Part 1

1. Minimax Search

```python
class MinimaxAgent(MultiAgentSearchAgent):
    def getAction(self, gameState):
        "*** YOUR CODE HERE ***"
        # Begin your code
        def minimax(agent, depth, gameState):
            if gameState.isWin() or gameState.isLose():
                return self.evaluationFunction(gameState), None
            if depth == self.depth:
                return self.evaluationFunction(gameState), None

            next_agent = agent + 1
            if next_agent >= gameState.getNumAgents():
                next_agent = 0
                depth += 1
            moves = gameState.getLegalActions(agent)
            if not moves:
                return self.evaluationFunction(gameState), None

            results = []
            for move in moves:
                next_state = gameState.getNextState(agent, move)
                score, _ = minimax(next_agent, depth, next_state)
                results.append((score, move))
            if agent == 0:
                # Pacman's turn, maximize the value
                best_value = max(results, key=lambda x: x[0])
            else:
                # Ghost's turn, minimize the value
                best_value = min(results, key=lambda x: x[0])
            return best_value

        _, action = minimax(0, 0, gameState)
        return action  # Return the action component of the tuple
        # End your code
```

The **getAction** method uses a recursive **minimax** function to evaluate game states. The function checks for terminal states (win or loss) and the maximum search depth as base cases to return the evaluated score. It then determines the next agent's turn and increases depth appropriately after each cycle of agents. For each possible move, it recursively evaluates future game states, collecting scores for each move. Pacman aims to maximize his score, while ghosts aim to minimize it. The algorithm ultimately returns the optimal move for Pacman based on the evaluated outcomes of all possible actions.

2. Expectimax Search

```python
class ExpectimaxAgent(MultiAgentSearchAgent):
    def getAction(self, gameState):
        "*** YOUR CODE HERE ***"
        # Begin your code
        def expectimax(agent, depth, gameState):
            if gameState.isWin() or gameState.isLose() or depth == self.depth
                * gameState.getNumAgents():
                return self.evaluationFunction(gameState)

            actions = gameState.getLegalActions(agent)
            if agent == 0:
                # Pacman's turn, maximize
                return max(expectimax((agent + 1) % gameState.getNumAgents(),
                    depth + 1, gameState.getNextState(agent, action))
                           for action in actions)
            else:
                # Ghosts' turn, expected value
                return sum(expectimax((agent + 1) % gameState.getNumAgents(),
                    depth + 1, gameState.getNextState(agent, action))
                           for action in actions) / len(actions)

        # Start from Pacman (agent 0) and depth 0
        bestAction = max(gameState.getLegalActions(0), key=lambda x:
            expectimax(1, 1, gameState.getNextState(0, x)))
        return bestAction
        # End your code
```

The method employs a recursive **expectimax** function to evaluate game states by considering all possible future scenarios within a specified depth. The function checks for terminal states (win or loss) and stops further recursion when it reaches the maximum depth. For each agent's turn, Pacman aims to maximize his expected utility by choosing the move with the highest score, while the ghosts calculate the expected utility across all possible moves, reflecting their average impact on the game state. The best move for Pacman is selected by evaluating the expected outcomes of all legal actions from the initial state.

Part 2

1. Value Iteration

```python
class ValueIterationAgent(ValueEstimationAgent):
    def runValueIteration(self):
        "*** YOUR CODE HERE ***"
        # Begin your code
        for _ in range(self.iterations):
            newValues = util.Counter()
            for state in self.mdp.getStates():
                if self.mdp.isTerminal(state):
                    continue
                max_value = float('-inf')
                for action in self.mdp.getPossibleActions(state):
                    q_value = self.computeQValueFromValues(state, action)
                    if q_value > max_value:
                        max_value = q_value
                newValues[state] = max_value if max_value != float('-inf')
                    else 0
            self.values = newValues
        # End your code

    def computeQValueFromValues(self, state, action):
        "*** YOUR CODE HERE ***"
        # Begin your code
        q_value = 0
        trans = self.mdp.getTransitionStatesAndProbs(state, action)
        for nextState, prob in trans:
            reward = self.mdp.getReward(state, action, nextState)
            q_value += prob * (reward + self.discount * self.values[nextState
                ])
        return q_value
        # End your code

    def computeActionFromValues(self, state):
        "*** YOUR CODE HERE ***"
        # Begin your code
        #check for terminal
        if self.mdp.isTerminal(state):
            return None
        best_action = None
        max_value = float('-inf')
        for action in self.mdp.getPossibleActions(state):
            q_value = self.computeQValueFromValues(state, action)
            if q_value > max_value:
                max_value = q_value
                best_action = action
        return best_action
        # End your code
```

The **runValueIteration** method performs the main iteration loop for a specified number of iterations. In each iteration, it updates the value for each non-terminal state based on the maximum expected utility, calculated through the **computeQValueFromValues** method, which computes the Q-value for a given state-action pair by summing the expected utilities of possible future states. The algorithm assumes future rewards are discounted by a factor. Once the iterations are complete, the **computeActionFromValues** method determines the optimal action for any state by selecting the action with the highest Q-value.

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
        self.qValues = util.Counter()
        # End your code


    def getQValue(self, state, action):
        "*** YOUR CODE HERE ***"
        # Begin your code
        return self.qValues[(state, action)]
        # End your code

    def computeValueFromQValues(self, state):
        "*** YOUR CODE HERE ***"
        # Begin your code
        legalActions = self.getLegalActions(state)
        if not legalActions:
            return 0.0
        return max(self.getQValue(state, action) for action in legalActions)
        # End your code

    def computeActionFromQValues(self, state):
        "*** YOUR CODE HERE ***"
        # Begin your code
        legalActions = self.getLegalActions(state)
        if not legalActions:
            return None
        bestVal = self.computeValueFromQValues(state)
        bestActions = [action for action in legalActions if self.getQValue(
            state, action) == bestVal]
        return random.choice(bestActions)
        # End your code

    def update(self, state, action, nextState, reward):
        "*** YOUR CODE HERE ***"
        # Begin your code
        sample = reward + self.discount * self.computeValueFromQValues(
```

```
            nextState)
41          self.qValues[(state, action)] = (1 - self.alpha) * self.getQValue(
            state, action) + self.alpha * sample
42          # End your code
```

- Q-value Retrieval
  The **getQValue** method retrieves the current Q-value for a given state-action pair, defaulting to zero if it has not been previously encountered.
- Value Calculation
  The **computeValueFromQValues** method determines the highest Q-value for any action available from a given state, returning zero if there are no legal actions.
- Action Determination
  The **computeActionFromQValues** method identifies the action that yields the highest Q-value for a given state. If multiple actions have the same value, one is randomly selected, providing an exploration mechanism.
- Q-value Update
  The **update** method adjusts the Q-values based on new data from the environment. It computes a sample Q-value using the reward received for taking an action and the discounted maximum Q-value of the resulting state. The Q-value for the state-action pair is then updated towards this sample, scaled by a learning rate.

3. Epsilon-greedy Action Selection

```
1  class QLearningAgent(ReinforcementAgent):
2      def getAction(self, state):
3          # Pick Action
4          legalActions = self.getLegalActions(state)
5          action = None
6          "*** YOUR CODE HERE ***"
7          # Begin your code
8          if legalActions:
9              if util.flipCoin(self.epsilon):
10                 action = random.choice(legalActions)
11             else:
12                 action = self.computeActionFromQValues(state)
13         return action
14         # End your code
```

The **getAction** method in the **QLearningAgent** class determines the next action for the agent to take from a given state. It first retrieves all legal actions available from the current state. The method then employs an ϵ-greedy strategy to balance exploration and exploitation:

- Exploration
  With a probability of ϵ, the agent randomly selects an action from the available legal actions. This randomness introduces exploration, allowing the agent to discover new strategies.
- Exploitation
  With a probability of $1 - \epsilon$, the agent selects the action that maximizes the expected utility based on current Q-values, calculated through the **computeActionFromQValues** method. This approach ensures that the agent exploits its current knowledge to maximize the expected reward.

4. Approximate Q-learning (Bonus)

```python
class ApproximateQAgent(PacmanQAgent):
    def getQValue(self, state, action):
        "*** YOUR CODE HERE ***"
        # Begin your code
        # get weights and feature
        features = self.featExtractor.getFeatures(state, action)
        return sum(self.weights[f] * value for f, value in features.items())
        # End your code

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

- Q-value Computation

  The **getQValue** method calculates the estimated Q-value for a given state-action pair. It does this by extracting relevant features of the state-action context through featExtractor.getFeatures, and then computing the dot product of these features with their corresponding weights. This results in a linear approximation of the Q-value based on the weighted sum of the features.

- Q-value Update

  For each feature in the current state-action pair, the corresponding weight is updated to better align the predicted Q-values with the observed values. The update rule applies a learning rate **alpha**, the correction term, and the feature's value to adjust the weight, promoting convergence to optimal policies over time.

Results:

```
$ python pacman.py -p ApproximateQAgent -a extractor=SimpleExtractor -x 50 -n
    60 -l mediumGrid
Average Score: 527.8
Scores:        529.0, 527.0, 525.0, 527.0, 529.0, 529.0, 529.0, 525.0, 529.0,
    529.0
Win Rate:      10/10 (1.00)
Record:        Win, Win, Win, Win, Win, Win, Win, Win, Win, Win
```

Part 3

Results:

```
$ python pacman.py -p PacmanDQN -n 25 -x 20 -l smallClassic -q
Average Score: 1391.6
Scores:        964.0, 1512.0, 1179.0, 1764.0, 1539.0
Win Rate:      5/5 (1.00)
Record:        Win, Win, Win, Win, Win
```

1. What is the difference between On-policy and Off-policy?
   - **On-policy** methods learn the value of the policy being carried out by the agent, including the exploration steps. The policy that is evaluated and improved is the same policy used to make decisions. A classic example of an on-policy method is **SARSA**.

$$Q(S,\ A) \leftarrow Q(S,\ A) + \alpha[R + \gamma Q(S',\ A') - Q(S,\ A)]$$

$$S \leftarrow S',\ A \leftarrow A'$$

   - **Off-policy** methods learn the value of an optimal policy independently of the agent's actions, including learning from actions that are outside the current policy. A prominent example of an off-policy method is **Q-learning**.

$$Q(S,\ A) \leftarrow Q(S,\ A) + \alpha[R + \gamma max_a Q(S', a) - Q(S,\ A)]$$

$$S \leftarrow S'$$

2. Briefly explain value-based, policy-based and Actor-Critic. Also, describe the value function $V^{\pi}(s)$.
   - Value-based Methods
     These methods focus on finding the value function that estimates how good it is for an agent to be in a given state. The policy is derived from the value function, typically by choosing the action that maximizes the expected value. Q-learning and Value Iteration are common examples of value-based methods.
   - Policy-based Methods
     These methods directly parameterize and optimize the policy that dictates the action to be taken in each state. They do not explicitly compute value functions. Policy Gradient is a typical example where the policy is directly optimized through gradients.
   - Actor-Critic Methods
     These combine features of both value-based and policy-based methods. The "actor" updates the policy distribution in the direction suggested by the "critic," which evaluates the action taken by the actor according to a value function. This approach aims to use the strengths of both value estimation and policy optimization.
   - Value Function $V^{\pi}(s)$
     The value function under a policy π, denoted as $V^{\pi}(s)$, is the expected return (sum of discounted rewards) starting from state s and following policy π thereafter. Mathematically, it's given by:

$$V^{\pi}(s) = \mathbb{E}[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s, \pi]$$

3. What is the difference between Monte-Carlo (MC) based approach and Temporal-difference (TD) approach for estimating $V^{\pi}(s)$?
   - Monte-Carlo Methods
     These methods estimate $V^{\pi}(s)$ by averaging the returns observed after visiting a state s. This approach does not update estimates based on other estimates, and each update requires a complete episode to finish, making it only applicable to episodic tasks.
   - Temporal-Difference Methods: TD methods estimate $V^{\pi}(s)$ by updating estimates based on other learned estimates, without waiting for the episode to end. They update estimates based partly on the observed reward and partly on estimates of future states. This allows TD methods to learn online and from incomplete episodes.
4. Describe State-action value function $Q^{\pi}(s,\ a)$ and the relationship between $V^{\pi}(s)$ in Q-learning.
   The state-action value function $Q^{\pi}(s,\ a)$ represents the expected return starting from state s, taking an action a, and thereafter following policy π. The relationship between $Q^{\pi}(s,\ a)$ and $V^{\pi}(s)$ is given by:
   $$V^{\pi}(s) = \sum_{a \in A} \pi(a|s) Q^{\pi}(s,\ a)$$
   In Q-learning, which is a value-based method, the optimal state-action value function $Q^{*}(s,\ a)$ is learned, and the optimal policy can be derived by selecting the action that maximizes $Q^{*}(s,\ a)$ at each state.
5. Describe following tips Target Network, Exploration and Replay Buffer using in Q-learning.
   - Target Network
     In Deep Q-learning, a separate network (target network) is used to estimate the target values during updates, which helps in stabilizing the learning. The target network's weights are periodically updated to the weights of the primary network.
   - Exploration
     This refers to the technique used to balance the exploration of the environment (trying new things) and exploitation (using known information). Common strategies include ϵ-greedy, where ϵ represents the probability of choosing a random action.
   - Replay Buffer
     This is a technique where experiences (state, action, reward, next state) are stored in a buffer and sampled randomly to update the network. This helps in breaking the correlation between consecutive learning updates and stabilizing learning.
6. Explain what is different between DQN and Q-learning.
   - **DQN** extends Q-learning by using a deep neural network to approximate the Q-value function. Unlike traditional Q-learning, which uses a table to store Q-values, DQN can handle environments with high-dimensional state spaces.
   - **Q-learning** is a tabular method generally suitable for smaller discrete state spaces and can suffer from scalability issues in complex environments. DQN, on the other hand, can generalize across similar states due to its function approximation approach.

Comparison

The following table shows a clear progression in performance from simpler to more advanced Pacman agents. The ReflexAgent performs poorly with the lowest score and no wins, while Minimax and Expectimax show moderate improvements, with Expectimax notably achieving a positive average score and a 23% win rate. The advanced ApproximateQAgent and PacmanDQN display the best performance, achieving the highest win rates and scores respectively.

	ReflexAgent	MinimaxAgent	ExpectimaxAgent	ApproximateQAgent	PacmanDQN
AvgScore	-284.25	-127.0	134.46	816.62	1222.83
WinRate	0/100 (0.00)	6/100 (0.06)	23/100 (0.23)	87/100 (0.87)	80/100 (0.80)

1. ReflexAgent

```
$ python pacman.py -p ReflexAgent -n 100 -l smallClassic -q
Average Score: -284.25
Scores:        -261.0, -286.0, -153.0, -222.0, -446.0, -365.0, -223.0, -153.0, -385.0, -410.0,
    -283.0, -293.0, -295.0, -366.0, -266.0, -318.0, -287.0, -242.0, -62.0, -347.0, -437.0,
    -446.0, -134.0, -302.0, -315.0, -146.0, -393.0, -446.0, -298.0, -150.0, -428.0, -376.0,
    -335.0, -392.0, -304.0, -67.0, -392.0, -248.0, -419.0, -101.0, -321.0, -343.0, -420.0,
    -357.0, -329.0, -124.0, -392.0, -419.0, -349.0, -429.0, -72.0, -300.0, -32.0, -438.0,
    -113.0, -257.0, -428.0, -275.0, -332.0, 69.0, -358.0, -320.0, -297.0, -397.0, -100.0,
    -280.0, -331.0, -305.0, -326.0, -419.0, 104.0, -446.0, -419.0, 233.0, -91.0, -338.0, -96.0,
    -447.0, -374.0, -338.0, -322.0, -446.0, -313.0, -348.0, -280.0, -385.0, -376.0, -321.0,
    -372.0, -410.0, -447.0, 249.0, -239.0, -332.0, 65.0, -321.0, -383.0, -55.0, -383.0, -208.0
Win Rate:      0/100 (0.00)
Record:        Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss
    , Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss,
    Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss,
    Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss,
    Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss,
    Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss,
    Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss
```

2. MinimaxAgent

```
$ python pacman.py -p MinimaxAgent -n 100 -l smallClassic -q
Average Score: -127.0
Scores:        -196.0, -167.0, -174.0, 67.0, 140.0, -175.0, -329.0, -174.0, -94.0, -448.0, 14.0,
    -216.0, -613.0, 1257.0, -519.0, -315.0, -389.0, -47.0, 1095.0, -372.0, 1077.0, -16.0, 64.0,
    -336.0, -444.0, -33.0, -273.0, -374.0, -192.0, -360.0, -215.0, -260.0, 0.0, 408.0, -219.0,
    -49.0, -240.0, 467.0, 157.0, -202.0, -182.0, -266.0, -175.0, -290.0, -241.0, -73.0, -162.0,
    -495.0, 33.0, -184.0, -161.0, -353.0, -262.0, -140.0, -516.0, 1043.0, 38.0, -184.0, -252.0,
    -206.0, -28.0, -260.0, -220.0, -184.0, -68.0, 47.0, -335.0, -376.0, 198.0, -214.0, -180.0,
    -134.0, -464.0, -105.0, 408.0, -212.0, -310.0, -211.0, -368.0, 148.0, -29.0, -221.0, -386.0,
     -174.0, -399.0, -455.0, -308.0, -128.0, -328.0, -329.0, -159.0, -409.0, -226.0, 770.0,
    23.0, -249.0, -423.0, 16.0, -594.0, -101.0
Win Rate:      6/100 (0.06)
Record:        Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Win,
     Loss, Loss, Loss, Loss, Win, Loss, Win, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss,
    Loss, Loss, Loss, Loss, Win, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss,
    Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Win, Loss, Loss, Loss,
    Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss,
    Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss,
    Loss, Loss, Loss, Loss, Win, Loss, Loss, Loss, Loss, Loss, Loss
```

3. ExpectimaxAgent

```
$ python pacman.py -p ExpectimaxAgent -n 100 -l smallClassic -q
Average Score: 134.46
Scores:        21.0, 584.0, 608.0, -79.0, -323.0, -165.0, -204.0, 157.0, -235.0, -197.0, 590.0,
    -157.0, 371.0, 83.0, -132.0, 1093.0, -192.0, -148.0, -261.0, -358.0, -131.0, -153.0, -192.0,
     -182.0, 251.0, 1052.0, -13.0, 122.0, 1264.0, -137.0, 845.0, -355.0, 107.0, 545.0, 1277.0,
    43.0, -87.0, 910.0, 461.0, -167.0, -155.0, -377.0, -465.0, 4.0, 1115.0, 1004.0, -245.0,
    -199.0, -956.0, 401.0, 836.0, -250.0, 1421.0, -209.0, 1290.0, -251.0, -96.0, -201.0, -293.0,
     284.0, -24.0, -420.0, -133.0, 1191.0, 1053.0, -229.0, -162.0, 48.0, -323.0, 1043.0, 119.0,
    28.0, 110.0, -219.0, 1321.0, -133.0, -191.0, -229.0, -143.0, -87.0, 277.0, -243.0, -76.0,
    214.0, -126.0, 219.0, 1439.0, 327.0, -1.0, -117.0, -183.0, 306.0, 467.0, -135.0, -246.0,
    55.0, -398.0, -540.0, 261.0, 152.0
Win Rate:      23/100 (0.23)
Record:        Loss, Loss, Win, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Win, Loss, Win, Loss,
    Loss, Win, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Win, Win, Loss, Win, Loss,
    Win, Loss, Loss, Win, Win, Loss, Loss, Win, Win, Loss, Loss, Loss, Loss, Loss, Win, Win,
    Loss, Loss, Loss, Loss, Win, Loss, Win, Loss, Win, Loss, Loss, Loss, Win, Loss, Loss, Loss,
    Loss, Win, Win, Loss, Loss, Loss, Loss, Win, Loss, Loss, Loss, Loss, Win, Loss, Loss, Loss,
    Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Win, Loss, Loss, Loss, Loss, Loss, Loss,
    Loss, Loss, Loss, Loss, Loss, Loss, Loss
```

4. ApproximateQAgent

```
$ python pacman.py -p ApproximateQAgent -a extractor=SimpleExtractor -n 200 -x 100 -l
    smallClassic -q
Average Score: 816.62
Scores:        972.0, 964.0, 975.0, 970.0, 974.0, 962.0, 981.0, 965.0, 970.0, 982.0, 960.0,
    985.0, 961.0, 977.0, 951.0, 987.0, 1164.0, -296.0, 964.0, -358.0, 1178.0, 969.0, 978.0,
    983.0, -154.0, 978.0, 957.0, 960.0, 968.0, 973.0, 962.0, 986.0, 981.0, -361.0, 977.0, 978.0,
     983.0, -105.0, 984.0, 960.0, 974.0, 980.0, -319.0, 976.0, 968.0, 981.0, 968.0, 968.0,
    968.0, 954.0, 971.0, 977.0, 968.0, 971.0, 981.0, 970.0, 969.0, 973.0, 974.0, 1154.0, 963.0,
    975.0, 980.0, 967.0, 970.0, -258.0, 987.0, 968.0, 969.0, 963.0, 973.0, 975.0, 983.0, 1161.0,
     976.0, 965.0, 975.0, 966.0, -380.0, 980.0, 975.0, 958.0, -260.0, -346.0, 961.0, -388.0,
    -313.0, 962.0, 951.0, 943.0, 981.0, 976.0, -243.0, 973.0, 972.0, 963.0, 958.0, 970.0,
    1171.0, 969.0
Win Rate:      87/100 (0.87)
Record:        Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win,
    Win, Loss, Win, Loss, Win, Win, Win, Win, Loss, Win, Win, Win, Win, Win, Win, Win, Win, Loss
    , Win, Win, Win, Loss, Win, Win, Win, Win, Loss, Win, Win, Win, Win, Win, Win, Win, Win, Win
    , Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Loss, Win, Win, Win, Win,
     Win, Win, Win, Win, Win, Win, Win, Win, Loss, Win, Win, Win, Loss, Loss, Win, Loss, Loss,
    Win, Win, Win, Win, Win, Loss, Win, Win, Win, Win, Win, Win, Win
```

5. PacmanDQN

```
$ python pacman.py -p PacmanDQN -n 1000 -x 900 -l smallClassic -q
Average Score: 1222.83
Scores:        956.0, 1745.0, 1771.0, 591.0, 1558.0, 1766.0, 1695.0, 471.0, 239.0, 1358.0,
    1353.0, 1152.0, 1545.0, 1573.0, 108.0, 1372.0, 1560.0, 1374.0, 1150.0, 1356.0, 688.0, 336.0,
     707.0, 1744.0, 1365.0, 1553.0, 1340.0, 1713.0, 1772.0, 609.0, 1551.0, 1535.0, 1764.0,
    1766.0, 247.0, 1156.0, -152.0, 1177.0, 1170.0, 1354.0, 1548.0, 166.0, 1100.0, 1484.0,
    1150.0, 1332.0, 1544.0, 1547.0, 408.0, 1511.0, 1737.0, 1552.0, 1171.0, 1751.0, 1229.0,
    1359.0, 1139.0, 202.0, 1746.0, 1690.0, 1364.0, 1732.0, 1538.0, 1563.0, 945.0, 1777.0, 163.0,
     1140.0, -195.0, 1334.0, 952.0, 423.0, 1511.0, 1507.0, -294.0, 1338.0, 1560.0, 1327.0,
    1555.0, 1739.0, 1560.0, 1360.0, 324.0, 1728.0, 1137.0, 968.0, 1760.0, 1337.0, 1568.0,
    1524.0, 1551.0, 1321.0, 1566.0, 1511.0, 1364.0, -120.0, 553.0, 1544.0, 1767.0, 1527.0
Win Rate:      80/100 (0.80)
Record:        Win, Win, Win, Loss, Win, Win, Win, Loss, Loss, Win, Win, Win, Win, Win, Loss, Win
    , Win, Win, Win, Win, Loss, Loss, Loss, Win, Win, Win, Win, Win, Win, Loss, Win, Win, Win,
    Win, Loss, Win, Loss, Win, Win, Win, Win, Loss, Win, Win, Win, Win, Win, Win, Loss, Win, Win
    , Win, Win, Win, Win, Win, Win, Loss, Win, Win, Win, Win, Win, Win, Win, Win, Loss, Win,
    Loss, Win, Win, Loss, Win, Win, Loss, Win, Win, Win, Win, Win, Win, Win, Loss, Win, Win, Win
    , Win, Win, Win, Win, Win, Win, Win, Win, Win, Loss, Loss, Win, Win, Win
```

Appendix

1. Reference
   (a) OpenAI. (2024). ChatGPT (4) [Large language model]. https://chat.openai.com
   (b) Hung-yi lee. (2021, May 22). 【機器學習 2021】概述增強式學習 (Reinforcement Learning, RL). https://shorturl.at/inFR4