Reinforcement Learning

Intro

- Reinforcement learning lets an agent "learn to act well even when it doesn't know how the world works.

> Idea: wander around in the world, getting positive and negative rewards. Eventually learn to act optimally (i.e., so as to maximize expected total reward).

> Attractive bc: lets the agent "learn by itself", using only feedback from the world. (In practice, not always so easy!)

- How does RL relate to MDPs?

> MDP Review: \(<s, a, Pr(s' | s, a), R(s)\>

- state: actions: transition model: reward

- assumed: fully observable environment

- probabilistic action outcomes

- known values of transition model & reward function

- goal: find policy \(\pi: S \to A\) to maximize expected total reward

> RL: \(<s, a>\)

- still assume:

  - fully observable environment

  - probabilistic action outcomes

- still goal:

  - find policy \(\pi: S \to A\)

- now we don't know:

  - transition model: \(Pr(s' | s, a)\)

  - reward function: \(R(s)\)

Possible approaches to the problem:

- model-based: learn utility function on states (and then most also learn transition model)

- model-free:

  - Q-learning: learn action-value Q-function, giving expected utility of taking an action in a state

  - policy search: learn policy directly, that maps states to actions

- situations in which standard supervised learning is inappropriate (e.g., control tasks, game strategies, etc.) (agent always knows current state)
- passive vs active learning:
  - passive: assume you know \( \mathcal{M} : S \rightarrow A \), and you want to learn how the world works (utilizing function, transition model, etc.) (complement of solving an MDP)
  - active: want to learn both how the world works and what actions to take to maximize total expected reward

**Passive Learning**

- Structure of the problem
  - fixed policy \( \pi : S \rightarrow A \)
  - goal: learn utility function \( V^\pi(s) \) according to policy \( \pi \)
    (i.e. quality of policy \( \pi \), or the way the world works)
    [Might also need to learn \( \Pr(s' | s,a) \) and \( R(s) \)]

  *Series of trials in the environment*, using \( \pi \), starting in some state, ending in a terminal state

  Use observed rewards to learn expected utility \( \hat{V}^\pi(s) \) of each nonterminal state:

  \[
  \hat{V}^\pi(s) = \mathbb{E} \left[ \sum_{t=0}^{\infty} \gamma^t R(s_t) | \pi, s_0 = s \right]
  \]

  (same utility function as MDPs)

**Adaptive Dynamic Programming (ADP)**

- Idea: learn transition model \( \Pr(s' | s,a) \) of environment, then solve corresponding MDP using dynamic programming

- Learning transition model: easy; life world is fully observable

  - Simple counting to get ML estimate — keep track of how often \( s' \) is reached when executing \( a \) in \( s \) .
    (Could be Bayesian here)

  - Learning utility function: solve Bellman equation using learned transition model and observed rewards.

  \[
  V^\pi(s) = R(s) + \gamma \sum_{s'} \Pr(s' | s, \pi(s)) V^\pi(s')
  \]

  (Just linear programming, because no max operator.)

- Performance:
  - learns as quickly as possible, given its ability to learn the transition model
  - But — it requires solving \( |S| \) equations with \( |S| \) unknowns, so not really tractable for large state spaces.

*Example*
Temporal Difference Learning (TD)

- Idea: Use each observed transition to adjust the utility values of observed states to agree with constraint equations.

- Constraint equations that should hold at equilibrium:
  \[ V^\pi(s) = R(s) + \gamma \sum_{s'} p(s'|s, a) V^\pi(s') \]  
  [Bellman Eq]

  Update that moves current estimate towards this:
  \[ V^\pi(s) \leftarrow V^\pi(s) + \alpha (R(s) + \gamma V^\pi(s') - V^\pi(s)) \]  
  [TD Eq]

  Note: only involves successor state s', not all possible next states!

  Why is this the right thing?
  - The frequency of each successor in the set of transitions is approximately proportional to its probability. Thus, like a crude but efficient first approximation to ADP -
  - The model-free! Doesn’t need to estimate \( p(s'|s, a) \), because the environment supplies the connection between adjacent states, as observed transitions.

- Active Learning:
  - TD(\alpha): Updates values of all states in sequence leading up to each transition, by an amount that drops off as \( \alpha \) for \( \alpha \) steps into the future.
  - Rewards resulting from action may not be received for several time steps in future.

Now we must learn what actions to take, as well as how the world works.

- Greedy ADP:
  - Learn complete model for all actions (not just fixed policy).
  - Learn utilities as defined by optimal policy:
    \[ V(s) = R(s) + \gamma \max_a \sum_{s'} p(s'|s, a) V(s') \]  
    [Bellman Eq]

  - Choose actions with best expected utility using one-step lookahead and the learned \( V(s) \), or with learned policy \( \pi \).

- The learned model is not the same as the true environment, especially early in learning. So choosing the optimal action always according to the model can lead to convergence upon a very non-optimal policy.
e.g. Might learn if early training sequences provide these paths.

> Actions both:

- provide rewards according to current model, and
- help learn true model by providing information about how the world works.

(and a better model will increase expected rewards in the future.)

- Tradeoff between:
  - exploitation: maximizing reward according to current utility function
  - exploration: improving knowledge, to maximize long-term reward.

- Cannot successfully do either alone - will either get stuck in a rut, or never actually use learned knowledge.

- Can we define an optimal exploration policy?
  - This has been studied formally in "bandit problems"
    - n-armed bandit has n levers
    - which lever to play on this coin?
      - one that has paid off best
      - one that has not been tried, or tried very rarely?

  - Can be solved exactly for one-step games under certain conditions, but very difficult for sequential decision problems.
    - But we can find a reasonable scheme that will lead to optimal behavior, as long as it is "greedy in the limit of infinite exploration" (GLIE) (i.e., tries each state-action pair an unbounded number of times, to avoid missing optimal action.)
    - Using a GLIE exploration scheme will allow the agent to eventually learn the true model.
    - But the agent must eventually become greedy, to exploit that learned model (i.e. choose optimal actions according to it.)
- Possible GLIE schemes:
  - Choose random action \( a \) of the time (follow greedy policy otherwise). Will converge, but very slowly.
  - Be optimistic about utility of relatively unexplored state-action pairs (i.e. pretend reward everywhere, until proved wrong)

- Exploration function: \( f(u, n) \)
- Determines trade-off between greed (high \( u \)) and curiosity (low \( n \))
- Should be increasing in \( u \) and decreasing in \( n \)
  \[ f(u, n) = \begin{cases} R^+ & \text{if } n < N_e \\ u & \text{otherwise} \end{cases} \]
- For ADP with value iteration:
  \[ V^*(s) \leftarrow R(s) + \gamma \max_a \left[ \sum_{s'} P_r(s'|s,a)V^*(s') + N_s(a) \right] \]

> \( V^+ \) is on both sides of equation - important.
- Benefits of exploration are propagated back from unexplored regions.
- Actions leading towards unexplored regions are preferred, not just unfamiliar actions themselves.

**Active TD(0) Learner**
- Can learn utility function, just like passive case:
  \[ V(s) \leftarrow V(s) + \alpha (R(s) + \gamma V(s') - V(s)) \]
- Must use some exploration scheme, just as in active ADP.
- Now must also learn \( P_r(s'|s,a) \) - need to be able to choose action to maximize expected utility, using one-step lookahead.
  - e.g. MLE estimates
- Will converge to some values as ADP in the limit of infinite training sequences.
Q-learning

- Alternate TD method that learns values on state-action pairs $Q(s, a)$, instead of utilities.

$$V(s) = \max_{a} Q(s, a)$$

- Model-free! Needs no model of the world, for learning or action selection.

- Constraint equation that must hold at equilibrium:

$$Q^*(s, a) = R(s) + \gamma \sum_{s'} P_r(s' | s, a) \max_{a'} Q^*(s', a')$$

$$V^*(s) = \arg\max_{a} Q^*(s, a)$$

But this uses a model of the world, so don't want to use for learning.

- TD update equation for Q-learning:

$$Q(s, a) \leftarrow Q(s, a) + \alpha(N(s, a))(R(s) + \gamma \max_{a'} Q(s', a') - Q(s, a))$$

Perform update for each $(s, a, s')$ tuple.

- Guaranteed to converge (if the world is really an MDP!)
- But like model-free, cannot perform look-ahead, so learning is slower. (Doesn't enforce consistency among values.

- Need to handle exploration here as well:
  - Exploration function of $N(s, a)$
  - Decreasing with time percentage of random actions

Model-based vs. Model-free?

- Is it better to learn a transition model + utility function, or just learn a model-free action-value function?

  - Unclear, depends on problem
  - Seems that, the more complex the environment, the more leverage a model-based approach provides.

Generalization in RL

- So far, assumed tabular utility, Q-functions. Time to convergence increases as state space grows.

E.g. Chess and backgammon have $10^{50}$ to $10^{120}$ states!
**Function Approximation**

- Anything other than table representation.
- E.g., weighted linear function or set of features

\[ V_i(s) = \theta_1 f_1(s) + \theta_2 f_2(s) + \ldots + \theta_n f_n(s) \]

(used in chess)

Now, learning \( \theta = \sum \Theta \) is sufficient to represent utility function for entire state space \((20 \text{ vs. } 10^{120})\) - huge savings in space!

- More important: allows learner to generalize from states it has seen, to unseen parts of the state space! Crucial.

- Difficulty: true utility or Q-function might not exist in chosen hypothesis space (i.e., of the given form) - possible to find a good approximator?

> For function approximators that are linear in the parameters (which themselves can be any function of state variables), 
TD and Q-learning update equations will converge to closest possible approximation to true function (for some technical definition of "close").

> For non-linear functions:
- Neural nets
- Regression trees (e.g., when states are represent relatively)

This is not proven convergence.

**Other open areas in RL**
- Factored state representations
- Partially observable worlds
- Hierarchical RL
- Changing world dynamics

**In Practice**
- Hard to define reward function!

- Works well in problems with small state spaces, or that can learn in simulation:
  - TD-Gradient: [Tesauro, 1992]
  - Neural network function approximator, TD(a)
  - Learned from self-play in simulation, only reward at the end
  - Using precomputed board features, 300,000 training games, learned to play as well as top 3 players in world

- Inverted pendulum
- Andrew Ng's PEGASUS (2000) - used policy gradient descent and simulation for training, transferred to real world.