DD2431 Machine Learning
Lab 4: Reinforcement Learning
Python version

Örjan Ekeberg

October 5, 2010

1 Your Task

In this assignment you will use reinforcement learning to search for a working control algorithm for a simple walking robot. Your first task is to use *policy iteration* while designing a suitable reward strategy which makes it possible for the robot to learn how to walk. In the second part, you will use Q-learning to make the robot learn without having direct access to the model of the environment.

2 System Description

The task we want to solve in this assignment is to control a two-legged robot so that it walks forward. To make the task simple, we ignore most practical problems like balance and dynamics. We will assume that each leg can be in one of four positions: down&back, up&back, up&forward, and down&forward. Since the two legs can be positioned independently, the system can be in any of 16 states (see figure 1).

The control system has only four actions to choose from, according to table 1. Whether the leg moves up or down for action 0 and 2 is determined by the current state: if the leg is up it will move down, if it is down it will move up. Forward–backward is handled correspondingly.

The objective of learning is to find a *policy*, i.e. a rule which states which action to take in each state. What constitutes a good policy is determined by the reward received when interacting with the environment. In this case we will define a reward function which returns positive values when

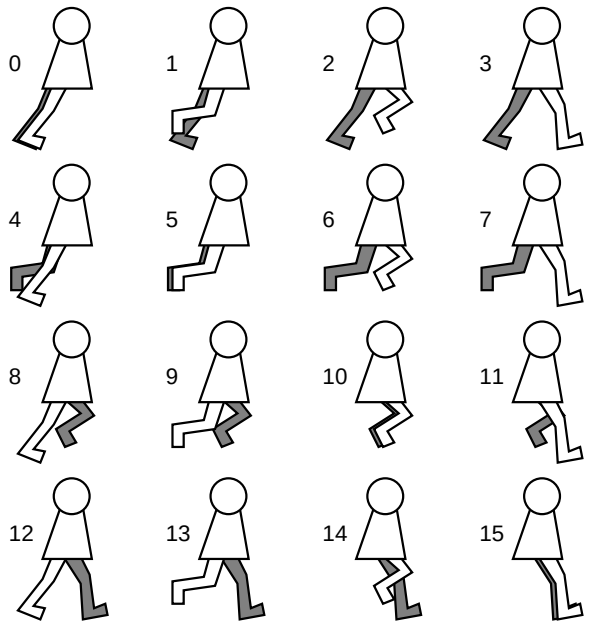

Figure 1: The two-legged robot can be in 16 different states, numbered from 0 to 15 as shown.

Action	Effect
0	Move right leg up or down
1	Move right leg back or forward
2	Move left leg up or down
3	Move left leg back or forward

Table 1: Possible actions and their effect on the robot.

the robot moves forward and negative values when it moves backwards or performs other actions which are undesirable. When working with reward-based algorithms it will usually require some thinking and experimenting to find what rewards are needed to get a desirable behavior.

3 Theoretical Background

Policy iteration is an algorithm which can be used to compute optimal policies given a model of the environment. Policy iteration solves problems where the state transitions and reward functions are known beforehand, and utilizes the principles of dynamic programming. When the environment is unknown and only accessible by interacting with it, the agent will have to use learning to solve the problem. *Temporal difference learning* is a technique which resembles policy iteration but without the need for a known environment.

The central idea of both policy iteration and temporal difference learning is to estimate *value functions*, which in turn can be used to identify the optimal policy. The value function is essential in that it makes the value of predicted future events accessible to the process which decides what to do now.

3.1 Policy Iteration

Policy iteration is an algorithm for calculating the optimal policy when the rules for state transitions and rewards are known beforehand. An important property of this algorithm is that it is able to find solutions which involve *planning* in the sense that actions leading to reward at a later stage will also be considered. This is achieved by estimating a value of each state.

3.1.1 Calculating Values

Each state is assigned a value which is used to guide the search for an optimal policy. The value is defined as the expected total future reward. When the future is infinite one has to include a *discount factor*, γ ($0 < \gamma < 1$), which states that rewards are less valuable if they are received far in the future. A low γ-value means that the planning becomes more shortsighted.

The value of a state depends on what we will do later, i.e. on what policy is being used. This is somewhat of a chicken-and-egg problem: the reason for calculating the values was to be able to find a good policy! This

is normally solved by iteratively computing the policy and values together, a process called *policy iteration*.

First, let us consider how to compute the value function V^π for a given arbitrary policy π. The value function has to obey the *Bellman equation*:

$$V^\pi(s) = r(s, \pi(s)) + \gamma V^\pi(\delta(s, \pi(s))) \tag{1}$$

where $\delta(s, a)$ is the state transition function, and $r(s, a)$ the reward function. This equation can be solved iteratively by using it as an update rule:

$$V^\pi_{k+1} = r(s, \pi(s)) + \gamma V^\pi_k(\delta(s, \pi(s))) \tag{2}$$

The sequence of V^π_k will converge to V^π as $k \to \infty$.

3.1.2 Improving the Policy

Our main motivation for computing the value function for a policy is to construct a better policy. This can be done by, for each state s, selecting the action a which maximizes $r(s, a) + \gamma V^\pi(\delta(s, a))$:

$$\pi'(s) = \arg\max_a \left[r(s, a) + \gamma V^\pi(\delta(s, a))\right] \tag{3}$$

Once a policy π has been improved using V^π to yield a better policy π', we can then compute $V^{\pi'}$ and improve it again to yield an even better π''. *Policy iteration* intertwines policy evaluation and policy improvement according to

$$\begin{aligned} \pi_k(s) &= \arg\max_a \left[r(s, a) + \gamma V_k(\delta(s, a))\right] \\ V_{k+1}(s) &= r(s, \pi_k(s)) + \gamma V_k(\delta(s, \pi_k(s))) \end{aligned} \tag{4}$$

Policy iteration will eventually converge to the optimal policy.

3.2 Temporal Difference Learning

Policy iteration can only be used when a model of the environment is available, i.e. $r(s, a)$ and $\delta(s, a)$ are both known functions. In most situations, however, these need to be estimated from experience. In reinforcement learning, the agent interacts with the environment by performing actions and observing the rewards and new states encountered during this interaction. Most reinforcement learning methods are based on estimating the value of each state.

Temporal difference (TD) is a class of learning methods which improve the estimated state values in every time step and is based on comparing the predicted and actual reward; hence the name "temporal difference". We will use one of the most common TD methods, called *Q-learning*.

3.2.1 Q-learning

Q-learning is a temporal difference learning algorithm which uses the Q-function for estimating the total future reward based on both state and action.

Every combination of state and action is assigned a value, often referred to as the Q-value, defined as the expected total future reward when starting from a particular state (s) and making a particular action (a).

Q-learning is based on comparing two estimates of a Q-value and using the difference to improve the estimate. Let us assume that the agent is in state s and there performs action a. We can use our current $Q(s,a)$ value as a prediction of the future reward. Now, after actually performing the action the environment will respond by moving the agent to a new state s' and giving a reward r. After this step we are actually able to make a better estimate of $Q(s,a)$, namely $r+\gamma \max_{a'} Q(s',a')$. Note that we are using the Q-function for the next state (s') to estimate how much reward we will get after this step. We now have two estimates of $Q(s,a)$:

$$\begin{aligned} Q^{(1)}(s,a) &= Q(s,a) \\ Q^{(2)}(s,a) &= r+\gamma \max_{a'} Q(s',a') \end{aligned} \tag{5}$$

The trick in Q-learning is to use the fact that the second estimate is better than the first and we can therefore update the stored estimate. A fraction η of the difference between $Q^{(2)}$ and $Q^{(1)}$ is added to the estimate of Q for the state-action pair we just left. We get this update formula:

$$Q(s,a) \Leftarrow Q(s,a) + \eta \left[r+\gamma \max_{a'} Q(s',a') - Q(s,a) \right] \tag{6}$$

Q-learning is capable of *off-policy* learning. This means that it is able to estimate the Q-values correctly independently of the policy being followed. To take an extreme example, even if the agent constantly makes random actions, the estimates of the Q-values will converge towards the expected total future rewards when the optimal policy is used. This is because we use the maximum Q-value for the new state (s') in the update rule (6), independently of the policy we actually follow.

3.3 ϵ-Greedy Policy

The agent learns through interaction with the environment and will eventually converge to a good estimate of the Q-values. A prerequisite is that the agent actually visits all states and performs all actions there sufficiently often. This potentially poses a problem, because we normally want the follow the best policy, based on the current Q estimates. This is what is called a *greedy policy*.

All temporal difference methods have a need for active exploration, which requires that the agent every now and then tries alternative actions that are not necessarily optimal according to its current estimates of $Q(s, a)$. An *ϵ-greedy policy* satisfies this requirement, in that most of the time with probability $1 - \epsilon$ it picks the optimal action according to

$$\pi(s) = \arg\max_a Q(s, a) \tag{7}$$

but with small probability ϵ it takes a random action. As the agent collects more and more evidence the policy can be shifted towards a deterministic greedy policy.

The Q-learning algorithm is summarized here:

1. Initialize $Q(s, a)$ arbitrarily $\forall s, a$
2. Initialize s
3. Repeat for each step:
   - Choose a from s using ϵ-greedy policy based on $Q(s, a)$
   - Take action a, observe reward r, and next state s'
   - Update $Q(s, a) \leftarrow Q(s, a) + \eta \left[r + \gamma \max_{a'} Q(s', a') - Q(s, a) \right]$
   - Replace s with s'
4. until T steps

4 Experiments

You will first use policy iteration, i.e. equation (4) to find a policy which makes our simple robot walk. Since all states and actions are discrete we can represent $\pi(s)$ and $V(s)$ as vectors while $\delta(s, a)$ and $r(s, a)$ both correspond to matrices. In Python we can use lists, and lists of lists, respectively. For constant vectors and matrices we can use tuples (and tuples of tuples) which are Pythons datatype for immutable lists.

The transition matrix can be defined like this:

```
trans = ((1, 3, 4, 12),
         (0, 2, 5, 13),
         (3, 1, 6, 14),
         (2, 0, 7, 15),
         (5, 7, 0, 8),
         (4, 6, 1, 9),
         (7, 5, 2, 10),
         (6, 4, 3, 11),
         (9, 11, 12, 4),
         (8, 10, 13, 5),
         (11, 9, 14, 6),
         (10, 8, 15, 7),
         (13, 15, 8, 0),
         (12, 14, 9, 1),
         (15, 13, 10, 2),
         (14, 12, 11, 3))
```

4.1 Policy Iteration

Your first task is to design a suitable reward function which will produce a policy that makes the robot walk. You will have to make reasonable guesses of what actions to reward and what to penalize in order to get a reasonable walking behavior. See how simple you can make the reward function and still get a reasonable walking behavior.

Policy iteration as defined by equation (4) will compute the optimal value function $V(s)$ and policy $\pi(s)$ based on the transition and reward matrices. The following code implements this in Python:

```
for p in range(100):
    for s in range(len(policy)):
        policy[s] = argmax(
            lambda(a):
                rew[s][a] + gamma * value[trans[s][a]],
            range(4))

    for s in range(len(value)):
        a = policy[s]
        value[s] = rew[s][a] + gamma * value[trans[s][a]]
```

We have here used a utility function `argmax` for finding which argument gives the largest value of a function. This can be defined like this:

```
def argmax(f, args):
    mi = None
    m = -1e10
    for i in args:
        v = f(i)
        if v > m:
            m = v
            mi = i
    return mi
```

In this code we have assumed that `rew` is a matrix (stored as a tuple-of-tuples) containing the reward function $r(s, a)$. You will have to define `rew` with suitable values. The reward matrix should be a 16×4 matrix where each row corresponds to a state (according to figure 1), and each column corresponds to an action. The elements should contain the reward received when doing a particular action in a given state.

You will also have to choose a suitable value for the discount factor γ (`gamma`). The lists holding $V(s)$ and $\pi(s)$ can be initialized like this:

```
policy = [None for s in trans]
value = [0 for s in trans]
```

Suggestion: Start with zeros throughout the reward matrix and then enter positive values for actions that correspond to the robot moving forward.

Also enter negative values for actions that should be avoided, such as lifting one leg when the other is already lifted, or moving backward.

Note: You should *only* give reward to actions that actually move the robot forward, not all the intermediate actions necessary such as moving the leg forward when lifted. The idea is that the learning algorithm should do the actual planning of how to move the legs. Therefore, try *not* to guide it by rewarding all steps throughout the whole step cycle.

In order to test the resulting policy, make a short simulation by starting in an arbitrary state and successively making actions according to the policy. Check that the behavior is reasonable, i.e. that it looks like a good walking pattern.

You may use the code in `cartoon.py` or `animate.py` to see the resulting movement graphically.

4.2 Q-Learning

We will now make use of Q-learning. The transition and reward functions should not be directly accessable to the algorithm. To mimic this in your program, you can define a Python class `Environment` like this:

```
class Environment:
    def __init__(self, state=0):
        self.state = state
        self.trans = //insert definition here//
        self.rew = //insert definition here//

    def go(self, a):
        r = self.rew[self.state][a]
        self.state = self.trans[self.state][a]
        return self.state, r
```

This should be the only place where the matrices `trans` and `rew` are used. The idea is that the algorithm will only have indirect access to them via the `go` function, which makes an action and returns both the new state and the reward received.

Implement the Q-learning algorithm. Starting from an arbitrary state, the algorithm should follow its current policy to generate actions which are sent to the `go`-function to move around in the state space. For each move, the Q-values should be updated appropriately. Make sure you use the ϵ-greedy policy (what happens if you don't?).