| **Problem Set 4 - Reinforcement Learning** | **Due Date: 10 October 2017** |
|---|---|

In this problem set, we are going to implement how to solve MDPs using value iterations and doing online learning with Q-Learning (i.e. reinforcement learning).

- The project is based on the Problem Set from Berkeley's Intro to AI course. The site you need is here: http://ai.berkeley.edu/reinforcement.html

- There is an autograder you can use to ensure you have the right answers.

- You are only required to complete all the questions:
    - Q1: Value Iteration
    - Q2: Bridge Crossing Analysis
    - Q3: Policies
    - Q4: Q-Learning
    - Q5: Epsilon Greedy
    - Q6: Bridge Crossing Revisited
    - Q7: Q-Learning and Pacman
    - Q8: Approximate Q-Learning

- For the neural network portion, we are going to build on our approximate q-learning agent, by replacing the feature extraction and linear weightage of features to approximate a q-value with a neural network.

  **First of all, this doesn't work that well for me, so it might not be as satisfying to solve as the other questions. I'm just looking for your understanding of using neural networks in reinforcement learning with a running implementation. It doesn't matter if your agent still ends up being not better than random.**

  Why do we want to do this? Perhaps we do not have the domain expertise to figure out the best features to extract for our problem. And it also leads to a more generalised abstraction where we don't need to know a lot about our problem domain in order to solve it.
    - First, understand how to use the keras and theano/tensorflow libraries by going through the example on this site: http://outlace.com/Reinforcement-Learning-Part-3/
    - Next, install keras and theano or tensorflow on your development machine. This might take a bit of time because of installation issues. I find it best to use conda to help in the installation, though you might need administrative access on the machine
    - We are going to create a similar class to ApproximateQAgent in qlearningAgents.py. Copy the following code into your qlearningAgents.py file:

```
class NeuralNetQAgent(PacmanQAgent):
    """
       NeuralNetQAgent

       You should only have to overwrite getQValue
       and update.  All other QLearningAgent functions
```

```
      should work as is.
    """
    def __init__(self, extractor='IdentityExtractor', **args):
        PacmanQAgent.__init__(self, **args)
        """
          Initialise our neural network here. Instead of figuring out
          features, we will throw in our entire state as a list

          We don't know the size of our game here, so we can't create our
          neural network yet. We'll delay the creation of our neural network
          until the first time we are asked about it
        """
        self.nnet = None

    def getQValue(self, state, action):
        """
          Should return Q(state,action) by asking our neural network
        """
        if self.nnet is None:
            self.nnet = NeuralNetwork(state)

        return self.nnet.predict(state, action)

    def update(self, state, action, nextState, reward):
        """
            Should update our neural network based on transition
        """
        if self.nnet is None:
            self.nnet = NeuralNetwork(state)

        "*** YOUR CODE HERE ***"
```

- Instead of a featExtractor object, we are going to rely on a neural network. Otherwise, everything else is similar to the ApproximateQAgent class.

- I've given you a starting point for your NeuralNetwork class with the input inspired by the example on the site provided earlier: http://outlace.com/Reinforcement-Learning-Part-3.

```
from keras.models import Sequential
from keras.layers.core import Dense, Dropout, Activation
from keras.optimizers import RMSprop
import numpy as np

class NeuralNetwork():
    def __init__(self, state):
        """
        We can only initialise our neural network with a state
        so we can determine the size of our world
        Size will be:
            a) walls + food + ghost: 3 * width * height
            b) pacmanPosition: width * height
            c) nextPacmanPosition: width * height
        """
        walls = state.getWalls()
        self.width = walls.width
        self.height = walls.height
        self.size = 5 * self.width * self.height

        "*** YOUR CODE HERE ***"


    def predict(self, state, action):
        reshaped_state = self.reshape(state, action)
```

```
        "*** YOUR CODE HERE ***"

    def update(self, state, action, y):
        reshaped_state = self.reshape(state, action)
        "*** YOUR CODE HERE ***"

    def reshape(self, state, action):
        """
            a) walls + food + ghost: 3 * width * height
            b) pacmanPosition: width * height
            c) nextPacmanPosition: width * height
        """
        reshaped_state = np.empty((1, 2 * self.size))

        food = state.getFood()
        walls = state.getWalls()
        for x in range(self.width):
            for y in range(self.height):
                reshaped_state[0][x * self.width + y] = int(food[x][y])
                reshaped_state[0][self.size + x * self.width + y] = int(walls[x][y])

        ghosts = state.getGhostPositions()
        ghost_states = np.zeros((1, self.size))
        for g in ghosts:
            ghost_states[0][int(g[0] * self.width  + g[1])] = int(1)

        # compute the location of pacman after he takes the action
        x, y = state.getPacmanPosition()
        dx, dy = Actions.directionToVector(action)
        next_x, next_y = int(x + dx), int(y + dy)

        pacman_state = np.zeros((1, self.size))
        pacman_state[0][int(x * self.width + y)] = 1

        pacman_nextState = np.zeros((1, self.size))
        pacman_nextState[0][int(next_x * self.width + next_y)] = 1

        reshaped_state = np.concatenate((reshaped_state, ghost_states, pacman_state,
pacman_nextState), axis=1)
        return reshaped_state
```

- As it turns out, the neural network needs a lot of training in order to do something just slightly better than random. I've created a simple layout for you to use to train in. Create a file customTestGrid.lay in the layouts/ subdirectory with the following contents:

```
%%%%%%%%
%P    .%
% .%. %
%. % .%
%     .%
%%%%%%%%
```

- And try testing with this command: `python pacman.py -p NeuralNetQAgent -x 500 -n 505 -l customTestGrid`

- Submit your answer as a zip file with all the files, including the supporting files which you did not edit.