Machine, Data and Learning

Assignment 2 : Part 2

Team Number : 69
Anishka Sachdeva (2018101112)
Satyam Viksit Pansari (2018101088)

Code the **Value Iteration Algorithm** for the given scenario to obtain the optimal policy and the state reward values corresponding to it.

Consider the following state encoding scheme

MD's Health : [0,25,50,75,100] -> [0,1,2,3,4]

Number Of Arrows : [0, 1, 2, 3] -> [0,1,2,3]

Stamina : [0,50,100] -> [0,1,2]

Total number of states = 5 x 4 x 3 = 60

Initial State Utilities = 0

Value Iteration Algorithm:

Value iteration starts at the "end" and then works backward, refining an estimate of either Q^* or V^*. There is really no end, so it uses an arbitrary end point. Let V_k be the value function assuming there are k stages to go, and let Q_k be the Q-function assuming there are k stages to go. These can be defined recursively. Value iteration starts with an arbitrary function V_0 and uses the following equations to get the functions for $k+1$ stages to go from the functions for k stages to go:

$$\mathbf{V[s]=max_a \sum_{s'} P(s'|s,a)\ (R(s,a,s')+ \gamma V[s']).}$$

P(s'|s,a) is probability of going state s' from state s after taking action a, $\sum_{s'}$ is the sum for all possible states s' for which P(s'|s,a) is non zero, R(s,a,s') is the reward gained on reaching state s' from state s on taking action a. $V_k(s)$ is the utility value of state s after k iterations and γ is the discount factor.

Python libraries/modules used:

- **NumPy** : NumPy is a general-purpose array-processing package. It provides a high-performance multidimensional array object, and tools for working with these arrays.

  Functions Of Numpy Library Used :

  - **numpy.zeros(shape,dtype=None) :** Return a new array of given shape and type,with zeroes.

    Parameters :
    shape : integer or sequence of integers
    dtype : [optional, float(byDefault)] Data type of returned array.

    Returns : ndarray of zeros having given shape, order and datatype.

- **Os :** The OS module in python provides functions for interacting with the operating system. OS, comes under Python's standard utility modules. This module provides a portable way of using operating system dependent functionality.
  In the code , we have used it to create the directory structure for both the tasks.

Task 1:

- **Parameters:**
  1. **Penalty(Step Cost)** = -20 for non absorbing states and 10 (-20 + 10) for absorbing states.
  2. **Gamma (Discount Factor)** = 0.99
  3. **Delta (Convergence or Bellman error)** = 0.001

- **Observations:**
  1. **Total Number of Iterations** : 125
  2. **Inference :**
     - When it has low arrows, it prefers to dodge.

- It only recharges when it has zero stamina (except when enemy has full health, where it prefers to shoot).
- When it has all three arrows, it prefers to shoot almost always, even if it has low stamina.
- When it has low arrows and enemy has very high health, it prefers to dodge more often than usual.

Task 2:

Part 1:

- **Parameters:**
  1. **Penalty(Step Cost)[Dodge,Recharge]** = -2.5
  2. **Penalty(Step Cost)[Shoot] =** -0.25
  3. **Gamma (Discount Factor)** = 0.99
  4. **Delta (Convergence or Bellman error)** = 0.001

- **Observations:**
  1. **Total Number of Iterations** : 99
  2. **Inference** :

     - The agent consistently chooses to shoot instead of recharge (as was in task 1) whenever its stamina is at 50%. This is the primary difference in the policy observed from task 1 on reducing the step cost for shoot i.e. on reducing the step cost, the Value iteration algorithm converges in lesser iterations as compared to task 1. This was expected since now the cost for doing a shoot is lower.

Part 2:

- **Parameters:**
    1. **Penalty(Step Cost)[All Actions]** = -2.5
    2. **Gamma (Discount Factor)** = 0.1
    3. **Delta (Convergence or Bellman error)** = 0.001

- **Observations:**
    1. **Total Number of Iterations** : 4
    2. **Inference** :
        - Here we decreased gamma value significantly. Increasing gamma value relates to increasing rememberance of the history of the agent (i.e., the agent values its history significantly.) Thus, in this case, agent will start to forget its previous iterations.

        - We notice that utilities for all the actions are the same in part 2. Therefore, the policy of the agent at particular instances are random. This results in non-sensical values for certain states. For example, at state (3, 0, 2) the agent prefers recharge, even though it does not help kill the dragon in any way. It was equally possible that the action the agent chose was to dodge, since the utilities for all of them came out to be same. This is the weird behavior being asked in the question.

        - Since discount factor is very low, the agent is short-sighted (myopic), so if he does not see an immediate win (i.e., health of dragon in current state = 1), then he will opt for any random action, and he is not really affected by it in any way.

        - Due to the low value of gamma, the player tries to complete the game as quickly as possible and there is huge difference between the number of iterations in part-I and this part. This is evident in the number of iterations which is only 4.

Part 3:

- **Parameters:**
    1. **Penalty (Step Cost)[All Actions]** = -2.5
    2. **Gamma (Discount Factor)** = 0.1
    3. **Delta (Convergence or Bellman error)** = 10^ (-10)

- **Observations:**
    1. **Total Number of Iterations** : 11
    2. **Inference** :
        - This takes higher time to converge as compared to part 2, because the delta value was reduced by a significant factor.

        - The error in part 2 (of non-sensical actions being printed) in certain states gets fixed in part 3, due to more iterations, the agent is able to differentiate between utility values and produce a more accurate result.