2D1431 Machine Learning
Lab 3: Reinforcement Learning

Örjan Ekeberg

November 20, 2005

1 Introduction

In this lab you will use reinforcement learning to search for a working control algorithm for a simple walking robot.

We will assume that you are familiar with the basic concepts of reinforcement learning and that you have read chapter 13 in the course book *Machine Learning* (Mitchell, 1997). As supplementary material you may read the first four chapters of the survey on reinforcement learning by Kaelbling et al. (1996). For further reading and a detailed discussion of policy iteration and reinforcement learning, the textbook "Reinforcement Learning" is highly recommended (Sutton and Barto, 1999).

Policy iteration is an algorithm which can be used to compute optimal policies given a model of the environment. Policy iteration solves problems that can be formulated as Markov decision processes, i.e. when 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. Reinforcement based methods are therefore able to do a basic form of planning.

2 Task Description

The task we want to solve in this lab 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 1 and 3 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 task for the algorithm 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 the robot moves forward and negative values when it moves backwards or performs other actions which are undesirable, for example lifting both legs simultaneously (states 6, 7, 10 and 11). 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 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 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 the policy 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

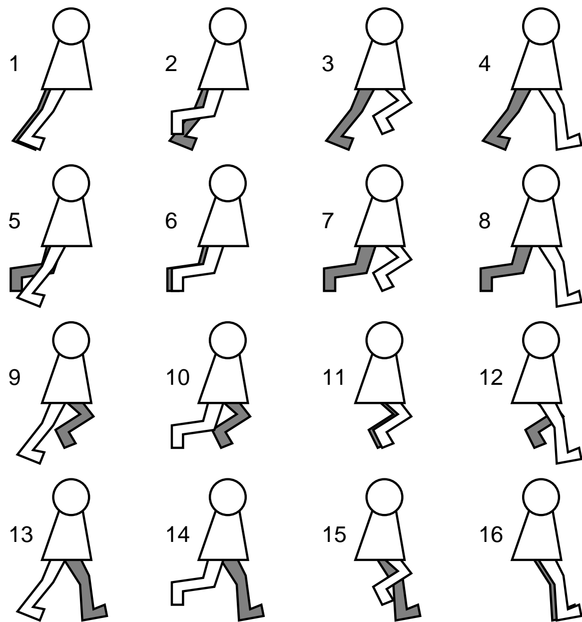

Figure 1: The two-legged robot can be in 16 different states, numbered from 1 to 16 as shown.

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

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

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

First, let us consider how to compute the state-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 $\delta(s, a) : S \times A \to S$ is the state transition function, and $r(s, a) : S \times A \to \Re$ the reward function. This equation can in fact be solved directly, by solving a linear equation of the type

$$V = R + BV \tag{2}$$

where V and R are vectors and B is a matrix. An alternative is to solve equation (1) by successive approximation, considering the Bellmann equation as an update rule

$$V^\pi_{k+1} = r(s, \pi(s)) + \gamma V^\pi_k(\delta(s, \pi(s))) \tag{3}$$

The sequence of V^π_k will converge to V^π as $k \to \infty$. This method is called *iterative policy evaluation*.

3.2 Improving the Policy

Our main motivation for computing the value function for a policy is to construct a better policy. For some state s we can improve our current policy by picking an alternative action $a \neq \pi(s)$ that deviates from our current policy $\pi(s)$ if

$$r(s, a) + \gamma V^\pi(\delta(s, a)) > r(s, \pi(s)) + \gamma V^\pi(\delta(s, \pi(s))). \tag{4}$$

This process is called *policy improvement*. In other words, for each state s we greedily choose the action a that maximizes $r(s, a) + \gamma V^\pi(\delta(s, a))$

$$\pi'(s) = \operatorname*{argmax}_a \left[r(s, a) + \gamma V^\pi(\delta(s, a))\right] \tag{5}$$

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

$$\begin{aligned} \pi_k(s) &= \operatorname*{argmax}_a \left[r(s, a) + \gamma V_k(\delta(s, a))\right] \\ V_{k+1}(s) &= r(s, \pi_k(s)) + \gamma V_k(\delta(s, \pi_k(s))) \end{aligned} \tag{6}$$

It can be shown that policy iteration converges to the optimal policy.

We will now use policy iteration, i.e. equation (6) to make 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.

Assignment 1:

- Write down the transition matrix, $\delta(s, a)$, for the system. The transition matrix is a 16×4 matrix where each row corresponds to a state (according to figure 1), and each column corresponds to an action. Each element contains the state (i.e. a number between 1 and 16) one reaches after doing the action. Fill in the table by hand by matching the actions listed in table 1 with pairs of states (figure 1).

- Write down a suitable reward matrix, $r(s, a)$. This is also a 16×4 matrix with a similar organization, but the elements here contain the reward received when doing a particular action in a certain state.

  From the task formulation, the actual contents of the reward matrix is not well defined. You will have to make reasonable guesses of what actions to reward and what to penalize in order to get a reasonable walking behavior.

  Suggestion: Start with zeros throughout and then enter positive values for actions that correspond to moving forward, typically moving one of the legs from down&forward to down&back while the other leg is lifted. Also enter negative values for actions that should be avoided, such as lifting one leg when the other is already lifted.

- Use policy iteration based on equation (6) to compute the optimal value function $V(s)$ and policy $\pi(s)$ based on the transition and reward matrices you have just constructed. Assign the two matrices to variables in Matlab and write the necessary code based on equation (6). The result will be a policy vector which states what action to take in every state.

- 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. Save 20 successive states in a vector and use the function `walkshow(statevec)` from the course directory to show the behavior graphically. This function will display a sequence of cartoon images.

The image will also be saved in a file called `cartoon.png` in your working directory.

- Check that the behavior is reasonable, i.e. that it looks like a good walking pattern. You may have to improve you reward function in order to avoid strange behaviors.
- Does it make any difference what initial values are used for the value function? What happens if you initialize the value function with random values? Does the algorithm converge to a different policy?

3.3 The Q-function

One problem with policy iteration as formulated in equation (6) is that we need to make explicit use of the reward and transition functions, both when estimating the values and when improving the policy. We ultimately aim for an algorithm which can operate when the reward and transition functions are not explicitly known, and a step in this direction is to reformulate equation (6) using estimated values of state×action pairs. 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).

V and Q can be expressed in terms of each other and, thus, basically carries the same information.

$$V(s) = Q(s, \pi(s)) \tag{7}$$

$$Q(s, a) = r(s, a) + \gamma V(\delta(s, a)) \tag{8}$$

The main advantage of using Q instead of V, is that it becomes simpler to update the policy. The policy iterations now become:

$$\begin{aligned} \pi_k(s) &= \operatorname*{argmax}_a Q_k(s, a) \\ Q^{\pi}_{k+1}(s, a) &= r(s, a) + \gamma \max_{a'} Q^{\pi}_k(\delta(s, a), a') \end{aligned} \tag{9}$$

Note that the Q function needs to be stored in a matrix, which will be much larger than the vector used for V. When the number of possible actions is large, this may in fact make it practically impossible to use the Q function.

Assignment 2:

- Modify your program to use a Q-function instead of V. Does it produce the same policy?

4 Temporal Difference Learning

This part of the lab deals with the general reinforcement learning problem, that is, we will no longer assume that the state transition and reward functions are known. 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 V or Q function as defined above. We will here use the Q-function since this makes it simple to extract the policy.

One possibility is to use the *Monte Carlo*-method to directly measure the Q-values. By starting in an arbitrary state and performing an arbitrary action and then continuing to the end (or for sufficiently long), accumulating received rewards as we go, we will get a direct measure of the Q-value. Remember that the Q-value is the expected total future reward. The disadvantage with the Monte Carlo-method is that it is prohibitively slow. In practice, this method is never used.

Temporal difference (TD) is a class of learning methods which are able to learn much quicker than the Monte Carlo method. TD improves the estimated Q-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*. For more details on temporal difference learning read chapters six and seven of the reinforcement learning book Sutton and Barto (1999).

4.1 Q-learning

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 then 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 it 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 $Q(s, a)$:

$$\begin{aligned} Q^{(1)}(s,a) &= Q(s,a) \\ Q^{(2)}(s,a) &= r + \gamma \max_{a'} Q(s',a') \end{aligned} \tag{10}$$

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. It is

normally wise to limit the steps taken and therefore only a fraction η of the difference is allowed to change the estimate. We get the following update formula:

$$Q(s,a) \Leftarrow Q(s,a) + \eta \left[r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right] \tag{11}$$

In this way, 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.*

Fortunately, 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 used. To take an extreme example, even if the agent constantly produces 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 (11), independently of the policy we actually follow.

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

$$\pi(s) = \operatorname*{argmax}_a Q(s,a) \tag{12}$$

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. This can be achieved by decreasing ϵ with an increasing number of observations, for example according to

$$\epsilon(t) = \epsilon_0(1 - t/T) \tag{13}$$

where T is the total number of iterations. Reasonable values for learning and exploration rate are $\eta = 0.1$ and $\epsilon_0 = 0.2$.

The Q-learning algorithm is summarized in table 2.

Table 2: Summary of the Q-learning algorithm.

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

Assignment 3:

- Write a separate Matlab function

  ```
  [newstate reward] = go(state, action)
  ```

  which uses the transition and reward matrices to return the result of making a single action. This function encapsulates the knowledge about the environment and the learning algorithm should no longer be allowed to directly use the two matrices.

- Implement the Q-learning algorithm. Starting in 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.

- What happens if a pure greedy policy is used? Implement and compare with the ϵ-greedy policy. Does it matter what value of ϵ you use?

- Approximately how many steps are necessary for the Q-learning algorithm to find an optimal policy?

References

L. P. Kaelbling, M. L. Littman, and A. W. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

T. M. Mitchell. *Machine Learning*. McGraw Hill, 1997.

R. Sutton and A. Barto. *Reinforcement Learning*. MIT Press, 1999. Also available online at
`http://www-anw.cs.umass.edu/~rich/book/the-book.html`