Kevin Murphy, 1998.

See also Rich Sutton's FAQ on RL

A brief introduction to reinforcement learning

Reinforcement learning is the problem of getting an agent to act in the world so as to maximize its rewards. For example, consider teaching a dog a new trick: you cannot tell it what to do, but you can reward/punish it if it does the right/wrong thing. It has to figure out what it did that made it get the reward/punishment, which is known as the credit assignment problem. We can use a similar method to train computers to do many tasks, such as playing backgammon or chess, scheduling jobs, and controlling robot limbs.

We can formalise the RL problem as follows. The environment is a modelled as a stochastic finite state machine with inputs (actions sent from the agent) and outputs (observations and rewards sent to the agent)

- State transition function P(X(t)|X(t-1),A(t))
- Observation (output) function P(Y(t) | X(t), A(t))
- Reward function E(R(t) | X(t), A(t))

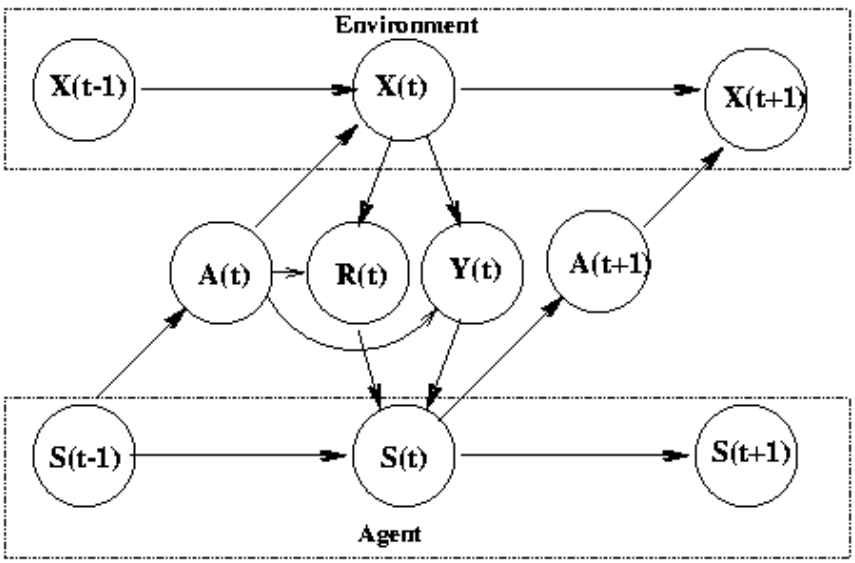
(Notice that what the agent sees depends on what it does, which reflects the fact that perception is an active process.) The agent is also modelled as stochastic FSM with inputs (observations/rewards sent from the environment) and outputs (actions sent to the environment).

- State transition function: S(t) = f (S(t-1), Y(t), R(t), A(t))
- Policy/output function: A(t) = pi(S(t)))

The agent's goal is to find a policy and state-update function so as to maximize the the expected sum of discounted rewards

```
E  [ R_0 + g R_1 + g^2 R_2 + ...]  = E sum_{t=0}^infty gamma^t R_t
```

where 0 <= gamma <= 1 is a discount factor which models the fact future reward is worth less than immediate reward (because tomorrow you might die). (Mathematically, we need gamma < 1 to make the infinite sum converge, unless the environment has absorbing states with zero reward.)



In the special case that Y(t)=X(t), we say the world is fully observable, and the model becomes a Markov Decision Process (MDP). In this case, the agent does not need any internal state (memory) to act optimally. In the more realistic case, where the agent only gets to see part of the world state, the model is called a Partially Observable MDP (POMDP), pronounced "pom-dp". We give a bried introduction to these topics below.

MDPs

A Markov Decision Process (MDP) is just like a Markov Chain, except the transition matrix depends on the action taken by the decision maker (agent) at each time step. The agent receives a reward, which depends on the action and the state. The goal is to find a function, called a policy, which specifies which action to take in each state, so as to maximize some function (e.g., the mean or expected discounted sum) of the sequence of rewards. One can formalize this in terms of Bellman's equation, which can be solved iteratively using policy iteration. The unique fixed point of this equation is the optimal policy.

More precisely, let us define the transition matrix and reward functions as follows.

```
T(s,a,s') = Pr[S(t+1)=s' | S(t)=s, A(t)=a]
R(s,a,s') = E[R(t+1)| S(t)=a, A(t)=a, S(t+1)=s']
```

(We are assuming states, actions and time are discrete. Continuous MDPs can also be defined, but are usually solved by discretization.)

We define the value of performing action a in state s as follows:

```
Q(s,a) = sum_s' T(s,a,s') [ R(s,a,s') + g V(s') ]
```

where 0 < g <= 1 is the amount by which we discount future rewards, and V(s) is overall value of state s, given by Bellman's equation:

```
V(s) = max_a Q(s,a) = max_a sum_s' T(s,a,s') [ R(s,a,s') + g V(s) ]
```

In words, the value of a state is the maximum expected reward we will get in that state, plus the expected discounted value of all possible successor states, s'. If we define

```
R(s,a) = E[ R(s,a,s') ] = sum_{s'} T(s,a,s') R(s,a,s')
```

the above equation simplifies to the more common form

```
V(s) = max_a R(s,a) + sum_s' T(s,a,s') g V(s')
```

which, for a *fixed policy* and a *tabular (non-parametric) representation* of the V/Q/T/R functions, can be rewritten in matrix-vector form as V = R + g T V Solving these n simultaneous equations is called value determination (n is the number of states).

If V/Q satisfies the Bellman equation, then the greedy policy

```
  p(s) = argmax_a Q(s,a)
```

is optimal. If not, we can set p(s) to argmax_a Q(s,a) and re-evaluate V (and hence Q) and repeat. This is called policy iteration, and is guaranteed to converge to the unique optimal policy. (Here is some Matlab software for solving MDPs using policy iteration.) The best theoretical upper bound on the number of iterations needed by policy iteration is exponential in n (Mansour and Singh, UAI 99), but in practice, the number of steps is O(n). By formulating the problem as a linear program, it can be proved that one can find the optimal policy in polynomial time.

For AI applications, the state is usually defined in terms of state *variables*. If there are k binary variables, there are n = 2^k states. Typically, there are some independencies between these variables, so that the T/R functions (and hopefully the V/Q functions, too!) are structured; this can be represented using a Dynamic Bayesian Network (DBN), which is like a probabilistic version of a STRIPS rule used in classical AI planning. For details, see

- "Decision Theoretic Planning: Structural Assumptions and Computational Leverage".
  Craig Boutilier, Thomas Dean and Steve Hanks
  JAIR (Journal of AI Research) 1999. Postscript (87 pages)

Reinforcement Learning

If we know the model (i.e., the transition and reward functions), we can solve for the optimal policy in about n^2 time using policy iteration. Unfortunately, if the state is composed of k binary state *variables*, then n = 2^k, so this is way too slow. In addition, what do we do if we don't know the model?

Reinforcement Learning (RL) solves both problems: we can approximately solve an MDP by replacing the sum over all states with a Monte Carlo approximation. In other words, we only update the V/Q functions (using temporal difference (TD) methods) for states that are actually visited while acting in the world. If we keep track of the transitions made and the rewards

received, we can also estimate the model as we go, and then "simulate" the effects of actions without having to actually perform them.

There are three fundamental problems that RL must tackle: the exploration-exploitation tradeoff, the problem of delayed reward (credit assignment), and the need to generalize. We will discuss each in turn.

We mentioned that in RL, the agent must make trajectories through the state space to gather statistics. The *exploration-exploitation tradeoff* is the following: should we explore new (and potentially more rewarding) states, or stick with what we know to be good (exploit existing knowledge)? This problem has been extensively studied in the case of k-armed bandits, which are MDPs with a single state and k actions. The goal is to choose the optimal action to perform in that state, which is analogous to deciding which of the k levers to pull in a k-armed bandit (slot machine). There are some theoretical results (e.g., Gittins' indices), but they do not generalise to the multi-state case.

The problem of delayed reward is well-illustrated by games such as chess or backgammon. The player (agent) makes many moves, and only gets rewarded or punished at the end of the game. Which move in that long sequence was responsible for the win or loss? This is called the *credit assignment* problem. We can solve it by essentially doing stochastic gradient descent on Bellman's equation, backpropagating the reward signal through the trajectory, and averaging over many trials. This is called temporal difference learning.

It is fundamentally impossible to learn the value of a state before a reward signal has been received. In large state spaces, random exploration might take a long time to reach a rewarding state. The only solution is to define higher-level actions, which can reach the goal more quickly. A canonical example is travel: to get from Berkeley to San Francisco, I first plan at a high level (I decide to drive, say), then at a lower level (I walk to my car), then at a still lower level (how to move my feet), etc. Automatically learning action hierarchies (temporal abstraction) is currently a very active research area.

The last problem we will discuss is *generalization*: given that we can only visit a subset of the (exponential number) of states, how can know the value of all the states? The most common approach is to approximate the Q/V functions using, say, a neural net. A more promising approach (in my opinion) uses the factored structure of the model to allow safe state abstraction (Dietterich, NIPS'99).

RL is a huge and active subject, and you are recommended to read the references below for more information.

- "Reinforcement Learning: An Introduction",
  Richard Sutton and Andrew Barto, MIT Press, 1998.

- "Neuro-dynamic programming"
  Dimitri P. Bertsekas and John Tsitsiklis, Athena Scientific, 1996.

- "Reinforcement Learning: A Survey".
  Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore
  JAIR (Journal of AI Research), Volume 4, 1996. Postscript (40 pages) or HTML version

- Harmon's tutorial on RL

- Sutton's RL software

There have been a few successful applications of RL. The most famous is probably Tesauro's TD-gammon, which learned to play backgammon extremely well, using a neural network function approximator and TD(lambda). Other applications have included controlling robot arms and various scheduling problems. However, these are still very simple problems by the standards of AI, and required a lot of human engineering; we are a far cry from the dream of fully autonomous learning agents.

POMDPs

MDPs assume that the complete state of the world is visible to the agent. This is clearly highly unrealistic (think of a robot in a room with enclosing walls: it cannot see the state of the world outside of the room). POMDPs model the information available to the agent by specifying a function from the hidden state to the observables, just as in an HMM. The goal now is to find a mapping from observations (not states) to actions. Unfortunately, the observations are not Markov (because two different states might look the same), which invalidates all of the MDP solution techniques. The optimal solution to this problem is to construct a belief state MDP, where a belief state is a probability distribution over states. For details on this approach, see

- "Planning and Acting in Partially Observable Stochastic Domains".
  Leslie Pack Kaelbling, Michael L. Littman and Anthony R. Cassandra
  Artificial Intelligence, Vol. 101, 1998. Postscript (45 pages)

Control theory is concerned with solving POMDPs, but in practice, control theorists make strong assumptions about the nature of the model (typically linear-Gaussian) and reward function (typically negative quadratic loss) in order to be able to make theoretical guarantees of optimality, etc. By contrast, optimally solving a generic discrete POMDP is wildly intractable. Finding tractable special cases (e.g., structured models) is a hot research topic.

For more details on POMDPs, see Tony Cassandra's POMDP page.

First-order models

A major limitation of (PO)MDPs is that they model the world in terms of a fixed-size set of state variables, each of which can take on specific values, say true and false, or -3.5. These are called propositional models. It would seem more natural to use a *first-order model*, which allows for (a variable number of) *objects and relations*. However, this is a completely open research problem. Leslie Kaelbling is doing interesting work in this area (see her "reifying robots" page).

Recommended reading

Books

- "Reinforcement Learning: An Introduction",
  Richard Sutton and Andrew Barto, MIT Press, 1998.

- "Neuro-dynamic programming"
  Dimitri P. Bertsekas and John Tsitsiklis, Athena Scientific, 1996.

Papers

- "Reinforcement Learning: A Survey".
  Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore
  JAIR (Journal of AI Research), Volume 4, 1996. Postscript (40 pages) or HTML version

- "Planning and Acting in Partially Observable Stochastic Domains".
  Leslie Pack Kaelbling, Michael L. Littman and Anthony R. Cassandra
  Artificial Intelligence, Vol. 101, 1998. Postscript (45 pages)

- "Decision Theoretic Planning: Structural Assumptions and Computational Leverage".
  Craig Boutilier, Thomas Dean and Steve Hanks
  JAIR (Journal of AI Research) 1999. Postscript (87 pages)

Online stuff

-->
- Tony Cassandra's POMDP page
- Perez-Uribe's list of RL references
- Harmon's tutorial on RL
- Michael Kearns' list of recommended reading
- Stuart Reyonold's bibliography, contains pointers to many good online articles