Reinforcement Learning for Game 2048

Research Question:

We hypothesize that an AI agent, utilizing reinforcement learning methods like Q-learning and Temporal Difference (TD) Learning, can effectively play the game 2048 by selecting optimal moves to maximize the score, achieve tile values of 2048 or higher, and minimize decision time compared to other strategies like Expectimax and baseline approaches.

Introduction:

2048 is a popular single-player puzzle game where players combine numbered tiles to maximize their score and reach tiles with values like 2048 or higher. While its rules are simple, the game presents a complex challenge due to random tile generation, limited board space, and the need for strategic decision-making. Solving this problem efficiently is important because it involves optimizing gameplay strategies under uncertainty, which is a key challenge in artificial intelligence.

Developing AI agents to play 2048 using reinforcement learning techniques, such as Q-learning and Temporal Difference (TD) Learning, to make smart decisions in uncertain conditions can lead to more effective decision-making. By comparing these methods with Expectimax and baseline models, this research not only aims to improve game performance but also demonstrates how such techniques can be applied to real-world problems that require complex decision-making under constraints.

Related Literature:

1. On Reinforcement Learning for the Game of 2048
2. Hung Guei's On Reinforcement Learning for the Game of 2048
3. Multistage Temporal Difference Learning for 2048 like Games

These three research papers offered valuable insights for improving our 2048 AI project. On Reinforcement Learning for the Game of 2048 enhanced Q-Learning and TD-Learning with effective reward systems. Hung Guei's On Reinforcement Learning for the Game of 2048 introduced optimistic temporal difference learning, n-tuple networks, Monte Carlo tree search, and deep reinforcement learning, achieving state-of-the-art performance. Multistage TD (MS-TD) learning for 2048 like Games explored hierarchical reinforcement learning techniques to enhance large tile achievements and extend adaptability to similar games like Threes. Unlike these studies, our approach combines Q-learning, TD learning, and Expectimax, focusing on a detailed comparison using metrics such as win rate, decision time, and convergence, providing a broader analysis of gameplay strategies.

AI Strategies for Solving 2048:

1. **RandomAgent (Baseline Model):** *Baseline random Strategy.*
   (Describes the simplest model relying on random moves.)
2. **Q-Learning:** *Reinforcement Learning Optimization*
   (Highlights the learning-based approach for state-action value improvement.)
3. **TD-Learning:** *Temporal Difference Prediction*
   (Focuses on predicting rewards using dynamic updates.)
4. **Expectimax:** *Heuristic Search Algorithm*
   (Emphasizes decision-making through tree-based evaluation.)

Pseudocodes of Implemented AI Strategies :

C: > Users > pnred > OneDrive > Desktop > AI TERM PROJECT > pmr.txt

```
Initialize move_options = [0, 1, 2, 3]  # Directions for moves
Function choose_random_move():
    Return random choice from move_options
Function find_max_tile(board):
    Return the maximum tile from all cells in the board
Function run_simulation():
    For each game from 1 to 10000:
        Initialize board and add two tiles
        While game is not lost:
            move = choose_random_move()
            board, score = perform_move(board, move)
            Add new tile to the board
            If game state is 'lose':
                Record max tile for the game
    Print average score and standard deviation of scores
    For each tile value:
        Calculate frequency of reaching that tile
        Print the frequency
```

Random Agent

```
function EVALUATE(s, a):
    return V_a(s)

function LEARN_EVALUATION(s, a, r, s', s"):
    v_next <- max_a'(E(s"))
    V_a(s) <- V_a(s) + a(r + v_next - V_a(s))
```

Q - Learning

Reinforcement Learning for Game 2048

```
1: function PLAY_GAME
2:    score ← 0
3:    s ← INITIALIZE_GAME_STATE
4:    while ¬IS_TERMINAL_STATE(s) do
5:        a ← arg max_a'(E(s)) EVALUATE(s, a')
6:        r, s' ← MAKE_MOVE(s, a)
7:        if LEARNING_ENABLED then
8:            LEARN_EVALUATION(s, a, r, s', s")
9:        score ← score + r
10:       s ← s'
11:   return score

12: function MAKE_MOVE(s, a)
13:    r ← COMPUTE_AFTERSTATE(s, a)
14:    s' ← ADD_RANDOM_TILE(s')
15:    return (r, s')
```

TD-Learning

```
Define actions = [0, 1, 2, 3], max_depth = 2
Function get_move(state):
    Return maximize(state, 1, 3 * max_depth)
Function maximize(state, depth, max_depth):
    If lost(state) Return -∞
    If depth > max_depth Return evaluate(state)
    best_value = -∞
    For action in actions:
        If valid(state, action): best_value = max(best_value, expect(state, action, depth + 1))
    Return best_value
Function expect(state, action, depth):
    Return sum(evaluate_tile(state, action) for empty_cells) / len(empty_cells)
Function evaluate_tile(state, action):
    Return 0.9 * maximize(state, action) + 0.1 * maximize(state, action)
Function evaluate(state):
    Return (350 * empty_count) + (800 * merges) - (20 * monotonicity)
Function main():
    For i in range(5): state = new_game(4); While not lost(state): state = perform_move(state, get_move(state))
If __name__ == "__main__": main()
```

Expectimax

Game Environment(game.py):

We developed game.py to efficiently manage the grid and moves for the 2048 game. It starts with an empty grid (new_game function) and adds tiles 2 or 4 randomly (add_two function). Moves (up, down, left, right) shift and combine tiles, handled by cover_up and merge. Helpers like reverse and transpose assist with grid operations. The game checks for a win or loss using game_state, and perform_move updates the board, score, and game status. The game ends when no moves are left.

```
NEW GAME
[[0. 0. 0. 0.]
 [0. 0. 0. 4.]
 [0. 2. 0. 0.]
 [0. 0. 0. 0.]]

Enter move (0:up, 1:left, 2:right, 3:down): 1
[[0. 0. 0. 0.]
 [4. 0. 0. 2.]
 [2. 0. 0. 0.]
 [0. 0. 0. 0.]]

Enter move (0:up, 1:left, 2:right, 3:down):
```

```
Enter move (0:up, 1:left, 2:right, 3:down): 2
[[  2.   8.  32.   2.]
 [256.  16.   4.  16.]
 [  2.   8.  32.   8.]
 [  2.   4.  16.   4.]]

Enter move (0:up, 1:left, 2:right, 3:down): 0
[[  2.   8.  32.   2.]
 [256.  16.   4.  16.]
 [  4.   8.  32.   8.]
 [  2.   4.  16.   4.]]

YOU LOSE!
SCORE:  2192.0
```

Observed Results:

Agent	Average Score	Standard Deviation	Highest Tile Frequency
Expectimax	31,958.4	7502.50	2048: 80%, 1024: 100%, 512: 100%
TD-Learning	3149.44	1514.27	512: 15%, 256: 62%, 128: 96%
Q-Learning	2547.97	1225.03	512: 4.76%, 256: 50.94%, 128: 90.57%
Baseline	964.02	481.22	512: 0.01%, 256: 4.65%, 128: 45.87%

- **Q-Learning**: Moderate performance, improved over time. Reached 512 tiles (4.76% frequency). Average score: 2547.97. High time complexity due to iterative learning.
- **TD-Learning**: Performed better than Q-Learning. Reached 512 tiles (15% frequency). Average score: 3149.44. High time complexity due to state evaluations.
- **Expectimax**: Best performance with the highest average score (31,958.4). Frequently reached the 2048 tile (80%). Very high time complexity due to tree search.
- **Random Agent(Baseline)**: Rarely reached significant tiles like 512 (0.01% frequency). Lowest average score (964.02). Low computational time.

Challenges and Workarounds:

- **game.py**: Managing grid updates and user inputs was tricky due to complexity and the need for validation. We used functions like cover_up() and merge(), along with strict input checks, to ensure smooth transitions.
- **qlearning.py**: Balancing random actions and learned actions, and making sure the Q-table improved, was a challenge. We solved this by slowly reducing random actions (epsilon decay) and adjusting learning rates.
- **expectimax_agent.py**: High computational cost due to tree depth and stochastic outcomes was reduced by limiting tree depth, pruning, and using expected utility calculations.
- **tdlearning.py**: Delayed rewards and maintaining stability were difficult. We used eligibility traces and adjusted learning rates to solve these issues.
- **randomagent.py**: Random decision-making lowered performance, but restricting moves to valid ones provided slight improvement for baseline evaluation.

Future Work:

We aim to improve the AI to consistently achieve higher scores and reach the 2048 tile or beyond. To do this, we will refine its decision-making with advanced algorithms like Q-learning and Expectimax to better balance exploration and exploitation. By increasing the search depth in Expectimax, the AI will evaluate moves more effectively, leading to higher average scores. In TD Learning and Q-learning, increasing the number of training games will enhance performance and helps us to achieve higher tiles. Additionally, we will improve grid management and tile-merging strategies to avoid early congestion and and focuses on building higher-value tiles efficiently. These changes will make the AI smarter and more efficient in gameplay.

Github Repository Link: https://github.com/rishitharani24/AI_Team---ReinforceX