TicTacPro: Exactly Solving a Combinatorial Piece-Placement Game and Stress-Testing Deep Q-Learning Against It

Abdul Khan abdulskhan47@gmail.com https://github.com/abdulllkhan/ticTACpro-sim
July 2026

Abstract

We study TicTacPro, a combinatorial board game extending Tic-Tac-Toe with three piece sizes per player and two winning conditions. An exhaustive alpha-beta solver proves the game is a *first-player win from every one of the 27 opening moves* (700M nodes, 246 s on 20 cores). Notably, time-limited heuristic minimax self-play had first converged to the *opposite* conclusion — a deterministic second-player win — a spurious result the exact solver refutes, illustrating how agreement among strong but budget-bounded agents can masquerade as a game-theoretic proof. We further document a block-priority bug that recurred across nine independent code paths, and train a Dueling Double DQN (792K parameters) with a curriculum of rule-based opponents, reaching 100% win rates against random and BullseyeAgent opponents. A post-hoc audit surfaces schedule-calibration and checkpoint-selection pitfalls, reported as negative results.

1 Game Description

TicTacPro is played on a 3×3 board. Each player holds nine pieces: three Small (S), three Medium (M), and three Large (L). On each turn a player places one piece on any cell. A cell has one slot per size, so a cell can hold up to three pieces. A player wins by achieving either:

- **3-in-a-row (same size):** three pieces of identical size along any row, column, or diagonal; or
- **Bullseye:** one S, one M, and one L piece placed in the same cell.

A game lasts at most 18 plies, with up to 27 moves per ply — substantially harder to search than Tic-Tac-Toe, yet still small enough to solve exactly, which proved essential (Section 2).

2 Game-Theoretic Analysis

2.1 Exact Solution: a First-Player Win

We solved the game with a pure negamax alpha-beta search over the complete game tree: no heuristic evaluation, no depth cap, terminal-only scoring (win/loss/draw), with a transposition table, parallelised over the 27 opening moves on 20 CPU cores. **Every one of the 27 opening moves is a proven win for the first player (RED).** Individual openings required 9.2M (TR-M) to 53.7M (BC-S) node evaluations; 246 s wall-clock total. The per-opening table and solver are in the repository (`exact_solve.py`, `exact_solve_results.json`).

2.2 The Search-Horizon Trap

Before the exact solve, we ran time-limited iterative-deepening minimax (3 s/move, heuristic leaf evaluation, late-move reductions, aspiration windows) against itself. Five independent games produced an *identical* 16-move sequence in which BLUE (the second player) wins via a "sequential bullseye fork": build two sizes of a bullseye in one cell, force RED to spend a piece blocking, pivot to a fresh cell, repeat until RED runs out of blocks. Its determinism made this look like a solution — we initially recorded the game as a second-player win.

The exact solver shows why this was wrong. Proving RED's forced win from the minimax engine's own book opening (TC-L) requires ∼31.8M node evaluations; the online engine evaluates only ∼150–300K nodes per move — two orders of magnitude short. The fork is a real and potent strategy *against bounded search*, but it is not the game value. Nor does reactive play execute it safely: a rule-based BullseyeAgent implementing the fork beats the rule-based anti-diagonal OptimalAgent 20/20 in *both* colour roles (heuristic dominance, not a second-player advantage), yet loses 10/10 to the same bounded minimax via double attacks no reactive policy can prevent. **Lesson:** deterministic agreement among strong, budget-bounded agents is weak evidence about game value; exhaustive search reversed the answer.

3 The Block-Priority Bug Family

Nine move-selection code paths shared a subtle correctness bug: a single `block_mv` variable was set on the *first* blocking move found in scan order (Small→Large, cell 0→8). When both a *bullseye threat* (opponent has 2/3 sizes in one cell) and a *line threat* (opponent has 2/3 of a same-size row) coexist, scan order — not danger — determined which was blocked.

With flat board index idx $= 9r + 3c + \text{sz}$, two partner indices i_1, i_2 of a win line lie in the same cell iff $\lfloor i_1/3 \rfloor = \lfloor i_2/3 \rfloor$, identifying bullseye threats in $O(1)$. The fix tracks the two threat classes in separate variables (`bullseye_block`, `line_block`) and prefers the bullseye block. It was propagated to all nine affected paths: OptimalAgent, NeuralMCTS fast-path, NeuralMCTS move sorting, the C move-sorting extension, MCTS move sorting, the Python rollout policy, the C rollout, the C single-move selector, and the ParallelMCTS fast-path. Each site received a regression test. The recurrence of one latent flaw across nine independent implementations of "win > block > rest" is itself the finding: shared decision logic should be shared code.

4 Deep Q-Learning Agent

The 61-dimensional state tensor is normalised to the current player's perspective: channels 0–26 encode the current player's pieces (one per cell-size slot), 27–53 the opponent's, 54–59 the two players' remaining piece counts (scaled by 1/3), and channel 60 flags whether the current player is RED (the first mover).

Negamax Double-DQN target. The zero-sum structure lets us write the target from the opponent's perspective, with the online network selecting among *legal* actions and

Component	Value
Architecture	Dueling Double DQN
Parameters	792,604
Action space	27 (9 cells × 3 sizes)
State dimension	61
Precision	BF16 AMP + `torch.compile`
Replay buffer	10M (GPU-side PER)
Symmetry augmentation	8-fold (D_4), at collection
Parallel collectors	16 CPU workers (spawn)
Episodes (run 8, 2 h)	1,003,520
Hardware	NVIDIA GB10 DGX Spark

Table 1: Training configuration for the final run.

the target network evaluating:

$$\hat{Q} = r - (1-d)\,\gamma\, Q_{\theta^-}\big(s', \arg\max_{a' \in \mathcal{A}(s')} Q_\theta(s', a')\big),$$

where $r \in \{+1, 0, -1\}$ is the terminal reward, d the done flag, and $\mathcal{A}(s')$ the legal-action mask. The opponent's best continuation is *subtracted*, matching the minimax Bellman equation.

8-fold symmetry. The dihedral group D_4 yields 8 board symmetries. Each collected transition is expanded into its 8 symmetric variants *before insertion* into the replay buffer (states via inverse permutation gather, actions via forward permutation; count channels and the first-mover flag are invariant), multiplying stored experience by 8×.

Systems optimisations. A C rollout extension runs entire greedy rollouts in one call (4.3× over Python); collector workers use a pure-NumPy inference path (19× over PyTorch CPU); PER sampling runs on-GPU via prefix-sum and `searchsorted`.

Curriculum. Pure self-play never encounters the bullseye-fork strategy, so the DQN never learns to block it. The final run therefore replaced one player with the rule-based BullseyeAgent in a majority (≈56%, measured) of collected episodes, in both colour roles.

5 Results

Match	W	D	L
DQN vs Random (both roles, 40g)	40	0	0
DQN vs Bullseye (both roles, 40g)	40	0	0
DQN (RED) vs Optimal (BLUE), 20g	0	20	0
DQN (BLUE) vs Optimal (RED), 20g	0	0	20

Table 2: Post-training tournament (rule-based agents).

Overall win rates: BullseyeAgent and DQN 66.7% each, OptimalAgent 50.0%, Random 0.0%. Three observations. **(1) Curriculum transfer:** before the BullseyeAgent curriculum the DQN lost every benchmarked game to it (0/20); after, it wins 40/40 — it learned to block the sequential bullseye fork. **(2) Colour asymmetry:** the DQN draws OptimalAgent when playing first but loses 20/20 when playing second; it never saw the anti-diagonal opening plan in training. **(3) Tree search:** in the tournament series with search agents, pure MCTS with win-then-block greedy rollouts reached a 91.7% overall score (tournament v8), and replacing its random rollouts with deterministic OptimalAgent rollouts made it *worse*: rollout diversity mattered more than rollout strength.

6 Limitations (Post-Hoc Training Audit)

Auditing the pipeline after the final run surfaced three flaws. **Stale-throughput schedules:** epsilon, PER-β, and LR schedules assumed a hard-coded 120K gradient steps/hour; the run achieved ∼31K/h, ending at $\varepsilon = 0.457$ and $\beta = 0.649$ — no exploitation phase, incomplete importance-sampling correction. Schedules should track measured progress. **Saturating checkpoint metric:** the best-checkpoint criterion (win rate vs. a deterministic heuristic) hit 100% eleven minutes in and could never improve again, so the benchmarked checkpoint reflects 143,872 of the 1M episodes; deterministic-vs-deterministic evaluation also carries only 1 bit of signal per colour role. **Silent loss signal:** in heuristic-opponent games only DQN-side transitions are stored; when the opponent's move ends the game the DQN's final transition keeps $r = 0$, $d = 0$, so losses must be learned through bootstrapped opponent-to-move states — exactly the states such games never store.

7 Conclusions

Our agent combines the standard value-based stack — DQN [1], Double Q-learning [2], dueling heads [3], prioritized replay [4] — with UCT [5] baselines and a classical alpha-beta solver; the need to verify agent consensus exhaustively echoes solved-game efforts such as checkers [6]. Contributions:

1. **Solved game:** TicTacPro is a first-player win from all 27 openings; bounded minimax self-play had confidently produced the opposite answer.
2. **Bug pattern:** one latent block-priority flaw recurred in nine independent implementations; dual-tracking threat classes is the reusable fix.
3. **Curriculum beats self-play blindness:** a rule-based adversary in collection taught the DQN a defence self-play never discovered (0/20 → 40/40).
4. **Audit your pipeline:** the flaws in Section 6 were invisible in headline win rates and surfaced only through post-hoc log analysis.

Reproducibility. `exact_solve.py` reproduces the game-value proof (∼4 min, 20 cores); `train_spark.py` and `tournament.py` in the repository reproduce the training run and Table 2.

References

[1] V. Mnih et al. Human-level control through deep reinforcement learning. *Nature*, 518:529–533, 2015.

[2] H. van Hasselt, A. Guez, D. Silver. Deep reinforcement learning with Double Q-learning. *AAAI*, 2016.

[3] Z. Wang et al. Dueling network architectures for deep reinforcement learning. *ICML*, 2016.

[4] T. Schaul, J. Quan, I. Antonoglou, D. Silver. Prioritized experience replay. *ICLR*, 2016.

[5] L. Kocsis, C. Szepesvári. Bandit based Monte-Carlo planning. *ECML*, 2006.

[6] J. Schaeffer et al. Checkers is solved. *Science*, 317:1518–1522, 2007.