Assignment 4: Markov Decision Processes and Reinforcement Learning

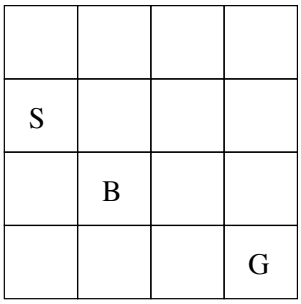
CS486/686 – Spring 2017

Out: July 5, 2017
Due: July 21 (11:59 pm), 2017.

Submit an electronic copy of your assignment via LEARN. Late submissions incur a 2% penalty for every rounded up hour past the deadline. For example, an assignment submitted 5 hours and 15 min late will receive a penalty of ceiling(5.25) * 2% = 12%.

Be sure to include your name and student number with your assignment.

In this assignment you will implement in Python the value iteration and Q-learning algorithms for the following simple grid-world problem in Questions 1 and 2.



S = start state

B = bad state

G = goal state

An agent starting in the start state S must reach the goal state G. At each time step, the agent can go up, $down$, $left$ or $right$. However, the agent's movements are a bit noisy since it goes in the intended direction with a high probability a and in one of the two lateral directions with a low probability b. For instance, when executing the action up, the agent will indeed go up by one square with probability a, but may go left with probability b and right with probability b (here $a + b + b = 1$). Similarly, when executing the action $left$, the agent will indeed go left with probability a, but may go up with probability b and down with probability b, When an action takes the agent out of the grid world, the agent simply bounces off the wall and stays in its current location. For example, when the agent executes $left$ in the start state it stays in the start state with probability a, it goes up with probability b and down with probability b. Similarly, when the agent executes up from the start state, it goes up with probability a, right with probability b and stays in the start state with probability b. Finally, when the agent is in the goal state, the task is over and the agent transitions to a special *end* state with probability 1 (for any action). This end state is *absorbing*, meaning that the agent cannot get out of the end state (i.e., it stays in the end state with probability 1 for every action).

The agent receives a reward of 100 when it reaches the goal state, -70 for the bad state and -1 for every other state, except the end state, which has a 0 reward. The agent's task is to find a policy to reach the goal state as quickly as possible, while avoiding the bad state.

In case you are not certain about the transition and reward model, download the file gridWorld.py from the course website. It contains a precise description (in Python) of the transition and reward models. Feel free to directly use the code in this file as part of your code in Questions 1 and 2.

1. **[35 pts]** Value iteration

   Compute the optimal policy by implementing the value iteration algorithm in Python. Use a discount factor of 0.99 and run value iteration until the difference between two successive value functions is at most 0.01 (i.e., $|V_{t+1}(s) - V_t(s)| < 0.01 \ \forall s$). Run value iteration once with $a = 0.9$, $b = 0.05$ and a second time with $a = 0.8$, $b = 0.1$

   What to hand in:

   - Your Python code.
   - The optimal policies and optimal value functions found for $a = 0.9$, $b = 0.05$ and for $a = 0.8$, $b = 0.1$.
   - Discuss the differences found in the optimal policies and value functions for the different combinations of a and b. Explain briefly how a and b impact the optimal policy.

2. **[35 pts]** Q-learning

   Assuming the transition and reward models are unknown, compute the optimal policy by Q-learning (a.k.a active temporal difference) in Python. In contrast to Question 1, the transition and reward models should only be used to *simulate* the environment when the agent executes an action. Use a transition model with $a = 0.9$ and $b = 0.05$, a discount factor of 0.99 and a learning rate of $\alpha = 1/N(s, a)$ where $N(s, a)$ is the number of times that action a was executed in state s. Always starting from the start state, run Q-learning for 10,000 episodes, where an episode consists of a sequence of moves from the start state until the end state is reached. Try two different ϵ-greedy exploration functions by setting ϵ to 0.05 and then to 0.2. In other words, when $\epsilon = 0.05$, select the optimal action with probability 0.95 and a random action with probability 0.05. Similarly, when $\epsilon = 0.2$, select the optimal action with probability 0.8 and a random action with probability 0.2.

   What to hand in:

   - Your Python code.
   - The optimal policies and optimal value functions found for $\epsilon = 0.05$ and $\epsilon = 0.2$.
   - Discuss the impact of ϵ on the convergence of Q-learning. More specifically, discuss the impact on the **rate** of the convergence and the **policy** that it will eventually converge to.

3. **[30 pts]** Deep Q Network

   Train an agent to solve the CartPole problem in OpenAI Gym by implementing a Deep Q Network. Since CartPole has a continuous state space, we cannot use a table representation for the Q-function. Instead, you will use a neural network to represent the Q-function. Follow these steps:

   - Go to `gym.openai.com` and install the *Gym* package by following the instructions in the documentation.
   - Follow the instructions in the documentation to run a simple agent that executes actions at random in the CartPole environment.
   - Replace the default random agent by a Deep Q Network (DQN) agent. More precisely, implement Q-learning where the Q-function is represented by a neural network that takes as input a state and outputs a Q-value for each possible action. You have three choices for the neural network:
     (a) Use TensorFlow to implement a neural network. Go to `tensorflow.org` and install TensorFlow. Click on *getting started* and follow the first two tutorials: *Getting Started with TensorFlow* and *MNIST for ML Beginners*. This should be enough for you to implement a basic neural network.
     (b) Use the class `sklearn.neural_network.MLPRegressor` in scikit-learn to implement a neural network.
     (c) Implement your own neural network from scratch based on the slides from Lecture 17.

Among the choices above, TensorFlow is the preferred environment to design scalable neural networks that run on GPUs (although you do not need a GPU for this assignment) as well as neural networks of any shape. Sklearn provides a very basic neural network class that is sufficient for this assignment. Finally, if you are really eager, you can also implement your own neural network from scratch, but this will be time consuming. You will earn the same grade for any of the above choices.

What to hand in:

- Your Python code.
- Further instructions will be posted on the course website