Reinforcement Learning

Ron Parr
CompSci 370
Department of Computer Science
Duke University

With thanks to Kris Hauser for some content

RL Highlights

- Everybody likes to learn from experience
- Use ML techniques to generalize from *relatively small amounts* of experience
- Some notable successes:
  - Backgammon, Go, Starcraft
  - Flying a helicopter upside down
  - Dogfighting in realistic simulators
  - Atari Games

From Andrew Ng's home page

- Sutton & Barto RL Book is one of the most cited references in CS (~46K citations as of 9/21)

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?

Source: By Damnsoft 09 at English Wikipedia, CC BY 3.0, https://commons.wikimedia.org/w/index.php?curid=11802152

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 for Our Game Show

- Problem: We don’t know probability of answering correctly

Source: Wikipedia page
For “Who Wants to be a Millionaire”

- 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) = m/n
  - Fill in transition matrix for each action
  - Compute avg. reward for each state
- Solve learned model as an MDP (previous lecture)

Limitations of Model Learning

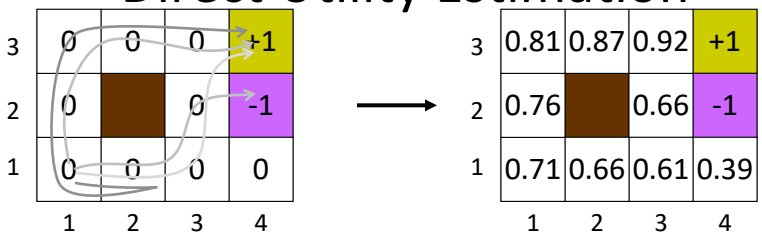
- Partitions learning, solution into two phases
- Model may be large
  - Hard to visit every state lots of times
  - Note: Can't completely get around this problem…
- Model storage is expensive
- Model manipulation is expensive

First steps: Passive RL

- Observe execution **trials** of an agent that acts according to some unobserved policy π
- Problem: estimate the value function V^{π}

- Important alternate view of $V^{\pi}(s)$ calculation
  - Recall $V^{\pi}(s)$ is the ***expected, discounted value*** of following policy π from state s
  - $V^{\pi}(s) = E\Sigma_{St}[\gamma^t R(S_t)]$ where S_t is the random variable denoting the distribution of states at time t]

Direct Utility Estimation



1. Observe trials $t^{(i)}=(s_0^{(i)},a_1^{(i)},s_1^{(i)},r_1^{(i)},\ldots,a_{ti}^{(i)},s_{ti}^{(i)},r_{ti}^{(i)})$ for i=1,…,n
2. For each state $s \in S$:
   3. Find all trials $t^{(i)}$ that pass through s at, e.g., time step k
   4. Compute subsequent value $V^{t(i)}(s)=\Sigma_{t=k \text{ to } ti}\gamma^{t-k} r_t^{(i)}$
   5. Set $V^{\pi}(s)$ to the average observed values

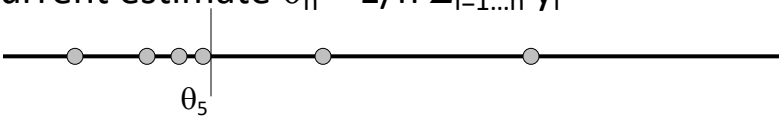
Limitations: Clunky, learns only when an end state is reached

Incremental ("Online") Function Learning

- Data is streaming into learner
  $x_1, y_1, \ldots, x_n, y_n \quad y_i = f(x_i)$
- Observes x_{n+1} and must make prediction for next time step y_{n+1}
- "Batch" approach:
  - Store **all data** at step n
  - Use your learner of choice on all data up to time n, predict for time n+1
- Can we be more efficient? (space & memory)

Example: Mean Estimation

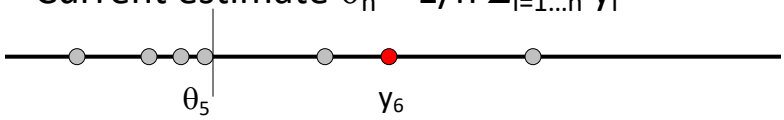
- $y_i = \theta$ + error term (constant - no x's)
- Current estimate $\theta_n = 1/n \, \Sigma_{i=1\ldots n} \, y_i$



- $\theta_{n+1} = 1/(n+1) \, \Sigma_{i=1\ldots n+1} \, y_i$
  $= 1/(n+1) \, (y_{n+1} + \Sigma_{i=1\ldots n} \, y_i)$
  $= 1/(n+1) \, (y_{n+1} + n \, \theta_n)$
  $= 1/(n+1) \, (y_{n+1} + (n+1) \, \theta_n - \theta_n)$
  $= \theta_n + 1/(n+1) \, (y_{n+1} - \theta_n)$

Example: Mean Estimation

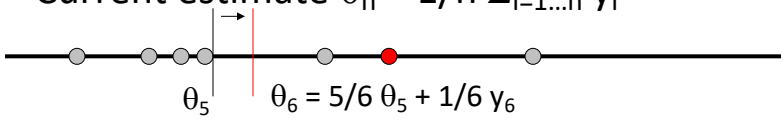
- $y_i = \theta$ + error term (constant - no x's)
- Current estimate $\theta_n = 1/n \, \Sigma_{i=1\ldots n} \, y_i$



- $\theta_{n+1} = 1/(n+1) \, \Sigma_{i=1\ldots n+1} \, y_i$
  $= 1/(n+1) \, (y_{n+1} + \Sigma_{i=1\ldots n} \, y_i)$
  $= 1/(n+1) \, (y_{n+1} + n \, \theta_n)$
  $= 1/(n+1) \, (y_{n+1} + (n+1) \, \theta_n - \theta_n)$
  $= \theta_n + 1/(n+1) \, (y_{n+1} - \theta_n)$

Example: Mean Estimation

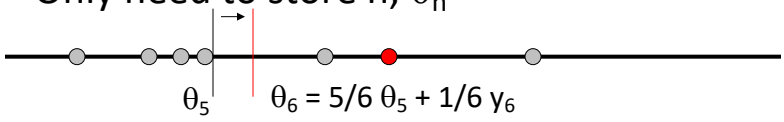
- $y_i = \theta$ + error term (constant - no x's)
- Current estimate $\theta_n = 1/n \, \Sigma_{i=1\ldots n} \, y_i$



- $\theta_{n+1} = 1/(n+1) \, \Sigma_{i=1\ldots n+1} \, y_i$
  $= 1/(n+1) \, (y_{n+1} + \Sigma_{i=1\ldots n} \, y_i)$
  $= 1/(n+1) \, (y_{n+1} + n \, \theta_n)$
  $= 1/(n+1) \, (y_{n+1} + (n+1) \, \theta_n - \theta_n)$
  $= \theta_n + 1/(n+1) \, (y_{n+1} - \theta_n)$

Example: Mean Estimation

- $\theta_{n+1} = \theta_n + 1/(n+1)\ (y_{n+1} - \theta_n)$
- Only need to store n, θ_n



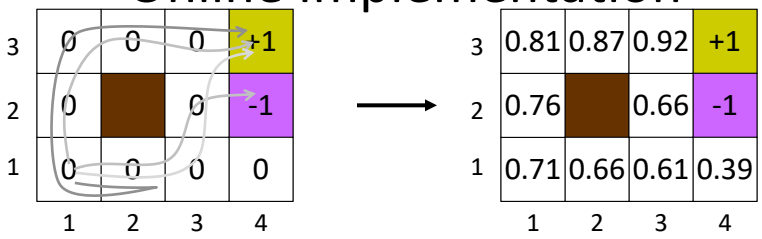
Learning Rates

- In fact, $\theta_{n+1} = \theta_n + \alpha_n\ (y_{n+1} - \theta_n)$ converges to the mean for any α_n such that:
  - $\alpha_n \rightarrow 0$ as $n \rightarrow \infty$
  - $\Sigma\alpha_n \rightarrow \infty$
  - $\Sigma\alpha_n^2 \rightarrow C < \infty$
- O(1/n) does the trick
- If α_n is close to 1, then the estimate shifts strongly to recent data; close to 0, and the old estimate is preserved

Learning Rates in RL in Practice

- Maintain a per-state count N[s]
- Learning rate is function of N[s], α(N[s])
- Sufficient to satisfy theory: α(N[s])=1/N(s)
- Often viewed as too slow
  - α drops quickly
  - Convergence is slow
- In practice, often a floor on, α, e.g., α = 0.01
- Floor leads to faster learning, but less stability

Online Implementation



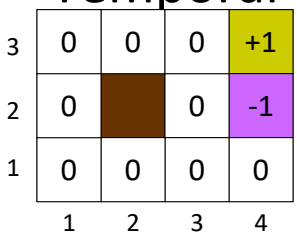
1. Store counts N[s] and estimated values $V^{\pi}(s)$ (initialize to 0, typically)
2. After a trial t, for each state s in the trial:
3. Set N[s] ← N[s]+1
4. Adjust value $V^{\pi}(s) \leftarrow V^{\pi}(s)+\alpha(N[s])(V^{t}(s)-V^{\pi}(s))$

Value of s in trial t (from discounted sum)

$\alpha(N[s])=1/N(s)$

- Doesn't require storing all trajectories, but…
  - Simple averaging
  - Slow learning, because Bellman equation is not used to pass knowledge between adjacent states

Temporal Difference Learning



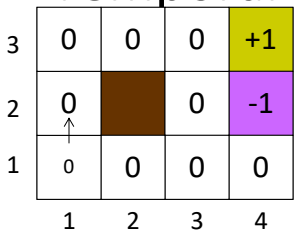
$$V_{t+1}(s) = R(s) + \gamma \sum_{s' \in Succ(s,a)} P(s'|s,a) V_t(s')$$

Online estimation of mean over value next states

1. Store counts N[s] and estimated values $V^{\pi}(s)$
2. For each observed transition (s,r,a,s'):
   3. Set N[s] ← N[s]+1
   4. Adjust value $V^{\pi}(s) \leftarrow V^{\pi}(s)+\alpha(N[s])(r+\gamma V^{\pi}(s')-V^{\pi}(s))$

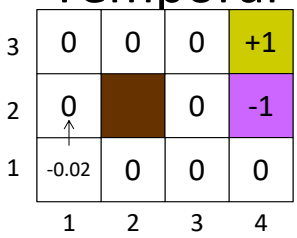
- Instead of averaging at the level of trajectories…
- Average at the level of states

Temporal Difference Learning



1. Store counts N[s] and estimated values $V^{\pi}(s)$
2. For each observed transition (s,r,a,s'):
   3. Set N[s] ← N[s]+1
   4. Adjust value $V^{\pi}(s) \leftarrow V^{\pi}(s)+\alpha(N[s])(r+\gamma V^{\pi}(s')-V^{\pi}(s))$

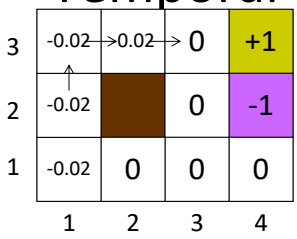
Temporal Difference Learning



With learning rate
α=0.5

1. Store counts N[s] and estimated values $V^{\pi}(s)$
2. For each observed transition (s,r,a,s'):
   3. Set N[s] $\leftarrow$ N[s]+1
   4. Adjust value $V^{\pi}(s) \leftarrow V^{\pi}(s)+\alpha(N[s])(r+\gamma V^{\pi}(s')-V^{\pi}(s))$

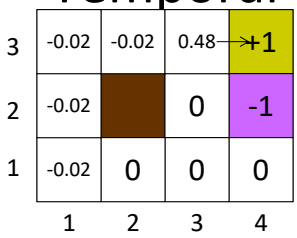
Temporal Difference Learning



With learning rate
α=0.5

1. Store counts N[s] and estimated values $V^{\pi}(s)$
2. For each observed transition (s,r,a,s'):
   3. Set N[s] $\leftarrow$ N[s]+1
   4. Adjust value $V^{\pi}(s) \leftarrow V^{\pi}(s)+\alpha(N[s])(r+\gamma V^{\pi}(s')-V^{\pi}(s))$

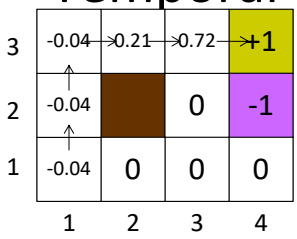
Temporal Difference Learning



With learning rate
α=0.5

1. Store counts N[s] and estimated values $V^{\pi}(s)$
2. For each observed transition (s,r,a,s'):
   3. Set N[s] ← N[s]+1
   4. Adjust value $V^{\pi}(s) \leftarrow V^{\pi}(s)+\alpha(N[s])(r+\gamma V^{\pi}(s')-V^{\pi}(s))$

Temporal Difference Learning

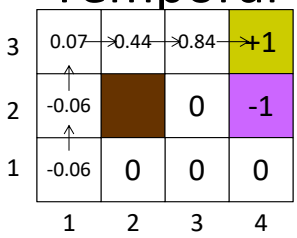


With learning rate
α=0.5

After a second trajectory from start to +1

1. Store counts N[s] and estimated values $V^{\pi}(s)$
2. For each observed transition (s,r,a,s'):
   3. Set N[s] ← N[s]+1
   4. Adjust value $V^{\pi}(s) \leftarrow V^{\pi}(s)+\alpha(N[s])(r+\gamma V^{\pi}(s')-V^{\pi}(s))$

Temporal Difference Learning

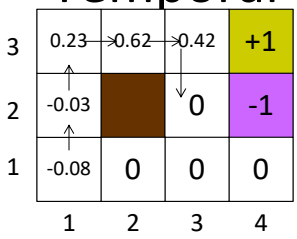


With learning rate
α=0.5

After a third trajectory
from start to +1

1. Store counts N[s] and estimated values $V^{\pi}(s)$
2. For each observed transition (s,r,a,s'):
   3. Set N[s] ← N[s]+1
   4. Adjust value $V^{\pi}(s) \leftarrow V^{\pi}(s)+\alpha(N[s])(r+\gamma V^{\pi}(s')-V^{\pi}(s))$

Temporal Difference Learning

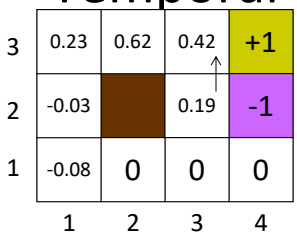


With learning rate
α=0.5

Our luck starts to run
out on the fourth trajectory

1. Store counts N[s] and estimated values $V^{\pi}(s)$
2. For each observed transition (s,r,a,s'):
   3. Set N[s] ← N[s]+1
   4. Adjust value $V^{\pi}(s) \leftarrow V^{\pi}(s)+\alpha(N[s])(r+\gamma V^{\pi}(s')-V^{\pi}(s))$

Temporal Difference Learning

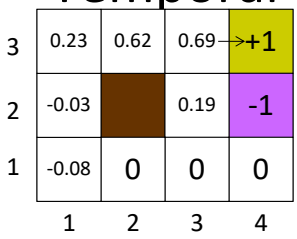


With learning rate α=0.5

But we recover…

1. Store counts N[s] and estimated values $V^{\pi}(s)$
2. For each observed transition (s,r,a,s'):
   3. Set N[s] ← N[s]+1
   4. Adjust value $V^{\pi}(s) \leftarrow V^{\pi}(s)+\alpha(N[s])(r+\gamma V^{\pi}(s')-V^{\pi}(s))$

Temporal Difference Learning



With learning rate α=0.5

…and reach the goal!

1. Store counts N[s] and estimated values $V^{\pi}(s)$
2. For each observed transition (s,r,a,s'):
   3. Set N[s] ← N[s]+1
   4. Adjust value $V^{\pi}(s) \leftarrow V^{\pi}(s)+\alpha(N[s])(r+\gamma V^{\pi}(s')-V^{\pi}(s))$

- For any s, distribution of s' approaches $P(s'|s,\pi(s))$
- Uses relationships between adjacent states to adjust utilities toward equilibrium
- Unlike direct estimation, learns before trial is terminated

Using TD for Control

- Recall value iteration:

$$V^{i+1}(s) = \max_a R(s,a) + \gamma \sum_{s'} P(s' \mid s,a) V^i(s')$$

- Why not pick the maximizing **a** and then do:

$$V(s) = V(s) + \alpha(N(s))(r + \gamma V(s') - V(s))$$

  – s' is the observed next state after taking action **a**

What breaks?

- Action selection
  - How do we pick a?
  - Need to P(s'|s,a), but the reason why we're doing RL is that we don't know this!
- Even if we magically knew the best action:
  - Can only learn the value of the policy we are following
  - If initial guess for V suggests a stupid policy, we'll never learn otherwise

Q-Values

- Learning V is not enough for action selection because a transition model is needed
- Solution: learn Q-values: Q(s,a) is the utility of choosing action a in state s
- "Shift" or "split" Bellman equation
  - $V(s) = \max_a Q(s,a)$
  - $Q(s,a) = R(s) + \gamma \Sigma_{s'} P(s'|s,a) \max_{a'} Q(s',a')$
- So far, everything is the same… but what about the learning rule?

Q-learning Update

- Recall TD:
  - Update: $V(s) \leftarrow V(s)+\alpha(N[s])(r+\gamma V(s')-V(s))$
  - Use P to pick actions? $a \leftarrow \arg\max_a \Sigma_{s'} P(s'|s,a)V(s')$
- Q-Learning:
  - Update: $Q(s,a) \leftarrow Q(s,a)+\alpha(N[s,a])(r+\gamma \max_{a'} Q(s',a')-Q(s,a))$
  - Select action: $a \leftarrow \arg\max_a Q(s,a)$
- Key difference: average over $P(s'|s,a)$ is "baked in" to the Q function
- Q-learning is therefore a **model-free** learner

Q-Learning Demo (see video from Zoom)

- Keyboard controlled Q-learning agent using robot grid world from R&N (and previous MDP slides)
- Differences:
  - Discount of 0.9
  - No step cost (previously -0.04)
  - Bad state has value 0 (previously -1)
- How to think of terminal states:
  - Make transition to another state (absorbing state) with value that is always 0
  - Problem then resets to start (no transition from absorbing state to start)

Q-learning vs. TD-learning

- TD converges to value of policy you are following
- Q-learning converges to values of optimal policy independent of of whatever policy you follow during learning!
- Caveats:
  - Converges in limit, assuming all states are visited infinitely often
  - In case of Q-learning, all states and actions must be tried infinitely often

Note: If there is only one action possible in each state, then Q-learning and TD-learning are identical

Brief Comments on Learning from Demonstration

- LfD is a powerful method to convey human expertise to (ro)bots
- Useful for imitating human policies
- Less useful for surpassing human ability (but can smooth out noise in human demos)
- Used, e.g., for acrobatic helicopter flight

Advanced (but unavoidable) Topics

- Exploration vs. Exploitation
- Value function approximation

Exploration vs. Exploitation

- Greedy strategy purely **exploits** its current knowledge
  - The quality of this knowledge improves only for those states that the agent observes often
- A good learner must perform **exploration** in order to improve its knowledge about states that are not often observed
  - But pure exploration is useless (and costly) if it is never exploited

Restaurant Problem

Exploration vs. Exploitation in Theory and Practice

- Can assign an "exploration bonus" to parts of the world (or state-action combinations) you haven't experienced much
  - Versions of this are provably efficient, e.g., R-Max (will eventually learn the optimal policy requiring polynomial effort in size of problem)
  - Works for small state spaces
- In practice ε-greedy action selection is used most often
  - Choose greedy action w.p. 1-ε
  - Choose random action w.p. ε

Value Function Representation

- Fundamental problem remains unsolved:
  - TD/Q 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
  - Neural networks
  - State aggregation (violates Markov property)
- Idea: Approximate f(s) with g(s;w)
  - g is some easily computable function of s and w
  - Try to find w that minimizes the error in g

Updates with Approximation

- Recall regular TD update:

$$V(s) \leftarrow V(s)+\alpha(N[s])(r+\gamma V(s')-V(s))$$

- With function approximation:

$$V(s) \approx V(s;w)$$

Vector operations

- Update:

$$w^{i+1} = w^i + \alpha(r+\gamma V(s';w)-V(s;w))\nabla_w V(s;w)$$

Neural networks are a special case of this.

Linear Regression Review

- Define a set of basis functions (vectors)

$$\varphi_1(s), \varphi_2(s) \ldots \varphi_k(s)$$

- Approximate f with a weighted combination of these

$$g(s; w) = \sum_{j=1}^{k} w_j \varphi_j(s)$$

- Example: Space of quadratic functions:

$$\varphi_1(s) = 1, \varphi_2(s) = s, \varphi_3(s) = s^2$$

- Orthogonal projection minimizes SSE

For linear value functions

- Gradient is trivial:

$$V(s; w) = \sum_{j=1}^{k} w_j \varphi_j(s)$$

$$\nabla_{w_j} V(s; w) = \varphi_j(s)$$

- Update is trivial:

Individual components

$$w_j^{i+1} = w_j^{i} + \alpha(r + \gamma V(s'; w) - V(s; w))\varphi_j(s)$$

Properties of approximate RL

- Exact case (tabular representation) = special case
- Can be combined with Q-learning

- Convergence not guaranteed
  - Policy evaluation with linear function approximation converges if samples are drawn "on policy"
  - In general, convergence is not guaranteed
    - Chasing a moving target
    - Errors can compound
- Success has traditionally required very carefully chosen features
- Deepmind has recently had success using no feature engineering but lots of training data

How'd They Do That???

- Backgammon (Tesauro)
  - Neural network value function approximation
  - TD sufficient (known model)
  - Carefully selected inputs to neural network
  - About 1 million games played against self
- Atari games (DeepMind)
  - Used convolutional neural network for Q-functions
  - Days of play time per game
- Helicopter (Ng et al.)
  - Learning from expert demonstrations
  - Constrained policy space
  - Trained on a simulator

Conclusions

- Reinforcement learning solves an MDP
- Converges for exact value function representation
- Can be combined with approximation methods
- Good results require good features and/or lots of data