Part 1-1: Minimax Search (10%)

```python
# Begin your code
def minimax_search(gameState, depth, agentIndex):
    # terminal states
    if gameState.isWin() or gameState.isLose() or depth == self.depth:
        return self.evaluationFunction(gameState)

    # Pacman's turn
    if agentIndex == 0:
        maxScore = -float("inf")
        actions = gameState.getLegalActions(0)

        # get maximum
        for action in actions:
            nextState = gameState.getNextState(0, action)
            maxScore = max(
                maxScore, minimax_search(nextState, depth, 1))

        return maxScore

    # ghosts' turns
    else:
        minScore = float("inf")
        actions = gameState.getLegalActions(agentIndex)

        # get minimum
        for action in actions:
            nextState = gameState.getNextState(agentIndex, action)
```

```python
            # next agent is the Pacman
            if agentIndex == (gameState.getNumAgents() - 1):
                minScore = min(minScore, minimax_search(
                    nextState, depth + 1, 0))

            # next agent is a ghost
            else:
                minScore = min(minScore, minimax_search(
                    nextState, depth, agentIndex + 1))

        return minScore

# initial condition
actions = gameState.getLegalActions(0)
maxScore = -float("inf")
returnAction = None

# get maximum and decide next action
for action in actions:
    nextState = gameState.getNextState(0, action)
    score = minimax_search(nextState, 0, 1)

    if score > maxScore:
        returnAction = action
        maxScore = score

return returnAction
# End your code
```

In the function minimax_search(), there are three parameters: gameState, depth and agentIndex. While gameState represents the current condition of the game, we record the depth of the search tree to check if we have gone through enough predictions. Every time when minimax_search() is called, we first check if it's in the terminal condition (win or lose), or we have reached the bottom of the search tree (by checking depth). Then, if agentIndex = 0, which means the Pacman should act, we perform a depth first search on all legal actions to find the maximum score of the particular action. On the other hand, if agentIndex > 0, which means a ghost should act, we also perform a depth first search on all legal actions to find the minimum score of the particular action. Notice that in ghosts' turns, we should check if the next agent is the Pacman or a ghost to ensure we transfer to the right state. Finally, set the initial condition (the Pacman starts first), and choose the action which will lead to maximum score gained.

```
*** 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%)

```python
class ExpectimaxAgent(MultiAgentSearchAgent):
    """
      Your expectimax agent (part1-2)
    """

    def getAction(self, gameState):
        """
        Returns the expectimax action using self.depth and self.evaluationFunction

        All ghosts should be modeled as choosing uniformly at random from their
        legal moves.
        """
        "*** YOUR CODE HERE ***"
        # Begin your code
        def expectimax_search(gameState, depth, agentIndex):
            # terminal states
            if gameState.isWin() or gameState.isLose() or depth == self.depth:
                return self.evaluationFunction(gameState)

            # Pacman's turn
            if agentIndex == 0:
                maxScore = -float("inf")
                actions = gameState.getLegalActions(0)

                # get maximum
                for action in actions:
                    nextState = gameState.getNextState(0, action)
                    maxScore = max(
                        maxScore, expectimax_search(nextState, depth, 1))
```

```python

                return maxScore

            # ghosts' turns
            else:
                actions = gameState.getLegalActions(agentIndex)
                averageScore = 0

                # get expected value (mean)
                for action in actions:
                    nextState = gameState.getNextState(agentIndex, action)

                    # next agent is the Pacman
                    if agentIndex == (gameState.getNumAgents() - 1):
                        averageScore += (expectimax_search(nextState,
                                         depth + 1, 0) / len(actions))

                    # next agent is a ghost
                    else:
                        averageScore += (expectimax_search(nextState,
                                         depth, agentIndex + 1) / len(actions))

                return averageScore
```

```python
        # initial condition
        actions = gameState.getLegalActions(0)
        maxScore = -float("inf")
        returnAction = None

        # get maximum and decide next action
        for action in actions:
            nextState = gameState.getNextState(0, action)
            score = expectimax_search(nextState, 0, 1)

            if score > maxScore:
                returnAction = action
                maxScore = score

        return returnAction
        # End your code


better = scoreEvaluationFunction
```

The only difference between part 1-1 and part 1-2 is that in expectimax_search(), we don’t expect the ghosts to choose the worst actions towards us (Pacman). In contrast, we assume the actions of ghosts are random (with equal probability). Therefore, we just need to change the formula (line 240 ~ 247) of the returned score in ghosts’ turns (agentIndex > 0).

```
*** 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 2-1: Value Iteration (10%)

```python
    def runValueIteration(self):
        # Write value iteration code here
        "*** YOUR CODE HERE ***"
        # Begin your code
        for i in range(self.iterations):
            tempValues = util.Counter()
            states = self.mdp.getStates()

            for state in states:
                if not self.mdp.isTerminal(state):
                    Q_values = util.Counter()
                    actions = self.mdp.getPossibleActions(state)

                    for action in actions:
                        Q_values[action] = self.computeQValueFromValues(
                            state, action)

                    tempValues[state] = max(Q_values.values())

            self.values = tempValues
        # End your code
```

In each iteration (the outer loop), we go through all states and all actions to find the maximum Q value for each state (by calling computeQValueFromValues()). Keep the value in a table (tempValues), and update self.values in every 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
        transitionStatesAndProbabilities = self.mdp.getTransitionStatesAndProbs(
            state, action)

        for (nextState, probability) in transitionStatesAndProbabilities:
            QValue += probability * \
                (self.mdp.getReward(state, action, nextState) +
                 self.discount * self.values[nextState])

        return QValue
        # End your code
```

In computeQValueFromValues(), we calculate Q value, which is the expected total reward, by summing the total reward of each action times the probability to make that action. The total reward involves current reward (self.mdp.getReward(state, action, nextState)) and the future reward, which will be multiplied by a discount factor (self.discount * self.values[nextState]).

```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.computeQValueFromValues(state, action)

            return QValues.argMax()
        # End your code
```

If the current state is a terminal state, then return no action (None). Otherwise, return the action which is corresponding with the largest Q value.

```
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 (10%)

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

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

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

It would return 0.0 if we have never seen a state since self.values = util.Counter() is initialized with 0.0.

```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)

        return 0.0 if not actions else max([self.getQValue(state, action) for action in actions])
        # End your code
```

If no legal actions, return 0. Otherwise, return the maximum Q value.

```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)
        maxQValue = self.computeValueFromQValues(state)

        return None if not actions else random.choice([action for action in actions if self.getQValue(state, action) == maxQValue])
        # End your code
```

As comments.

```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
        self.values[(state, action)] = (1 - self.alpha) * \
            self.values[(state, action)] + self.alpha * (reward +
                                                         self.discount * self.computeValueFromQValues(nextState))
        # End your code
```

By the formula:

$$q_\pi(s,a) = (1-\alpha)q_\pi(s,a) + \alpha[R + \gamma \max_{a'} q_\pi(s',a')]$$

```
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 (5%)

```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):
            return random.choice(legalActions)
        else:
            return self.computeActionFromQValues(state)
        # End your code
```

• You can also observe the following simulations for different epsilon values. Does that behavior of the agent match what you expect?

Answer: python gridworld.py -a q -k 100 --noise 0.0 -e 0.9

The behavior is as I expected (↑ ↑ →→→). The average returns from the start state will decrease as epsilon becomes large, due to the random actions when epsilon > 0.

```
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 (10%)

```python
    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
        return self.getWeights() * self.featExtractor.getFeatures(state, action)
        # End your code
```

As comments.

```python
    def update(self, state, action, nextState, reward):
        """
           Should update your weights based on transition
        """
        "*** YOUR CODE HERE ***"
        # Begin your code
        features = self.featExtractor.getFeatures(state, action)
        difference = reward + self.discount * \
            self.computeValueFromQValues(
                nextState) - self.getQValue(state, action)

        for feature in features:
            self.weights[feature] += self.alpha * \
                difference * features[feature]
        # End your code
```

By the formula:

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

```python
class SimpleExtractor(FeatureExtractor):
    """
    Returns simple features for a basic reflex Pacman:
    - whether food will be eaten
    - how far away the next food is
    - whether a ghost collision is imminent
    - whether a ghost is one step away
    """

    def getFeatures(self, state, action):
        # extract the grid of food and wall locations and get the ghost locations
        food = state.getFood()
        walls = state.getWalls()
        activeGhostPosition = []
        capsules = state.getCapsules()
        scaredGhost = []
        activeGhost = []
        features = util.Counter()

        for agentIndex, ghost in enumerate(state.getGhostStates(), start=1):
            if not ghost.scaredTimer:
                activeGhost.append(ghost)
                activeGhostPosition.append(
                    state.getGhostPosition(agentIndex))
            else:
                scaredGhost.append(ghost)
                features["safe-time"] = ghost.scaredTimer

        features["bias"] = 1.0
```

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

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

        # if there is no danger of ghosts then add the food feature
        if not features["#-of-active-ghosts-1-step-away"]:
            if (next_x, next_y) in capsules and not scaredGhost:
                features["eats-capsule"] = 1.0
            elif food[next_x][next_y]:
                features["eats-food"] = 1.0

        else:
            features["escape"] = 1.0

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

        features.divideAll(10.0)

        return features
```

```
Reinforcement Learning Status:
        Completed 100 test episodes
        Average Rewards over testing: 1245.32
        Average Rewards for last 100 episodes: 1245.32
        Episode took 6.61 seconds
Average Score: 1245.32
Scores:         983.0, 1182.0, 1576.0, 1364.0, 1362.0, 1570.0, 1178.0, 1180.0, 1177.0, 1371.0, 1371.0, 1184.0, 1382.0, 1380.0, 1168.0, 974.0, 1561.0, 1370.0, 1174.
0, 154.0, 1382.0, 975.0, 1172.0, 980.0, 1381.0, 1367.0, 1380.0, 1570.0, 1582.0, 981.0, 1172.0, 1365.0, 1377.0, 1371.0, 1369.0, 1552.0, 1170.0, 1173.0, 1178.0, 118
0.0, 1181.0, 1162.0, 1577.0, 1183.0, 1175.0, 1175.0, 1574.0, 1371.0, 1170.0, 1185.0, 980.0, 975.0, 1379.0, 1761.0, 977.0, 1180.0, 1367.0, 976.0, 1566.0, 1377.0, 1
167.0, 1362.0, 1368.0, 1378.0, 1371.0, 983.0, 1178.0, 1180.0, 1377.0, 1179.0, 1376.0, 966.0, 1161.0, 1170.0, 1167.0, 984.0, 1576.0, 1370.0, 1570.0, 984.0, 1563.0,
 1185.0, 1349.0, 1172.0, 1370.0, 1174.0, 1184.0, 1368.0, 1174.0, 973.0, 1366.0, 1180.0, 1172.0, 1173.0, 1162.0, 967.0, 1183.0, 978.0, 1372.0, 971.0
Win Rate:       99/100 (0.99)
Record:         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, W
in, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win
, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win,
Win, Win, Win, Win, Win, Win
*** PASS: test_cases\part2-4\grade-agent.test (8 of 8 points)
*** EXTRA CREDIT: 1 points
***     Grading agent using command:  python pacman.py -p ApproximateQAgent -x 2000 -n 2100 -l smallClassic -q -f --fixRandomSeed -a extractor=SimpleExtractor
***     1245.32 average score (4 of 4 points)
Question part2-4: 9/10
------------------
Total: 34/35
```

Part 3 : DQN (10%)

```
Average Score: 1546.6
Scores:        1756.0, 1159.0, 1558.0, 1711.0, 1549.0
Win Rate:      5/5 (1.00)
Record:        Win, Win, Win, Win, Win
```

```python
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F

""" Deep Q Network """


class DQN(nn.Module):
    def __init__(self, num_inputs=6, num_actions=4):
        super(DQN, self).__init__()

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

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

```

```python
# model parameters
model_trained = True

GAMMA = 0.8  # discount factor
LR = 0.02     # learning rate

batch_size = 32            # memory replay batch size
memory_size = 100000           # memory replay size
start_training = 5000      # start training at this episode
TARGET_REPLACE_ITER = 100  # update network step

epsilon_final = 0.1   # epsilon final
epsilon_step = 10000
```

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

Answer: An off-policy learner learns the value of the optimal policy independently of the agent's actions. Q-learning is an off-policy learner. An on-policy learner learns the value of the policy being carried out by the agent including the exploration steps.

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

Answer: In value-based methods, the agent learns to estimate the value of different actions or states in the environment. In policy-based methods, the agent learns directly the policy that maps states to actions. Actor-Critic methods combine the benefits of both value-based and policy-based methods. The agent has two components: an actor that learns the policy, and a critic that estimates the value function. $V^{\pi}(S)$ is the expected cumulative reward that an agent can achieve starting from state S and following policy π.

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

Answer: The main difference between MC and TD methods is the way they estimate the value function. MC methods rely on the empirical average of returns, which are the cumulative rewards obtained from a particular state or action until the end of the episode. On the other hand, TD methods estimate the value function by bootstrapping, which means updating the estimate using a new estimate. TD methods update the value function incrementally after each time step, by combining the reward obtained in that step with the estimated value of the next state. To be specific, MC methods require the agent to complete an entire episode before updating the value function, while TD methods can update the value function at each time step.

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

Answer: The state-action value function $Q^{\pi}(s,a)$ is the expected cumulative reward that an agent can achieve starting from state s, taking action a, and following policy π. The relationship between $Q^{\pi}(s,a)$ and $V^{\pi}(S)$ in Q-learning:

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

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

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

Answer: The target network is a copy of the Q-network, and its parameters are frozen for a certain number of iterations. This technique reduces the correlation between the target value and the Q-value and leads to a more stable and efficient learning process. Exploration allows the agent to discover new and potentially better policies. One common exploration strategy used in Q-learning is the epsilon-greedy policy. The epsilon parameter determines the trade-off between exploration and exploitation, and it is gradually reduced over time as the agent learns. Replay buffer is a memory buffer that stores the agent's experience in the form of (state, action, reward, next state) tuples. During learning, the agent samples a batch of experiences from the replay buffer and uses them to update the Q-function.

• Explain what is different between DQN and Q-learning.

Answer: The difference between DQN and Q-learning include their function approximation, experience replay, and the target network. In Q-learning, the Q-values are updated based on the maximum Q-value of the next state after every step based on the current experience. On the other hand, in DQN, it introduces the concept of target network and experience replay, which help to make the learning process more stable, and it uses a neural network to approximate the Q-function, which allows DQN to handle high-dimensional state spaces and continuous action spaces.

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

Answer:

```
PS D:\courses\Intro-to-AI\HW3\Adversarial_search> python pacman.py -p MinimaxAgent -l trappedClassic -a depth=3 -q -n 10
Pacman died! Score: -501
Pacman died! Score: -501
Pacman died! Score: -501
Pacman died! Score: -501
Pacman died! Score: -501
Pacman died! Score: -501
Pacman died! Score: -501
Pacman died! Score: -501
Pacman died! Score: -501
Pacman died! Score: -501
Average Score: -501.0
Scores:        -501.0, -501.0, -501.0, -501.0, -501.0, -501.0, -501.0, -501.0, -501.0, -501.0
Win Rate:      0/10 (0.00)
Record:        Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss
PS D:\courses\Intro-to-AI\HW3\Adversarial_search>
```

```
PS D:\courses\Intro-to-AI\HW3\Adversarial_search> python pacman.py -p ExpectimaxAgent -l trappedClassic -a depth=3 -q -n 10
Pacman emerges victorious! Score: 532
Pacman emerges victorious! Score: 532
Pacman emerges victorious! Score: 532
Pacman died! Score: -502
Pacman emerges victorious! Score: 532
Pacman died! Score: -502
Pacman died! Score: -502
Pacman died! Score: -502
Pacman emerges victorious! Score: 532
Pacman died! Score: -502
Average Score: 15.0
Scores:        532.0, 532.0, 532.0, -502.0, 532.0, -502.0, -502.0, -502.0, 532.0, -502.0
Win Rate:      5/10 (0.50)
Record:        Win, Win, Win, Loss, Win, Loss, Loss, Loss, Win, Loss
PS D:\courses\Intro-to-AI\HW3\Adversarial_search>
```

For part 1, by minimax search, Pacman rushes to the closest ghost since in its prediction (worst case), it'll all eventually lose the game. However, it's not the case that the ghosts will choose the worst actions towards Pacman. Therefore, by expectimax search, Pacman will try to grab food and sometimes it wins.

```
--------------------------------------------
Pacman died! Score: -404
Pacman died! Score: -383
Pacman died! Score: -374
Pacman died! Score: -228
Pacman died! Score: -420
Pacman died! Score: -375
Pacman died! Score: -400
Pacman died! Score: -413
Pacman died! Score: -428
Pacman died! Score: -412
Average Score: -383.7
Scores:        -404.0, -383.0, -374.0, -228.0, -420.0, -375.0, -400.0, -413.0, -428.0, -412.0
Win Rate:      0/10 (0.00)
Record:        Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss
```

(python pacman.py -p ApproximateQAgent -x 2000 -n 2010 -l smallGrid)

```
--------------------------------------------
Pacman emerges victorious! Score: 527
Pacman emerges victorious! Score: 525
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
Pacman emerges victorious! Score: 525
Pacman emerges victorious! Score: 527
Pacman emerges victorious! Score: 527
Average Score: 527.2
Scores:        527.0, 525.0, 529.0, 529.0, 529.0, 527.0, 527.0, 525.0, 527.0, 527.0
Win Rate:      10/10 (1.00)
Record:        Win, Win, Win, Win, Win, Win, Win, Win, Win, Win
```

(python pacman.py -p ApproximateQAgent -a extractor=SimpleExtractor -x 50 -n 60 -l mediumGrid)

```
PS D:\courses\Intro-to-AI\HW3\Q-learning> python pacman.py -p ApproximateQAgent -a extractor=SimpleExtractor -x 20 -n 25 -l smallClassic
Beginning 20 episodes of Training
Training Done (turning off epsilon and alpha)
--------------------------------------------
Pacman emerges victorious! Score: 955
Pacman died! Score: -127
Pacman died! Score: -291
Pacman emerges victorious! Score: 970
Pacman emerges victorious! Score: 972
Average Score: 495.8
Scores:        955.0, -127.0, -291.0, 970.0, 972.0
Win Rate:      3/5 (0.60)
Record:        Win, Loss, Loss, Win, Win
```

(python pacman.py -p ApproximateQAgent -a extractor=SimpleExtractor -x 20 -n 25 -l smallClassic)

```
Average Score: 1484.8
Scores:        1372.0, 1698.0, 1347.0, 1349.0, 1658.0
Win Rate:      5/5 (1.00)
Record:        Win, Win, Win, Win, Win
```

(python pacman.py -p PacmanDQN -n 25 -x 20 -l smallClassic)

For part 2 and part 3, we can see that the default PacmanQAgent performs poorly, while the performance ApproximateQAgent with SimpleExtractor is good. If we put ApproximateQAgent with SimpleExtractor and PacmanDQN in the same situation (same layout and same training times), we can see that the performance of PacmanDQN is even better.

- Describe problems you meet and how you solve them.

Answer: 執行 Part3 時遭遇警告訊息： 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 storages to the CPU. 解決方法如連結。此外，寫作業時在實作與公式上有不懂的地方，我主要參考了這些網站:

MDP 模型之 Grid World(值迭代方法)_UncoDong 的博客-CSDN 博客

MDP 模型之 Grid World(Q Learining 方法)_gridworld 价值迭代_UncoDong 的博客-CSDN 博客

什么是 Q Learning (Reinforcement Learning 强化学习) - YouTube

李宏毅_DRL Lecture 3: Q-learning (Basic Idea) - HackMD