Reinforcement Learning

Jose M Vidal

Thu May 8 17:53:48 EDT 2003

This talk is based on

- Tom M. Mitchell. Machine Learning. McGraw Hill. 1997. Chapter 13.
- and his slides.
- An excellent resource is Reinforcement Learning: An Introduction by Sutton and Barto (free!).

1 Introduction

- So far, our machine learning problems have assumed:
  1. that we can present the learner with correct examples,
  2. and that learning goes first, then we test.
- Imagine a robot learning to dock with the batter charger.
- It gets a reward (charge) when successful, but must determine how to encourage actions that led there (temporal credit assignment problem).
  - The reward might be probabilistic.
- It can explore the domain, finding a better route? (exploration vs. exploitation problem)
- The state might only be partially observable.
- It might have to learn several related task with the same environment and using the same sensors.

1.1 TD-Gammon

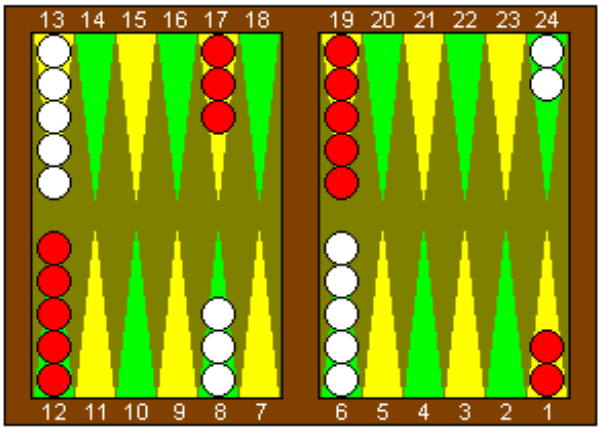


Figure 2. An illustration of the normal opening position in backgammon. TD-Gammon has sparked a near-universal conversion in the way experts play certain opening rolls. For example, with an opening roll of 4-1, most players have now switched from the traditional move of 13-9, 6-5, to TD-Gammon's preference, 13-9, 24-23. TD-Gammon's analysis is given in Table 2.

- The TD-Gammon system learned to play championship Backgammon by playing 1.5 million games against itself.
- It received an immediate reward for +100 for a win, -100 for a loss.
- Equal to the best human.

1.2 Reinforcement Learning Problem

- The agent is in a state s_0, takes action a_0 for which it receives a reward of r_0 and ends up at s_1:

$$s_0 \rightarrow^{r_0} s_1 \rightarrow^{r_1} s_2 \rightarrow^{r_2} s_3 \cdots$$

- We define the problem:

S finite set of states
A finite set of actions
$r_t = r(s_t, a_t)$
$s_{t+1} = \delta(s_t, a_t)$

- We also assume that s_{t+1} and r_t depend *only* on the current state and action.
- That is, the system is a **Markov Decision Process** .
- Note that δ and r might be nondeterministic.
- Also, the agent might not know δ and r

1.3 The Learning Task

- Given that the agent inhabits a Markov process it must now learn action policy $\pi : S \rightarrow A$ that maximizes

$$E[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots]$$

from any starting state in S.

- $0 \leq \gamma$¡1 is the discount factor for future rewards.
- The target function is $\pi : S \rightarrow A$.
- The training examples are of the form $\langle \langle s, a \rangle, r \rangle$

1.4 Value function

- For each possible policy π the agent might adopt, we can define an evaluation function over states

$$V^{\pi}(s) \equiv r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots$$

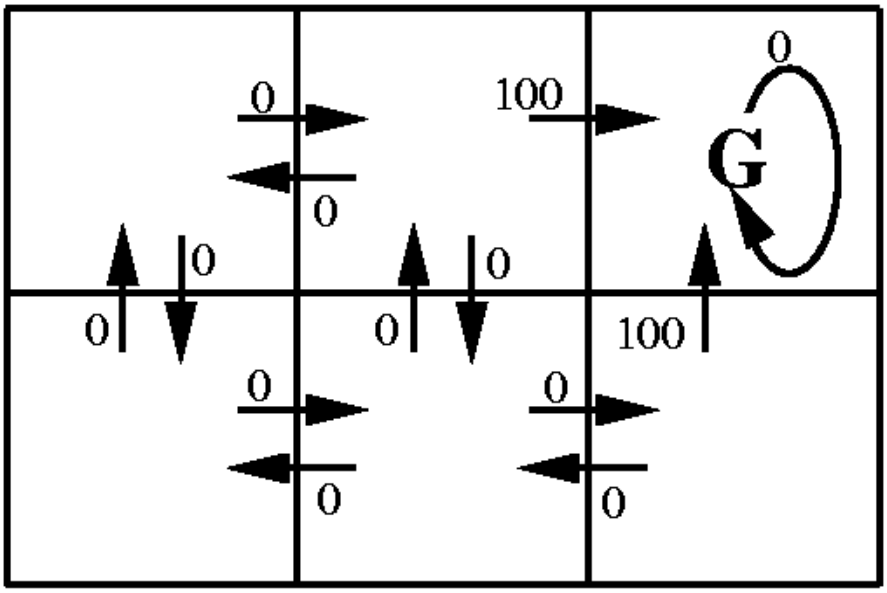
$$V^{\pi}(s) \equiv \sum_{i=0}^{\infty} \gamma^i r_{t+i}$$

where $r_t, r_{t+1}, \ldots$ are generated by following policy π starting at state s

- $V^{\pi}(s)$ is the **discounted cumulative reward** of policy π.
- We can now say that the learning task is to learn the optimal policy π^*

$$\pi^* \equiv \arg\max_{\pi} V^{\pi}(s), (\forall s)$$

1.5 Example



- $Q(s, a)$ values are associated with s, a pairs.
- $V(s)$ values decrease with distance from goal.

2 Q-Learning Motivation

- So we want to learn $V^{\pi^*} \equiv V^*$.
- The agent could just do a lookahead search to choose the best action for each state:

$$\pi^*(s) = \arg\max_a [r(s, a) + \gamma V^*(\delta(s, a))]$$

  easy, right?
- Yes, but only if we know δ and r.
- Most often, we don't.

2.1 Q Function

- Define new function very similar to V^*

$$Q(s, a) \equiv r(s, a) + \gamma V^*(\delta(s, a))$$

If agent learns Q, it can choose optimal action even without knowing δ because this

$$\pi^*(s) = \arg\max_a [r(s,a) + \gamma V^*(\delta(s,a))]$$

is the same as this

$$\pi^*(s) = \arg\max_a Q(s,a)$$

- Our agent will learn Q.

2.2 Learning Q

- We notice that Q and V^* are closely related, specifically:

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

- This allows us to write Q recursively as

$$Q(s_t,a_t) = r(s_t,a_t) + \gamma V^*(\delta(s_t,a_t)))$$

$$Q(s_t,a_t) = r(s_t,a_t) + \gamma \max_{a'} Q(s_{t+1},a')$$

- Now, we let $\hat{Q}$ denote learner's current approximation to Q and use the training rule

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

where s' is the state resulting from applying action a in state s

2.3 Q-Learning Algorithm

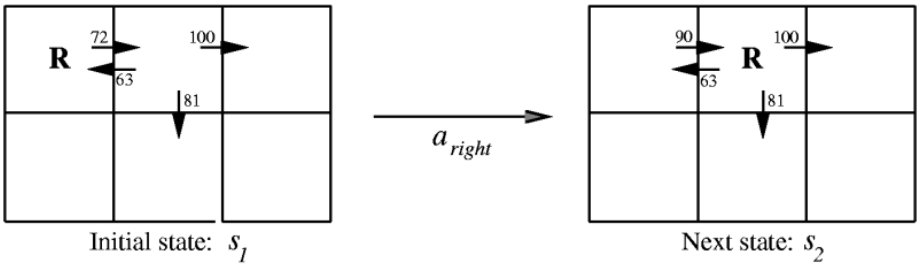
Q-Learning Algorithm

1. For each s, a set $\hat{Q}(s,a) = 0$.
2. Observe the current state s.
3. Select an action a and execute it.
4. Receive reward r.
5. Update the table entry for $\hat{Q}(s,a)$ with

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

6. $s \leftarrow s'$ //Observe the new state.
7. Goto 3.

2.4 Q-Learning Example



- Assumes $\gamma = .9$ and the agent takes the action a_{right} and gets $r = 0$.

2.5 Q-Learning Convergence

- If the system is a deterministic MDP and
- the immediate rewards are bounded (i.e., there is some c s.t. $|r(s,a)|¡c$ and
- the agent visits every possible state infinitely often, then
- Q-learning is proven to converge, eventually.
- That is $\hat{Q}$ will eventually equal Q.

2.6 How to Choose an Action

- Notice that the algorithm did not specify how to choose an action.
- However, convergence requires that every state be visited infinitely often.
- The is the classic explore vs. exploit problem! (aka, the n-armed bandit problem).
- A common strategy is to decide stochastically using

$$P(a_i|s) = \frac{k^{\hat{Q}(s,a_i)}}{\sum_j k^{\hat{Q}(s,a_j)}}$$

2.7 Variations

- Updating after each move can make it take very long to converge if the only reward is at the end.
- Instead, we can save the whole set of rewards and update at the end, in reverse order.
- Another technique is to store past state-action transitions and their rewards and re-train on them periodically.
- This helps when the Q values of the neighbors have changed.

2.8 Nondeterministic Rewards and Actions

- The Q-learning algorithm mentioned only works for deterministic r and δ. Lets fix it.
- First we redefine V, Q by taking expected values

$$V^{\pi}(s) \equiv E[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots$$

$$V^{\pi}(s) \equiv E\left[\sum_{i=0}^{\infty} \gamma^i r_{t+i}\right]$$

so that

$$Q(s,a) \equiv E[r(s,a) + \gamma V^*(\delta(s,a))]$$

2.8.1 Nondeterministic Q-Learning

- We can then alter the training rule to be

$$\hat{Q}_n(s,a) \leftarrow (1-\alpha_n)\hat{Q}_{n-1}(s,a) + \alpha_n[r + \max_{a'} \hat{Q}_{n-1}(s',a')]$$

where

$$\alpha_n = \frac{1}{1 + \text{visits}_n(s,a)}$$

- Note that when $\alpha = 1$ we have the old equation. As such, all we are doing is making revisions to $\hat{Q}$ more gradual.
- We can still prove convergence of $\hat{Q}$ to Q.

3 Temporal Difference Learning

- Q learning acts to reduce the difference between successive Q estimates, in one-step time differences:

$$Q^{(1)}(s_t, a_t) \equiv r_t + \gamma \max_a \hat{Q}(s_{t+1}, a)$$

So, why not use two steps?

$$Q^{(2)}(s_t, a_t) \equiv r_t + \gamma r_{t+1} + \gamma^2 \max_a \hat{Q}(s_{t+2}, a)$$

Or n steps?

$$Q^{(n)}(s_t, a_t) \equiv r_t + \gamma r_{t+1} + \dots + \gamma^{(n-1)} r_{t+n-1} + \gamma^n \max_a \hat{Q}(s_{t+n}, a)$$

- We can blend all these by letting λ be the number of steps we wish to use:

$$Q^{\lambda}(s_t, a_t) \equiv (1-\lambda)\left[Q^{(1)}(s_t, a_t) + \lambda Q^{(2)}(s_t, a_t) + \lambda^2 Q^{(3)}(s_t, a_t) + \dots\right]$$

3.1 Temporal Difference

- We can then rewrite it as a recursive function

$$Q^\lambda(s_t, a_t) = r_t + \gamma[(1-\lambda)\max_a \hat{Q}(s_t, a_t) + \lambda Q^\lambda(s_{t+1}, a_{t+1})]$$

- The TD(λ) algorithm (by Sutton) uses this training rule.
- TD(λ) sometimes converges faster than Q-learning.
- TD-Gammon uses it.

4 Generalization from Examples

- Often there are way too many states to build a table,
- or the state is not completely visible.
- We can fix this by replacing $\hat{Q}$ with a neural net or other generalizer.
  1. For example, encode s, a as the network inputs and train it to output the target values of $\hat{Q}$ given by the training rules.
  2. Or, train a separate network for each action, using the state as input and $\hat{Q}$ as output.
  3. Or, train one network with the state as input but with one $\hat{Q}$ output for each action.
- TD-Gammon used neural nets and Backpropagation.

This talk is available at `http://jmvidal.cse.sc.edu/talks/reinforcementlearning`