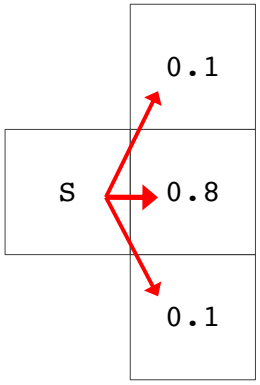
Machine Learning (CS 419/519)
Assignment 05 (50 points)
Due 5:00 PM, Wednesday, 12 December 2018

In this assignment, you will be implementing two versions of Q-learning for a straightforward navigation task; one version will use the basic algorithm, whereas the other will adjust weights that can then be used to compute Q-values for all state-action pairs.

The download for this assignment contains a single text-file, `pipe_world.txt`, which describes a simple environment in which:

- `S` indicates the starting location for an agent.
- `G` indicates a goal location.
- `_` indicates an empty location.
- `M` indicates the location of an explosive mine.

In this environment, the agent begins in the starting location, and can move in one of four directions (up, down, left, and right). Any move that attempts to leave the bounds of the map fails (and the agent stays where it is). For any other move, the move succeeds with probability 0.8, and the agent enters the desired square; with probability 0.2, the agent slips to one side or the other as it moves in the expected direction, with each direction of slippage being equally probable. Thus, for instance, if the agent starts in the state marked `S` in the figure below and attempts to go to the right, it will end up in one of 3 possible locations, each with the probability given.



As the agent moves, it receives rewards as follows:

- If it enters a location containing a mine, it receives reward $r_M = -100$.
- If it enters the goal location, it receives reward $r_G = 0$.
- Any other result leads to reward $r_O = -1$.

For full points, you will write code that does the following:

1. (*20 pts.*) Your code will first perform Q-learning on the problem. Your code should:

   (a) Use a data structure to store explicit values $Q(s, a)$ for each state (location in the grid-world) and action encountered during learning.

   (b) Perform $10,000$ learning episodes; each episode will continue until the agent:

      i. Reaches the goal location.

      ii. Hits a mine.

      iii. Takes a number of steps equal to the entire size (width $\times$ height) of the environment.

      At the beginning of each episode, the agent will begin from the starting location.

   (c) You will use control parameters as follows:

      i. Future discount: $\gamma = 0.9$.

      ii. Policy randomness: you will start with $\epsilon = 0.9$; at the end of every 200 learning episodes, you will reduce the rate, setting it to $\epsilon = 0.9/(\lfloor E/200 \rfloor + 1)$. where E is the number of the episode just completed. This means that after $E = 200, 400, 600, \ldots$, we will have $\epsilon = 0.45, 0.3, 0.225, \ldots$. **Note**: on the final run of learning, the agent should set $\epsilon = 0$, so that it acts in a purely greedy fashion.

      iii. Learning rate: you will start with $\alpha = 0.9$; at the end of every 1000 learning episodes, you will reduce the rate, setting it to $\alpha = 0.9/(\lfloor E/1,000 \rfloor + 1)$. where E is the number of the episode just completed. This means that after $E = 1000, 2000, 3000, \ldots$, we will have $\alpha = 0.45, 0.3, 0.225, \ldots$.

   (d) After every 100 learning episodes, your code will pause and do 50 test-runs of its current greedy policy. That is, it will do 50 runs, from the starting location, choosing the action a that maximizes $Q(s, a)$ for each state s (location) it encounters; each run will continue until one of the three stopping conditions indicated above is met. During the runs, the program will keep track of the total reward received, and at the end will write out the average reward to a file. (I recommend writing to a CSV file, as this will make graphing your results later easier.) **Note**: on the last run of learning, this will be a test of the average performance of the final learned policy.

2. (*20 pts.*) After the above is complete, your code will do a feature-based version of Q-learning. In this version, the number of learning episodes and all control parameters will be exactly as before, and again the code will print out the average reward gained over 50 test-runs of its current greedy policy, every 100 learning episodes. However, in this version, there will be no explicit data structure to store Q-values; instead the code will do the following:

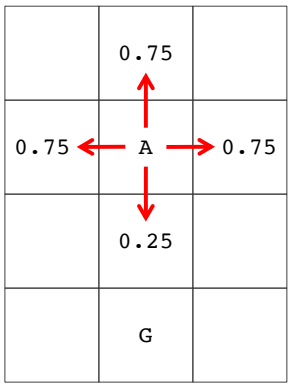
   (a) It will store a weight vector $\mathbf{w} = (w_1, w_2)$; initially, $\mathbf{w} = (0, 0)$.

   (b) For any state-action pair (s, a), there will be a feature vector $\mathbf{f}_{(s,a)} = (f_1, f_2)$ as follows:

      i. f_1 is a *normalized Manhattan distance* metric, calculated by assuming that the action a occurs with no slippage, and then returning $f_1 = MD(s')/MD^+$, where s' is the location that would result from that action, $MD(s')$ is the Manhattan distance* from s' to the goal location, and MD^+ is the maximum possible such distance.

*See: `https://en.wikipedia.org/wiki/Taxicab_geometry`

As an example, consider the image below. If we suppose the agent is in location s marked with an `A`, then for each of the four actions s' would be the state indicated by the corresponding arrow. In this smaller version of the grid, $MD^+ = 4$ (from either of the top corners to the goal marked `G`). Thus, each of the possible directions gives us the f_1 value indicated; for instance, for the down direction, we have $f_1 = 1/4 = 0.25$.



ii. f_2 is based upon the action a and two *inverse distance* metrics, calculated based upon the distance to the mines to the left and right of an agent. As shown in the image below, each inverse distance is $1/d$, where d is the number of moves needed to reach the mine in that direction. For an agent starting in the square marked `A`, for instance, the nearest mine to the left is 2 squares away, for an inverse distance of $1/2 = 0.5$, and to the right the inverse distance is $1/4 = 0.25$.

We set f_2 dependent upon action a as follows:

- If a is a move left or right, then f_2 is set to the inverse distance in that direction.
- If a is a move up or down, then f_2 is set to the *minimum* of the two distances.

Thus, for the example just given, we have:

$$f_2 = 0.5 \text{ if } a = \text{left} \qquad f_2 = 0.25 \text{ if } a = \text{up}$$
$$f_2 = 0.25 \text{ if } a = \text{right} \qquad f_2 = 0.25 \text{ if } a = \text{down}$$

(c) For any state-action pair the Q-value is calculated as follows:

$$Q(s,a) = \mathbf{w} \cdot \mathbf{f}_{(s,a)} = w_1 f_1 + w_2 f_2$$

(d) After each Q-learning iteration, we update weights as explained in lecture 24:

$$\delta = r + \gamma \max_{a'} Q(s', a') - Q(s,a)$$
$$\forall i,\ w_i \leftarrow w_i + \alpha\, \delta\, f_i(s,a)$$

3. (*10 pts.*) Along with your code (written in clear, well organized, and well documented fashion), you will hand in the following:

   - A text-file containing the final policy learned by each of the two algorithms. The policies should each be labeled to indicate which is which. For each square of the grid, print a single character (row by row, column by column):
     - For any square containing a mine or the goal, simply print `M` or `G`, respectively, as in the original input.
     - For any other square, print out the action chosen, using `U`, `D`, `L`, and `R` for up, down, left, and right, respectively.
   - A graph containing the average-value data generated by the two algorithms after every 100 episodes. The graph axes should be clearly labeled, and it should be clear which data-series is which.
   - A text-file containing some analysis of the results shown in the graph and final policies. You need a couple of paragraphs to explain what you are seeing in the data and why (you believe) you get the results that you do.

You will hand in a single compressed archive containing:

- Complete source code.
- Any instructions necessary to compile and/or execute the code.
- The text-files and graph described above.