Spencer Doyle

COMP560 HW2

Spring 2020

Implementation

My program (Java) takes in the test file from the standard input (should be piped in from an input file to stop the Scanner). It then creates an "Environment" which includes object classes representing States, Actions, and Transitions. I began with the model-free approach to active learning. I decided to implement a Q-Learning function which seemed like a nice, simplistic way to determine an optimal policy without updating an actual model. My program creates a Q-value and Reward value for each "Action" that is contained within a "State." It then picks a random state and an action, determined by the max Q-value for all of its possible actions or a random action, all depending on the epsilon value. The algorithm continues running and updating state/action Q-values until the goal state is reached, in which it picks a new random state and continues the same algorithm for a certain number of total iterations or episodes. The Q-values are then updated by the Bellman-derived update function:

$$Q^{new}(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\Big(\underbrace{\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}}}_{\text{new value (temporal difference target)}} - \underbrace{Q(s_t, a_t)}_{\text{old value}} \Big)}^{\text{temporal difference}}$$

Ultimately the algorithm terminates after the desired amount of episodes and the State/Action/Q-values are outputted to the command line. The optimal policy is then easily inferred from the highest Q value associated with a given state.

I then moved on to my model-based approach which uses a clever algorithm called Dyna-Q. This algorithm is almost identical to the Q-Learning one, except for two key differences. First,

the model gets updated in this algorithm. The transition probabilities are updated by incrementing an observance of that particular transition, and also incrementing the action that it resulted from. The reward for that given state/action is then updated by the function:

$$R'[s,a] = (1-\alpha)R[s,a] + \alpha \cdot r$$

The second difference is that this algorithm exploits the learned model with a “cheap” inner loop that updates n Q-values(often in the hundreds) by picking n random states and actions and inferring their resulting states from the Transition model and reward values from the Reward Model. It then similarly updates these Q-values. After this algorithm terminates the state/action/state/probabilities are printed and so are the state/action/Q-values which we can use to infer the optimal policy.

Questions

Both my model based and model free implementations use the epsilon value in the same way. Basically, I have a value between 0 and 1, which is the probability of us exploring. My program will only allow this “possibility” of exploring to last until halfway through the episodes. To do this my epsilon value is decremented by epsilon / (½ * #episodes). In the Q-learning approach I determined a value of .8 was ideal. Smaller than this greatly increased the chances of a different resulting policy every time. I slowly worked by way towards .8 until this became fairly stable. For the Dyna-Q approach, a .8 epsilon value seemed a little too high, because we have an inner loop that is trying to basically exploit what we have previously learned in the model. I eventually found a fairly stable value of .5 which made sure the outer loop didn’t explore too much, in return skewing the work of the inner loop.

Considering the small nature of this MDP, it was possible to continue my Q-learning algorithm for upwards of 10 million episodes; however, it is clear that we want to reduce this as much as possible to promote scalability and more effective RL. I slowly started decreasing the number of episodes by factors of 10 until I dropped below 10 thousand, which then also began producing less consistent results. The Dyna-Q algorithm is slightly more tricky because we set a number of total episodes, but also a number of "fake" episodes for the inside loop to run. I call these "fake" because it is not really acting in the environment, just simply inferring some more Q-values based on what we have learned. Ultimately we have to consider that the algorithm will run approximately (#episodes + #episodes*#fake_episodes) times. First I set the number of fake episodes at 200 and the outer episodes at 10,000. While playing with these numbers I decided to actually wait for 1000 outer episodes until I began exploiting with fake episodes. That way there is a good chance we have seen all of the state/action combinations several times before exploiting the learned model. I ultimately settled on 50 inner episodes and 8000 outer episodes so that the majority (7000) of episodes still ran inner fake episodes. 50 was chosen as to not infer from our model too much. Ultimately, these values gave me fairly consistent policies. However, an inconsistent policy still results on occasion with these values and is not perfect.

The discount factor in both functions had a similar effect. Inconsistent results began emerging for discount rates much less than .9. Higher than .9 made my results too close to converging and hard to easily read(based on glances at the Q-values). For the Dyna-Q algorithm I actually played around with two separate discount values: one for the outer loop and one for the inner loop (since both update Q-values with the same equation). I kept the outer value at .9, but decided to bump it down to .8 for the inner loop, which helped keep the policy more consistent each time it ran. I speculate that this helped keep this exploiting inner function less affected by

the earlier (and possibly less valuable) reward values that were inferred from the model which was likely less accurate at the beginning.