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. Value function approximation
5. modules
   (a) credit assignment
   (b) inverse RL

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 4 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

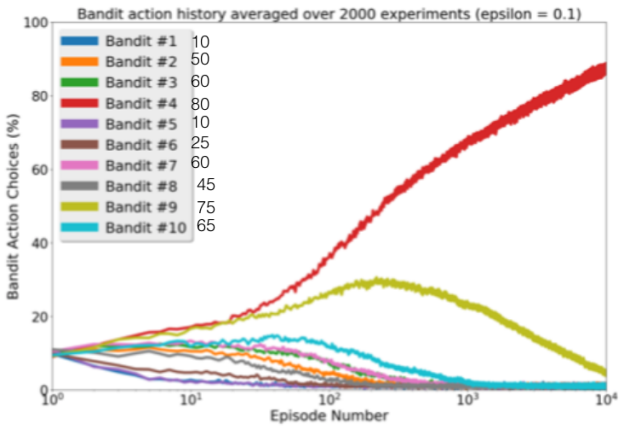

Figure 3: given a cell with ten actions, with different rewards,the epsilon greedy strategy learns to sample the best option most of the time[Alison Wong's simulation].

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

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

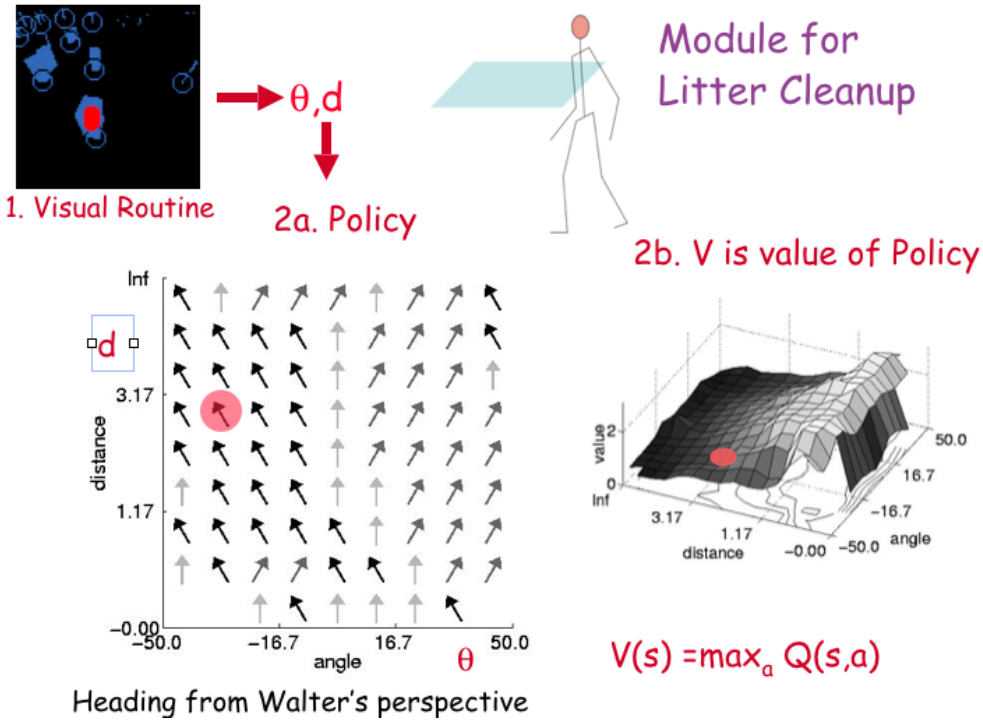

Figure 4: Q learning example. Learning to pick up litter from a sidewalk. The first step is to extract the state information from an image. The possible litter sites are marked by a computer vision algorithm. Next, the closest litter's state is coded as an angular heading θ and distance d. Using this representation, successive episodes are updated during repeated trials. The final result is shown in 2a as a policy map and in 2b as a value function.

4 Modules

Reinforcement learning suffers from what Bellman called "the curse of dimensionality." As the dimension of the state space increases, the cost of learning algorithms goes up exponently, making the search of high dimensional spaces impractical.

One approach is to factor a large problem into separate sub-problems. The sub-problems might be independent, but in a more common setting such as a mobile agent, they are coupled by having to agree on the policy.

Thus the MDP becomes $(S^{(i)}, R^{(i)}, T^{(i)}, \gamma^{(i)}, A), i = 1, \ldots, M$, with the superscript denoting the modules. Note that the actions do not have a script, signifying that the action taken is common as it is shared between all the modules.

Another issue that surfaces with modules is that of their number. How many modules should be active at any one time? The main problem is that as the number of modules grows, the management of the action selection becomes increasingly complex. Because of this difficulty, the set of active modules is best kept small.

However many situations can exhibit complex responses if the set of active modules can be managed so as to respond to the exigences in the moment as depicted in Fig. 5.

Modules' action is currently an open problem and several possibilities are possible.

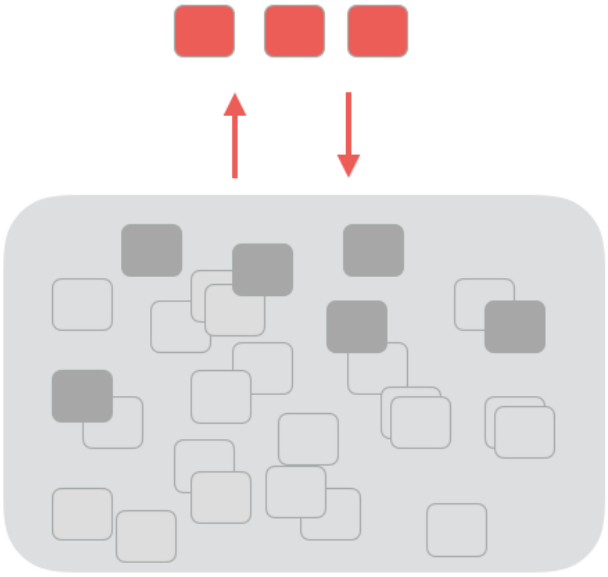

Figure 5: A modules activation "operating system." A small number of active modules (red) can be constantly evaluated as to their relevance. At the same time, potential substitute models (grey) can be activated on the basis of their expected reward. If they become more valuable than an active module, they can be swapped in.

1. In the case of compatible actions such as headings, choose the average:

$$a = \sum_i a^{(i)}$$

2. choose the action recommended by

$$\max_a Q^i(s, a)$$

3. Choose the softmax. Pick the action probabilistically from

$$\frac{P(Q^{(i)}(s, a)}{\sum_i P(Q^{(i)}(s, a)}$$

4.1 Module credit assignment

In the formulation of reinforcement learning up to this point it has been assumed that at each action a reward $R(s, a)$ is handed out. However in a biological setting it might be more appropriate to assume that, any time t only the sum G_t is known, where

$$G_t = \sum_i R^{(i)}(s, a)$$

In this situation there is an efficient algorithm to recover the appropriate reward estimates. The key assumption is that each module has access to the co-active module's reward estimates in addition to the instantaneous global reward G_t. Given this constraint, we can iteratively update the estimates as follows.

$$\hat{R}^{(i)}(s,a) = \beta \hat{R}^{(i)}(s,a) + (1-\beta)[G_t - \sum_{jnei} \hat{R}^{(j)}(s,a)]$$

This formula is vert stable and allows modules to learn their reards sor each s, a pair by being active with different sets of other modules.

4.2 Module inverse reinforcement learning

The inverse reinforcement learning is describe as follows. Suppose we are given episodes of state-action pairs that were generated by another agent, say agent 2. Can an appropriate policy for an observing agent, say agent 1.

The original algorithm by Ramanchandran and Amir for solving this problem used a gradient search in the context of a standard RL algorithm. For each perturbation of reward, the RL algorithm had to be resolved, a very expensive process.

The algorithm that we use makes additional assumptions as to the process, but is vastly cheaper that that of R & A. The asumption is that agent 1 has all the modules that are used by agent 2, but does not have to scale the modules with respect to each other. The initial setup is shown below

Observations from a subject:

$$O_{\mathcal{X}} = \{(s_1, a_1), (s_2, a_2) \dots (s_k, a_k)\}$$

Probability of seeing the individual states/actions, given the reward, is a product of individual probs.

$$\begin{aligned} Pr_{\mathcal{X}}(O_{\mathcal{X}}|\boldsymbol{R}) &= Pr_{\mathcal{X}}((s_1, a_1)|\boldsymbol{R}) Pr_{\mathcal{X}}((s_2, a_2)|\boldsymbol{R}) \\ &\quad \dots Pr_{\mathcal{X}}((s_k, a_k)|\boldsymbol{R}) \end{aligned}$$

Using Bayes

$$Pr_{\mathcal{X}}(\boldsymbol{R}|O_{\mathcal{X}}) = \frac{Pr_{\mathcal{X}}(O_{\mathcal{X}}|\boldsymbol{R}) P_R(\boldsymbol{R})}{Pr(O_{\mathcal{X}})}$$

Let's choose priors for the reward $\mathbf{R}$

Use priors that emphasize sparse rewards, e.g:

$$P_{Laplace}(\boldsymbol{R}(s) = r) = \frac{1}{2\sigma} e^{-\frac{|r|}{2\sigma}}, \forall s \in S$$

$$P_{Beta}(\boldsymbol{R}(s) = r) = \frac{1}{(\frac{r}{R_{max}})^{\frac{1}{2}} (1 - \frac{r}{R_{max}})^{\frac{1}{2}}}, \forall s \in S$$

So to finish up: So instead of having to tediously search through adjustments in reward, the

Lets make the assumption of the following PDF:

$$Pr_{\mathcal{X}}(O_{\mathcal{X}}|\boldsymbol{R}) = \frac{1}{Z} e^{\alpha_{\mathcal{X}} E(O_{\mathcal{X}}, \boldsymbol{R})}$$

And furthermore:

$$E(O_{\mathcal{X}}, \boldsymbol{R}) = \sum_i Q^*(s_i, a_i, \boldsymbol{R})$$

So that:

$$Pr_{\mathcal{X}}((s_i, a_i)|\boldsymbol{R}) = \frac{1}{Z_i} e^{\alpha_{\mathcal{X}} Q^*(s_i, a_i, \boldsymbol{R})}$$

Modular rewards Have individual factors c that are scaled

$$\sum_i c_i = 1$$

Search the space of scale factors to make observed data most probable

$$P(s_j, a_j|Q^*) = \frac{1}{Z} e^{\sum_i c_i Q(s_j^{*(i)}, a_j)}$$

modular inverse reinforcement learning algorithm just has to scale each module with respect to the others, a very easy task.

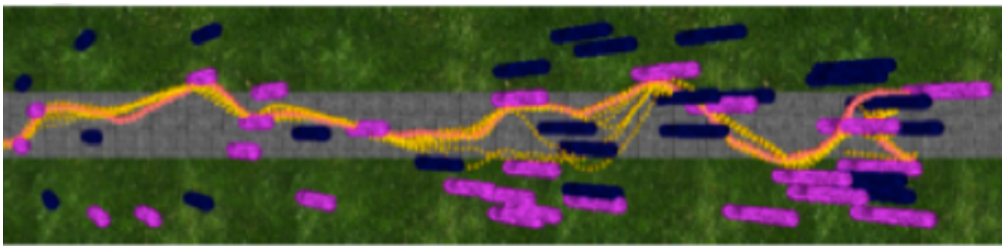

Figure 6: In this IRL overhead view example, agent2 walks down the path using a weighted combination modules' Q tables for staying on the sidewalk, picking up litter (lavender) and avoiding obstacles (blue). Given the orange path's sequence of state and action data, IRL can recover the weights used by agent 2 and use them to generate colored paths(yellow).