Reinforcement Learning

What is it doing?

Reinforcement Learning

Outline

- Today
  - Passive Learning
  - TD Updates
  - Q-value iteration
  - Q-learning
- Next time
  - Linear function approximation

Basic idea:

- Receive feedback in the form of rewards
- Agent's utility is defined by the reward function
- Must learn to act so as to maximize expected rewards

Reinforcement Learning

CIS 471: Artificial Intelligence
Fall 2012

Reinforcement Learning
Daniel Lowd

Many slides over the course adapted from Luke Zettlemoyer, Dan Klein, Stuart Russell and/or Andrew Moore
**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

**Example: Backgammon**

- Reward only for win / lose in terminal states, zero otherwise
- TD-Gammon learns a function approximation to $V(s)$ using a neural network
- Combined with depth 3 search, one of the top 3 players in the world
- You could imagine training Pacman this way…
- … but it’s tricky! (It’s also P3)

**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!

**Detour: 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
  \[ x_i \sim P(x) \]
  \[ \hat{P}(x) = \text{count}(x)/k \]
  \[ E[f(x)] \approx \frac{1}{k} \sum_i f(x_i) \]
- 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!

**Simple Case: Direct Estimation**

- Episodes:
  - (1,1) up -1
  - (1,2) up -1
  - (1,3) right -1
  - (2,3) right -1
  - (3,3) right -1
  - (3,2) up -1
  - (2,3) up -100
  - (3,3) right -1
  - (4,3) exit +100
- $\gamma = 1$, $R = -1$
- $V(2,3) = (96 + -103) / 2 = -3.5$
- $V(3,3) = (99 + 97 + -102) / 3 = 31.3$

**Review: Model-Based Policy Evaluation**

- Simplified Bellman updates to calculate $V$ for a fixed policy:
  - New $V$ is expected one-step-look-ahead using current $V$
  - Unfortunately, need $T$ and $R$
  - $V_0^\pi(s) = 0$
  - $V_{i+1}^\pi(s) = \sum_{s'} T(s, \pi(s), s') \left[ R(s, \pi(s), s') + \gamma V_i^\pi(s') \right]$
Model-Based Learning

- Idea:
  - Learn the model empirically (rather than values)
  - Solve the MDP as if the learned model were correct
- Empirical model learning
  - Simplest case:
    - Count outcomes for each s,a
    - Normalize to give estimate of $T(s,a,s')$
  - More complex learners are possible (e.g. if we know that all squares have related action outcomes, e.g. "stationary noise")

\[
V_{n+1}(s) \leftarrow \sum_{s'} T(s, \pi(s), s') [R(s, \pi(s), s') + \gamma V_n(s')] 
\]

Sample Avg to Replace Expectation?

- Who needs T and R? Approximate the expectation with samples (drawn from T!)

\[
V_{n+1}(s) \leftarrow \sum_{s'} T(s, \pi(s), s') [R(s, \pi(s), s') + \gamma V_n(s')] 
\]

Temporal Difference Learning

- Big idea: learn from every experience
  - Update $V$ each time we experience a transition
  - Likely $s'$ will contribute updates more often
- Temporal difference learning (TD)
  - Policy still fixed!
  - Move values toward value of whatever successor occurs: running average!

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

Exponential Moving Average

- Exponential moving average
  - The running update:
    \[
    \tilde{x}_n = (1 - \alpha) \cdot \tilde{x}_{n-1} + \alpha \cdot x_n 
    \]
  - Makes recent samples more important
  - Forgets about the past (distant past values were wrong anyway)
  - Easy to compute from the running average
  - Decreasing learning rate can give converging averages

\[
\tilde{x}_n = x_n + (1 - \alpha) \cdot \tilde{x}_{n-1} + (1 - \alpha)^2 \cdot \tilde{x}_{n-2} + \ldots 
\]

TD Policy Evaluation

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

Example: Model-Based Learning

- Episodes:
  - (1,1) up -1
  - (1,2) up -1
  - (1,3) right -1
  - (3,3) right -1
  - (4,3) exit +100
- $\gamma = 1$

\[
T(<3,3>, \text{right}, <4,3>) = 1 / 3 
\]

\[
T(<2,3>, \text{right}, <3,3>) = 2 / 2 
\]

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

Updates for $V(<3,3>)$:

\[
V(<3,3>) = 0.5 \cdot V(<3,3>) + 0.5 \cdot [1 + 1 - 100] = 0.5 
\]

\[
V(<3,3>) = 0.5 \cdot V(<3,3>) + 0.5 \cdot [0.5 \cdot 1 + 1 - 100] = 49.475 
\]

\[
V(<3,3>) = 0.5 \cdot V(<3,3>) + 0.5 \cdot [0.5 \cdot 1 + 1 - 0.75] 
\]

Take $\gamma = 1$, $\alpha = 0.5$, $V_0(<4,3>) = 100$, $V_0(<4,2>) = -100$, $V_0 = 0$ otherwise
Problems with TD Value Learning

- TD value learning 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 Reinforcement 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…

Detour: Q-Value Iteration

- 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

- Several schemes for action selection
  - Simplest: random actions (\( \varepsilon \)-greedy)
    - Every time step, flip a coin
    - With probability \( \varepsilon \), act randomly
    - With probability \( 1-\varepsilon \), 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 \( \varepsilon \) over time
    - Another solution: exploration functions
Q-Learning: $\varepsilon$ 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. $f(u, n) = u + \frac{k}{n}$ (exact form not important)
  - Exploration policy $\pi(s') = \max_a Q(s', a')$ vs. $\max_a f(Q(s', a'), N(s', a'))$

---

Q-Learning Final Solution

- Q-learning produces tables of $q$-values:

---

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 naive 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/(dist to dot)
  - Is Pacman in a tunnel? (0/1)
  - … etc.
- 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:

\[ \text{transition} = (s, a, r, s') \]

\[ \text{difference} = [r + \gamma \max_{a'} Q(s', a')] - Q(s, a) \]

\[ Q(s, a) \leftarrow Q(s, a) + \alpha \text{ [difference]} f(s, a) \]

- 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

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

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

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

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

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

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

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

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

\[ Q(s, a) = 3.0 f_{\text{DOT}}(s, a) - 3.0 f_{\text{GHOST}}(s, a) \]

Linear Regression

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

\[ \hat{y}_i = w_0 + w_1 f_1(x_i) + w_2 f_2(x_i) \]

Ordinary Least Squares (OLS)

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

Observation \( y \)

Prediction \( \hat{y} \)

Error or ‘residual’
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:

$$
w_m \leftarrow w_m + \alpha \left[ r + \gamma \max_{a'} Q(s', a') - Q(s, a) \right] f_m(s, a)
$$

Overfitting

Which Algorithm?

Q-learning, no features, 50 learning trials:

| SCORE: 0 |

Which Algorithm?

Q-learning, no features, 1000 learning trials:

| SCORE: 0 |

Which Algorithm?

Q-learning, simple features, 50 learning trials:

| SCORE: 0 |

Policy Search*

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

Policy Search*

- 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.