Introduction to Reinforcement Learning

OCNC 2019

In [1]:
1   import numpy as np
2   import matplotlib.pyplot as plt
3   %matplotlib inline

Classes for minimum environment and agent

In [2]:
1   class Environment:
2       """Class for a reinforcement learning environment""
3
4       def __init__(self, nstate=3, naction=2):
5           """Create a new environment""
6           self.Ns = nstate  # number of states
7           self.Na = naction  # number of actions
8
9       def start(self):
10          """start an episode""
11          # randomly pick a state
12          self.state = np.random.randint(self.Ns)
13          return self.state
14
15       def step(self, action):
16          """step forward given an action""
17          # random reward
18          self.reward = np.random.random()  # between 0 and 1
19          # random state transition
20          self.state = np.random.randint(self.Ns)
21          return self.reward, self.state
In [3]:
1 class Agent:
2     """Class for a reinforcement learning agent""
3     
4     def __init__(self, nstate, naction):
5         """Create a new agent""
6         self.Ns = nstate  # number of states
7         self.Na = naction  # number of actions
8     
9     def start(self, state):
10        """first action, without reward feedback""
11        # randomly pick an action
12        self.action = np.random.randint(self.Na)
13        return self.action
14     
15     def step(self, reward, state):
16        """learn by reward and take an action""
17        # do nothing for reward
18        # randomly pick an action
19        self.action = np.random.randint(self.Na)
20        return self.action

In [4]:
1 class RL:
2     """Reinforcement learning by interaction of Environment and Agent""
3     
4     def __init__(self, environment, agent, nstate, naction):
5         """Create the environment and the agent""
6         self.env = environment(nstate, naction)
7         self.agent = agent(nstate, naction)
8     
9     def episode(self, tmax=50):
10        """One episode""
11        # First contact
12        state = self.env.start()  
13        action = self.agent.start(state)
14        # Table of t, r, s, a
15        Trsa = np.zeros((tmax+1, 4))
16        Trsa[0, :] = [0, 0, state, action]
17        # Repeat interaction
18        for t in range(1, tmax+1):
19            reward, state = self.env.step(action)
20            action = self.agent.step(reward, state)
21            Trsa[t, :] = [t, reward, state, action]
22        return Trsa
23     
24     def run(self, nrun=10, tmax=50):
25        """Multiple runs of episodes""
26        Return = np.zeros(nrun)
27        for n in range(nrun):
28            r = self.episode(tmax)[n, 1]  # reward sequence
29            Return[n] = sum(r)
30        return Return
Q learning of Pain-Gain task

In [5]:

class PainGain(Environment):
    """Pain-Gain environment ""
    def __init__(self, nstate=4, naction=2, gain=6):
        super().__init__(nstate, naction)
        # setup the reward function as an array
        self.R[:,1] = -1  # small pains for action 1
        self.R[0,0] = -gain # large pain
        self.R[-1,1] = gain  # large gain
    def step(self, action):
        """step by an action""
        self.reward = self.R[self.state, action]  # reward
        self.state = (self.state + 2*action-1)%self.Ns  # move left
        return(self.reward, self.state)
class QL(Agent):
    """Class for a Q-learning agent"""
    def __init__(self, nstate, naction):
        super().__init__(nstate, naction)
        # allocate Q table
        self.Q = np.zeros((nstate, naction))
        # default parameters
        self.alpha = 0.1  # learning rate
        self.beta = 2.0   # inverse temperature
        self.gamma = 0.9  # discount factor

    def boltzmann(self, s, q):
        """Boltzmann selection""
        pa = np.exp(s.beta*q)  # unnormalized probability
        pa = pa/sum(pa)        # normalize
        return np.random.choice(s.Na, p=pa)

    def start(self, s, state):
        """first action, without reward feedback""
        # Boltzmann action selection
        self.action = self.boltzmann( self.Q[state,:])
        # remember the state
        self.state = state
        return self.action

    def step(self, s, reward, state):
        """learn by reward and take an action""
        # TD error: self.state/action retains the previous ones
        delta = reward + s.gamma*max(s.Q[state,:]) - s.Q[:]
        # Update the value for previous state and action
        s.Q[s.state,s.action] += s.alpha*delta
        # Boltzmann action selection
        s.action = self.boltzmann( s.Q[state,:])
        # remember the state
        s.state = state
        return s.action

Setup and Run

In [7]:
# Setup Pain-Gain environment and Q-learning agent
pgq = RL(PainGain, QL, 4, 2)
pq.env.R

Out[7]: array([[-6., -1.],
               [ 1., -1.],
               [ 1., -1.],
               [ 1.,  6.]])
# Run an episode of 50 trials
trsa = pgg.episode(50)

# plot the time course of state, action, reward,
plt.plot(trsa[::,1:], "o-")
plt.legend(['reward', 'state', 'action'])
plt.xlabel("time")

# plot the action values
plt.bar(np.arange(pgg.agent.Ns)-0.2, pgg.agent.Q[::, 0], 0.4)  # act.
plt.bar(np.arange(pgg.agent.Ns)+0.2, pgg.agent.Q[::, 1], 0.4)  # act.
plt.legend(['action0', 'action1']);
plt.xticks(range(pgg.agent.Ns)); plt.xlabel("state"); plt.ylabel("Q")

# reduce the discount factor
pgg.agent.gamma = 0.3

# restore the discount factor
pgg.agent.gamma = 0.9

# increase the inverse temperature
pgg.agent.beta = 10

# reduce the inverse temperature
pgg.agent.beta = 1