - U^\pi(s))\]
  weighted average

• Implementation
  Propagate current weighted TD onto past states
  Must memorize states visited from start of epoch

Notes III

• Online: update immediately after actions
  Works even if epochs are infinitely long

• Offline: wait until the end of an epoch
  Can perform updates in any order
  E.g. backwards in time
  Converges faster if rewards come at epoch end
  Why?!

• ADP Prioritized sweeping heuristic
  Bound # of value iteration steps (small \Delta ave)
  Only update states whose successors have \Pi\Delta
  Sample complexity \approx ADP
  Speed \approx TD

Q-Learning

• Version of TD-learning where
  instead of learning value funct on states
  we learn funct on [state,action] pairs
  \[U^\pi(s) \leftarrow U^\pi(s) + \alpha(R(s) + \gamma U^\pi(s') - U^\pi(s))\]
  becomes
  \[Q(a,s) \leftarrow Q(a,s) + \alpha(R(s) + \gamma \max Q(a',s') - Q(a,s))\]

• [Helpful for model-free policy learning]
Part II

- So far, we've assumed agent had policy
- Now, suppose agent must learn it
  While acting in uncertain world

Active Reinforcement Learning

Suppose agent must make policy while learning

First approach:
  Start with arbitrary policy
  Apply Q-Learning
  New policy:
    In state $s$,
    Choose action $a$ that maximizes $Q(a,s)$
  Problem?

Utility of Exploration

- Too easily stuck in non-optimal space
  "Exploration versus exploitation tradeoff"
- Solution 1
  With fixed probability perform a random action
- Solution 2
  Increase est expected value of infrequent states

  $U(s) \leftarrow R(s) + \gamma \max_a f(\sum_{s'} P(s' \mid a, s) U(s'), N(a, s))$

  Properties of $f(u, n)$??
  - If $n > N_e$ $U$ i.e. normal utility
  - Else, $R$: i.e. max possible reward

Part III

- Problem of large state spaces remain
  Never enough training data!
  Learning takes too long

  What to do??

Function Approximation

- Never enough training data!
  Must generalize what learning to new situations
- Idea:
  Replace large state table by a smaller, parameterized function
  Updating the value of state will change the value assigned to many other similar states

Linear Function Approximation

- Represent $U(s)$ as a weighted sum of features (basis functions) of $s$

  $\hat{U}_d(s) = \theta_1 f_1(s) + \theta_2 f_2(s) + \ldots + \theta_n f_n(s)$

- Update each parameter separately, e.g:

  $\theta_i \leftarrow \theta_i + \alpha (R(s) + \gamma \hat{U}_d(s') - \hat{U}_d(s)) \frac{\partial \hat{U}_d(s)}{\partial \theta_i}$
Example

• $U(s) = \theta_0 + \theta_1 x + \theta_2 y$
• Learns good approximation

But What If…

• $U(s) = \theta_0 + \theta_1 x + \theta_2 y + \theta_3 z$
• Computed Features:
  \[
  z = \sqrt{(x_g-x)^2 + (y_g-y)^2}
  \]

Neural Nets

• Can create powerful function approximators
  Nonlinear
  Possibly unstable
• For TD-learning, apply difference signal to neural net output and perform back-propagation

Policy Search

• Represent policy in terms of Q functions
• Gradient search
  Requires differentiability
  Stochastic policies; softmax
• Hillclimbing
  Tweak policy and evaluate by running
• Replaying experience

Helicopter Demos

Autonomous, hover

~Worlds Best Player

• Neural network with 80 hidden units
  Used computed features
• 300,000 games against self
Applications to the Web
Focused Crawling

- Limited resources
  Fetch most important pages first
- Topic specific search engines
  Only want pages which are relevant to topic
- Minimize stale pages
  Efficient re-fetch to keep index timely
  How track the rate of change for pages?

Standard Web Search Engine Architecture

- crawl the web
- check for duplicates
- extract links
- create an inverted index
- inverted index

Performance

Rennie & McCallum (ICML-99)

Methods

- Agent Types
  - Utility-based
  - Action-value based (Q function)
  - Reflex
- Passive Learning
  - Direct utility estimation
  - Adaptive dynamic programming
  - Temporal difference learning
- Active Learning
  - Choose random action 1/n of the time
  - Exploration by adding to utility function
  - Q-learning (learn action/value f directly - model free)
- Generalization
  - Function approximation (linear function or neural networks)
- Policy Search
  - Stochastic policy reg / Softmax
  - Reusing past experience

Summary

- Use reinforcement learning when
  - Model of world is unknown and/or rewards are delayed
- Temporal difference learning
  - Simple and efficient training rule
- Q-learning eliminates need for explicit T model
- Large state spaces can (sometimes) be handled
  - Function approximation, using linear functions
  - Or neural nets