Report

Adversarial Search

1.1 Minimax Search

- getAction(self , gameState) → 決定下一步
- minimax(gameState , depth , agent) 有三種可能的情形
    a. 已經贏了、已經輸了、已到達最深深度 (self.depth)
    b. agent = 0 (pacman)
    c. agent ≠ 0 (ghost)

```
class MinimaxAgent(MultiAgentSearchAgent):

    def getAction(self, gameState):

        def minimax(gameState,depth,agent):
            '''
            if a pacman lose, win, or reach the deepest depth(self.depth),
            we can return self.evaluationFunction(gameState) directly
            '''
            if gameState.isLose() or gameState.isWin() or depth == self.depth:
                return self.evaluationFunction(gameState)
            if agent == 0: # pacman
                # get legal actions of a pacman
                action_list = gameState.getLegalActions(0)
                maxi = -100000 # initial maxi
                # Iterate all possible action of a pacman
                for action in action_list:
                    nextstate = gameState.getNextState(0,action)
                    # call recursive function minimax
                    maxi = max(maxi,minimax(nextstate,depth,1))
                return maxi # return max score
            else: # ghost
                next_agent = agent + 1 # num of next ghost
                # get legal actions of the ghost
                action_list = gameState.getLegalActions(agent)
                mini = 100000 # initial mini
                '''
                if current ghost is the last ghost, next agent will be a pacman
                and go deeper (depth+=1)
                '''
                if next_agent == gameState.getNumAgents():
                    next_agent = 0
                    depth += 1
                # Iterate all possible action of the ghost
                for action in action_list:
                    nextstate = gameState.getNextState(agent,action)
                    # call recursive function minimax
                    mini = min(mini,minimax(nextstate,depth,next_agent))
                return mini # return min score

        # get legal actions of a pacman
        pacman_action_list = gameState.getLegalActions(0)
        maxscore = -100000 # initial maxscore
        next_action = '' # initial next_aaction
        # Iterate all possible action of a pacman
        for action in pacman_action_list:
            nextstate = gameState.getNextState(0,action)
            score = minimax(nextstate,0,1) # score of the state
            # find max score and next action
            if score > maxscore:
                next_action = action
```

```
            maxscore = score
        return next_action # return next action
```

📌 python pacman.py -p MinimaxAgent -l trappedClassic -a depth=3

minimax 算法會算出 pacman 無論往左或往右都會死，而且玩的時間越久分數會越低，所以 pacman 在知道自己必死無疑的情況下，他會選擇去自殺，這樣分數會比死撐在那卻還是輸來的高

📌 python pacman.py -p MinimaxAgent -l minimaxClassic -a depth=4 -q -n 500

a. Average Score: 116.102
b. Win Rate : 302/500 (0.60)

1.2 Expectimax Search

Minimax 是假設對手發揮最佳，但並不是每個對手都是最佳對手，所以 Expectimax 採用的是隨機對手的概念，Minimax 在實作 ghost 的時候是回傳最小值，但在 Expectimax 則是回傳期望值，但無論 ghost 採取哪一種行動都是相同的，所以這邊可以採用平均值來代替，除了以上部分和 Minimax 不同外，其他地方的幾乎都相同

```
class ExpectimaxAgent(MultiAgentSearchAgent):

    def getAction(self, gameState):

        def expectimax(gameState,depth,agent):
            '''
            if a pacman lose, win, or reach the deepest depth(self.depth),
            we can return self.evaluationFunction(gameState) directly
            '''
            if gameState.isLose() or gameState.isWin() or depth == self.depth:
                return self.evaluationFunction(gameState)
            if agent == 0: # pacman
                # get legal actions of a pacman
                action_list = gameState.getLegalActions(0)
                maxi = -100000 # initial maxi
                # Iterate all possible action of a pacman
                for action in action_list:
                    nextstate = gameState.getNextState(0,action)
                    # call recursive function expectimax
                    maxi = max(maxi,expectimax(nextstate,depth,1))
                return maxi # return max score
            else: #  ghost
                next_agent = agent + 1 # num of next ghost
                # get legal actions of thr ghost
                action_list = gameState.getLegalActions(agent)
                total_expected = 0 # initial total_expected
                length = len(action_list) # num of legal actions
                '''
                if current ghost is the last ghost, next agent will be a pacman
                and go deeper (depth+=1)
                '''
                if next_agent == gameState.getNumAgents():
                    next_agent = 0
                    depth += 1
                # Iterate all possible action of the ghost
```

```
            for action in action_list:
                nextstate = gameState.getNextState(agent,action)
                # call recursive function expectimax and sum up
                total_expected += expectimax(nextstate,depth,next_agent)
            return float(total_expected)/float(length) # return average

     # get legal actions of a pacman
    pacman_action_list = gameState.getLegalActions(0)
    maxscore = -100000 # initial maxscore
    next_action = '' # initial next_action
    # Iterate all possible action of a pacman
    for action in pacman_action_list:
        nextstate = gameState.getNextState(0,action)
        score = expectimax(nextstate,0,1) # score of the state
        # find maxscore and next action
        if score > maxscore:
            next_action = action
            maxscore = score
    return next_action # return next action
```

📌 python pacman.py -p ExpectimaxAgent -l trappedClassic -a depth=3

在 Expectimax 裡，pacman 不考慮最壞情況，而是考慮平均情況，所以如果遇到 pacman 被包夾這種情況，他可能會選擇去多吃幾個點點，而不是像 Minimax 選擇去自殺

📌 python pacman.py -p ExpectimaxAgent -l minimaxClassic -a depth=3 -q -n 500

→ Average Score : 135.058

→ Win Rate : 314/500 (0.63)

1.3 Better Evaluation Function

一開始的 Evaluation Function 只會回傳終端狀態的分數，但這並不能每次都贏得比賽，甚至是拿到高分，所以我便實作了以下的 Better Evaluation Function (1.4的地方會進行比較)

```
def betterEvaluationFunction(currentGameState):

    if currentGameState.isWin(): # if the pacman win, return a high score
        return 10000000
    elif currentGameState.isLose(): # if the pacman lose, return a low score
        return -10000000

    foodlist = currentGameState.getFood().asList() # get a list of food
    food_num = len(foodlist) # get the number of remaining food
    ghost_states = currentGameState.getGhostStates() # get ghost states
    pacman_pos = currentGameState.getPacmanPosition() # get the position of a pacman
    # get the num of remaining capsules
    capsule_num = len(currentGameState.getCapsules())
    score = currentGameState.getScore() # get current score

    # fewer food (False in foodlist) will have higher score
    score += foodlist.count(False)

    food_distance = 0.0 # initial food_distance

    # Iterate all food in foodlist
```

```
    for food in foodlist:
        # utilize util.manhattanDistance to calculate food_distance
        food_distance += util.manhattanDistance(pacman_pos,food)
    if food_distance != 0: # denominator can't be zero
        '''
        smaller food_distance will have higher score
        Therefore, I use reciprocal to calculate score in this part
        '''
        score += 1.0/food_distance
    total_scared = 0 # initial total_scared
    total_ghost_distance = 0 # initial total_ghost_distance
    # Iterate all states in ghost_states
    for ghost in ghost_states:
        total_scared += ghost.scaredTimer # calculate all scaredTimer
        # utilize util.manhattanDistance to calculate total_ghost_distance
        total_ghost_distance += util.manhattanDistance(pacman_pos,ghost.getPosition())
    '''
    if there are some ghosts in scared_time state, pacman can touch the ghost.
    Therefore, more scare_time, smaller total_ghost_distance and fewer capsule_num
    will have higher score
    However, if there are no ghost in scared_time state, more total_ghost_distance
    and more capsule_num will have higher score
    '''
    if total_scared == 0:
        score += total_ghost_distance + capsule_num
    else:
        score += total_scared
        score -= total_ghost_distance + capsule_num

    return score
```

1.4 Comparison

a. 讓 Minimax 和 Expectimax 在 minimaxClassic 這個 layout 進行不同 depth 的測試，每次皆進行五百次的測試，接著比較平均分數和勝率，從下方圖表可以發現 depth 越大勝率和平均分數都會越高，除此之外，Expectimax (depth=2) 的勝率已經和 Minimax (depth=4) 相等，所以我們可以推斷，Expectimax 應該比 Minimax 還要來的好，因為大部分對手應該都不會是最佳對手。

	Depth	Average Score	Win Game	Win Rate
Minimax	2	- 121.932	185 / 500	37 %
	3	- 114.072	189 / 500	38 %
	4	116.102	302 / 500	60 %
Expectimax	2	111.404	301 / 500	60 %
	3	135.058	314 / 500	63 %
	4	354.076	420 / 500	84 %

b. 讓 Minimax 和 Expectimax 在 smallClassic 這個 layout 進行一百次的測試，接著比較 score Evaluation Function 和 better Evaluation Function 的平均分數和勝率，從下方圖表可以發現，在 smallClassic 這個 layout，Minimax 和 Expectimax 的勝率和平均分數都下降了，但如果使用 better Evaluation Function，勝率甚至可以到達 100%，平均分數也高很多

	Average Score	Win Game	Win Rate
Minimax (scoreEvaluationFunction)	11.75	15 / 100	15 %
Minimax (betterEvaluationFunction)	1356.79	100 / 100	100 %
Expectimax (scoreEvaluationFunction)	304.65	30 / 100	30 %
Expectimax (betterEvaluationFunction)	1449.23	100 / 100	100 %

Q-learning

2.1 Value Iteration

value iteration 是一種在 MDP 假設下能找到最佳策略的算法，主要概念是要先想辦法計算出每個狀態出發能得到的最佳獎勵值，將之記錄起來，再找出每個狀態採取哪個行動得到的獎勵值最高，這樣就能知道在每個狀態時需要採取怎樣的行動，這就是最佳的策略

```
class ValueIterationAgent(ValueEstimationAgent):

    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
        self.runValueIteration()

    def runValueIteration(self):

        states = self.mdp.getStates() # get states

        for it in range(self.iterations): # run self.iteration times
            temp = util.Counter() # a counter used to store temp value
            # if current state is terminal state, set its value to 0
            for state in states:
                if self.mdp.isTerminal(state):
                    self.values[state] = 0
                    continue
                maxi = -100000 # initial maxi
                # get all possible actions
                actions = self.mdp.getPossibleActions(state)
                # Iterate all possible actions
                for action in actions:
                    # get all possible states and their correspoding probability
                    probs = self.mdp.getTransitionStatesAndProbs(state,action)
                    # compute qvalue
                    value = self.computeQValueFromValues(state,action)
                    maxi = max(maxi,value) # find max qvalue
                if maxi > -100000:
                    temp[state] = maxi # store max qvalue in temp
            # Iterate all states
            for state in states:
                # store temp[state] in self.values[state]
                self.values[state] = temp[state]

    def getValue(self, state):
        return self.values[state] # get value


    def computeQValueFromValues(self, state, action):
        # get all possible states and their correspoding probability
        probs = self.mdp.getTransitionStatesAndProbs(state,action)
        value = 0.0 # initial value
        # compute qvalue
        for nextstate,prob in probs:
            value += prob*(self.mdp.getReward(state,action,nextstate) + self.discount*self.values[nextstate])
        return value # return qvalue

    def computeActionFromValues(self, state):
        maxaction = None # initial maxaction
        maxi = -100000 # initial maxi
        # get all possible actions
        actions = self.mdp.getPossibleActions(state)
        # Iterate all possible actions
        for action in actions:
            # compute qvalue
           value = self.computeQValueFromValues(state,action)
           if value > maxi:
                maxi = value # maxi always be max qvalue
```

```
                maxaction = action # maxaction always be action of max qvalue
        return maxaction return maxaction

    def getPolicy(self, state):
        return self.computeActionFromValues(state)

    def getAction(self, state):
        return self.computeActionFromValues(state)

    def getQValue(self, state, action):
        return self.computeQValueFromValues(state, action)
```

📌 python gridworld.py -a value -i 100 -k 10

episode 10 complete : return was 0.47829690000000014

average returns from start state : 0.5005896415996667

2.2 Q-learning

在有多種狀態的遊戲中，不太可能運用 value iteration 得到正確的值，所以我們需要用 Q-learing，Q-learing 要記錄學習過的政策，並告訴 agent 採取什麼行動會有最大的獎勵值

```
class QLearningAgent(ReinforcementAgent):

    def __init__(self, **args):
        ReinforcementAgent.__init__(self, **args)
        self.qvalue = util.Counter() # initial self.qvalue

    def getQValue(self, state, action):
        return self.qvalue[state,action] # return corresponding qvalue

    def computeValueFromQValues(self, state):
        legal_actions = self.getLegalActions(state) # get legal actions
        # if no legal actions, return 0.0
        if not legal_actions:
            return 0.0

        max_Q = -100000 # initial max_Q
        # Iterate all legal actions
        for action in legal_actions:
            # update max_Q
            max_Q = max(max_Q,self.getQValue(state,action))
        return max_Q  # return max_Q

    def computeActionFromQValues(self, state):
        legal_actions = self.getLegalActions(state) # get legal actions
        best_action = [] # initial best_action

        max_Q = -100000 # initial max_Q
        # Iterate all legal actions
        for action in legal_actions:
            q = self.getQValue(state,action) # get qvalue of given state and action
            if q > max_Q:
                max_Q = q # update max_Q
                best_action = [action]  # update best_action
            # if qvalue equals max_Q, append action in best_action
            elif q == max_Q:
                best_action.append(action)
        # if best_action is none, action = None
        if not best_action:
            action = None
        # if best_action isn't none, choose a random action in best_action
```

```
        else:
            action = random.choice(best_action)
        return action # return action

    def getAction(self, state):
        # get legal actions
        legal_actions = self.getLegalActions(state)
        action = None # initial action
        # Implementation of epsilon greedy
        if util.flipCoin(self.epsilon):
            if len(legal_actions) != 0:
                # select an action randomly
                action = random.choice(legal_actions)
        else:
            # use qvalue to get the action
            action = self.computeActionFromQValues(state)
        return action # return action


    def update(self, state, action, nextState, reward):
        # use formula to update self.qvalue[state,action]
        trans = reward + self.discount * self.computeValueFromQValues(nextState)
        self.qvalue[state,action] = self.alpha * trans + (1.0 - self.alpha) * self.getQValue(state,action)

    def getPolicy(self, state):
        return self.computeActionFromQValues(state)

    def getValue(self, state):
        return self.computeValueFromQValues(state)
```

📌 python gridworld.py -a q -k 100

episode 10 complete : return was 0.16677181699666577

average returns from start state : 0.2980386385898118

📌 python gridworld.py -a q -k 100 --noise 0.0 -e 0.9

Compare different epsilon values

根據下表可以發現基本上 epsilon 越高，回傳值越低，這是因為 epsilon 越高，隨機採取行動的機率越高，一般來說，利用 qvalue 算出所要採取的行動會比隨機採取行動來的好，這就是為什麼epsilon 越高，回傳值會越低

epsilon	episode 10 complete	average returns from start state
0.9	- 0.08862938119652508	- 0.02207070789706160
0.8	0.08862938119652508	0.07647480856248293
0.7	0.28242953648100017	0.19106597516979842
0.6	0.31381059609000017	0.23032417731921145
0.5	0.43046721000000016	0.30334724025743126
0.4	0.5904900000000002	0.3171844461934324
0.3	0.47829690000000014	0.4442559792774145

epsilon	episode 10 complete	average returns from start state
0.2	0.5314410000000002	0.3984586878331274
0.1	0.5904900000000002	0.4545070312623784

2.3 Approximate Q-learning

Approximate q-learning 是為了解決狀態空間過大的問題，它會依照狀態和動作來學習特徵的權重，更新權重的方法和更新 qvalue 幾乎一樣

```
class ApproximateQAgent(PacmanQAgent):

    def __init__(self, extractor='IdentityExtractor', **args):
        self.featExtractor = util.lookup(extractor, globals())()
        PacmanQAgent.__init__(self, **args)
        self.weights = util.Counter()

    def getWeights(self):
        return self.weights

    def getQValue(self, state, action):
        # get weights and features
        features = self.featExtractor.getFeatures(state,action)
        q = 0.0 # initial q
        # dot product of weight * feature
        for key,value in features.items():
            q += value * self.weights[key]
        return q # return q

    def update(self, state, action, nextState, reward):
        # calculate correction
        correction = reward + self.discount * self.getValue(nextState)- self.getQValue(state,action)
        # get weights and features
        features = self.featExtractor.getFeatures(state,action)
        for key,value in features.items():
            # upsate self.weights
            self.weights[key] += self.alpha * correction * value

    def final(self, state):
        # call the super-class final method
        PacmanQAgent.final(self, state)
```

📌 python pacman.py -p ApproximateQAgent -x 2000 -n 2010 -l smallGrid

在 smallGrid 的 layout 中，ApproximateQAgent 可以到達勝率百分之百，平均分數是 498.4 分，但我接著將 layout 改成 mediumGrid，勝率直接掉到 20%，平均分數也下降成 -308.6

📌 python pacman.py -p ApproximateQAgent -a extractor=SimpleExtractor -x 50 -n 60 -l mediumGrid

如果改成使用 SimpleExtractor，確實能贏得比賽，實測之後勝率為 100%，平均分數為498.4

2.4 Comparison

讓 Q-learning 和Approximate Q-learning 在 smallClassic 這個 layout 進行測試，兩者皆是訓練2000 episodes，再進行一百次的測試，接著比較平均分數和勝率，從下方圖表可以發現 Q-learning 的勝率比較低，甚至沒贏過任何一場，但在其他 layout，Q-learning 還是有勝率的，我認為是因為 Q-learning 只有使用當前遊戲狀態的分數來更新 qvalue，對於 smallClassic 這種狀態過多 layout，可能會找不到最佳策略。但如果是 Approximate Q-learning，它是使用自己寫的 feature 來更新權重，所以 agent 可以知道正確的方向，並更新 qvalue。

Method	Average Score	Win Rate
Q-learning	-396.72	0 %
Approximate Q-learning	823.48	86 %

Deep Q-Network (DQN)

DQN 是利用深度神經網絡來取代 Q-learning 的 Q-table，並得到遊戲狀態的最佳動作，原先 Q值的更新過程變成了神經網絡的反向傳播

📌 python pacman.py -p PacmanDQN -n 10000 -x 10000 -l smallClassic

一開始先進行訓練

📌 python pacman.py -p PacmanDQN -n 10000 -x 10000 -l smallClassic

→ Average Score: 1363.17

→ Win Rate: 87/100 (0.87)

Comparison

condition :

1. times = 100
2. layout = smallClassic

Method	Average Score	Win Rate
Mimimax (depth=2)	11.75	15 %
Mimimax (depth=2, betterEvaluationFunction)	1356.79	100 %
Expectimax (depth=2)	304.65	30 %
Expectimax (depth=2, betterEvaluationFunction)	1449.23	100 %
Q-learning (train 2000 episodes)	-396.72	0 %
Approximate Q-learning (train 2000 episodes)	823.48	86 %
DQN (train 10000 episodes)	1363.17	87 %

從上方表格可以看到最佳方法是 depth 為 2 且使用 betterEvaluationFunction 的Expectimax，這樣的方式可以到達勝率百分之百，平均分數也可以到達1449.23，我認為最大的關聯應該是在 betterEvaluationFunction，它不像 scoreEvaluationFunction 只回傳遊戲狀態的分數，反而可以提供 agent 更多關於目前狀態的資訊。

勝率第二高的是 DQN，原先我認為 DQN 應該要是勝率最高的，或是至少要接近 100%，但從結果看起來似乎不是這樣，我覺得有可能是因為 10000 次還太少，搞不好訓練的次數變多會提高勝率。

接著，我們可以發現 Q-learning 的勝率最低，甚至沒贏過任何一場，但在2.3，我有測試過其他 layout，Q-learning 還是有勝率的，所以我認為是因為 Q-learning 只有使用當前遊戲狀態的分數來更新 qvalue，對於 smallClassic 這種狀態過多 layout，可能會找不到最佳策略。但如果是 Approximate Q-learning，它是使用自己寫的 feature 來更新權重，所以 agent 可以知道正確的方向，並更新 qvalue。

Problem

1. 一開始寫這份作業的時候，我完全不知道該如何下手，後來查了一些資料，慢慢了解三個 part 的內容和差異後，便自己開始有了頭緒，後來也越寫越順，成功的完成作業。

2. 在實作 betterEvaluationFunction 的時候，我一直不知道要如何處理 food_distance，因為food_distance 越小，要得到越高的分數，我原鄉想的方法是用扣的，但想一想覺得很怪，但又想不到其他更好的辦法，想了很久後來才想到可以用倒數的方式來達到目標。除此之外，我一直沒辦法在 autograder 拿到滿分，測試了很多方法，最後真正讓我成功的方法是運用 scared_time，來加減一些分數。

3. 要訓練 DQN 的時候，我一直想用自己電腦的 gpu 來跑，但我一直無法成功，上網查了一些方法，也安裝了很多東西，但都一直用 cpu 在跑，後來下了一個也是在網路上查到的指令，結果突然就成功了，似乎是我之前下載的版本錯誤，才導致之前一直無法使用 gpu。