Value Iteration, Policy Iteration, and Q-Learning

April 6, 2009

1 Introduction

In many cases, agents must deal with environments that contain nondeterminism. Perhaps the agent's actions can fail, or its sensors can be inaccurate, or outside forces might change the environment. In this assignment we will focus on a particular sort of nondeterminism: nondeterministic actions.

Recall that in the case of deterministic environments, we can describe the effect of an action A taken in a state S using a *transition function* $S \times A \rightarrow S$. In other words, if an agent is in a particular state and takes a particular action, it will always wind up in the same successor state.

Deterministic environments are nice, because they let us apply search algorithms such as A* to find an optimal sequence of actions. However, many environments are not deterministic. A block might slip out of a robot's gripper, or a web page might fail to load, or a doorway might be blocked. In these environments, taking the same action from the same state could lead to two different outcomes. In this case, our transition function takes as input a state s, an action a, and a successor state s' and returns the probability of reaching s' from s when taking action a.

We can alternatively think of a probability distribution being attached to state-action pairs, indicating each possible successor state and the probability of that state being reached.

1.1 Markovian environments

We should pause at this point to discuss the role of history in determining a successor state. In some cases, the probability of reaching state s' from s given action a might also depend on states that were visited previous to s. If the number of previous states can be bounded, we call this environment *Markovian* or say that the environment obeys the *Markov assumption.*

In particular, we will focus on problems in which the number of previous states is bounded at 1, and the transition from s to s' does not depend on any history. This is what is called a *first-order Markov process* or a *Markov decision process (MDP)*. There are two reasons for this assumption:

- Reasoning about long chains of events is complicated and expensive.
- It is usually possible to encapsulate any important history in a state variable and convert such a problem into a first-order MDP.

We will also assume that the transition probabilities do not change while our agent is solving the MDP. This is what is referred to as a *stationary distribution*.

1.2 Solution concepts

In deterministic sequential environments, our solution concept was a series of actions that would move us from a start state to a goal state. In a nondeterministic sequential environment, this will not work, as we cannot be guaranteed of a unique path through a state space with any series of actions.

Instead, we construct a *policy*. A policy (typically denoted π) is a mapping from states to actions that tells our agent the optimal action to take in any given state with respect to a particular goal.

One thing to note is that a policy is only useful with respect to a particular goal that our agent wants to achieve. If it changes goals, it will need a new policy.

1.3 Markov Decision Processes

Markov decision processes (or MDPs) are a way to model sequential decision making under uncertainty. To formalize this, we must introduce a few concepts.

- Our problem has an initial state s_0
- States and actions are discrete.
- Each state in our world will have a reward r associated with it.
- Our problem has a transition function $T(s, a, s') \rightarrow [0, 1]$ that indicates the probability of transitioning from state s to s' when action a is taken.
- A discount factor $\gamma \in [0, 1]$ applied to future rewards. This represents the notion that a current reward is more valuable that one in the future. If γ is near zero, future rewards are almost ignored; a γ near one places great value on future reward.

We can then formulate the total reward from a policy π as:
$TR = EU(\sum_{t=0}^{\infty} \gamma^t R(s_t) | \pi)$

In words, this says that the reward from a policy is the sum of the discounted expected utility of each state visited by that policy.

The optimal policy is the policy that maximizes this equation. In this assignment, we will look at three algorithms for discovering this policy.

2 Value Iteration

The first algorithm we will look at is value iteration. The essential idea behind value iteration is this: if we knew the true value of each state, our decision would be simple: always choose the action that maximizes expected utility. But we don't itinially know a state's true value; we only know its immediate reward. But, for example, a state might have low initial reward but be on the path to a high-reward state.

The true value (or utility) of a state is the immediate reward for that state, plus the expected discounted reward if the agent acted optimally from that point on:

$U(s) = R(s) + \gamma max_a \sum_{s'} T(s, a, s')U(s')$

Note that the value for each state can potentially depend on all of its neighbors' values. If our state space is acyclic, we can use dynamic programming to solve this. Otherwise, we use value iteration.

1. Assign each state a random value

2. For each state, calculate its new U based on its neighbor's utilities.

3. Update each state's U based on the calculation above:

   $U_{i+1}(s) = R(s)\gamma max_a \sum_{s'} T(s, a, s')U_i(s')$

4. if no value changes by more than δ, halt.

This algorithm is guaranteed to converge to the optimal (within δ) solutions.

3 Policy Iteration

Value iteration works fine, but it has two weaknesses: first, it can take a long time to converge in some situations, even when the underlying policy is not changing, and second, it's not actually doing what we really need. We actually don't care what the value of each state is; that's just a tool to help us find the optimal policy. So why not just find that policy directly?

We can do so by modifying value iteration to iterate over policies.We start with a random policy, compute each state's utility given that policy, and then select a new optimal policy. Here's the actual algorithm:

1. Create a random policy by selecting a random action for each state.

2. While not done:

   (a) Compute the utility for each state given the current policy.

   (b) Update state utilities

   (c) Given these new utilities, select the optimal action for each state.

3. If no action changes, halt

4 Q-learning

Value Iteration and Policy Iteration work wonderfully for determining an optimal policy, but they assume that our agent has a great deal of domain knowledge. Specifically, they assume that the agent accurately knows the transition function and the reward for all states in the environment. This is actually quite a bit of information; in many cases, our agent may not have access to this.

Fortunately, there is a way to learn this information. In essence, we can trade learning time for *a priori* knowledge. One way to do this is through a form of reinforcement learning known as *Q-learning*. Q-learning is a form of model-free learning, meaning that an agent does not need to have any model of the environment; it only needs to knoe what states exist and what actions are possible in each state.

The way this works is as follows: we assign each state an estimated value, called a *Q-value*. When we visit a state and receive a reward, we use this to update our estimate of the value of that state. (Since our rewards might be stochastic, we may need to visit a state many times.)

Our Q-value can be written as:

$Q(s,a) = R(s,a) + \gamma max_{a'} Q(s',a')$

This says that the value of taking action a in state s is the immediate reward for the (s,a) pair, plus the value of the best possible state-action pair for the successor state.

We can use this to update the Q-value of a state-action pair as a reward r is observed:

$Q_{t+1}(s,a) = r + \gamma max_{a'} Q_t(s',a')$.

We still have one problem left to solve, however. How should an agent choose which action to test? We are assuming that the agent is learing in an *online* fashion. This means that it is interleaving learning and execution. This brings up a trade-off: learning is costly; time that our agent spends making mistakes is time that cannot be spent performing correct actions. One the other hand, time is needed to learn, and some time spent making errors early on can lead to improved overall performance.

Our intuition is as follows: In the early stages of execution, when little is known about the world, it is important to explore and try unknown actions. There is a great deal to be gained from learning, and the agent does not have enough information to act well in any case. Later in its life, the agent may want to almost always choose the action that looks best; there is little information to be gained, and so the value of learning is negligible.

We will model this with a function that assigns a probability of being chosen for each possible action in a given state. This function should tend to choose actions with higher Q values, but should sometimes select lower Q-value actions. The probability of selecting the highest Q-value action should increase over time.

We will use a distribution known as a Boltzmann distribution to do this. The Boltzmann distribution is as follows:

$P(a|s) = \frac{e^{Q(s,a)/k}}{\sum_j e^{Q(s,a_j)/k}}$

The k parameter (often referred to as temperature) controls the probability

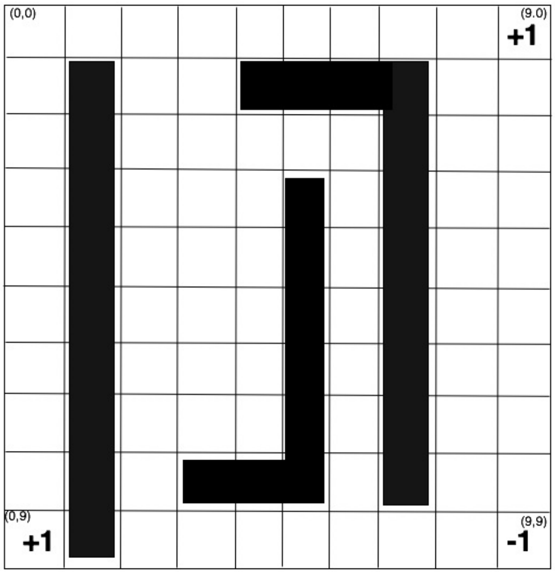

Figure 1: Map for problem 2. Agent moves in the intended direction with $p = 0.7$, and in each other direction with $p = 0.1$.

of selecting non-optimal actions. If k is large, all actions will be selcted fairly uniformly. If k is close to zero, the best action will always be chosen. We begin with k large and gradually descrease it over time.

5 Assignment

In this assignment, you will solve an MDP using each of the algorithms discussed above for two different problems. Skeleton code in Python has been provided for you to help get you started.

The first problem is the same map that is found in Chapter 17 of Russell and Norvig. This is included to help you debug your code; it is strongly recommended that you start with this problem. The second map can be seen below; it is larger and more complex. In this second map, an agent moves in the intended direction with $p = 0.7$ and each of the other three directions.

The code begins by defining a State. A state has a reward, a set of transitions, a set of legal actions, a policy, and an indicator of whether it is a goal state. It contains methods to compute the expected utility of an action and to find the best action.

The next class defined is the Map. A map is just a set of states, a tolerated error, and a gamma.

To begin, you should implement the valueIteration method within Map. You may assume R=-0.04 for all non-goal states, and γ = 0.8. Begin on the Russell and Norvig problem. Does your solution match the one in the text? If not, why not?

Then apply your algorithm to the second map (you shouldn't need to change any code). Does this version converge to a solution? How many iterations does each map require?

Next, implement the policyIteration method within Map and run it on both of the problems provided. Do ValueIteration and PolicyIteration find the same solutions? What is the time required for each algorithm to find a solution¿

Here is what an instance of running the code might look like:

```
>>> import mdp
>>> m = mdp.makeRNProblem()
>>> m.valueIteration()
>>> [(s.coords, s.utility) for s in m.states.values()]
  [(0, 0), (1, 0.30052947656142465), (2, 0.47206207850545195), (3,
0.68209220953458682), (4, 0.18120337982169335), (5,
0.34406397771608599), (6, 0.09080843870176547), (7,
0.095490116585228102), (8, 0.18785929363720655), (9,
0.00024908649990546677), (10, 1.0), (11, -1.0)]
>>> m.policyIteration()
>>> [(s.coords, s.utility, s.policy) for s in m.states.values()]
  [(0, 0, None), (1, 0.28005761520403155, ’right’), (2, 0.469081407274
```

Once you have implemented both value and policy iteration, run both algorithms on both maps. Do you reach the same solutions in each case? How many iterations does each algorithm require?

Finally, you should complete the q-Learner found in qLearner.py. There are three methods you must complete:

- selectAction: you should use Boltzmann exploration, as described above, to choose an action.
- update: This should update the agent’s Q-table based on the reward received.
- learn: This is a wrapper method. It should start up the agent, set starting temperature values (use 2 to start), set an alpha (use 0.2 to start) and lower the temperature over time.