Coming Up With Better Policies

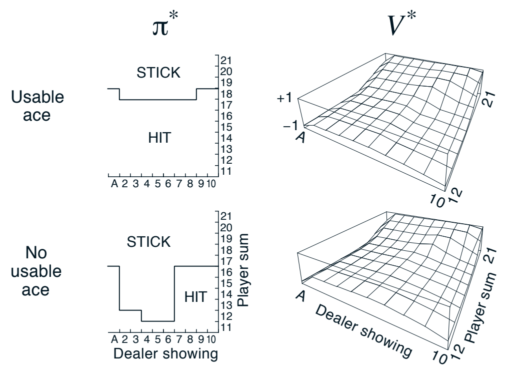
We can interleave policy evaluation with policy improvement as before.

$$\pi_0 \xrightarrow{E} Q^{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} \cdots \xrightarrow{I} \pi^* \xrightarrow{E} Q^*$$

We've just figured out how to do policy evaluation.

Policy improvement is even easier because now we have the direct expected rewards for each action in each state $Q(s,a)$ so just pick the best action among these

The optimal policy for Blackjack:



On-Policy Learning

On-policy methods attempt to evaluate the same policy that is being used to make decisions

Get rid of the assumption of exploring starts. Now use an ϵ-greedy method where some ϵ proportion of the time you don't take the greedy action, but instead take a random action

Soft policies: all actions have non-zero probabilities of being selected in all states

For any ϵ-soft policy π, any ϵ-greedy strategy with respect to Q^π is guaranteed to be an improvement over π.

If we move the ϵ-greedy requirement inside the environment, so that we say *nature* randomizes your action $1-\epsilon$ proportion of the time, then the best one can do with general strategies in the new environment is the same as the best one could do with ϵ-greedy strategies in the old environment.

Adaptive Dynamic Programming

Simple idea – take actions in the environment (follow some strategy like ϵ-greedy with respect to your current belief about what the value function is) and update your transition and reward models according to observations. Then update your value function by doing full dynamic programming on your current believed model.

In some sense this does as well as possible, subject to the agent's ability to learn the transition model. But it is highly impractical for anything with a big state space (Backgammon has 10^{50} states)

Temporal-Difference Learning

What is MC estimation doing?

$$V(s_t) \leftarrow (1-\alpha_t)V(s_t) + \alpha_t R_t$$

where R_t is the return received following being in state s_t.

Suppose we switch to a constant step-size α (this is a trick often used in nonstationary environments)

TD methods basically bootstrap off of existing estimates instead of waiting for the whole reward sequence R to materialize

$$V(s_t) \leftarrow (1-\alpha)V(s_t) + \alpha[r_{t+1} + \gamma V(s_{t+1})]$$

(based on actual observed reward and new state)

This target uses the current value as an estimate of V whereas the Monte Carlo target

uses the sample reward as an estimate of the expected reward

If we actually want to converge to the optimal policy, the decision-making policy must be GLIE (greedy in the limit of infinite exploration) – that is, it must become more and more likely to take the greedy action, so that we don't end up with faulty estimates (this problem can be exacerbated by the fact that we're bootstrapping)

Q-Learning: A Model-Free Approach

Even without a model of the environment, you can learn effectively. Q-learning is conceptually similar to TD-learning, but uses the Q function instead of the value function

1. In state s, choose some action a using policy derived from current Q (for example, ϵ-greedy), resulting in state s' with reward r.

2. Update:

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha(r + \gamma \max_{a'} Q(s',a'))$$

You don't need a model for either learning or action selection!

As environments become more complex, using a model can help more (anecdotally)

Generalization in Reinforcement Learning

So far, we've thought of Q functions and utility functions as being represented by tables

Question: can we parameterize the state space so that we can learn (for example) a linear function of the parameterization?

$$V_\theta(s) = \theta_1 f_1(s) + \theta_2 f_2(s) + \cdots + \theta_n f_n(s)$$

Monte Carlo methods: We obtain sample of $V(s)$ and then learn the θ's to minimize squared error.

In general, often makes more sense to use an online procedure, like the Widrow-Hoff rule:

Suppose our linear function predicts $V_\theta(s)$ and we actually would "like" it to have predicted something else, say v. Define the error as $E(s) = (V_\theta(s) - v)^2/2$. Then the update rule is:

$$\theta_i \leftarrow \theta_i - \alpha \frac{\partial E(s)}{\partial \theta_i}$$

$$= \theta_i + \alpha(v - V_\theta(s)) \frac{\partial V_\theta(s)}{\partial \theta_i}$$

If we look at the TD-learning updates in this framework, we see that we essentially replace what we'd "like" it to be with the learned backup (sum of the reward and the value function of the next state:

$$\theta_i \leftarrow \theta_i + \alpha[R(s) + \gamma V_\theta(s') - V_\theta(s)] \frac{\partial V_\theta(s)}{\partial \theta_i}$$

This can be shown to converge to the closest function to the true function when linear function approximators are used, but it's not clear how good a linear function will be at approximating non-linear functions in general, and all bets on convergence are off when we move to non-linear spaces.

The power of function approximation: allows you to generalize to values of states you haven't yet seen!

In backgammon, Tesauro constructed a player as good as the best humans although it only examined one out of every 10^{44} possible states. Caveat: this is one of the few successes that has been achieved with function approximation and RL. Most of the time it's hard to get a good parameterization and get it to work.