


| | |
|---|--|
|  | Zewail City for Science and Technology 2024-2025 |
|---|--|

| | |
|----------------|-------------------------|
| Course: | Artificial Intelligence |
| Code: | CSAI 301 |
| type | Project |

| | |
|----------------------|--|
| Due Date: | 28 dec 2024 |
| Project name: | Tic-Tac-Toe Q & Reinforcement learning |

Under The Supervision of

Dr. Doaa Shawky

Team:

| <u>Name:</u> | <u>ID:</u> |
|-------------------------|------------|
| Nour eldin mohamed | 202201310 |
| Ahmed Mohamed Abouelela | 202202070 |
| Yousef ahmed | 202202018 |

Table of Contents

| | |
|--|-----------|
| 1.About The Phase: | 3 |
| 2.Problem Modeling: | 4 |
| 1.State space representation: | 4 |
| 2.Action Space : | 4 |
| 3. Assumptions: | 4 |
| 4. Reward Structure | 4 |
| 3.Design Choices: | 5 |
| 1.Environment : | 5 |
| 2.Q - Learning agent design : | 5 |
| 3.System architecture for training (Training Based on Episodes): | 5 |
| 4.Functions: | 6 |
| 5. IMPLEMENTATION ANALYSIS: | 8 |
| 1. code structure : | 8 |
| 2.Algorithm Implementation: | 8 |
| 6.Results: | 10 |
| 1.Training Metrics: | 10 |
| 2.learning performance: | 10 |
| 7. CONCLUSIONS: | 11 |
| 1.strengthens: | 11 |
| 2.Limitations: | 11 |
| 7.Appendices: | 12 |
| Code Link: Phase II.ipynb | 12 |

Table of changes

| Date | Description | Version |
|----------|---|----------------|
| 25/12/24 | change the environment from maze to tic-tac-toe grid. | 2.0 (phase II) |
| 25/12/24 | changed problem modeling to suit the new environment. | 2.0 (phase II) |

1.About The Phase:

In this phase we implement Q-learning algorithm and reinforcement learning techniques to create an agent capable of analysing his environment and learn from his previous iterations. We implemented the agent in a tic-tac-toe game environment in which the agent is capable of recognizing the legal moves and play it and as he plays each game he evaluates the quality of the move which allows him in the future to evaluate which move is the best. Ultimately resulting in an agent that is competent in playing the game with high efficiency.

2.Problem Modeling:

1.State space representation:

- 3x3 Grid
- Values : 0 for (empty), 1 for (x) and -1 for (o)
- Space Size : 3^9

2.Action Space :

- Actions: Coordinates of (x,y)
- Possible Action : when cell value = 0
- Possible Action Size : 9

3. Assumptions:

1. Markov Decision Process (MDP):

- The current state contains all relevant information
- The future states depends on current state and its action

2.Environmental Determinism:

- Actions have outcomes that are deterministic
- No randomness in transition states

4. Reward Structure

- Win : + 1
- Draw : + 0.5
- invalid move : - 10
- Ongoing Moves : 0

3.Design Choices:

1.Environment :

Tic Tac Toe ENV class:

- A compact and efficient board representation using (numpy)
- Effective state management
- Algorithms for checking win conditions

2.Q - Learning agent design :

1.Key elements:

- Q- table : make sparse representation using (defaultdict)
- Use Epsilon- greedy for exploration strategy
- Algorithms for checking won conditions
- Use Epsilon decay

2.Hyper parameters:

- Learning rate (α) :0.1
- Discount factor (γ): 0.95
- Initial Epsilon (ϵ) : 0.1
- Epsilon decay rate : 0.995

3.System architecture for training (Training Based on Episodes):

- 10000 training episodes
- Evaluation intervals are regular
- Tracking performance metrics
- Demonstrating capabilities

4.Functions:

1.Class QLearningAgent:

A Q-Learning agent implementation for reinforcement learning tasks. This agent learns optimal action-value mappings through experience, using an epsilon-greedy strategy for exploration and the Q-learning update rule for value updates. The agent maintains a Q-table that stores state-action values and supports dynamic epsilon decay for balanced exploration-exploitation.

2.def get_action(self, state, valid_actions):

Selects an action for the current state using an epsilon-greedy strategy. With probability epsilon, the agent explores by choosing a random action then it exploits by selecting the action with the highest Q-value. When multiple actions share the maximum Q-value, one is chosen randomly to break ties.

3.def learn(self, state, action, reward, next_state, next_valid_actions):

Updates the Q-value for a state-action pair using the Q-learning update rule. This implementation follows the standard temporal difference learning approach, where the current Q-value is adjusted based on the received reward and the maximum Q-value of the next state. The update magnitude is controlled by the learning rate, and future rewards are discounted by gamma.

4.def decay_epsilon(self, decay_rate=0.995):

Reduces the exploration rate (epsilon) over time using exponential decay, allowing the agent to transition smoothly through exploration. The decay is set to maintain a minimum exploration rate of 1%, ensuring the agent never completely stops exploring in later stages of learning.

5.def display_game_step(env, state, action, q_values, step):

Provides a detailed visualization of each game step during demonstration mode. This function displays the current board state, the action taken, and the Q-values for all available actions.

6.def demonstrate_game(agent, game_number):

Executes and visualizes a complete game played by the Q-learning agent, providing detailed insights into the agent's decision-making process. This function displays each move's Q-values, indicates whether decisions were made through exploration or through the agent's knowledge, and shows the final game outcome.

7.def train_agent(episodes=10000):

Implements the main training loop for the Q-learning agent in the Tic-tac-toe environment. This comprehensive training function initializes both the environment and agent, executes the specified number of training episodes, tracks performance metrics including win rates and rewards, and preserves agent states for the visualisation.

8.def evaluate_agent(agent, n_games=100):

Assesses the agent's performance by running multiple evaluation games with exploration disabled. This function provides a more accurate measure of the agent's learned policy by focusing purely on exploitation, calculating key metrics such as win rate, draw rate, and average reward.

9.def plot_training_results(win_rates, rewards_history):

Generates visualization plots to track the agent's learning progress over time. This function creates two subplots: one showing the evolution of win rates and another displaying the history of accumulated rewards. The plots help in analyzing the agent's learning curve, convergence behavior, and overall training effectiveness.

5. IMPLEMENTATION ANALYSIS:

1. code structure :

In our project we implement the Tic-Tac-Toe problem as we did the implementation using Q-learning which is divided into different cells. In our project the first class is the class TicTacToe, which acts as an environment where the game takes place. This class will manage everything in relation to the game: checking the validity of moves, checking the status of the game.

We've added functions to reset the game, list all valid actions, and update the board when a move is made. We also have a QLearningAgent class in our project. This is where we implement the Q-learning algorithm. The agent uses a Q-table—what it learns is stored in this table, which we set up as a nested dictionary. Here, keys of the outer dictionary represent the state of a board, and keys in the inner dictionary represent an agent's possible actions over that state. The agent makes a move by the epsilon-greedy method, balancing exploration-exploitation trade-offs. It will also have an update rule by which it updates the values of Q depending on the outcome after each move.

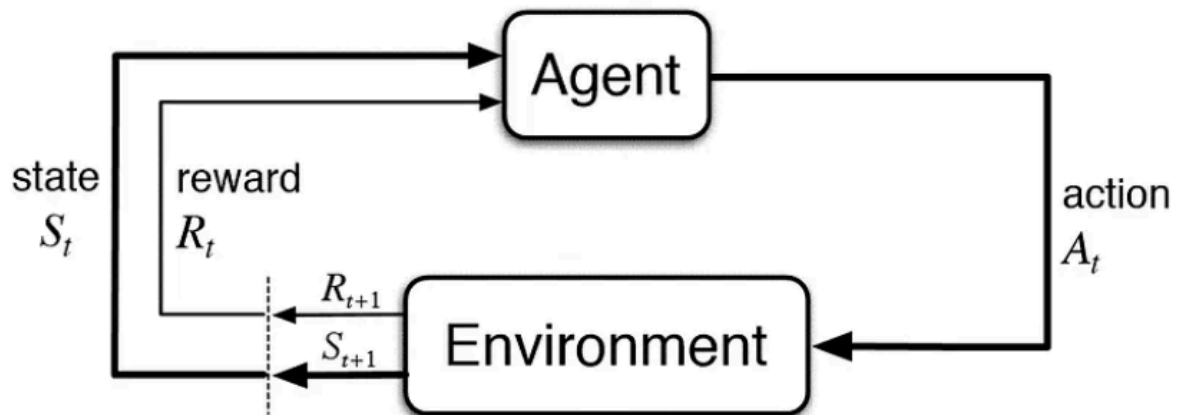
Our project also includes a training function to let the agent play many games (episodes). This helps it learn over time by improving its decision-making based on the outcomes of past games. Additionally, we've added a visualization feature to track the agent's progress, like plotting a graph to show how much better it gets as it learns. Our project structure helped us to divide the code into parts, making it easy to understand and maintain and to also incorporate new features in the future like adding additional visualization.

2.Algorithm Implementation:

The Q-learning algorithm used and implemented in our project is in a nested dictionary for the Q-table, where each state is serialized as a string and the actions are cell coordinates. The agent will select an action based on an epsilon-greedy strategy

balancing between exploration and exploitation. Now, the Q-learning update rule is applied as:

$$[Q(s, a) \leftarrow Q(s, a) + \alpha[R + \gamma \max_{a'} Q(s', a') - Q(s, a)]$$



This will help in updating the Q-values concerning immediate rewards but also future expectations. The structured approach allows an agent to learn optimal strategies iteratively in training.

6.Results:

1.Training Metrics:

The training metrics of the Q-learning where the agent monitors the win rate record every 100 episodes showing an improvement curve that shows that the agent is actually improving. To evaluate and validate the agent a final evaluation over 1,000 games was performed to show the agent's learning abilities as shown in the code:

```
Final Win Rate: 99.80%
```

2.learning performance:

The learning performance of the agent demonstrated its strength in a couple of important areas. Its ability to do good move selection allowed it to make decisions from an informed position, taking in the state of the board at that time. Second, it showed pattern recognition identifying winning configurations the Q-values proved to be useful as they indicated a stable learning progression and effective state-value estimation

7. CONCLUSIONS:

1.strengthens:

The implementation of our project has many features using the Q-learning has proven to be effective in helping the agent make directions. Through our implementation, we have successfully designed an efficient state representation that capitalize quick access to the game states, allowing the agent to make the best decisions. secondly the learning progression shown throughout the training process demonstrates the strength of our approach and the agent's ability to adapt and improve over time

2.Limitations:

Despite our implementation strengths there are some tradeoffs for example an increase in memory usage due to the large state space as the Q-table becomes more and more memory expensive.

7.Appendices:

Code Link:  Phase II.ipynb

GitHub Link: [tic-tac-toe_Qlearning](#)