Q-Learning

Q-Learning seems quite simple, the goal of this tutorial is to implement it on **421** game to create an AI capable of learning by itself to play to this game.

IMT Nord Europe
École Mines-Télécom
IMT-Université de Lille

Set-up

So the goal is to:

1. Implement a new *PlayerQ*
2. At initialization **Q-values** is created empty.
3. At perception steps the player update its **Q-value**
4. At decision step, the player chooses a new action to perform (the best known of for exploring more...)

But as a first move, you can download a template script in python and start to implement our Q-Leraner on top on it.

1. Download a blanc playerQ.py script
2. rename the `playerQQQ.py` in `playerQNames.py` with a Names build in reference to the initials of the developers and add the developers' names on the top command area of the script.
3. Read the script and understand its content.
4. test it: `python3 playerQNames.py`

`playerQQQ.py` is composed of 4 parts: `# MAIN:`, `# ACTIONS:`, `# Q LEARNER:` and, at the end, `# SCRIPT EXECUSION:`. `# MAIN:` define the main function to execute if `playerQQQ.py` is executed as a script. `# ACTIONS:` defines a global variable with the list of possible actions. `# Q LEARNER:` a squeletom of class to implement Q learner players. The particularity of this implementation is that an initial state is arbitrary set on `wakeUp` function and it includes a `stateStr` method that generates a snapshot of the game state as a string variable.

```
# State Machine :
def stateStr(self):
    s= str(self.turn)
    for d in self.dices :
        s += '-' + str(d)
    return s
```

At the end, the group will return to the teacher, for evaluation, this file and only this file. So execute, take the times to properly execute those first instructions and do not hesitate to add all the comments that would help a reader to understand your work.

Q as a dictionary

A simple way to implement **Q** in python language is to implement it as a Dictionnary of dictionaries.

- Python documentation
- On w3school

Implement

At `__init__` method, initializing an empty **Q-values** dictionary will look like:

```
self.qvalues= {}
```

Then, each time the player reaches a new state, it has to generate an initial value for all possible action it would have. So, initializing values for a given state will look like:

```
if state not in self.Q.keys() :
    self.qvalues[state]= { "keep-keep-keep":0.0, "roll-keep-keep":0.0, "keep-roll-
        keep":0.0, "roll-roll-keep":0.0, "keep-keep-roll":0.0, "roll-keep-roll":0.0,
        "keep-roll-roll":0.0, "roll-roll-roll":0.0 }
```

A new state requires to be added to `qvalues` each time it is necessary in the `wakeUp` (the arbitrary initial state: `9-1-1-1`) and the `perceive` methods. Implement its.

Test

To test if the increase in the code works well, you can print the entire dictionary add the end (i.e. in the `main` function, after printing the average score).

```
for st in player.qvalues :
    print( st +": "+ str(player.qvalues[st]) )
```

Update the Q value

Then you can implement the update of **Q** value for the last visited state (`Q[stateStr][actionStr]`). To notice that *updateQ* will require another method to select the maximal value in **Q** for a given state.

Implement

Modifying a value in **qvalues** dictionary will look like (naturally the state and action strings would be variables):

```
self.qvalues["2-6-3-2"]["roll-roll-roll"]= ...
```

At perception step, before to record the new `turn` and `dices` values of the new reached game state, you have to memorize the last reached state: `last= self.stateStr()`. Then, with `last`, `self.action`, `self.stateStr()` end `self.reward` you have all the ingredients to compute the q-value of the `last` state knowing that you performed the `self.action` action and reached `self.stateStr()` state with `self.reward`.

Test

We will consider that a certain number of games match an episode in the learning process. 1000 games for instance. The goal is to play several episodes and observe increase in the *Q-values*.

Modify the `main` function in order to print the *Q-value* of the initial state for instance.

```
print( player.qvalues['9-1-1-1'] )
```

You can also print the number of entrances in the dictionary, the average of the best values of states (the average over the states, by considering the best action to perform) etc...

Choose to exploit or explore

Now the *action* method can randomly select an exploration or an exploitation action. In case of exploration, a random action if performed. In case of exploitation, the playerQ have to search for the best action to perform in the current state.

Implement

Modify the `decide` method to handle exploration/exploitation. It is recommended to separate the 'selection of the best action' in a dedicated method.

Test

The system would mainly choose the best action to perform (considering its knowledge). The goal is to play several episodes and observe increase in the average scores.

Modify the `main` function` in order to compute and print a new average every 1000 games.

- You can now try to answer how many episodes of 1000 games are required to learn a good enough policy (more than an average of 250 points).

Going further:

Do not forget You ~~can~~ must test your code at each development step by executing the code for few games and validate that the output is as expected.

Update our PlayerQ:

1. *PlayerQ* constructor permits users to customize the algorithms parameters ϵ, γ … Let's do it in the `__init__` method with default parameters value.
    - Handle default parameters value in python with w3schools.
2. *PlayerQ* save its learned **Q-values** on a file. To notice that with json module, you can easily read and write a dictionary from a file.
3. *PlayerQ* initialize its **Q-values** by loading a file.
4. A new *PlayerBestQ* simply play the best action always from a given **Q-values** dictionary (without upgrading **Q**).
5. You are capable of plotting the sum over **Q** with one point per episode (with pyplot for instance).