Intro to artificial Intelligence
Hw3

109550198 卜銳凱

1 Adversarial Search

For Part 1-1: Implementation of Minimax Algorithm:

The Minimax algorithm is a searching method that is utilized in several game AI. It seeks a specific activity in order to maximize a minimum possible score.

The code for implementing the minimax algorithm in the Pacman game is shown below.

```
# Begin your code
def Minimax(gameState, depth, agentIdx):
    if agentIdx == 0:
        if gameState.isWin() or gameState.isLose() or depth + 1 == self.depth:
            return self.evaluationFunction(gameState)
        value = []
        actions = gameState.getLegalActions(0)
        value = [Minimax(gameState.getNextState(0,action), depth + 1, 1) for action in actions]
        return max(value)
    else:
        if gameState.isWin() or gameState.isLose():
            return self.evaluationFunction(gameState)
        value = []
        actions = gameState.getLegalActions(agentIdx)
        if agentIdx < gameState.getNumAgents() - 1:
            value = [Minimax(gameState.getNextState(agentIdx, action), depth, agentIdx+1) for action in actions]
        else:
            value = [Minimax(gameState.getNextState(agentIdx,action), depth, 0) for action in actions]
        return min(value)

Statescore = []
actions = gameState.getLegalActions(0)
Statescore= [Minimax(gameState.getNextState(0, action), 0, 1) for action in actions]
return actions[Statescore.index(max(Statescore))]

# End your code
```

To round off this section, I created a subfunction called "Minimax." If the agentIdx is 0, the agent is Pacman. First, we must determine whether this run is the base case by determining whether we will win or lose this game and whether the next depth is as deep as the self.depth. Return the score generated by "self.evaluationFunction()" if one of the preceding conditions is met. Otherwise, we proceed to the depth that will produce the smallest amount of value for all legal actions generated by "getLegalAction()."

Finally, return the maximum value among all of the values returned by the following depth. If agentIdx is greater than zero, the agent is a ghost. What we need to do first is similar to what Pacman does, except that the base case condition is simply whether we win or lose. And we have to traverse through all of this depth's ghosts to acquire the recursive function's value. Furthermore, if the ghost is the final one at that depth, we switch to calling the pacman at that depth to retrieve the highest value.

After defining the "Minimax" subfunction, I assign all of Pacman's 0-depth actions to it and record the score in a list. Finally, restore the action with the highest score.

For Part1-2: Implementation of Expectimax Algorithm

Rather than using the minimum value in the min layer. The Expectimax algorithm selects the expectation value from any possible values. Every action taken by the ghosts in the Pacman game has the same probability. As a result, the expected value of all actions equals the average of all potential scores.

The pacman game's expectimax algorithm is implemented in the code below.

```
# Begin your code
def Expectimax(gameState, depth, agentIdx):
    if agentIdx == 0:
        if gameState.isWin() or gameState.isLose() or depth + 1 == self.depth:
            return self.evaluationFunction(gameState)

        value = []
        actions = gameState.getLegalActions(0)
        value = [Expectimax(gameState.getNextState(0, action), depth + 1, 1) for action in actions]
        return max(value)
    else:
        if gameState.isWin() or gameState.isLose():
            return self.evaluationFunction(gameState)
        actions = gameState.getLegalActions(agentIdx)
        if len(actions) == 0:
            return 0
        Val = []
        if agentIdx < gameState.getNumAgents() - 1:
            Val = [Expectimax(gameState.getNextState(agentIdx, action), depth, agentIdx + 1) for action in actions]
        else:
            Val = [Expectimax(gameState.getNextState(agentIdx, action), depth, 0) for action in actions]
        return float(sum(Val) / len(actions))

actions = gameState.getLegalActions(0)
Statescore = []
Statescore = [Expectimax(gameState.getNextState(0, action), 0, 1) for action in actions]
return actions[Statescore.index(max(Statescore))]
# End your code
```

This section's requirements are comparable to the previous sections. The only difference is that when the agent is a ghost, I return the value that the sum of all the ghosts' values divided by the number of all the ghosts' possible actions to realize what the spec said, "choose among its legal actions uniformly at random" in the subfunction "Expectimax."

2.-Qlearning

For part 2-1: Value Iteration:

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

The implementation of ValueIterationAgent is shown below.

```
def runValueIteration(self):
    "*** YOUR CODE HERE ***"
    # Begin your code
    for _ in range(self.iterations):
        tmp = util.Counter()
        for state in self.mdp.getStates():
            maxValue = float('-inf')
            for action in self.mdp.getPossibleActions(state):
                value = self.computeQValueFromValues(state, action)
                maxValue = max(maxValue, value)
                tmp[state] = maxValue
        for state in self.mdp.getStates():
            self.values[state] = tmp[state]

    # End your code
```

for $h \leftarrow 0$ to $H-1$ do
  foreach s do
    $V^*(s) \leftarrow \max_a \sum_{s'} \mathrm{P}(s'|s;a)[R(s,a,s') + \gamma V^*(s')]$;
  end
end

We must implement a value iteration algorithm for this function. The number of steps required to obtain the totally reward is self.Iterations. First, I create a dictionary to temporarily store the maximum value and its corresponding state. Following that, we go over all the states and their actions. To get the value, use "computeQValueFromValues()" and keep the maximum values. Last but not least, preserve the temporary dictionary value in self.values.

```
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
    nextStatePr = self.mdp.getTransitionStatesAndProbs(state, action)
    val = [pr*(self.mdp.getReward(state, action, nextState) + self.discount * self.values[nextState]) for nextState, pr in nextStatePr]
    return sum(val) # return q value
    #util.raiseNotDefined()
    # End your code
```

(function) getTransitionStatesAndProbs: Any

$$\sum_{s'} P(s'|s;a)[R(s,a,s') + \gamma V^*(s')]$$

This formula just occurred to me at this section. At first, I retain a list of all the probabilities of the next state based on the present state and activity. Second, depending on the probability, run a for loop, multiply the probability and the sum of the reward plus the discount value by the value of the next state, and store the result in a list. Finally, return the list's summation.

```
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
    maxValue = float('-inf')
    returnAction = None
    for action in self.mdp.getPossibleActions(state):
        value = self.computeQValueFromValues(state, action)
        if maxValue < value:
            maxValue = value
            returnAction = action

    return returnAction

    #check for terminal
    #util.raiseNotDefined()
    # End your code
```

$$\pi^*(s) \leftarrow \arg\max_a \sum_{s'} P(s'|s;a)[R(s,a,s') + \gamma V^*(s')];$$

In this section, I run through all of the possible actions, use "computeQValueFromValues" to acquire the relevant value, set a variable maxValue to keep

the maximal value for comparison, and keep the action with the maximal value in returnAction. Furthermore, if it is in the terminal condition, we just return None.

For Part 2-2: Q- value:

```
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.qvalue[state, action]
    # End your code
```

Get the Q-value from the qvalue dictionary according to state and action.

```
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)
    q_value = []

    if len(actions) == 0: #terminal state
        return 0.0
    else:
        q_value = [self.getQValue(state, action) for action in actions]

    return max(q_value)
    # End your code
```

We run through all of the actions based on the given state, getting the Q-value using "getQValue()" and storing them in a list. Finally, return the maximum value in the list. In addition, if the state is terminal 0, return zero.

```
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)
    maxValAction = []
    q_value = float('-inf')

    if len(actions) == 0:
        return None
    else:
        for s in actions:
            value = self.getQValue(state, s)
            if value > q_value:
                q_value = self.getQValue(state, s)
                maxValAction = [s]
            elif value == q_value:
                maxValAction.append(s)
        return random.choice(maxValAction)
    # End your code
```

This part is like the previous one, except that if it is not the terminal state, we keep a list of all actions with the highest Q-value in advance. Finally, using random.choice, return one of them at random.().

```
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
    # print('debug',self.qvalue[state, action])
    # print('debug',self.qvalue[state])
    self.qvalue[state,action] = (1 - self.alpha) * self.qvalue[state,action] + self.alpha * (reward + self.discount * self.computeValueFr
    # End your code
```

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma max_{a'} Q(s',a') - Q(s,a)]$$

I obtain the original Q-value and "computeValueFromQValues" to obtain the maximum Q-value from the next state. Then, as the formula indicates, update the Q-value determined.

For part2-3: epsilon-greedy action selection

```
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 len(legalActions) == 0:
        return action
    else:
        if util.flipCoin(1 - self.epsilon):
            return self.computeActionFromQValues(state)
        else:
            return random.choice(legalActions)
    # End your code
```

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

In this section, we implement the epsilon-greedy policy. If the probability (1- epsilon) is true, use "computeActionFromQValues()" to get the action with the maximal Q-value. Otherwise, return any action at random.

For part 2-4: Approximate Q-learning:

```
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)
    res = []
    # print(featureVectors)
    res= [self.weights.get(feature, 0) * val for feature, val in featureVectors.items()]
    return sum(res)
    # End your code
```

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

As the formula shows, we acquire all the features that correspond to the state and action. Then have them complete the dot product using the weights given on the feature.

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

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

To begin, if the current state is not the terminal one, we retrieve the maximal Q-value using the "getQValue" defined in this section and select the maximal one. Otherwise, the maximum Q-value will be 0. After that, compute the correction using the formula. Finally, update the new weight.

3.DQN

For Part 3:

What is the difference between On-policy and Off-policy?

In reinforcement learning, there are two approaches: on-policy and off-policy:

On-policy methods learn and update the policy based on the experience gained from following current policy. The data used for learning is derived from the same policy that collects the data. On-policy approaches require a lot of interaction with the environment to converge and might be sample inefficient.

Off-policy methods learn and update the policy using data supplied by a different policy, known as a behavior policy. This behavior policy collects the agent's experience, which is then used to update the target policy. Off-policy algorithms are more sample efficient because they can reuse previous experience and learn from a broader range of data.

In summary, the main difference between on-policy and off-policy methods is whether the data used for learning and policy updates is derived from the same policy (on-policy) or from a different policy (off-policy). Off-policy methods have the advantage of being flexible and efficient when learning from many data sources

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

Value-based methods: Value-based methods focus with directly calculating the value function or the action-value function. Under a given policy, the value function describes the expected return or cumulative reward an agent can earn from a particular state or state-action pair. The agent in value-based methods seeks the ideal value function by iteratively updating its estimates based on observed rewards and transitions. Q-learning and Deep Q-Networks (DQN) are examples of popular algorithms in this field**.**

Policy-based methods: Policy-based methods directly optimize the policy function that maps states to actions. Instead of estimating the value function, these algorithms aim for the policy that maximizes the expected return. Policy-based methods typically use gradient ascent to iteratively change policy parameters based on observed rewards. REINFORCE and Proximal Policy Optimization (PPO) are two policy-based methods.

Actor-Critic methods: Actor-Critic methods combine aspects of both value-based and policy-based methods. They maintain a value function as well as a policy function. The actor oversees

selecting actions based on current policy, whereas the critic evaluates the value of different states or state-action pairs. The critic gives the actor feedback by estimating the advantage or quality of the actor's chosen actions. The advantage represents how much better the actor's action is in comparison to the typical action in each state. Actor-Critic methods are recognized for their ability to handle continuous action spaces and deal with the benefits of both value-based and policy-based methods. Advantage Actor-Critic (A2C) and Deep Deterministic Policy Gradient (DDPG) are two Actor-Critic algorithms.

The value function $V^{\pi}(S)$ reflects the expected return or cumulative reward that an agent can earn by being in a particular state of S and then following a policy. In other words, it measures how good it is for the agent to be in a specific state while adhering to a specified policy. It is defined mathematically as the expected sum of discounted future rewards:

$V^{\pi}(S) = E [R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... | S_t = s, \pi]$
In this case, R_t denotes the reward obtained at time step t, γ is the discount factor that defines the importance of future rewards, and π is the policy in place. The value function is important for evaluating states and guiding the decision-making process of the agent.

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

To summarize, the main difference between the MC and TD methods is how they estimate the value function $V^{\pi}(S)$. MC methods require full episodes and run a batch update after each episode, whereas TD methods update their estimates progressively based on individual time steps. While MC methods are more suited for episodic tasks, TD approaches can handle both episodic and continuous jobs.

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

The following is the relationship between the state-value function $V^{\pi}(S)$and the state-action value function $Q^{\pi}(s,a)$ in Q-learning:

The following is the relationship between the state-value function $V^{\pi}(S)$ and the state-action value function $Q^{\pi}(s,a)$ in Q-learning:

Q-learning is a popular off-policy reinforcement learning technique used to find the optimal policy in a Markov Decision Process (MDP). It use the state-action value function $Q^{\pi}(s,a)$ to calculate the maximum predicted cumulative rewards that can be obtained by taking action an in state s and then implementing an optimal policy.

The Q-learning method iteratively updates the Q-values based on observed rewards and estimated Q-values of upcoming states. The updating policy is as follows:

$Q(s, a) \leftarrow Q(s, a) + \alpha * [r + \gamma * \max(Q(s', a')) - Q(s, a)]$

The relationship to $V^{\pi}(S)$ is that $V^{\pi}(S)$ can be derived from $Q^{\pi}(s,a)$ by obtaining the maximum Q-value over all possible actions an in state s under policy π :

$V^{\pi}(S) = \max(Q^{\pi}(s,a))$ over all actions a

In other words, V(s) represents the greatest expected cumulative rewards achievable from state s by following policy, whereas $Q^{\pi}(s,a)$ reflects the expected cumulative rewards achievable by taking action an in state s and then following policy π.

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

Target Network: The target network is a copy of the Q-network used to estimate the optimal Q-values. The main objective of the target network is to stabilize the training process by decoupling the target Q-values used in the Bellman equation from the Q-values that are updated during training. In other words, the target network is used to generate the target Q-values needed to compute the loss function during training, while the Q-network is updated to minimize the loss.

Exploration: Exploration is the process of randomly selecting actions for the agent to take in order to strike a balance between utilizing the current best policy and exploring new states and actions. It is important to explore the environment in order to avoid getting stuck in local optima and to establish the global optimal policy. Epsilon-greedy is a popular exploration technique in which the agent chooses the optimal action with probability (1-epsilon) and a random action with probability epsilon.

The replay buffer is a memory buffer used for storing the agent's past experiences, which are then randomly sampled during training to update the Q-network. The replay buffer enables the agent to learn from previous experiences while avoiding catastrophic forgetting of previous knowledge. The agent can learn from a wide range of experiences by randomly sampling from the replay buffer, improving the stability and efficiency of the learning process.

Explain what is different between DQN and Q-learning.

Overall, DQN can handle high-dimensional state spaces and has shown performance in a variety of environments, although it is computationally expensive and requires careful tuning of hyperparameters. In contrast, Q-learning is a simpler algorithm that is frequently used as a baseline for comparison with other reinforcement learning methods.

Compare the performance of every method and do some discussions in your report.

	Average Score	**Win rate (%)**
Minimax	**-59.375**	**15**
Expectimax	**103.475**	**21**
Q-learning(epsilon=0.05, alpha=0.2)	**-416.4**	**0**
Approximate Q-learning	**748.73**	**80**
DQN	**1233.67**	**77**

All the statics shown above is under the condition that layout = smallClassic, n = 200 (x = 100).

In terms of Minimax search and Expectimax search, we can see that the latter outperforms the former in terms of both average score and win rate.
In terms of Q-learning and approximate Q-learning, we can see that the former has a zero victory rate. In my own modest view, this is due to the layout being too wide to select optimal policy because there are too many options. In contrast, we may achieve a high average score and victory rate with approximation Q-learning.

I suppose this is because, given a set of features, it can select the optimal state as the next one based on the updated weight.
Furthermore, the search approach will outperform Q-learning. Because Q-learning is a type of reinforcement learning, I believe it will optimize the reward based on the environment, whereas the search approach will not.

To fix the difficulty that we encountered in smallClassic for Q-learning, where the information is too much to store and calculate, failure occurs. Deep Q-learning obtains the Q-value using a deep neural network rather than constructing a table to store the value. It is surprising that DQN doesn't outperform approximate Q-learning. Nonetheless, I suspect that it is due to a lack of resources on my pc.

Describe problems you meet and how you solve them:

For part 3 :

This error : OMP: Error #15: Initializing libiomp5.dylib, but found libomp.dylib already initialized.
OMP: Hint This means that multiple copies of the OpenMP runtime have been linked into the program. That is dangerous, since it can degrade performance or cause incorrect results. The best thing to do is to ensure that only a single OpenMP runtime is linked into the process, e.g. by avoiding static linking of the OpenMP runtime in any library. As an unsafe, unsupported, undocumented workaround you can set the environment variable KMP_DUPLICATE_LIB_OK=TRUE to allow the program to continue to execute, but that may cause crashes or silently produce incorrect results. For more information, please see http://www.intel.com/software/products/support/.
zsh: abort python pacman.py -p PacmanDQN -n 10000 -x 10000 -l smallClassic

I spent a long-time figuring thing out. With the help of stack overflow, I solve this error.