CMPT 882: „Machine Learning“
spring semester 2004

Scribe of March, 23rd 2004, by Jakob von Recklinghausen (jvonreck@sfu.ca)

Reinforcement Learning

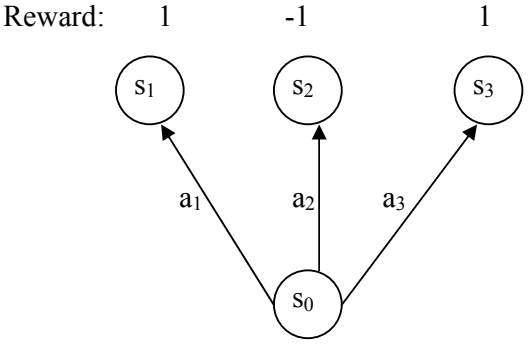
> *"Reinforcement learning (...) is a computational approach to learning whereby an agent tries to maximize the total amount of reward it receives when interacting with a complex, uncertain environment."*
>
> *Barto, Sutton, inside-cover*

So far we looked at problems where the learner were given input-output pairs of the form $<x_i,f(x_i)>$ and should give us a function approximation. But here the learner gets no a priori feedback but has to explore the environment itself.

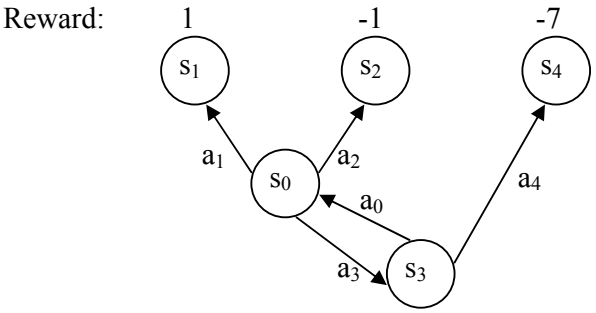
Introduction

states $S = \{s_0, s_1, s_2, \ldots, s_n\}$ (internal (discrete) representation of environment)
actions $A = \{a_0, a_1, a_2, \ldots, a_n\}$ (possible actions the learner can perform)
rewards $R = \{r_0, r_1, r_2, \ldots, r_n\}$ (direct reward from the environment)

Not in every state there is an immediate reward, but maybe different actions in this state lead to different successor-states which themselves have an immediate reward. Problem: What is the utility of intermediate state?



The utility of s_0 is 1, because I can choose a_1 or a_3 and will get reward 1. This is easy if we have a tree-structure. But suppose the following setting:



If the learner is in state s_0, it could perform action a_1 and get reward 1, but it also could perform action a_3 with the plan to perform actions a_0 and a_1 afterwards and get the same reward in the end. As long as going in circles doesn't cost anything, you can go in circles.

Before exploring the formal settings of reinforcement learning that deal with the problems mentioned above, we look at a successful example:

TD-Gammon

In 1995 Tesauro presented a backgammon playing agent who uses reinforcement learning.

The backgammon world has the problem that the only states with immediate rewards are the final states (Win or Loss). But how to assign utilities to intermediate states?

Tesauro let the agent play against itself (1.5 million times). Normally the self-play-trick doesn't work fine, because, for example in a chess setting, a bad agent never visits interesting states. Playing against a more proficient player later, the agent is forced into states it didn't visit before and therefore in which it doesn't know what to do. But because in backgammon there are dices which always lead the game into new positions, self-playing worked out fine.

Because it is very memory consuming to have a complete transition table, Tesauro used a Neuronal Net to calculate an approximation function for the function from states to rewards / utilities. This is easily done, if you assume one input node per field, whose activation level represents the number of figures on this field (one color positive, the other negative?), and whose output is the V*-value for this state.

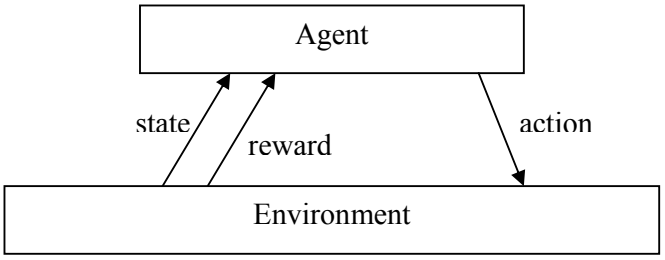
Other Applications

- other games (e. g. Checkers)
- robot learning to dock on battery charger
- optimize factory output
- real-time scheduling of passenger-elevators

Problems R-Learners have to Manage

- delayed reward
- state is only partially observable
- bottleneck: huge state space and learner might not come to visit all states
- trade-off between exploration (of unknown states) and exploitation (of known rewards)

Formal Setting



In general, the agent is in state s_0, performs an action a_0 (message to environment), gets an immediate reward r_0 (message from environment, often 0) and is told the new state s_1 (message from environment).

$$s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} r_1\ s_2 \xrightarrow{a_2} r_2\ s_3$$

Goal: learn to choose actions that maximize the total reward.

Reward Models

There are different models how to calculate the total reward for a plan (chain of actions):

1. Discounted cumulative reward, ∞-horizon: $\sum_{i=0}^{\infty} \gamma^i r_{t+i}$
   (This model is often used in economics and serves the purpose of introducing a reward discount dependent on time spent, steps made, or the like, so that earlier rewards count for more)

2. Cumulative, finite horizon: $\sum_{i=0}^{h} r_{t+i}$

3. Average reward: $\lim_{h\to\infty} \frac{1}{h} \sum_{i=0}^{h} r_{t+i}$

Problem of Infinity

Note that we assumed that the number of states and actions is finite. But in the real world we have an infinite number of states. On the one hand side you could say, this problem is only the problem of representing a continuous external world in a discrete manner. But granted this, the memory of the agent could be part of the state. And in the real world we would assume that no agent ever has the same state of memory. Resolution: restrict size of memory.

Markov Assumption

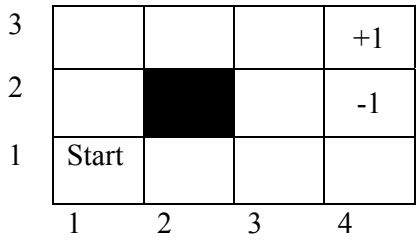
Assuming we have a finite amount of states and actions, then the Markov assumption is that r_t, s_{t+1} depend only on current state s_t and action a_t. We have two important functions:

1. Reward function $r_t = r(s_t, a_t)$ means that the reward at time step t depends only on the state and the action
2. State transition function $s_{t+1} = \delta(s_t, a_t)$ means that new state depends only on old state and action.

These functions are part of the environment and not necessarily known to the agent; and they might be stochastic:

1. $P(S_{t+1} = s' | S_t = s, A_t = a) = P(s' | s, a)$ means that given state s and action a, there is a certain probability that the agent comes into state s'.
2. $P(R_t = r | S_t = s, A_t = a, S_{t+1} = s_{t+1}) = P(r | s_t, a_t, s_{t+1})$ means that given state s_t, action a_t and successor state s_{t+1} there is a certain probability that the agent gets reward r.

A Simple Example



Agent can move north, south, east and west, terminates reaching [4,2] or [4,3] (goal-states).

Markov transition model: $M_{ij}^{a} = P(S_{t+1} = j | S_t = i, A = a)$

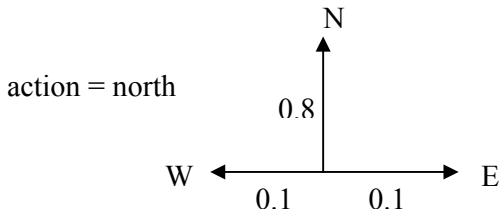
This is a general transition model. The deterministic case would be $M_{ij}^{a} \in \{0,1\}$

BUT: If actions are not reliable, we might have the following case:

$M_{[ix,iy][ix,iy+1]}^{north} = 0.8$

$M_{[ix,iy][ix-1,iy]}^{north} = 0.1$

$M_{[ix,iy][ix+1,iy]}^{north} = 0.1$



The Agent's Learning Task

The learning task of the agent is to execute actions in the environment, observe results, and learn an action policy $\pi : S \rightarrow A$ that maximizes the discounted cumulative reward from any starting state in S.

Here we see again that reinforcement learning is not simply function approximation by examples: We would need examples of the form $< s, a >$, but have examples of the form $<< s, a >, r >$.

We call the discounted cumulative reward also the *utility*:

Utility of policy π at $s_0 \in S \equiv \sum_{t=0}^{\infty} \gamma^t r(s_t)$ with $s_{t+1} = \delta(s_t, \pi(s_t))$

In the non-deterministic case we talk about the expected utility

Expected utility of policy π at s is

$$U_\pi(s) = r(s) + \gamma \sum_{s'} P(s' | s, \pi(s)) * U_\pi(s')$$

The goal then is to find an

optimal policy $\pi^* = \arg\max_\pi (U_\pi(s))$ for all s

Immediate Reward and Total Utility

Instead of using the discounted cumulative reward, we can sometimes design the environment in a manner, that every state transition has a certain cost. Then a shorter way to the same result is always more desirable. This can easily be illustrated with our simple example above:

Define the immediate reward in the following way:

$$r(s_{t+1}) = \begin{cases} -0.04 \text{ if } s_{t+1} \neq [4,2] \text{ or } [4,3] \\ 0.96 \text{ if } s_{t+1} = [4,3] \\ -1.04 \text{ if } s_{t+1} = [4,2] \end{cases}$$

Define an episode as a series of states $[s_0, s_1, s_2, \ldots, s_n]$ through which the agent went until it ended up in s_n (normally used for a series of states which leads from a start state to a goal state).

Then the utility of an episode is $\begin{cases} 1 - n * 0.04 \text{ if } s_n = [4,3] \\ -1 - n * 0.04 \text{ if } s_n = [4,2] \\ n * 0.04 \text{ else} \end{cases}$

With this setup it will always be higher rewarded to reach a goal state in a shorter time.