learningAgents.py (original)

```python
# learningAgents.py
# -----------------
# Licensing Information: Please do not distribute or publish solutions to this
# project. You are free to use and extend these projects for educational
# purposes. The Pacman AI projects were developed at UC Berkeley, primarily by
# John DeNero (denero@cs.berkeley.edu) and Dan Klein (klein@cs.berkeley.edu).
# For more info, see http://inst.eecs.berkeley.edu/~cs188/sp09/pacman.html

from game import Directions, Agent, Actions

import random,util,time

class ValueEstimationAgent(Agent):
  """
    Abstract agent which assigns values to (state,action)
    Q-Values for an environment. As well as a value to a
    state and a policy given respectively by,

    V(s) = max_{a in actions} Q(s,a)
    policy(s) = arg_max_{a in actions} Q(s,a)

    Both ValueIterationAgent and QLearningAgent inherit
    from this agent. While a ValueIterationAgent has
    a model of the environment via a MarkovDecisionProcess
    (see mdp.py) that is used to estimate Q-Values before
    ever actually acting, the QLearningAgent estimates
    Q-Values while acting in the environment.
  """

  def __init__(self, alpha=1.0, epsilon=0.05, gamma=0.8, numTraining = 10):
    """
    Sets options, which can be passed in via the Pacman command line using -a
alpha=0.5,...
    alpha    - learning rate
    epsilon  - exploration rate
    gamma    - discount factor
    numTraining - number of training episodes, i.e. no learning after these many
episodes
    """
    self.alpha = float(alpha)
    self.epsilon = float(epsilon)
    self.gamma = float(gamma)
    self.numTraining = int(numTraining)

  ####################################
  #    Override These Functions      #
  ####################################
  def getQValue(self, state, action):
    """
    Should return Q(state,action)
    """
    util.raiseNotDefined()

  def getValue(self, state):
    """
    What is the value of this state under the best action?
    Concretely, this is given by

    V(s) = max_{a in actions} Q(s,a)
    """
    util.raiseNotDefined()

  def getPolicy(self, state):
    """
```

```
    What is the best action to take in the state. Note that because
    we might want to explore, this might not coincide with getAction
    Concretely, this is given by

    policy(s) = arg_max_{a in actions} Q(s,a)

    If many actions achieve the maximal Q-value,
    it doesn't matter which is selected.
    """
    util.raiseNotDefined()

  def getAction(self, state):
    """
    state: can call state.getLegalActions()
    Choose an action and return it.
    """
    util.raiseNotDefined()

class ReinforcementAgent(ValueEstimationAgent):
  """
    Abstract Reinforcemnt Agent: A ValueEstimationAgent
      which estimates Q-Values (as well as policies) from experience
      rather than a model

      What you need to know:
          - The environment will call
            observeTransition(state,action,nextState,deltaReward),
            which will call update(state, action, nextState, deltaReward)
            which you should override.
      - Use self.getLegalActions(state) to know which actions
            are available in a state
  """
  ####################################
  #    Override These Functions      #
  ####################################

  def update(self, state, action, nextState, reward):
    """
        This class will call this function, which you write, after
        observing a transition and reward
    """
    util.raiseNotDefined()

  ####################################
  #    Read These Functions          #
  ####################################

  def getLegalActions(self,state):
    """
      Get the actions available for a given
      state. This is what you should use to
      obtain legal actions for a state
    """
    return self.actionFn(state)

  def observeTransition(self, state,action,nextState,deltaReward):
    """
        Called by environment to inform agent that a transition has
        been observed. This will result in a call to self.update
        on the same arguments

        NOTE: Do *not* override or call this function
    """
    self.episodeRewards += deltaReward
    self.update(state,action,nextState,deltaReward)

  def startEpisode(self):
    """
      Called by environment when new episode is starting
```

```
    """
    self.lastState = None
    self.lastAction = None
    self.episodeRewards = 0.0

  def stopEpisode(self):
    """
      Called by environment when episode is done
    """
    if self.episodesSoFar < self.numTraining:
          self.accumTrainRewards += self.episodeRewards
    else:
          self.accumTestRewards += self.episodeRewards
    self.episodesSoFar += 1
    if self.episodesSoFar >= self.numTraining:
      # Take off the training wheels
      self.epsilon = 0.0    # no exploration
      self.alpha = 0.0      # no learning

  def isInTraining(self):
      return self.episodesSoFar < self.numTraining

  def isInTesting(self):
      return not self.isInTraining()

  def __init__(self, actionFn = None, numTraining=100, epsilon=0.5, alpha=0.5,
gamma=1):
    """
    actionFn: Function which takes a state and returns the list of legal actions

    alpha    - learning rate
    epsilon  - exploration rate
    gamma    - discount factor
    numTraining - number of training episodes, i.e. no learning after these many
episodes
    """
    if actionFn == None:
        actionFn = lambda state: state.getLegalActions()
    self.actionFn = actionFn
    self.episodesSoFar = 0
    self.accumTrainRewards = 0.0
    self.accumTestRewards = 0.0
    self.numTraining = int(numTraining)
    self.epsilon = float(epsilon)
    self.alpha = float(alpha)
    self.gamma = float(gamma)

  ################################
  # Controls needed for Crawler  #
  ################################
  def setEpsilon(self, epsilon):
    self.epsilon = epsilon

  def setLearningRate(self, alpha):
    self.alpha = alpha

  def setDiscount(self, discount):
    self.gamma = discount

  def doAction(self,state,action):
    """
        Called by inherited class when
        an action is taken in a state
    """
    self.lastState = state
    self.lastAction = action

  ###################
  # Pacman Specific #
```

```
  ###################
  def observationFunction(self, state):
    """
        This is where we ended up after our last action.
        The simulation should somehow ensure this is called
    """
    if not self.lastState is None:
        reward = state.getScore() - self.lastState.getScore()
        self.observeTransition(self.lastState, self.lastAction, state, reward)
    return state

  def registerInitialState(self, state):
    self.startEpisode()
    if self.episodesSoFar == 0:
        print 'Beginning %d episodes of Training' % (self.numTraining)

  def final(self, state):
    """
      Called by Pacman game at the terminal state
    """
    deltaReward = state.getScore() - self.lastState.getScore()
    self.observeTransition(self.lastState, self.lastAction, state, deltaReward)
    self.stopEpisode()

    # Make sure we have this var
    if not 'episodeStartTime' in self.__dict__:
        self.episodeStartTime = time.time()
    if not 'lastWindowAccumRewards' in self.__dict__:
        self.lastWindowAccumRewards = 0.0
    self.lastWindowAccumRewards += state.getScore()

    NUM_EPS_UPDATE = 100
    if self.episodesSoFar % NUM_EPS_UPDATE == 0:
        print 'Reinforcement Learning Status:'
        windowAvg = self.lastWindowAccumRewards / float(NUM_EPS_UPDATE)
        if self.episodesSoFar <= self.numTraining:
            trainAvg = self.accumTrainRewards / float(self.episodesSoFar)
            print '\tCompleted %d out of %d training episodes' % (
                   self.episodesSoFar,self.numTraining)
            print '\tAverage Rewards over all training: %.2f' % (
                    trainAvg)
        else:
            testAvg = float(self.accumTestRewards) / (self.episodesSoFar -
self.numTraining)
            print '\tCompleted %d test episodes' % (self.episodesSoFar -
self.numTraining)
            print '\tAverage Rewards over testing: %.2f' % testAvg
        print '\tAverage Rewards for last %d episodes: %.2f'  % (
                NUM_EPS_UPDATE,windowAvg)
        print '\tEpisode took %.2f seconds' % (time.time() - self.episodeStartTime)
        self.lastWindowAccumRewards = 0.0
        self.episodeStartTime = time.time()

    if self.episodesSoFar == self.numTraining:
        msg = 'Training Done (turning off epsilon and alpha)'
        print '%s\n%s' % (msg,'-' * len(msg))
```