Code Explanation:

Part 1-1: Minimax Search:

這部分程式碼的解釋已包含在截圖裡的註釋中

class MinimaxAgent(MultiAgentSearchAgent):

    """

    Your minimax agent (par1-1)

    """

    def getAction(self, gameState):

```
# Begin your code
numGhosts = gameState.getNumAgents() - 1
# pacman agent wants max value, so call maxValue(), not minValue()
return self.maxValue(gameState, 1, numGhosts)

util.raiseNotDefined()
# End your code
```

```
def maxValue(self, gameState, depth, numGhosts):
    # Check if in terminal state
    if gameState.isWin() or gameState.isLose():
        return self.evaluationFunction(gameState)
    # maxVal stores the max score we can take
    maxVal = float("-inf")
    # bestAction stores the action we do when we have the highest score
    bestAction = Directions.STOP
    # iterate through every action taken from gameState.getLegalActions(0)
    # (0) means the agentIndex=0 (pacman's agentIndex=0)
    for action in gameState.getLegalActions(0):
        # get min value from minValue because we assume that the ghosts take best actions
        # this means that the ghosts wants us to take the lowest scores
        ##########################################################################################
        # parameters explanation for self.minValue(gameState.getNextState(0, action), depth, numGhosts, 1)
        # gameState.getNextState(0, action): nextState of agentIndex=0, action=action
        # depth=depth (Why don't we pass depth+1? It's because
        # "A single level of the search is considered to be one pacman move and all the ghosts' responses"
        # numGhosts=numGhosts
        # pass 1 to agentIndex because we want to run the first ghost
        ##########################################################################################
        val = self.minValue(gameState.getNextState(0, action), depth, numGhosts, 1)
        if val > maxVal:
            maxVal = val
            bestAction = action
    # if depth > 1, then return max value we can get
    # else, depth==1, we return bestAction to def getAction(self, gameState):
    if depth > 1:
        return maxVal
    return bestAction
```

```python
def minValue(self, gameState, depth, numGhosts, agentIndex):
    # Check if in terminal state
    if gameState.isWin() or gameState.isLose():
        return self.evaluationFunction(gameState)
    # minVal stores the min score we can take
    minVal = float("inf")
    # if agentIndex == numGhosts check that whether we're going through the last ghost
    if agentIndex == numGhosts:
        # if depth == self.depth check that whether we're at the last layer
        if depth == self.depth:
            for action in gameState.getLegalActions(agentIndex):
                minVal = min(minVal, self.evaluationFunction(gameState.getNextState(agentIndex, action)))
        else:
            # we're going to next layer, so depth=depth+1
            # we'll start from the pacman agent, so calling maxValue(), not minValue()
            for action in gameState.getLegalActions(agentIndex):
                minVal = min(minVal, self.maxValue(gameState.getNextState(agentIndex, action), depth+1, numGhosts))
    else:
        # we want to go through next ghost at the same layer, so depth=depth, agentIndex=agentIndex+1
        for action in gameState.getLegalActions(agentIndex):
            minVal = min(minVal, self.minValue(gameState.getNextState(agentIndex, action), depth, numGhosts, agentIndex+1))
    return minVal
```

Part 1-2: Expectimax Search:

這部分程式碼的解釋已包含在截圖裡的註釋中

class ExpectimaxAgent(MultiAgentSearchAgent):

"""

Your expectimax agent (part1-2)

"""

def getAction(self, gameState):

```
# Begin your code
numGhosts = gameState.getNumAgents() - 1
return self.maxValue(gameState, 1, numGhosts)

util.raiseNotDefined()
# End your code
```

```
# this maxValue() part is almost the same as the one in class MinimaxAgent(MultiAgentSearchAgent)
def maxValue(self, gameState, depth, numGhosts):
    if gameState.isWin() or gameState.isLose():
        return self.evaluationFunction(gameState)
    maxVal = float("-inf")
    bestAction = Directions.STOP
    for action in gameState.getLegalActions(0):
        # we call expectValue(), not minValue() because not all situations will be deterministic
        val = self.expectValue(gameState.getNextState(0, action), depth, numGhosts, 1)
        if val > maxVal:
            maxVal = val
            bestAction = action
    if depth > 1:
        return maxVal
    return bestAction
```

```
# this expectValue() part is almost the same as minValue() in class MinimaxAgent(MultiAgentSearchAgent)
def expectValue(self, gameState, depth, numGhosts, agentIndex):
    if gameState.isWin() or gameState.isLose():
        return self.evaluationFunction(gameState)
    # initial expected value to 0.0, because we still want the pacman agent to lower scores
    expectVal = 0.0
    # we assume that we have the same probability for every action, so probability=1/(the number of actions)
    prob = 1.0 / len(gameState.getLegalActions(agentIndex))
    # expectVal += prob * (...) means that we sum up all values for next state with timing each probability
    if agentIndex == numGhosts:
        if depth == self.depth:
            for action in gameState.getLegalActions(agentIndex):
                expectVal += prob * self.evaluationFunction(gameState.getNextState(agentIndex, action))
        else:
            for action in gameState.getLegalActions(agentIndex):
                expectVal += prob * self.maxValue(gameState.getNextState(agentIndex, action), depth + 1, numGhosts)
    else:
        for action in gameState.getLegalActions(agentIndex):
            expectVal += prob * self.expectValue(gameState.getNextState(agentIndex, action), depth, numGhosts,
                                                 agentIndex + 1)
    return expectVal
```

Part 1-3: Evaluation Function:

這部分程式碼的解釋已包含在截圖裡的註釋中

```
def betterEvaluationFunction(currentGameState):
    """
    Your extreme ghost-hunting, pellet-nabbing, food-gobbling, unstoppable
    evaluation function (part1-3).

    DESCRIPTION: <write something here so we know what you did>
    """
    "*** YOUR CODE HERE ***"
    # Begin your code
    # get pacman position
    newPos = currentGameState.getPacmanPosition()
    # get all food situation
    ############################################################
    # def getFood(self):
    #     """
    #     Returns a Grid of boolean food indicator variables.
    #
    #     Grids can be accessed via list notation, so to check
    #     if there is food at (x,y), just call
    #
    #     currentFood = state.getFood()
    #     if currentFood[x][y] == True: ...
    #     """
    #     return self.data.food
    ############################################################
    newFood = currentGameState.getFood()
    # get ghost states
    newGhostStates = currentGameState.getGhostStates()
```

```
# get scared times of the ghosts
newScaredTimes = [ghostState.scaredTimer for ghostState in newGhostStates]

# get the minimum scared time among all the ghosts
minScaredTime = min(newScaredTimes)
# get the minimum distance between pacman and all ghosts
closestGhostDist = min(manhattanDistance(newPos, ghostState.configuration.pos) for ghostState in newGhostStates)
# initialize minimum distance between pacman and all foods to infinity
closestFoodDist = float("inf")
remainingFoodCount = 0
# if there are no foods remaining, then set closestFoodDist to 0
if not newFood:
    closestFoodDist = 0
else:
    # newFood is of type "Grid", so we can call newFood.width, newFood.height to iterate through every place in the screen
    for x in range(newFood.width):
        for y in range(newFood.height):
            # check if there's food at (x, y)
            if currentGameState.hasFood(x, y):
                closestFoodDist = min(closestFoodDist, manhattanDistance(newPos, [x, y]))
                remainingFoodCount += 1
```

```python
# the higher ghostScore is, the better situation the pacman is in
ghostScore = 0
# We / (closestGhostDist+1) because the more the distance is, it is less important
# (closestGhostDist+1) because we don't want denominator to be 0
if minScaredTime == 0:
    # if some ghosts are not scared, we are in bad situation, so the value is minus
    ghostScore = -2.5 / (closestGhostDist+1)
else:
    # if all ghosts are scared, then our situation is better, so the value is plus
    ghostScore = 1 / (closestGhostDist+1)
# the higher currentGameState.getScore() is, the better situation the pacman is in
# the higher minScaredTime is, the better situation the pacman is in
# the higher remainingFoodCount is, the worse situation the pacman is in because remaining foods more,
# so the sign is minus
# the weights are determined by experiments
return 0.8*currentGameState.getScore()+0.6/(closestGhostDist+1)+0.7*minScaredTime-0.5*remainingFoodCount+ghostScore
# End your code
```

Part 2-1: Value Iteration:

這部分程式碼的解釋已包含在截圖裡的註釋中

class ValueIterationAgent(ValueEstimationAgent):

```
def runValueIteration(self):
    "*** YOUR CODE HERE ***"
    # Begin your code
    # iterate self.iterations times
    for i in range(self.iterations):
        # get a copy from self.values because
        #########################################################################################
        # Use the “batch” version of value iteration where each vector V[K] is
        # computed from a fixed vector V[K-1], not the “online” version
        # where one single weight vector is updated in place.
        # This means that when a state’s value is updated in iteration k
        # based on the values of its successor states, the successor state values used in the
        # value update computation should be those from iteration k-1
        ##########################################################################################
        copyValues = self.values.copy()
        # iterate through all states
        for state in self.mdp.getStates():
            # check for terminal
            if self.mdp.isTerminal(state):
                copyValues[state] = 0
                continue
            maxQValue = float("-inf")
            # iterate through all possible actions
            for action in self.mdp.getPossibleActions(state):
                # we want the maximum QValue from all possible actions
                maxQValue = max(maxQValue, self.getQValue(state, action))
            copyValues[state] = maxQValue
        # update values in iteration k based on the values in iteration k-1
        self.values = copyValues
    # End your code
```

```
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
    QValue = 0
    # iterate through all possible nextStates, prob means the probability to transit to that nextState
    for nextState, prob in self.mdp.getTransitionStatesAndProbs(state, action):
        QValue += prob * (self.mdp.getReward(state, action, nextState) + self.discount * self.values[nextState])
    return QValue
    # End your code
```

```python
def computeActionFromValues(self, state):
    """
      The policy is the best action in the given state
      according to the values currently stored in self.values.

      You may break ties any way you see fit.  Note that if
      there are no legal actions, which is the case at the
      terminal state, you should return None.
    """
    "*** YOUR CODE HERE ***"
    # Begin your code
    #check for terminal
    if self.mdp.isTerminal(state):
        return None
    # initialize values to a dictionary
    values = util.Counter()
    # iterate through all possible actions and store the QValue in values
    for action in self.mdp.getPossibleActions(state):
        values[action] = self.getQValue(state, action)
    # argMax returns the key with the highest value
    # find the best action with the highest value
    return values.argMax()
    # End your code
```

上面的程式中，計算了下面 value iteration 的公式

$$V^*(s) = \max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^*(s')]$$

Part 2-2: Q-learning:

class QLearningAgent(ReinforcementAgent):

```
def __init__(self, **args):
    "You can initialize Q-values here..."
    ReinforcementAgent.__init__(self, **args)

    "*** YOUR CODE HERE ***"
    # Begin your code
    # initialize self.QValues to a dictionary
    self.QValues = util.Counter()
    # End your code
```

```
def getQValue(self, state, action):
    """
      Returns Q(state,action)
      Should return 0.0 if we have never seen a state
      or the Q node value otherwise
    """
    "*** YOUR CODE HERE ***"
    # Begin your code
    return self.QValues[(state, action)]
    # End your code
```

```
def computeValueFromQValues(self, state):
    """
      Returns max_action Q(state,action)
      where the max is over legal actions.  Note that if
      there are no legal actions, which is the case at the
      terminal state, you should return a value of 0.0.
    """
    "*** YOUR CODE HERE ***"
    # Begin your code
    # get all possible actions
    actions = self.getLegalActions(state)
    # check if no legal actions
    if not actions:
        return 0.0
    # return maximum QValue among all actions as value
    return max(self.getQValue(state, action) for action in actions)
    # End your code
```

```python
def computeActionFromQValues(self, state):
    """
    Compute the best action to take in a state.  Note that if there
    are no legal actions, which is the case at the terminal state,
    you should return None.
    """
    "*** YOUR CODE HERE ***"
    # Begin your code
    # get all legal actions
    actions = self.getLegalActions(state)
    # check if no legal actions
    if not actions:
        return None
    # store all best actions which have the same max QValue in a list
    best_actions = []
    # compute best QValue and store it in max_QValue
    max_QValue = self.computeValueFromQValues(state)
    # iterate through all legal actions
    for action in actions:
        # check if making this action can also get the best QValue
        if self.getQValue(state, action) == max_QValue:
            best_actions.append(action)
    # randomly choose a action among all best actions
    return random.choice(best_actions)
    # End your code
```

```python
def update(self, state, action, nextState, reward):
    """
    The parent class calls this to observe a
    state = action => nextState and reward transition.
    You should do your Q-Value update here

    NOTE: You should never call this function,
    it will be called on your behalf
    """
    "*** YOUR CODE HERE ***"
    # Begin your code
    # update QValues due to the formula
    self.QValues[(state, action)] = (1-self.alpha)*self.getQValue(state, action)+\
                                    self.alpha*(reward+self.discount*self.computeValueFromQValues(nextState))
    # End your code
```

Update formula for Q-Learning:

$$q^{new}(s,a) = (1-\alpha)\underbrace{q(s,a)}_{\text{old value}} + \alpha\overbrace{\left(R_{t+1} + \gamma \max_{a'} q(s',a')\right)}^{\text{learned value}}$$

Part 2-3: epsilon-greedy action selection:

```python
def getAction(self, state):
    """
      Compute the action to take in the current state.  With
      probability self.epsilon, we should take a random action and
      take the best policy action otherwise.  Note that if there are
      no legal actions, which is the case at the terminal state, you
      should choose None as the action.

      HINT: You might want to use util.flipCoin(prob)
      HINT: To pick randomly from a list, use random.choice(list)
    """
    # Pick Action
    # get all legal actions
    legalActions = self.getLegalActions(state)
    action = None
    "*** YOUR CODE HERE ***"
    # Begin your code
    # check if no legal actions
    if not legalActions:
        return None
```

```python
###############################
# def flipCoin(p):
#     r = random.random()
#     return r < p
###############################
# check if the random value if smaller than self.epsilon
# if random value is smaller, then explore
# else, exploit
if util.flipCoin(self.epsilon):
    # explore, so randomly choose an action among all legal actions
    action = random.choice(legalActions)
else:
    # exploit, so choose the best action which has the maximum QValue
    action = self.computeActionFromQValues(state)
return action
# End your code
```

在 epsilon-greedy action selection 中，會遵循下圖中的規則，而 epsilon 會從大到小，因此會從大部分 exploration 轉變到大部分 exploitation

```python
if random_num > epsilon:
# choose action via exploitation
else:
# choose action via exploration
```

Part 2-4: Approximate Q-learning:

class ApproximateQAgent(PacmanQAgent):

```
def getQValue(self, state, action):
    """
      Should return Q(state,action) = w * featureVector
      where * is the dotProduct operator
    """
    "*** YOUR CODE HERE ***"
    # Begin your code
    # get weights and feature
    features = self.featExtractor.getFeatures(state, action)
    # return the summation of all features[i]*weights[i]
    return sum(self.weights[feature]*value for feature, value in features.items())
    # End your code
```

因為在 SARSA 中，Q-table 站的空間太大，所以使用 Feature Approximation。他的想法是 learn a reward function as a linear combination of features，並使用下列程式計算 QValue。

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

```
def update(self, state, action, nextState, reward):
    """
      Should update your weights based on transition
    """
    "*** YOUR CODE HERE ***"
    # Begin your code
    # get features
    features = self.featExtractor.getFeatures(state, action)
    # calculate correction due to the formula
    correction = reward + self.discount * self.getValue(nextState) - self.getQValue(state, action)
    # iterate through all features
    for feature, value in features.items():
        # update weights due to the formula
        self.weights[feature] += self.alpha * correction * value
    # End your code
```

在 Approximate Q-Learning 中，會使用 gradient descent 來更新 weights，correction 代表想達到的預計值與現在狀態下的差值，update 的公式如下

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

Discussion:

I. Value Iteration VS Q-Learning:

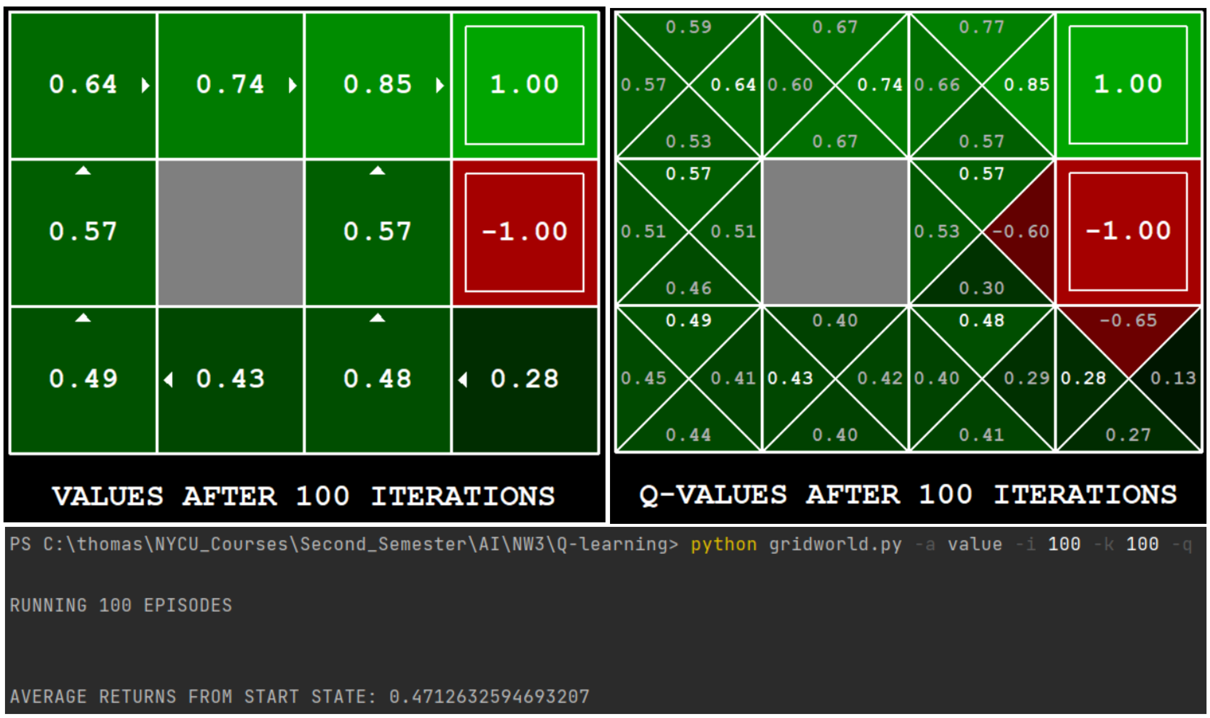
A. Value Iteration:

It is a model based-learning where we know the reward for a state and action pair, and the transitions for every action from a state. It is used for deterministic questions.

The results for command “python gridworld.py -a value -i 100 -k 100 -q” are the following pictures.

// (-i 100) means 100 rounds of value iteration

// (-k 100) means 100 episodes of execution of MDP

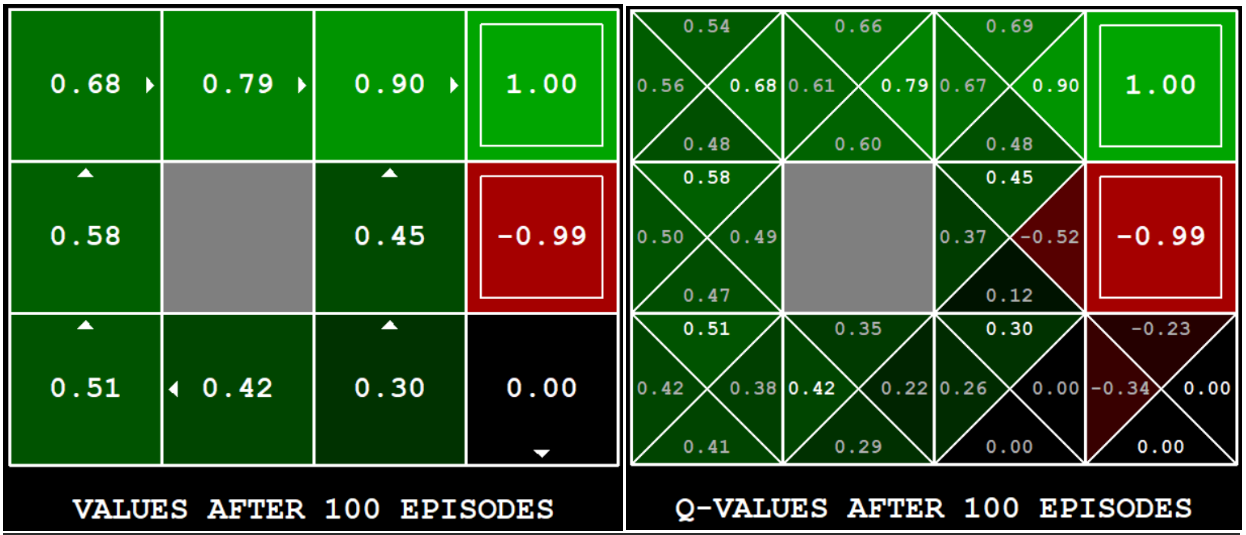


B. Q-Learning:

In Q-learning, the agent does not know state transition probabilities or rewards. The agent only discovers that there is a reward for going from one state to another via a given action when it does so and receives a reward. The transition probability is similar to that, too. Therefore, it is model-free.

The results for command “python gridworld.py -a q -k 100” are the following pictures.

// (-k 100) means 100 episodes of execution of MDP




```
PS C:\thomas\NYCU_Courses\Second_Semester\AI\NW3\Q-learning> python gridworld.py -a q -k 100 -q

RUNNING 100 EPISODES

AVERAGE RETURNS FROM START STATE: 0.3227089654151382
```

II. Evaluation Function in Expectimax Search:

A. When better = scoreEvaluationFunction, the results for command "python autograder.py" are the following pictures.

```
Average Score: -109.73
Scores:        833.0, 1285.0, -260.0, -487.0, -178.0, -139.0, -584.0, 779.0, -494.0, 390.
51.0, 496.0, 386.0, 666.0, 154.0, 992.0, -467.0, -1126.0, 50.0, -519.0, -115.0, -117.0, -
477.0, -159.0, -207.0, -533.0, 675.0, 158.0, 426.0, -157.0, -401.0, 902.0, -387.0, 888.0,
51.0, -178.0, 76.0, 711.0, -4528.0, -257.0, -64.0, -315.0, -1024.0, -3.0, -243.0, -709.0,
5.0, -5112.0, 174.0, 1015.0, -331.0, 450.0
Win Rate:      61/100 (0.61)
```

B. When better = betterEvaluationFunction, the results for command "python autograder.py" are the following pictures. The win rate and average score are higher than previous evaluation function.

```
Average Score: 968.63
Scores:        1166.0, 876.0, 906.0, 1040.0, 1128.0, 1208.0, 914.0, 990.0,
2.0, 1157.0, 998.0, 998.0, 422.0, 770.0, 580.0, 933.0, 1082.0, 932.0, 894.
.0, 881.0, 1056.0, 707.0, 1200.0, 738.0, 1146.0, 1038.0, 802.0, 994.0, 120
1200.0, 1062.0, 900.0, 1094.0, 1013.0, 1186.0, 810.0, 654.0, 696.0, 977.0,
.0, 1018.0, 952.0, 1148.0
Win Rate:      100/100 (1.00)
```

III. Comparison between Expectimax Search, Approximate Q-Learning, and DQN:

I compare three methods by using these parameters: -n 100 (100 games) and -l smallClassic (smallClassic layout).

A. Expectimax Search:

The result for command “python pacman.py -p ExpectimaxAgent -l smallClassic -a depth=3 -q -n 100” is the following picture.

```
Average Score: 643.76
Scores:        795.0, 248.0, 1313.0, 26.0, 1682.0, 83.0, 62.0, 339.0, 104
 1214.0, -194.0, 1297.0, 1409.0, 1486.0, -266.0, 996.0, 141.0, 266.0, -21
0, 1268.0, 1034.0, 44.0, 188.0, 1050.0, 1290.0, 1365.0, 1430.0, 1233.0, -
255.0, -117.0, 64.0, -463.0, 776.0, 1226.0, 1500.0, 102.0, 1085.0, 823.0,
1021.0, 1648.0, -58.0, 1576.0
Win Rate:      54/100 (0.54)
```

The win rate for Expectimax Search was not high and the computing time for this method was very long. If the depth=4, then the computing time will be too long to run the game.

B. Approximate Q-Learning:

The SimpleExtractor function and its feature description is written as the following.

```python
def getFeatures(self, state, action):
    # extract the grid of food and wall locations and get the ghost locations
    food = state.getFood()
    walls = state.getWalls()
    ghosts = state.getGhostPositions()

    features = util.Counter()

    features["bias"] = 1.0

    # compute the location of pacman after he takes the action
    x, y = state.getPacmanPosition()
    dx, dy = Actions.directionToVector(action)
    next_x, next_y = int(x + dx), int(y + dy)

    # count the number of ghosts 1-step away
    features["#-of-ghosts-1-step-away"] = sum((next_x, next_y) in Actions.getLegalNeighbors(g, walls) for g in ghosts)

    # if there is no danger of ghosts then add the food feature
    if not features["#-of-ghosts-1-step-away"] and food[next_x][next_y]:
        features["eats-food"] = 1.0

    dist = closestFood((next_x, next_y), food, walls)
    if dist is not None:
        # make the distance a number less than one otherwise the update
        # will diverge wildly
        features["closest-food"] = float(dist) / (walls.width * walls.height)
    features.divideAll(10.0)
    return features
```

PacMan features from lab

- "bias" always 1.0
- "#-of-ghosts-1-step-away" the number of ghosts (regardless of whether they are safe or dangerous) that are 1 step away from Pac-Man
- "closest-food" the distance in Pac-Man steps to the closest food pellet (does take into account walls that may be in the way)
- "eats-food" either 1 or 0 if Pac-Man will eat a pellet of food by taking the given action in the given state

The result for command “python pacman.py -p ApproximateQAgent -a extractor=SimpleExtractor -x 50 -n 150 -l smallClassic -q” is the following picture.

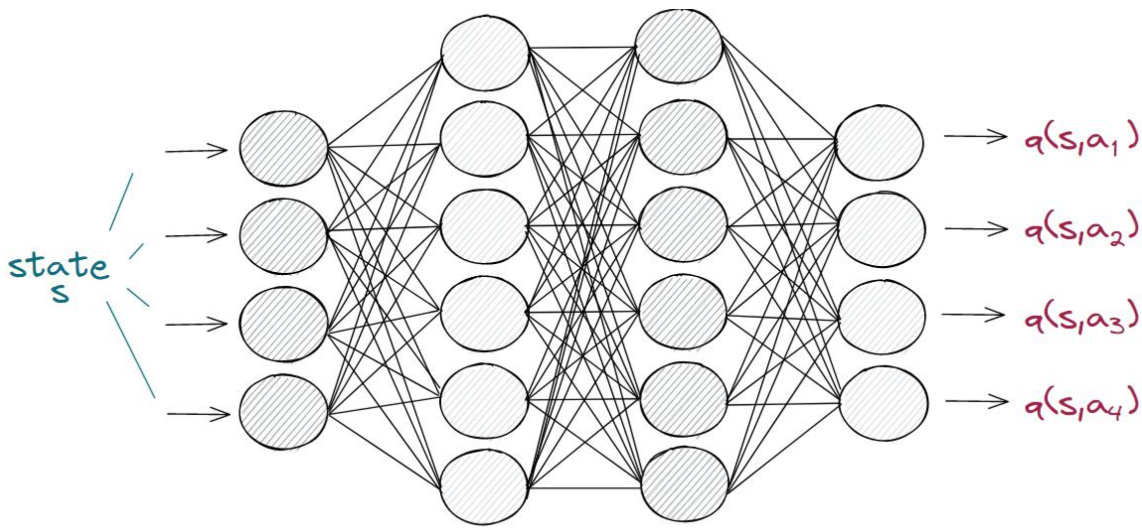
```
Scores:        975.0, 957.0, 969.0, 987.0, 971.0, 1166.0, 1158.0, 963.0, 1159.
0, 980.0, 975.0, 968.0, -143.0, 972.0, 966.0, 1341.0, 976.0, 976.0, 978.0, 982
0, 979.0, -146.0, 954.0, 967.0, -185.0, 977.0, 959.0, 976.0, -64.0, 976.0, -25
 979.0, 980.0, 966.0, 971.0, 973.0, 978.0, 962.0, 983.0, -349.0, 950.0, 987.0,
Win Rate:      88/100 (0.88)
```

The win rate for Approximate Q-Learning was high, and its computing time was much shorter than using Expectimax Search.

C. Deep Q-Learning (DQN):

While Q-Learning performs quite well in the pacman game, its performance will drop-off considerably when working in more complex and sophisticated environments. In large environment, each state in the environment would be represented by a set of pixels, and the agent may be able to take several actions from each state. The iterative process of computing and updating Q-values for each state-action pair in a large state space becomes computationally inefficient and perhaps infeasible due to the computational resources and time this may take.

Therefore, we can use DQN to solve the problem. In DQN, we have a deep neural network that accepts states from a given environment as input. For each given state input, the network outputs estimated QValues for each action that can be taken from that state. The objective of this network is to approximate the optimal Q-function. Next, the loss from the network is calculated by comparing the outputted QValues to the target QValues from the Bellman equation. After the loss is calculated, the weights within the network are updated via SGD and backpropagation to minimize the loss. This process is done over and over again for each state in the environment until we sufficiently minimize the loss and get an approximate optimal Q-function. The following is a schematic diagram for DQN.



The DQN architecture in this pacman game is as the following.

```
""" Deep Q Network """
class DQN(nn.Module):
    def __init__(self, num_inputs=6, num_actions=4):
        super(DQN, self).__init__()

        self.conv1 = nn.Conv2d(num_inputs, 32, kernel_size=3, stride=1)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=2, stride=1)
        self.fc3 = nn.Linear(4352, 512)
        self.fc4 = nn.Linear(512, num_actions)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = F.relu(self.conv2(x))
        # print(x.view(x.size(0), -1).shape)
        x = F.relu(self.fc3(x.view(x.size(0), -1)))
        return self.fc4(x)
```

I train the model with 10000 episodes. The result for command "python pacman.py -p PacmanDQN -n 10000 -x 10000 -l smallClassic" is the following picture.

```
Episode no = 9961; won: True; Q(s,a) = 238.29149839582405; reward = 678.0; and epsilon = 0.1
Episode no = 9962; won: False; Q(s,a) = 218.2335448414027; reward = -273.0; and epsilon = 0.1
Episode no = 9963; won: True; Q(s,a) = 219.87824857803213; reward = 644.0; and epsilon = 0.1
Episode no = 9964; won: False; Q(s,a) = 221.98219308381346; reward = 2.0; and epsilon = 0.1
Episode no = 9965; won: False; Q(s,a) = 226.7128961674246; reward = -9.0; and epsilon = 0.1
Episode no = 9966; won: True; Q(s,a) = 220.77536225989547; reward = 749.0; and epsilon = 0.1
Episode no = 9967; won: True; Q(s,a) = 212.53860800687738; reward = 748.0; and epsilon = 0.1
Episode no = 9968; won: False; Q(s,a) = 200.0770074414617; reward = -342.0; and epsilon = 0.1
Episode no = 9969; won: True; Q(s,a) = 227.8366021347603; reward = 786.0; and epsilon = 0.1
Episode no = 9970; won: False; Q(s,a) = 227.5500288744018; reward = 63.0; and epsilon = 0.1
Episode no = 9971; won: True; Q(s,a) = 219.86522775681058; reward = 749.0; and epsilon = 0.1
Episode no = 9972; won: True; Q(s,a) = 219.72594644937837; reward = 701.0; and epsilon = 0.1
Episode no = 9973; won: False; Q(s,a) = 222.30283907937135; reward = 25.0; and epsilon = 0.1
Episode no = 10000; won: True; Q(s,a) = 222.42646064155; reward = 823.0; and epsilon = 0.1
UPDATING target network
```

After training, I used the command "python pacman.py -p PacmanDQN -n 200 -x 100 -l smallClassic -q". The result is the following picture.

```
Episode no = 191; won: False; Q(s,a) = 232.60911673586477; reward = -46.0; and epsilon = 0.
Pacman emerges victorious! Score: 1326
Episode no = 192; won: True; Q(s,a) = 234.26560533896486; reward = 672.0; and epsilon = 0.0
Pacman emerges victorious! Score: 1746
Episode no = 193; won: True; Q(s,a) = 229.68038789218977; reward = 772.0; and epsilon = 0.0
Pacman emerges victorious! Score: 1563
Episode no = 194; won: True; Q(s,a) = 236.3426160467681; reward = 738.0; and epsilon = 0.0
Pacman emerges victorious! Score: 1737
Episode no = 195; won: True; Q(s,a) = 238.6111717796316; reward = 752.0; and epsilon = 0.0
Pacman emerges victorious! Score: 1109
Episode no = 196; won: True; Q(s,a) = 228.25633336428635; reward = 593.0; and epsilon = 0.0
Pacman emerges victorious! Score: 1365
, 1563.0, 1737.0, 1109.0, 1365.0, 1378.0, 1099.0, 1215.0
Win Rate:      85/100 (0.85)
```

The win rate for DQN was high, and its computing time was much shorter than using Expectimax Search.

IV. 心得

在這項作業中，雖然原本對 Reinforcement Learning 了解不多，但在大量查找網路資料並配合作業中程式碼後，學到了不少關於這方面的知識。作業中最困難的部份應該是要如何從這個大致上寫好的 project 中找到並正確使用各項函式，在這方面花了不少時間，其餘部分反而沒太大問題。