Reinforcement Learning
Winter 2017
Assignment 1 - Policy Planning

Note: this assignment is an extended version of the assignment proposed here:

http://ai.berkeley.edu/reinforcement.html#Welcome

where we propose additional tasks, but also a significantly larger amount of coding hints.

Please have an environment-friendly policy, avoid printing these sheets !

Please report typos or errors to Sebastien Gros (grosse@chalmers.se), we will correct them asap and update the document on Pingpong accordingly !

Introduction to the environment

If you are not a regular Python user, you may want to have a look at the Python tutorial on Pingpong!

In these assignments, you will implement value-iteration, Q-learning, TD-learning and policy learning. You will test your agents first on Gridworld, then apply them to a simulated robot controller (Crawler) and finally to a Pacman game.

You will be working inside a Python code, modifying some very specific entries in that code. Once you think you have a working solution for a question, you can test it by using:

```
python autograder.py
```

It can be run for one particular question, such as q2, by:

```
python autograder.py -q q2
```

It can be run for one particular test by commands of the form:

```
python autograder.py -t test_cases/q2/1-bridge-grid
```

The code for this project contains the following files, which are available in a zip archive available on pingpong.
Here are the files you'll have to edit in these assignments:

- valueIterationAgents.py: A value iteration agent for solving known MDPs.
- qlearningAgents.py: Q-learning agents for Gridworld, Crawler and Pacman.
- analysis.py: A file to put your answers to questions given in the project.

Here are files you may want to have a look at but NOT edit:

- mdp.py: Defines methods on general MDPs.
- learningAgents.py: Defines the base classes ValueEstimationAgent and QLearningAgent, which your agents will extend.
- util.py: Utilities, including util.Counter, which is particularly useful for Q-learners.
- gridworld.py: The Gridworld implementation.

- featureExtractors.py: Classes for extracting features on (state,action) pairs. Used for the approximate Q-learning agent (in qlearningAgents.py).

We will start this assignments with a simple **stochastic game**. To get started, run Gridworld in **manual** control mode, which uses the arrow keys, by prompting your terminal with the command (you need to be in the right folder!):

```
python gridworld.py -m
```

You will see a two-gates layout, with reward $+1$ (good) and -1 (not good). You can move the blue token in the game, but you have to move around the pillar (grey square). The game is stochastic insofar as the blue token executes the command you ask 80% of the time, and responds randomly otherwise.

The game terminates once the blue token is in one of the gates and once the command 'exit' is prompted (only action allowed once the blue token is in one of the exit state). You can observe in your terminal, as you play, the states, actions, and rewards in the game. At the end of the game, the complete return (overall score) is computed and displayed in the terminal.

You can control many aspects of the simulation. A full list of options is available by running:

```
python gridworld.py -h
```

The Gridworld MDP is such that you first must enter a pre-terminal state (either one of the boxes labelled $+1$ or -1 in the GUI) and then take the special 'exit' action before the episode actually ends (the game then takes the true terminal state called TERMINAL_STATE, which is not shown in the GUI). If you run an episode manually, your total return may be less than you expected, due to the discount rate (-d to change; 0.9 by default).

Look at the console output that accompanies the graphical output (or use -t for all text). You will be told about each transition the agent experiences (to turn this off, use -q).

1. **MDP** In this first assignment, we will compute the value function in the game and the associated optimal policy via a value-iteration approach. Note that we are not doing reinforcement learning per se yet, i.e. we do have a model of the MDP in the form of the probability of transitions between states.

   We will work within the **valueIterationAgents.py** file, and edit the *class* **ValueIterationAgent** within that file (first one in the file).

   You will have to perform the following tasks:

   a Write down a mathematical expression relating the Q-value function $Q(s,a)$ to the value function $V(s)$ (this should appear in the report).

   b Implement this in the function **computeQValueFromValues** such that it returns the Q-value for a pair of state-action (state, action) pair, based on value function (can be stored as the dictionary, with e.g. name self.values, or as a counter, see next bullet point!). This function returns its value as a 'classic' return, i.e. by terminating the function with

   return *the value we need to return*

   c Edit the method **computeActionFromValues** such that it computes the best action for a given state, based on the Q-value function. Make sure you call your method **computeQValueFromValues** to do that. Remember that the **util** class provides you with a dictionary-style structure called 'counter' that gives you sorting goodies. E.g. if you store the values of the different actions available for a given state in a counter, e.g. ExpectedValueofAction = util.Counter() by attributing:

   ExpectedValueofAction[Action] = self.computeQValueFromValues(state, Action)

   over all admissible actions, you can e.g.

   - sort your counter by using ExpectedValueofAction.sortedKeys()
   - get the best action by doing ExpectedValueofAction.argMax()

   Your method **computeActionFromValues** must return the action (e.g. return *my best action*).

   Important: if there is no legal action for the provided state, your function must return **None** (existing Python type, this is not a string)

   d Write down mathematically how to perform the value iteration (this should be reported)

   e Implement this in the class constructor **__init__** (see def __init__) in your **valueIterationAgents.py** code. Here are some useful remarks:

   - The function receives the MDP in its arguments. 'mdp' is a class itself, equipped with a number of methods that can be prompted via mdp.*method*. E.g.

mdp.getPossibleActions(s)	returns a list of the available actions for state s
mdp.getTransitionStatesAndProbs(s,a)	returns a list of tuples *(state, probability)*
mdp.getReward(s,a,s_+)	returns the reward for the transition $s,a \rightarrow s_+$

   - Throughout our various games, the states and actions come in the form of immutables (tuples or strings), which can be directly used as keys in dictionaries and counters.
   - The value function you are building should be a dictionary (keys are states) attributed to the self, so that it is visible in all methods of the class. Hence you need to fill in your value function as e.g.

     self.values[state] $=$ *the value of that state*

     such that self.values[state] return the scalar value of a particular state.
   - **Important**: we want a "batch" version of value iteration where each vector V_k is computed from a fixed vector V_{k-1}. This means that when a state's value is updated in iteration k based on the values of its suc-

cessor states, the successor state values used in the value update computation should be those from iteration $k-1$ (even if some of the successor states had already been updated in iteration k). The difference is discussed in Sutton & Barto in the 6th paragraph of chapter 4.1 (see https://webdocs.cs.ualberta.ca/ sutton/book/ebook/node41.html). In practice, that means that at each iteration you need to make a copy of your value function, make updates in the copy, and then replace the 'old' value function by the (updated) copy. **Beware of shallow copies in Python!!**

- Use your method **computeQValueFromValues** for a simpler code

f Briefly explain how value iteration works and motivate why the value function changes the way it does with the first, say, 10 iterations. The idea is again to keep it short and just focus on some main properties (this should be in the report)

Your value iteration agent is an offline planner, not a reinforcement learning agent, and so the relevant training option is the number of iterations of value iteration it should run (option -i) in its initial planning phase. ValueIterationAgent takes an MDP on construction and runs value iteration for the specified number of iterations before the constructor returns.

The command

```
python gridworld.py -a value -i 100 -k 10
```

loads your ValueIterationAgent, which will compute a policy using 100 value iterations and then run the game 10 times with it. Press a key to cycle through values, Q-values, and the simulation. You should find that the value of the start state (V(start), which you can read off of the GUI) and the empirical resulting average reward (printed after the 10 rounds of execution finish) are quite close.

Running value iteration for 5 iterations, i.e. running

```
python gridworld.py -a value -i 5
```

should give you this output:

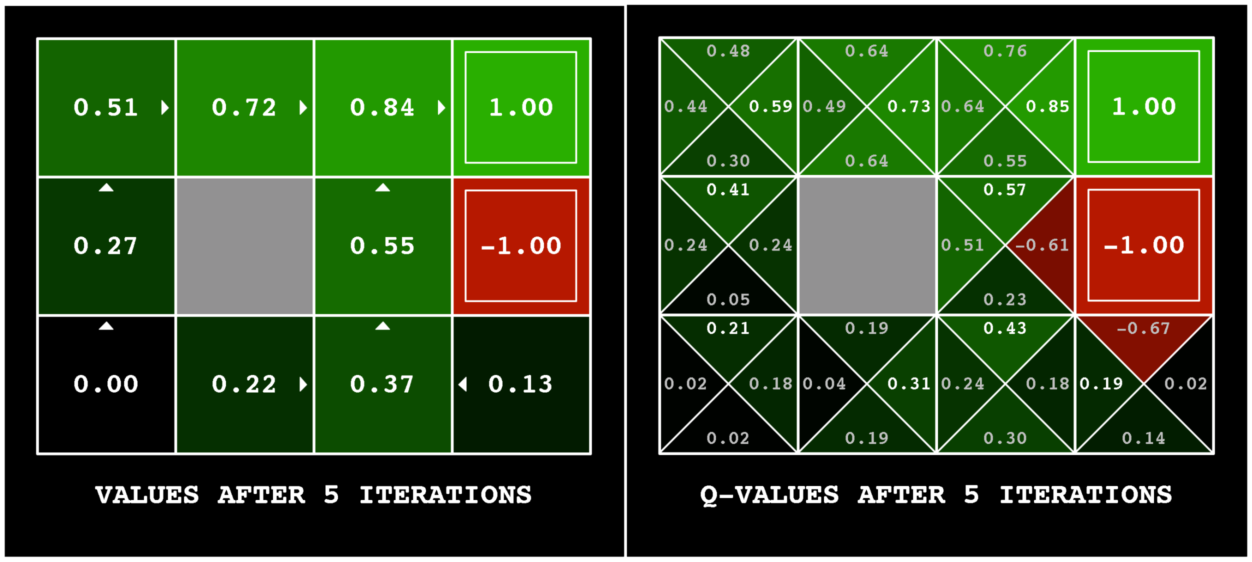

Figure 1: Value function (left) and Q-value function (right) (the GUI toggles from the first to the second figure if you press a key, another key will make it play the game using the optimal policy).

To evaluate your implementation, run the autograder:

```
python autograder.py -q q1
```

2. **Bridge Crossing Analysis**

   Let's play with our agent !! **BridgeGrid** is a grid world map with a low-reward terminal state and a high-reward terminal state separated by a narrow "bridge", on either side of which is a chasm of high negative reward. The agent starts near the low-reward state. With the default discount of 0.9 and the default noise of 0.2, the optimal policy does not cross the bridge, i.e. if you run

```
python gridworld.py -a value -i 100 -g BridgeGrid
```

   you will get the output displayed in Figures 2 and 3

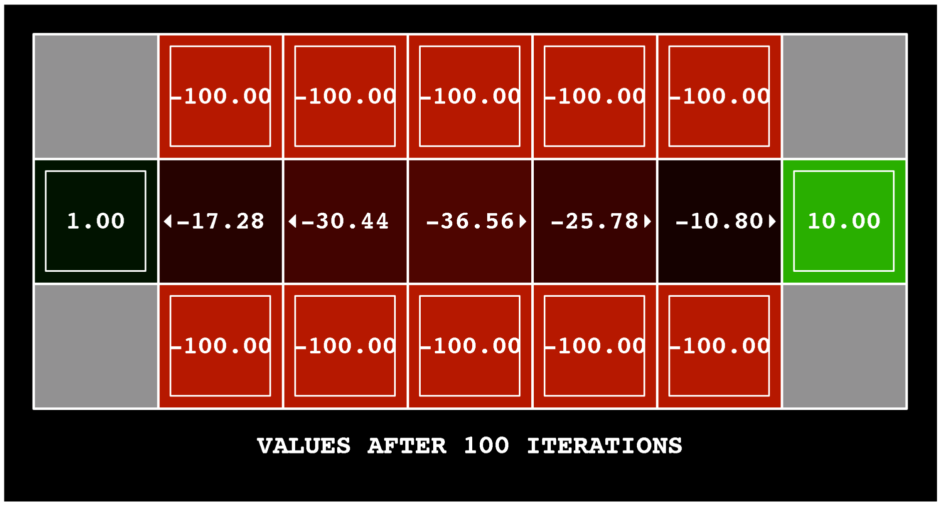

Figure 2: Value function of the bridge game. There is a strong incentive to not cross the bridge, and get the low reward on the left instead of the high reward on the right.

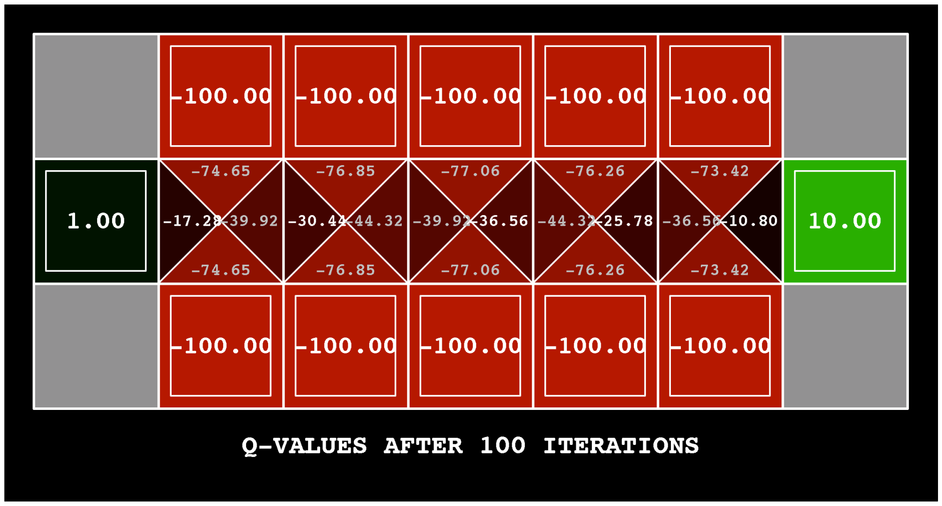

Figure 3: Q-value function of the bridge game. It has a strong incentive to not cross the bridge, and get the low reward on the left instead of the high reward on the right.

   You can play with your bridge with different levels of discount (option -d) and noise (option -n). E.g. the command:

```
python gridworld.py -d 1. -n 0.0 -a value -i 100 -g BridgeGrid
```

will use a discount of 1 and no noise.

Change only ONE of the discount and noise parameters so that the optimal policy causes the agent to attempt to cross the bridge. Put your answer in **question2()** of **analysis.py**. Noise refers to how often an agent ends up in an unintended successor state when they perform an action. you can assess your answer by running

```
python autograder.py -q q2
```

3. **Policies** Consider the DiscountGrid layout, shown in Fig. 4. This grid has two terminal states with a positive payoff (in the middle row), a close exit with payoff $+1$ and a distant exit with payoff $+10$. The bottom row of the grid consists in a "cliff" of terminal states with large negative payoffs of -10 (shown in red). The starting state is the yellow square. The player can then select two paths:

   1. the red path that "risks the cliff" (remember that the token does not necessarily obey the demanded actions!) and travel near the bottom row of the grid, it is shorter but risks the sanction of a large negative payoff if the token falls off the cliff.
   2. the green path is the safe path that "avoid the cliff" and travel along the top edge of the grid. It is longer but less likely to incur a large negative payoff.

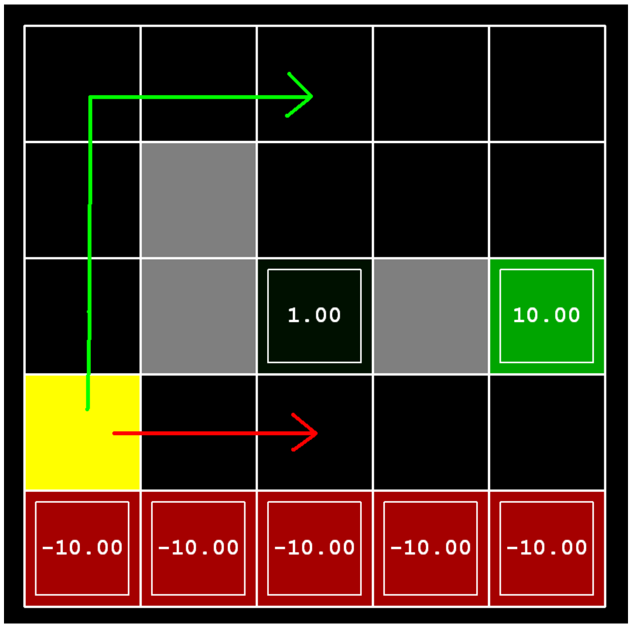

Figure 4:

In this question, you will choose settings of the discount, noise, and living reward parameters for this MDP to produce optimal policies of several different types. Your setting of the parameter values for each part should have the property that, if your agent followed its optimal policy without being subject to any noise, it would exhibit the demanded behavior. If a particular behavior is not achieved for any setting of the parameters, assert that the policy is impossible by returning the string 'NOT POSSIBLE'.

If you fancy you can test your grid by running:

```
python gridworld.py -a value -i 100 -g DiscountGrid
```

Here are the optimal policy types you should attempt to produce:

a Prefer the close exit (+1), risking the cliff (-10)
b Prefer the close exit (+1), but avoiding the cliff (-10)
c Prefer the distant exit (+10), risking the cliff (-10)
d Prefer the distant exit (+10), avoiding the cliff (-10)
e Avoid both exits and the cliff (so an episode should never terminate)

Edit question3a() through question3e() in **analysis.py**. To check your answers, run the autograder:

```
python autograder.py -q q3
```

Note: You can check your policies in the GUI. For example, using a correct answer to 3(a), the arrow in (0,1) should point east, the arrow in (1,1) should also point east, and the arrow in (2,1) should point north.

Note: On some machines you may not see an arrow. In this case, press a button on the keyboard to switch to qValue display, and mentally calculate the policy by taking the arg max of the available qValues for each state.

Questions

- Give an expression describing how you can compute the Q-value function from the value function (task a, question 1)
- Give an expression describing how you can compute the value iteration (task d, question 1)
- Report your answer for task f, question 1
- Briefly detail your reasoning behind your answers in question 3.