3547 Assignment 4

This assignment uses Deep Q Learning to enable you to build an agent that will learn to play WumpusWorld from experience interacting with your Environment code from Assignment 1. The algorithm we will use is DQN with a replay buffer and target network. A code example that is a good start for what you will need can be found at https://github.com/DeepReinforcementLearning/DeepReinforcementLearningInAction/blob/master/Chapter%203/Ch3_book.ipynb. This example is for a simpler GridWorld than WumpusWorld but has all of the key elements you'll need. It uses PyTorch for the deep neural net, which you can either use, or if you prefer, convert to an equivalent TensorFlow network.

Instructions

1) Create a new agent called DeepQAgent. The agent's belief state will need to include the following features[1]:
   a) A 4 x 4 array with 1 where the agent is (0 everywhere else)
   b) A set of 4 indicator variables representing the agent's orientation
   c) A 4 x 4 array with 1 everywhere the agent has visited (else 0)
   d) A 4 x 4 array with 1 everywhere a Stench has been detected (else 0)
   e) A 4 x 4 array with 1 everywhere a Breeze has been detected (else 0)
   f) A Boolean which is True if the agent has the Gold
   g) A Boolean which is True if the agent perceives a Glitter
   h) A Boolean which is True is the agent has the arrow (i.e. has not used it)
   i) A Boolean which is True if the agent has ever heard a Scream
2) Replace the environment and agent in the example code with yours and modify the neural net to use this larger list of features. The gridworld in the example code only has walls[2] (W), pits (P) and a goal (+) so it is much simpler than WumpusWorld. It also has 4 move direction actions and the only state is the agent's location vs. 6 WumpusWorld actions and state features as above.
3) Since the game has a built-in -1 penalty for each action and is episodic there is no requirement for a discount factor (gamma) but you may find empirically that having one is helpful.
4) **You want to use the version of the example code labelled Listing 3.7**. This is the code that has both the replay buffer and the target network DQN extensions.
5) You will need to experiment with the size of the net (depth and width). The size of the first layer will need to match the size of the input (4 * 16 + 4 + 4 = 72) and the size of the output should be 6 to produce a Q value over the 6 possible actions. One architecture that seems to work is 72/512/128/128/128/6.
6) Use an epsilon-greedy policy to select the action during training.
7) If the agent ever finds itself in a state it has already been in during the same episode, it is in a loop in the state graph and should be terminated with the usual -1,000 penalty.

[1] or equivalent representations that can be mapped to these features for input to the neural net
[2] Remember that W in the code means wall, not Wumpus.

8) Be careful to be sure that your environment is working correctly so that the training episode data are correct.
9) When your agent has been trained, run it for 1,000 episodes and add up its total score.

Extensions for the Project

If you would like to do an extension of Assignment 4 as your project, here are some ideas:

1) Add a 3x3 convolutional input and train the net to play on grids that are other than 4x4.
2) Try different depths/widths of neural nets and compare their performance.
3) Reduce the agent state so that it doesn't know what its location or orientation is but does know its history of moves and when it received a Bump. (This one is much more difficult to train and may require an RNN to provide some short-term memory rather than making the state space a lot larger and to deal with the fact that the list of percepts will vary in length from episode to episode).
4) Similar to 3, remove the feature that keeps track of where the agent has previously been but again provide the history of moves and when it received a Bump. Also challenging.
5) Use your ProbAgent as a Critic to train an Actor-Critic agent. Compare the training time to the DQNAgent's.
6) Try removing the feature that keeps track of whether the agent has the arrow available or not and see how much it degrades the agent's performance.
7) Try implementing a different algorithm such as a Policy Gradient method.