

## Part1

### 1-1 Minimax

#### Code explanation

在 minimax 演算法中各種情況的 V 如下:

$$V_{\text{minimax}}(s, d) = \begin{cases} \text{Utility}(s) & \text{IsEnd}(s) \\ \text{Eval}(s) & d = 0 \\ \max_{a \in \text{Actions}(s)} V_{\text{minimax}}(\text{Succ}(s, a), d) & \text{Player}(s) = \text{agent} \\ \min_{a \in \text{Actions}(s)} V_{\text{minimax}}(\text{Succ}(s, a), d - 1) & \text{Player}(s) = \text{opp} \end{cases}$$

因此我使用了 Vminimax() 來獲得各個情境下的 V

```
145 def Vminimax (self, gameState, tmpDepth, agentIndex):
146     #遞迴終止條件 不想了或玩完了
147     """
148     IsEnd(s): Vminimax(s) = Utility(s)
149     """
150     if tmpDepth== self.depth or gameState.isWin() or gameState.isLose():
151         return self.evaluationFunction(gameState)
```

150 與 151 行表示遞迴碰到了中止條件，即超過要考慮的 depth 或是遊戲結束，此時會回傳用 evaluationFunction() 計算出來的 Utility。

```
152     #精靈 做max
153     """
154     Player = agent: Vminimax(s) = max_action{Vminimax(Succ(s,a))}
155     """
156     if agentIndex==0:
157         MaxScore=float('-inf')
158         legalMoves = gameState.getLegalActions(agentIndex)
159         for move in legalMoves:
160             nextState = gameState.getNextState(agentIndex, move)
161             #預期下一步的鬼會做啥
162             nextStateScore =self.Vminimax(nextState,tmpDepth,agentIndex+1)
163             MaxScore = max(MaxScore,nextStateScore) #所有可選的之中挑最大
164         return MaxScore
```

如果 player 輪到小精靈，他會在所有可能的 action 中，挑出能讓下一步 V 最大的 action (即第 159 行的迴圈)。由於下一步的 V 是由鬼來做決定，因此在 162 行呼叫的是鬼。

```
165     #鬼 做min
166     """
167     Player = opp: Vminimax(s) = min_action{Vminimax(Succ(s,a))}
168     """
169     if agentIndex>0:
170         MinScore=float('inf')
171         legalMoves = gameState.getLegalActions(agentIndex)
172         for move in legalMoves:
173             nextState = gameState.getNextState(agentIndex, move)
174             if agentIndex < gameState.getNumAgents()-1: #還要讓下一隻鬼跑
175                 nextStateScore = self.Vminimax(nextState,tmpDepth,agentIndex+1)
176             else: #鬼都跑過一步了 換精靈
177                 nextStateScore = self.Vminimax(nextState,tmpDepth+1,0)
178             MinScore = min(MinScore,nextStateScore) #鬼就是故意要選最爛的給你
179         return MinScore
```

如果 player 輪到鬼，則他會在所有可能的 action 中，挑出能讓下一步 V 最小的 action (即第 172 行的迴圈)。此外，因為鬼不只一隻，所以在第 174 行會檢查是否所有的鬼都有移動了，如果還有鬼還沒動作的話就要呼叫下一隻鬼，否則就換小精靈。

而在 `getAction()` 中，由於第一步是小精靈採取行動，因此與  $V_{\text{minimax}}$  中小精靈的部分很像，都是跑遍所有可能的 action，只不過最後是回傳 action 而非 V。

```

134     legalMoves = gameState.getLegalActions(0) # 這一步是小精靈
135     MaxScore=float('-inf')
136     Move = None
137     for move in legalMoves:
138         nextState = gameState.getNextState(0, move)
139         Score =self.Vminimax(nextState,0,1) # 下一步是鬼
140         if Score>MaxScore:
141             MaxScore = Score
142             Move = move
143     return Move

```

## Observation

1. 小精靈會自殺是因為他預期鬼會選  $V_{\text{min}}$  的行為，但是實際上鬼的 action 是 random 的。由於小精靈預估錯誤，所以看起來就像自殺。
2. 我試跑了 `python pacman.py -p MinimaxAgent -l minimaxClassic -a depth=4 -n 10`，確實如 SPEC 所述有七成的勝率。

```

D:\Download\HW3\HW3\Adversarial_search>python pacman.py -p MinimaxAgent -l minimaxClassic -a depth=4 -q -n 10
Pacman emerges victorious! Score: 513
Pacman emerges victorious! Score: 516
Pacman emerges victorious! Score: 516
Pacman died! Score: -492
Pacman emerges victorious! Score: 516
Pacman emerges victorious! Score: 516
Pacman emerges victorious! Score: 516
Pacman died! Score: -492
Pacman emerges victorious! Score: 516
Pacman died! Score: -495
Average Score: 213.0
Scores: 513.0, 516.0, 516.0, -492.0, 516.0, 516.0, 516.0, -492.0, 516.0, -495.0
Win Rate: 7/10 (0.70)
Record: Win, Win, Win, Loss, Win, Win, Win, Loss, Win, Loss

```

## 1-2 Expectimax Search

### Code explanation

在 expectimax 演算法中各種情況的 V 如下：

$$V_{\text{expectimax}}(s) = \begin{cases} \text{Utility}(s) & \text{IsEnd}(s) \\ \max_{a \in \text{Actions}(s)} V_{\text{expectimax}}(\text{Succ}(s, a)) & \text{Player}(s) = \text{agent} \\ \sum_{a \in \text{Actions}(s)} \pi_{\text{opp}}(s, a) V_{\text{expectimax}}(\text{Succ}(s, a)) & \text{Player}(s) = \text{opp} \end{cases}$$

因此我使用了 Vexptmax() 來獲得各個情境下的 V

```
208     def Vexptmax (self,gameState,tmpDepth,agentIndex):
209         #遞迴終止條件 不想玩了或玩完了
210         """
211         IsEnd(s): Vexptmax(s) = Utility(s)
212         """
213         if tmpDepth== self.depth or gameState.isWin() or gameState.isLose():
214             return self.evaluationFunction(gameState)
```

213 與 214 行表示遞迴碰到了中止條件，即超過要考慮的 depth 或是遊戲結束，此時會回傳用 evaluationFunction() 計算出來的 Utility。

```
215         #精靈 做max
216         """
217         Player = agent: Vexptmax(s) = max_action{Vexptmax(Succ(s,a))}
218         """
219         if agentIndex==0:
220             MaxScore=float('-inf')
221             legalMoves = gameState.getLegalActions(agentIndex)
222             for move in legalMoves:
223                 nextState = gameState.getNextState(agentIndex, move)
224                 #預期下一步的鬼會做啥
225                 nextStateScore =self.Vexptmax(nextState,tmpDepth,agentIndex+1)
226                 MaxScore = max(MaxScore,nextStateScore) #所有可選的之中挑最大
227             return MaxScore
```

如果 player 輪到小精靈，他會在所有可能的 action 中，挑出能讓下一步 V 最大的 action (即第 222 行的迴圈)。由於下一步的 V 是由鬼來做決定，因此在 225 行呼叫的是鬼。

```
228         #鬼做exp
229         """
230         Player = opp: Vexptmax(s) = sum_action{pi_opp(s,a)*Vexptmax(Succ(s,a))}
231         """
232         if agentIndex>0:
233             Score=0
234             legalMoves = gameState.getLegalActions(agentIndex)
235             for move in legalMoves:
236                 nextState = gameState.getNextState(agentIndex, move)
237                 if agentIndex < gameState.getNumAgents()-1: #還要讓下一隻鬼跑
238                     Score += self.Vexptmax(nextState,tmpDepth,agentIndex+1)
239                 else: #鬼都跑過一步了 換精靈
240                     Score += self.Vexptmax(nextState,tmpDepth+1,0)
241             ExpScore = Score/len(legalMoves)
242             return ExpScore
```

如果 player 輪到鬼，則他會在所有可能的 action 中，挑出 V 的期望值最小的 action (即第 235 行的迴圈)。此外，因為鬼不只一隻，所以在第 237 行會檢查是否所有的鬼都有移動了，如果還有鬼還沒動作的話就要呼叫下一隻鬼，否則就換小精靈。當所有的可能都窮舉完後，因為這些 action 的發生機率都一樣，所以期望值就是總值除以可能的 action 數(即第 241 行)。

而在 `getAction()` 中，由於第一步是小精靈採取行動，因此與 `Vexptmax` 中小精靈的部分很像，都是跑遍所有可能的 `action`，只不過最後是回傳 `action` 而非 `V`。

```
197     legalMoves = gameState.getLegalActions(0) # 這一步是小精靈
198     MaxScore=float('-inf')
199     Move = None
200     for move in legalMoves:
201         nextState = gameState.getNextState(0, move)
202         Score =self.Vexptmax(nextState,0,1) # 下一步是鬼
203         if Score>MaxScore:
204             MaxScore = Score
205             Move = move
206     return Move
```

## Observation

當執行了 `python pacman.py -p ExpectimaxAgent -l trappedClassic -a depth=3 -n 10` 確實如 SPEC 所述，在左圖的情況下，雖然小精靈往左下衝有可能會像中圖被鬼吃掉，但他也有可能像右圖一樣逃脫，所以他還是會往左下衝。



這與 `minimax` 不同是因為 `minimax` 會假設鬼一定會往上衝來圍捕他，但 `expectimax` 只有一半的機率會往上衝，有另一半是有可能順利吃到豆子的。

## Part2

### 2-1 Value Iteration

#### Code explanation

Value Iteration 的演算法如下



#### Algorithm: value iteration [Bellman, 1957]

Initialize  $V_{\text{opt}}^{(0)}(s) \leftarrow 0$  for all states  $s$ .  
For iteration  $t = 1, \dots, t_{\text{VI}}$ :  
For each state  $s$ :  
$$V_{\text{opt}}^{(t)}(s) \leftarrow \max_{a \in \text{Actions}(s)} \sum_{s'} T(s, a, s') [\text{Reward}(s, a, s') + \gamma V_{\text{opt}}^{(t-1)}(s')]$$
  
$$Q_{\text{opt}}^{(t-1)}(s, a)$$

從最小的部分開始，即  $Q_{\text{opt}}$  的計算。其計算公式如下：

Optimal value if take action  $a$  in state  $s$ :

$$Q_{\text{opt}}(s, a) = \sum_{s'} T(s, a, s') [\text{Reward}(s, a, s') + \gamma V_{\text{opt}}(s')].$$

底下為 `computeQValueFromValues()` 的程式碼

```
107     """
108     公式
109     Q(s,a) = sum_{T(s,a,s')} [R(s,a,s') + discount * arg_max_{a' in actions} Q(s',a')]
110     """
111     QValue = 0
112     Transition = self.mdp.getTransitionStatesAndProbs(state, action)
113     for nextState, prob in Transition:
114         reward = self.mdp.getReward(state, action, nextState)
115         QValue = QValue + prob * (reward + self.discount * self.values[nextState])
116     return QValue
```

首先先用 `mdp.getTransitionStatesAndProbs()` 取得在  $s$  情況下做  $a$  所有可能的下一個情境  $s'$  以及他們的機率  $T(s,a,s')$ 。再來針對所有  $s'$ ， $\text{Reward}(s,a,s')$  可以使用 `mdp.getReward` 取得，而  $V_{\text{opt}}(s')$  可從 `self.values` 這個 Counter 取得。只要針對所有  $s'$  跑迴圈(第 113 行)，累計  $T(s,a,s')[\text{Reward}(s,a,s') + \text{discount} * V_{\text{opt}}]$  極為我們要求的  $Q_{\text{Value}}$ 。

有了  $Q_{\text{Value}}$  後就可以計算  $\pi_{\text{opt}}$ ，計算公式如下：

Given  $Q_{\text{opt}}$ , read off the optimal policy:

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

底下為 computeActionFromValues ()的程式碼:

```
135     legalActions = self.mdp.getPossibleActions(state)
136     #如果沒有legal actions 要回傳None
137     if len(legalActions)==0:
138         return None
139     ActionValue = util.Counter()
140     for action in legalActions:
141         ActionValue[action] = self.computeQValueFromValues(state, action)
142     bestAction = ActionValue.argmax()
143     return bestAction
```

就如同之前的公式所述，要先針對所有可能的 action 計算其  $Q_{\text{opt}}$  (即第 140 與 141 行)，之後再取  $\text{argMax}$  即可得到  $\pi_{\text{opt}}$  (第 142 行)。

有了上述兩個函數之後就可以計算  $V_{\text{opt}}$ ，公式如下

Optimal value from state  $s$ :

$$V_{\text{opt}}(s) = \begin{cases} 0 & \text{if } \text{IsEnd}(s) \\ \max_{a \in \text{Actions}(s)} Q_{\text{opt}}(s, a) & \text{otherwise.} \end{cases}$$

底下為 runValueIteration ()的程式碼:


```
71     """
72     公式
73     isEnd(s): V(s) = 0
74     otherwise: V(s) = max_{a in actions} Q(s,a)
75     """
76     for i in range (self.iterations):
77         StateValue = util.Counter()
78         for state in self.mdp.getStates():
79             #isEnd(s)
80             if self.mdp.isTerminal(state):
81                 StateValue[state] = 0
82                 continue
83             MaxAction = self.getPolicy(state)
84             MaxValue = self.getQValue(state, MaxAction)
85             StateValue[state] = MaxValue
86         #統一更新
87         self.values = StateValue
```

在一開始的演算法有提到，針對每個 iteration 都要更新  $V_{\text{opt}}(s)$ ，即第 76 行的迴圈。因次需要針對每一個 state  $s$  (第 78 行的迴圈)，去計算  $V_{\text{opt}}(s)$ 。如果該 state 要結束了，則  $V_{\text{opt}}(s) = 0$  (即第 80~82 行)，不然就是所有 action 中最大  $Q$  Value (第 83~85 行)。

## 2-2 Q-learning

### Code explanation

Q-learning 的演算法如下:

 **Algorithm: Q-learning [Watkins/Dayan, 1992]**

On each  $(s, a, r, s')$ :

$$\hat{Q}_{\text{opt}}(s, a) \leftarrow (1 - \eta) \underbrace{\hat{Q}_{\text{opt}}(s, a)}_{\text{prediction}} + \eta \underbrace{(r + \gamma \hat{V}_{\text{opt}}(s'))}_{\text{target}}$$

Recall:  $\hat{V}_{\text{opt}}(s') = \max_{a' \in \text{Actions}(s')} \hat{Q}_{\text{opt}}(s', a')$

底下為初始化的程式碼:

```
52         #紀錄QValue
53         self.QValue = util.Counter()
```

因為沒有見過的 state  $s$  其  $Q_{\text{opt}}(s) = 0$ ，而 `util.Counter` 具有對沒見過的 key 會補 0 的特性。

接著如果有需要知道  $Q_{\text{opt}}$  就用 `getQValue()` 回傳

```
68
69         return self.QValue[(state, action)]
```

有了  $Q_{\text{opt}}$  後就可以算  $\pi_{\text{opt}}$ ，公式與上方 value iteration 類似，底下為 `computeActionFromQValues()` 的程式碼

```
104         """
105         policy(s) = arg_max_{a in actions} Q(s,a)
106         """
107         legalActions = self.getLegalActions(state)
108         if len(legalActions) == 0:
109             return None
110         ActionValue = util.Counter()
111         for action in legalActions:
112             ActionValue[action] = self.getQValue(state, action)
113
114         #break ties randomly
115         MaxValue= ActionValue[ActionValue.argmax()]
116         MaxActions = [action for action in ActionValue if ActionValue[action] == MaxValue]
117         Action = random.choice(MaxActions)
118         return Action
```

先針對所有可能的 action 計算其  $Q_{\text{opt}}$  (第 110 與 111 行)。但是可能有不只一個 `argMax`，所以要先記錄最佳解的候選人 (第 115 與 116 行)，之後再用 `random.choice()` 隨機選一個做為  $\pi_{\text{opt}}$  (第 117 行)。

再來就可以去計算  $V_{opt}$ ，公式也與上方 value iteration 類似，底下為 `computeValueFromQValues()` 的程式碼

```

83     """
84     公式
85     isEnd(s): V(s) = 0
86     otherwise: V(s) = max_{a in actions} Q(s,a)
87     """
88     legalActions = self.getLegalActions(state)
89     if len(legalActions)==0:
90         return 0.0
91     MaxAction = self.getPolicy(state)
92     MaxValue = self.getQValue(state, MaxAction)
93     return MaxValue

```

如果該 state 要結束了，即沒有辦法再走下一步，則  $V_{opt}(s) = 0$  (即第 89、90 行)，不然就是所有 action 中最大 Q Value(第 91~93 行)。

最後是 update 的部分，公式如下：

$$\hat{Q}_{opt}(s, a) \leftarrow (1 - \eta) \underbrace{\hat{Q}_{opt}(s, a)}_{\text{prediction}} + \eta \underbrace{(r + \gamma \hat{V}_{opt}(s'))}_{\text{target}}$$

底下為 `update()` 的程式碼

```

164     """
165     Qopt(s,a) <- (1-eta)Qopt(s,a)+ eta(reward + discount* Vopt(s'))
166     Qopt(s,a)為 prediction
167     reward + discount* Vopt(s')為target
168     """
169     prediction = self.getQValue(state, action)
170     prediction_ = (1-self.alpha)*prediction
171     Value = self.getValue(nextState)
172     target = reward + self.discount* Value
173     target_ = self.alpha * target
174     self.QValue[(state,action)] = prediction_ + target_

```

首先在第 169、170 行，使用 `getQValue` 取得  $Q_{opt}(s,a)$ ，即 prediction 的部分，之後再乘以  $1-\eta$  (在程式裡  $\eta$  是 `self.alpha`)。然後在第 171~173 行，使用 `getValue` 取得  $V_{opt}(s')$ ，並計算 target 項，再乘以  $\eta$ 。最後將兩項相加，更新  $Q_{opt}(s,a)$ 。

## 2-3 epsilon-greedy action selection

### Code explanation

epsilon-greedy 的演算法如下：



#### Algorithm: epsilon-greedy policy

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



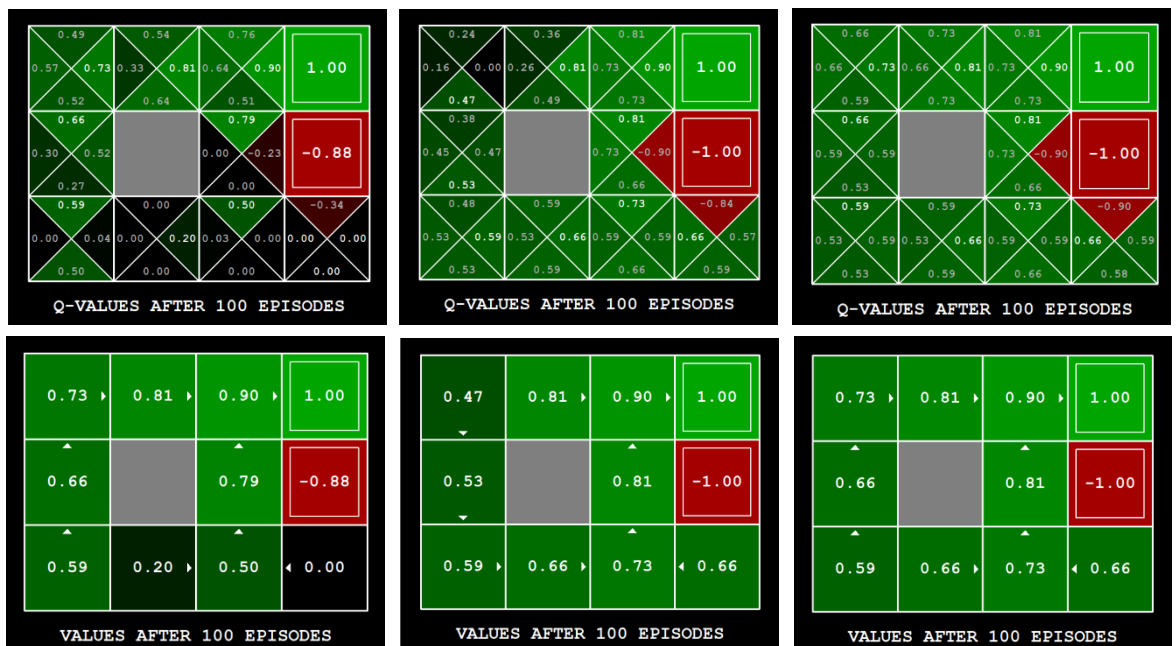
底下為 `getAction()` 的程式碼:

```
139         if len(legalActions) == 0:
140             return None
141         explore = util.flipCoin(self.epsilon)
142         #硬幣擲到要隨機選action的話
143         if explore:
144             action = random.choice(legalActions)
145             #硬幣說要直接走
146         else:
147             action = self.getPolicy(state)
148         return action
```

如果沒得做下一步要回傳 `None` (第 139、140 行)。如果有得選就使用 `util.flipCoin(epsilon)`，即第 141 行，決定是否做 `exploration`。其中有 `epsilon` 的機率會是 `True`，就從所有可行的 `action` 中隨機選取，即做 `exploration`(第 143、144 行)。如果沒有要 `exploration`，則用 `getPolicy` 找到這個 `state` 下該做的 `action`(第 146、147 行)。

## Observation

當使用不同的 `epsilon`，會發現探索的程度差很多，下圖從左到右的 `epsilon` 分別為 0.1, 0.5 及 0.9。可以看到最左邊還有很多未探索的(尤其是右下角區域)，而右邊全部都走過了。



## 2-4 Approximate Q-learning

### Code explanation

Approximate Q-learning 的演算法如下:

Algorithm: Q-learning with function approximation

$$Q(s, a) = \sum_i^n f_i(s, a) w_i$$

$\underbrace{s, a; \mathbf{w}}_{\text{prediction}} - \underbrace{(r + \gamma \hat{V}_{\text{opt}}(s'))}_{\text{target}} \phi(s, a)$

其中 Q 的估計值計算方法如下:

底下為 getQValue()的程式碼:

```
245 featureVector = self.featExtractor.getFeatures(state, action)
246 Q = 0
247 for feature in featureVector.keys():
248     Q += self.weights[feature]*featureVector[feature]
249 return Q
```

先透過 featureExtractor.getFeatures()取得 features (第 245 行)，再將所有 feature<sub>i</sub> 乘上對應的權重 w<sub>i</sub>，加起來就是 Q 的估計值(第 247、248 行)。

再來是更新權重的部分，更新的公式如下:

$$w_i \leftarrow w_i + \alpha[\text{correction}]f_i(s, a)$$
$$\text{correction} = (R(s, a) + \gamma V(s')) - Q(s, a)$$

底下為 update()的程式碼:

```
260 featureVector = self.featExtractor.getFeatures(state, action)
261 correction = (reward + self.discount* self.getValue(nextState))-(self.getQValue(state, action))
262 for feature in featureVector.keys():
263     self.weights[feature] = self.weights[feature] + self.alpha* correction *featureVector[feature]
264
```

一樣是先用 featureExtractor.getFeatures()取得 features (第 260 行)。然後按照上面的第二條公式計算 correlation (第 261 行)。之後再用第一條公式去更新權重(第 262、263 行)。

### Observation

我對於 featureExtractors.py 中的 SimpleExtractor 做了一些更動。

起因是我知道當小精靈吃了 capsule 後，它就不怕鬼甚至還可以反過來吃鬼得分。但是當我在還沒更動任何程式前執行 python pacman.py -p

ApproximateQAgent -x 50 -n 60 -l smallClassic -a extractor=SimpleExtractor 時，發現小精靈不會去吃 capsule。

以下是我做的更動:

首先用 `getCapsules()` 取得 capsule 所在的座標(第 80 行)。再用 `getNumAgents()` 減去一(小精靈本人)後得到 ghost 總數 (第 81 行)。接著針對所有的 ghost，使用 `getGhostState()`獲取每隻鬼的狀態是沒有被嚇(值為 0)，或是被嚇到的狀態還可以持續多久(第 82 行)。

```
80 capsules = state.getCapsules()
81 num_ghost = state.getNumAgents()-1
82 ghostState = [state.getGhostState(agentID).scaredTimer for agentID in range(1,num_ghost+1)]
```

在原先的程式的第 99 行可以看到，食物的 feature 為 1。而我為了鼓勵小精靈去吃 capsule， 在第 102 行設定如果該座標是 capsule 的話，feature 為 1.5，這會使小精靈比起吃普通食物，更想吃 capsule。接著當小精靈吃到 capsule 後，它就不用躲鬼而是去吃鬼贏得更高的分數，因此我在第 104~106 行寫到，如果隔壁的那隻鬼是被嚇到的(`ghostState>0`)，就可以去反吃那隻鬼，feature 為 2。

```
97 # if there is no danger of ghosts then add the food feature
98 if not features["#-of-ghosts-1-step-away"] and food[next_x][next_y]:
99     features["eats-food"] = 1.0
100 # no danger + capsule
101 if (next_x, next_y) in capsules:
102     features["eats-food"] = 1.5
103 # eat scared ghost
104 for g in ghosts:
105     if g[0] == next_x and g[1] == next_y and ghostState[ghosts.index(g)]>0:
106         features["eats-food"] = 2
```

### 3-1 DQN

#### Result

完整比較結果寫在 4-1 Compare the performance of every method

```
D:\Download\HW3\HW3\DQN>python pacman.py -p PacmanDQN -n 25 -x 20 -l smallClassic
Started Pacman DQN algorithm
Model has been trained
Episode no = 1; won: True; Q(s,a) = 194.3805343864543; reward = 716.0; and epsilon = 0.0
Episode no = 2; won: True; Q(s,a) = 238.13077796364138; reward = 657.0; and epsilon = 0.0
Episode no = 3; won: True; Q(s,a) = 221.15298305427393; reward = 779.0; and epsilon = 0.0
Episode no = 4; won: False; Q(s,a) = 225.0778003949599; reward = -7.0; and epsilon = 0.0
Episode no = 5; won: True; Q(s,a) = 227.44598633541997; reward = 814.0; and epsilon = 0.0
Episode no = 6; won: False; Q(s,a) = 223.03364716480195; reward = 97.0; and epsilon = 0.0
Episode no = 7; won: True; Q(s,a) = 213.58284060658954; reward = 701.0; and epsilon = 0.0
Episode no = 8; won: True; Q(s,a) = 205.1219826244877; reward = 678.0; and epsilon = 0.0
Episode no = 9; won: True; Q(s,a) = 218.921194447013; reward = 721.0; and epsilon = 0.0
Episode no = 10; won: True; Q(s,a) = 217.88522852700507; reward = 788.0; and epsilon = 0.0
Episode no = 11; won: True; Q(s,a) = 208.02762309193636; reward = 693.0; and epsilon = 0.0
Episode no = 12; won: True; Q(s,a) = 219.65331375528075; reward = 650.0; and epsilon = 0.0
Episode no = 13; won: True; Q(s,a) = 210.21357982584308; reward = 779.0; and epsilon = 0.0
Episode no = 14; won: True; Q(s,a) = 215.0678586673732; reward = 708.0; and epsilon = 0.0
Episode no = 15; won: True; Q(s,a) = 224.26182294144573; reward = 520.0; and epsilon = 0.0
Episode no = 16; won: True; Q(s,a) = 221.43914575178968; reward = 753.0; and epsilon = 0.0
Episode no = 17; won: True; Q(s,a) = 209.51981341750798; reward = 669.0; and epsilon = 0.0
Episode no = 18; won: True; Q(s,a) = 214.9876773823759; reward = 718.0; and epsilon = 0.0
Episode no = 19; won: True; Q(s,a) = 199.49614793395088; reward = 629.0; and epsilon = 0.0
Episode no = 20; won: True; Q(s,a) = 218.8541459685461; reward = 813.0; and epsilon = 0.0
Pacman emerges victorious! Score: 1777
Episode no = 21; won: True; Q(s,a) = 211.38800653322068; reward = 814.0; and epsilon = 0.0
Pacman emerges victorious! Score: 1567
Episode no = 22; won: True; Q(s,a) = 210.50210546235883; reward = 753.0; and epsilon = 0.0
Pacman emerges victorious! Score: 1328
Episode no = 23; won: True; Q(s,a) = 216.18152690167653; reward = 652.0; and epsilon = 0.0
Pacman emerges victorious! Score: 1360
Episode no = 24; won: True; Q(s,a) = 210.92649252833226; reward = 695.0; and epsilon = 0.0
Pacman emerges victorious! Score: 1555
Episode no = 25; won: True; Q(s,a) = 211.68175285163506; reward = 741.0; and epsilon = 0.0
Average Score: 1517.4
Scores: 1777.0, 1567.0, 1328.0, 1360.0, 1555.0
Win Rate: 5/5 (1.00)
Record: Win, Win, Win, Win, Win
```

#### Questions

1. What is the difference between On-policy and Off-policy

當跟環境互動的 agent 與訓練的 agent 是同一個，也就是邊做邊學的話，屬於 On-policy。如果訓練的 agent 與跟環境互動的 agent 不同，而是看別人互動來學習的話，即為 Off-policy。

2. Briefly explain value-based, policy-based and Actor-Critic. Also, describe the value function  $V^\pi(S)$

policy-based 是訓練一個負責執行動作的 neural network ~ actor，當輸入 observation 後會輸出各個 action 的機率。

value-based 則是訓練一個對目前狀況給與評價的 critic，也就是給定某個 actor  $\pi$ ，然後衡量  $\pi$  的好壞程度。而 State value function ~  $V^\pi(S)$  即為一種 critic。 $V^\pi(S)$  的值是給定某 actor  $\pi$  並看到某個 observation state  $s$  後，評估到遊戲結束會得到的期望值。可以用 Monte-Carlo 或是 Temporal-difference 來評估  $V^\pi(S)$ 。

Actor-Critic 就是結合上述兩種，它會先有一個 actor  $\pi$ ， $\pi$  會與環境互動並取得很多資料，接著 critic 會計算  $V^\pi(S)$ ，再基於  $V^\pi(S)$  更新  $\pi$  成  $\pi'$ ，然後繼續跟環境互動，一直進行互動、評估、更新的循環。

3. What is the difference between Monte-Carlo (MC) based approach and Temporal-difference (TD) approach for estimating  $V^\pi(S)$

MC 是讓 critic 觀察 actor  $\pi$  的行為，假設 actor 做了 action  $a$ ，critic 就會估計出  $V^\pi(S_a)$ ，而這個值應該要跟最後結束時真正的 reward  $G_a$  越接近越好。

TD 則是只讓 critic 觀察 actor  $\pi$  一小段的行為，假設在 state  $s_t$  時， $\pi$  做了 action  $a_t$  後會得到 reward  $r_t$ ，那麼就表示  $V^\pi(S_t) + r_t = V^\pi(S_{t+1})$ 。也就是說  $V^\pi(S_{t+1}) - V^\pi(S_t)$  要越接近  $r_t$  越好。

兩者的差異為，MC 是考慮累計到最後的總 reward  $G$ ，而 TD 是考慮單一個 reward  $r$ 。因此 MC 估計出的 variance 較大，但是 TD 的結果會有 bias。

4. Describe State-action value function  $Q^\pi(s, a)$  and the relationship between  $V^\pi(S)$  in Q-learning.

$Q^\pi(s, a)$  是輸入 state-action pair 後，估計出在 state  $s$  下做 action  $a$  可能的 reward。

其實從參數就可以看出  $V^\pi(S)$  與  $Q^\pi(s, a)$  的差異，前者只考慮 state，而後者同時考慮了 state 跟 action。

5. Describe following tips Target Network, Exploration and Replay Buffer using in Q-learning.

Target Network: Q-learning 是要使  $Q^\pi(s_t, a_t) = r_t + Q^\pi(s_{t+1}, \pi(s_{t+1}))$ ，但是這個  $\text{target} \sim r_t + Q^\pi(s_{t+1}, \pi(s_{t+1}))$  是會不斷更新變動的，會導致不好訓練，因此訓練時可以把  $Q^\pi(s_{t+1}, \pi(s_{t+1}))$  固定住，在訓練過程中不去更新它的參數，這樣他就可以產生出固定的 target，而這個就是 Target Network。

Exploration: 因為 Q-learning 會取該 state 底下 Q value 最大的 action，問題是一開始大家都是 0，但是只要某 action 被挑中並且它的 Q value 變正，則之後永遠都選這個 action，因為只有它是正的其餘都是 0。因此就要用 Exploration，有時隨機挑出 action 而非基於 policy。

Replay Buffer: 當 policy  $\pi$  與環境互動後，可將每一步的( $s_t, a_t, r_t, s_{t+1}$ ) 存進 Replay Buffer。好處是 Replay Buffer 可以存放不同 policy  $\pi$  互動的資料，這樣可以提升訓練資料的 diversity。

6. Explain what is different between DQN and Q-learning

傳統 Q-learning 會建一個 Q 表來存某 state-action pair 的  $Q\pi(s, a)$ ，而 DQN 是使用神經網絡來估 Q 值。

#### 4-1 Compare the performance of every method

##### 1. Pacman

以下的結果都是在 map layout= smallClassic，並採用 fixRandomSeed 下，玩 10 場遊戲的結果。不過因為 Approximate Q-learning 需要訓練 episodes，所以玩的不是同一場遊戲。

- Minimax (depth = 4)

平均得分: 710.2

最高分: 1392

最低分: -439

獲勝率: 6/10

```
D:\Download\HW3\HW3\Adversarial_search>python pacman.py -p MinimaxAgent -n 10 -l smallClassic -f --fixRandomSeed -a depth=4 -q
Pacman died! Score: 27
Pacman emerges victorious! Score: 1339
Pacman emerges victorious! Score: 1107
Pacman emerges victorious! Score: 1328
Pacman died! Score: -439
Pacman emerges victorious! Score: 1260
Pacman emerges victorious! Score: 1274
Pacman died! Score: 198
Pacman died! Score: -384
Pacman emerges victorious! Score: 1392
Average Score: 710.2
Scores: 27.0, 1339.0, 1107.0, 1328.0, -439.0, 1260.0, 1274.0, 198.0, -384.0, 1392.0
Win Rate: 6/10 (0.60)
Record: Loss, Win, Win, Win, Loss, Win, Win, Loss, Loss, Win
```

- Expectimax (depth = 4)

平均得分: 1116.6

最高分: 1715

最低分: 384

獲勝率: 7/10

```
D:\Download\HW3\HW3\Adversarial_search>python pacman.py -p ExpectimaxAgent -n 10 -l smallClassic -f --fixRandomSeed -a depth=4 -q
Pacman died! Score: 384
Pacman emerges victorious! Score: 1255
Pacman emerges victorious! Score: 1130
Pacman died! Score: 449
Pacman emerges victorious! Score: 1715
Pacman died! Score: 668
Pacman emerges victorious! Score: 1254
Pacman emerges victorious! Score: 1754
Pacman emerges victorious! Score: 1385
Pacman emerges victorious! Score: 1172
Average Score: 1116.6
Scores: 384.0, 1255.0, 1130.0, 449.0, 1715.0, 668.0, 1254.0, 1754.0, 1385.0, 1172.0
Win Rate: 7/10 (0.70)
Record: Loss, Win, Win, Loss, Win, Loss, Win, Win, Win, Win
```

- Approximate Q-learning (training episode = 500)

平均得分: 1017.1

最高分: 1372

最低分: 46

獲勝率: 9/10

```
Training Done (turning off epsilon and alpha)
-----
Pacman emerges victorious! Score: 974
Pacman died! Score: 46
Pacman emerges victorious! Score: 1372
Pacman emerges victorious! Score: 1149
Pacman emerges victorious! Score: 955
Pacman emerges victorious! Score: 1176
Pacman emerges victorious! Score: 1178
Pacman emerges victorious! Score: 1175
Pacman emerges victorious! Score: 1174
Pacman emerges victorious! Score: 972
Average Score: 1017.1
Scores:      974.0, 46.0, 1372.0, 1149.0, 955.0, 1176.0, 1178.0, 1175.0, 1174.0, 972.0
Win Rate:    9/10 (0.90)
Record:      Win, Loss, Win, Win, Win, Win, Win, Win, Win, Win
```

- DQN

平均得分: 1313.2

最高分: 1760

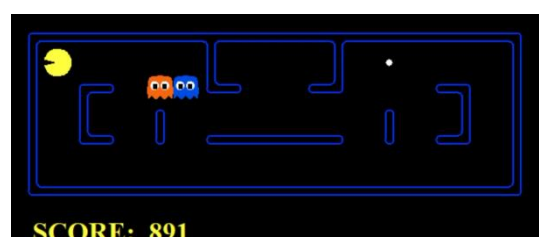
最低分: 80

獲勝率: 8/10

```
Average Score: 1313.2
Scores:      1328.0, 1731.0, 1760.0, 1349.0, 286.0, 1751.0, 80.0, 1753.0, 1532.0, 1562.0
Win Rate:    8/10 (0.80)
Record:      Win, Win, Win, Win, Loss, Win, Loss, Win, Win, Win
```

以整體來說，Minimax 表現最差。

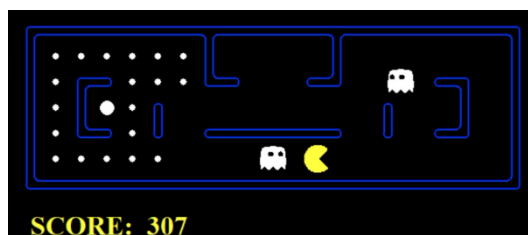
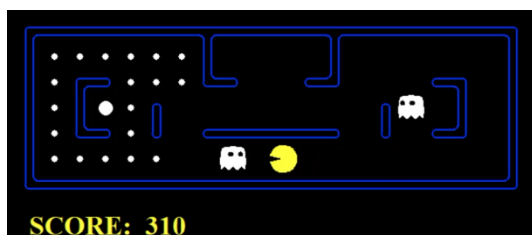
而 Expectimax 比我想像中的高分，所以我就去觀察小精靈的行為，並發現雖然 Expectimax 不會特別積極去吃 capsule，但是一旦他吃到 capsule 就會很積極的去吃鬼。此外，我還發現由於 depth 的限制，如果 Expectimax 小精靈距離食物太遠時，他就會放棄不動，直到有鬼來追他。如下圖所示，從 1044 扣到 891 他都不動。



但是光  $\text{depth} = 4$  小精靈每步就要想停久的，所以我就沒嘗試增加更多的 depth。

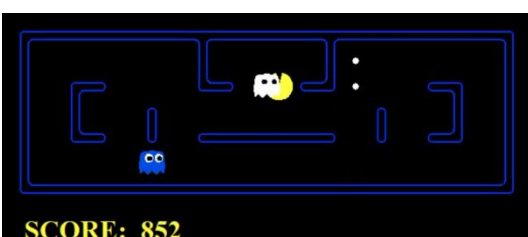
Approximate Q-learning 中我為了增加存活率，所以小精靈的行為相對保守，他雖然會很積極的去吃 capsule 來防身，但是當小精靈遇到被嚇到變成白色的鬼時，他竟然會回過身逃跑(如下圖所示)。





而他只有在碰巧可以吃到鬼時才會順便吃鬼。

DQN 就相對積極很多，他不但會去吃 capsule，而且含會特別繞路去吃被嚇到的鬼(如下圖所示)。



## 2. GridWorld

以下的結果都是玩 100 場遊戲，分無 noise 與 noise = 0.2 兩個版本。

- Value Iteration (100 iterations)

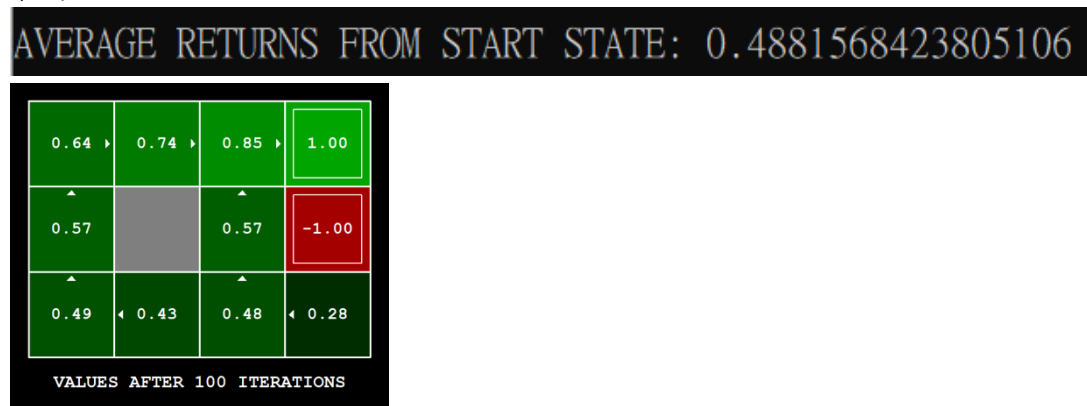
### 1. 無 noise

平均 return: 0.5904



### 2. 有 noise

平均 return: 0.488





- Q-learning without exploration

- 無 noise

平均 return: 0.5650

AVERAGE RETURNS FROM START STATE: 0.5650789198099183



- 有 noise

平均 return: 0.2291

AVERAGE RETURNS FROM START STATE: 0.2291431122374289



- Q-learning with epsilon-greedy action selection (epsilon = 0.5)

- 無 noise

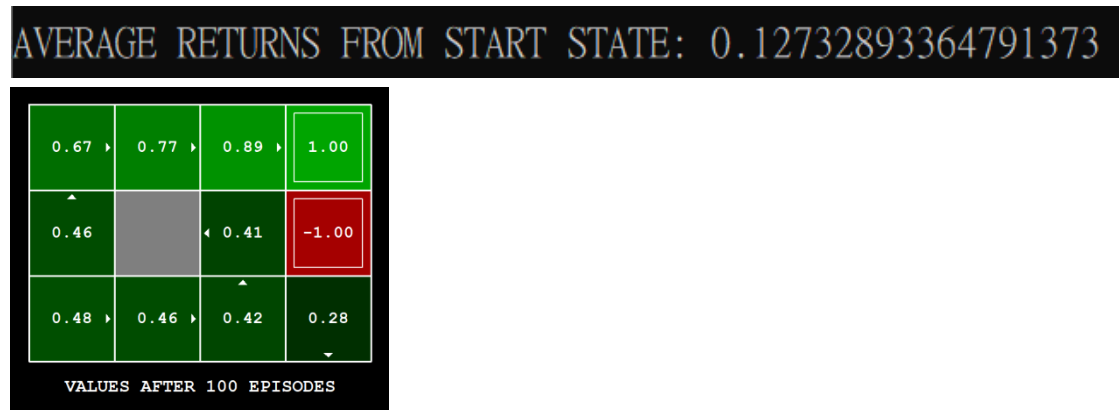
平均 return: 0.3315

AVERAGE RETURNS FROM START STATE: 0.3315787456372349



## 2. 有 noise

平均 return: 0.1273



以 return 來說，Value Iteration 表現最好，因為他已經有參照標準了。沒有 exploration 的 Q-learning 可以從他的 Value 表看出，他基本上有一條習慣走的路，尤其在無 noise 時更為明顯，因為他完全沒有可能亂走，所以除了走過的那條路之外其他的值都是 0。不過兩次實驗因為第一個 episode 走的路不同，所以兩條路線有落差(無 noise 是先往右再往上；有 noise 是先往上再往右)。而 epsilon-greedy 的結果最差，這是因為他有 0.5 的機率會去探索，而不是順著最高分的走法，這就有可能導致他的表現變差。