## Week 7. (Oct 1) Introduction to Q-learning

# Read *Adventures in Machine Learning*: [Reinforcement learning tutorial using Python and Keras](http://adventuresinmachinelearning.com/reinforcement-learning-tutorial-python-keras/) up to but not including the section titled “Reinforcement learning with Keras.”

* Examine, understand, and run the code examples. Do you understand the differences among them and what is important about those differences?
* An enhanced version, which also uses epsilon-greedy learning, is available [here](https://drive.google.com/file/d/1bwxdmIJUIKTTP-kSif4Z5caH2eVEXBZJ/view?usp=sharing). Read it, understand it, and run it. Try different values for the parameters: eps, test\_episodes\_per\_trial, training\_episodes\_per\_trial, and alpha.
* Note that run\_a\_trial is nested within run\_trials and that all\_states\_distinguishable and value\_rounded are nested within show\_trial\_results. There are a couple of reasons to do this.

# The nested method is used only by the method within which it is nested. Nesting it avoids cluttering up the list of other methods.

# The nested method uses variables that are defined in the method within which it is nested. It has access to those variables even though they aren’t passed as arguments.

# Write a class to represent the grid in this example. (It has one row and 5 “columns.”) Make it a subclass of the GridMDP class.

# Add it to the [studentGridWorld](https://drive.google.com/file/d/1N0oAWqAMYM1tV1vwtNWR4Lrnag-J75yC/view?usp=sharing) code and run qvalue iteration on it. Since I called my class FourChain I ran it as follows.

ValueIterationAgent(FourChain(), discount=0.95, maxIterations=250, valueType='qvalue')

# Compare the final qvalues the [ValueIteration](https://drive.google.com/file/d/1vhGKjrvugYQ3KlVV-6DtFTv289bQyOQr/view?usp=sharing) agent generates with the final qvalues produced by running the code in step 1. Should they be similar? Should they be identical? Can you explain the actual results?

1. Add methods and values so that the Grid classes can serve as environments. (This is also in the [studentGridWorld](https://drive.google.com/file/d/1N0oAWqAMYM1tV1vwtNWR4Lrnag-J75yC/view?usp=sharing) code.) Most of the work has been done. The following have already been added to the GridMDP class:

# A START\_STATE constant of (0, 0) and a currentState variable that keeps track of the current state.

# The following methods.

**def** isDone(self, nextState):  
"""Return True/False depending on whether the episode is done. The argument  
nextState is the state to which the environment has just transitioned*.*"""  
**return** nextState == self.TERMINAL

**def** reset(self):  
 """  
 Reset the environment to start an episode.  
 """  
 currentState = self.START\_STATE  
 **return** self.START\_STATE  
  
**def** step(self, action):  
 """  
 Take a step with the indicated action.  
 Return the tuple: (nextState, reward, self.isDone(nextState), effAct)  
 effAct is the action actually taken. Because of noise it may not be the same as action.  
 """  
 # Step through the possible effective actions for the given action and select one based on a random choice  
 ceiling = 1  
 choice = random()  
 effAct = **None  
 for** (effAct, prob) **in** self.getEffectiveActionProbs(action):  
 **if** choice > ceiling - prob:  
 **break** ceiling -= prob  
 (reward, nextState) = self.getRewardNextState(self.currentState, effAct)  
 self.currentState = nextState  
 **return** (nextState, reward, self.isDone(nextState), effAct)

# Your job is to add variables and methods to your FourChain class to complete the environment. The FourChain class has no terminal state. An episode ends after 1000 steps. So add a step counter and a constant step limit (=1000) to FourChain. Reset it in reset(). Increment it on each step. Check it to determine whether the episode is over. Let the methods shown above defined in the GridMDP class do most of the work.

# This code at the end of the file will check your work.

**if** \_\_name\_\_ == **'\_\_main\_\_'**:  
  
 env = FourChain()  
 state = env.reset()  
 done = **False  
 while not** done:  
 # Select a random move  
 action = env.getPossibleActions()[choice(range(2))]  
 # Call env.step() with it.  
 (nextState, reward, done, effAct) = env.step(action)  
 # Print results  
 print(**'{}. state: {}, action: {}, effAct: {}, reward: {}, nextState: {}, done: {}'**\  
 .format(env.stepCount, state, action, effAct, reward, nextState, done))  
 # Update the state  
 state = nextState

You should get something like this.

1. state: (0, 0), action: west, effAct: west, reward: 2, nextState: (0, 0), done: False

2. state: (0, 0), action: east, effAct: east, reward: 0, nextState: (1, 0), done: False

3. state: (1, 0), action: west, effAct: west, reward: 2, nextState: (0, 0), done: False

4. state: (0, 0), action: east, effAct: west, reward: 2, nextState: (0, 0), done: False

5. state: (0, 0), action: east, effAct: east, reward: 0, nextState: (1, 0), done: False

6. state: (1, 0), action: east, effAct: east, reward: 0, nextState: (2, 0), done: False

7. state: (2, 0), action: east, effAct: west, reward: 2, nextState: (0, 0), done: False

8. state: (0, 0), action: east, effAct: east, reward: 0, nextState: (1, 0), done: False

9. state: (1, 0), action: west, effAct: east, reward: 0, nextState: (2, 0), done: False

10. state: (2, 0), action: east, effAct: east, reward: 0, nextState: (3, 0), done: False

11. state: (3, 0), action: west, effAct: west, reward: 2, nextState: (0, 0), done: False

12. state: (0, 0), action: east, effAct: east, reward: 0, nextState: (1, 0), done: False

13. state: (1, 0), action: west, effAct: west, reward: 2, nextState: (0, 0), done: False

14. state: (0, 0), action: west, effAct: west, reward: 2, nextState: (0, 0), done: False

15. state: (0, 0), action: west, effAct: west, reward: 2, nextState: (0, 0), done: False

16. state: (0, 0), action: west, effAct: west, reward: 2, nextState: (0, 0), done: False

17. state: (0, 0), action: east, effAct: east, reward: 0, nextState: (1, 0), done: False

18. state: (1, 0), action: west, effAct: west, reward: 2, nextState: (0, 0), done: False

19. state: (0, 0), action: west, effAct: west, reward: 2, nextState: (0, 0), done: False

20. state: (0, 0), action: east, effAct: east, reward: 0, nextState: (1, 0), done: False

21. state: (1, 0), action: east, effAct: east, reward: 0, nextState: (2, 0), done: False

22. state: (2, 0), action: west, effAct: west, reward: 2, nextState: (0, 0), done: False

23. state: (0, 0), action: east, effAct: east, reward: 0, nextState: (1, 0), done: False

24. state: (1, 0), action: east, effAct: east, reward: 0, nextState: (2, 0), done: False

25. state: (2, 0), action: east, effAct: east, reward: 0, nextState: (3, 0), done: False

26. state: (3, 0), action: east, effAct: east, reward: 0, nextState: (4, 0), done: False

27. state: (4, 0), action: east, effAct: east, reward: 10, nextState: (4, 0), done: False

28. state: (4, 0), action: east, effAct: east, reward: 10, nextState: (4, 0), done: False

29. state: (4, 0), action: west, effAct: west, reward: 2, nextState: (0, 0), done: False

30. state: (0, 0), action: east, effAct: east, reward: 0, nextState: (1, 0), done: False

...