111550113 謝詠晴

Intro_to_AI HW3

Part 1

Part 1-1: Minimax Search (10%)

```python
class MinimaxAgent(MultiAgentSearchAgent):
    def getAction(self, gameState):
        """*** YOUR CODE HERE ***"""
        # Begin your code
        def minimax(gameState, depth, agentIndex):
            # terminal condition
            if gameState.isWin() or gameState.isLose() or (depth == self.depth):
                return self.evaluationFunction(gameState)
            actions = gameState.getLegalActions(agentIndex)
            # Pacman's move (maximize)
            if agentIndex == 0:
                bestScore = float("-inf")
                for action in actions:
                    nextState = gameState.getNextState(0, action)
                    bestScore = max(bestScore, minimax(nextState, depth, 1))
                return bestScore
            # Ghosts' move (minimize)
            else:
                bestScore = float("inf")
                for action in actions:
                    if agentIndex < gameState.getNumAgents()-1: # next is ghost
                        nextState = gameState.getNextState(agentIndex, action)
                        bestScore = min(bestScore, minimax(nextState, depth, agentIndex+1))
                    else: # last ghost
                        nextState = gameState.getNextState(agentIndex, action)
                        bestScore = min(bestScore, minimax(nextState, depth+1, 0))
                return bestScore
```

```python
        # initial setting
        actions = gameState.getLegalActions(0)
        scores = []
        for action in actions:
            nextState = gameState.getNextState(0, action)
            scores.append((action, minimax(nextState, 0, 1)))
        action, _ = max(scores, key=lambda x: x[1])
        return action
        # End your code
```

1) In minimax( ), it takes three parameters: 'gameState' representing the current state of the game, 'depth' representing the depth of the search tree and 'agentIndex' representing the index of the current character.
2) If the termination condition is met, it returns the evaluation value of the current state. If the termination condition isn't met, we retrieve the list of actions by getLegalActions().
3) If the current character is Pacman (agentIndex = 0), it iterates over each action, calculates the score of the next state, and selects the action with the highest score.

4) On the other hand, if the character is ghost, we'll need to check if it is the last ghost first to ensure we go to the correct state. And perform the similar operation like (3) but to find the action with minimum score.
5) Result:

```
*** Running MinimaxAgent on smallClassic 1 time(s).
Pacman died! Score: 84
Average Score: 84.0
Scores:        84.0
Win Rate:      0/1 (0.00)
Record:        Loss
*** Finished running MinimaxAgent on smallClassic after 0 seconds.
*** Won 0 out of 1 games. Average score: 84.000000 ***
*** PASS: test_cases\part1-1\8-pacman-game.test

### Question part1-1: 10/10 ###
```

Part 1-2: Expectimax Search (10%)

```
class ExpectimaxAgent(MultiAgentSearchAgent):
    def getAction(self, gameState):
        "*** YOUR CODE HERE ***"
        # Begin your code
        def expectimax(gameState, depth, agentIndex):
            # terminal condition
            if gameState.isWin() or gameState.isLose() or (depth == self.depth):
                return self.evaluationFunction(gameState)
            actions = gameState.getLegalActions(agentIndex)
            # Pacman's move (maximize)
            if agentIndex == 0:
                bestScore = float("-inf")
                for action in actions:
                    nextState = gameState.getNextState(0, action)
                    bestScore = max(bestScore, expectimax(nextState, depth, 1))
                return bestScore
            # Ghosts' move (minimize)
            else:
                avgScore = 0
                for action in actions:
                    if agentIndex < gameState.getNumAgents()-1: # next is ghost
                        nextState = gameState.getNextState(agentIndex, action)
                        avgScore += (expectimax(nextState, depth, agentIndex+1)/len(actions))
                    else: # last ghost
                        nextState = gameState.getNextState(agentIndex, action)
                        avgScore += (expectimax(nextState, depth+1, 0)/len(actions))
                return avgScore
```

```
        # initial setting
        actions = gameState.getLegalActions(0)
        scores = []
        for action in actions:
            nextState = gameState.getNextState(0, action)
            scores.append((action, expectimax(nextState, 0, 1)))
        action, _ = max(scores, key=lambda x: x[1])
        return action
        # End your code
```

1) In expectimax() function, the structure is quite similar to minimax().
2) The difference (line 198, 201 and 202) is that we don't assume ghost will choose the worst action for us. Instead, we assume ghost will choose random action, so we return the average score.
3) Result:

```
*** Running ExpectimaxAgent on smallClassic 1 time(s).
Pacman died! Score: 84
Average Score: 84.0
Scores:        84.0
Win Rate:      0/1 (0.00)
Record:        Loss
*** Finished running ExpectimaxAgent on smallClassic after 0 seconds.
*** Won 0 out of 1 games. Average score: 84.000000 ***
*** PASS: test_cases\part1-2\7-pacman-game.test

### Question part1-2: 10/10 ###
```

Part 2

Part 2-1: Value Iteration (10%)

```python
class ValueIterationAgent(ValueEstimationAgent):
    def runValueIteration(self):
        # Write value iteration code here
        "*** YOUR CODE HERE ***"
        # Begin your code
        for i in range(self.iterations):
            states = self.mdp.getStates()
            tmpCounter = util.Counter()
            for state in states:
                if self.mdp.isTerminal(state):
                    self.values[state] = 0
                else:
                    maxValue = float("-inf")
                    actions = self.mdp.getPossibleActions(state)
                    for action in actions:
                        QValue = self.computeQValueFromValues(state, action)
                        maxValue = max(maxValue,QValue)
                    tmpCounter[state] = maxValue
            self.values = tmpCounter
        # End your code
```

1) In each iteration, we get the states by mdp.getStates() and then declare a dictionary counter called 'tmpCounter'. For each state, we'll find the maximum Q value for each state by computeQValueFromValues() and store it to tmpCounter and update self.values in each iteration.

```python
    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
        transitionStatesAndProbs = self.mdp.getTransitionStatesAndProbs(state, action)
        for (nextState, prob) in transitionStatesAndProbs:
            reward = self.mdp.getReward(state, action, nextState)
            discount = self.discount
            QValue += prob*(reward + discount * self.values[nextState])
        return QValue
        # End your code
```

2) In computeQValueFromValues(), we use mdp.getTransitionStatesAndProbs() to get the nextState and prob and calculate the Qvalue by the below formula, where α is prob, r is reward and γ is discount.

$$Q(s,a) = \sum \alpha(r + \gamma * maxQ(s',a'))$$

```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
        else:
            QValues = util.Counter()
            actions = self.mdp.getPossibleActions(state)
            for action in actions:
                QValues[action] = self.getQValue(state, action)
            return QValues.argMax()
        # End your code
```

3) In computeActionFromValues(), we first check if the state is a terminal state. Then we iterate all possible action to find the action with maximum Q value.

4) Result:

> python gridworld.py -a value -i 5


```
Question part2-1
================

*** PASS: test_cases\part2-1\1-tinygrid.test
*** PASS: test_cases\part2-1\2-tinygrid-noisy.test
*** PASS: test_cases\part2-1\3-bridge.test
*** PASS: test_cases\part2-1\4-discountgrid.test

### Question part2-1: 10/10 ###
```

Part 2-2: Q-learning (15%)

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

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

```python
    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
```

1) In 2-1, we use the MDP model but not actually learn from the environment. Therefore, in this part, we'll learn from experience and keep updating.

```python
    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
        actions = self.getLegalActions(state)
        if len(actions) == 0:
            return 0.0
        else:
            maxValue = float("-inf")
            for action in actions:
                maxValue = max(maxValue, self.getQValue(state, action))
            return maxValue
        # End your code
```

2) In computeValueFromQValues(), if there are no legal actions then we'll return 0.0. Otherwise, we'll return the maximum Q value for the input state.

```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
        actions = self.getLegalActions(state)
        bestAction = None
        if len(actions) != 0:
            maxValue = self.computeValueFromQValues(state)
            candidate = []
            for action in actions:
                if self.getQValue(state, action) == maxValue:
                    candidate.append(action)
            bestAction = random.choice(candidate)

        return bestAction
        # End your code
```

3) If there are legal actions, we'll return the action which has the maximum Q value. Notice that, if there're more than one action match the maximum value then we'll use random.choice() to choose the return action. If there are no legal actions then we'll return None.

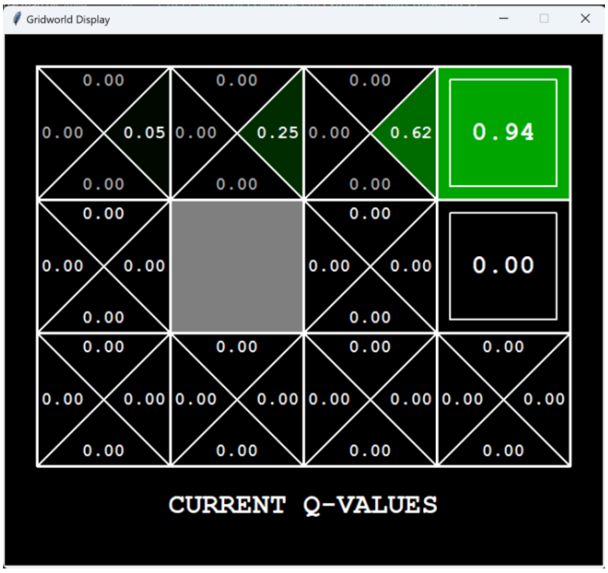
```python
    def update(self, state, action, nextState, reward):
        """
          The parent class calls this to observe a
          state = action => nextState and reward transition.
          You should do your Q-Value update here

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

4) We update the Q Value according to the below formula:

$$Q(s,a) = (1-\alpha)Q(s,a) + \alpha[r + \gamma maxQ(s',a')]$$

5) Result:

```
> python gridworld.py -a q -k 4 -m
```



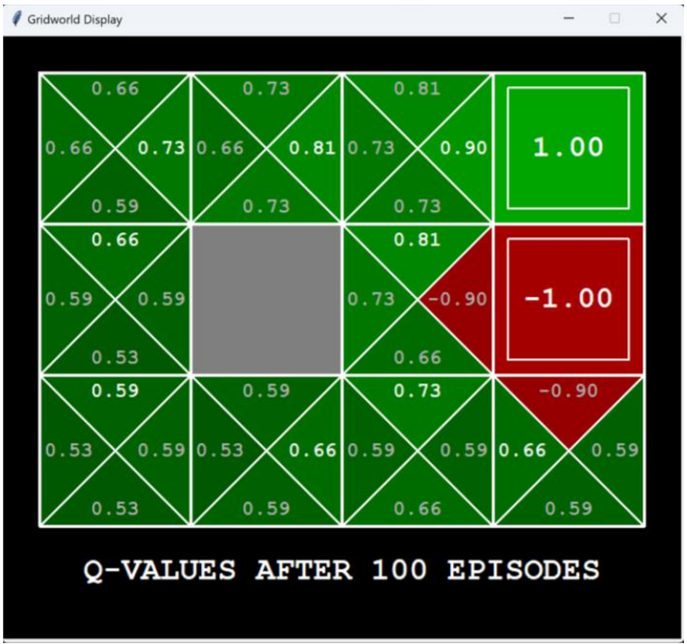

```
Question part2-2
================

*** PASS: test_cases\part2-2\1-tinygrid.test
*** PASS: test_cases\part2-2\2-tinygrid-noisy.test
*** PASS: test_cases\part2-2\3-bridge.test
*** PASS: test_cases\part2-2\4-discountgrid.test

### Question part2-2: 10/10 ###
```

Part 2-3: epsilon-greedy action selection (10%)

```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
        legalActions = self.getLegalActions(state)
        action = None
        "*** YOUR CODE HERE ***"
        # Begin your code
        if util.flipCoin(self.epsilon): #ϵ random
            if len(legalActions) != 0:
                action = random.choice(legalActions)
        else: #1-ϵ max
            action = self.computeActionFromQValues(state)
        return action
        # End your code
```

1) In epsilon-greedy action, there's a probability of 1−ϵ to perform actions using argmax, selecting the action that is shown to be the most ideal in the Q-table. Then, with a probability of ϵ, random actions are taken, disregarding the results in the Q-table.
2) Result:

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



- The resulting step (up, up ,right, right, right) match with what I expected.

- As the ϵ become smaller, the average returns from start state will increase. Probably because of the random action of ϵ probability.

e 0.9
```
EPISODE 100 COMPLETE: RETURN WAS 0.28242953648100017

AVERAGE RETURNS FROM START STATE: 0.027229153803138603
```

e 0.7
```
EPISODE 100 COMPLETE: RETURN WAS 0.13508517176729928

AVERAGE RETURNS FROM START STATE: 0.1546626089027024
```

e 0.5
```
EPISODE 100 COMPLETE: RETURN WAS 0.22876792454961012

AVERAGE RETURNS FROM START STATE: 0.3437379606569935
```

```
Question part2-3
================

*** PASS: test_cases\part2-3\1-tinygrid.test
*** PASS: test_cases\part2-3\2-tinygrid-noisy.test
*** PASS: test_cases\part2-3\3-bridge.test
*** PASS: test_cases\part2-3\4-discountgrid.test

### Question part2-3: 5/5 ###
```

Part 2-4: Approximate Q-learning (Bonus) (10%)

```python
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
        featureVectors = self.featExtractor.getFeatures(state, action)
        totalValue = 0
        for feature in featureVectors:
            totalValue += featureVectors[feature] * self.weights[feature]
        return totalValue
        # End your code
```

1) Calculate Q value by this formula:

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

```python
    def update(self, state, action, nextState, reward):
        """
          Should update your weights based on transition
        """
        "*** YOUR CODE HERE ***"
        # Begin your code
        featureVectors = self.featExtractor.getFeatures(state, action)
        correction = (reward + self.discount * self.getValue(nextState)) - self.getQValue(state, action)
        for feature in featureVectors:
            self.weights[feature] = self.weights[feature] + self.alpha * correction * featureVectors[feature]
        # End your code
```

2) And follow the below formula to update the weight vector.

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

3) I didn't modify the featureExtractor.py and use the default settings to run.

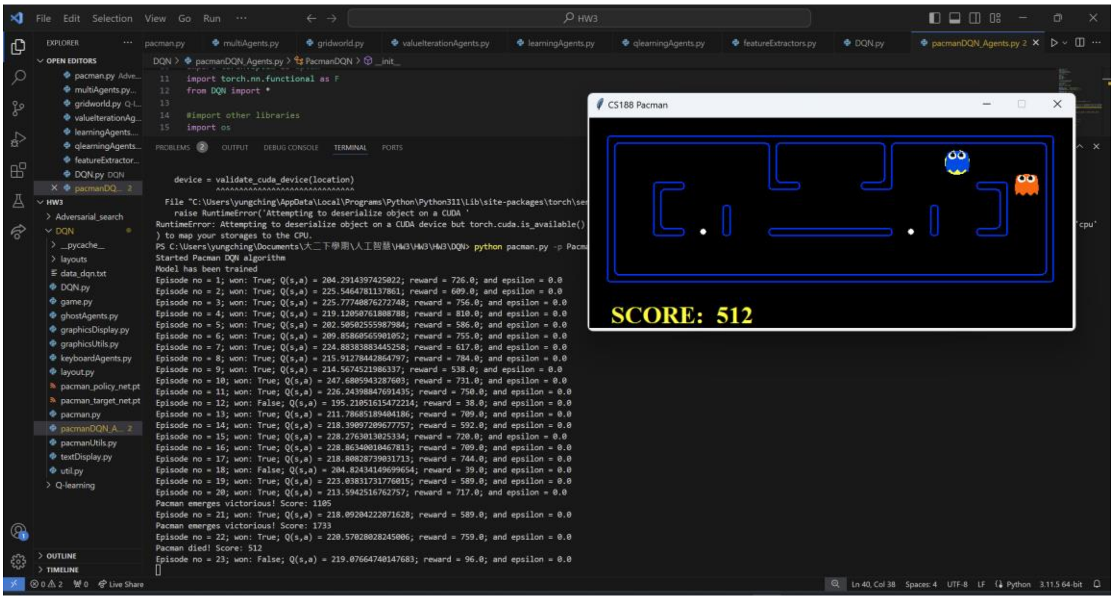
Below is the result I get:

```
Reinforcement Learning Status:
        Completed 100 test episodes
        Average Rewards over testing: 823.48
        Average Rewards for last 100 episodes: 823.48
        Episode took 7.88 seconds
Average Score: 823.48
Scores:        960.0, 979.0, 973.0, 977.0, 963.0, 962.0, 964.0, 975.0, 976.0, 965.0, 969.0, 971.0, 951.0, -109.0, -243.0, 949.0, 974.0, -205.0, 976.0, 983.0,
 968.0, -132.0, 958.0, 1352.0, 985.0, 957.0, 956.0, 984.0, -96.0, 981.0, 975.0, 957.0, 971.0, -48.0, 964.0, 974.0, 965.0, 980.0, 973.0, 985.0, 964.0, 952.0,
976.0, 113.0, 963.0, 950.0, 972.0, 971.0, -204.0, 983.0, -130.0, 979.0, 955.0, 981.0, 983.0, 966.0, 980.0, 968.0, 974.0, 984.0, 1142.0, 1130.0, -225.0, 976.0
, -326.0, 957.0, 971.0, 984.0, 989.0, 985.0, 960.0, 972.0, 971.0, 975.0, 972.0, 939.0, 980.0, 953.0, 960.0, 983.0, 970.0, 1364.0, 985.0, 955.0, 966.0, -268.0
, -309.0, 984.0, 983.0, -368.0, 962.0, 1127.0, 974.0, 963.0, 958.0, 972.0, 978.0, 1177.0, 952.0, 976.0
Win Rate:      86/100 (0.86)
Record:        Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Loss, Loss, Win, Win, Loss, Win, Win, Win, Loss, Win, Win, Win, Win, Win, Win
, Loss, Win, Win, Win, Win, Loss, Win, Win, Win, Win, Win, Win, Win, Win, Win, Loss, Win, Win, Win, Win, Loss, Win, Loss, Win, Win, Win, Win, Win, Win, Win,
Win, Win, Win, Win, Loss, Win, Loss, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Loss, Loss, Win, Win
, Loss, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win
*** FAIL: test_cases\part2-4\grade-agent.test (7 of 8 points)
```

```
Provisional grades
==================
Question part2-4: 7/10
------------------
Total: 7/10
```

Part 3

```
Model has been trained
Episode no = 1; won: True; Q(s,a) = 224.98729821836827; reward = 783.0; and epsilon = 0.0
Episode no = 2; won: True; Q(s,a) = 219.0291149939279; reward = 642.0; and epsilon = 0.0
Episode no = 3; won: True; Q(s,a) = 197.82242683566224; reward = 717.0; and epsilon = 0.0
Episode no = 4; won: True; Q(s,a) = 206.25263308304005; reward = 729.0; and epsilon = 0.0
Episode no = 5; won: True; Q(s,a) = 220.9753228550046; reward = 777.0; and epsilon = 0.0
Episode no = 6; won: True; Q(s,a) = 203.04519261680784; reward = 651.0; and epsilon = 0.0
Episode no = 7; won: True; Q(s,a) = 197.38700913912248; reward = 677.0; and epsilon = 0.0
Episode no = 8; won: True; Q(s,a) = 197.605981601667; reward = 699.0; and epsilon = 0.0
Episode no = 9; won: True; Q(s,a) = 230.4522272941136; reward = 780.0; and epsilon = 0.0
Episode no = 10; won: True; Q(s,a) = 199.09679880403428; reward = 718.0; and epsilon = 0.0
Episode no = 11; won: True; Q(s,a) = 207.82385886066234; reward = 727.0; and epsilon = 0.0
Episode no = 12; won: True; Q(s,a) = 197.15051539771105; reward = 748.0; and epsilon = 0.0
Episode no = 13; won: True; Q(s,a) = 216.03693762530384; reward = 702.0; and epsilon = 0.0
Episode no = 14; won: True; Q(s,a) = 228.81606776125037; reward = 648.0; and epsilon = 0.0
Episode no = 15; won: True; Q(s,a) = 200.04122642943798; reward = 741.0; and epsilon = 0.0
Episode no = 16; won: True; Q(s,a) = 197.61000222817847; reward = 639.0; and epsilon = 0.0
Episode no = 17; won: True; Q(s,a) = 215.44654941518357; reward = 741.0; and epsilon = 0.0
Episode no = 18; won: True; Q(s,a) = 221.31875919305062; reward = 717.0; and epsilon = 0.0
Episode no = 19; won: True; Q(s,a) = 233.81067074702128; reward = 512.0; and epsilon = 0.0
Episode no = 20; won: True; Q(s,a) = 204.52033261335202; reward = 711.0; and epsilon = 0.0
Pacman died! Score: 742
Episode no = 21; won: False; Q(s,a) = 213.37487784088177; reward = 177.0; and epsilon = 0.0
Pacman emerges victorious! Score: 1761
Episode no = 22; won: True; Q(s,a) = 216.3934268198223; reward = 776.0; and epsilon = 0.0
Pacman emerges victorious! Score: 1567
Episode no = 23; won: True; Q(s,a) = 213.27907494289886; reward = 764.0; and epsilon = 0.0
Pacman emerges victorious! Score: 1575
Episode no = 24; won: True; Q(s,a) = 203.06078666503063; reward = 750.0; and epsilon = 0.0
Pacman emerges victorious! Score: 1751
Episode no = 25; won: True; Q(s,a) = 197.52198232557586; reward = 788.0; and epsilon = 0.0
Average Score: 1479.2
Scores:        742.0, 1761.0, 1567.0, 1575.0, 1751.0
Win Rate:      4/5 (0.80)
Record:        Loss, Win, Win, Win, Win
```



I use the pre-trained models and set 'model_trained' into True. Then run the test command: "python pacman.py -p PacmanDQN -n 25 -x 20 -l smallClassic" and get the above screenshot's results.

Questions:

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

   In On-policy, the agent learns and updates its policy based on its current policy behavior, and it only considers experiences under the current policy. While in Off-policy, the agent can use a different action selection policy, the policy used for updating and interacting with the environment and generating data are different policies.

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

   1) In value-based, the agent learns the value function, which represents the expected cumulative rewards from any given state. And it chooses the action which has the highest value. While policy -based aims to learn the policy directly using a parameterized function and map state to action.
   2) Actor-critic is the combination of both value-based and policy-based method.
   3) The value function $V^{\pi}(S)$ represents the expected value of cumulative reward that an agent can achieve from a given state S under a policy π.

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

   1) The main difference is how they update their estimates based on the received rewards. In Monte-Carlo (MC) approach, each update must wait until the end of an episode. While Temporal-difference (TD) method updates value estimates incrementally after each time step. It is a kind of bootstrapping method which means updating value estimates based on other value estimates rather than waiting for an outcome.
   2) Therefore, TD method learns more efficiently, and it is suitable for learning in continuous tasks.

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

   1) $Q^{\pi}(s, a)$ means the expected reward of taking action a in state s and then continuing according to policy π.
   2) the relationship between $Q^{\pi}(s, a)$ and $V^{\pi}(s)$:

$V^{\pi}(s)$ is state value function, and it can be written as the sum of the probability of selecting each action or policy multiplied by the action-value of each action.

$$V^{\pi}(s) = \sum_{a \in A} \pi(a|s) * Q^{\pi}(s,a)$$

$$Q^{\pi}(s,a) = \sum_{s' \in S} P(s'|s,a) * [R(s,a,s') + \gamma V^{\pi}(s')]$$

P is the state transition matrix that gives the probability of reaching the next state s' from state s, R is the reward, and V is the state value of the next state.

5. **Describe following tips Target Network, Exploration and Replay Buffer using in Q-learning.**
    1) Target Network has the same architecture as the main Q-network but its parameters are updated at a slower rate. If we only have one network, then each update not only changes the Q-value $Q(s,a)$ being trained, but also our target Q-value $Q(s',a')$. The purpose of target network is leading to a more stable learning process.
    2) Exploration is a strategy used by the agent to explore the environment and discover new actions or states that may lead to better rewards. ε-greedy is an example of exploration.
    3) Replay Buffer is a memory buffer that stores and replays experiences collected from interactions of the agent with the environment. Agent will use them to update Q function during learning process.
6. **Explain what is different between DQN and Q-learning.**
    1) DQN merges Q-Learning with deep learning by employing a deep neural network to approximate the action-value function, while Q-Learning relies on table storage. Unlike Q-Learning, which directly learns from the data of the next state in the table, DQN utilizes experience replay, randomly sampling from historical data stored in a replay buffer.Besides experience replay, another innovation of DQN is target network which is explained in question 5. It helps stabilize the training process.

Discussion

1. Compare the performance of every method and do some discussions.
   1) In Minimax method, setting depth to 3 only has 0.4 win rate, but increasing the depth to 4 will result in 0.8 win rate.

```
> -p MinimaxAgent -l minimaxClassic -a depth=3 -q -n 10
```

```
PS C:\Users\yungching\Documents\大二下學期\人工智慧\HW3\HW3\HW3\Adversarial_search> python pacman.py -p MinimaxAgent -l minimaxClassic -a depth=3 -q -n 10
Pacman emerges victorious! Score: 514
Pacman died! Score: -495
Pacman emerges victorious! Score: 512
Pacman emerges victorious! Score: 513
Pacman died! Score: -495
Pacman died! Score: -492
Pacman died! Score: -495
Pacman died! Score: -496
Pacman died! Score: -492
Pacman emerges victorious! Score: 514
Average Score: -91.2
Scores:        514.0, -495.0, 512.0, 513.0, -495.0, -492.0, -495.0, -496.0, -492.0, 514.0
Win Rate:      4/10 (0.40)
Record:        Win, Loss, Win, Win, Loss, Loss, Loss, Loss, Loss, Win
```

```
> -p MinimaxAgent -l minimaxClassic -a depth=4 -q -n 10
```

```
PS C:\Users\yungching\Documents\大二下學期\人工智慧\HW3\HW3\HW3\Adversarial_search> python pacman.py -p MinimaxAgent -l minimaxClassic -a depth=4 -q -n 10
Pacman died! Score: -495
Pacman emerges victorious! Score: 516
Pacman emerges victorious! Score: 516
Pacman died! Score: -495
Pacman emerges victorious! Score: 516
Pacman emerges victorious! Score: 516
Pacman emerges victorious! Score: 513
Pacman emerges victorious! Score: 516
Pacman emerges victorious! Score: 516
Pacman emerges victorious! Score: 516
Average Score: 313.5
Scores:        -495.0, 516.0, 516.0, -495.0, 516.0, 516.0, 513.0, 516.0, 516.0, 516.0
Win Rate:      8/10 (0.80)
Record:        Loss, Win, Win, Loss, Win, Win, Win, Win, Win, Win
```

   2) In Expectimax, when using the same command as in minimax, it shows better performance compared to minimax. Specifically, at depths 3 and 4, it achieves win rates of 0.5 and 1.0, respectively.

```
> -p ExpectimaxAgent -l minimaxClassic -a depth=3 -q -n 10
```

```
PS C:\Users\yungching\Documents\大二下學期\人工智慧\HW3\HW3\HW3\Adversarial_search> python pacman.py -p ExpectimaxAgent -l minimaxClassic -a depth=3 -q -n 10

Pacman died! Score: -498
Pacman emerges victorious! Score: 515
Pacman died! Score: -495
Pacman emerges victorious! Score: 511
Pacman emerges victorious! Score: 507
Pacman emerges victorious! Score: 515
Pacman died! Score: -494
Pacman died! Score: -494
Pacman emerges victorious! Score: 511
Pacman died! Score: -516
Average Score: 6.2
Scores:        -498.0, 515.0, -495.0, 511.0, 507.0, 515.0, -494.0, -494.0, 511.0, -516.0
Win Rate:      5/10 (0.50)
Record:        Loss, Win, Loss, Win, Win, Win, Loss, Loss, Win, Loss
```

```
> -p ExpectimaxAgent -l minimaxClassic -a depth=4 -q -n 10
```

```
PS C:\Users\yungching\Documents\大二下學期\人工智慧\HW3\HW3\HW3\Adversarial_search> python pacman.py -p ExpectimaxAgent -l minimaxClassic -a depth=4 -q -n 10
Pacman emerges victorious! Score: 516
Pacman emerges victorious! Score: 516
Pacman emerges victorious! Score: 516
Pacman emerges victorious! Score: 516
Pacman emerges victorious! Score: 516
Pacman emerges victorious! Score: 516
Pacman emerges victorious! Score: 516
Pacman emerges victorious! Score: 516
Pacman emerges victorious! Score: 516
Pacman emerges victorious! Score: 516
Average Score: 516.0
Scores:        516.0, 516.0, 516.0, 516.0, 516.0, 516.0, 516.0, 516.0, 516.0, 516.0
Win Rate:      10/10 (1.00)
Record:        Win, Win, Win, Win, Win, Win, Win, Win, Win, Win
```

3) In smallGrid, the ApproximateQAgent successfully win all games. And in mediumGrid, we use the SimpleExtractor and the result seems good.

`> -p ApproximateQAgent -x 2000 -n 2010 -l smallGrid`

```
Training Done (turning off epsilon and alpha)
---------------------------------------------
Pacman emerges victorious! Score: 503
Pacman emerges victorious! Score: 499
Pacman emerges victorious! Score: 503
Pacman emerges victorious! Score: 503
Pacman emerges victorious! Score: 499
Pacman emerges victorious! Score: 495
Pacman emerges victorious! Score: 495
Pacman emerges victorious! Score: 499
Pacman emerges victorious! Score: 495
Pacman emerges victorious! Score: 499
Average Score: 499.0
Scores:        503.0, 499.0, 503.0, 503.0, 499.0, 495.0, 495.0, 499.0, 495.0, 499.0
Win Rate:      10/10 (1.00)
Record:        Win, Win, Win, Win, Win, Win, Win, Win, Win, Win
```

`> -p ApproximateQAgent -a extractor=SimpleExtractor -x 50 -n 60 -l mediumGrid`

```
Beginning 50 episodes of Training
Training Done (turning off epsilon and alpha)
---------------------------------------------
Pacman emerges victorious! Score: 527
Pacman emerges victorious! Score: 529
Pacman emerges victorious! Score: 525
Pacman emerges victorious! Score: 529
Pacman emerges victorious! Score: 529
Pacman emerges victorious! Score: 529
Pacman emerges victorious! Score: 529
Pacman emerges victorious! Score: 529
Pacman emerges victorious! Score: 527
Pacman emerges victorious! Score: 527
Average Score: 528.0
Scores:        527.0, 529.0, 525.0, 529.0, 529.0, 529.0, 529.0, 529.0, 527.0, 527.0
Win Rate:      10/10 (1.00)
Record:        Win, Win, Win, Win, Win, Win, Win, Win, Win, Win
```

`> -p ApproximateQAgent -a extractor=SimpleExtractor -x 2000 -n 2010 -l smallClassic`

```
Training Done (turning off epsilon and alpha)
---------------------------------------------
Pacman died! Score: -287
Pacman emerges victorious! Score: 968
Pacman emerges victorious! Score: 971
Pacman emerges victorious! Score: 972
Pacman emerges victorious! Score: 977
Pacman emerges victorious! Score: 979
Pacman emerges victorious! Score: 977
Pacman emerges victorious! Score: 986
Pacman emerges victorious! Score: 973
Pacman emerges victorious! Score: 987
Average Score: 850.3
Scores:        -287.0, 968.0, 971.0, 972.0, 977.0, 979.0, 977.0, 986.0, 973.0, 987.0
Win Rate:      9/10 (0.90)
Record:        Loss, Win, Win, Win, Win, Win, Win, Win, Win, Win
```

4) In the same setting and environment, DQN only has 0.8 win rate but ApproximateQAgent has 1.0 win rate. It seems like ApproximateQAgent perform better. But I think it is mainly because we only count in 5 games, the number of sample is too small so there may have some bias.

`> -p PacmanDQN -n 25 -x 20 -l smallClassic`

```
Average Score: 1479.2
Scores:        742.0, 1761.0, 1567.0, 1575.0, 1751.0
Win Rate:      4/5 (0.80)
Record:        Loss, Win, Win, Win, Win
```

`> -p ApproximateQAgent -a extractor=SimpleExtractor -n 25 -x 20 -l smallClassic`

```
PS C:\Users\yungching\Documents\大二下學期\人工智慧\HW3\HW3\HW3\Q-learning> python pacman.py -p ApproximateQAgent -a extractor=SimpleExtractor -n 25 -x 20 -l
 smallClassic -q
Beginning 20 episodes of Training
Training Done (turning off epsilon and alpha)
---------------------------------------------
Pacman emerges victorious! Score: 959
Pacman emerges victorious! Score: 955
Pacman emerges victorious! Score: 976
Pacman emerges victorious! Score: 984
Pacman emerges victorious! Score: 974
Average Score: 969.6
Scores:        959.0, 955.0, 976.0, 984.0, 974.0
Win Rate:      5/5 (1.00)
Record:        Win, Win, Win, Win, Win
```

5) In the same environment: smallClassic. The win rates of three methods are the same but DQN has the highest average score. Although expectimax has higher average score than approximateQAgent, but it takes a long time for expectimax to run.

Method	Average score	Win rate
ExpectimaxAgent (depth=4, -n 10)	1255.5	9/10
ApproximateQAgent (`SimpleExtractor`) (`-n 25 -x 15`)	850.0	9/10
DQN (`-n 25 -x 15`)	1310.1	9/10

```
PS C:\Users\yungching\Documents\大二下學期\人工智慧\HW3\HW3\HW3\Adversarial_search> python pacman.py -p ExpectimaxAgent -l smallClassic -a depth=4 -q -n 10
Pacman died! Score: 71
Pacman emerges victorious! Score: 1163
Pacman emerges victorious! Score: 1636
Pacman emerges victorious! Score: 1095
Pacman emerges victorious! Score: 1250
Pacman emerges victorious! Score: 1563
Pacman emerges victorious! Score: 1217
Pacman emerges victorious! Score: 1522
Pacman emerges victorious! Score: 1434
Pacman emerges victorious! Score: 1604
Average Score: 1255.5
Scores:        71.0, 1163.0, 1636.0, 1095.0, 1250.0, 1563.0, 1217.0, 1522.0, 1434.0, 1604.0
Win Rate:      9/10 (0.90)
Record:        Loss, Win, Win, Win, Win, Win, Win, Win, Win, Win
```

```
PS C:\Users\yungching\Documents\大二下學期\人工智慧\HW3\HW3\HW3\Q-learning> python pacman.py -p ApproximateQAgent -a extractor=SimpleExtractor -n 25 -x 15 -l
 smallClassic
Beginning 15 episodes of Training
Training Done (turning off epsilon and alpha)
---------------------------------------------
Pacman emerges victorious! Score: 966
Pacman emerges victorious! Score: 1166
Pacman emerges victorious! Score: 955
Pacman emerges victorious! Score: 977
Pacman died! Score: -399
Pacman emerges victorious! Score: 968
Pacman emerges victorious! Score: 953
Pacman emerges victorious! Score: 965
Pacman emerges victorious! Score: 981
Pacman emerges victorious! Score: 968
Average Score: 850.0
Scores:        966.0, 1166.0, 955.0, 977.0, -399.0, 968.0, 953.0, 965.0, 981.0, 968.0
Win Rate:      9/10 (0.90)
Record:        Win, Win, Win, Win, Loss, Win, Win, Win, Win, Win
```

```
Pacman emerges victorious! Score: 1303
Episode no = 25; won: True; Q(s,a) = 195.26003879294274; reward = 649.0; and epsilon = 0.0
Average Score: 1310.1
Scores:        1551.0, 1528.0, 1305.0, 1547.0, 1449.0, 1541.0, 1280.0, 1761.0, -164.0, 1303.0
Win Rate:      9/10 (0.90)
Record:        Win, Win, Win, Win, Win, Win, Win, Win, Loss, Win
```

2. Describe problems you meet and how you solve them.

   When run the command in part 3, there's an error:

   `RuntimeError: Attempting to deserialize object on a CUDA device but torch.cuda.is_available is False. If you are running on a CPU-only machine, please use torch.load with map_location=torch.device('cpu') to map your storage to the CPU.`

   I found a similar issue on the discussion forum on Teams, seemingly related to the computer's CPU limitations.

   Based on the error indication and the actions taken by that student, I modified `torch.load('pacman_policy_net.pt').to(self.device)` to `torch.load('pacman_policy_net.pt',map_location=torch.device('cpu')).to(self.device)`, which successfully resolved the problem.