Lecture 8: Reinforcement Learning
10/26/2010

Luke Zettlemoyer

Many slides over the course adapted from either Dan Klein, Stuart Russell or Andrew Moore
Outline

- Reinforcement Learning
  - Passive Learning (review)
  - TD Updates (review)
  - Q-value iteration
  - Q-learning
  - Linear function approximation
Announcements

- PS1 grades are out
- PS2 due today!
- PS3 will go out tomorrow
  - MDPs and RL - should finish all of content in lecture today
Recap: MDPs

- **Markov decision processes:**
  - States $S$
  - Actions $A$
  - Transitions $P(s'|s,a)$ (or $T(s,a,s')$)
  - Rewards $R(s,a,s')$ (and discount $\gamma$)
  - Start state $s_0$

- **Quantities:**
  - Policy = map of states to actions
  - Utility = sum of discounted rewards
  - Values = expected future utility from a state
  - Q-Values = expected future utility from a q-state
Recap: Value Iteration

- **Idea:**
  - Start with $V_0^*(s) = 0$, which we know is right (why?)
  - Given $V_i^*$, calculate the values for all states for depth $i+1$:

  $$V_{i+1}(s) \leftarrow \max_a \sum_{s'} T(s, a, s') \left[ R(s, a, s') + \gamma V_i(s') \right]$$

  - Throw out old vector $V_i^*$
  - Repeat until convergence
  - This is called a **value update** or **Bellman update**

- **Theorem:** will converge to unique optimal values
  - Basic idea: approximations get refined towards optimal values
  - Policy may converge long before values do
Recap: Policy Iteration

- Problem with value iteration:
  - Considering all actions each iteration is slow: takes $|A|$ times longer than policy evaluation
  - But policy doesn’t change each iteration, time wasted

- Alternative to value iteration:
  - Step 1: Policy evaluation: calculate utilities for a fixed policy (not optimal utilities!) until convergence (fast)
  - Step 2: Policy improvement: update policy using one-step lookahead with resulting converged (but not optimal!) utilities (slow but infrequent)
  - Repeat steps until policy converges
What is it doing?

Step Delay: 0.10000
Discount: 0.800
Epsilon: 0.500
Learning Rate: 0.800

Step: 75
Position: 63
Velocity: -6.04
100-step Avg Velocity: 0.68
Recap: Reinforcement Learning

- Reinforcement learning:
  - Still have an MDP:
    - A set of states \( s \in S \)
    - A set of actions (per state) \( A \)
    - A model \( T(s,a,s') \)
    - A reward function \( R(s,a,s') \)
  - Still looking for a policy \( \pi(s) \)
  - New twist: don’t know \( T \) or \( R \)
    - I.e. don’t know which states are good or what the actions do
    - Must actually try actions and states out to learn
Recap: Passive Learning

- **Simplified task**
  - You don’t know the transitions $T(s,a,s')$
  - You don’t know the rewards $R(s,a,s')$
  - You are given a policy $\pi(s)$
  - **Goal: learn the state values** (and maybe the model)
  - I.e., policy evaluation

- **In this case:**
  - Learner “along for the ride”
  - No choice about what actions to take
  - Just execute the policy and learn from experience
  - We’ll get to the active case soon
  - This is NOT offline planning!
Recap: Sampling Expectations

- Want to compute an expectation weighted by \( P(x) \):

\[
E[f(x)] = \sum_x P(x) f(x)
\]

- Model-based: estimate \( P(x) \) from samples, compute expectation

\[
x_i \sim P(x) \\
\hat{P}(x) = \text{count}(x)/k \\
E[f(x)] \approx \sum_x \hat{P}(x) f(x)
\]

- Model-free: estimate expectation directly from samples

\[
x_i \sim P(x) \\
E[f(x)] \approx \frac{1}{k} \sum_i f(x_i)
\]

- Why does this work? Because samples appear with the right frequencies!
Recap: Model-Based Learning

- **Idea:**
  - Learn the model empirically (rather than values)
  - Solve the MDP as if the learned model were correct
  - Better than direct estimation?

- **Empirical model learning**
  - Simplest case:
    - Count outcomes for each s,a
    - Normalize to give estimate of $T(s,a,s')$
    - Discover $R(s,a,s')$ the first time we experience $(s,a,s')$
  - More complex learners are possible (e.g. if we know that all squares have related action outcomes, e.g. “stationary noise”
Recap: Model-Free Learning

- Big idea: why bother learning $T$?
  - Update $V$ each time we experience a transition
- Temporal difference learning (TD)
  - Policy still fixed!
  - Move values toward value of whatever successor occurs: running average!

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

$sample = R(s, \pi(s), s') + \gamma V^\pi(s')$

$$V^\pi(s) \leftarrow (1 - \alpha)V^\pi(s) + (\alpha) sample$$

$$V^\pi(s) \leftarrow V^\pi(s) + \alpha(sample - V^\pi(s))$$
Example: TD Policy Evaluation

\[ V^\pi(s) \leftarrow (1 - \alpha)V^\pi(s) + \alpha \left[ R(s, \pi(s), s') + \gamma V^\pi(s') \right] \]

(1,1) up -1  (1,1) up -1
(1,2) up -1  (1,2) up -1
(1,3) right -1 (1,3) right -1
(2,3) right -1 (3,3) right -1
(3,3) right -1 (3,2) up -1
(3,2) up -1  (4,2) exit -100
(3,3) right -1  (done)
(4,3) exit +100
(done)

Take \( \gamma = 1, \alpha = 0.5 \)

- TD value leaning is model-free for policy evaluation (passive learning)

- However, if we want to turn our value estimates into a policy, we’re sunk:

\[
\pi(s) = \arg \max_a Q^*(s, a)
\]

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

- Idea: learn Q-values directly
- Makes action selection model-free too!
Active Learning

- **Full reinforcement learning**
  - You don’t know the transitions $T(s,a,s')$
  - You don’t know the rewards $R(s,a,s')$
  - You can choose any actions you like
  - **Goal:** learn the optimal policy
  - … what value iteration did!

- **In this case:**
  - Learner makes choices!
  - Fundamental tradeoff: exploration vs. exploitation
  - This is NOT offline planning! You actually take actions in the world and find out what happens…
Value iteration: find successive approx optimal values
- Start with $V_0^*(s) = 0$
- Given $V_i^*$, calculate the values for all states for depth $i+1$:

$$V_{i+1}(s) \leftarrow \max_a \sum_{s'} T(s, a, s') \left[ R(s, a, s') + \gamma V_i(s') \right]$$

But Q-values are more useful!
- Start with $Q_0^*(s,a) = 0$
- Given $Q_i^*$, calculate the q-values for all q-states for depth $i+1$:

$$Q_{i+1}(s, a) \leftarrow \sum_{s'} T(s, a, s') \left[ R(s, a, s') + \gamma \max_{a'} Q_i(s', a') \right]$$
Q-Learning Update

- Q-Learning: sample-based Q-value iteration

\[ Q^*(s, a) = \sum_{s'} T(s, a, s') \left[ R(s, a, s') + \gamma \max_{a'} Q^*(s', a') \right] \]

- Learn \( Q^*(s,a) \) values
  - Receive a sample \((s,a,s',r)\)
  - Consider your old estimate: \( Q(s, a) \)
  - Consider your new sample estimate:

\[ \text{sample} = R(s, a, s') + \gamma \max_{a'} Q(s', a') \]

  - Incorporate the new estimate into a running average:

\[ Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + (\alpha) [\text{sample}] \]
Q-Learning: Fixed Policy
Exploration / Exploitation

- Several schemes for action selection
  - Simplest: random actions ($\epsilon$ greedy)
    - Every time step, flip a coin
    - With probability $\epsilon$, act randomly
    - With probability $1-\epsilon$, act according to current policy
  - Problems with random actions?
    - You do explore the space, but keep thrashing around once learning is done
    - One solution: lower $\epsilon$ over time
    - Another solution: exploration functions
Q-Learning: $\epsilon$ Greedy
Exploration Functions

- **When to explore**
  - Random actions: explore a fixed amount
  - Better idea: explore areas whose badness is not (yet) established

- **Exploration function**
  - Takes a value estimate and a count, and returns an optimistic utility, e.g. $f(u, n) = u + k/n$ (exact form not important)
  - Exploration policy $\pi(s') = \max_{a'} Q_i(s', a')$ vs. $\max_{a'} f(Q_i(s', a'), N(s', a'))$
Q-Learning Final Solution

- Q-learning produces tables of q-values:

\[ \begin{array}{cccc}
0.59 & 0.66 & 0.64 & 1.00 \\
0.58 & 0.60 & 0.68 & 0.88 \\
0.54 & 0.66 & 0.58 & -0.51 \\
0.61 & 0.61 & -0.51 & -1.00 \\
0.52 & 0.52 & -0.43 & -0.89 \\
0.49 & 0.42 & 0.15 & -0.38 \\
0.54 & 0.40 & 0.31 & 0.28 \\
0.45 & 0.33 & 0.28 & 0.11 \\
0.47 & 0.42 & 0.27 & 0.11 \\
\end{array} \]

Q-VALUES AFTER 1000 EPISODES
Q-Learning Properties

- Amazing result: Q-learning converges to optimal policy
  - If you explore enough
  - If you make the learning rate small enough
  - ... but not decrease it too quickly!
  - Not too sensitive to how you select actions (!)

- Neat property: off-policy learning
  - learn optimal policy without following it (some caveats)
Q-Learning

- In realistic situations, we cannot possibly learn about every single state!
  - Too many states to visit them all in training
  - Too many states to hold the q-tables in memory

- Instead, we want to generalize:
  - Learn about some small number of training states from experience
  - Generalize that experience to new, similar states
  - This is a fundamental idea in machine learning, and we’ll see it over and over again
Example: Pacman

- Let’s say we discover through experience that this state is bad:

- In naïve q learning, we know nothing about related states and their q values:

- Or even this third one!
Feature-Based Representations

- **Solution:** describe a state using a vector of features (properties)
  - Features are functions from states to real numbers (often 0/1) that capture important properties of the state
  - Example features:
    - Distance to closest ghost
    - Distance to closest dot
    - Number of ghosts
    - $1 / (\text{dist to dot})^2$
    - Is Pacman in a tunnel? (0/1)
    - …… etc.
    - Is it the exact state on this slide?
- Can also describe a q-state $(s, a)$ with features (e.g. action moves closer to food)
Linear Feature Functions

- Using a feature representation, we can write a q function (or value function) for any state using a few weights:

\[ V(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s) \]

\[ Q(s, a) = w_1 f_1(s, a) + w_2 f_2(s, a) + \ldots + w_n f_n(s, a) \]

- Advantage: our experience is summed up in a few powerful numbers

- Disadvantage: states may share features but actually be very different in value!
Function Approximation

\[ Q(s, a) = w_1 f_1(s, a) + w_2 f_2(s, a) + \ldots + w_n f_n(s, a) \]

- **Q-learning with linear q-functions:**
  
  \[ \text{transition} = (s, a, r, s') \]
  
  \[ \text{difference} = \left[ r + \gamma \max_{a'} Q(s', a') \right] - Q(s, a) \]
  
  \[ Q(s, a) \leftarrow Q(s, a) + \alpha \text{[difference]} \]

  \[ w_i \leftarrow w_i + \alpha \text{[difference]} f_i(s, a) \]

  - **Exact Q’s**
  - **Approximate Q’s**

- **Intuitive interpretation:**
  - Adjust weights of active features
  - E.g. if something unexpectedly bad happens, disprefer all states with that state’s features

- **Formal justification:** online least squares
Example: Q-Pacman

\[ Q(s, a) = 4.0 f_{DOT}(s, a) - 1.0 f_{GST}(s, a) \]

\[ f_{DOT}(s, \text{NORTH}) = 0.5 \]

\[ f_{GST}(s, \text{NORTH}) = 1.0 \]

\[ Q(s, a) = +1 \]

\[ R(s, a, s') = -500 \]

\[ \text{correction} = -501 \]

\[ w_{DOT} \leftarrow 4.0 + \alpha [-501] 0.5 \]

\[ w_{GST} \leftarrow -1.0 + \alpha [-501] 1.0 \]

\[ Q(s, a) = 3.0 f_{DOT}(s, a) - 3.0 f_{GST}(s, a) \]
**Linear Regression**

Prediction

\[ \hat{y} = w_0 + w_1 f_1(x) \]

Prediction

\[ \hat{y}_i = w_0 + w_1 f_1(x) + w_2 f_2(x) \]
Ordinary Least Squares (OLS)

\[
\text{total error} = \sum_i (y_i - \hat{y}_i)^2 = \sum_i \left( y_i - \sum_k w_k f_k(x_i) \right)^2
\]
Minimizing Error

Imagine we had only one point \( x \) with features \( f(x) \):

\[
\text{error}(w) = \frac{1}{2} \left( y - \sum_k w_k f_k(x) \right)^2
\]

\[
\frac{\partial \text{error}(w)}{\partial w_m} = -\left( y - \sum_k w_k f_k(x) \right) f_m(x)
\]

\[
w_m \leftarrow w_m + \alpha \left( y - \sum_k w_k f_k(x) \right) f_m(x)
\]

Approximate q update:

“target” \quad “prediction”

\[
w_m \leftarrow w_m + \alpha \left[ r + \gamma \max_a Q(s', a') - Q(s, a) \right] f_m(s, a)
\]
Degree 15 polynomial
Which Algorithm?

Q-learning, no features, 50 learning trials:
Which Algorithm?

Q-learning, no features, 1000 learning trials:

SCORE: 0
Which Algorithm?

Q-learning, simple features, 50 learning trials:
Policy Search*
Policy Search*

- Problem: often the feature-based policies that work well aren’t the ones that approximate $V / Q$ best
  - E.g. your value functions from project 2 were probably horrible estimates of future rewards, but they still produced good decisions
  - We’ll see this distinction between modeling and prediction again later in the course

- Solution: learn the policy that maximizes rewards rather than the value that predicts rewards

- This is the idea behind policy search, such as what controlled the upside-down helicopter
Policy Search*

- **Simplest policy search:**
  - Start with an initial linear value function or q-function
  - Nudge each feature weight up and down and see if your policy is better than before

- **Problems:**
  - How do we tell the policy got better?
  - Need to run many sample episodes!
  - If there are a lot of features, this can be impractical
Advanced policy search:

- Write a stochastic (soft) policy:

\[ \pi_w(s) \propto e^{\sum_i w_i f_i(s,a)} \]

- Turns out you can efficiently approximate the derivative of the returns with respect to the parameters \( w \) (details in the book, optional material)

- Take uphill steps, recalculate derivatives, etc.
Review: MDPs

- **Markov decision processes:**
  - States $S$
  - Actions $A$
  - Transitions $P(s'|s,a)$ (or $T(s,a,s')$)
  - Rewards $R(s,a,s')$ (and discount $\gamma$)
  - Start state dist. $b_0$
Partially observable MDPs

- **Markov decision processes:**
  - States $S$
  - Actions $A$
  - Transitions $P(s'|s,a)$ (or $T(s,a,s')$)
  - Rewards $R(s,a,s')$ (and discount $\gamma$)
  - Start state distribution $b_0 = P(s_0)$

- **POMDPs, just add:**
  - Observations $O$
  - Observation model $P(o|s,a)$ (or $O(s,a,o)$)
A POMDP: Ghost Hunter
POMDP Computations

- **Sufficient statistic: belief states**
  - \( b_o = \Pr(s_o) \)
  - \( b'(s') = \Pr(s' | o, a, b) \)
  
  \[
  b'(s') = \frac{O(s', a, o) \sum_{s \in S} T(s, a, s') b(s)}{\Pr(o | a, b)}
  \]

- **POMDPs search trees**
  - max nodes are belief states
  - expectation nodes branch on possible observations
  - (this is motivational; we will not discuss in detail)