2D1431 Machine Learning
Lab 4: Reinforcement Learning

Frank Hoffmann
e-mail: hoffmann@nada.kth.se

December 2, 2002

1 Introduction

In this lab you will learn about dynamic programming and reinforcement learning. It is assumed that you are familiar with the basic concepts of reinforcement learning and that you have read chapters 13 in the course book *Machine Learning* [2] as well as first four chapters of the survey on reinforcement by KAELBLING [1]. For further reading and a detailed discussion of policy iteration and reinforcement learning the textbook "Reinforcement Learning" is highly recommendable [3]. In particular studying chapters 3,4 and 6 is of immense help for this lab. The predefined Matlab functions for this lab are located in the course directory `/info/mi02/labs/lab4`.
Dynamic programming refers to a class of algorithms that can be used to compute optimal policies given a complete model of the environment. Dynamic programming solves problems that can be formulated as Markov decision processes. Unlike in the reinforcement learning case, dynamic programming assumes that the state transition and reward functions are known. The central idea of dynamic programming and reinforcement learning is to learn value functions, which in turn can be used to identify the optimal policy.

2 Policy Evaluation and Policy Iteration

First we consider policy evaluation, namely how to compute the state-value function V^π for an arbitrary policy π. For the deterministic case the value function has to obey the Bellman equation.

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

where $\delta(s,a) : SxA \rightarrow S$ and $r(s,a) : SxA \rightarrow \Re$ are the deterministic state transition and reward function. This equation can be either 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, and considering the Bellmann equation as an update rule

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

The sequence of V^{π}_{k} can be shown to converge to V^{π} as $k \rightarrow \infty$. This method is called *iterative policy evaluation*. For the non-deterministic case, the transition and reward functions have to be replaced by probabilistic functions. In that case the Bellman equations become:

$$V^{\pi}(s) = \sum_{s'} P(s'|s,a) \sum_{a} \pi(s,a)(R(s',s,a) + \gamma V^{\pi}(s')) \tag{4}$$

where $P(s'|s,a)$ is the probability that the next state is s' when executing action a in state s and $R(s',s,a)$ is the reward when executing action a in state s and transitioning to the next state s'. Policy evaluation for the non-deterministic case, can be formulated as an update rule similar to equation 3 by

$$V^{\pi}_{k+1} = \sum_{s'} P(s'|s,a) \sum_{a} \pi(s,a)(R(s',s,a) + \gamma V^{\pi}_{k}(s')) \tag{5}$$

Our main motivation for computing the value function for a policy is to improve on our current 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 $\pi(s,a)$ if it has a higher action value function $Q(s,a) > Q(s,\pi(s))$. This process is called *policy improvement*. In other words, for each state s we greedily choose the action that maximizes $Q^{\pi}(s,a)$

$$\pi'(s) = argmax_a Q^{\pi}(s,a) = argmax_a r(s,a) + \gamma V(\delta(s,a) \tag{6}$$

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

$$\begin{aligned} V^{\pi}_{k+1}(s) &= \max_{a} Q(s,a) = \max_{a} (r(s,a) + \gamma V^{\pi}_{k}(\delta(s,a))) \\ \pi_{k+1}(s,a) &= argmax_a Q(s,a) = argmax_a \, (r(s,a) + \gamma V^{\pi}_{k}(\delta(s,a))) \end{aligned} \tag{7}$$

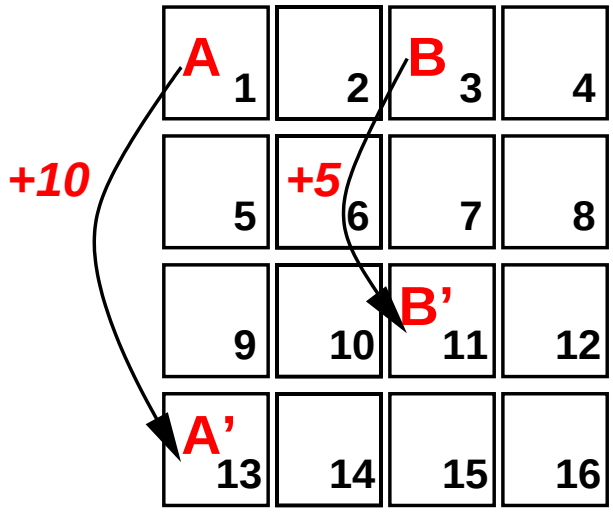

Figure 1: Grid world. Independent of the action taken by the agent in cell A, it is beamed to cell A' and receives a reward of +10. The same applies to B and B' with a reward of +5.

For the non-deterministic case we obtain

$$\begin{aligned} V^{\pi}_{k+1}(s) &= \max_a Q(s,a) \\ &= \max_a \sum_{s'} P(s'|s,a)(R(s',s,a) + \gamma V^{\pi}(s')) \\ \pi_{k+1}(s,a) &= argmax_a Q(s,a) \\ &= argmax_a \sum_{s'} P(s'|s,a)(R(s',s,a) + \gamma V^{\pi}_k(s')) \end{aligned} \tag{8}$$

It can be shown that policy iteration converges to the optimal policy. Notice, that each policy evaluation, itself an iterative computation, is started with the value function for the previous policy.
Assume a grid world of 4x4 cells that correspond to 16 states enumerated $s_1, \ldots, s_{16}$ as shown in Figure 1. In each state the agent can choose one of the four possible actions (North, West, South, East) in order to move to a neighboring cell. If the agent attempts to move beyond the limits of the grid world, for example going east in state s_8 located at the right edge, it remains in the original cell but incurs a penalty of -1. There are two special cells A (s_1) and B (s_3) from which the agent is *beamed* to the cells A' (s_{13}) respectively B' (s_{11}) independent of the action it chooses. When being beamed it receives a reward of +10 for the transition from A to A'

and a reward of +5 for the transportation from B to B'. For all other moves that do not attempt to lead outside the grid world the reward is zero. There are no terminal states and the agent tries to maximize its future discounted rewards over an infinite horizon. Assume a discount factor of $\gamma = 0.9$. Due to the discount factor the accumulated reward remains finite even if the problem has an infinite horizon. Notice, that returning from B' to B, only takes a minimum of two steps, whereas going back to A from A' takes at least three steps. Therefore, it is not immediately obvious which policy is optimal.

Assignment 1:

- Use value iteration to compute the value function $V^\pi(s)$ for an equiprobable policy in which at each state all four possible actions (including the ones that attempt to cross the boundary of the grid world) have the same uniform probability $\pi(s, a) = 1/4$. Assume a discount factor $\gamma = 0.9$. Use policy iteration according to the Bellman equations in 3 to approximate the value function. You can either use two arrays, one for the old values $V_k^\pi(s)$ and one for the new values $V_{k+1}^\pi(s)$. This way the new values can be computed one by one from the old values without the old values being changed. It turns out however, that it is easier to use asynchronous updates, with each new value immediately overwriting the old one. Asynchronous updates also converges to V^π, in fact it usually converges faster than the synchronous update two-array version. As an example we compute the new value of state s_8. For the four possible actions North, West, South, East the successor states are $\delta(s_8, North) = s_4$, $\delta(s_8, South) = s_{12}$, $\delta(s_8, West) = s_7$ and $\delta(s_8, East) = s_8$ (the agent attempts to leave the grid world and remains in the same square). The rewards are all zero except for the penalty $r(s_8, East) = -1$ when taking the East action. All actions are equally likely, therefore $\pi(s_8, North) = \pi(s_8, South) = \pi(s_8, West) = \pi(s_8, East) = 1/4$. In Matlab we use a vector of length 16 to store the value function. The update rule for state s_8 would look like:

```
>> gamma=0.9;
>> V=zeros(16,1);
>> V(8) = 1/4 * (-1 + gamma* (V(4) + V(7) + V(12) + V(8)))
```

The Matlab function `plot_v(V,range,pi)` plots the state value function as a color plot. The first argument `V` is a $16x1$-vector with the

state values $V(s_i)$. The second optional argument `range` is a $2x1$-vector to specify the lower and upper bound of the value function for scaling the color-plot. The default range is $[-10\ 30]$. The third optional argument `pi` is a $16x1$-vector for specifying the current policy $\pi(s) : S \rightarrow A$, where by definition, the actions North, East, South, West are clockwise enumerated from 1 to 4.

- Use policy iteration based on equation 7 to compute the optimal value function V^* and policy $\pi^*(s, a)$. It might be easier to use the action value function $Q(s, a)$ rather than the state value function$V(s)$. In Matlab you represent $Q(s, a)$ by a $16x4$-matrix, where the first dimension corresponds to the state, and the second dimension to the action. Visualize the optimal value function and policy using `plot_v`. After how many iterations does the algorithm find an optimal policy, assuming the initial state values are zero? Is the optimal policy unique? What happens if you initialize the state value function with random values rather than zero

  ```
  >> V=10.0*rand(16,1);
  ```

  Does the algorithm converge to a different policy?

Assignment 2:
Assume, that the transition function is no longer deterministic, but given by the probability $P(s'|s, a)$. Compute the optimal value function V^* and policy $\pi^*(s, a)$ using policy iteration according to equations 8, for a non-deterministic state transition function. Assume that with probability $p = 0.7$, the agent moves to the "correct" square as indicated by the desired action, but with probability $1 - p = 0.3$ a random action is taken that pushes the agent to a random neighboring square. The random square can be coincidentally the very same cell that was originally preferred by the action. A random action can also be an *illegal* move, that incurs a penalty of -1. Visualize the optimal value function and policy using `plot_v`. After how many iterations does the algorithm find an optimal policy, assuming the initial state values are zero? Is the optimal policy unique? Does the optimal policy change when you reduce the deterministic component of actions to $p = 0.6$?

3 Reinforcement Learning

This assignment deals with the general reinforcement learning problem, in that we no longer assume that the state transition and reward functions are known. *Temporal difference (TD)* learning directly learn from experience and do not rely on a model of the environment's dynamics. TD methods update the estimate of the action value function based on learned estimates, in other words unlike Monte Carlo methods which update their estimates only at the end of an episode, they bootstrap and update their beliefs immediately after each state transition. For more details on temporal difference learning read chapters six and seven of the reinforcement learning book [3]. Temporal difference learning is easier formulated using the action value function $Q(s, a)$ rather than the state value function $V(s)$ which are related through

$$Q^{\pi}(s, a) = \sum_{s}^{'} P(s'|s, a)R(s', s, a) + \gamma V^{\pi}(s') \tag{9}$$

In contrast to dynamic programming, the agent learns through interaction with the environment. There is a need for active exploration of the state space and the possible actions. At each state s the agent chooses an action a according to its current policy, and observes an immediate reward r and a new state s'. This sequence of state, action, reward, state, action motivates the name SARSA for this form of learning.
The action value function can be learned by means of off-policy TD learning also called *Q-learning*. In its simplest form, one step Q-learning, it is defined by the update rule

$$Q(s, a) = Q(s, a) + \alpha(r + \gamma \max_{a'} Q(s', a') - Q(s, a)) \tag{10}$$

In this case, the learned action-value function $Q(s, a)$ directly approximates the optimal value function $Q^*(s, a)$, independent of the policy followed, hence off-policy learning. However, the policy $\pi(s, a) : SxA \rightarrow \Re$ ($\pi(s, a)$ is the probability of taking action a in state s) still has an effect in that it determines which state-action pairs are visited and updated.
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 $Q(s, a)$. The policy is generally *soft*, meaning that $\pi(s, a) > 0$ for all states and actions. An ϵ-greedy policy satisfies this requirement, in that most of the time with probability $1 - \epsilon$ it picks the optimal action according to

$$\pi(s) = argmax_a Q(s, a) \tag{11}$$

but with small probability ϵ it takes a random action. Therefore, all non-greedy actions a are taken with the probability $\pi(s,a) = \epsilon/A(s)$, where $A(s)$ is the number of alternative actions in state s. As the agent collects more and more evidence the policy shifts towards a deterministic optimal policy. This can be achieved by decreasing ϵ with an increasing number of observations, for example according to

$$\epsilon(t) = \epsilon_0(1 - t/T) \tag{12}$$

where T is the total number of iterations. Reasonable values for learning and exploration rate are $\alpha = 0.1$ and $\epsilon_0 = 0.2$.
The off-policy TD algorithm can be summarized as

- Initialize $Q(s,a)$ arbitrarily
- Initialize s
- 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 $Q(s,a) = Q(s,a) + \alpha(r + \gamma \max_{a'} Q(s',a') - Q(s,a))$
  - Replace s with s'
- until T steps

Assignment 3:
For an unknown environment the agent is supposed to learn the optimal policy by means of off-policy temporal difference learning. The state space consists of 25 states $s_1, \ldots, s_{25}$, corresponding to a $5x5$ grid-world. In each state the agent has the choice between four possible actions $a_1, \ldots, a_4$, which can be associated to the four directions North, East, South, West. However, the transition function is not deterministic, which means the agent sometimes ends up in a non-neighboring square. Assume, that the exact model of the environment and the rewards are unknown. The dynamics of the environment are determined by the Matlab functions `s = startstate` and `[s_new reward] = env(s_old,action)`. The function `startstate` returns the initial state. The states $s_1, \ldots, s_{25}$ are represented by the integers $1, \ldots, 25$, and the actions $a_1, \ldots, a_4$ are enumerated by $1, \ldots, 4$. The function `[s_new reward] = env(s_old,action)` computes the next state `s_new` and the reward `reward` when executing action `action` in the current state `s_old`. Represent the action value function $Q(s,a)$ by a $25x4$-matrix `Q`.

Given `Q` you can compute the optimal policy `pi(s)` and state value function `V` and visualize it with `plot_v_td(V,range,pi)` using the following code

```
>> [V pi] = max(Q,[],2);
>> plot_v_td(V,[-5 15],pi);
```

The function `plot_v_td(V,range,pi)` is the counterpart to the Matlab function `plot_v(V,range,pi)` for the $4x4$-gridworld used in the earlier assignments. The function `plot_trace(states,actions,tlength)` can be used to plot a trace of the most recently visited states. The parameter `states` is a $Nx1$-vector that contains the history of recent states $s(t), \ldots, s(t+N)$, the parameter `actions` is a $Nx1$-vector that stores the history of recent actions $a(t-1), \ldots, a(t+N-1)$, and `tlength` determines how many states from the past are plotted. Build a history of states, actions and rewards when iterating the TD-learning algorithm, by appending the new state `s`, action `a` and reward `r` to the history of previous `states`, `actions` and `rewards`.

```
>> for k=1:iterations
>> ...
>> states = [states s];
>> actions = [actions a];
>> rewards = [rewards r];
>> ...
>> end
>> plot_trace(states,actions,12);
```

Run the off-policy TD learning algorithm for 20000 steps. Initialize the $Q(s,a)$ with small positive values (e.g. 0.1) in order to bias the TD-learning to explore alternative actions in the early stages, when most of the time the rewards are zero.
Every 500 steps

- visualize the current state value function $V(s)$, optimal policy $\pi(s)$
- plot a trace of the recently visited states and actions. and
- compute the average reward over the past 500 steps and plot the evolution of the average and accumulated reward as a function of the number of iterations.

Experiment with different settings for the exploration parameter ϵ_0 and learning rate α. Can you think of an extension to the one-step TD-learning algorithm that would help to learn the optimal policy in a fewer number of iterations? If you have time, try to implement this extension.

References

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

[2] Tom M. Mitchell. *Machine Learning*. McGraw Hill, 1997.

[3] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning*. MIT Press, 1999. also available online at http://www-anw.cs.umass.edu/ rich/book/the-book.html.