**ENPM808F ROBOT LEARNING**

**HOMEWORK 4**

**Shanmukha Manoj**

**Q-learning:**

Algorithm to create a map/ policy π that maps states to actions. A table Q is maintained which stores value at every state and action, and policy is chosen such that an action at a state maximizes the Q value. During the learning phase, initially Q’s are initialized to be zeros, and after every step or at the end of an episode Q’s are updated based on the following equation,

Here is the state achieved after applying action on state

**AI player for Tic-Tac-Toe:**

A basic 3x3 tic-tac-toe board is chosen and an AI player that learns the game based on Q-learning and self-play is designed in python.

The following code is a python class Board that can create a 3x3 tic-tac-toe board,

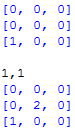
1. **class** Board:
2. #initial tiles with 0
3. #X->1, O->2
4. **def** \_\_init\_\_(self):
5. self.board=[[0,0,0],[0,0,0],[0,0,0]]
6. **def** update(self,pos,mark):
7. **assert**(mark==1 **or** mark==2)
8. **assert**(self.board[pos[0]][pos[1]]==0)
9. self.board[pos[0]][pos[1]]=mark
10. **def** possible\_moves(self):
11. moves=[]
12. **for** i **in** xrange(3):
13. **for** j **in** xrange(3):
14. **if** self.board[i][j]==0:
15. moves.append((i,j))
16. **return** moves
17. **def** get\_winner(self):
18. #rows
19. **for** i **in** xrange(3):
20. mark=self.board[i][0]
21. **if** mark==0:
22. **continue**
23. **elif** self.board[i][1]==mark **and** self.board[i][2]==mark:
24. **return** mark
25. #columns
26. **for** i **in** xrange(3):
27. mark=self.board[0][i]
28. **if** mark==0:
29. **continue**
30. **elif** self.board[1][i]==mark **and** self.board[2][i]==mark:
31. **return** mark
32. #diagonals
33. mark=self.board[0][0]
34. **if** mark!=0 **and** self.board[1][1]==mark **and** self.board[2][2]==mark:
35. **return** mark
36. mark=self.board[0][2]
37. **if** mark!=0 **and** self.board[1][1]==mark **and** self.board[2][0]==mark:
38. **return** mark
39. #if draw
40. **return** 0
41. **def** is\_game\_over(self):
42. **if** len(self.possible\_moves())==0:
43. **return** True
44. **if** self.get\_winner():
45. **return** True
46. **return** False
47. **def** reset(self):
48. self.board=[[0,0,0],[0,0,0],[0,0,0]]
49. **def** \_\_str\_\_(self):
50. **return** str(self.board[0])+'\n'+str(self.board[1])+'\n'+str(self.board[2])+'\n'

The board is visualized as follows,



In the above board, 0’s represent the empty tiles where the next player can mark, and 1 in the bottom left corner is the place where player 1 has marked. When human player is playing, input is of the form ‘i, j’ where i, j are integers in

The following figure shows the first cycle (player1 plays followed by player 2 plays) of the game, where player 1 is the AI player and human is player 2



The following code is class Player that describes the AI player,

1. **import** random
3. **class** Player:
4. #id=X or O; X->1, O->2
5. **def** \_\_init\_\_(self,board,mark):
6. self.board=board
7. self.id=mark
8. self.Q={}
9. self.moves=[]
10. self.states=[]
11. self.REWARDS=[-2,10,-10] #-2 for draw, 10 for win, -10 for loss
12. **def** Q\_value(self,s,a):
13. **if** self.Q.get((s,a)) **is** None:
14. self.Q[(s,a)]=0
15. **return** self.Q[(s,a)]
16. #exploiting the fact that game (X,O) w.r.t X is game (O,X) w.r.t O
17. **def** state\_wrt\_id(self):
18. s=[]
19. **for** i **in** xrange(3):
20. **for** j **in** xrange(3):
21. **if** self.board.board[i][j]==self.id:
22. c=1
23. **else**:
24. c=2
25. **if** self.board.board[i][j]==0:
26. c=0
27. s.append(c)
28. **return** tuple(s)
29. **def** random\_play(self):
30. s=self.state\_wrt\_id()
31. moves=self.board.possible\_moves()
32. m=random.choice(moves)
33. self.board.update(m,self.id)
34. self.moves.append(m)
35. self.states.append(s)
36. **def** play(self,g):
37. s=self.state\_wrt\_id()
38. moves=self.board.possible\_moves()
39. **if** random.random()<0.2/(g+1):
40. m=random.choice(moves)
41. **else**:
42. q=[self.Q\_value(s,move) **for** move **in** moves]
43. maxQ=max(q)
44. **if** q.count(maxQ)>1:
45. best\_moves=[i **for** i **in** range(len(moves)) **if** q[i]==maxQ]
46. i=random.choice(best\_moves)
47. **else**:
48. i=q.index(maxQ)
49. m=moves[i]
50. self.board.update(m,self.id)
51. self.moves.append(m)
52. self.states.append(s)
53. **def** update\_Q(self):
54. **for** i **in** xrange(len(self.moves)-1,-1,-1):
55. s=self.states[i]
56. m=self.moves[i]
57. **if** i==len(self.moves)-1:
58. winner=self.board.get\_winner()
59. **if** winner==self.id:
60. c=1
61. **else**:
62. c=2
63. **if** winner==0:
64. c=0
65. r=self.REWARDS[c]
66. q=self.Q\_value(s,m)
67. self.Q[(s,m)]=q+0.8\*(r-q)
68. **else**:
69. r=0
70. q=self.Q\_value(s,m)
71. s\_=self.states[i+1]
72. moves=[]
73. **for** j **in** xrange(3):
74. **for** k **in** xrange(3):
75. **if** s\_[3\*j+k]==0:
76. moves.append((j,k))
77. q\_=[self.Q\_value(s\_,move) **for** move **in** moves]
78. maxQ=max(q\_)
79. self.Q[(s,m)]=q+0.8\*(r+0.9\*maxQ-q)
80. self.states=[]
81. self.moves=[]

A player object is instantiated by providing the board on which it is playing and it’s corresponding mark (1 for ‘X’ and 2 for ‘O’) as input arguments to the constructer. The player object also has the attribute ‘Q’ which stores the Q values corresponding to a state, action pair. The attribute Q is a dictionary with keys as tuple (s, a) where s is state which in turn is a tuple of 9 elements representing the marks at respective places in the 3x3 board, and a is the action which is a tuple i, j denoting the place at which the player would mark at this step.

AI Player is rewarded with -2 if the game ends as a tie, +10 if he wins and -10 he loses. All moves before the game ends have rewards 0. The following code describes the learning process,

1. **from** board **import** Board
2. **from** player **import** Player
3. **import** pickle
5. **if** \_\_name\_\_=='\_\_main\_\_':
6. board=Board()
7. player1=Player(board,1)
8. player2=Player(board,2)
9. ##
10. **for** i **in** xrange(2000):
11. **while** True:
12. **if** board.is\_game\_over():
13. **break**
14. player1.random\_play()
15. #print board
16. **if** board.is\_game\_over():
17. **break**
18. player2.random\_play()
19. #print board
20. #print 'player '+str(board.get\_winner())+' wins'
21. #print '--------------------------------------'
22. player1.update\_Q()
23. player2.update\_Q()
24. board.reset()
25. **for** i **in** xrange(2000,4000):
26. **while** True:
27. **if** board.is\_game\_over():
28. **break**
29. player1.play(i/100)
30. #print board
31. **if** board.is\_game\_over():
32. **break**
33. player2.random\_play()
34. #print board
35. #print 'player '+str(board.get\_winner())+' wins'
36. #print '--------------------------------------'
37. player1.update\_Q()
38. player2.update\_Q()
39. board.reset()
40. **for** i **in** xrange(4000,6000):
41. **while** True:
42. **if** board.is\_game\_over():
43. **break**
44. player1.random\_play()
45. #print board
46. **if** board.is\_game\_over():
47. **break**
48. player2.play(i/100)
49. #print board
50. #print 'player '+str(board.get\_winner())+' wins'
51. #print '--------------------------------------'
52. player1.update\_Q()
53. player2.update\_Q()
54. board.reset()
55. **for** i **in** xrange(6000,10000):
56. **while** True:
57. **if** board.is\_game\_over():
58. **break**
59. player1.play(i/100)
60. #print board
61. **if** board.is\_game\_over():
62. **break**
63. player2.play(i/100)
64. #print board
65. #print 'player '+str(board.get\_winner())+' wins'
66. #print '--------------------------------------'
67. player1.update\_Q()
68. player2.update\_Q()
69. board.reset()
70. ##
71. Q=player1.Q
72. with open('Q\_1','wb') as f:
73. pickle.dump(Q,f,protocol=pickle.HIGHEST\_PROTOCOL)
74. Q=player2.Q
75. with open('Q\_2','wb') as f:
76. pickle.dump(Q,f,protocol=pickle.HIGHEST\_PROTOCOL)

Two AI players are instantiated with the same board but different marks – player1 with mark 1 and player2 with mark 2. For each game, player1 makes the first move and player2 makes the second. Each player saves the actions it took, and states it obtained from those actions in the player object attributes moves and states respectively. And then at the end of the game, the Q values are updated by calling the player class member function update\_Q for both the players and the board is reset.

For the first 2000 games, both players play randomly, then for the next 2000 games, player1 plays according to policy depending on Q values and player 2 plays random, then for the next 2000 games player1 makes random moves and player2 plays according to policy and for the last 4000 games both the players play according to policy. This is for improving exploration of new states. Over a total of 10000 games good policies have attained.

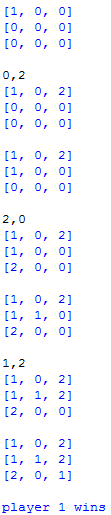
Q\_1 is the file with the policy with AI player starting first and Q\_2 is the policy file with AI player starting second. Using those two, the game is coded as follows,

1. **from** player **import** Player
2. **from** board **import** Board
3. **import** pickle
4. **import** random
6. board=Board()
7. s=[]
8. **for** i **in** xrange(3):
9. **for** j **in** xrange(3):
10. s.append(board.board[i][j])
12. **def** human\_play(b,f):
13. m,n=input()
14. **if** f=='human':
15. b.update((m,n),1)
16. **else**:
17. b.update((m,n),2)
19. **if** \_\_name\_\_=='\_\_main\_\_':
20. first\_togo='human'
21. **if** random.random()<=0.5:
22. first\_togo='AI'
23. **if** first\_togo=='human':
24. player=Player(board,2)
25. with open('Q\_2','rb') as f:
26. player.Q=pickle.load(f)
27. **while** True:
28. **if** board.is\_game\_over():
29. **break**
30. human\_play(board,first\_togo)
31. **print** board
32. **if** board.is\_game\_over():
33. **break**
34. player.play(200)
35. **print** board
36. **else**:
37. player=Player(board,1)
38. with open('Q\_1','rb') as f:
39. player.Q=pickle.load(f)
40. **while** True:
41. **if** board.is\_game\_over():
42. **break**
43. player.play(200)
44. **print** board
45. **if** board.is\_game\_over():
46. **break**
47. human\_play(board,first\_togo)
48. **print** board
49. **print** 'player '+str(board.get\_winner())+' wins'

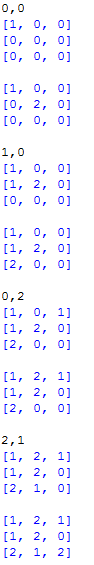
The variable first\_togo decides if human goes first or the AI player goes first in the game. If human goes first then the AI player is assigned policy according to Q\_2, otherwise it is assigned policy according to Q\_1. The AI player has learnt the best policy.

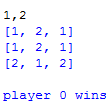
If AI player starts first, and the human player plays optimally to win, the best possible outcome for the human player is to draw the game, and the only we it can happen if human’s first move is at position (1, 1), any other move results in win to AI player.

With human player starting first and he playes optimally to win, everytime the game results in draw.



In the above screenshot, game represents AI player starting first and human player making his first move at a different position than (1, 1). The result is player1 i.e. AI player wins.





In the above screenshot, game represents human player starting first and AI player making its first move at position (1, 1). The result is a tie.

Video showing game play: <https://www.youtube.com/watch?v=9ybEc3Ifp_k>

**Replacing Q table with ANN:**

ANNs can be used for function approximation. Here Q table can be realized as a function mapping a 11 dimensional input – state of 9 dimensions (marked 3x3 board) and action of 2 dimensions, onto a 1 dimensional output i.e. Q value. Hence instead of maintaining a table for that, it can be realized as a neural network and outputs of new inputs can be approximated with the network.

Keras is a python library that works on Theano for designing NNs. Following code is an attempt in applying Keras library to design a network replacing the Q table for the tic-tac-toe AI player,

1. **from** keras.models **import** Sequential
2. **from** keras.layers **import** Dense
3. **import** numpy
4. **import** time
5. **import** pickle
6. seed=7
7. numpy.random.seed(seed)
8. # load data
9. with open('Q\_s','rb') as f:
10. Q\_data=pickle.load(f)
11. n=len(Q\_data)
12. d=11
13. x\_=Q\_data.keys()
14. # split into input (X) and output (Y) variables
15. X=[]
16. **for** i **in** range(n):
17. x=[]
18. **for** j **in** range(9):
19. x.append(x\_[i][0][j])
20. **for** j **in** range(2):
21. x.append(x\_[i][1][j])
22. X.append(x)
23. Y=Q\_data.values()
24. # create model
25. model=Sequential()
26. model.add(Dense(20,input\_dim=d,init='uniform',activation='relu'))
27. model.add(Dense(20,init='uniform',activation='relu'))
28. model.add(Dense(20,init='uniform',activation='relu'))
29. model.add(Dense(1,init='uniform',activation='sigmoid'))
30. # Compile model
31. model.compile(loss='mean\_squared\_error',optimizer='adam',metrics=['accuracy'])
32. # Fit the model
33. model.fit(X,Y,nb\_epoch=50000,batch\_size=100,verbose=0)
34. time.sleep(0.1)
35. # evaluate the model
36. scores=model.evaluate(X,Y)
37. **print**("%s: %.2f%%" % (model.metrics\_names[1],scores[1]\*100))
38. with open('NN\_keras','wb') as f:
39. pickle.dump(model,f,protocol=pickle.HIGHEST\_PROTOCOL)

The work is still in progress. But I believe with the use of ANN as function approximator, number of training episodes would reduce. Without ANNs for learning 10000 games are being played by the AI player during the learning phase. Since outputs from the new inputs can be approximated using the ANN, for a fewer training episodes the player would learn a good strategy for playing the game.

**Reference for the code:** <http://machinelearningmastery.com/tutorial-first-neural-network-python-keras/>