

# Introduction to Artificial Intelligence HW3 Report

110550088 李杰穎

May 1, 2022

## 1 Adversarial Search

### 1.1 Implementation of Minimax Algorithm

Minimax algorithm is a searching algorithms used in many game AI. It wants to find a specific action to maximize the minimum possible score.

Below is the code that implement the minimax algorithm in the pacman game.

Code 1: `class MinimaxAgent`

```
1 # I have removed the original comment in class.
2 class MinimaxAgent(MultiAgentSearchAgent):
3     def getAction(self, gameState):
4         *** YOUR CODE HERE ***
5         # Begin your code
6         actions = gameState.getLegalActions(0) # Get Legal Action of pacman (pacman's
7             ↪ index is 0)
8         candidates = [] # Initialize a list to track legal action and its score for the
9             ↪ first max layer.
10        for action in actions: # Iterate all possible action
```

```

9         candidates.append((action, self.minimax(gameState.getNextState(0, action),
10             ↪ self.depth-1, 1, False))) # Call recursive function self.minimax
11     action, _ = max(candidates, key=lambda item: item[1][1]) # Get the action with
12     ↪ the highest score
13     return action # return that action
14     # End your code
15 def minimax(self, gameState, depth, agentIdx, maximize):
16     if gameState.isWin() or gameState.isLose() or (depth == 0 and agentIdx == 0): #
17     ↪ If current game state is terminal state
18         return (gameState, self.evaluationFunction(gameState)) # return (state,
19         ↪ score) pair
20     actions = gameState.getLegalActions(agentIdx) # Get Legal action of a character
21     ↪ (pacman or ghosts)
22     candidates = [] # Initialize a list to track legal action and its corresponding
23     ↪ score
24     if maximize: # If current layer is a max layer
25         # Because current layer is a max layer, the next layer will be a min layer
26         ↪ with the first ghost whose index equals to 1.
27         for action in actions:
28             candidates.append(self.minimax(gameState.getNextState(agentIdx,
29                 ↪ action), depth-1, 1, False))
30             stateScore = max(candidates, key=lambda item: item[1]) # Max Layer, take
31             ↪ max over the candidates' score
32
33     elif agentIdx < gameState.getNumAgents()-1: # If current layer is a min layer,
34     ↪ and current ghost is not the last ghosts
35         # Because current layer is a min layer and current ghost is not the last
36         ↪ ghost, the next layer will still be a min layer with a ghost whose index
37         ↪ is the current index + 1
38         for action in actions:
39             candidates.append(self.minimax(gameState.getNextState(agentIdx,
40                 ↪ action), depth, agentIdx+1, False))
41             stateScore = min(candidates, key=lambda item: item[1]) # Min Layer, take
42             ↪ min over the candidates' score
43     else: # If current layer is a min layer, and current ghost is the last ghost
44         # Current ghost is the last ghost, the next layer will be a max layer with the
45         ↪ pacman, whose index equals to 1
46         for action in actions:
47             candidates.append(self.minimax(gameState.getNextState(agentIdx,
48                 ↪ action), depth, 0, True))

```

```

33         stateScore = min(candidates, key=lambda item: item[1]) # Min Layer, take
           ↪ min over the candidates' score
34
35     return stateScore # Return (state, score) pair

```

## 1.2 Implementation of Expectimax Algorithm

Instead of taking the minimum value in the min layer. Expectimax algorithm take the expectation value of all possible values. In the pacman game, every action taken by the ghosts is same, therefore, the expectation value of all actions equals to the average all of possible score.

The code below implement the expectimax algorithm for the pacman game.

Code 2: class ExpectimaxAgent

```

1 class ExpectimaxAgent(MultiAgentSearchAgent):
2     # The code in expectimax is almost same as in minimax algorithms, therefore I will
           ↪ only explain the different part, that is, the calculatoin of value in min
           ↪ layer.
3     def getAction(self, gameState):
4         actions = gameState.getLegalActions(0)
5         candidates = []
6         for action in actions:
7             candidates.append((action, self.expectimax(gameState.getNextState(0,
           ↪ action), self.depth-1, 1, False)))
8         action, _ = max(candidates, key=lambda item: item[1])
9         return action
10    # End your code
11    def expectimax(self, gameState, depth, agentIdx, maximize):
12        if gameState.isWin() or gameState.isLose() or (depth == 0 and agentIdx == 0):
13            return self.evaluationFunction(gameState)
14        actions = gameState.getLegalActions(agentIdx)
15        candidates = []
16        if maximize:
17            for action in actions:

```

```

18         candidates.append(self.expectimax(gameState.getNextState(agentIdx,
    ↪ action), depth-1, 1, False))
19     score = max(candidates)
20     elif agentIdx < gameState.getNumAgents()-1:
21         # Instead of taking the min value over the candidates, in expectimax
    ↪ algorithm, we will take the average instead.
22         tmp = 0 # Initailze the variable to sum all possible score.
23         for action in actions:
24             tmp += (self.expectimax(gameState.getNextState(agentIdx, action),
    ↪ depth, agentIdx+1, False))
25         score = tmp / len(actions) # Take average.
26     else:
27         # Same as above
28         tmp = 0
29         for action in actions:
30             tmp += (self.expectimax(gameState.getNextState(agentIdx, action),
    ↪ depth, 0, True))
31         score = tmp / len(actions)
32
33     return score

```

### 1.3 Comparisons of Minimax and Expectimax Algorithms

### 1.4 Better Evaluation Function

In the terminal state of pacman, we need a evaluation function to evaluation the score of current terminal state. This score will later be used to decide which action we should take. Therefore, the design of evaluation function is important to both expectimax and minimax algorithm.

But in the original code, the evaluation function just return the score of the terminal state. This is not sufficient to achieve high score and also win the game every time.

Therefore, I decide to design a better evaluation function to evaluate the current state.

After some experiments, I have designed a great evaluation function that achieve 100% win rate in 100 games, and the average score of these 100 games is 1155.57.

Below is the implementation of betterEvaluationFunction.

Code 3: betterEvaluationFunction

```
1 def betterEvaluationFunction(currentGameState):
2     """
3     Your extreme ghost-hunting, pellet-nabbing, food-gobbling, unstoppable
4     evaluation function (part1-3).
5
6     DESCRIPTION: <write something here so we know what you did>
7     """
8     """ YOUR CODE HERE """
9     # Begin your code
10    # If the current game state is a loss state, then return a small value to avoid
11    ↪ this game state happened
12    # This make the game very hard to lose
13    if currentGameState.isLose():
14        return -1e20
15    # Otherwise, if this game state is a win state, then return a large value
16    elif currentGameState.isWin():
17        return 1e20
18
19    score = currentGameState.getScore() # get current score of the game
20    cnt_food = currentGameState.getNumFood() # get the number of remaining food
21    cnt_cap = len(currentGameState.getCapsules()) # get the number of remaining
22    ↪ capsules
23    dis = closestFood(currentGameState.getPacmanPosition(), currentGameState.getFood(),
24    ↪ currentGameState.getWalls()) # call function closestFood to get the distance of
25    ↪ the closest food
26    val = 1 * score # Initialize the return value with 1 * score
27    if dis is not None: # If dis is not None
28        val -= 10 * dis # Because we want to minimize the distance between pacman and
29        ↪ food. Therefore, if the distance is smaller, then the return value will be
30        ↪ higher.
31    val -= cnt_food * 100 # Same as closest food. We also want to minimize the number
32    ↪ of food. In this way, the pacman will eat food instead of stay in one place.
33    ↪ Therefore, fewer food will have higher value.
```

```

27     val -= 30 * cnt_cap # Same as food, but the weight is slightly different
28     return val # return the final value
29     # End your code

```

Code 4: closestFood

```

1 def closestFood(pos, food, walls):
2     """
3     This function is actually from the q-learning part.
4     It uses BFS to find the closest food.
5     """
6     fringe = [(pos[0], pos[1], 0)] # Initialize a list. This list will be later used as
    ↪ a queue.
7     expanded = set() # Initialize a set to track the visited positions
8     while fringe: # While queue is not empty
9         pos_x, pos_y, dist = fringe.pop(0) # Pop the first element in queue
10        if (pos_x, pos_y) in expanded: # If this position has already visited
11            continue
12        expanded.add((pos_x, pos_y)) # Add current position to visited set
13        # if we find a food at this location then exit
14        if food[pos_x][pos_y]:
15            return dist
16        # otherwise spread out from the location to its neighbours
17        nbrs = Actions.getLegalNeighbors((pos_x, pos_y), walls)
18        for nbr_x, nbr_y in nbrs:
19            fringe.append((nbr_x, nbr_y, dist+1))
20    # no food found
21    return None

```

## 2 Q-learning

### 2.1 Value Iteration

Value iteration is an algorithm that can find the best policy under the assumption of Markov decision process (MDP).

The update equation of a state is written as:

$$V_{\text{opt}}^{(t)}(s) \leftarrow \max_{a \in \text{Actions}(s)} \sum_{s'} T(s, a, s') \left( \text{Reward}(s, a, s') + \gamma V_{\text{opt}}^{(t-1)}(s') \right) \quad (1)$$

, where  $V_{\text{opt}}^{(t)}(s)$  is the optimal value for the time  $t$  with given game state  $s$ .  $T(s, a, s')$  is the transition probability to game state  $s'$  if agent is in game state  $s$  and take action  $a$ .  $\text{Reward}(s, a, s')$  is the reward with given current game state  $s$ , action  $a$  and the next game state  $s'$ .  $\gamma$  is the discount factor.

After taking  $t$  times of update, the optimal policy  $\pi_{\text{opt}}(s)$  with given state  $s$  can be expressed as:

$$\pi_{\text{opt}}(s) = \arg \max_{a \in \text{Actions}(s)} Q_{\text{opt}}(s, a) \quad (2)$$

And  $Q_{\text{opt}}(s, a)$  is defined as:

$$Q_{\text{opt}}(s, a) = \sum_{s'} T(s, a, s') \left( \text{Reward}(s, a, s') + \gamma V_{\text{opt}}(s') \right) \quad (3)$$

Below is the code of the implementation of **ValueIterationAgent**. I have removed functions that are already completed for the simplicity of report.

Code 5: `class ValueIterationAgent`

```

1  class ValueIterationAgent(ValueEstimationAgent):
2      """
3          * Please read LearningAgents.py before reading this.*
4
5          A ValueIterationAgent takes a Markov decision process
6          (see mdp.py) on initialization and runs value iteration
7          for a given number of iterations using the supplied
8          discount factor.
9      """
10
11     def runValueIteration(self):
12         # Write value iteration code here
13         """** YOUR CODE HERE **"""
14         # Begin your code
15         for _ in range(1, self.iterations+1): #Run self.iteration times of value
16             ↪ iteration algorithm
17             previous_value = self.values.copy() # Copy old self.values to avoid
18             ↪ overwriting problem when updating the current self.values

```

```

17         for state in self.mdp.getStates():
18             if self.mdp.isTerminal(state): # If current state is terminal state,
19                 ↪ then set its value to 0
20                 self.values[state] = 0
21                 continue
22             maxi_value = -1e9 # Initialize maxi_value to a small value, this
23                 ↪ variable will track the value of all possible actions
24             for action in self.mdp.getPossibleActions(state): # Iterate all
25                 ↪ possible action
26                 sumOfAllState = 0
27                 for (nextState, prob) in
28                     ↪ self.mdp.getTransitionStatesAndProbs(state, action): # Get all
29                     ↪ the possible state and their corresponding probability
30                     # Use the main formula of value iteration method to update
31                     ↪ self.values
32                     # Noticed that I use previous_value to compute the correct
33                     ↪ update value
34                     sumOfAllState += prob * (self.mdp.getReward(state, action,
35                     ↪ nextState) + self.discount*previous_value[nextState])
36                 maxi_value = max(maxi_value, sumOfAllState) # Taking max over the
37                     ↪ corresponding value of all possible state
38             self.values[state] = maxi_value # set self.values to the maximum
39                 ↪ possible value
40
41         # End your code
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99

```

```

def getValue(self, state):
    """
    Return the value of the state (computed in __init__).
    """
    return self.values[state]

def computeQValueFromValues(self, state, action):
    """
    Compute the Q-value of action in state from the
    value function stored in self.values.
    """
    """** YOUR CODE HERE **"""
    # Begin your code

```



```

48     res = 0 # Initialize q value
49     for (nextState, prob) in self.mdp.getTransitionStatesAndProbs(state, action): #
        ↳ Get all teh possible state and their correspodng probability
50         res += prob * (self.mdp.getReward(state, action, nextState) +
        ↳ self.discount*self.values[nextState]) # Compute q-value using formula
51     return res # return q value
52     # End your code
53
54 def computeActionFromValues(self, state):
55     """
56     The policy is the best action in the given state
57     according to the values currently stored in self.values.
58
59     You may break ties any way you see fit. Note that if
60     there are no legal actions, which is the case at the
61     terminal state, you should return None.
62     """
63     """** YOUR CODE HERE **"""
64     # Begin your code
65     #check for terminal
66     if self.mdp.isTerminal(state): # If this state is a terminal state, then agents
        ↳ can't move. Therefore, return None
67         return None
68     actions = self.mdp.getPossibleActions(state) # Otherwise, get all possible
        ↳ actions
69     qValues = util.Counter() # Initailze a Counter to track every q value after
        ↳ taking action
70     for action in actions:
71         qValues[action] = self.getQValue(state, action) # Using getQValue to get
        ↳ q-value after taking this action
72
73     return qValues.argmax() # argMax will return the key (which is action in this
        ↳ function) that has the highest q-value
74
75     # End your code

```

## 2.2 Q-learning

Value iteration algorithm is used in MDP, where we can get  $T(s, a, s')$ . But in most of the game including Pacman, there are too many game states thus it's impossible to get accurate  $T(s, a, s')$  for every pair of  $(s, a, s')$ .

Therefore, we need to use q-learning instead. Q-learning is a model-free algorithm to learn the value of an action  $a$  in a given state  $s$ . The q-values are updated for each  $(s, a, r, s')$ , and the update process can be represented as below:

$$\hat{Q}_{\text{opt}}(s, a) \leftarrow (1 - \eta)\hat{Q}_{\text{opt}}(s, a) + \eta \left( r + \gamma \max_{a' \in \text{Actions}(s')} \hat{Q}_{\text{opt}}(s', a') \right) \quad (4)$$

, where  $\hat{Q}_{\text{opt}}(s, a)$  is the optimal q-value with given state  $s$  and action  $a$ .  $\eta$  is learning rate.  $r$  is the reward of after taking action  $a$  to transit from state  $s$  to state  $s'$ .  $\gamma$  is discount factor.

And the optimal action  $a$  for a given game state  $s$  is same as in eq. 2.

But vanilla q-learning has a problem. The agent always takes the action with maximum q-value. This will make agent never explore to other state that may lead to a higher reward. Therefore, there has a method called epsilon-greedy, it makes agent has a probability  $\epsilon$  to randomly decide the actions it takes. The algorithms can be described as below:

$$\pi_{\text{opt}}(s) = \begin{cases} \arg \max_{a \in \text{Actions}} \hat{Q}_{\text{opt}}(s, a) & \text{probability } 1 - \epsilon, \\ \text{random from } \text{Actions}(s) & \text{probability } \epsilon. \end{cases} \quad (5)$$

The code below is the implementation of q-learning with epsilon-greedy.

Code 6: `class QLearningAgent`

```

1 class QLearningAgent(ReinforcementAgent):
2     """
3     Q-Learning Agent
4
5     Functions you should fill in:
6     - computeValueFromQValues
7     - computeActionFromQValues
8     - getQValue
9     - getAction
10    - update
11
12    Instance variables you have access to
13    - self.epsilon (exploration prob)
14    - self.alpha (learning rate)
15    - self.discount (discount rate)
16
17    Functions you should use
18    - self.getLegalActions(state)
19      which returns legal actions for a state
20    """
21    def __init__(self, **args):
22        "You can initialize Q-values here..."
23        ReinforcementAgent.__init__(self, **args)
24
25        """ YOUR CODE HERE """
26        # Begin your code
27        self.value = defaultdict(lambda: defaultdict(float)) # Use nested defaultdict
28        ↪ to store q-value of given state and action as self.value[state][action]
29
30        # End your code
31
32    def getQValue(self, state, action):
33        """
34        Returns Q(state,action)
35        Should return 0.0 if we have never seen a state
36        or the Q node value otherwise
37        """
38        """ YOUR CODE HERE """
39        # Begin your code
40        return self.value[state][action] # Just return the corresponding q value

```

```

41     # End your code
42
43
44 def computeValueFromQValues(self, state):
45     """
46     Returns max_action Q(state,action)
47     where the max is over Legal actions. Note that if
48     there are no legal actions, which is the case at the
49     terminal state, you should return a value of 0.0.
50     """
51     """ YOUR CODE HERE """
52     # Begin your code
53     actions = self.getLegalActions(state) # Get all legal actions
54     if len(actions) == 0: # If no legal actions
55         return 0.0 # then return 0
56     else:
57         q_value = -1e9 # Initialize a variable to track the maximum q value of
58         ↪ current game state
59         for a in actions:
60             q_value = max(q_value, self.getQValue(state, a)) # Get q-value of given
61             ↪ state and action, and update the maximum q-value
62
63     return q_value # return maximum q state
64     # End your code
65
66
67 def computeActionFromQValues(self, state):
68     """
69     Compute the best action to take in a state. Note that if there
70     are no legal actions, which is the case at the terminal state,
71     you should return None.
72     """
73     """ YOUR CODE HERE """
74     # Begin your code
75     legalActions = self.getLegalActions(state) # Get all legal actions
76     action = None # Initialize the variable to track optimal action
77     """ YOUR CODE HERE """
78     # Begin your code
79     if len(legalActions) != 0:
80         q_value = -1e9 # Initialize a variable to track maximum q value
81         for a in legalActions:

```

```

79         if self.getQValue(state, a) > q_value: # Update maximum q value and the
           ↳ corresponding action
80             q_value = self.getQValue(state, a)
81             action = a
82
83     return action # return optimal action
84     # End your code
85
86 def getAction(self, state):
87     """
88     Compute the action to take in the current state. With
89     probability self.epsilon, we should take a random action and
90     take the best policy action otherwise. Note that if there are
91     no legal actions, which is the case at the terminal state, you
92     should choose None as the action.
93
94     HINT: You might want to use util.flipCoin(prob)
95     HINT: To pick randomly from a list, use random.choice(list)
96     """
97     # Pick Action
98     legalActions = self.getLegalActions(state)
99     action = None
100    """ YOUR CODE HERE """
101    # Begin your code
102    if util.flipCoin(self.epsilon): # Implementation of epsilon greedy
103        # Random sample
104        if len(legalActions) != 0: # If have legal actions
105            action = random.choice(legalActions) # then randomly select an action
106        else:
107            action = self.computeActionFromQValues(state) # Otherwise, use q value to
           ↳ get the optimal actions of current game state
108
109    return action # return that action
110    # End your code
111
112
113 def update(self, state, action, nextState, reward):
114     """
115     The parent class calls this to observe a
116     state = action => nextState and reward transition.
117     You should do your Q-Value update here

```

```

118
119     NOTE: You should never call this function,
120     it will be called on your behalf
121     """
122     """** YOUR CODE HERE """
123     # Begin your code
124     # Use q-learning update formula to update Q(s, a)
125     self.value[state][action] = (1 - self.alpha) * self.value[state][action] +
126     ↪ self.alpha * (reward + self.discount *
127     ↪ self.computeValueFromQValues(nextState))
128     # End your code
129
130 def getPolicy(self, state):
131     return self.computeActionFromQValues(state)
132
133 def getValue(self, state):
134     return self.computeValueFromQValues(state)

```

## 2.3 Approximate Q-learning

Approximate q-learning learn the weights of features with given state and action. In other words, approximate q-learning assume that there are  $n$  features vectors  $f_1(s, a), f_2(s, a), \dots, f_n(s, a)$  with given state  $s$  and action  $a$ . And the corresponding weights are  $w_1, w_2, \dots, w_n$ . With these assumptions, the q-value of given state and action is:

$$Q(s, a) = \sum_{i=1}^n w_i \cdot f_i(s, a) \quad (6)$$

To update these  $n$  weights, we can use the equations below:

$$w_i \leftarrow w_i + \alpha (\text{Reward}(s, a, s') + \gamma V(s') - Q(s, a)) f_i(s, a) \quad (7)$$

The notation here is same as normal q-learning.

Below is the implementation of approximate q-learning agent.

Code 7: class ApproximateQAgent

```
1 class ApproximateQAgent(PacmanQAgent):
2     """
3     ApproximateQLearningAgent
4
5     You should only have to overwrite getQValue
6     and update. ALL other QLearningAgent functions
7     should work as is.
8     """
9     def __init__(self, extractor='IdentityExtractor', **args):
10         self.feateExtractor = util.lookup(extractor, globals())()
11         PacmanQAgent.__init__(self, **args)
12         self.weights = util.Counter()
13
14     def getWeights(self):
15         return self.weights
16
17     def getQValue(self, state, action):
18         """
19         Should return  $Q(state, action) = w * featureVector$ 
20         where  $*$  is the dotProduct operator
21         """
22         """** YOUR CODE HERE **"""
23         # Begin your code
24         # get weights and feature
25         featureVectors = self.feateExtractor.getFeatures(state, action) # Get feature
26         ↪ vectors (type = util.Counter()) using getFeatures(state, action)
27         res = 0 # Initalize return value
28         # Dot product of  $w * featureVector$ 
29         for feature in featureVectors:
30             res += featureVectors[feature] * self.weights[feature]
31
32         return res
33         # End your code
34
35     def update(self, state, action, nextState, reward):
36         """
37         Should update your weights based on transition
38         """
```

```

37         """
38         """
39         # Begin your code
40         featureVectors = self.featsExtractor.getFeatures(state, action) # Get feature
41         ↪ vectors (type = util.Counter()) using getFeatures(state, action)
42         # Using ApproximateQLearningAgent's formula to update every weights that
43         ↪ corresponds to a specific feature
44         for feature in featureVectors:
45             correction = reward + self.discount *
46             ↪ self.computeValueFromQValues(nextState) - self.getQValue(state, action)
47             self.weights[feature] = self.weights[feature] + self.alpha * correction *
48             ↪ featureVectors[feature]
49         # End your code
50
51     def final(self, state):
52         "Called at the end of each game."
53         # call the super-class final method
54         PacmanQAgent.final(self, state)

```

Below is the implementation of feature extractor.

Noticed that the implementation of function `closestFood` is same as Code: 4.

#### Code 8: class SimpleExtractor

```

1 class SimpleExtractor(FeatureExtractor):
2     """
3     Returns simple features for a basic reflex Pacman:
4     - whether food will be eaten
5     - how far away the next food is
6     - whether a ghost collision is imminent
7     - whether a ghost is one step away
8     """
9
10    def getFeatures(self, state, action):
11        # extract the grid of food and wall locations and get the ghost locations
12        food = state.getFood()

```



```

13     capsules = state.getCapsules()
14     walls = state.getWalls()
15     ghosts = state.getGhostPositions()
16     features = util.Counter()
17     features["bias"] = 1.0
18
19     # compute the location of pacman after he takes the action
20     x, y = state.getPacmanPosition()
21     dx, dy = Actions.directionToVector(action)
22     next_x, next_y = int(x + dx), int(y + dy)
23
24     # count the number of ghosts 1-step away which is not in scared status
25     # We can use state.data.agentStates[i+1].scaredTimer to determine a ghost is
26     ↳ scared now. If scaredTimer == 0, then this ghost is not scared. Otherwise,
27     ↳ it's scared now.
28     # Pacman can eat these scared ghosts to get a higher score
29     features["#-of-ghosts-1-step-away"] = sum(((next_x, next_y) in
30     ↳ Actions.getLegalNeighbors(g, walls) and
31     ↳ state.data.agentStates[i+1].scaredTimer == 0) for i, g in
32     ↳ enumerate(ghosts))
33
34     # if there is no danger of ghosts then add the food feature
35     if not features["#-of-ghosts-1-step-away"] and food[next_x][next_y]:
36         features["eats-food"] = 1.0
37
38     features["cnt-food"] = state.getNumFood() / 200 # Get total number of remaining
39     ↳ food
40
41     dist = closestFood((next_x, next_y), food, walls) # Get closest food using BFS
42     dist_cap = None # Distance of closest capsule
43     dist_scared = None # Distance of closest scared ghost
44
45     if len(capsules) != 0: # If has remaining capsules
46         # Using Manhattan distance to evaluate closest capsules
47         # Using BFS here will make training process really slow
48         dist_cap = abs(next_x-capsules[0][0]) + abs(next_y-capsules[0][1])
49         for cap in capsules[1:]:
50             dist_cap = min(dist_cap, abs(next_x-cap[0]) + abs(next_y-cap[1]))
51
52     for i in range(1, len(ghosts)):
53         if state.data.agentStates[i].scaredTimer != 0:
54             # If this ghost is scared now

```

```

48         # Using Manhattan distance to evaluate closest ghosts
49         if dist_scared == None:
50             dist_scared = abs(next_x-ghosts[i-1][0]) + abs(next_y-ghosts[i-1]
51                 ↪ [1])
52         else:
53             dist_scared = min(dist_scared, abs(next_x-ghosts[i-1][0]) +
54                 ↪ abs(next_y-ghosts[i-1][1]))
55
56     if dist is not None:
57         # make the distance a number less than one otherwise the update
58         # will diverge wildly
59         # Using different weight 2.5
60         features["closest-food"] = float(dist) / (walls.width * walls.height) * 2.5
61
62     if dist_cap is not None and dist_scared is None:
63         # Using different weight 10
64         features["closest-capsule"] = float(dist_cap) / (walls.width * walls.height)
65         ↪ * 10
66
67     if dist_scared is not None:
68         # Using different weight 1
69         features["closest-scared"] = float(dist_scared) / (walls.width *
70             ↪ walls.height)
71
72     features.divideAll(10.0) # Divide all features value with 10.0
73     return features

```

### 3 Deep Q-learning

Deep Q-learning (DQN) uses a deep neural network to get the optimal action with given game state. In other words, DQN use a neural network to replace the original Q-table. The update process of Q-value turns into the back-propagation of the neural network.

I have trained DQN using the provided code. In the next section, I will compare DQN with other methods I implemented in this homework.

## 4 Comparisons

Table 1: Different Method Comparisons (`random.seed(0)`, 100 games, 1 ghost, smallClassic)

| Method   | Win Rate | Average Score  |
|--|----------|----------------|
| Mimimax (depth=2)                              | 0.44     | -203.13        |
| Mimimax (depth=2, betterEvalFn)                | <b>1</b> | 1154.24        |
| Expectimax (depth=2)                           |          |                |
| Expectimax (depth=2, betterEvalFn)             | <b>1</b> | <b>1176.32</b> |
| Vanilla Q-learning (trained 2000 episodes)     | 0        | -405.8         |
| Approximate Q-learning (trained 2000 episodes) | 0.86     | 1054.41        |
| DQN (trained 10000 episodes)                   | 0.92     | 1161.17        |

For q-learning method,  $\epsilon = 0.05$ ,  $\gamma = 0.8$ ,  $\alpha = 0.2$ .

As we can see in the table. The best method is Expectimax with search depth equals to 2 and use custom evaluation function I implement as Code 3 to evaluate a given state. This method have achieved 100% win rate and highest average score 1176.32. I think this is because the human-written evaluation function make agent get more information about current state. Compared with the default evaluation function, which only return the score of given game state.

We can also notice that two q-learning method, vanilla q-learning and approximate q-learning, have huge difference in both win rate and average score. I think the difference is because vanilla q-learning only update q-value using score of current game state. And for layout like smallClassic that I use for testing each methods, there are too many possible state. This make agent can't find a optimal policy. This will happen even if I adjust  $\epsilon$  to 0.5, making the agent explore more game state.

Compared with approximate q-learning, which use human-written features to update its weights. In this way, agent know what is right direction to update its q-value.

Finally, DQN use deep neural network to get optimal action with given state. Although it seems

like this method can outperform the traditional searching method like expetimax. In fact, it doesn't. I think the reason is that 10000 episodes are still too few for DQN. Therefore, I think after some tuning in hyperparameters, DQN may outperform traditional searching methods.

## 5 Discussion

### 5.1 Pacman Rushes to The Closest Ghost in trappedClassic



Figure 1: Pacman rushes to the orange ghost and lose the game

I think it's because the minimax algorithm make pacman impossible to go left. If pacman go left, the worst case is that the blue ghost go right, and this will make pacman lose the game. Therefore, the only possible action is to go right, which also make the pacman lose the game.

But for expectimax algorithm, the pacman doesn't consider the worst case, instead, it considers the average case. So if the blue ghost chooses to go down at first move. The pacman will have the chance to eat that four dots. Like the figure below.



Figure 2: Expectimax algorithm successfully wins trappedClassic

## 6 Conclusion

In this homework, I learn how to write minimax, expectimax, q-learning and approximate q-learning algorithm.

Pacman game is a great example for testing search algorithm and q-learning algorithm.