hw3_109700046

Code Implementations

Part 1

My final version split the functions into two parts. one is Pacman, and the other is the ghosts for better workflow and structure.

Minimax Search

- Pacman (max layer)

```
def pacmanMove(state, depth):
    if state.isWin() or state.isLose() or depth == self.depth:
        return evalFn(state)
```

We first determine if the state is terminal, or the depth is reached so we have to return evaluation function value.

```
actions    = state.getLegalActions(0)
nextStates = list(map(
    lambda action:state.getNextState(0, action),
    actions)
)
```

Then we get the available actions for Pacman at the state, and the mapping next states (consequences).

```
scores  = list(map(
    ghostMove,
    nextStates,
    [depth]*len(actions),
    [1]*len(actions))
)
bestIdx = max(enumerate(scores), key = lambda x:x[1])[0]
```

For each next state, we call ghostMove() for the ghosts to choose the worst scores for us, and we'll choose the best one among the worst scores.

```
return actions[bestIdx] if depth == 0 else scores[bestIdx]
```

if we're in depth 0 than we'll return the action or we'll return the value evaluation which is scores[bestIdx].

- ghosts (mini layer)

```
def ghostMove(state, depth, ghost):
    if state.isWin() or state.isLose():
        return evalFn(state)
    if ghost == 0:
        return pacmanMove(state, depth+1)
```

same as Pacman that determine the terminal state, but we don't need to care about depth since it will be terminated by Pacman in all cases.

Because we have recursive calls for the ghosts, if all ghosts have done evaluation than we should move on to the next layer which is depth + 1 and make Pacman start the work.

```
nextAgent = (ghost + 1) % state.getNumAgents()
```

record the next agent would be called recursively later.

```
actions = state.getLegalActions(ghost)

nextStates = list(map(
    lambda action:state.getNextState(ghost, action),
    actions
))
```

Like what we do in PacmanMove(), we record the available actions for the ghost at the state and the mapping next states.

```
    scores  = list(map(
        ghostMove,
            nextStates,
            [depth]*len(actions),
            [nextAgent]*len(actions))
    )

    return min(scores)

return pacmanMove(gameState, 0)
```

Here's the most important part, we'll recursively call the ghosts one by one to evaluate the former ghost's scores, the ghost will select the worst score that evaluated by the nextAgent on the next state resulted from the actions.

```
return pacmanMove(gameState, 0)
```

Finally, we call pacmanMove() in getAction() with gameState derived from tparameters starts at depth = 0, and the whole functions will run recursively,

Expectimax Search

> Actually, there is only one line that differs from the minimax search for this part.

- Pacman (max layer)

```python
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
    evalFn = self.evaluationFunction

    def pacmanMove(state, depth):
        if state.isWin() or state.isLose() or depth == self.depth:
            return evalFn(state)

        actions    = state.getLegalActions(0)
        nextStates = list(map(lambda action:state.getNextState(0, action), actions))

        scores  = list(map(ghostMove, nextStates, [depth]*len(actions), [1]*len(actions)))
        bestIdx = max(enumerate(scores), key = lambda x:x[1])[0]

        return actions[bestIdx] if depth == 0 else scores[bestIdx]
```

Expectimax has totally the same max layer structure as Minimax, which is Pacman selection, so we can infer that the code of this part is also totally the same as Minimax's.

- ghost (expected layer)

```python
def ghostMove(state, depth, ghost):
    if state.isWin() or state.isLose():
        return evalFn(state)
    if ghost == 0:
        return pacmanMove(state, depth+1)

    nextAgent = (ghost + 1) % state.getNumAgents()
    actions = state.getLegalActions(ghost)

    nextStates = list(map(lambda action:state.getNextState(ghost, action), actions))

    scores  = list(map(
        ghostMove,
            nextStates,
            [depth]*len(actions),
            [nextAgent]*len(actions))
    )

    return sum(scores)/len(scores)
```

For the ghost part, they are also the same except one line which is the last one that evaluates the score.

```
return sum(scores)/len(scores)
```

Logically, minimax consider the worst outcome at the opponent layer, and expectimax consider the average outcome at the opponent layer, so we can just instinctively return the mean among the scores of a ghost, then we'll get the correct outcome, other parts in the structure are also called recursively.

- getAction(self, gameState)

```
return pacmanMove(gameState, 0)
```

Finally, we call pacmanMove() in getAction() with gameState derived from the parameters starts at depth = 0, and the whole functions will run recursively,

Screenshots:

```
class MinimaxAgent(MultiAgentSearchAgent):
    def getAction(self, gameState):

        def pacmanMove(state, depth):
            if state.isWin() or state.isLose() or depth == self.depth:
                return evalFn(state)

            actions    = state.getLegalActions(0)
            nextStates = list(map(lambda action:state.getNextState(0, action), actions))

            scores  = list(map(ghostMove, nextStates, [depth]*len(actions), [1]*len(actions)))
            bestIdx = max(enumerate(scores), key = lambda x:x[1])[0]

            return actions[bestIdx] if depth == 0 else scores[bestIdx]

        def ghostMove(state, depth, ghost):
            if state.isWin() or state.isLose():
                return evalFn(state)
            if ghost == 0:
                return pacmanMove(state, depth+1)

            nextAgent = (ghost + 1) % state.getNumAgents()
            actions = state.getLegalActions(ghost)

            nextStates = list(map(lambda action:state.getNextState(ghost, action), actions))

            scores  = list(map(
                ghostMove,
                    nextStates,
                    [depth]*len(actions),
                    [nextAgent]*len(actions))
            )

            return min(scores)

        return pacmanMove(gameState, 0)
```

minimax

```python
class ExpectimaxAgent(MultiAgentSearchAgent):
    def getAction(self, gameState):
        def pacmanMove(state, depth):
            if state.isWin() or state.isLose() or depth == self.depth:
                return evalFn(state)

            actions    = state.getLegalActions(0)
            nextStates = list(map(lambda action:state.getNextState(0, action), actions))

            scores  = list(map(ghostMove, nextStates, [depth]*len(actions), [1]*len(actions)))
            bestIdx = max(enumerate(scores), key = lambda x:x[1])[0]

            return actions[bestIdx] if depth == 0 else scores[bestIdx]

        def ghostMove(state, depth, ghost):
            if state.isWin() or state.isLose():
                return evalFn(state)
            if ghost == 0:
                return pacmanMove(state, depth+1)

            nextAgent = (ghost + 1) % state.getNumAgents()
            actions = state.getLegalActions(ghost)

            nextStates = list(map(lambda action:state.getNextState(ghost, action), actions))

            scores  = list(map(
                ghostMove,
                    nextStates,
                    [depth]*len(actions),
                    [nextAgent]*len(actions))
            )

            return sum(scores)/len(scores)

        return pacmanMove(gameState, 0)
```

expectimax

Part 2

Value Iteration

- **def** runValueIteration(**self**)

```
runValueIteration(self):
# Write value iteration code here
"*** YOUR CODE HERE ***"
# Begin your code
iterations = self.iterations
mdp = self.mdp
states = mdp.getStates()

for i in range(iterations):
    temp = util.Counter()
    for state in states:
        action = self.getAction(state)
        if action:
            temp[state] = self.computeQValueFromValues(state, action)

    self.values = temp
    # print(temp)
```

1. First, we run through the iterations in the first for loop.
    a. Then we create a temp Counter() for value update in this iteration, since we want to acquire QValues from former iteration but not the new updated ones.
    b. we then loop through all states for QValue update.
        i. we select the action for the state base on the algorithm in the getAction(),
        ii. if there is an available action, we compute the new QValue for (state, action) and update the temp Counter()
    c. At the end of the loop, we update the values Counter with temp Counter for next iteration.

- **def** computeQValueFromValues(**self, state, action**)

```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
    mdp = self.mdp
    discount = self.discount
    values = self.values

    qVal = 0
    probabilities = mdp.getTransitionStatesAndProbs(state, action)

    for nextState, p in probabilities:
        qVal += p * (values[nextState] * discount + mdp.getReward(state, action, nextState))
    # self.values[state] = qVal

    return qVal
    # End your code
```

1. First, we get the next states possibility distribution for the action by getTransitionStatesAndProbs()
2. Second, we loop through all the possible next states, calculate $\sum_{s' \in S} P_a(s'|s)[\gamma V(s') + r(s, a, s')]$ which is added up and stored in qVal.

- **def** computeActionFromValues(**self, state**)

```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
    mdp = self.mdp
    actions = mdp.getPossibleActions(state)
    if mdp.isTerminal(state):
        return None

    values = self.values
    v = []
    for action in actions:
        qVal = self.computeQValueFromValues(state, action)
        v.append(qVal)

    bestAction = actions[max(enumerate(v),key = lambda x:x[1])[0]]
    return bestAction

    # End your code
```

1. First, we want to make sure if the state is terminal which we'll return None.
2. Second, we create list v to store the estimated QValues mapped by the possible actions.
3. Third, we loop through the actions to calculate $\sum_{s' \in S} P_a(s'|s)[\gamma V(s') + r(s,a,s')]$ for each action and stores the values in v.
4. Last, we derive $max_{a \in A(s)} \sum_{s' \in S} P_a(s'|s)[\gamma V(s') + r(s,a,s')]$ by selecting the best action which maps the highest value in v among the actions, and we return the action.

Q-learning

- __init__

```
class QLearningAgent(ReinforcementAgent):
    def __init__(self, **args):
        "*** YOUR CODE HERE ***"
        # Begin your code
        self.qTable = Counter()
        # End your code
```

Create a qTable to store the qValues.

- getQValue

```
class QLearningAgent(ReinforcementAgent):
    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.qTable[(state, action)]
        # End your code
```

Because qTable is usually 2D, and we only got Counter, which is a 1D container, I use tuple (state, action) as the key for the Counter to access the qTalbe.

- computeValueFromQValues

```python
class QLearningAgent(ReinforcementAgent):
    def computeValueFromQValues(self, state):
        """
          Returns max_action Q(state,action)
          where the max is over legal actions.  Note that if
          there are no legal actions, which is the case at the
          terminal state, you should return a value of 0.0.
        """
        "*** YOUR CODE HERE ***"
        # Begin your code
        actions = self.getLegalActions(state)
        if len(actions) == 0:
            return 0.0
        return max([self.getQValue(state,action) for action in actions])

        # End your code
```

First, we get the possible actions for the state, we'll return 0.0 if there is no possible action (terminal states). Then we check our qTable, returning max(qTable[(state, action)]) for action in possible actions.

- computeActionFromQValues

```python
class QLearningAgent(ReinforcementAgent):
    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)
        if len(actions) == 0:
            return None
        return max(actions, key = lambda action:self.getQValue(state,action))
        # End your code
```

we'll return the legal action resulting the highest QValue in qTable[(state, action)] by max (actions, **key =** lambda action: self.getQValue(state,action)). By the way, we'll return None if there is no possible action (terminal states).

- getAction

```python
class QLearningAgent(ReinforcementAgent):
    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 None

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

Pretty close to the former function, but we consider exploration. there's a possibility that we'll just select a random action among legal actions, otherwise we'll return the possible action resulting the highest QValue in qTable[(state, action)].

- update

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

Here we are just directly applying the formula below:

$$Q'(S_t, A_t) \leftarrow (1 - lr)Q(S_t, A_t) + lr * (R_{t+1} + \gamma \max_a Q(S_{t+1}, a))$$

- (2-4) getQValue

```
class ApproximateQAgent(PacmanQAgent):
    def getQValue(self, state, action):
          where * is the dotProduct operator
        """
        "*** YOUR CODE HERE ***"
        # Begin your code
        # get weights and feature
        features = self.featExtractor.getFeatures(state, action)

        dot = 0
        for key in features:
            dot += self.weights[key] * features[key]
        return dot
        # End your code
```

Since the QValue here is the dot product of weight vector and feature vector, so we are just applying the dot product.

$$V(s) = w_1 f_1(s) + w_2 f_2(s) + ... + w_n f_n(s)$$

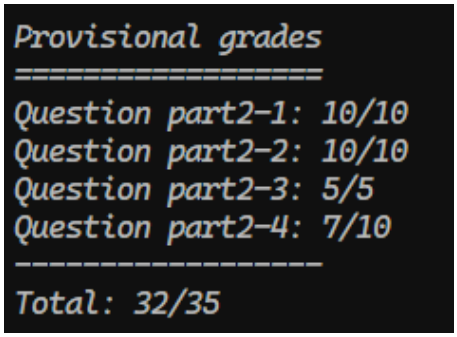
- (2-4) update

```
class ApproximateQAgent(PacmanQAgent):
    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)
        if self.weights.totalCount() == 0:
            for key in features:
                if util.flipCoin(0.9):
                    self.weights[key] = util.random.random()
                    if util.flipCoin(0.5):
                        self.weights[key] *= -1

        correction = (reward + self.discount * self.computeValueFromQValues(nextState)) - self.getQValue(state, action)
        for key in features:
            self.weights[key] = self.weights[key] + self.alpha * correction * features[key]

        # End your code
```

1. First, we randomize the start point of the weighs by possibly initial them to value between -1 and 1 and keep some of them 0.

2. Then we compute the correction which is $[r + \gamma max Q(s', a')] - Q(s, a)$
3. for each value in weight vector, we update the value by $w_i \leftarrow w_i + \alpha[correction] f_i(s, a)$
    - This process is a pretty like gradient descent.

Screenshots:



result for part 2

Part 3

```
model_trained = True

GAMMA = 0.94  # discount factor
LR = 0.08     # learning rate

batch_size = 32         # memory replay batch size
memory_size = 50000         # memory replay size
start_training = 300        # start training at this episode
TARGET_REPLACE_ITER = 100  # update network step

epsilon_final = 0.11    # epsilon final
epsilon_step = 7500
```

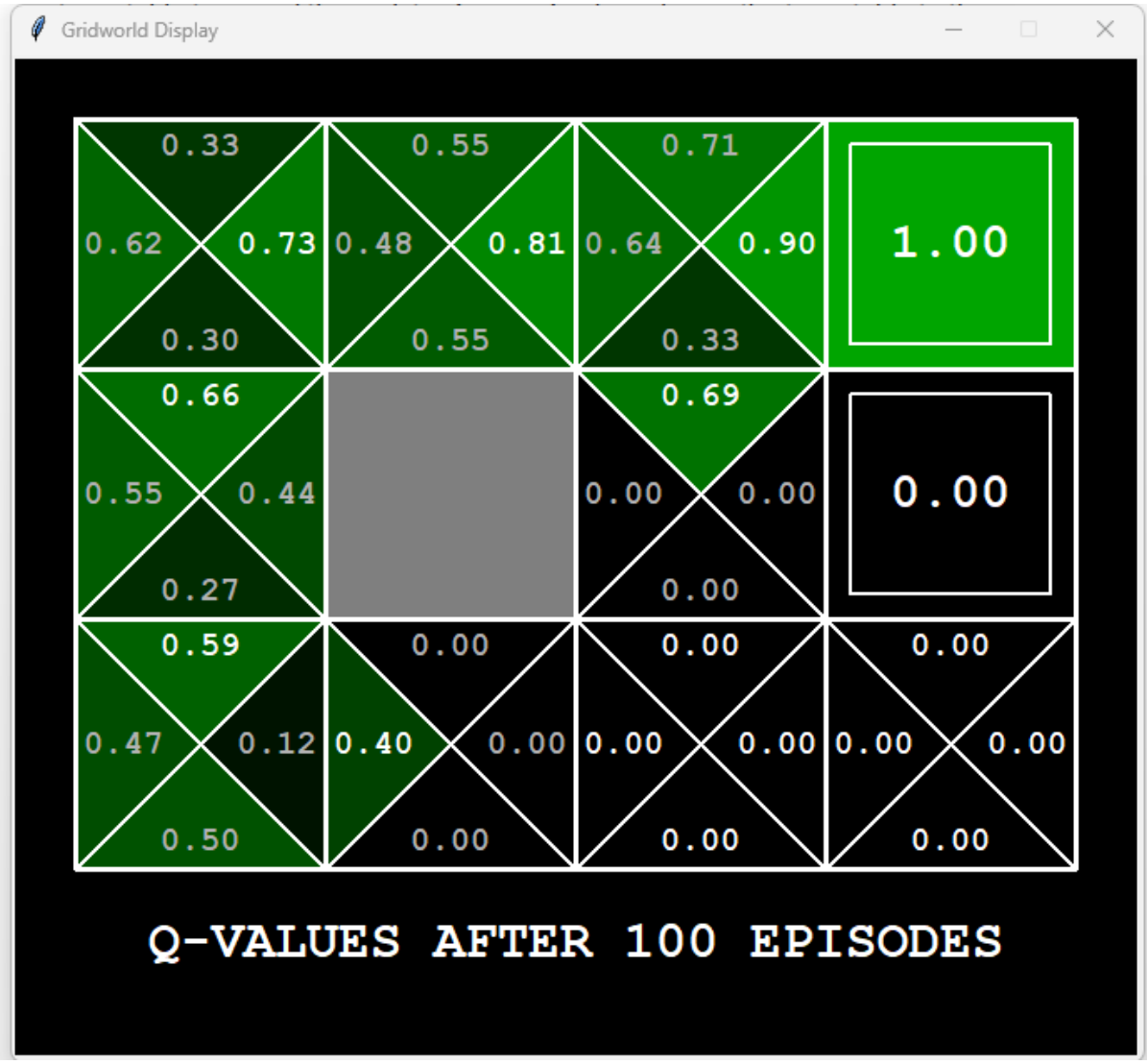
I only tuned some hyper parameters; other code remains the same.

```
PS D:\TEMP\uuu\.data\home\JJmow\Labs\aItro\HW3\HW3\DQN> python pacman.py -p PacmanDQN -n 150 -x 50 -l smallClassic -q
```

I train 50 epochs and remain 100 rounds for testing. result is showed in the comparison below.

Questions

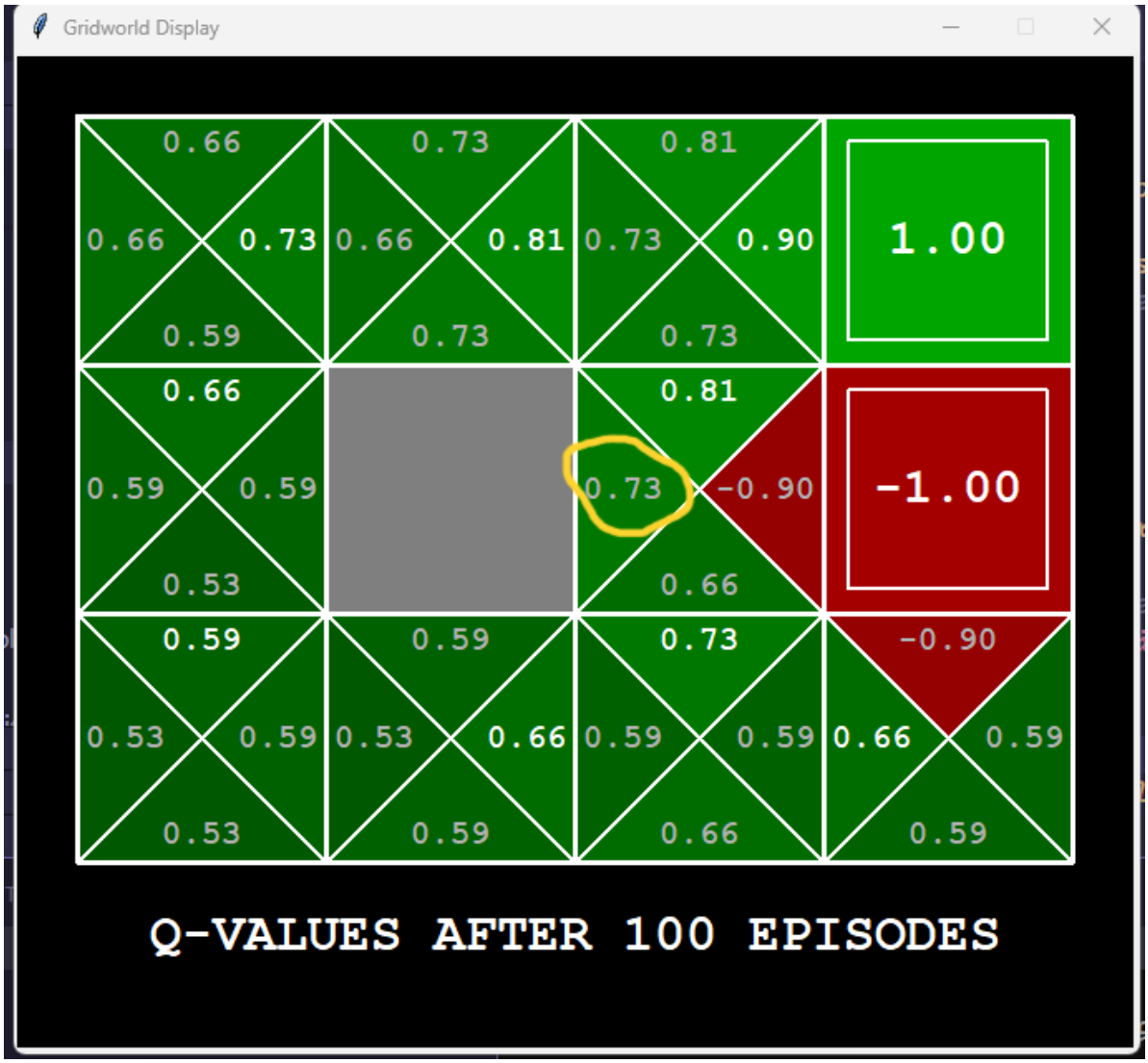
- (2-3) You can also observe the following simulations for different epsilon values. Does that behavior of the agent match what you expect?



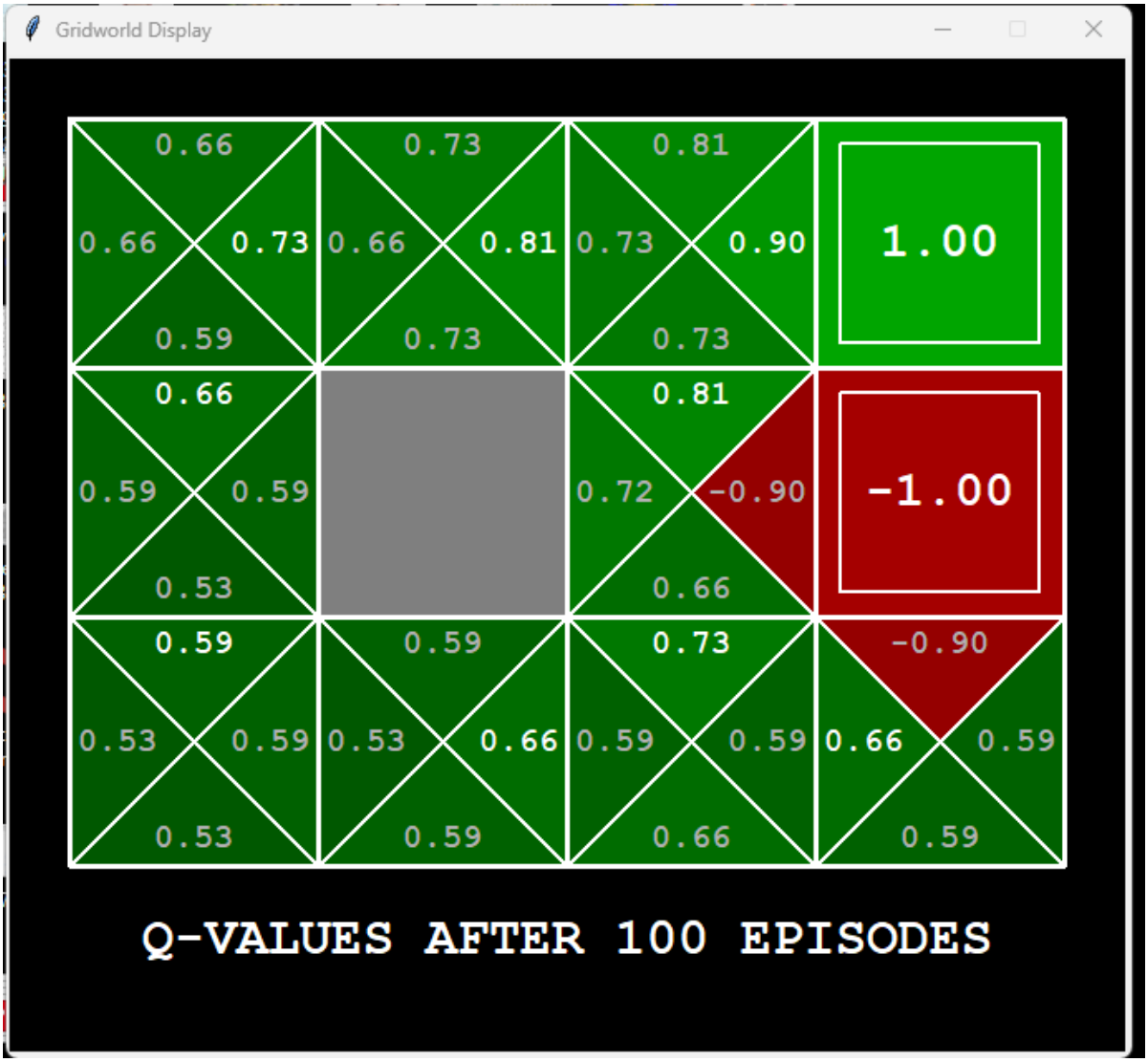
epsilon = 0.1


Gridworld Display
0.66
0.66
0.73
0.59
0.73
0.66
0.81
0.73
0.73
0.81
0.73
0.90
0.73
1.00
0.66
0.59
0.59
0.53
0.81
0.54
-0.23
0.00
-0.75
0.59
0.53
0.48
0.53
0.39
0.53
0.33
0.45
0.36
0.43
0.00
0.00
0.00
0.00
0.00
0.00
Q-VALUES AFTER 100 EPISODES

epsilon = 0.5


Gridworld Display
0.66
0.66 0.73
0.59
0.73
0.66 0.81
0.73
0.81
0.73 0.90
0.73
1.00
0.66
0.59 0.59
0.53
0.81
0.73 -0.90
0.66
-1.00
0.59
0.53 0.59
0.53
0.59
0.53 0.66
0.59
0.73
0.59 0.59
0.66
-0.90
0.66 0.59
0.59
Q-VALUES AFTER 100 EPISODES

epsilon = 0.9



epsilon = 1

- The result is pretty close to my expectations.
- when epsilon is small (≤0.5), we are exploring too much, making us very hard to find better QValues for updates.
- epsilon 0.9 and epsilon 1 have almost the same result. However, there is one QValue of epsilon 0.9 (0.73) beat epsilon 1 (0.72), which means it successfully find a hidden QValue which can't be found by fully greedy methods, that is very encouraging and prove the effectiveness of exploration.

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

    - On-policy learns from the policy we choose which is usually the best one in the environment and use the policy to take actions. this easier to implement but may not be able to find global maximum since this method is in lack of exploration.
    - Off-policy not only learns from the current policy it is using, but it may also try other policies at a state and be able to explore more in the environment, which is considered more flexible, but it is also more complex since it considers more than one policy at a state.
    - Summing up, On-policy learns from the current policy during training, but Off-policy doesn't always do that. On-policy is easier than Off-policy for implementation but get lower data efficiency.
- Briefly explain value-based, policy-based and Actor-Critic. Also, describe the value function $V^{\pi}(S)$.
    - value-based methods want to learn value function during training so it can determine policies based on the estimated values on a given state, it chooses the action that will lead to the highest expected reward.
    - policy-based methods learn policy function directly without estimating values, which maps states to actions that maximizes the expected future reward, it is usually represented by a neural network that trained to output the best action.
    - Actor-Critic combines the benefits of the two methods, the "Actor" is the policy function that selects actions, and the "Critic" is the value function that evaluates the value of those actions. The Critic's value estimates are used to update the Actor's policy, and they are trained together so that the Actor can learn to select actions that lead to higher rewards.
    - value function $V^{\pi}(S)$ is a function that we want to learn which estimate the expected total future reward on state S following policy π, it tells the agent how good a general state is.
- What is the difference between Monte-Carlo (MC) based approach and Temporal-difference (TD) approach for estimating $V^{\pi}(S)$.
    - The main difference between MC and TD is how they utilize their experience to update $V^{\pi}(S)$

    - Monte-Carlo based approach updates $V^{\pi}(S)$ after the end of an episode, which means it
      rely on the
      complete episode to estimate $V^{\pi}(S)$, which is already display in my code, while Temporal-difference approach updates $V^{\pi}(S)$ based on **one-step** experience transitions, updating the value function after each step, but not wait until the whole episode is finished.
    - Monte-Carlo based approach is usually considered more unbiased with higher variance since they rely solely on the outcome of an episode, including more random factors, while Temporal-difference approach is usually considered more biased with lower variance due to the smoother step-by-step update and the reliance on the current $V^{\pi}(S)$ itself.
- Describe State-action value function $Q^{\pi}(s, a)$ and the relationship between $V^{\pi}(s)$ in Q-learning.
    - $Q^{\pi}(s, a)$ is the expected future (long-term) reward for an agent taking action a at a given state s and then following policy π thereafter, essentially representing how good a state-action pair is.

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

    - by the above equation we can clearly notice that $V^{\pi}(s)$ is just the maximum Q-value achievable by taking any possible action a at given state s, representing the best Q-value achievable from the state.
- Describe following tips Target Network, Exploration and Replay Buffer using in Q-learning.
    - Target Network
        - problem → normal network's weights update is based on predicted QValues and expected future reward which leads to instability since we are chasing a moving target, computing target and predicted QValues with the same network.
        - Target Network use two networks to address this problem
            - The first network is the main network, the primary network the agent interacts with the environment and learns from, which is updated

regularly during training based on the difference between the reward, the discounted target QValue, and the current QValue.

- The second network is the target network, which is actually the copy of the main network but updated less frequently, remaining fixed for a certain period, and the agent use the weight in the target network to compute the target QValues. This network provides a more stable target for updating the QValues of the main network, leading to smoother learning.
- This technique helps to improve the stability and the convergence of the learning process and reduces the impact of immediate rewards on long-term expected value estimation.

- Exploration
  - The agent needs to explore the environment to discover more actions and their consequence, or the agent may miss some nice (state, action) pairs that leads to higher expected future reward and not able to get the optimum policies without Exploration.
  - Some common strategies are:
    - epsilon-greedy → strategy we used in this homework. With a small probability epsilon, the action will choose a random action among the possible actions to explore the environment. And in the rest of the time (1-epsilon), the agent selects the action with the highest estimated QValue (exploitation).
    - Boltzmann Exploration → assigning a probability to every action base on its estimated QValue and a temperature parameter, higher temperatures encourage exploration, and vice versa.
- Replay Buffer
  - problem → The agent learns from the experience it collects during training, however, storing and learning from all experiences can be:
    - computationally expensive
    - may lead to correlated or redundant data, slowing down the learning process.
  - Replay Buffer is a storage mechanism used to improve the efficiency, during training, the agent's experiences (state, action, reward, next state)

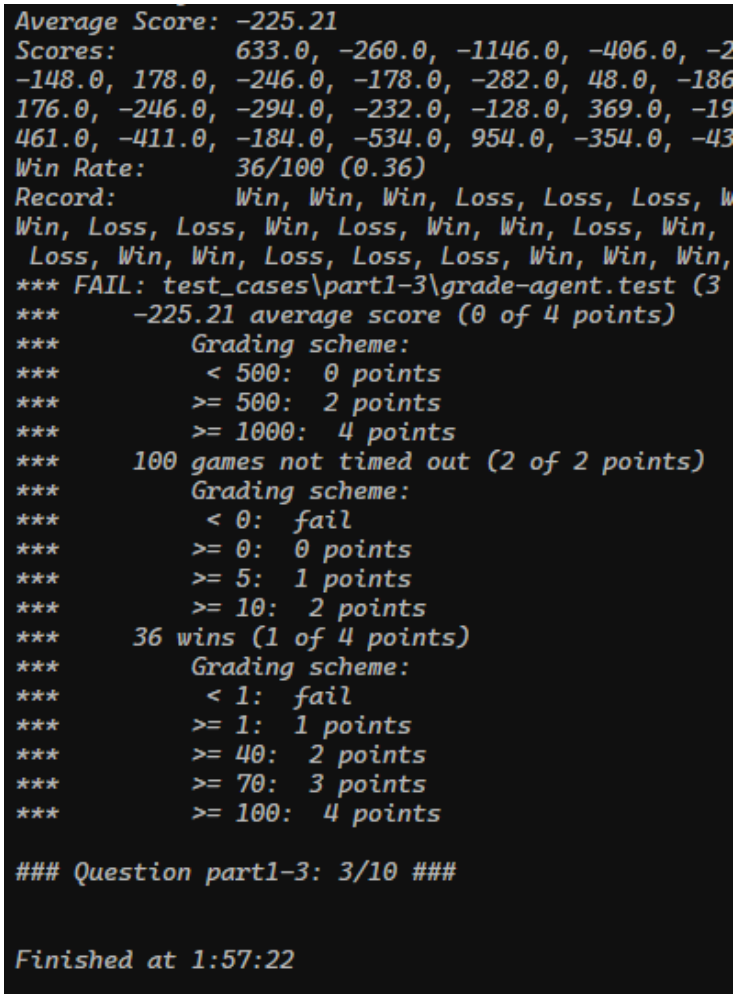
are stored in a finite buffer, a mini batch of experiences is sampled randomly from the replay buffer, and the network is updated based on these samples, unlike the traditional way that based on the most recent experience.

        - The benefits are:
            - Increased Efficiency → we reuse past experiences, so the agent can learn from a larger dataset.
            - Reduced Correlation → since we sample randomly from the replay buffer, decorrelating the training data.
            - Breaking Temporal Dependence → we are not just learning from the most recent experience, but from a variety of past experiences, reducing the impact of temporal dependence and smoothing out the learning process.

- Explain what is different between DQN and Q-learning.
    - In brief, DQN replace QTable in Q-learning with deep neural network.
    - QTable may face capacity problem when the environment is very complicated, while neural network in DQN (ex. MLP) can conquer that, handling high-dimensional state spaces more effectively.
    - The input of the network in DQN is the current state, and the output are the QValues of all the possible actions.
    - DQN introduces (mentioned above)
        - Replay Buffer → storing the agent's experience in a replay buffer and randomly sample from this buffer during training.
        - Target Network → uses the separate network to compute QValues for the Bellman equation, updated periodically.
        - but normal Q-learning doesn't.
    - Under the assumption of exploring all state-action pairs infinitely often, Q-learning can coverage to the optimal QValues, however, DQN does not have theoretical convergence guarantees due to the non-linear function approximation and the methods it introduces.

Discussions

- Compare the performance of every method and do some discussions in your report.
  - Map → smallClassic
  - results in the terminal



Minimax

```
1.0, 450.0
Win Rate:      61/100 (0.61)
Record:        Win, Win, Win, Loss, Win, Loss, Win, W
oss, Win, Loss, Loss, Win, Loss, Loss, Win, Win, Win,
Win, Win, Loss, Win, Win, Win, Win, Loss, Win, Loss,
*** FAIL: test_cases\part1-3\grade-agent.test (4 of 8
***     -109.73 average score (0 of 4 points)
***         Grading scheme:
***          < 500:  0 points
***         >= 500:  2 points
***         >= 1000:  4 points
***     100 games not timed out (2 of 2 points)
***         Grading scheme:
***          < 0:  fail
***         >= 0:  0 points
***         >= 5:  1 points
***         >= 10:  2 points
***     61 wins (2 of 4 points)
***         Grading scheme:
***          < 1:  fail
***         >= 1:  1 points
***         >= 40:  2 points
***         >= 70:  3 points
***         >= 100:  4 points

### Question part1-3: 4/10 ###
```

Expectimax

```
Average Score: 845.32
Scores:        968.0, -45.0, 975.0, 984.0, 972.0, 1161.0, 1152.0, 958.0, -186.0, 976.0, 960.0, 966.0, 981.0, -141.0, -318.0, 969.0, -63.0, 1154.0, 966.0, -329.0, 957.0, 96
.0, 977.0, 959.0, -152.0, 982.0, 966.0, 969.0, -56.0, 979.0, -397.0, 974.0, 961.0, 982.0, 952.0, 959.0, 981.0, 963.0, 978.0, 976.0, 980.0, 977.0, 972.0, 975.0, 970.0, 962.
, 960.0, 952.0, 980.0, 964.0, 951.0, 971.0, -311.0, 961.0, -64.0, 975.0, 963.0, 974.0, 976.0, 984.0, 970.0, 978.0, 1155.0, 975.0, 967.0, 972.0, 957.0, 978.0, 965.0, 986.0,
968.0, -85.0, 982.0, 1166.0, 976.0, 983.0, 981.0, 911.0, 1175.0, 966.0, 974.0, 982.0, 974.0, 1165.0, 969.0, 984.0, 983.0, 979.0, 959.0, 985.0, 974.0, 956.0, 945.0, 978.0,
60.0, 984.0, 965.0, 923.0, 974.0, 984.0
Win Rate:      88/100 (0.88)
Record:        Win, Loss, Win, Win, Win, Win, Win, Win, Loss, Win, Win, Win, Win, Loss, Loss, Win, Loss, Win, Win, Loss, Win, Win, Win, Win, Loss, Win, Win, Win, Loss, Win
 Loss, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Loss, Win, Loss, Win, Win, Win, Win, Win, Win, Win, Win, Wi
, Win, Win, Win, 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, Win, Win, Win, Win, Win, Win, Win
 Win, Win
*** FAIL: test_cases\part2-4\grade-agent.test (7 of 8 points)
***     Grading agent using command:  python pacman.py -p ApproximateQAgent -x 2000 -n 2100 -l smallClassic -q -f --fixRandomSeed -a extractor=SimpleExtractor
***     845.32 average score (2 of 4 points)
***         Grading scheme:
***          < 500:  0 points
***         >= 500:  2 points
***         >= 1000:  4 points
***     100 games not timed out (2 of 2 points)
***         Grading scheme:
```

Approximate Q-Learning

```
Episode no = 150; won: True; Q(s,a) = 232.62158269623762; reward = 772.0; and epsilon = 0.0
Average Score: 1298.42
Scores:        1732.0, -186.0, 101.0, -238.0, 1460.0, 1326.0, 930.0, 1117.0, 1166.0, 455.0, 1649.0, 13
.0, 288.0, 1141.0, 136.0, 474.0, 1365.0, 1685.0, 1344.0, 305.0, 531.0, 1693.0, 1511.0, 1119.0, 1196.0,
72.0, 918.0, 1101.0, 1042.0, 902.0, 1720.0, 1130.0, 1437.0, 1683.0, 1295.0, 1349.0, 1536.0, 1763.0, 15
.0, 1531.0, 1754.0, 1144.0, 1696.0, 1537.0, 1538.0, 1168.0, 1767.0, 1367.0, 1364.0, 642.0, 1521.0, 135
, 1358.0, 1739.0, 1161.0, 1649.0, 1568.0, 1756.0, 746.0, 1354.0, 1554.0, 1754.0, 1534.0, 1309.0, 1752.
 1551.0, -59.0, 1164.0, 1527.0, 1135.0, 1570.0, 1354.0, 1559.0, 1731.0, 1746.0
Win Rate:      87/100 (0.87)
Record:        Win, Loss, Loss, Loss, Win, Win, Win, Win, Win, Loss, Win, Win, Win, Win, Win, Win, Wi
ss, Loss, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Wi
 Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Loss, Win, Win, Win, Loss, Win, Win, Win, Win,
Win, Win, Win, Win, Win, Win, Win, Win, Win, Loss, Win, Win, Win, Win, Win, Win, Win, Win
PS D:\TEMP\uuu\.data\home\JJmow\Labs\aItro\HW3\HW3\DQN>
```

DQN

- result table

	MiniMax	Expectimax	Approximate Q-Learning	DQN
Win Rate (100 games)	36%	61%	88%	87%
Average score	-225.21	-109.73	845.32	1298.42

- It is obvious that:
  - Adversarial searchs have the worst performance of all.
    - MiniMax got the worst performance with 36%-win rate.
  - win rates of Approximate Q-Learning and are DQN are pretty close.
  - DQN got the best Average score of all, beating second place by around 450 points.
- Adversarial searchs has to consider all possible outcomes, and the space of possible outcomes for complex game like Pacman is extremely large, making it computationally expensive and may lead to sub-optimal decisions. And design of evaluation function is also crucial which isn't good enough in my opinion in my case, I think they are the main reason leading to bad scores.
- MiniMax performed the worst; I think it is because the lack of Probabilistic Reasoning, its spirit is trying to minimize the possible loss for a worst case, which is very pessimistic, while expectimax is more trying to optimize play for the expected score. On the other hand, MiniMax is more suitable for deterministic games, while there is some randomness among the ghosts' actions (?) which makes the environment stochastic and not that suitable for MiniMax search.
- Approximate Q-Learning use a method pretty close to gradient descent and the structure is pretty like a neural network, the performances of Approximate Q-Learning and DQN strongly indicate the effectiveness and efficiency of applying neural networks, however, the average score of Approximate Q-Learning isn't as good as DQN, which I considered the reason to be not applying mechanisms like Target Network and Replay Buffer used in DQN, and the network structure isn't as robust as DQN.
- DQN got the highest average score, which means that it can learn to navigate the game more efficiently with its neural networks, and the power of nn and

those mechanisms also make DQN outstanding in both win rate and average scores.

- Describe problems you meet and how you solve them. 5/5
  - I spent at least 10 hours debugging part1, which made me almost gone insane. At first, my max layer and the other layer are merged in one function tree, the structure is quite fuzzy.

```
def tree(depth,gameState):
    if gameState.isWin() or gameState.isLose():
        return 'Stop'

    myActions = gameState.getLegalActions(0)

    scores = []
    for action in myActions:
        aPossibleNextState = gameState.getNextState(0, action)
        numStateAgents = gameState.getNumAgents()

        if aPossibleNextState.isWin():return action
        if aPossibleNextState.isLose():
            scores.append(-1e9)
            continue

        score = 0
        for enemy in range(1,numStateAgents):

            consequences = list(map(
                lambda action:aPossibleNextState.getNextState(enemy,action),
                aPossibleNextState.getLegalActions(enemy)
            ))
            if depth == self.depth:
                score += sum(list(map(evalFn,consequences))) / len(consequences)
            else:
                scores = []
                for consequence in consequences:
                    if consequence.isWin() or consequence.isLose():
                        scores.append(evalFn(consequence))
                        continue
                    actionWillDo = tree(depth+1, consequence)
                    scores.append(evalFn(consequence.getNextState(0,actionWillDo)))
                score += sum(scores) / len(scores)

        scores.append(score/(numStateAgents-1))
```

    - I tried to run pacman.py and the result is between 0.5~0.7 (win rate), which I thought was good enough to pass the autograder.py. However, always got 0 point in both 1-1 and 1-2 due to the incorrectness of tree structure and some wrong decisions. After stuck for a long period, I started to discover some articles and open-source code, than I realize the problem in my code.
    - The max layer is no problem, and I spited it out to be a pacmanMove function. In my former implementation, I consider all ghosts' moves at the same time, which means the ghosts are not considering each other's consequences but only Pacman's, leading to incorrectness of the tree structure and some wrong decisions. The correct way is to calculate the ghosts' scores one by one (ghost); one ghost's scores (list) are determined by another ghost's moves, and another ghost's scores are determined by

the other ghost’s moves... and so on, both feasible for expectimax or minimax. And the way to select these scores / moves is based on the algorithm we are using (expectimax ,minimax, etc....)

- After applying this ghostMove structure and made it a ghostMove function, calling pacmanMove() which will recursively call ghostMove made me get full points in 1-1 and 1-2.

- doing part 2 and part 3 are quite smoothly for me, the only mistake I made is in runValueIteration() in value iteration implementation. We should update the table only at the end of an episode, but not at the end on a single state update which I did do it previously, it will lead to one-step experience transitions which happens in TD approach, but we are implementing MC approach. So, I created a temp table to record the updates in an episode and copy the temp table to the real table at the end of the episode.