Limitations of the MDP Framework

- Assumes that transition probabilities are known
  - How do we discover these?
  - How do we store them?
    - Big problems have big models
    - Model size is quadratic in state space size
      - Sparse representations
      - Bayesian networks (later…)
- 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

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 function approximation—with some caveats
- Has been used successfully in many domains

Model Learning Approach

- Model learning + solving = 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:
  \[ V^{i+1}(s) = R(s) + \gamma \sum_s P(s'|s)V^i(s') \]
  Problem: We don’t know this.

First Idea: Monte Carlo Sampling

- Assume that we have a black box:
  \[ S \rightarrow \text{black box} \rightarrow S' \]
- 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:
  \[ V^{i+1}(s) = R(s) + \gamma \sum_s P(s'|s)V^i(s') \]
- Compute an update as if the observed \( s' \) and \( r \) were the only possible outcomes:
  \[ V^{\text{temp}}(s) = r + \gamma V'(s') \]
- Make a small update in this direction:
  \[ V^{i+1}(s) = (1-\alpha)V^i(s) + \alpha V^{\text{temp}}(s) \quad 0 < \alpha \leq 1 \]

Idea: Value Function Soup

Suppose: \( \alpha = 0.1 \)

Upon observing \( s' \):
- Discard 10% of soup
- Refill with \( V^{\text{temp}}(s) \)
- Stir
- Repeat

\[ V^{\text{temp}}(s) = r + \gamma V'(s') \]

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
  - $0 \leq \gamma < 1$
- Every state visited infinitely often
- Learning rate decays so that:
  - $\sum \alpha_i(s) = \infty$
  - $\sum \alpha_i^2(s) \leq \infty$

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

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 $f(s)$
  - Perceptron
  - Linear regression
  - Neural networks
- 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)
  - $h_1(s), h_2(s), \ldots, h_k(s)$
- Approximate $f$ with a weighted combination of these basis functions
  - $g(s) = \sum_m w_m h_m(s)$
- 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

Neural Networks

- $s =$ input into neural network
- $w =$ weights of neural network
- $g(s, w) =$ output of network
- Try to minimize
  - $E = \sum (f(s) - g(s, w))^2$
  - Compute gradient of error wrt weights
    - $\frac{\partial E}{\partial w}$
  - Adjust weights in direction that minimizes error
Combining NNs with TD

- Recall TD:
  \[ V^\text{temp}(s) = R(s) + \gamma V'(s') \]
  \[ V'^{i+1}(s) = (1-\alpha)V'(s) + \alpha V^\text{temp}(s) \]
- Compute error function:
  \[ E = \left( \hat{V}(s,w) - \hat{V}^\text{temp}(s,w) \right)^2 \]
- Update:
  \[ w^{i+1} = w' - \alpha \frac{\partial E}{\partial w} \]
  \[ = w' + 2\alpha \left[ \hat{V}^\text{temp}(s,w) - \hat{V}(s,w) \right] \frac{\partial \hat{V}(s,w)}{\partial w} \]

Gradient-based Updates

- Constant factor absorbed into learning rate
- Table-updates are a special case
- Perceptron, linear regression are special cases
- Converges for linear architectures
  (Tsitsiklis & Van Roy)

Using TD for Control

- Recall value iteration:
  \[ V'^{i+1}(s) = R(s) + \max_a \sum P(s'|s,a)V'(s') \]
- Why not pick the maximizing a and then do:
  \[ V'^{i+1}(s) = (1-\alpha)V'(s') + \alpha V^\text{temp}(s') \]
  - \( s' \) 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:
  \[ V'^{i+1}(s) = R(s) + \max_a \sum P(s'|s,a)V'(s') \]
- Can split this into two functions:
  \[ Q'^{i+1}(s,a) = R(s) + \gamma \sum P(s'|s,a)V'(s') \]
  \[ V'^{i+1}(s) = \max_a Q'^{i+1}(s,a) \]
Q-learning

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

\[ Q^{\text{new}}(s,a) = r + \gamma V'(s') \]

\[ Q^{\text{old}}(s,a) = (1-\alpha)Q'(s,a) + \alpha Q^{\text{new}}(s,a) \]

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:

\[ Q^{\text{new}}(s,a) = r + \gamma V'(s') \]

\[ Q^{\text{old}}(s,a) = (1-\alpha)Q'(s,a) + \alpha Q^{\text{new}}(s,a) \]

Q-learning Properties

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