AI_HW3_repo_111550076

notion link:

AI_HW3_repo_111550076

Part 1 : Adversarial search

Part 1-1: Minimax Search.

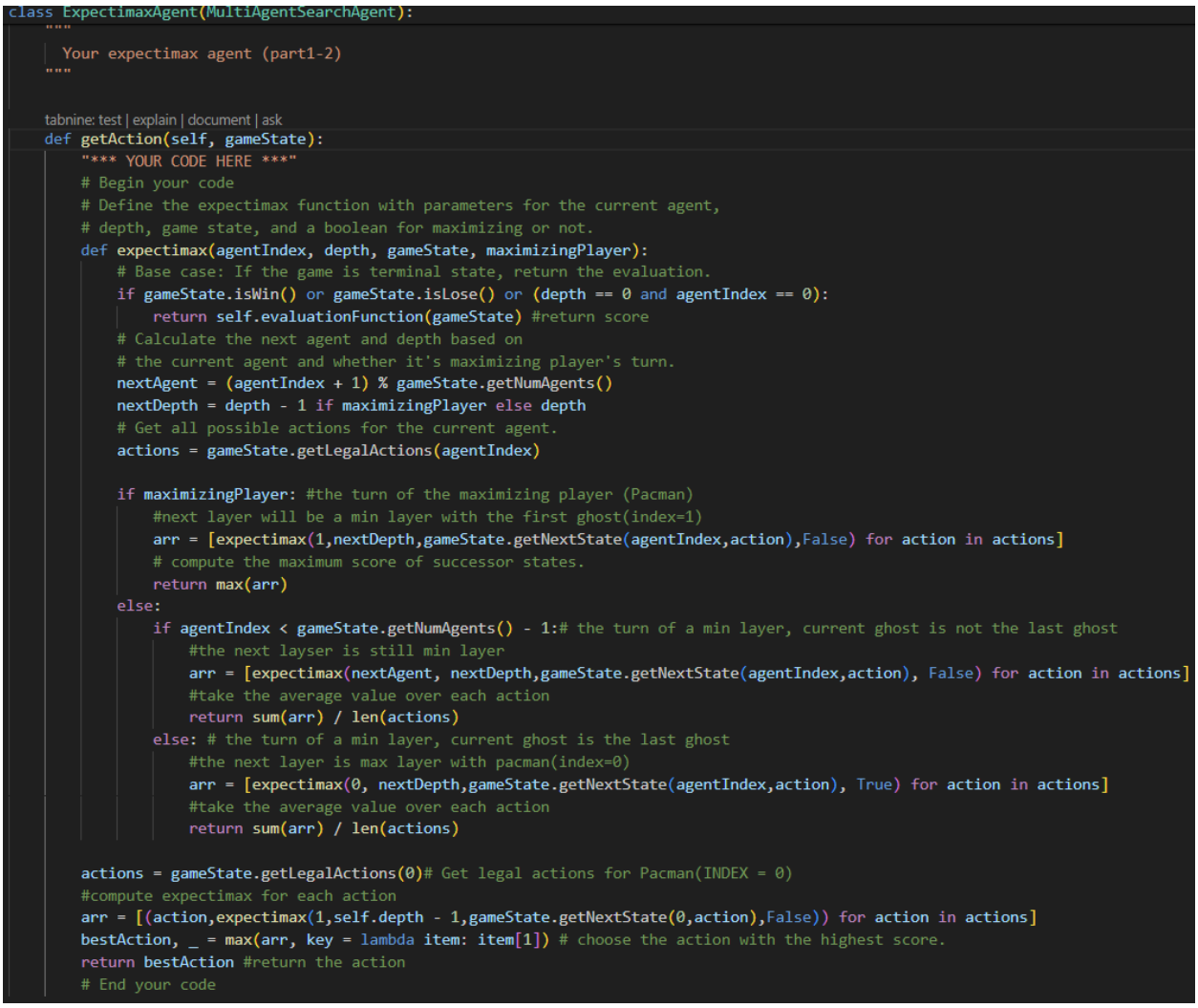
The code implements the Minimax algorithm for a game agent, recursively determining the best move by evaluating potential future game states. It alternates between maximizing and minimizing layers, representing the agent and its opponent, until it reaches a terminal state or the maximum search depth.

```python
class MinimaxAgent(MultiAgentSearchAgent):
    """
    Your minimax agent (par1-1)
    """
    tabnine: test | explain | document | ask
    def getAction(self, gameState):
        "*** YOUR CODE HERE ***"
        # Begin your code
        # Define the minimax function with parameters for the current agent,
        # depth, game state, and a boolean for maximizing or not.
        def minimax(agentIndex, depth, gameState, maximizingPlayer):
            # Base case: return the game state and its evaluation if the game is terminal state
            if gameState.isWin() or gameState.isLose() or (depth == 0 and agentIndex == 0):
                return (gameState,self.evaluationFunction(gameState))#return (state,score) pair
            # Calculate the next agent and depth based on
            # the current agent and whether it's maximizing player's turn.
            nextAgent = (agentIndex + 1) % gameState.getNumAgents()
            nextDepth = depth - 1 if maximizingPlayer else depth
            # Get all possible actions for the current agent.
            actions = gameState.getLegalActions(agentIndex)

            if maximizingPlayer: #the turn of the maximizing player (Pacman)
                #next layer will be a min layer with the first ghost(index=1)
                arr = [minimax(1,nextDepth,gameState.getNextState(agentIndex,action),False) for action in actions]
                # compute the maximum score of successor states.
                return max(arr, key = lambda item: item[1])
            else:
                if agentIndex < gameState.getNumAgents() - 1:# the turn of a min layer, current ghost is not the last ghost
                    #the next layser is still min layer
                    arr = [minimax(nextAgent, nextDepth,gameState.getNextState(agentIndex,action), False) for action in actions]
                    # compute the minimum score for non-final ghosts
                    return min(arr, key = lambda item : item[1])
                else:# the turn of a min layer, current ghost is the last ghost
                    #the next layer is max layer with pacman(index=0)
                    arr = [minimax(0, nextDepth,gameState.getNextState(agentIndex,action), True) for action in actions]
                    # compute the minimum score for final ghosts
                    return min(arr, key = lambda item : item[1])

        actions = gameState.getLegalActions(0) # Get legal actions for Pacman(INDEX = 0)
        #compute minimax for each action
        tup = [(action,minimax(1,self.depth - 1,gameState.getNextState(0,action),False)) for action in actions]
        bestAction, _ = max(tup, key = lambda item: item[1][1]) # choose the action with the highest score.
        return bestAction #return the action
        # End your code
```

```
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%)

This code segment outlines the Expectimax algorithm for an agent in a game, like Pac-Man. The Expectimax algorithm evaluates potential game states but, unlike Minimax, it considers that the opponent does not necessarily make optimal decisions. When the agent is maximizing, it chooses the action with the highest value. For other agents, it calculates the average of the values, simulating a chance node where each action has equal probability. The algorithm recurses until it reaches a win, loss, or the maximum depth. The best action for the agent is chosen by evaluating all possible actions from the root node.



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

    tabnine: test | explain | document | ask
    def getAction(self, gameState):
        "*** YOUR CODE HERE ***"
        # Begin your code
        # Define the expectimax function with parameters for the current agent,
        # depth, game state, and a boolean for maximizing or not.
        def expectimax(agentIndex, depth, gameState, maximizingPlayer):
            # Base case: If the game is terminal state, return the evaluation.
            if gameState.isWin() or gameState.isLose() or (depth == 0 and agentIndex == 0):
                return self.evaluationFunction(gameState) #return score
            # Calculate the next agent and depth based on
            # the current agent and whether it's maximizing player's turn.
            nextAgent = (agentIndex + 1) % gameState.getNumAgents()
            nextDepth = depth - 1 if maximizingPlayer else depth
            # Get all possible actions for the current agent.
            actions = gameState.getLegalActions(agentIndex)

            if maximizingPlayer: #the turn of the maximizing player (Pacman)
                #next layer will be a min layer with the first ghost(index=1)
                arr = [expectimax(1,nextDepth,gameState.getNextState(agentIndex,action),False) for action in actions]
                # compute the maximum score of successor states.
                return max(arr)
            else:
                if agentIndex < gameState.getNumAgents() - 1:# the turn of a min layer, current ghost is not the last ghost
                    #the next layser is still min layer
                    arr = [expectimax(nextAgent, nextDepth,gameState.getNextState(agentIndex,action), False) for action in actions]
                    #take the average value over each action
                    return sum(arr) / len(actions)
                else: # the turn of a min layer, current ghost is the last ghost
                    #the next layer is max layer with pacman(index=0)
                    arr = [expectimax(0, nextDepth,gameState.getNextState(agentIndex,action), True) for action in actions]
                    #take the average value over each action
                    return sum(arr) / len(actions)

        actions = gameState.getLegalActions(0)# Get legal actions for Pacman(INDEX = 0)
        #compute expectimax for each action
        arr = [(action,expectimax(1,self.depth - 1,gameState.getNextState(0,action),False)) for action in actions]
        bestAction, _ = max(arr, key = lambda item: item[1]) # choose the action with the highest score.
        return bestAction #return the action
        # End your code
```

```
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 : Q-learning

Part 2-1: Value Iteration (10%)

- The `runValueIteration` method updates state values by calculating expected returns for each possible action, ensuring terminal states are set to zero. It iteratively refines state values using the previous iteration's data.

```python
def runValueIteration(self):
    # Write value iteration code here
    "*** YOUR CODE HERE ***"
    # Begin your code
    # Iterate over each iteration.
    for i in range(self.iterations):
        # Keep a copy of the current values to use for updates.
        previous_value = self.values.copy()
        # Get all the states in the MDP.
        states = self.mdp.getStates()
        # Iterate over each state to update its value.
        for state in states:
            # If the state is terminal, its value is zero.
            if self.mdp.isTerminal(state):
                self.values[state] = 0
                continue
            # Initialize max_value to a very small number.
            max_value = float('-inf')
            # Get all possible actions from the current state.
            actions = self.mdp.getPossibleActions(state)
            # Iterate over each action to calculate its value.
            for action in actions:
                total = 0
                # Get the transition states and probabilities of the current action.
                transitions = self.mdp.getTransitionStatesAndProbs(state, action)
                # Sum up the value of all transitions from the current state-action pair.
                for (nextState, prob) in transitions :
                    total += prob * (self.mdp.getReward(state, action, nextState) + self.discount*previous_value[nextState])
                # Update max_value if the calculated total is greater.
                if total > max_value:
                    max_value = total
            # Update the value of the state with the maximum value found.
            self.values[state] = max_value
    # End your code
```

- The `computeQValueFromValues` function computes the Q-value for a state-action pair by aggregating the expected rewards of potential future states, adjusted for probability and discounted future values.

```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
    q_value = 0 # Initialize Q-value
    # Retrieve all transitions and their probabilities for the given state-action pair
    transitions = self.mdp.getTransitionStatesAndProbs(state, action)
     # Sum up the value for each transition
    for (nextState, prob) in transitions:
        # Each part contributes to the Q-value: probability * (immediate reward + discounted future value)
       q_value += prob * (self.mdp.getReward(state, action, nextState)  + self.discount*self.values[nextState])
    return q_value  # Return the calculated Q-value
    # End your code
```

- The `computeActionFromValues` function determines the best action from a state by calculating and comparing Q-values for all possible actions, returning the one with the highest value. If the state is terminal, it returns None.

```python
def computeActionFromValues(self, state):
    "*** YOUR CODE HERE ***"
    # Begin your code
    #check for terminal
    # If the state is terminal, there are no actions to take, return None.
    if self.mdp.isTerminal(state):
        return None
    q_value = util.Counter() # Initialize a Counter to store Q-values for each action.
    actions = self.mdp.getPossibleActions(state)  # Get all legal actions for the state.
    # Compute the Q-value for each action and store in the Counter.
    for action in actions:
        q_value[action] = self.computeQValueFromValues(state, action)
    # Return the action with the highest Q-value, breaking ties arbitrarily.
    return q_value.argMax()
    # End your code
```

```
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 (15%)

- **init**

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

    "*** YOUR CODE HERE ***"
    # Begin your code
    # Initialize Q-values as a util.Counter, which defaults all values to zero.
    # This is where all Q-values for state-action pairs will be stored.
    self.qValues = util.Counter()
    # End your code
```

- getQValue

```python
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 qValue base on state and action
    return self.qValues[(state, action)]
    # End your code
```

- The `computeValueFromQValues` function returns the highest Q-value for all legal actions from a given state, or 0.0 if no actions are available.

```python
def computeValueFromQValues(self, state):
    "*** YOUR CODE HERE ***"
    # Begin your code
    # Get all legal actions for the current state
    legalActions = self.getLegalActions(state)
    # If there are no legal actions, return 0.0
    if len(legalActions) == 0:
        return 0.0
    # Otherwise, return the maximum Q-value for the legal state
    return max([self.getQValue(state, action) for action in legalActions])
    # End your code
```

- The `computeActionFromQValues` function selects and returns a random action from those with the highest Q-value for the given state. If no legal actions exist, it returns `None`.

```python
def computeActionFromQValues(self, state):
    "*** YOUR CODE HERE ***"
    # Begin your code
    # Get all legal actions for the current state
    legalActions = self.getLegalActions(state)
    # If there are no legal actions, return None
    if not legalActions:
        return None
    # Otherwise, return the action with the maximum Q-value
    bestValue = self.computeValueFromQValues(state)
    bestActions = [action for action in legalActions if self.getQValue(state, action) == bestValue]
    return random.choice(bestActions)
    # End your code
```

```
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 ###
```

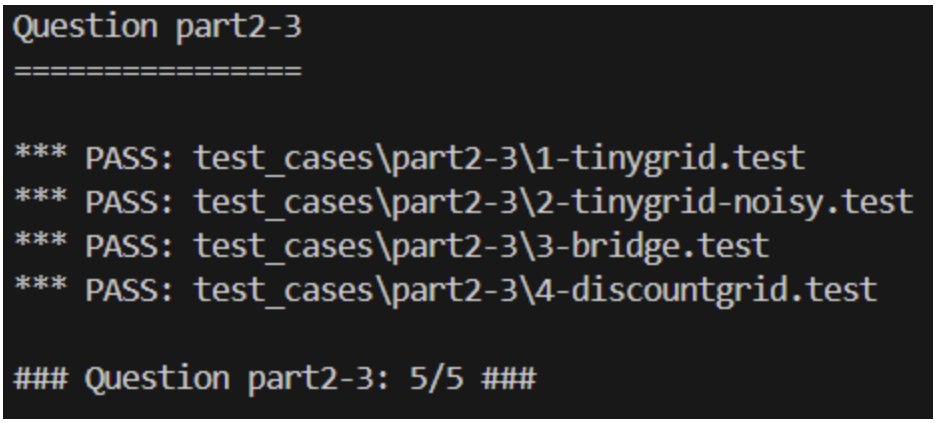
Part 2-3: epsilon-greedy action selection (10%)

- The `getAction` function decides the next action using an epsilon-greedy strategy. It returns `None` if no actions are available. Otherwise, it randomly selects an action with a probability of `epsilon` or chooses the best action based on Q-values.

```
def getAction(self, state):
    # Pick Action
    legalActions = self.getLegalActions(state)
    action = None
    "*** YOUR CODE HERE ***"
    # Begin your code
    # If there are no legal actions, return None
    if not legalActions:
      return None
    # With probability self.epsilon, take a random action
    if util.flipCoin(self.epsilon):
      return random.choice(legalActions)
    # Otherwise, take the best policy action
    else:
      return self.computeActionFromQValues(state)
    # End your code
```

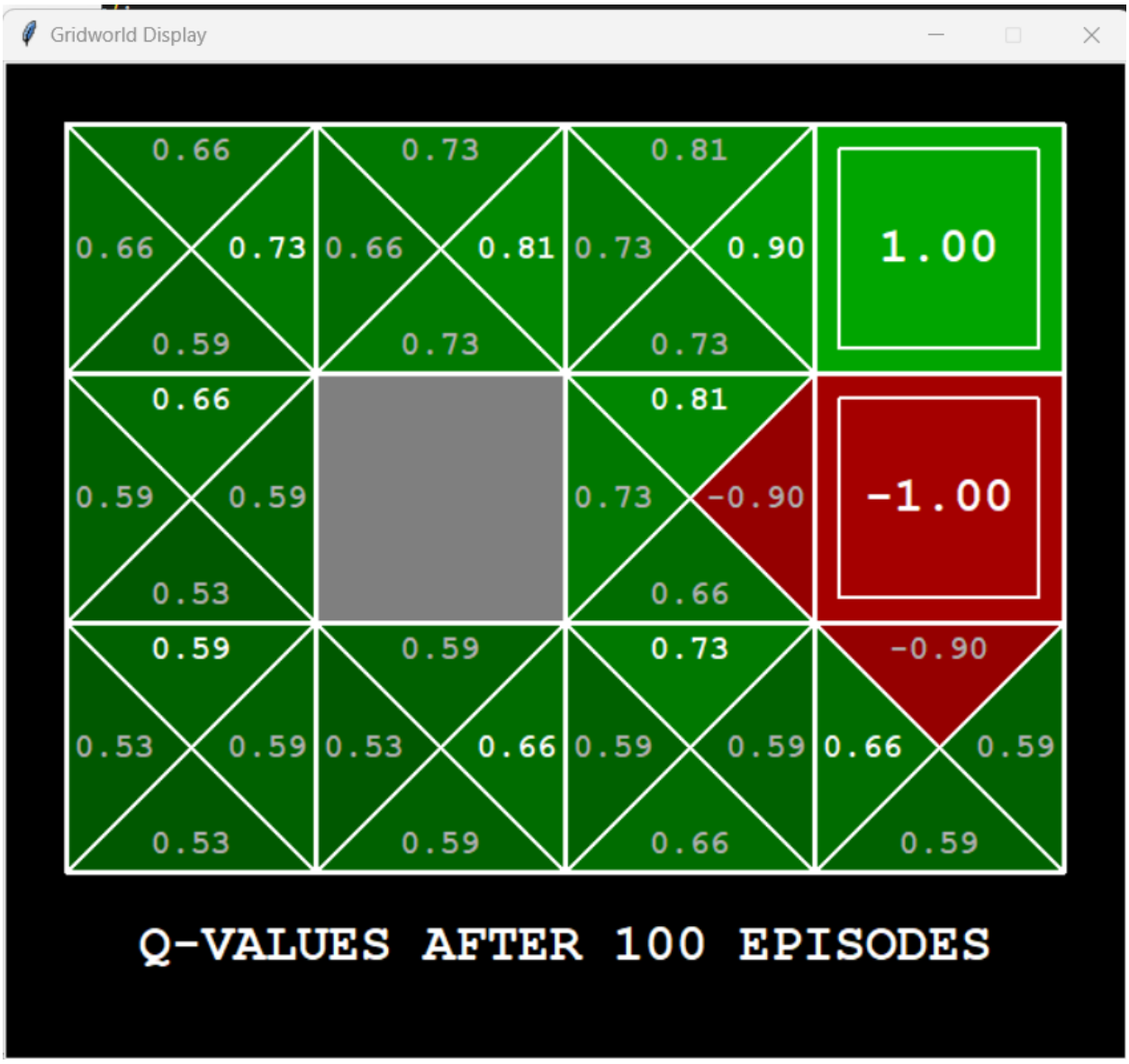
- The `update` function refines the Q-value for a state-action pair by blending the observed reward and the predicted future value using a learning rate, `alpha`, to improve decision accuracy over time.

```
def update(self, state, action, nextState, reward):
    "*** YOUR CODE HERE ***"
    # Begin your code
    # Compute the sample, which is the reward plus the discounted value of the next state
    sample = reward + self.discount * self.computeValueFromQValues(nextState)
    # Update the Q-value for the current state and action
    self.qValues[(state, action)] = (1 - self.alpha) * self.getQValue(state, action) + self.alpha * sample
    # End your code
```



- You can also observe the following simulations for different epsilon values. Does that behavior of the agent match what you expect? (discuss it in your report)

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



The outcome aligns with my expectations (⬆⬆➡➡➡) As epsilon increases, leading to a higher frequency of random actions, the average returns from the start state tend to diminish due to the decreased likelihood of consistently selecting the most rewarding actions.

Part 2-4: Approximate Q-learning (Bonus) (10%)

- The `getQValue` function calculates the Q-value for a given state-action pair by performing a dot product between the agent's weights and the feature vector extracted from the state and action.

```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 the dot product of the weights and the features
    return self.getWeights() * self.featExtractor.getFeatures(state, action)
    # End your code
```

- This `update` method adjusts the agent's weights based on the observed transition. It computes the correction using the reward and the discounted value of the next state, then updates each weight accordingly by multiplying the correction by the corresponding feature value and the learning rate alpha.

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

```python
def update(self, state, action, nextState, reward):
    "*** YOUR CODE HERE ***"
    # Begin your code
    # Get the features for the current state and action
    features = self.featExtractor.getFeatures(state, action)
    # Compute the correction, which is the reward plus the discounted value of the next state
    # minus the current Q-value for the state and action
    correction = reward + self.discount * self.computeValueFromQValues(nextState) - self.getQValue(state, action)
    # Update the weights for each feature
    for feature in features:
        self.weights[feature] += self.alpha * correction * features[feature]
    # End your code
```

- The `getFeatures` method in this code is responsible for extracting relevant features from the game state to represent the current state-action pair. It begins by extracting information about the layout of the game grid, including the locations of food, walls, and ghosts. Additionally, it identifies the presence of capsules and distinguishes between active and scared ghosts.

  The features extracted include:

- A bias feature, which is always set to 1.0 to represent the intercept term in linear regression.
- The number of active ghosts that are one step away from the agent's next position.
- An "escape" feature, indicating whether the agent should attempt to move away from nearby active ghosts.
- Features indicating whether the agent is about to eat a capsule or food.

Furthermore, the method calculates the distance to the closest food item and normalizes it based on the dimensions of the game grid. Finally, the feature values are divided by 10.0 to scale them down before returning them as a counter object.

```python
class SimpleExtractor(FeatureExtractor):

    def getFeatures(self, state, action):
        # extract the grid of food and wall locations and get the ghost locations
        food = state.getFood()
        walls = state.getWalls()
        ghosts = state.getGhostStates()
        features = util.Counter()
        ###  modify
        capsules = state.getCapsules() # get the capsules
        activePos = []
        scared = [] # get the scared ghost
        # get the scared ghost and the active ghost positions
        for i, g in enumerate(ghosts, start=1):
            if g.scaredTimer:
                scared.append(g)
                features["safe-time"] = g.scaredTimer #get the scared time
            else:
                # get the active ghost position
                activePos.append(state.getGhostPosition(i))
        ###
        features["bias"] = 1.0

        # 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 ###modify
        features["#-of-active-ghosts-1-step-away"] = sum(
            (next_x, next_y) in Actions.getLegalNeighbors(g, walls) for g in activePos)
```

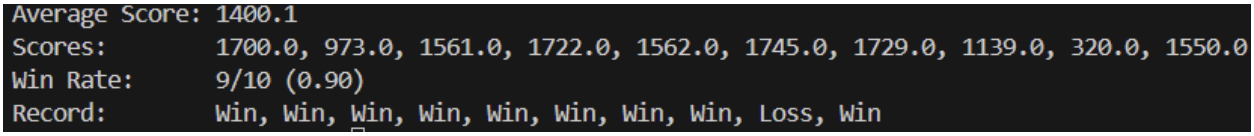
```python
# If there are active ghosts nearby, set the "escape" feature to 1.0
# indicating the agent should try to move away
if features["#-of-active-ghosts-1-step-away"]:
    features["escape"] = 1.0
else:
    # If no active ghosts are nearby, check for other conditions
    # If a capsule is present and there are no scared ghosts, set the "eats-capsule" feature to 1.0
    if (next_x, next_y) in capsules and not scared:
        features["eats-capsule"] = 1.0
    # If there is food at the next position, set the "eats-food" feature to 1.0
    elif food[next_x][next_y]:
        features["eats-food"] = 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
```

```
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, W
in, Loss, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, 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, Wi
n, 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 smallClassi
c -q -f --fixRandomSeed -a extractor=SimpleExtractor
***     1245.32 average score (4 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
***     99 wins (3 of 4 points)
***         Grading scheme:
***          < 1:  fail
***         >= 1:  1 points
***         >= 40:  2 points
***         >= 70:  3 points
***         >= 100:  4 points

### Question part2-4: 9/10 ###
```

Part 3 : DQN (10%)

Deep Q-learning uses a deep neural network to determine the optimal action given a game state. In other words, DQN uses a neural network to replace the traditional Q-table. The update process for the Q-value then becomes the back-propagation through the neural network.

I will use the provided model to compare with other method in this homework.



The result of provided model

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

On-policy and Off-policy learning are differing mainly in how they handle the data used for training:

On-policy Learning

On-policy methods learn exclusively from the actions taken by the current policy being evaluated. This means the policy used for learning is the same as the policy used for making decisions. Examples include SARSA and Actor-Critic methods.

Off-policy Learning

Off-policy methods learn from actions taken by a different policy from the one currently being evaluated. This allows for the use of data collected from past policies or different strategies. Examples include Q-learning and Deep Q-Networks (DQNs).

Key Differences:

1. **Data Usage**: On-policy uses data from the current policy only, while Off-policy can use data from any policy.

2. **Flexibility**: Off-policy is more flexible, allowing learning from a broader range of experiences.
3. **Risk**: On-policy requires direct exploration, which can be riskier; Off-policy can learn from safe, historical data.

These approaches cater to different scenarios based on the safety and efficiency needs of the learning environment.

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 optimizing the value function, which measures how good it is to be in a certain state. The policy is derived from this value function by selecting actions that maximize value.

Policy-Based Methods

These directly optimize the policy, which dictates the agent's actions. Unlike value-based methods, they can directly learn stochastic policies and are well-suited for continuous or high-dimensional action spaces.

Actor-Critic Methods

Actor-Critic methods combine value-based and policy-based approaches. They use a "critic" to evaluate actions and an "actor" to update the policy based on the critic's feedback, aiming to stabilize and improve the learning process.

Value Function $V^{\pi}(S)$

The value function $V^{\pi}(S)$ represents the expected return when starting from state S and following policy π. It's crucial for assessing the long-term benefit of states under a given policy.

The value function is defined as:

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

where R_{t+k+1} is the reward received after k steps, γ is the discount factor (which balances immediate vs. future rewards), and the expectation is over the stochasticity of the policy and the environment.

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

Monte Carlo (MC) Methods:

- **Data Requirement**: MC methods require complete episodes; they learn from the full return following each state until the end of the episode.
- **Update Mechanism**: Updates to the value estimates are made at the end of each episode based on the actual returns received.
- **Key Feature**: MC methods do not bootstrap (i.e., they do not update estimates based on other learned estimates).

Temporal Difference (TD) Methods:

- **Data Requirement**: TD methods can learn online, updating estimates based on incomplete sequences, one step at a time.
- **Update Mechanism**: Updates to the value estimates are made after each step using the observed reward and the estimated value of the subsequent state (bootstrap).
- **Key Feature**: TD methods bootstrap, which allows them to update estimates more frequently and learn before knowing the final outcome.

In summary, the main difference lies in how and when the value estimates are updated: MC methods wait until the end of an episode and use actual returns, while TD methods continuously update using estimated returns.

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, denoted as $Q^{\pi}(s, a)$, represents the expected return for selecting action aa in state s and thereafter following policy π. It captures the value of taking a specific action from a specific state and then continuing with a given policy. The function is defined as:

$$Q^{\pi}(s, a) = \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s, A_t = a\right]$$

This contrasts with the state value function V^{\pi}(S), which is the expected return starting from state s and following policy π for all subsequent decisions. It is defined as:

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

Relationship in Q-Learning:

In Q-learning, which is a model-free off-policy learning algorithm, the relationship between these functions is crucial:

- **From Q to V:** The state value $V^{\pi}(S)$is derived from the state-action value $Q^{\pi}(s, a)$ by taking the maximum over all possible actions from state *s*:

  $V^{\pi}(S) = max_a Q^{\pi}(s, a)$
  This implies that the best action's value dictates the value of being in a state under the optimal policy, which is central to Q-learning's approach of learning the optimal
  *Q*values to guide optimal decision-making.

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

1. Target Network

- **Purpose**: Stabilizes learning by providing consistent target values for training updates.
- **How it Works**: It's a copy of the main neural network model, but its weights are updated less frequently to avoid the moving target problem.

2. Exploration

- **Purpose**: Ensures the agent learns a robust policy by exploring enough of the state space.
- **How it Works**: Commonly implemented using an ε-greedy strategy where the agent chooses random actions with decreasing probability over time, balancing exploration with exploitation.

3. Replay Buffer

- **Purpose**: Enhances learning efficiency and stability by breaking correlations between consecutive training samples.
- **How it Works**: It stores experiences and allows the agent to learn from a random sample of past transitions, reusing this data multiple times.

These techniques are critical for managing the inherent challenges of applying Q-learning to complex environments, ensuring that the training process remains stable and effective.

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

1. **Representation**: Q-learning uses a tabular approach to store Q-values for each state-action pair, limiting its scalability to small state spaces. DQN, on the other hand, employs deep neural networks to approximate Q-values, enabling it to handle large and continuous state spaces.
2. **Function Approximation**: While Q-learning directly updates Q-values based on observed transitions, DQN learns to approximate the Q-function using a neural network. This allows DQN to generalize across similar states and handle high-dimensional input spaces.

3. **Stability Mechanisms**: DQN incorporates stability mechanisms like Experience Replay and Target Networks. Experience Replay stores past experiences in a replay buffer and samples mini-batches from it for training, reducing correlations between consecutive updates. Target Networks are used to stabilize training by maintaining a separate target network with fixed parameters.

In summary, DQN extends Q-learning to handle complex environments with large state spaces by using deep neural networks and introducing stability mechanisms, making it more scalable and effective in practice.

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

Different method comparison (smallClassic)

Method	game amount	Win Rate	Average Score
Minimax (depth=3)	10	20%	113.2
Expectimax (depth=3)	10	60%	682.5
Q-learning (2000 episodes)	10	0%	-407.6
Approximate Q-learning (2000 episodes)	100	94%	1174.51
DQN	100	91%	1400.1

As we can see, Minimax exhibits poor performance on this larger map due to its inability to handle the increased complexity. To enhance its effectiveness, we may consider improving the evaluation function to bolster its strategic decision-making capabilities.

Expectimax, on the other hand, performs relatively better, boasting around a fifty percent win rate. This improvement is attributed to its nature of considering all possible outcomes weighted by their probabilities. However, it still requires a refined evaluation function to avoid stagnation.

Moving on to Q-learning, it is observed that even after 2000 episodes, it fails to secure a single victory. It may be because Q-learning only updates Q-values using the score of the current game state. In contrast, approximate Q-learning

employs handcrafted features to update the Q-values, providing the agent with directional cues for updating its Q-values.

Lastly, DQN employs a neural network to determine the optimal action given a state. While theoretically expected to outperform traditional methods, its performance falls slightly behind approximate Q-learning. This suggests that further parameter tuning or increased training iterations may be required to optimize its performance.

```
PS C:\zichen\AI\HW3\Adversarial_search> python pacman.py -p MinimaxAge
nt -l smallClassic -a depth=3  -q  -n 10
Pacman emerges victorious! Score: 804
Pacman emerges victorious! Score: 1072
Pacman died! Score: -322
Pacman died! Score: 173
Pacman died! Score: 41
Pacman died! Score: -260
Pacman died! Score: 174
Pacman died! Score: -66
Pacman died! Score: 13
Pacman died! Score: -497
Average Score: 113.2
Scores:        804.0, 1072.0, -322.0, 173.0, 41.0, -260.0, 174.0, -66.
0, 13.0, -497.0
Win Rate:      2/10 (0.20)
Record:        Win, Win, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Los
s
```

Result of minimax agent

```
PS C:\zichen\AI\HW3\Adversarial_search> python pacman.py -p Expectimax
Agent -l smallClassic -a depth=3  -q  -n 10
Pacman emerges victorious! Score: 1151
Pacman emerges victorious! Score: 1119
Pacman emerges victorious! Score: 1101
Pacman died! Score: -31
Pacman died! Score: 300
Pacman died! Score: 50
Pacman died! Score: -237
Pacman emerges victorious! Score: 1337
Pacman emerges victorious! Score: 657
Pacman emerges victorious! Score: 1378
Average Score: 682.5
Scores:        1151.0, 1119.0, 1101.0, -31.0, 300.0, 50.0, -237.0, 133
7.0, 657.0, 1378.0
Win Rate:      6/10 (0.60)
Record:        Win, Win, Win, Loss, Loss, Loss, Loss, Win, Win, Win
```

Result of Expectimax agent

```
Average Score: -407.6
Scores:        -479.0, -382.0, -443.0, -403.0, -491.0, -399.0, -373.0, -342.0, -373.0, -391.0
Win Rate:      0/10 (0.00)
Record:        Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss
```

Result of q-learning agent

```
Reinforcement Learning Status:
        Completed 100 test episodes
        Average Rewards over testing: 1174.51
        Average Rewards for last 100 episodes: 1174.51
        Episode took 8.93 seconds
Average Score: 1174.51
Scores:        1563.0, 1171.0, 982.0, 1187.0, 1173.0, 985.0, 1373.0, 1176.0, 1770.0, 978.0, 1569.0, 1183.0, 1572.0,
1373.0, 1176.0, 1164.0, 1171.0, 983.0, 978.0, 1176.0, 1184.0, 1174.0, 1170.0, 1174.0, 1566.0, 1160.0, 1183.0, 1176.0
, 1157.0, 79.0, 1177.0, 1766.0, 1168.0, 1173.0, 1171.0, 1181.0, 1374.0, 68.0, -363.0, 1173.0, 1373.0, 1576.0, 1172.0
, 1378.0, 1378.0, 971.0, 1183.0, 1381.0, 1181.0, 1373.0, 1170.0, 1176.0, 1168.0, 966.0, 1176.0, 1160.0, 1179.0, 1365
.0, 1179.0, 1179.0, 145.0, 1372.0, 1158.0, 1173.0, 1375.0, -286.0, 976.0, 1381.0, 1546.0, 1171.0, 1372.0, 1362.0, 11
79.0, 1376.0, 1174.0, 344.0, 1378.0, 1179.0, 1168.0, 1375.0, 1176.0, 969.0, 1161.0, 1356.0, 1178.0, 1181.0, 1170.0,
1369.0, 1582.0, 982.0, 1183.0, 1181.0, 1166.0, 1579.0, 1336.0, 1180.0, 1158.0, 1770.0, 1172.0, 1386.0
Win Rate:      94/100 (0.94)
```

Result of approximate q-learning agent

```
Average Score: 1414.29
Scores:       1380.0, 1546.0, 1772.0, 1758.0, 1340.0, 1725.0, 1735.0, 1766.0, 1770.0, 1346.0, 99.0, 977.0, 1570.0, 1547.
0, 1747.0, 1345.0, 337.0, 1547.0, 1332.0, 1763.0, 1348.0, 1163.0, 1334.0, 1344.0, 1366.0, 1770.0, 1766.0, 1764.0, 1378.0,
 1722.0, 1558.0, 1570.0, 1746.0, 1766.0, 1344.0, 1762.0, 1363.0, 1167.0, 1551.0, 1678.0, 1366.0, 1742.0, 129.0, 277.0, 65
.0, 1741.0, 1748.0, 949.0, 332.0, 1565.0, 1757.0, 1561.0, 1757.0, 1760.0, 635.0, 1340.0, 1372.0, 1148.0, 1558.0, 1370.0,
1551.0, 1740.0, 1132.0, 1731.0, 1693.0, 1320.0, 1550.0, 1572.0, 1166.0, 1318.0, 1748.0, 1365.0, 1316.0, 1575.0, 1549.0, 5
00.0, 1159.0, 1750.0, 1532.0, 1351.0, 1307.0, 1526.0, 1564.0, 1131.0, 1562.0, 1747.0, 1763.0, 1703.0, 1163.0, 1743.0, 156
9.0, 535.0, 1567.0, 1752.0, 1557.0, 1564.0, 1157.0, 1556.0, 1759.0, 1552.0
Win Rate:      91/100 (0.91)
Record:       Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Loss, 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, Loss, Loss, Los
s, Win, Win, Win, Loss, Win, Win, Win, Win, Win, Loss, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, W
in, Win, Win, Win, Win, Win, Win, Loss, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Win, Loss,
Win, Win, Win, Win, Win, Win, Win, Win
```

Result of DQN agent

Describe problems you meet and how you solve them.

Unable to execute CUBA

Due to my computer lacking a CUBA processor and encountering errors in the code, I made modifications to the code accordingly.

```python
# init model
if(model_trained == True):
    #cuda
    # self.policy_net = torch.load('pacman_policy_net.pt').to(self.device)->
    self.policy_net = torch.load('pacman_policy_net.pt', map_location=torch.device('cpu'))

    #self.target_net = torch.load('pacman_target_net.pt').to(self.device)->
    self.target_net = torch.load('pacman_target_net.pt', map_location=torch.device('cpu'))
else:
    #self.policy_net = DQN().to(self.device)->
    self.policy_net = DQN().to(torch.device('cpu'))
    #self.target_net = DQN().to(self.device)->
    self.target_net = DQN().to(torch.device('cpu'))
```