PACMAN ASSIGNMENT  
Reinforcement Learning

**\_\_\_**

By Vasanti Mahajan & Ekta Chavan



# INTRODUCTION

This assignment aims to implement Value Iteration and Q-Learning on Gridworld and further implement Approximate Q-Learning on Pacman Game.

## Question 1: Value Iteration

**Problem:** To write a Value Iteration Agent that computes k-step estimates of the optimal values, Vk.

**About Value Iteration:** If we know the true value for each state, we would always choose the action which results in best reward. However, we just have the immediate reward for every state, thus it is possible that initially the actions yield low reward, but it is on the correct path of actions and would eventually lead to high rewards. Thus we require a value iteration function to determine the true value of a state:



The qvalue for each state is computed by summing the reward that we receive and maximizing the qvalue of the next states.

Solution:

Part 1: Value Iteration

1. First we set a counter to store all the values
2. Then we get all the possible MDP states and their corresponding actions
3. We iterate over the actions and get their Q-Values
4. We then use the Max function to get the best Q-value for the given set of State and Action pair
5. Finally we update the policy with the best possible Q-values for actions and state pairs

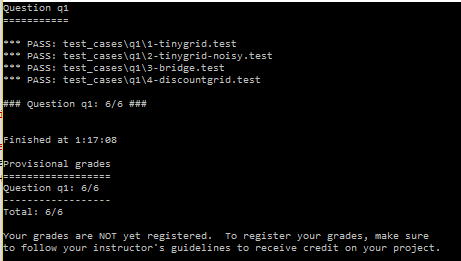
Part 1.1: ComputeActionFromQueueValues():

1. Get all Legal Actions for a given MDP state
2. Initialize a counter to hold the values
3. Iterate over all the legal actions and compute the Q-Value
4. Return the best possible Action

Part 1.2 : computeQValueFromValues():

1. Get the transition function and the Next State s’
2. Initialize the Value to zero
3. Iterate over Transition Functions Probability and next states
4. Compute Q-Value

Result: 6/6



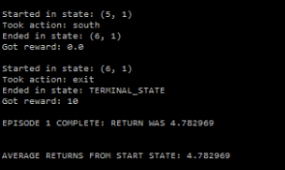
## Question 2: Bridge Cross Analysis



A BridgeGrid moves from a low-reward Terminal State and a high-reward Terminal State separated by a narrow “bridge” on either side of which is a chasm of high negative reward.

The problem is to change **ONLY ONE** of the parameters amongst ‘discount’ and ‘noise’ so that the policy causes the agent to cross the bridge.

**Solution**: We tried a variety of combinations of discount and noise. The perfect score of 10 was achieved with a combination of discount 0.9 and noise 0.75. However if we tried to re-run it with the same values it would sometimes fail and sometimes pass.

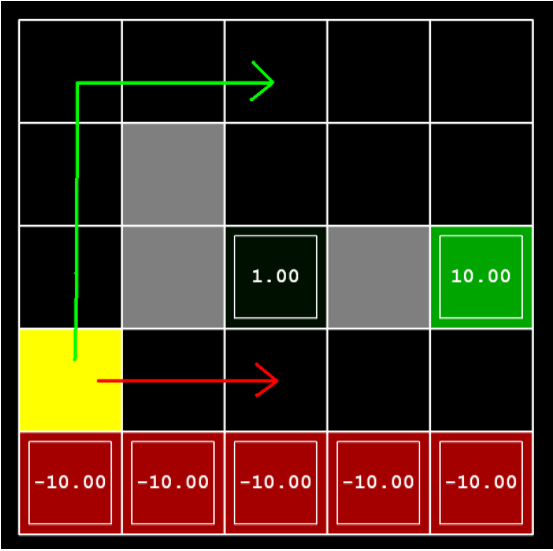


By understanding what noise means, we learnt that noise is the probability of random movement of the player. We need to player to only move in one direction to stay on the bridge. Setting a value of 0.0 for noise gives us a perfect score 10 for each run.

**Discount = 0.9, Noise = 0.0**

Result: We get a Return of 5.9

## Question 3: Policies



To choose settings of parameters : discount, noise and living reward for the given MDP to produce optimal policies of several different types.

1. Prefer the close exit (+1), risking the cliff (-10)

Solution:

answerDiscount = 0.9

answerNoise = 0.2

answerLivingReward = -2

b) Prefer the close exit (+1), but avoiding the cliff (-10)

Solution:

answerDiscount = 0.5

answerNoise = 0.4

answerLivingReward = -0.5

c) Prefer the distant exit (+10), risking the cliff (-10)

Solution:

answerDiscount = 0.9

answerNoise = 0

answerLivingReward = -3

d) Prefer the distant exit (+10), avoiding the cliff (-10)

Solution:

answerDiscount = 0.8

answerNoise = 0.8

answerLivingReward = -0.4

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

Solution:

answerDiscount = 0.5

answerNoise = 0.5

answerLivingReward = -0.8

Result: 4/5

## Question 4: Q-Learning

Value iteration does not learn from experience it just uses the MDP to identify the suitable policy before interacting with the environment. We can use a Q-Learning function which learns from its interactions with the environment by trial and error through its update method.



Solution:

Part 4.1: Setup a util.Counter to store our Q-Values

Part 4.2: getQValue()

1. Fetch Q-Value for existing State Value Pairs

Part 4.3: computeValueFromQValues()

1. Get all legal actions
2. Return 0 if there are no actions from this state
3. Iterate over actions and add all the Q-Values from any action from this state
4. Return the highest value by using the *max* function

Part 4.4: computeActionFromQValues()

1. Get all the Legal Actions for a given state
2. Return None if there are no actions from this state
3. Iterate over actions and append all the Q-Values and actions to an array consisting of all the actions
4. Get all the Q-Value, action pairs that are the best
5. Randomly choose a Q-Value, action pair if two or more are the best
6. Return the best action

Part 4.5: update()

1. Calculate Q(s,a) i.e. Q-Value for current State and Action
2. Get the sample by adding current reward to future discounted reward multiplied by Next State’s Q-Value
3. Update the Q-Value and add it to our Q-Value Counter

Result: 5/5 tests pass

## Question 5: Epsilon Greedy

Complete the Q-Agent by implementing Epsilon Greedy Action Method in the *getAction*() method.

Solution:

Step 5.1: Get all Actions for a given state

Step 5.2: Possibly pick a random action with probability epsilon

Step 5.3: Else return the best action

Result: 3/3 Test cases passed

QLearning Crawler works.

## Question 6: Bridge Crossing Revisited

First, train a completely random Q-learner with the default learning rate on the noiseless BridgeGrid for 50 episodes and observe whether it finds the optimal policy.Now try the same experiment with an epsilon of 0.

Result: Test passed

## Question 7: Q-Learning and Pacman

To check how Pacman behaves with a Q-Learning Agent by first training itself on 2000 in noiseless mode and then testing on the last 10 games.

Result: EPISODE 50 COMPLETE: RETURN WAS 0.9

AVERAGE RETURNS FROM START STATE: -60.763482



**Question 8: Approximate Q-Learning**

Implement an approximate Q-learning agent that learns weights for features of states, where many states might share the same features.

Solution: qLearningAgents.py→ ApproximateQAgent class

Part 8.1: getQValue()

1. Get features using the Identity extractor
2. Get weights
3. Compute dot product or features and weights

Part 8.2 : Update Weights

1. Calculate Difference as: differece = reward + gamma\*Q(s', a') - Q(s,a)
2. If weight vector is empty, initialize it to zero
3. Iterate over features and multiply them by the learning rate (alpha) and the difference
4. Sum the weights to their corresponding newly scaled features
5. Update weights

Result: For the Training, Approximate Q-Learning Agent performs gradually better as it completes 2000 episodes and goes from negative reward value to final 100 episodes reward alue of 197.08.

Once it has learnt, the approximate learning agent gets a perfect 10/10 with custom feature extractor in a medium sized Grid with a training of just 50 episodes.

In the large sized grid, it wins 9/10 times.