Artificial Intelligence:
Representation and Problem
Solving
15-381
May 3, 2007

Reinforcement Learning 2

Announcements again

- Don’t forget: fill out FCE’s
  www.cmu.edu/fce

- Review session for final:
  Tuesday, May 8
  6:00 - 8:00 pm
  Wean 4623
Reinforcement learning so far

- Goal is the same as MDPs: discover policy that maximizes reward
- But, unlike MDPs, we don’t know: state space (world), transition model, or reward function
  - only know what state you’re in and the available actions
- Need to explore state space to discover best policy
- Model estimation:
  - do a set of trials (each that runs until a terminal state is reached)
  - estimate the rewards and a transition probabilities using the counts across trials
- Now use estimates in MDP policy estimation techniques

Evaluating value function from estimates

- Problem: how long do we wait before estimating V(s) (in order to estimating policies)?
- If we want to adapt our policy or use it for more efficient exploration, we need to have an up-to-date estimate
- Estimating V(s) is expensive if done at each iteration.
- Ideas for improvement:
  - “one backup” only update the value estimate for state s, (ie one state back) instead of all
  - Prioritized sweeping: use priority queue to update states with large potential for change
Direct estimation of the value function

- Note, that we could also use the trials to estimate the value function directly:
  - at the end of each trial, calculate the observed future reward for each state
  - get estimates for $V(s)$ for each state by averaging over trials, and multiple visits within a trial.
- Also called Monte Carlo estimation, since it samples randomly using the policy $\pi(s)$.
- Utility of each state equals its own reward + the expected utility of successor states:
  $$V(s) = R(s) + \gamma \sum_{s'} T(s, \pi(s), s') V(s')$$
- But, this is also inefficient than the other methods and learning only occurs at the end of each trial.

Temporal distance learning

- After each trial we will have noisy estimates of the true values $V(s)$
- Consider transition $(3,2) \rightarrow (3,3)$ and its two noisy estimates.
- What value do we expect for $V(3,2)$?
  $$V(3,2) = -0.04 + V(3,3)$$
  $$= -0.04 + 0.94 = 0.90$$
- The observed value is lower than the current estimate, so $V(3,3)$ is too high.
- Why don’t we adjust it to make it more consistent with our new observation?
Temporal difference learning

• For any successor state \( s' \), we can adjust \( V(s) \) so it better agrees with the constraint equations, i.e. closer to the expected future rewards.

\[
\hat{V}(s) \leftarrow (1 - \alpha) \hat{V}(s) + \alpha (R(s) + \gamma \hat{V}(s'))
\]

The TD (or temporal difference) equation

\[
\text{current value} + \text{expected future reward}
\]

The TD (or temporal difference) equation

\[
\hat{V}(s) \leftarrow \hat{V}(s) + \alpha (R(s) + \gamma \hat{V}(s') - \hat{V}(s))
\]

• Note: this doesn’t involve the transitions \( T(s,a,s') \). We only update the values \( V(s) \).

• Not quite as accurate as earlier methods, but:
  - much simpler and requires much less computation

• Why “temporal”? In terms of a state sequence:

\[
V(s_t) \leftarrow V(s_t) + \alpha [R_t + \gamma V(s_{t+1}) - V(s_t)]
\]

TD algorithm

1. initialize \( V(s) \) (arbitrarily)
2. for each trial
   3. initialize \( s \)
   4. for each step in trial
      5. observe reward \( R(s) \); take action \( a = \pi(s) \); observe next state \( s' \)
      6. \( V(s) \leftarrow V(s) + \alpha [R(s) + \gamma V(s') - V(s)] \)
   7. \( s \leftarrow s' \)
3. until \( s \) is terminal
Example: driving home (from Sutton and Barto)

<table>
<thead>
<tr>
<th>State</th>
<th>Elapsed Time (minutes)</th>
<th>Predicted Time to Go</th>
<th>Predicted Total Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>leaving office, friday at 6</td>
<td>0</td>
<td>30</td>
<td>30</td>
</tr>
<tr>
<td>reach car, raining</td>
<td>5</td>
<td>35</td>
<td>40</td>
</tr>
<tr>
<td>exiting highway</td>
<td>20</td>
<td>15</td>
<td>35</td>
</tr>
<tr>
<td>2ndary road, behind truck</td>
<td>30</td>
<td>10</td>
<td>40</td>
</tr>
<tr>
<td>entering home street</td>
<td>40</td>
<td>3</td>
<td>43</td>
</tr>
<tr>
<td>arrive home</td>
<td>43</td>
<td>0</td>
<td>43</td>
</tr>
</tbody>
</table>

- rewards are elapsed time
- you update your predicted total time at each state
- TD learning shifts each estimate toward the next estimate
- Each error is proportional to the temporal differences in the predictions

Compare TD to the direct method

- Compare the TD update equation (assuming policy \( \pi(s) \))

\[
V(s) \leftarrow V(s) + \alpha(R(s) + \gamma V(s') - V(s))
\]

\[
= (1 - \alpha)V(s) + \alpha(R(s) + \gamma V(s'))
\]

\[
= R(s) + \gamma V(s') \quad \text{for} \quad \alpha \to 1
\]

- to the direct method (assuming the same policy)

\[
V(s) = R(s) + \gamma \sum_{s'} T(s, \pi(s), s')V(s')
\]

- The TD update can be viewed as an approximation that simply chooses an update of the observed successor \( s' \), rather than in the direct method that weights all possible successor according to their probability.
- For large numbers of transitions, the two will be equal, since the successor states will occur in proportion to their transition probability.
The temporal difference equation

\[ V(s) \leftarrow V(s) + \alpha (R(s) + \gamma V(s') - V(s)) \]

- How do we choose \( \alpha \) (the learning rate)?
- It's a weighted sum, so \( 0 < \alpha < 1 \).
- Too small: \( V(s) \) will converge slowly, biased toward current estimate
- Too large: \( V(s) \) will change too quickly, biased toward new estimate

- Decaying learning rate strategy: start large when we're not sure, then reduce as we become more sure of our estimates, e.g. \( \alpha_t = \frac{1}{t} \).

\[
\sum_{t=1}^{\infty} \alpha_t = \infty \\
\sum_{t=1}^{\infty} \alpha_t^2 < \infty
\]

What about policies?

- Previously, we estimated the policy using MDP approaches, e.g. via value iteration

\[
\pi^*(s) = \arg \max_a \left[ \hat{R}(s) + \gamma \sum_{s'} \hat{T}(s, a, s') V^*(s') \right]
\]

- But, this involves estimating the transition model, which is expensive.
- Can we do it without?
Q-learning

- Idea: Learn an action-value function that estimates the expected utility of taking action $a$ in state $s$.
- Define a state-action pair: $Q(s,a)$
  - value of taking action $a$ at state $s$
  - best expected sum of future rewards after taking action $a$ at state $s$

- This is like the value function before, except with an associated action $a$:
  \[
  Q(s, a) = R(s) + \gamma \sum_{s'} T(s, a, s') \max_{a'} Q(s', a')
  \]

- The max over actions defines the best over all possible succeeding states.
- Do we need $T(s,a,s')$?
- No. Idea: try estimating $Q(s,a)$ like we estimated $V(s)$.

Q-learning

- How do we avoid estimating the model, $T(s,a,s')$?
  \[
  Q(s, a) = R(s) + \gamma \sum_{s'} T(s, a, s') \max_{a'} Q(s', a')
  \]

- A model-free update of $Q$. Like before, after transitioning from $s$, to $s'$:
  \[
  Q(s, a) \rightarrow Q(s, a) + \alpha (R(s) + \gamma \max_{a'} Q(s', a') - Q(s, a))
  \]

- The policy is estimated as in value iteration for MDPs
  \[
  \pi(s) = \arg \max_a Q(s, a)
  \]

- Guaranteed to converge the optimal policy for appropriate $\alpha$. 
Q-learning: exploration strategies

- Here we have the same exploration / exploitation dilemma as before.
- How do we choose the next action while we learning $Q(s,a)$?
- Strategies:
  - random: i.e. no learning estimate best policy at end
  - greedy: always choose best estimated action $\pi(s)$
  - $\epsilon$-greedy: choose best estimated action with probability $1-\epsilon$
  - softmax (or Boltzmann): choose best estimated action with probability:
    \[
    p \sim \frac{e^{Q(s_i,a)}}{\sum_j e^{Q(s_j,a)}}/T
    \]

Evaluation

- How do we measure how well the learning algorithm is doing?
- Ideal is to compare to optimal value:
  - $V(s) =$ value estimate at current iteration
  - $V^*(s) =$ optimal value assuming we know true model
- RMS Error is one standard evaluation (where $N = \#$ states)
  \[
  \text{Error} = \frac{1}{N} \sqrt{\sum_s [V(s) - V^*(s)]^2}
  \]
- Or simple absolute value of difference $|V - V^*|$.
- Percentage of times optimal action was chosen is another.
Learning curves

• Constant learning rate

\[ \alpha = 0.001 \]

Data from Rohit & Vivek, 2005

\[ \alpha = 0.1 \]

Data from Rohit & Vivek, 2005

Learning curves

• Decaying learning rate

\[ \alpha = \frac{K}{K + \text{iteration #}} \]

Data from Rohit & Vivek, 2005
Learning curves

• Changing environments

Data from Rohit & Vivek, 2005

Learning curves

• Adaptive learning rate with changing environments

Data from Rohit & Vivek, 2005
Approximating utility functions for large state spaces

- For small state spaces we could keep tables.
- What do we do with large spaces?
  - How many states for chess?
  - How many for a real-world environment?
- Infeasible to use tables.
- Moreover, it wouldn’t generalize even if you did have sufficient data.
- For most large problems: large not all states are independent.
- Want to learn structure of state space and how it maps to the value function.
- That is, we want generalization.

Approximating state-value mappings

- If we can approximate large state spaces well, then we can apply RL to much larger and more complex problems.
- What is the learning objective?
- Common: minimize the mean squared error between true and predicted values:
  \[ E(\theta) = \sum_s p(s)[V(s) - f(s, \theta)]^2 \]
- Then we can apply optimization strategies, e.g. gradient descent:
  \[ \theta_{t+1} = \theta_t + \alpha \frac{\partial E(\theta)}{\partial \theta} \]
  \[ = \frac{\partial}{\partial \theta} \sum_s p(s)[V(s) - f(s, \theta)]^2 \]
  \[ = \theta_t + \alpha [V(s) - f(s, \theta)] \frac{\partial}{\partial \theta} f(s, \theta) \]

The function \( f() \) can be anything:
- decision trees
- polynomials
- neural networks
- cluster models

Whatever best fits the data.
Case study: TD-Gammon (Tesauro)

- The state space: number of black and white pieces at each location.
- That's \( \sim 10^{20} \) states!
- Branching factor \( \sim 400 \); prohibits direct search.
- Treat as RL problem:
  - actions are the set of legal moves.
- But table estimation is impossible: need to approximate state-value mapping.
  - Idea: use a neural net.

By playing random games (and both sides), it teaches itself!

Contrast that with Tesauro's previous effort, Neurogammon:
- train neural network on a large dataset of exemplary moves.
- NN is limited by number of training examples.

Amazing result:
- TD-Gammon can learn to play very well just starting from a random policy.
- Just by moving randomly and observing what resulted in a win or loss.
- Initial games lasted thousands of moves until a random win.

<table>
<thead>
<tr>
<th>Program</th>
<th>Hidden Units</th>
<th>Training Games</th>
<th>Opponents</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>TD-Gam 0.0</td>
<td>40</td>
<td>300,000</td>
<td>other programs</td>
<td>tied for best</td>
</tr>
<tr>
<td>TD-Gam 1.0</td>
<td>60</td>
<td>300,000</td>
<td>Robertie, Magriel,...</td>
<td>-13 pts / 51 games</td>
</tr>
<tr>
<td>TD-Gam 2.0</td>
<td>40</td>
<td>800,000</td>
<td>various Grandmasters</td>
<td>-7 pts / 38 games</td>
</tr>
<tr>
<td>TD-Gam 2.1</td>
<td>80</td>
<td>1,500,000</td>
<td>Robertie</td>
<td>-1 pt / 40 games</td>
</tr>
<tr>
<td>TD-Gam 3.0</td>
<td>80</td>
<td>1,500,000</td>
<td>Kazaros</td>
<td>+6 pts / 20 games</td>
</tr>
</tbody>
</table>
Many other successful applications of RL

- checker's (Samuel, 59)
- elevator dispatcher (Crites et al, 95)
- inventory management (Bertsekas et al 95)
- many pole balancing and acrobatic challenges
- robotic manipulation
- path planning
- ... and many others.

Announcements again

- Don’t forget: fill out FCE’s
  www.cmu.edu/fce

- Review session for final:
  Tuesday, May 8
  6:00 - 8:00 pm
  Wean 4623