Reinforcement Learning in Pacman

Sai Srinadhu Katta

1 Introduction

In this project, I implemented value iteration agent, Q-learning agent, approximate Q-learning agent in Pacman environment. Tuned the values of noise, discount and living reward to produce the given optimal policy in a given environment. I implemented epsilon greedy action selection for Q-Learning agent as well.

2 MDP and RL Techniques

2.1 Value Iteration

In this part batch version of Value Iteration is implemented. The values of states in previous iteration is used for updating values of states in next iteration which might not always be the case with online update. The getStates() function returns all the states in the environment. The function getPossibleActions(state) returns the possible actions from state. The function getQValue(state, action) returns the QValue. In the self.values indexed by state the values of states are stored. A point to note in this is that a policy synthesized from values of depth k will actually reflect the next k+1 rewards which is the case here.

```
def __init__(self, mdp, discount = 0.9, iterations = 100):

        self.mdp = mdp
        self.discount = discount
        self.iterations = iterations
        self.values = util.Counter() # A Counter is a dict with default 0

        # Write value iteration code here
        for state in self.mdp.getStates(): #get all the states
            self.values[state] = 0.0

        for i in range(self.iterations):   #run for these many iterations
            next_values = self.values.copy() #copy back the old values
            for state in self.mdp.getStates():
                state_values = util.Counter() #values for actions of this ↩
    state
                for action in self.mdp.getPossibleActions(state):
                    state_values[action] = self.getQValue(state, action)
                next_values[state] = state_values[state_values.argMax()]  #↩
    update for each state
```

```
19                 self.values = next_values.copy() #copy back the new values
```

Code Snippet for Value Iteration Agent.

2.2 Bridge Crossing Analysis

The default values for discount and noise are 0.9 and 0.2 respectively. My task is to change only one of them so that the optimal policy tries to cross the bridge. The main problem here is the noise and I kept reducing it by half till it succeeded or it could just have been made 0 and would have worked out as well.

```
1 def question2():
2     answerDiscount = 0.9
3     answerNoise = 0.00125
4     return answerDiscount, answerNoise
```

Code Snippet for Bridge Crossing Analysis.

2.3 Policies

In the DiscountGrid layout following optimal policy types are produced by tuning discount, noise and living reward:

```
1 def question3a():
2     answerDiscount = 0.2
3     answerNoise = 0.0
4     answerLivingReward = 0.0
5     return answerDiscount, answerNoise, answerLivingReward
```

Prefer the close exit, risking the cliff.

The idea here is that it should prefer close exit and so discount is set very less to 0.2 and the noise is also set to 0.0 for this.

```
1 def question3b():
2     answerDiscount = 0.2
3     answerNoise = 0.2
4     answerLivingReward = 0.0
5     return answerDiscount, answerNoise, answerLivingReward
```

Prefer the close exit but avoiding the cliff.

The idea here is that it should prefer close exit and so discount is set very less to 0.2 and the noise is also set to 0.2 for the reason that it shouldn't risk the cliff.

```
def question3c():
    answerDiscount = 0.9
    answerNoise = 0.0
    answerLivingReward = 0.0
    return answerDiscount, answerNoise, answerLivingReward
```

Prefer the distant exit, risking the cliff.

The idea is to increase discount or keep it high and so it's set to 0.9 our usual default value.

```
def question3d():
    answerDiscount = 0.9
    answerNoise = 0.2
    answerLivingReward = 0.0
    return answerDiscount, answerNoise, answerLivingReward
```

Prefer the distant exit, avoiding the cliff.

The idea is to increase noise from above the above part and it's set to 0.2

```
def question3e():
    answerDiscount = 0.9
    answerNoise = 0.2
    answerLivingReward = 1.0
    return answerDiscount, answerNoise, answerLivingReward
```

Avoid both exits and the cliff (so an episode should never terminate)

With the usual default values for discount and noise, set the living reward positive to 1, to avoid both exits and cliff.

2.4 Q-Learning

In this part Q-Learning is implemented. In real life the underlying MDP may not be always known and so we need to learn from experience. The getQValue function returns the QValue. The getValue returns the value of state. In self.QValues, QValues are stored indexed by state and action. From these QValues policy is constructed.

```
    def update(self, state, action, nextState, reward):
        newQValue = (1 - self.alpha) * self.getQValue(state, action)
        newQValue += self.alpha * (reward + (self.discount * self.getValue(↩
    nextState)))
        self.QValues[state, action] = newQValue
```

Update function of Q-Learning Agent.

2.5 Epsilon Greedy

The main idea here is that exploration of state space has to be done first and after getting good policy, exploitation of that policy has to done. With epsilon probability choose a random action from a state otherwise choose the present optimal policy. This method is also called epsilon-greedy action selection.

```
def getAction(self, state):
    """
    Compute the action to take in the current state.  With
    probability self.epsilon, we should take a random action and
    take the best policy action otherwise.  Note that if there are
    no legal actions, which is the case at the terminal state, you
    should choose None as the action.
    """

    legalActions = self.getLegalActions(state)
    action = None

    if (util.flipCoin(self.epsilon)):
        action = random.choice(legalActions)
    else:
        action = self.getPolicy(state)

    return action
```

Action function of Q-Learning Agent.

The crawler was also run for this part, the observation is that after decent number of iterations trying how to walk and after it has learned, epsilon should be lowered from initial value since now it should exploit the learnt one rather than to keep trying.

2.6 Bridge Crossing Revisited

In this within 50 iterations of QValue Iteration the optimal policy should cross the bridge for the values of epsilon and learning rate set with very high probability, it's simply not possible.

```
def question6():
    answerEpsilon = 0.0
    answerLearningRate = 0.5
    return 'NOT POSSIBLE'
```

Bridge Crossing Analysis.

2.7 Q-Learning and Pacman

In this part first reinforcement learning agent is trained for 2000 episodes and then tested on 100 games, my implementation won all the 100 games. This method doesn't scale up for bigger environments because of amount of QValues to be stored and amount of training that will be required to visit most of state space decent number of time.

2.8 Approximate Q-Learning

In this part implemented approximate Q-learning agent. self.weights contain the weights indexed by feature. First the agent is trained and then tested. This solves the major problem of Q-learning in larger state spaces where it's simply not feasible. The idea here is that QValue is represented as weighted linear sum of feature values for a state, now we need a way to get the feature value of state and way of updating weights based on what the agent sees, which is precisely what we are doing here.

```
1
2  def update(self, state, action, nextState, reward):
3      """
4      Should update your weights based on transition
5      """
6      QValue = 0
7      difference = reward + (self.discount * self.getValue(nextState) - self.↩
       getQValue(state, action))
8      features = self.featExtractor.getFeatures(state, action)
9
10     for feature in features:
11       self.weights[feature] += self.alpha * features[feature] * difference
```

Update function for Approximate Q-learning agent.

After taking a action from a state we go to new state and get a reward, based on features, reward and next state we are updating the weights. Number of iterations this takes for learning to win a game compared to normal Q-learning is less. In larger state spaces this is practically more useful than the Q-Learning agent. Finally we have a learning pacman agent.

References

[1] UC Berkeley CS 188 Intro to AI – Course Materials,
`http://ai.berkeley.edu/reinforcement.html`

[2] LaTeX Templates for Laboratory Reports,
`https://github.com/mgius/calpoly_csc300_templates`