Reinforcement Learning

CPS 270
Ron Parr

Limitations of the MDP Framework

- Assumes that transition probabilities known
  - How do we discover these?
  - How do we store them?
    - Big problems have big models
    - Model size is quadratic in state space size
- Assumes reward function is known

Reinforcement Learning

- Learn by "trial and error"
- No assumptions about model
- No assumptions about reward function
- Assumes:
  - True state is known at all times
  - Reward is known
  - Discount is known

Comparison to Other Kinds of Learning

- We will cover learning in more detail later, but…
- Learning often viewed as:
  - Classification (supervised), or
  - Model learning (unsupervised)
- RL characterized by delayed signal

RL Schema

- Act
- Perceive next state
- Update (something)
- Repeat

Highlights

- Trial and error methods can be mapped to the dynamic programming/MDP framework
- Easy to implement
  - Easy to code
  - Low CPU utilization
- Incremental approach converges in the limit to the same answer as traditional methods
- Can be combined with approximation methods –with some caveats

Highlights II

- Used successfully in many domains
- Most notable success: Backgammon
- Sutton's original "Learning to predict by the methods of temporal differences" paper, is 40th most cited paper in CS (citeseer, 9/2004)

Model Learning Approach

- Learn model, solve = Certainty Equivalence
- 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

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

Temporal Difference Learning

- One of the first RL algorithms
- Learn the value of a *fixed* policy (no optimization; just prediction)
- Compare with iterative value determination:

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

Problem: We don't know this.

First Idea: Monte Carlo Sampling

- Assume that we have a black box:

- Count the number of times we see each s'
  - Estimate P(s'|s) for each s'
  - Essentially learns a mini-model for state s
  - Can think of as numerical integration
- Problem: The world doesn't work this way

Next Idea

- Remember Value Determination:

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

- Compute an update as if the observed s' and r were the only possible outcomes:

$$V^{temp}(s) = r + \gamma V^i(s')$$

- Make a small update in this direction:

$$V^{i+1}(s) = (1-\alpha) V^i(s) + \alpha V^{temp}(s) \qquad 0 < \alpha \leq 1$$

Convergence?

- Why doesn't this oscillate?
  - e.g. consider some low probability s' with a very high (or low) reward value

  - This could still cause a big jump in V(s)

Ensuring Convergence

- Rewards have bounded variance
- $0 \le \gamma < 1$
- Every state visited infinitely often
- Learning rate decays so that:
  - $\sum_i \alpha_i(s) = \infty$
  - $\sum_i \alpha_i^2(s) < \infty$

These conditions are jointly *sufficient* to ensure convergence in the limit with probability 1.

How Strong is This?

- Bounded variance of rewards: easy
- Discount: standard
- Visiting every state infinitely often: Hmmm…
- Learning rate: Often leads to slow learning
- Convergence in the limit: Weak
  - Hard to say anything stronger w/o knowing the mixing rate of the process
  - Mixing rate can be low; hard to know a priori
- Convergence w.p. 1: Not a problem.

Value Function Representation

- Fundamental problem remains unsolved:
  - TD 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
  - Perceptron
  - Neural networks

  } Covered in more detail later in course

- 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

Linear Regression

- Define a set of basis functions (vectors)

$$h_1(s), h_2(s) .. h_k(s)$$

- Approximate f with a weighted combination of these basis functions

$$g(s) = \sum_{i=1}^{k} w_i h_i(s)$$

- Example: Space of quadratic functions:

$$h_1(s) = 1, h_2(s) = s, h_3(s) = s^2$$

- Orthogonal projection minimizes sum of squared errors

Updates with Approximation

- Recall regular TD update:

$$V^{i+1}(s) = (1-\alpha)V^{i}(s) + \alpha V^{temp}(s)$$

- With function approximation:

$$V(s) \approx V(s,\theta)$$

Vector operations

- Update:

$$\theta^{i+1}(s) = (1-\alpha)\theta^{i}(s) + \alpha V^{temp}(s)\nabla_{\theta}V(s,\theta)$$

For linear value functions

- Gradient is trivial:

$$V(s,\theta) = \sum_{i=1}^{k}\theta_i h_i(s)$$

$$\nabla_{\theta_i}V(s,\theta) = h_i(s)$$

Individual components

- Update is trivial:

$$\theta_i^{j+1}(s) = (1-\alpha)\theta_i^{j}(s) + \alpha V^{temp}(s)h_i(s)$$

Properties of approximate RL

- Note: Prediction only, so far
- Table-updates are a special case
- Converges for linear architectures (Tsitsiklis & Van Roy)
- May not converge for general function approximation architectures

Using TD for Control

- Recall value iteration:

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

- Why not pick the maximizing a and then do:

$$V^{i+1}(s) = (1-\alpha)V^{i}(s') + \alpha V^{temp}(s')$$

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

Problems

- How do you pick the best action w/o model?
- Must visit every state infinitely often
  - What if a good policy doesn't do this?
- Learning is done "on policy"
  - Taking random actions to make sure that all states are visited will cause problems
- Linear function approximation doesn't provably converge for optimization (but is still used successfully in many cases!)

Q-Learning Overview

- Want to maintain good properties of TD
- Learns good policies and optimal value function, not just the value of a fixed policy
- Simple modification to TD that learns the optimal policy regardless of how you act! (mostly)

Q-learning

- Recall value iteration:

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

- Can split this into two functions:

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

$$V^{i+1}(s) = \max_a Q^{t+1}(s,a)$$

Q-learning

- Store Q values instead of a value function
- Makes selection of best action easy
- Update rule:

$$Q^{temp}(s,a) = r + \gamma V^i(s')$$

$$Q^{i+1}(s,a) = (1-\alpha) Q^i(s,a) + \alpha Q^{temp}(s,a)$$

Q-learning Properties

- For table representations, converges under same conditions as TD
- Still must visit every state infinitely often
- Separates policy you are currently following from value function learning:

$$Q^{temp}(s,a) = r + \gamma V^i(s')$$

$$Q^{i+1}(s,a) = (1-\alpha) Q^i(s,a) + \alpha Q^{temp}(s,a)$$

Q-learning Properties

- Can't prove convergence with function approximation
- Introduces exploration vs. exploitation dilemma