Super Tic-Tac-Toe AI Project Report

1. Introduction

The Super Tic-Tac-Toe AI project aims to develop an intelligent agent for a modified version of Tic-Tac-Toe played on a 12x12 grid with restricted playable areas. Unlike traditional Tic-Tac-Toe, the game limits valid moves to five cross-shaped zones, excluding the four corner regions of the board. Players win by forming 4 consecutive pieces horizontally or vertically, or 5 consecutive pieces diagonally. The project employs reinforcement learning (RL) with a deep Q-network (DQN) to train the AI through self-play and includes a graphical user interface (GUI) built with Tkinter for human-AI interaction. This report details the project's objectives, design, implementation, and results.

2. Project Objectives

(1) Develop a Game Environment: Implement a robust game state management system to handle the 12x12 board, valid moves, and winning conditions.

(2) Train an RL Agent: Use a DQN-based RL approach to train an AI that learns optimal strategies through self-play.

(3) Create a User Interface: Build a Tkinter-based GUI for human players to interact with the trained AI.

(4) Incorporate Strategic Rewards: Design a reward system that encourages

the AI to form consecutive pieces (2, 3, or 4) while considering extendability and direction-specific priorities.

(5) Ensure Scalability and Performance: Optimize the implementation for efficiency, supporting both CPU and GPU execution.

3. Game Mechanics

3.1 Board and Valid Moves

Board Size: The game is played on a 12x12 grid (144 cells).

Valid Move Zones: Moves are restricted to five cross-shaped zones defined by CROSS_ZONES:

Central zone: (4,8,4,8)

Left zone: (0,4,4,8)

Right zone: (8,12,4,8)

Top zone: (4,8,0,4)

Bottom zone: (4,8,8,12)

Players can only place pieces in these zones where the cell is empty (data[row, col] == 0).

3.2 Winning Conditions

Horizontal/Vertical: A player wins by forming 4 consecutive pieces (same player value: 1 or -1) in a row or column.

Diagonal: A player wins by forming 5 consecutive pieces along the main diagonal or anti-diagonal.

Draw: The game ends in a draw if the board is full (data != 0) with no winner.

3.3 Reward System

The reward system incentivizes strategic play:

Horizontal/Vertical:

New extendable 3-consecutive pieces: +0.075 (0.15 * 0.5)

New 2-consecutive pieces: +0.025 (0.05 * 0.5)

Diagonal:

New extendable 4-consecutive pieces: +0.06 (0.12 * 0.5)

New extendable 3-consecutive pieces: +0.04 (0.08 * 0.5)

New 2-consecutive pieces: +0.015 (0.03 * 0.5)

Opponent Penalties: Negative rewards for opponent's new extendable consecutive pieces (e.g., -0.09 for horizontal/vertical 3-consecutive).

Game Outcome:

Win: +1.0

Loss: -1.0

Draw: 0.0

Extendability Check: For 3-consecutive (horizontal/vertical or diagonal) and 4-consecutive (diagonal) pieces, rewards are only given if at least one end of the sequence is empty, ensuring potential to form a winning line.

4. System Design

The project is structured into three main components:

(1) Game State Management (State class): Manages the board, validates moves, checks for consecutive pieces, and determines the winner.

(2) RL Agent (RLAgent and EnhancedDQN classes): Implements the DQN-based RL algorithm for training and decision-making.

(3) GUI (GameGUI class): Provides a visual interface for human-AI interaction using Tkinter.

4.1 State Class

Purpose: Represents the game state and handles core game logic.

Key Methods:

reset(): Initializes the 12x12 board as a NumPy array (np.zeros), sets winner to None, and end to False.

is_valid_position(row, col): Checks if a move is within the valid cross zones and the cell is empty.

check_consecutive(player, count, directions, row, col): Checks for count consecutive pieces in specified directions (horizontal, vertical, diagonal) around (row, col). Returns a tuple (has_consecutive, can_extend) to indicate if the sequence exists and is extendable.

update_state(row, col, player): Places a piece, updates the board, calculates rewards based on new consecutive pieces, and checks for a winner.

check_winner(): Detects 4-consecutive pieces horizontally/vertically or 5-consecutive pieces diagonally, or a draw if the board is full.

Data Structure: self.data is a NumPy array (int), where 0 represents an empty cell, 1 represents Player 1, and -1 represents Player 2.

4.2 RL Agent

Purpose: Trains an AI to play optimally using a DQN.

EnhancedDQN Class:

A convolutional neural network (CNN) with three Conv2D layers (64, 128, 256 filters) followed by two fully connected layers (512 units, output size 144).

Input: 12x12 board state (1 channel).

Output: Q-values for each of the 144 possible actions (flattened board positions).

RLAgent Class:

Initialization: Maintains a main model and a target model, with an AdamW optimizer, experience replay memory (deque, max 50,000), and epsilon-greedy exploration (epsilon: 1.0 to 0.05, decay 0.999).

get_action(state, training): Selects an action using epsilon-greedy during training or the model's Q-values during inference, restricted to valid moves.

store_experience(state, action, reward, next_state, done): Stores transitions in memory, converting board states to PyTorch tensors.

train_step(): Samples a batch (256), computes Q-loss using Smooth L1 Loss, updates the model, and periodically synchronizes the target model.

Training:

100 episodes of self-play, with Player 1 (AI) against a random opponent (Player 2).

Random move perturbations (50% chance to choose a nearby cell) add exploration.

Saves the model to best_model.pth after training.

4.3 GameGUI

Purpose: Provides a visual interface for human players to play against the trained AI.

Implementation:

Uses Tkinter with a 480x480 pixel canvas (12x12 cells, 40 pixels each).

Displays the board with cross-shaped valid zones, grid lines, and colored pieces (Player 1: green, Player 2: red).

Handles mouse clicks to place human moves, with a 50% chance of perturbing the move to a nearby cell.

AI responds with a move (epsilon=0 for deterministic play) after a 500ms delay.

Shows a game-over message box indicating the winner or draw.

5. Implementation Details

5.1 Technologies Used

Python Libraries:

NumPy: Board state management.

PyTorch: DQN implementation and tensor operations.

Tkinter: GUI rendering and interaction.

Hardware: Supports CPU or GPU (CUDA if available).

5.2 Key Features

Localized Consecutive Checks: The check_consecutive method restricts checks to a region around the latest move (row±count, col±count), improving performance.

Extendable Sequence Rewards: Rewards are only given for consecutive pieces that can extend to a winning line, enhancing strategic play.

Random Move Perturbations: Both training and GUI incorporate randomness (50% chance to adjust moves), encouraging exploration and robustness.

Experience Replay: The RL agent uses a replay memory to stabilize training and improve sample efficiency.

Batch Normalization and Dropout: The DQN includes batch normalization and dropout (0.3) to prevent overfitting.

5.3 Challenges and Solutions

Data Type Consistency: The check_consecutive method initially assumed self.data was a PyTorch tensor, causing a TypeError with NumPy arrays. The implementation was adjusted to use NumPy consistently, with conversions to PyTorch tensors in RLAgent.store_experience.

Balancing Rewards: Horizontal/vertical 3-consecutive pieces receive higher rewards (0.075) than diagonal 3-consecutive (0.04) due to their proximity to victory, fine-tuned through iterative testing.

Performance Optimization: Localized checks reduced the complexity of check_consecutive from O(1444count) to O(364count) for a 6x6 region.

6. Results and Performance

Training:

Trained for 100 episodes, with progress logged every 10 episodes (total reward and epsilon).

The AI learned to prioritize forming 3-consecutive pieces horizontally/vertically and blocking opponent's 4-consecutive diagonal sequences.

Gameplay:

The GUI provides a smooth experience, with responsive human-AI interaction.

The AI demonstrates competence in blocking human moves and pursuing winning sequences, though random perturbations occasionally lead to suboptimal moves.

7. Limitations

Limited Training Episodes: Only 100 episodes may not suffice for the AI to fully converge to an optimal policy.

Random Perturbations: The 50% move perturbation in training and GUI can disrupt strategic play, especially in critical positions.

Reward Tuning: The reward values (e.g., 0.15 for horizontal 3-consecutive) are empirically set and may require further optimization.

NumPy-PyTorch Integration: While functional, the reliance on NumPy for self.data and conversion to PyTorch tensors adds overhead.

8. Future Improvements

Increase Training Episodes: Extend training to 1000+ episodes to improve AI performance.

Refine Reward System: Use hyperparameter tuning to optimize reward values, potentially incorporating rewards for blocking opponent moves explicitly.

Remove Random Perturbations: Replace perturbations with a more controlled exploration strategy, such as softer epsilon-greedy policies.

Full PyTorch Migration: Convert self.data to a PyTorch tensor throughout

the State class to streamline computations and leverage GPU acceleration.
Advanced DQN Variants: Implement Double DQN or Dueling DQN to enhance learning stability and performance.
GUI Enhancements: Add features like move history, undo functionality, and difficulty levels.

9. Conclusion

The Super Tic-Tac-Toe AI project successfully delivers a functional RL-based AI capable of playing a complex variant of Tic-Tac-Toe. The implementation integrates a robust game state manager, a DQN-based RL agent, and an interactive GUI. Despite limitations in training duration and move randomness, the AI demonstrates strategic competence, and the GUI provides an engaging user experience. Future improvements in training, reward design, and PyTorch integration can further enhance the project's effectiveness.