DIT 821: Software Engineering for AI Systems **(Due: Oct 07, 2019)**

Lab on Reinforcement Learning

Instructor: Piergiuseppe Mallozzi

Note: this assignment is an reduced version of the assignment proposed at the AI course of UC Berkeley: http://ai.berkeley.edu/reinforcement.html, simplified and with lots more hints!

Please eport typos or errors to **Piergiuseppe Mallozzi** (mallozzi@chalmers.se).

1 Prerequisites

For this lab we are going to use Python 2.7 and PyCharm as IDE. You are free to choose another IDE as well.

- Install the last version of Python 2.7 on your machine
  https://www.python.org/downloads/release/python-2716/
- Install PyCharm, community edition (or get a student account and download the professional one, but it's the same really)
  https://www.jetbrains.com/pycharm/download/

If you are not a regular Python user, you may want to have a look at the Python tutorial on Canvas (https://canvas.gu.se/files/1487243), or look for the huge amount of forums/books/resources online!

1.1 Import the project

- Download the **zip** from Canvas: https://canvas.gu.se/files/1489867
- Create a new Python project with PyCharm and select the interpreter to be a new virtual environment based on Python 2.7 as shown in Figure 1.
- Unzip the downloaded file from Canvas and import all the files in the newly created project (you can select them all and drag and drop them into your project folder).

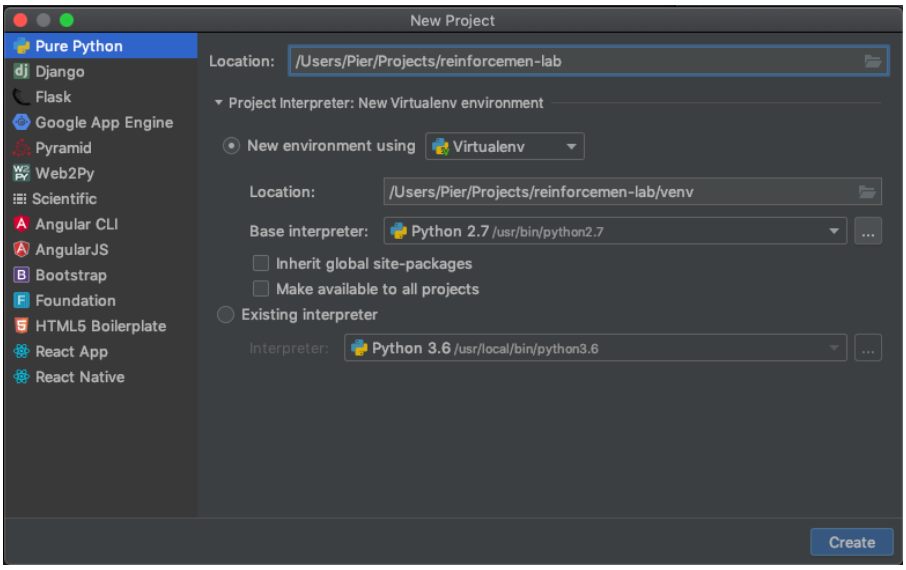

Figure 1: Create a new Python project with PyCharm

2 Introduction

In these assignments, you will implement Value-Iteration and Q-learning algorithms seen in class. 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 be test it by using:

```
python autograder.py
```

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

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

The code for this project contains the following files, which are available in a zip archive available on Canvas. Here are the files youll have to edit in the assignment:

- `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.

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).

As in our last game, Pacman, positions are represented by the (x,y) Cartesian coordinates (coordinate (0,0) is the lower-left corner), with north being the direction of increasing y, etc. By default, most transitions will receive a reward of zero, though you can change this with the living reward option (`-r`).

Deliverables: You have submit a zip with:

- Report containig only the answers to the questions below (pdf format).
- Python files that you have *modified* (not all of them!)

3 Value Iteration

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).

After completing all the problems, you can evaluate your implementation by running:

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

Problem 1 (1 points)

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).

Hint:
Given the transition probability matrix ($T(s, a, s')$) and the reward function ($R(s, a, s')$),
what is the value associated by taking action a from state s?

Problem 2 (2 points)

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. This function returns its value as a classic return, i.e. by terminating the function with `return value_we_need_to_return`.

Hints:
From the class **ValueIterationAgent** you can use access the following methods:

- `self.mdp.getTransitionStatesAndProbs(state, action)` returns list of (nextState, prob) pairs representing the states reachable from 'state' by taking 'action' along with their transition probabilities.
- `self.mdp.getReward(state, action, next_state)` returns the reward for the state, action, next_state transition.
- `self.getValue(state)` returns the value of state

Problem 3 (2 points)

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. The method must return the action (e.g. `return action_with_max_q_value`). If there is no *legal actions* for the provided state, your function must return **None** (existing Python type, this is not a string)

Hint:
From the class **ValueIterationAgent** you can use access the following method:

- `self.mdp.getPossibleActions(state)` returns the *list* of valid actions for *state*.

Problem 4 (1 points)

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

Problem 5 (3 points)

Implement the value iteration in the class constructor **__init__** (see `def __init__`) in your **valueIterationAgents.py** code.

- 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. Like the one that we have seen before:
  - `mdp.getPossibleActions(s)`
  - `mdp.getTransitionStatesAndProbs(s,a)`
  - `mdp.getReward(s,a,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*.

Hints:

- 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]` returns the scalar value of a particular state.
- **Important**: we want a batch version of value iteration where each element $V_k(s)$ (value of the state s at iteration k) of the vector V_k (that contains the value of all the states) is computed from a fixed vector V_{k-1} . This means that when a states value is updated in iteration **k** based on the values of its successor 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**). 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 the methods **computeQValueFromValues** and **computeActionFromValues** that you have written before.

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:

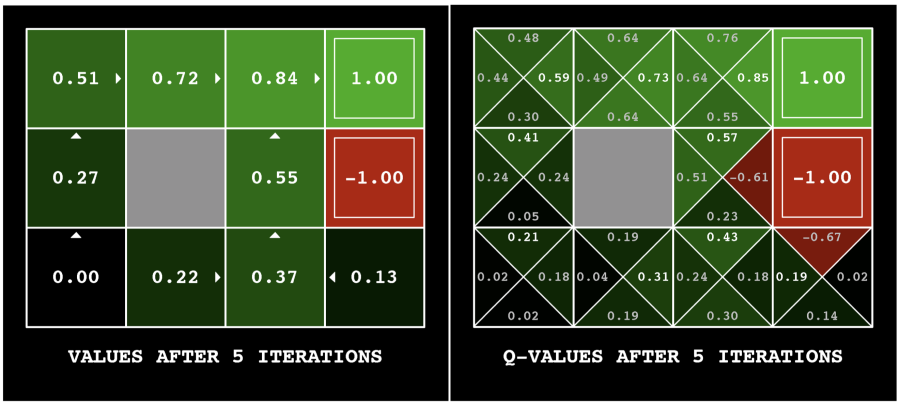

Figure 2: Values of the states after 5 iterations

Figure 3: 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.

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

should give the output as in Figure 2

3.1 Playing with the Agent

Now that we have a way to evaluate the gridworld, 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* (meaning 80% chances of succeeding its intended action, as before). By running the ValueIteration on the BridgeGrid we will have an optimal policy that **does not** cross the bridge, i.e. if you run

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

you will get the output as displayed in Figure 3 and Figure 4

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.

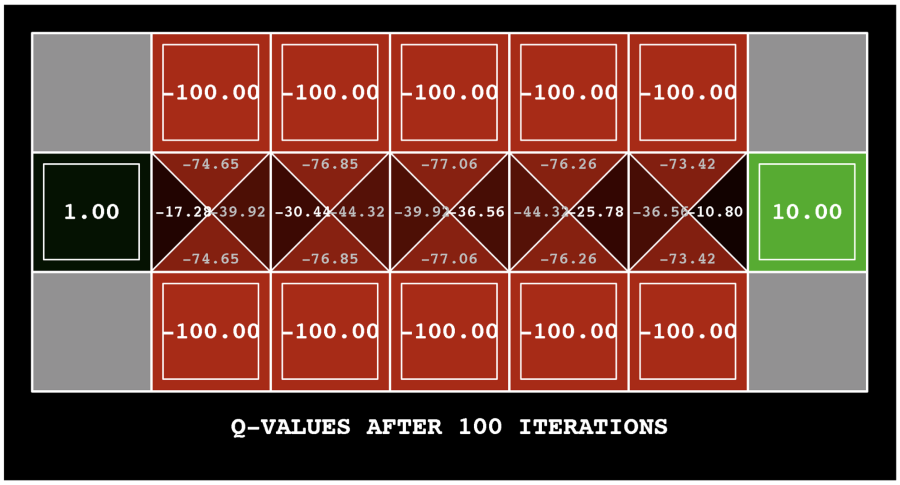

Figure 4: Q-value function of the bridge game. It has a strong incentive to not cross the bridge, and get the low reward on the left instead of the high reward on the right.

Problem 6 (1 points)

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**. Remember, 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 runnin

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

Also, **write in the report your answer** and the reason behind your choice.

4 Q-Learning

Note that the value iteration agent deployed in the previous section does not actually learn from experience. Rather, it ponders its MDP model to arrive at a complete policy before ever interacting with a real environment. When it does interact with the environment, it simply follows the precomputed policy (e.g. it becomes a reflex agent). This distinction may be subtle in a simulated environment like a Gridword, but its very important in the real world, where the real MDP is not available.

We will now turn to Q-learning, i.e. we will stop assuming that a model of the game is available to us to build the value or Q-value functions. You will write a Q-learning agent, which does very little on construction, but instead learns by trial and error from interactions with the environment through the method update(state, action, nextState, reward).

The core algorithm is the one detailed on slide 66 of the lecture. Here the policy applied in the game will be computed from:

$$a = \arg\max_{a} Q(s, a)$$

and the learning algorithm will be:

For given state s, action a, next state s', reward r, do

1: $\delta \leftarrow r + \gamma \max_{a'} Q(s', a') - Q(s, a)$
2: $Q(s, a) \leftarrow Q(s, a) + \alpha\delta$

where δ is some variable that you define and compute, γ is the *discount factor* and α is the *learning rate* and a' are the actions available in s'.

You will need to fill the **update** method taking in consideration the other methods that you can use:

- In the class initialization __**init**__ there is a data structure of type Counter used to store the learning Q-value function: `self.qValues = util.Counter()`. It is a dictionary that maps a tuple `(state, action)` to a real number (the q-value). Consider that we don't know beforehand the states to which our agent will be exposed, so we will figure that out on-the-fly! Compute the values and store them in `self.qValues`.

Problem 7 (2 points)

Implement the method **update**, where the Q-learning happens, and the core of our agent. It receives a state, an action, the next state and the reward obtained in this transition. From these information it must update the Q-value function, following a Q-learning algorithm.

Hints:

- `self.discount` contains the discount factor
- `self.alpha` contains the learning rate
- `self.getQValue(s, a))` returns $Q(s, a)$
- `self.setQValue(s,a, value)` sets $Q(s, a) \leftarrow$ `value`

4.1 Playing and Exploring with the Agent

You can play with the Q-learner agent using manual control by prompting:

```
python gridworld.py -a q -k 5 -m
```

and using the arrows on your keyboard to navigate the game.
The option **-k** will control the number of episodes your agent gets to learn. Watch how the agent learns about the state it was just in, not the one it moves to.

Problem 8 (1 points)

We will now modify the method **getAction** so that it does not necessarily return the best action, but also takes random ones with a given probability ε (given by the attribute `self.epsilon`). The update algorithm will remain the same as before! Modify your method **getAction** so that it picks a random action in the list of admissible actions (for the given state) with probability `self.epsilon`. If there are no legal action for a give state, then your function must return **None** (existing type).

Hints:

- `self.epsilon` contain the default value of epsilon ($= 0.5$)
- You might want to use the function `util.flipCoin(prob)`
- You might want to use the method `random.choice`

You can test your implementation by running

```
python gridworld.py -a q -k 100
```

Your final Q-values should resemble those of your value iteration agent, especially along well-traveled paths. However, your average returns will be lower than what the Q-value function predict because of the random actions and the initial learning phase.

To assess your code, run the autograder:

```
python autograder.py -q q5
```

With no additional code, you should now be able to run a Q-learning crawler robot:

```
python crawler.py
```

This will invoke the crawling robot from class using your Q-learner. Play around with the various learning parameters to see how they affect the agents policies and actions. Note that the step delay is a parameter of the simulation, whereas the learning rate and epsilon are parameters of your learning algorithm, and the discount factor is a property of the environment.

We will now try your Q-learner on the **bridge-of-death**!!

First, train a completely random Q-learner with the default learning rate on the noiseless BridgeGrid for 50 episodes and observe whether it finds the optimal policy. You can do that by running:

```
python gridworld.py -a q -k 50 -n 0 -g BridgeGrid -e 1
```

Now try the same experiment with $\varepsilon = 0$, i.e.

```
python gridworld.py -a q -k 50 -n 0 -g BridgeGrid -e 0
```

Problem 9 (1 points)

Is there an epsilon and a learning rate for which it is highly likely (greater than 99%) that the optimal policy will be learned after 50 iterations? Provide the answer in **question6()** in the class **analysis.py**. It should EITHER return the values that make this possible OR the string `NOT POSSIBLE` if there is none. Remember that you can control epsilon `-e` and the learning rate by `-l`.

Time to play some **Pacman**! Pacman will play games in two phases. In the first phase, training, Pacman will begin to learn about the values of positions and actions. Because it takes a very long time to learn accurate Q-values even for tiny grids, Pacmans training games run in quiet mode by default, with no GUI (or console) display. Once Pacmans training is complete, he will enter a testing (game) mode. When testing, Pacmans self.epsilon and self.alpha will be set to 0.0, effectively stopping Q-learning and disabling exploration, in order to allow Pacman to exploit his learned policy. Test games are shown in the GUI by default.

Note that PacmanQAgent is already defined for you in terms of the QLearningAgent youve already written. PacmanQAgent is only different in that it has default learning parameters that are more effective for the Pacman problem ($\varepsilon = 0.05$, $\alpha = 0.2$, $\gamma = 0.8$). Without any code changes you should be able to run Q-learning Pacman for very tiny grids as follows:

```
python pacman.py -p PacmanQAgent -x 2000 -n 2010 -l smallGrid
```

The command above plays a total of 2010 games, the first 2000 games will not be displayed because of the option -x 2000, which designates the first 2000 games for training (no output). Thus, you will only see Pacman play the last 10 of these games.

During training, you will see output every 100 games with statistics about how Pacman is faring. Epsilon is positive during training, so Pacman will play poorly even after having learned a good policy: this is because he occasionally makes a random exploratory move into a ghost. As a benchmark, it should take between 1,000 and 1400 games before Pacmans rewards for a 100 episode segment becomes positive, reflecting that hes started winning more than losing. By the end of training, it should remain positive and be fairly high (between 100 and 350).

If you want to watch your agent training, use the command:

```
python pacman.py -p PacmanQAgent -n 10 -l smallGrid -a numTraining=10
```

If you want to run proper statistics on the performance of your agent, you can run the command (option -q mutes the test games, allowing you to run fast statistics):

```
python pacman.py -p PacmanQAgent -x 2000 -n 2100 -l smallGrid -q
```

which will run 100 test games, and report the number of wins. Your agent should win 95%-100% of the time.

Problem 10 (2 points)

Complete the report by answering the following questions / reflections:

(a) Discuss briefly your experiments with the crawler, bridgecrossing and pacman.

(b) Detail your reasoning in the answers to Problem 9.

(c) Comment on the use of the ε-greedy policy vs. the best-policy approach? What does it do to the training process?

(d) Imagine and test in your code a very simple way to promote the exploration of unvisited states. What did you do and why?