Reinforcement learning notes

Dana H. Ballard
Department of Computer Science, The University of Texas at Austin
Austin, TX, 78712, USA
danab@utexas.edu, zharu@utexas.edu

Introduction

This short primer for reinforcement learning will cover a range of topics summarized in the following list:

1. Definitions
2. basic algorithms
3. Model-free algorithms
   (a) value iteration
   (b) policy iteration
4. modules
   (a) credit assignment
   (b) inverse RL
5. Value function approximation

1 Basic definitions

Reinforcement learning uses at its core the Markov decision process. Its basic elements are (S,A,R,T) elaborated by the following:

- S is the discrete sets describing the problem
- A is the set of actions from states.
- $T(s, a, s')$ is a transition function that results in an action a taken in state s results in being in another state s'.
- R is the value of reward received by taking an action.

The goal of reinforcement learning is to compute a $\pi(s)$, which is the action to be taken in state s. In addition each state should have a value V(s), which is the discounted reward from taking the action specified by $\pi(s)$.

The goal in reinforcement learning is maximize expected reward. A state sequence allows this to be written as:

$$E[R(s_0) + (s_1) + \gamma^2 R(s_2) + \cdots]$$

If this squence was the result of a policy $\pi(s)$, it can be denoted

$$V^{\pi}(s)|s_o = s, \pi$$

meaning start in s_0 and follow the policy thereafter. Now for a most important step. Puting the last two formulas together results in the well known *Bellman equation*:

$$V^{\pi}(s) = R(s) + \gamma \sum T(s, \pi, s')V^p i(s') \tag{1}$$

Another thing that we will need is the expected value of taking action a from state s, denoted as $Q(s, a)$. If we have $Q(s, a)$, we can always recover $V(s)$as:

$$V(s) = \max_a Q(s, a) \tag{2}$$

2 basic algorithms

If the state had a dynamic function $s_{t+1} = f(s_t, u)$, the it would be possible to compute the policy by starting at the terminal time t_f and working backward recursively computing the best action to take using **dynamic programing**. But while the function T is also know as the reinforcement learning setting's *dynamics*, it is *probabilistic*, meaning that we know the probabilities for going forward but not for going backward.

There are still algorithms that can work in this case but they require iteration to compute a network's V and π.

2.1 Value iteration

Value iteration works by initially choosing values for all the states in the network. Next, we loop over all the states in the network and loop over the actions from each state. Each such state has a version of the Bellman equation, and this allows to compute $Q(s, a)$ and udate $V(s)$ using Eq. 2.

initialize $V(s)$ arbitrarily

loop until policy good enough

loop for $s \in S$

loop for $a \in A$

$$Q(s, a) := R(s, a) + \gamma \sum_{s' \in S} T(s, a, s')V(s')$$

$$V(s) := \max_a Q(s, a)$$

end loop

end loop

Figure 1: Value learning algorithm

2.2 Policy Learning

Policy learning works by starting with an arbitrary function. Give that $\pi(s)$ is determined, the Bellman equations are reduced to a set of equations in $V(s)$ that can be solved. Once this is done the policy can improved by using a version of the Bellman equations that include a maximization step that chooses a best policy given the new values. The previous computation step with the resultant new policy is repeated. These steps are repeated until the policy converges.

choose an arbitrary policy π'

loop

$\pi := \pi'$

compute the value function of policy π :

solve the linear equations

$$V_\pi(s) = R(s, \pi(s)) + \gamma \sum_{s' \in S} T(s, \pi(s), s') V_\pi(s')$$

improve the policy at each state:

$$\pi'(s) := \arg\max_a \left(R(s, a) + \gamma \sum_{s' \in S} T(s, a, s') V_\pi(s') \right)$$

until $\pi = \pi'$

Figure 2: Policy learning algorithm

3 Model-free algorithms

The previous section focused on the case that had an accompanying dynamics aka model in the form of a transition function that records the probability of the states reachable by choosing an action from a state. However in may common cases, such information is not available. Still a number of approaches are possible. The most straightforward would be to just sample the problem space repeatably and keep track of what happened. thus for a transition function for an action a to a state s', one can keep track of the number of times the system ended up in s' and divided by the total of transitions. However this method is very tedious, as we are only interested in the paths that offer the most reward.

Given this reward focus, one can keep track of how the em rewarding action choices prove to be and let those figures bias the action choices. One popular choice is the *epsilon greedy* protocol. Given a state and its set of actions with their experienced rewards, and a small fraction for ϵ parameter, the strategy is to sample the best action $1 - \epsilon$ of the time and take a random action the rest of the time. Figure 3 shows that this strategy gradually allows the best choice to be taken more and more often.

Now we are ready to describe the Q-Learning version of model-free learning. the algorithm works by repeatedly picking *episodes*, which adjust the local policies, and keeping doing this until the policies converge. To construct an episode, pick a state and then use the epsilon greedy policy

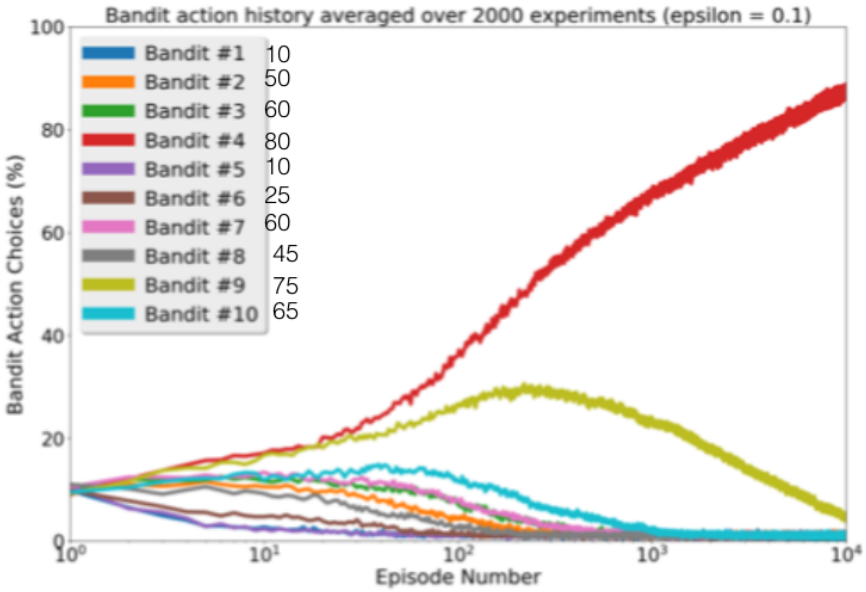

Figure 3: given a cell with ten actions, with different rewards,the epsilon greedy strategy learns to sample the best option most of the time[Alison Wong's simulation].

to choose an action to get to another state. Keep doing this to get to the end of an episode, which is another parameter. Finally, use Q-learning:

$$Q(s_t, a) = Q(s_t, a) + \Delta Q$$

where

$$\Delta Q = Q(s_t, a) - (R(s, a) + \gamma Q(s_{t+1}$$