COSC343/AIML402 S2 2024

Lab 6: Grid-world

Objectives:

- **Test your understanding of RL concepts**
- **Implement (and get a deeper understanding of) the Q-learning algorithm**

Reinforcement learning

1. What is it about Reinforcement learning (RL) that makes is a more suitable framework for learning in real world scenarios, as opposed to search methods introduced at the start of the paper?
2. Why is exploration important in RL? What can happen when agent doesn't explore its state space?

Q-learning

Exercise 1: In this exercise you will implement the Q-learning algorithm that figures out the policy for navigating Gridworld. Of course the agent is not going to be given any information about the world (which we know is treacherous and slippery), only rewards (positive or negative) when it ends up in either of the terminal states. Download `cosc343GridWorld.py` from Blackboard into a new project folder and open the folder in VSCode (don't forget to set the interpreter to the cosc343 environment). The `cosc343GridWorld.py` script provides an implementation of the Gridworld environment. You should instantiate the environment at the start of your script as follows:

```python
from cosc343GridWorld import GridWorldEnvironment
import numpy as np
import matplotlib.pyplot as plt

env = GridWorldEnvironment()
```

And then all you need to do is to implement the temporal difference Q-table based policy learning algorithm from Slide 17 of Lecture 12.

Start with defining variables and setting some values for T, γ and α. You should also set a variable for the total number of episodes and maximum number of steps of per episode. For Q-table use a dictionary, initialising it like so:

```
Q = dict()
```

The Gridworld environment allows four actions N, E, S and W in each state, but they are encoded as integer values $0, 1, 2, \text{and}, 3$ respectively. The 11 states of the Gridworld will be encoded as tuples with the (x,y) position of given square.

After instantiating the `gridworld` object you can get total number of states and actions per state using member variables `env.num_states` and `env.num_actions`.

The `env` object keeps track of its internal state – that is, the actions of the agent change the state of the environment. To reset the environment into the initial state (and get the reward for the initial state) you can do:

```
s,R = env.reset()
```

Your Q-table/dictionary should store a 4-element vector with Q-values for actions 0,1,2, and 3 in state s. Of course, it might happen that you haven't encountered s before and your Q-table/dictionary doesn't have a record of it. Hence, whenever you see a new state, you should fetch its Q-values like so:

```
if s not in Q:
  Q[s] = np.zeros((len(env.actions)))

qvector = Q[s]
```

Note that Q being a dictionary, and assuming you have a vector of 4 values at key `s`, then `Q[s]` returns a vector of 4 values and `Q[s][a]` returns the a^{th} values from vector `Q[s]`.

Now, if you want the best action to take in s according to policy conveyed in `Q`, you just need to do

```
a = np.argmax(qvector)
```

On the other hand, if you want to sample the policy (to explore a bit) and chose an action according to probability distribution of the softmax of `Q[s]`, first convert softmax qvalues like so:

```
p = qvalues-np.min(qvalues)
p = np.exp(p/T)
p = p/np.sum(p)
```

where it is assumed T is defined to be some positive value. Do you know why the first line (subtracting the min of `qvalues`) is there? Do you remember, from the lecture, the relationship between value of T and the degree of exploration?

To execute action $a \in \{0, ..., \text{env.num\_actions-1}\}$ and advance the environment to the next state you can do:

```
s_prime, R_prime = env.step(a)
```

As you see above, the `step` method also returns the reward for the new state.

To determine if environment is in a terminal state, you can test whether `env.terminal()` is True or False. For a terminal state s, you can simply set the Q-table/dictionary entry of the R like so:

```
Q[s]=R
```

For non-terminal state, the Q-values in the Q-table/dictionary need to be learned via temporal difference learning.

Finally, you can display a visualisation of the environment and the state of the agent in the environment by using the `evn.render` method like so:

```
env.render()
```

For instance, if you want to show the agent and where it is in its environment, you can call render every time in the loop (you might have to disable the "Show plots in toolwindow" option in Preferences→Tools→Python Scientific for it to work) passing `titleStr` which will include caption about current episode and step:

```
env.render(titleStr='Episode %d, step %d' % (episode+1, step+1))
```

where `episode` and `step` index (from 0) episodes and steps in episode.

After choosing an action, you might want to show visualisation of robot's intention to move in certain direction, and so you can invoke the render method like so:

```
env.render(a=a, titleStr='Episode %d, step %d' % (episode+1, step+1))
```

where it is assumed `a` is an integer from 0 to 3 (inclusive).

If you'd like to see the Q-table along with the visualisation of the environment, pass it into the render function like so:

```
env.render(Q=Q, titleStr='Episode %d, step %d' % (episode+1, step+1))
```

If not all states are defined in the dictionary that's fine – the environment will just assume they haven't been seen and their Q-value vector is all zeros. If you render the environment with the Q table after its every update (the entries that changed since previous update will be highlighted in red).

Finally, at the end of your program, you might want to see the final policy, in which case, invoke the render function for the last time like so:

```
env.render(Q=Q, policy=True)
```

Run your RL algorithm. Does your agent find the optimal policy for every possible starting state? Try to play with the exploration factor T to see how it affect the policy. Consider extending (or reducing) the number of steps per episode. Have a play with different values of γ (discount factor for future rewards) and see if it changes the policy. Finally, you can tell the environment at instantiation time to produce a default reward for any non-terminal state, like so:

```
env = gridworld(R=-0.04)
```

Try negative default reward to see how that affects the policy.