

# hw3\_111511198




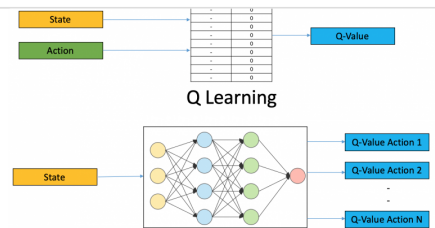
Author: 111511198 Heng-An Cheng

完整原始檔案見此

hw3\_111511198

Author: 111511198 Heng-An Cheng

 [https://ohmygod0193.notion.site/hw3\\_111511198-47576393ebab4d29ac616234b53d550e](https://ohmygod0193.notion.site/hw3_111511198-47576393ebab4d29ac616234b53d550e)



## Part 1

### Part1-1 Minimax Search

#### Code & Explanation

```
"""
Returns the optimal score and action for the given state and agent.
"""
def minimax(state, depth, agent):

    """
    Check if the game is finished or not
    """
    if state.isWin() or state.isLose() or depth == 0:
        return self.evaluationFunction(state), None

    actions = state.getLegalActions(agent)
    """
    if the agent is Pacman, we need to find the max score
    among the nextstate connect to current state.
    """
    if agent == 0: #agent=0 -> Pacman
        bestScore = float('-inf')
        for action in actions:
            nextState = state.getNextState(agent, action)
            """
            After Pacman's turn, pass to Pacman's turn.
            """
            score, _ = minimax(nextState, depth, 1)
            if score > bestScore:
                bestScore, bestAction = score, action
    """
    if the agent is ghost, we need to find the min score
    (which represents the better result for ghosts)
```

```

among the nextstate connect to current state.
"""
else:# otherwise -> ghosts
    bestScore = float('-inf')
    for action in actions:
        nextState = state.getNextState(agent, action)
        """
        if current agent is the last ghost,
        then we need to change into Pacman's turn, and make depth(round)-1
        """
        if agent == state.getNumAgents() - 1:
            score, _ = minimax(nextState, depth - 1, 0)
        else:
            score, _ = minimax(nextState, depth, agent + 1)
        if score < bestScore:
            bestScore, bestAction = score, action

    return bestScore, bestAction

_, action = minimax(gameState, self.depth, self.index)
return action

```

## Part1-2 Expectimax Search

### Code & Explanation

```

def expectimax(state, depth, agent):
    """
    Returns the optimal score and action for the given state and agent.
    """

    """
    Check if the game is finished or not
    """
    if state.isWin() or state.isLose() or depth == 0:
        return self.evaluationFunction(state), None

    actions = state.getLegalActions(agent)
    """
    if the agent is Pacman, we need to find the max score
    among the nextstate connect to current state.
    """
    if agent == 0:
        bestScore = float('-inf')
        for action in actions:
            nextState = state.getNextState(agent, action)
            score, _ = expectimax(nextState, depth, 1)
            if score > bestScore:
                bestScore, bestAction = score, action
    """
    if the agent is ghost, we need to find the expected score(sum/num)
    and random choose an action from all actions
    """
    else:
        scores = []
        for action in actions:
            nextState = state.getNextState(agent, action)
            if agent == state.getNumAgents() - 1:
                score, _ = expectimax(nextState, depth - 1, 0)
            else:
                score, _ = expectimax(nextState, depth, agent + 1)

```

```

        scores.append(score)
        bestScore = sum(scores) / len(scores)
        bestAction = random.choice(actions)

    return bestScore, bestAction

_, action = expectimax(gameState, self.depth, self.index)
return action

```

## Part 2

### Part 2-1 Value Iteration

#### Code & Explanation

```

def runValueIteration(self):
    # Begin your code
    for i in range(self.iterations):
        # Initialize a dictionary to store the updated values for each state
        new_values = util.Counter()
        # Iterate over all states
        for state in self.mdp.getStates():
            # If the state is terminal, set its value to 0
            if self.mdp.isTerminal(state):
                new_values[state] = 0
            else:
                # Find the best action to take in the current state
                best_action = None
                best_value = float("-inf")
                for action in self.mdp.getPossibleActions(state):
                    action_value = 0
                    statesandprobs = self.mdp.getTransitionStatesAndProbs(state, action)
                    for next_state, prob in statesandprobs :
                        # Calculate the expected value of taking the current action
                        reward = self.mdp.getReward(state, action, next_state)
                        Future_reward = self.discount*self.getValue(next_state)
                        action_value += prob*(reward + Future_reward)
                    # Update the best action and value found so far
                    if action_value > best_value:
                        best_action = action
                        best_value = action_value
                # Update the value of the current state with the best value found
                new_values[state] = best_value
        # Update the agent's values with the new values
        self.values = new_values
    # 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

```

```

# According to the formula to calculate the Q_value
Q_value = 0
for next_state, prob in self.mdp.getTransitionStatesAndProbs(state, action):
    reward = self.mdp.getReward(state, action, next_state)
    Q_value += prob * (reward + self.discount * self.getValue(next_state))
return Q_value
# End your code

```

```

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.
    """
    # Begin your code
    legal_actions = self.mdp.getPossibleActions(state)
    if not legal_actions:
        return None
    if self.mdp.isTerminal(state):
        return None
    best_action = None
    best_Q_value = float('-inf')
    #find the action which has the maximum Q_value
    for action in legal_actions:
        Q_value = self.computeQValueFromValues(state, action)
        if Q_value > best_Q_value:
            best_Q_value = Q_value
            best_action = action
    return best_action
# End your code

```

## Part 2-2 Q-learning

```

def __init__(self, **args):
    "You can initialize Q-values here..."
    ReinforcementAgent.__init__(self, **args)
    """ YOUR CODE HERE """
    # Begin your code
    self.q_values = util.Counter()
    self.epsilon = args['epsilon']
    self.alpha = args['alpha']
    self.discount = args['gamma']
    # 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

```

```

if (state,action) in self.q_values:
    return self.q_values[(state,action)]
return 0.0
# End your code

```

```

def computeValueFromQValues(self, state):
    """
    Returns max_action Q(state,action)
    where the max is over legal actions.
    """
    """ *** YOUR CODE HERE *** """
    # Begin your code
    """
    if there are no legal actions, which is the case at the
    terminal state, you should return a value of 0.0.
    """
    legalActions = self.getLegalActions(state)
    if not legalActions:
        return 0.0
    maxQValue = float("-inf")
    for action in legalActions:
        qValue = self.getQValue(state, action)
        if qValue > maxQValue:
            maxQValue = qValue
    return maxQValue
# End your code

```

```

def computeActionFromQValues(self, state):
    """
    Compute the best action to take in a state.
    """
    """ *** YOUR CODE HERE *** """
    # Begin your code
    legal_actions = self.getLegalActions(state)
    """
    if there are no legal actions, which is the case at the terminal state,
    you should return None.
    """
    if not legal_actions:
        return None
    best_action = []
    best_value = float('-inf')
    for action in legal_actions:
        value = self.getQValue(state,action)
        if value > best_value:
            best_value = value
            best_action = action
    return best_action
# End your code

```

## Part 2-3 epsilon-greedy action selection

```

def getAction(self, state):
    """

```

```

        Compute the action to take in the current state.
    """
    # Pick Action
    legal_Actions = self.getLegalActions(state)
    action = None
    """ *** YOUR CODE HERE *** """
    # Begin your code
    """
        if there are no legal actions, which is the case at the terminal state,
        you should choose None as the action.
    """
    if not legal_Actions:
        return
    else:
        """
            With probability self.epsilon, we should take a random action and
            take the best policy action otherwise.
        """
        if util.flipCoin(self.epsilon):
            action = random.choice(legal_Actions)
        """
            otherwise take the best policy action
        """
    else:
        action = self.getPolicy(state)
    return action
    # End your code

```

```

def update(self, state, action, nextState, reward):
    """
        The parent class calls this to observe a
        state = action => nextState and reward transition.
    """
    """ *** YOUR CODE HERE *** """
    # Begin your code
    """
        update Q_value through formula
        Q(s,a) <- Q(s,a) + alpha * (r + gamma * max_a' Q(s',a') - Q(s,a))
    """
    sampleEstimate = reward + self.discount * self.getValue(nextState)
    Qsa = self.getQValue(state, action)
    updatedQsa = (1 - self.alpha) * Qsa + self.alpha * sampleEstimate
    self.q_values[(state, action)] = updatedQsa
    # End your code

```

## 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
    features = self.featExtractor.getFeatures(state, action)
    Q_value = 0
    """

```

```

        calculate the w dot feature
        """
        for feature in features:
            Q_value += self.weights[feature] * features[feature]
        return Q_value
    # End your code

```

```

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

```

## Part 3

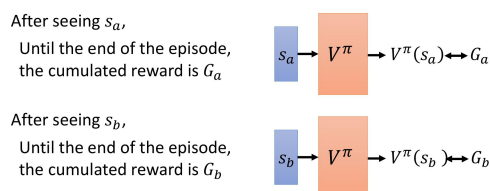
- What is the difference between On-policy and Off-policy ?
  - On-policy methods learn the value or policy based on the current policy being followed. In other words, the data used for learning comes from the same policy that is being updated. It directly optimizes the policy that is being evaluated, which can result in slow convergence if the exploration policy is too conservative. Examples of on-policy algorithms include Reinforce and Proximal Policy Optimization (PPO).
  - Off-policy methods, on the other hand, learn the value or policy based on data generated by a different policy. The data used for learning is typically collected using an exploratory policy, allowing for more efficient exploration. The advantage is that off-policy methods can learn from past experiences and re-use the data. Q-learning and Deep Q-Networks (DQN) are examples of off-policy algorithms.
- Briefly explain value-based, policy-based and Actor-Critic. Also, describe the value function  $V \cdot \pi(S)$  ?
  - Value-based methods focus on estimating the optimal value function or action-value function. They aim to find the best action to take in each state based on the estimated values. Examples are Q-learning and DQN.
  - Policy-based methods directly learn the policy without estimating the value function. They parameterize the policy and use gradient-based optimization techniques to update the policy parameters.

- Actor-Critic methods combine both value-based and policy-based approaches. They have an actor that learns the policy and a critic that estimates the value function. The critic provides feedback to the actor only by estimating the value of the state-action pairs.
- The value function  $V^\pi(S)$  provides an estimate of how good it is to be in state  $S$  under the policy  $\pi$ . A higher value indicates a more desirable state, as it represents the expected cumulative rewards that can be obtained by following the policy from that state.
- What is the difference between Monte-Carlo (MC) based approach and Temporal-difference (TD) approach for estimating  $V^\pi(S)$  ?
  - Monte-Carlo methods estimate the value function by averaging the returns observed from complete episodes. They require the agent to finish an episode before updating the value estimates, so they can only learn offline.
  - Temporal-Difference methods estimate the value function by bootstrapping from successor states. They update the value estimates at each time step based on the observed reward and the estimated value of the next state. TD methods combine ideas from both MC and dynamic programming and they can learn online.

#### Monte-Carlo

##### How to estimate $V^\pi(s)$

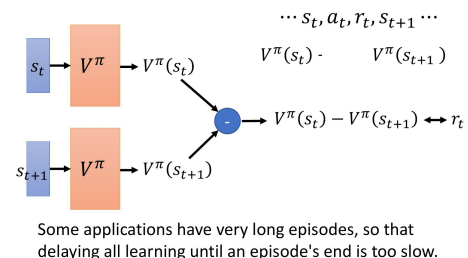
- Monte-Carlo based approach
  - The critic watches  $\pi$  playing the game



#### Temporal-difference

##### How to estimate $V^\pi(s)$

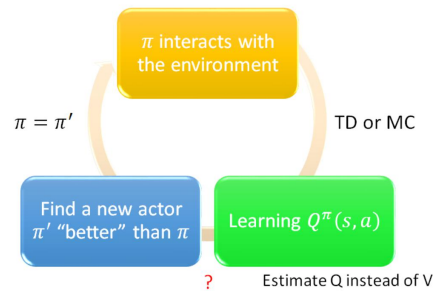
- Temporal-difference approach



- Describe State-action value function  $Q$  and the relationship between  $\pi(s, a)V$  in Q-learning.  $\pi(S)$  ?
  - The state-action value function  $Q(s,a)$  represents the expected return, or total discounted future rewards, when starting from state  $s$ , taking action  $a$ , and following a specific policy. Q-learning is an off-policy TD control algorithm that learns the optimal Q-function.



# Q-Learning



- Given  $Q^\pi(s, a)$ , find a new actor  $\pi'$  "better" than  $\pi$ 
  - "Better":  $V^{\pi'}(s) \geq V^\pi(s)$ , for all state  $s$

$$\pi'(s) = \arg \max_a Q^\pi(s, a)$$

- $\pi'$  does not have extra parameters. It depends on  $Q$
- Not suitable for continuous action  $a$

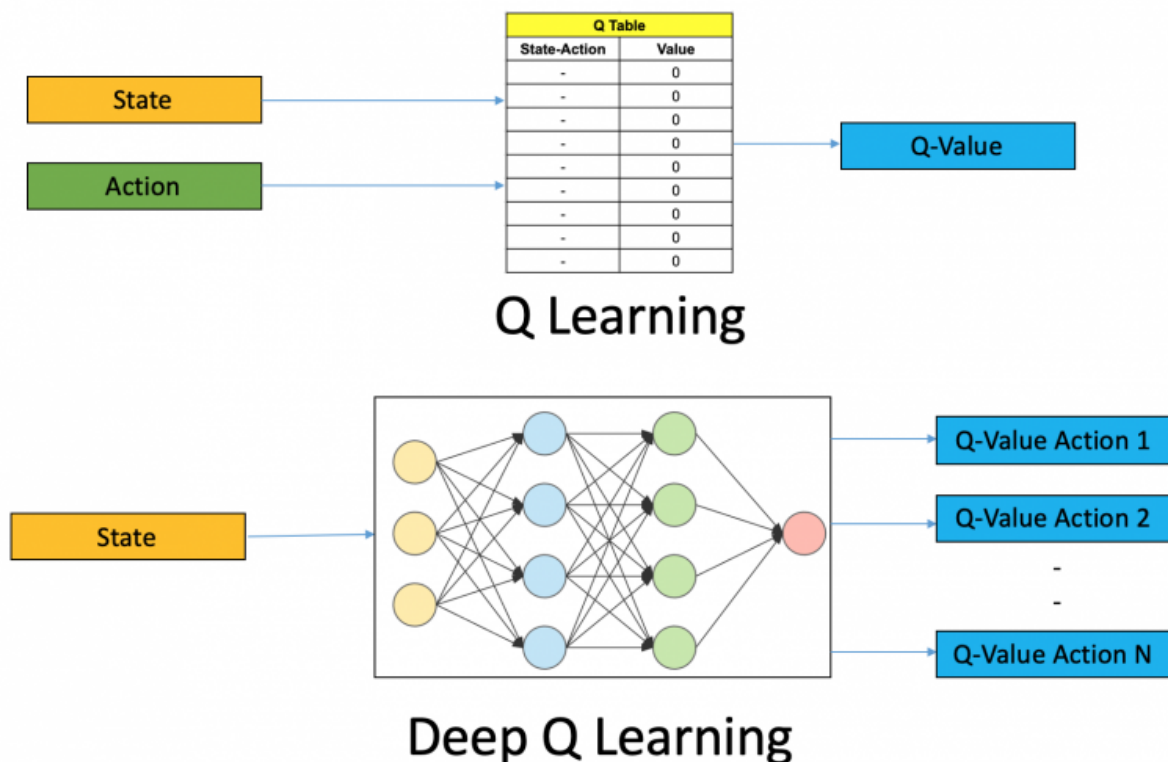
- The relationship between  $\pi(s,a)$ ,  $V$ , and  $Q$  in Q-learning is given by the optimal action-value function:  $Q^*(s,a)$ . This function represents the maximum expected return achievable by following any policy  $\pi$ , after taking action  $a$  in state  $s$ .  $Q^*(s,a)$  can be used to determine the optimal policy by selecting the action with the highest value for each state.

$$Q(s, a) = r(s, a) + \gamma \max_a Q(s', a)$$

- Describe following tips Target Network, Exploration and Replay Buffer using in Q-learning.
  - The **target network** in Q-learning is a separate copy of the Q-network. While the Q-network estimates action-values, the target network generates target values for updating the Q-network. **The target network parameters are updated less frequently than the Q-network to stabilize the learning process. Without the target network, the target values would be computed based on the same Q-network being updated, leading to instability.** By periodically updating the target network with the Q-network's parameters, **the learning process becomes more stable and reliable.**
  - **Exploration** is a crucial aspect of reinforcement learning, including Q-learning. It involves actively **seeking out new actions to gather information about the environment.** In Q-learning, a common strategy is **epsilon-greedy exploration**, where the agent primarily selects the action with the highest estimated value but occasionally chooses a random action with a small probability. **Exploration is essential for**

discovering new states, actions, and improving the agent's overall performance in the learning process.

- **Replay Buffer** is a memory structure used in Q-learning to **store experiences encountered by the agent during interactions** with the environment. Instead of immediately updating the Q-network after each experience, the agent stores these experiences in the replay buffer. **During the learning phase, a batch of experiences is randomly sampled from the replay buffer to train the Q-network.**
- Explain what is different between DQN and Q-learning.
  - The difference between these two methods is the Q-table. DQN is the method that replaces the original Q-table of Q-learning with a neural network to approximate the Q-value of every state-action pair. The reason to use a neural network instead of a Q-table, is because the Q-table doesn't scale well or too large. Another reason is that with a Q-table, it's not possible to have continuous states or actions.



## The performance of every method

### Minimax

```
python pacman.py -p MinimaxAgent -l minimaxClassic -a depth=4
```

```
Average Score: 213.3
Scores:        516.0, 516.0, 516.0, 516.0, -492.0, 516.0, 516.0, -492.0, -495.0, 516.0
Win Rate:      7/10 (0.70)
Record:        Win, Win, Win, Win, Loss, Win, Win, Loss, Loss, Win
```

## Expectimax

```
python pacman.py -p ExpectimaxAgent -l minimaxClassic -a depth=4 -n 10
```

```
Average Score: 212.5
Scores:        516.0, -493.0, -493.0, 516.0, 516.0, 516.0, -497.0, 516.0, 516.0, 512.0
Win Rate:      7/10 (0.70)
Record:        Win, Loss, Loss, Win, Win, Win, Loss, Win, Win, Win
```

## Approximate

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

```
Average Score: 730.6
Scores:        978.0, -66.0, 979.0, 958.0, 963.0, 978.0, 980.0, 974.0, 961.0, -399.0
Win Rate:      8/10 (0.80)
Record:        Win, Loss, Win, Win, Win, Win, Win, Win, Win, Loss
```

```
python pacman.py -p ApproximateQAgent -a extractor=SimpleExtractor -x 50 -n
60
```

```
Average Score: 1071.6
Scores:        1334.0, 1319.0, -44.0, 1323.0, 1326.0, 1299.0, 215.0, 1335.0, 1318.0, 1291.0
Win Rate:      8/10 (0.80)
Record:        Win, Win, Loss, Win, Win, Win, Loss, Win, Win, Win
```

## DQN

```
python pacman.py -p PacmanDQN -n 60 -x 50 -l smallClassic
```

```
Average Score: 1360.9
Scores:        1577.0, 755.0, 1472.0, 1536.0, 1689.0, 1568.0, 1560.0, 1365.0, 1728.0, 359.0
Win Rate:      8/10 (0.80)
Record:        Win, Loss, Win, Win, Win, Win, Win, Win, Win, Loss
```

# Discussion on the Performance of These Methods

Based on the experiments I have conducted, I have observed certain aspects regarding the performance of different methods. Specifically, I noticed that both the Minimax and Expectimax agents tend to exhibit a "wait-and-see" approach, where they consider the actions of ghosts before making their own decisions. Consequently, when these two agents are employed in a normal game setting, it takes a considerable amount of time to complete the game. Additionally, due to their algorithmic design that incorporates the consideration of random ghost actions, the win rates of these methods typically hover around 50%.

On the other hand, the performance of the remaining two methods, namely Approximate Q-learning and DQN agents, surprised me greatly. These methods not only perform exceptionally well but also outperform my own abilities. However, I made an intriguing observation in the case of the Approximate Q-learning agent: it rarely consumes "Power Pellets." Unfortunately, I haven't been able to determine the underlying reason for this behavior yet.

## Reference

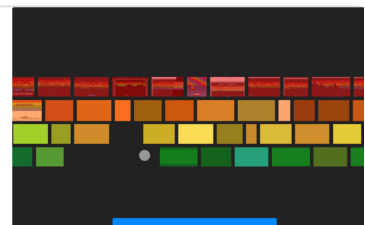
AINTU 講義

<https://ai.ntu.edu.tw/resource/handouts/ML23-2.html>

A Hands-On Introduction to Deep Q-Learning using OpenAI Gym in Python

An Introduction To Deep Reinforcement Learning. Learn about deep Q-learning, and build a deep Q-learning model in Python using keras and gym.

 <https://www.analyticsvidhya.com/blog/2019/04/introduction-deep-q-learning-python/>



<https://towardsdatascience.com/techniques-to-improve-the-performance-of-a-dqn-agent-29da8a7a0a7e>