COMP0215: Reinforcement Learning for Robotics and AI

Project Report:
Autonomous Gomoku Agent

Kishan Grewal — 24087033
Yash Joshi — 24003413
Xavier Parker — 24077390
Helitha Cooray — 24161798

Department of Computer Science
University College London

04/04/2026

Contents

1 Introduction **2**

2 Creating Game Dynamics and Visualisation **2**

3 Approach 1: DQN **3**

3.1 How DQN works . . . 3
3.2 Why we chose DQN . . . 5
3.3 Reward Tuning for DQN . . . 5

4 Approach 2: AlphaZero **19**

4.1 Algorithm Overview . . . 19
4.2 Adaptation for 9×9 Gomoku . . . 19
4.3 Batched MCTS for GPU Efficiency . . . 20
4.4 The Self-Play Distribution Gap . . . 20
4.5 Tactical Threat Detection in Training MCTS . . . 20
4.5.1 The 50% Randomisation . . . 21
4.6 Training and Hyperparameter Analysis . . . 22
4.6.1 Learning Rate . . . 22
4.6.2 Replay Buffer Sizing . . . 22
4.6.3 Training Epochs per Iteration . . . 22
4.6.4 MCTS Simulations . . . 22
4.7 Ablation: Comparing Two Models . . . 22
4.8 Limitations . . . 27

5 Benchmarking: Round Robin **28**

5.1 Win Rate Matrix . . . 28
5.2 Per-Agent Summary . . . 29
5.3 Game Length and Decisiveness . . . 29
5.4 First-Player Advantage . . . 30

A Team Contributions **31**

B Program Code **31**

C Video **31**

D Bibliography **31**

1 Introduction

Deterministic board games have long served as one of the best proving grounds for AI. Games like Go and chess offer a rigorous space for testing complex, strategic decision making. Unlike early heuristic-based game engines such as IBM's Deep Blue[1], AI in board games, with its current state-of-the-art architectures, no longer rely on human domain knowledge or pre-programmed instructions. This has been proven in many occasions, with notable ones being when AlphaGo[2] managed to beat 9-dan Go professionals. Newer models such as **AlphaZero**[3], which we have implemented for this coursework, have achieved superhuman performance, learning entirely through self play and generalised planning algorithms. This has made them achieve thinking patterns and make decisions that are similar to what humans would consider "intuitive" or "feels right", yet struggle to give a logical explanation. However, replicating these massive, compute-heavy successes under highly constrained hardware limits remains a significant and active challenge in the field.

Figure 1: 9-dan Go professional Lee Sedol playing against DeepMind's AlphaGo (Google DeepMind Challenge 2016)

Gomoku (Five-in-a-row) presents an interesting challenge within this domain. While the rules of the game are quite simple, the game has a complex state space and requires a delicate balance between proactive and reactive structural development, which would mean a good agent would be capable of being both playing aggressively and defending when required.

Our objective in this coursework was to develop an autonomous agent capable of learning and evolving its own strategies using **Reinforcement Learning**. By targeting Gomoku on a 9×9 grid, we establish an environment that forces our agent to improve its performance from zero knowledge or random play, without any hardcoded rules.

In this report, we discuss two fundamentally different paradigms of reinforcement learning to tackle this challenge. We have built a game environment, which feeds into our training pipelines. We have implemented a value-based approach using **Double Deep Q-Networks**[4], and also an implementation of an AlphaZero architecture. Here, we address our findings about the impact of network structure, reward shaping, curriculum learning, and the challenges of adapting an architecture originally designed for many TPUs to run parallelly rather than on a single consumer GPU, introducing a novel tactical threat detection mechanism. Finally, we will be demonstrating a round-robin competition to benchmark our trained models to show the distinct advantages of model-based tree search when navigating vast state spaces.

2 Creating Game Dynamics and Visualisation

We developed our game environment as a self-contained class that manages all board state. The board is represented by a 9×9 *NumPy* array, in which each cell holds an integer value corresponding to the colour **(0 for empty, 1 for black, and 2 for white)**. Representing the board this way is efficient, as checking occupancy, counting stones in a certain direction, and generating moves can all be done through simple array operations.

Each turn, the agent calls in a *step()*function with an action determining the target position as *(row, column)*. The method places the stone, records the move in the history, and then immediately checks for a win by scanning all directions through the placed stone (horizontal, vertical, and both

diagonals). Rather than checking the entire board after every move, the win check only examines lines passing through the most recently placed stone, which is both correct and efficient, as a new winning line must include the last move. If no winner is found and no empty cells remain, the result is set to a draw. The current player only flips if the game is still ongoing, which avoids an off by one situation when reading the result immediately after a terminal move.

The environment also supports full state cloning, which produces a deep copy of the board, move history, and game result, which can be used during self-play and tree search to simulate future positions without modifying the live game state.

aparFor visualisation, we built a *Pygame* renderer that has an interface similar to a traditional Gomoku board. The most recently placed stone is highlighted, giving an immediate visual cue about where the last move was played. A status bar at the bottom displays the current player who is yet to make a move, and updates to a result message when the game ends. The interface also handles inverse mapping from mouse coordinates back to board intersections, so a user would be able to click and play against an agent as well.

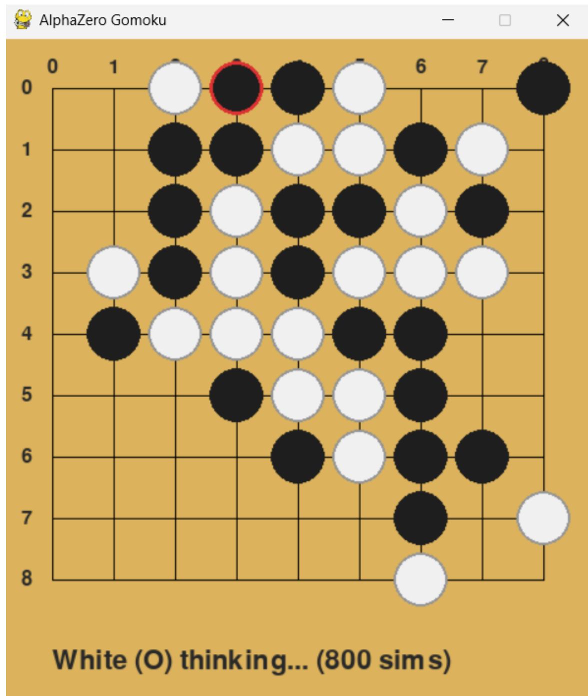


Figure 2: The *Pygame* visualisation, halfway through a self-play game.

Training metrics are tracked through a logger that writes simultaneously to a CSV file and TensorBoard. each episode records the outcome, total shaped reward, mean loss, epsilon, buffer size, and game length.

3 Approach 1: DQN

3.1 How DQN works

Deep Q-Networks ¡citation goes here¿ treats the problem of learning to play the game as a function approximation problem over action values. It stems from Q-learning, which involves maintaining an estimate $Q(s, a)$ that represents the expected cumulative discounted reward of taking action a in state s, and then acting greedily with respect to those estimates. Classical Q-learning would store this table explicitly, which is intractable for any large state space. DQN replaces this table with a neural network, that's parametrised by θ, and is trained to minimise the Squared Bellman error, which represents the DQN loss:

$$\mathcal{L}(\theta) = \left[\left(r_{t+1} + \gamma \max_{a_{t+1}} \hat{Q}_\theta(s_{t+1}, a_{t+1}) \right) - \hat{Q}_\theta(s_t, a_t) \right]^2 \tag{1}$$

DQNs make use of two stabilisation techniques. The first is **experience replay**. Transitions (s, a, r, a') are stored in a **replay buffer** and sampled uniformly at training time, which breaks temporal correlations that would otherwise result in the gradient updates being highly non stationary. As shown in 4, our implementation's buffer reaches its transition capacity quite rapidly during the initial stages of the training curriculum. The second is a **target network**, which is a copy of the online network used to compute the Bellman targets, that is periodically frozen. Without this, the target $r + \gamma \max Q$ is computed using the same network being updated, creating a moving target that would destabilise training. In our implementation, the target network was synchronised with the online network every 2000 gradient steps.

Our best implementation was a **Double DQN** (DDQN). The standard formulation couples action selection and action value evaluation in the Bellman target. Both these processes use the same network, which tends to overestimate Q-values because the max operator introduces an upward bias. DDQN fixes the issue by decoupling them, so the online network selections the best next action, and the target network evaluates it.

$$\mathcal{L}(\theta) = \left[\left(r_{t+1} + \gamma\, \hat{Q}_{\theta^-}\left(s_{t+1}, \arg\max_{a_{t+1}} \hat{Q}_{\theta}(s_{t+1}, a_{t+1}) \right) \right) - \hat{Q}_{\theta}(s_t, a_t) \right]^2 \tag{2}$$

For a board game like Gomoku, with an explicit legality constraint, we had to make a further modification with **action masking**. Before the online network selects the next action, occupied squares are identified from the state tensor (any square where channel 0 or channel 1 is non-zero) and their Q values are set to -10^9, which results in *argmax* never selecting an illegal move. Without this, the Bellman target can be computed using a Q value for an impossible state transition, introducing a systematic error into the value estimates. This is also applied during training, not just inference.

The network itself is a **residual convolutional architecture**. The input is a $3 \times 9 \times 9$ tensor with one binary channel for the agent's stones, one for the opponent's, and one for indicating whose turn it is. A stem 3×3 convolution projects this to 64 channels with batch normalisation and ReLU, followed by four residual blocks each containing two 3×3 convolutions with a skip connection. The residual structure is important, as a plain CNN with similar depth would have suffered from vanishing gradients, and a shallow receptive field, struggling to evaluate threats spanning the full board. The skip connections allow the network to propagate both local pattern information and global context. The convolutional output is flattened and passed through two fully connected layers, and finally produces 81 scalar Q values.

During training, our agent makes use of 8-folder symmetry augmentation. Gomoku on a square board is invariant under the transformations that make up the dihedral group (4 rotations $\times$ 2 reflections), so every transition is stored as 8 symmetric equivalents before they get added to the replay. Effectively, this multiplies the training data 8 fold and does not result in additional game generation cost, which encourages the network to learn position evaluations thate a consistent across equivalent board orientations.

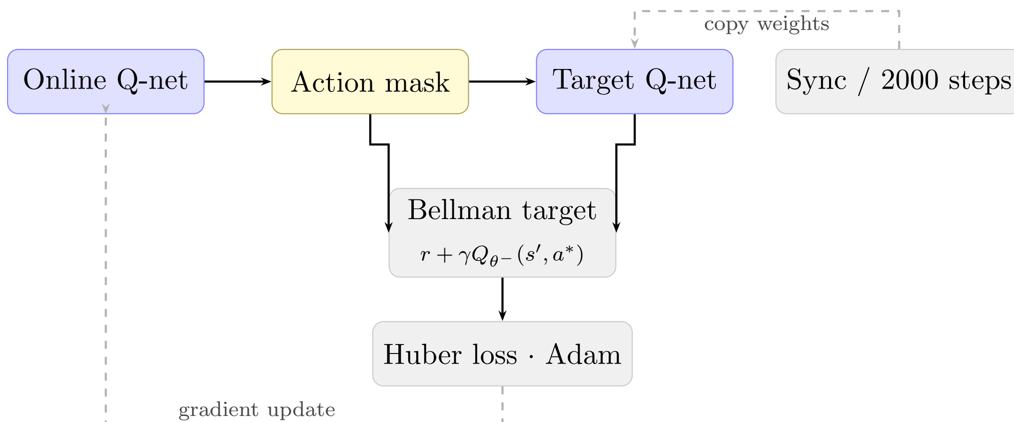


Figure 3: Double DQN training loop. The online network selects the best legal action after masking occupied squares; the target network evaluates it. The Bellman target is used to update the online network via Huber loss and Adam. Target network weights are copied every 2000 gradient steps.

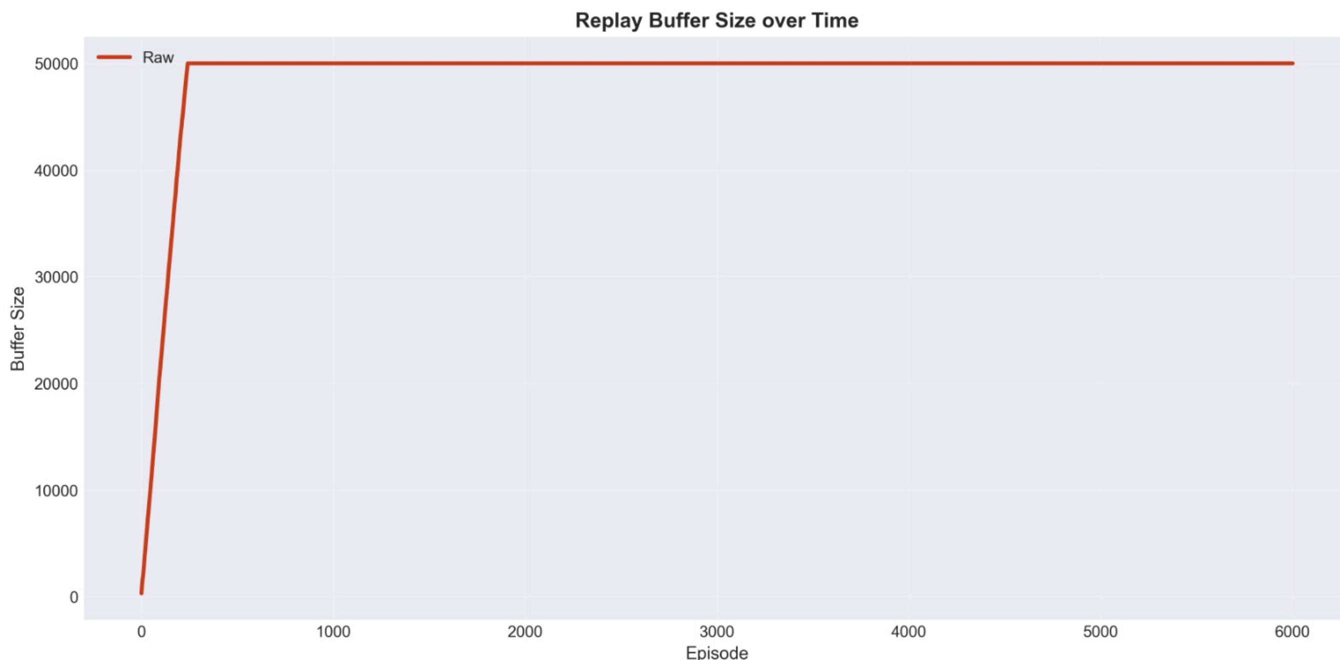


Figure 4: Rapid filling of the the experience replay buffer during the early stages of training. The buffer reaches it fixed capacity of 50,000 transitions within the first few hundred episodes, after which older transitions are sampled uniformly and replace to break temporal correlations for stable gradient updates

3.2 Why we chose DQN

Our primary reason for DQN was the nature of the state and action spaces in Gomoku. A 9×9 board, with three states (empty, black stone, white stone) has 3^{81} theoretically possible configurations, so any form of tabular representation would not be suitable. DQN handles this directly through function approximation, and the convolutional architecture is partially well suited to board games, as the spatial structure of the board is preserved throughout the network, so the same pattern detector (eg. a three in a row threat) can be recognised regardless of where it appears on the board.

We realised that DQN was also well matched to the training we wanted to implement. Since it learns off-policy, the replay buffer can contain transitions generated by any prior version of the agent, and it is naturally compatible with curriculum learning. We have trained it against a random agent, heuristic agents, and also via self play, and transitions from all three phases can coexist in the buffer.

Compared to value-free policy gradient methods, DQN also provides an explicit value signal at every legal move on every turn, which is useful for evaluation and debugging. During development we could inspect Q-values directly to understand whether the agent was correctly valuing threats, something that is less transparent with pure policy methods.

3.3 Reward Tuning for DQN

The hyper parameters in training that we could tune can be seen in the table below. We left almost everything in the learning category the same as we knew the training was stable and produced good results. We tweaked the parameters in Episodes and in Game to try and make our agent better.

Episodes	Learning	Game
• RANDOM_EPISODES	• EPSILON_DECAY	• OPEN_TWO
• HEURISTIC_EPISODES	• LEARNING_RATE	• HALF_THREE
• CHECKPOINT_INTERVAL	• TARGET_UPDATE_FREQ	• OPEN_THREE
• SELFPLAY_EPISODES	• TRAIN_STEPS_PER_EPISODE	• HALF_FOUR
• TOTAL_EPISODES	• BUFFER_CAPACITY	• OPEN_FOUR
• OPPONENT_EPSILON	• GAMMA	• BLOCK_THREE
• OLD_OPPONENT_CHANCE	• EPSILON_END	• BLOCK_FOUR
	• EPSILON_START	• WIN_REWARD
	• BATCH_SIZE	• LOSS_REWARD
		• DRAW_REWARD
		• STEP_PENALTY

The first curriculum consisted of 1000 games against a random move agent, then 1000 games against a subpar heuristic model, followed by 4000 games against itself. To facilitate this progression, we implemented **exponential epsilon decay**, visualised in 5, allowing the agent to move from pure exploration in the early random phase to a stable, greedy policy during the self-play refinement stage. We chose this split so that the agent learns some basic strategy for Gomoku against the random and heuristic bots, and then would hone its strategies playing itself. The reward structure can be seen in table 1. These rewards were chosen to sort of mimic the priorities a heuristic bot would follow. BLOCK_FOUR being fairly close (0.8) to WINREWARD was done on purpose to emphasise that blocking your opponent when they have 4 in a row is almost as good as winning because it means you lose. We decided to really punish losing by giving it a score of -2, to really discourage the agent from losing. OPEN_TWO and the rewards associated with 3 in a row are barely nudges, this was designed to encourage the agent slightly to place tiles near each other but not to force it to place them next to each other, additionally in the later game we want the agent to be playing 4 in a row and winning, if the reward for 2 in a row is too high it might have chosen to place another 2 in a row rather than place the fourth in a row somewhere else. This model took 30 minutes to train on a laptop without a GPU, allowing us to train multiple models quickly and iterate on reward design.

Variable	Reward
OPEN_TWO	0.01
HALF_THREE	0.02
OPEN_THREE	0.08
HALF_FOUR	0.15
OPEN_FOUR	0.6
BLOCK_THREE	0.2
BLOCK_FOUR	0.8
WIN_REWARD	1.0
LOSS_REWARD	-2.0
DRAW_REWARD	0.1
STEP_PENALTY	0.0

Table 1: Rewards for first training iteration

This agent was not very good against humans, as can be seen in figure 6. It does not block 4 in a row,

Figures 6a, 6d and 6b, a crucial flaw in its strategy. However, it clearly has a strategy of wanting to control the centre as it very rarely made moves on the edge of the board. In addition, the agent does not instantly place the fifth piece in a row, but when given enough time, even with 3 ways that I could win, in Figure 6b, the agent can win. In Figure 7a the outcome against the random agent became very good very quickly and then an expected sharp drop when the curriculum switched to heuristic as now the opponent plays with an intent. The outcome gradually increases, validating our decision to use the heuristic agent.

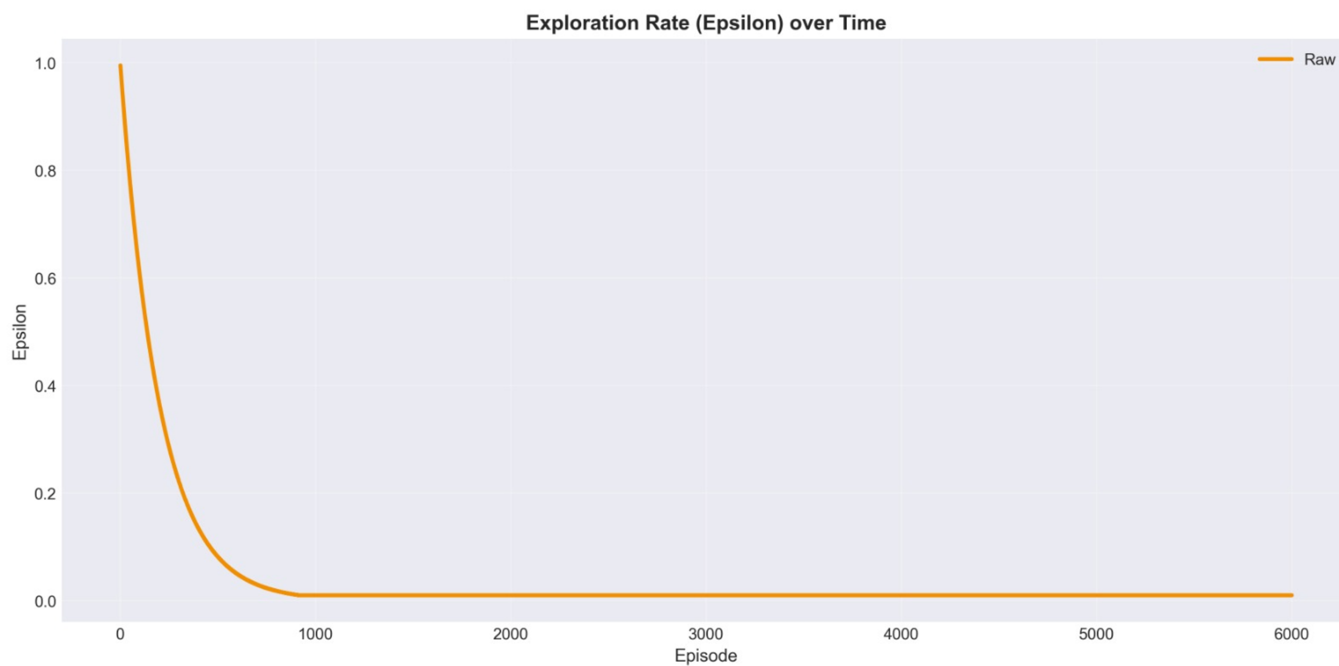
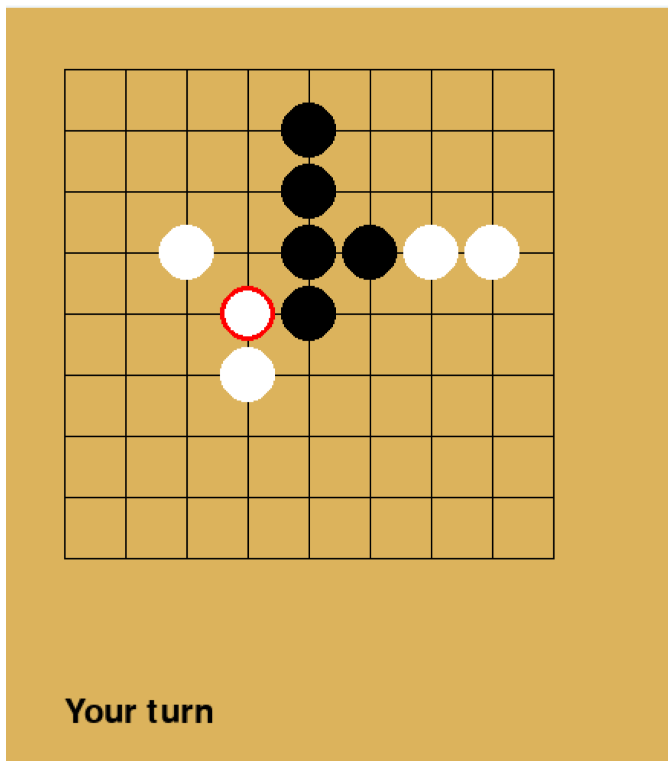
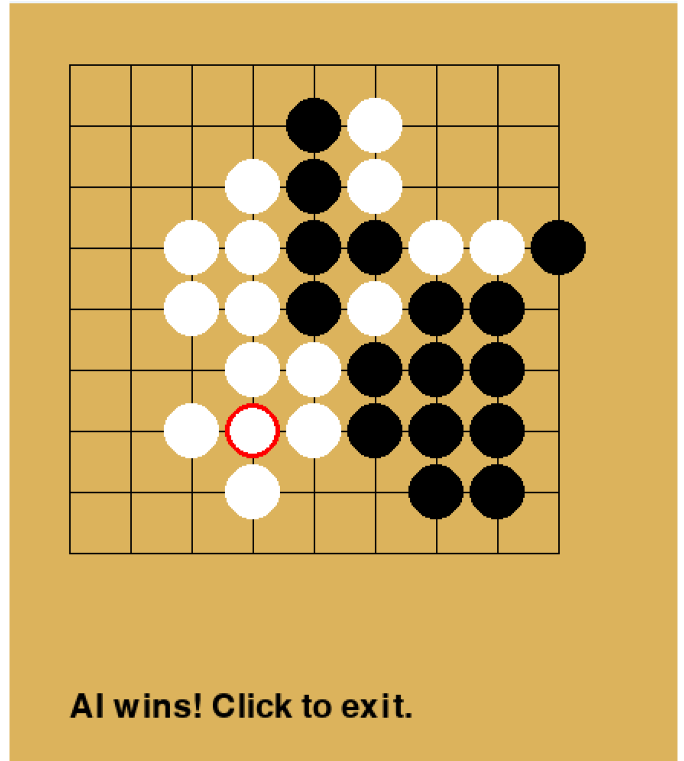


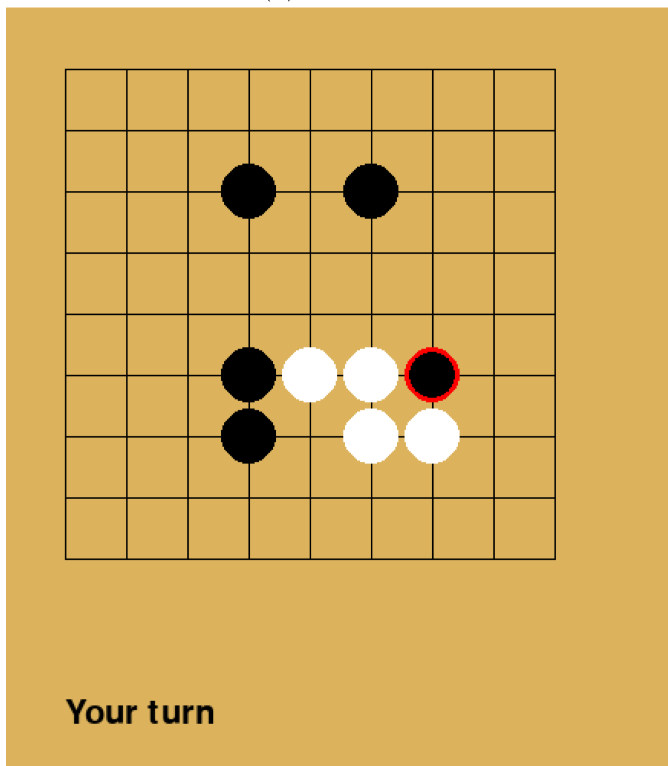
Figure 5: Decay of the of the epsilon (ϵ) parameter over the 6000 episode training curriculum. Agent starts with pure exploration and gradually transitions to a greedy policy.



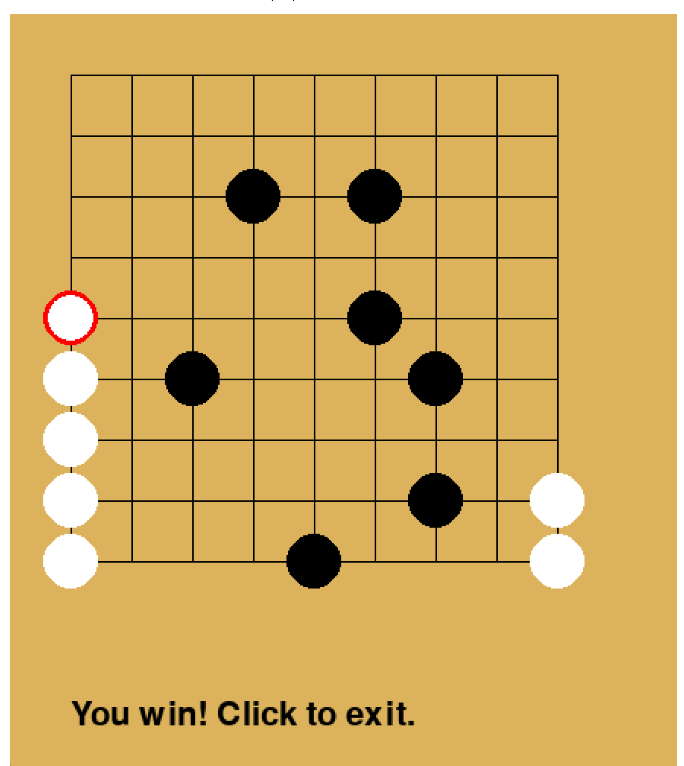
(a) AI white



(b) AI white

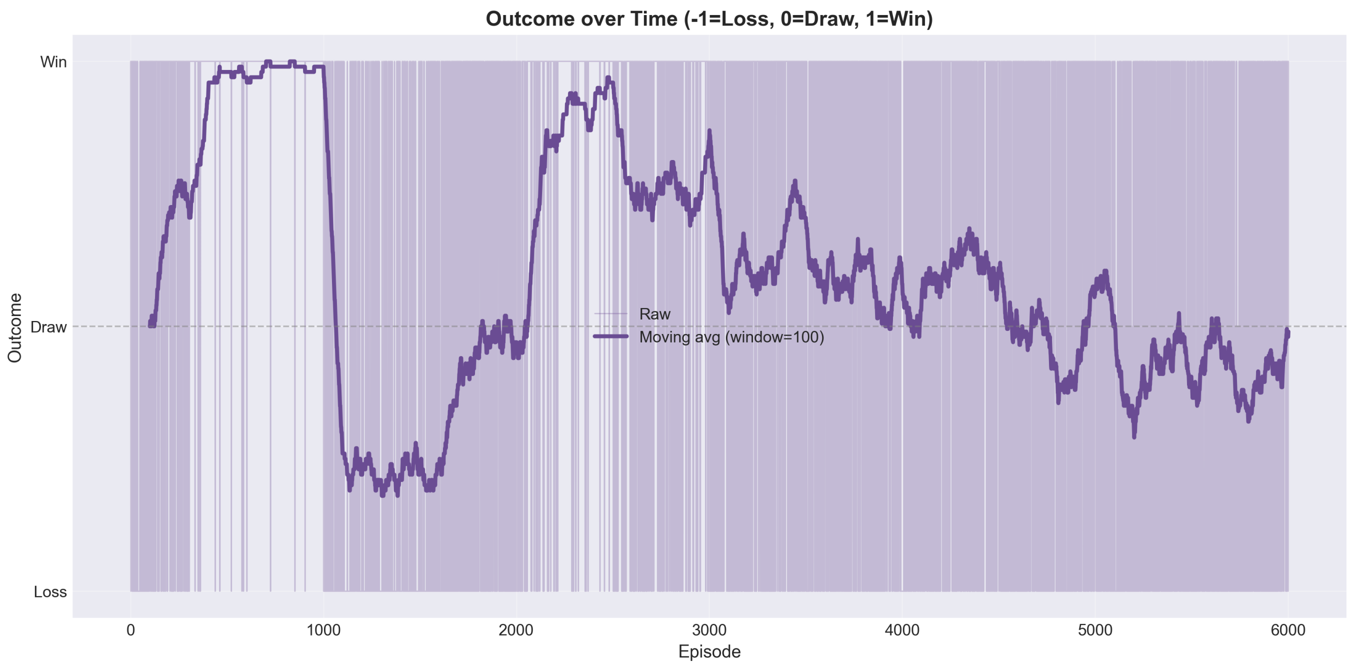


(c) AI Black

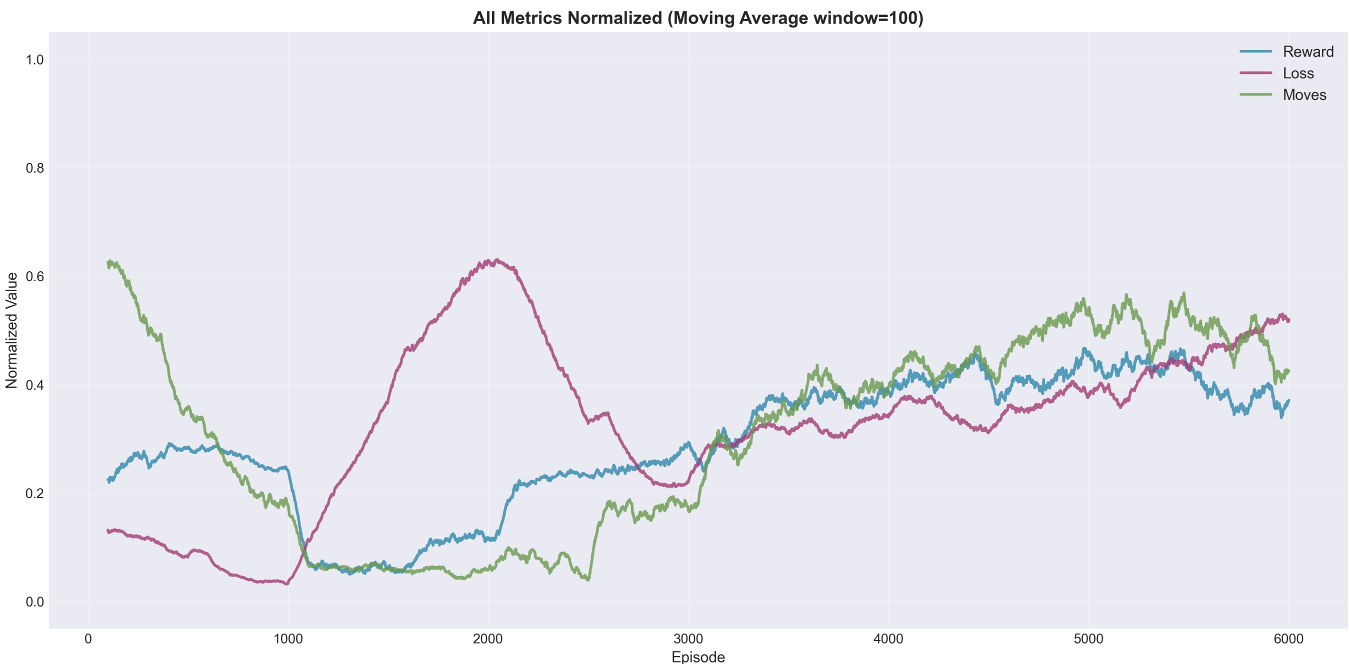


(d) AI Black

Figure 6: Figures showing certain interesting moves the DQN agent has taken. This was the first iteration of the DQN, with rewards structured in Table 1.



(a) Figure showing the rolling average outcome of the agent during its training. A win is 1, a loss is -1.



(b) Figure showing the rolling average normalised metrics for reward (blue), loss (red) and moves (green) over its training

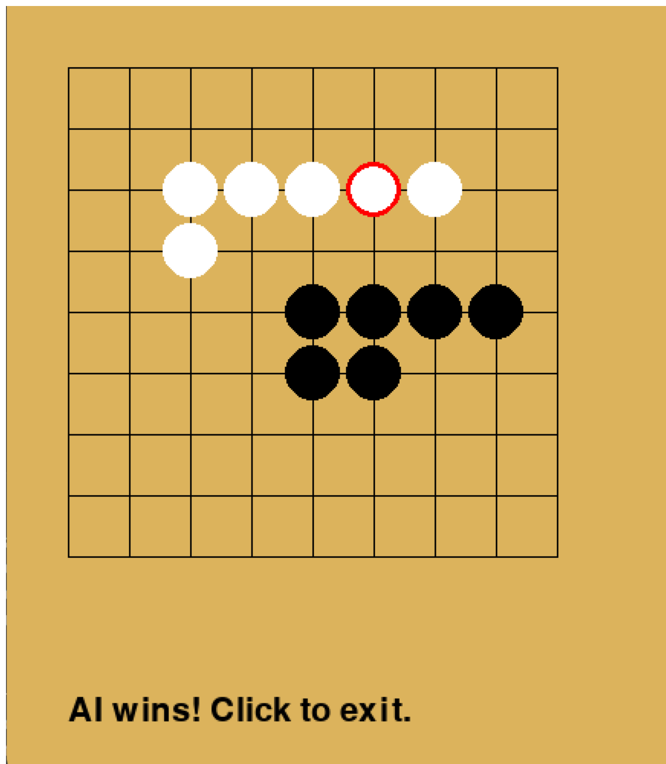
Figure 7: Figures showing the rolling average outcome (left) and the rolling average normalised metrics of reward, loss and moves (right). Both are for the first DQN iteration

To help fix these problems in the agent's strategy we tuned the rewards. We kept the curriculum the same, as we wanted to experiment on changing just the rewards. The biggest change was setting the reward for a win to 10, dramatically higher than any other reward. We also increased the rewards for three and fours, this was to aim to try and get the agent to create more structures on the board that it can then go on to win from. blocking a four was also increased to try and get the ai better at blocking.

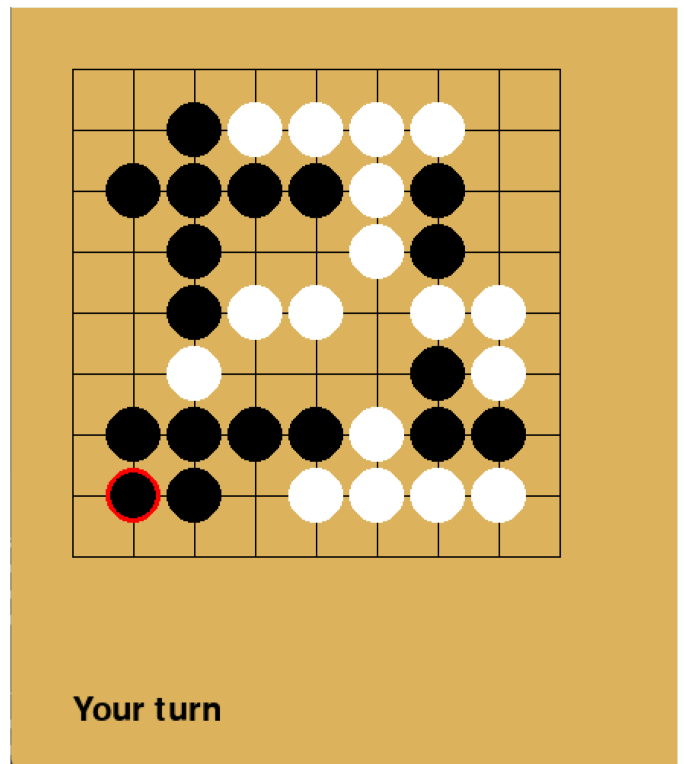
Variable	Previous Reward	New reward
OPEN_TWO	0.01	0.01
HALF_THREE	0.02	0.02
OPEN_THREE	0.08	0.2
HALF_FOUR	0.15	0.3
OPEN_FOUR	0.6	0.3
BLOCK_THREE	0.2	0.2
BLOCK_FOUR	0.8	0.9
WIN_REWARD	1.0	10.0
LOSS_REWARD	-2.0	-2.0
DRAW_REWARD	0.1	0.1
STEP_PENALTY	0.0	0.0

Table 2: Rewards for second training iteration

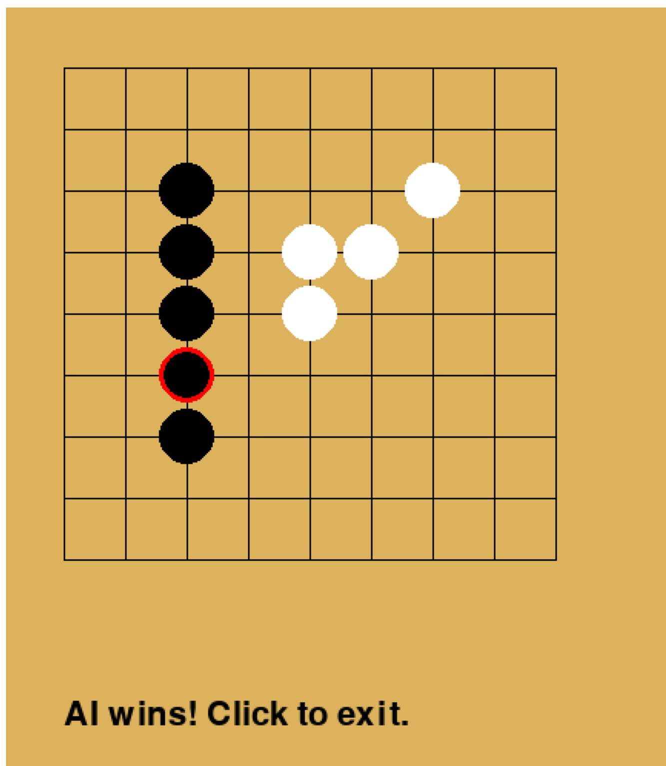
This agent saw significant improvement in certain areas and minimal improvement in others. Firstly, if not stopped, the agent will win in the first 5 moves, Figures 8a & 8c. However, both of these wins were when it could place the fifth tile not on the edge, it would not win in Figure 8b even though it had 3 different ways of getting 5 in a row, notably it also did not attempt to block my 5 in a rows. Overall, this agent performed better than the first but still needed significant improvement to beat a human. A downside in training from raising the win reward too high was made apparent in Figure 9b where the moves plateaued at the a minimal value, indicating that the agent was just placing 5 or 6 tiles each and then the game was over, leading to a situation where the agent only developed that strategy of win in 5 moves and could not adapt when that was easily blocked by a real player or a better agent. Even with this suboptimal strategy, the loss, Figure 9b (red), does decrease, although that may be because it is following this strategy it is intent is a good idea when it is not.



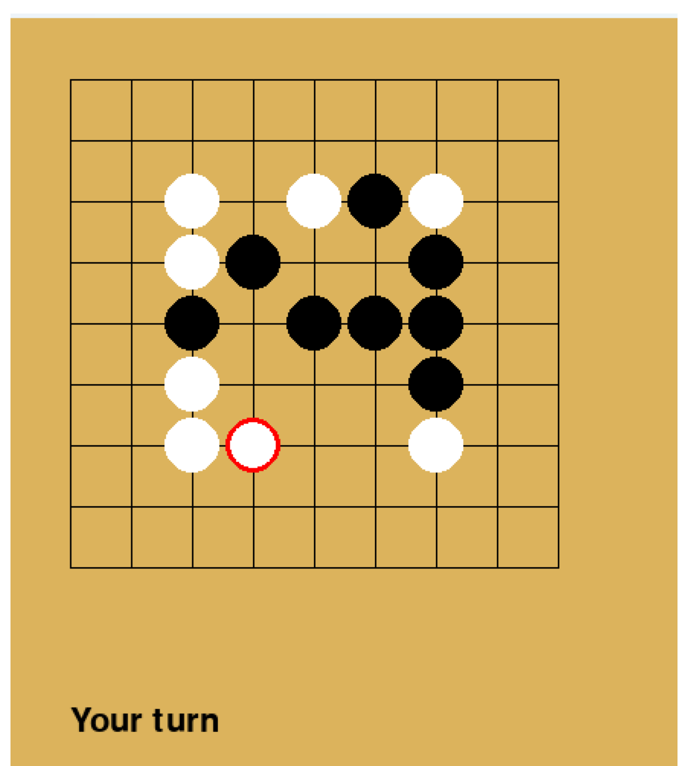
(a) AI Black



(b) AI white

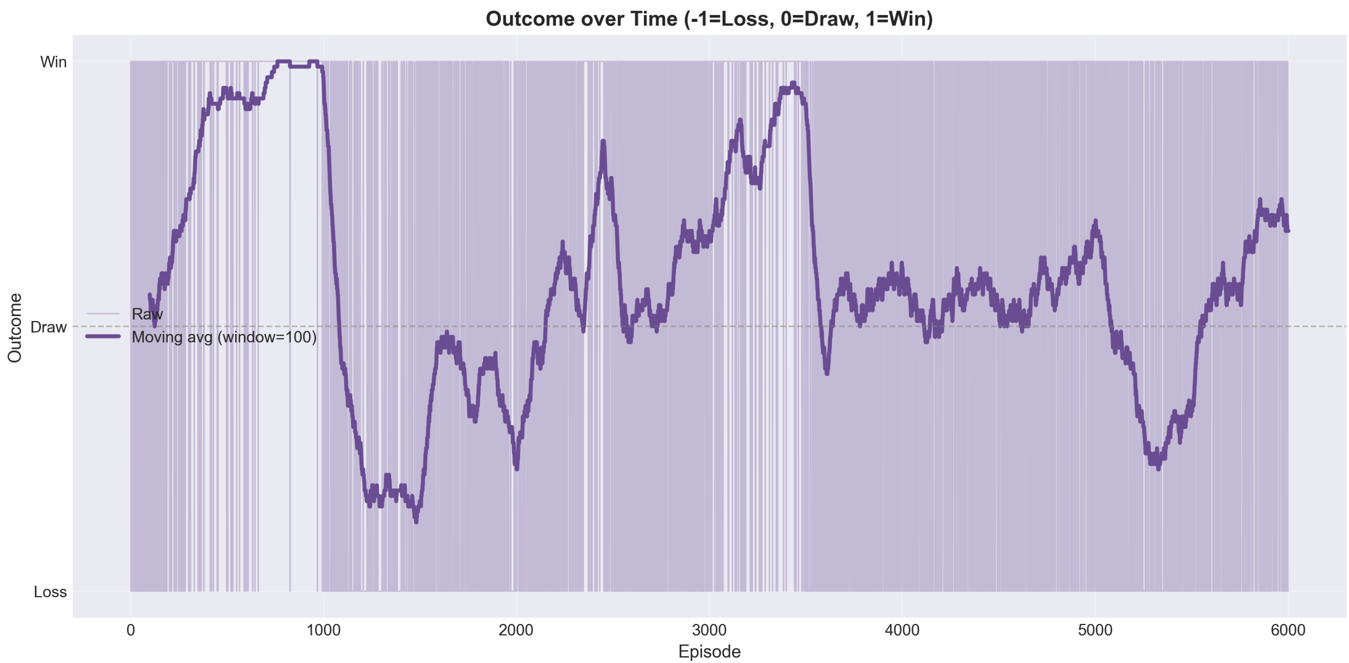


(c) AI Black

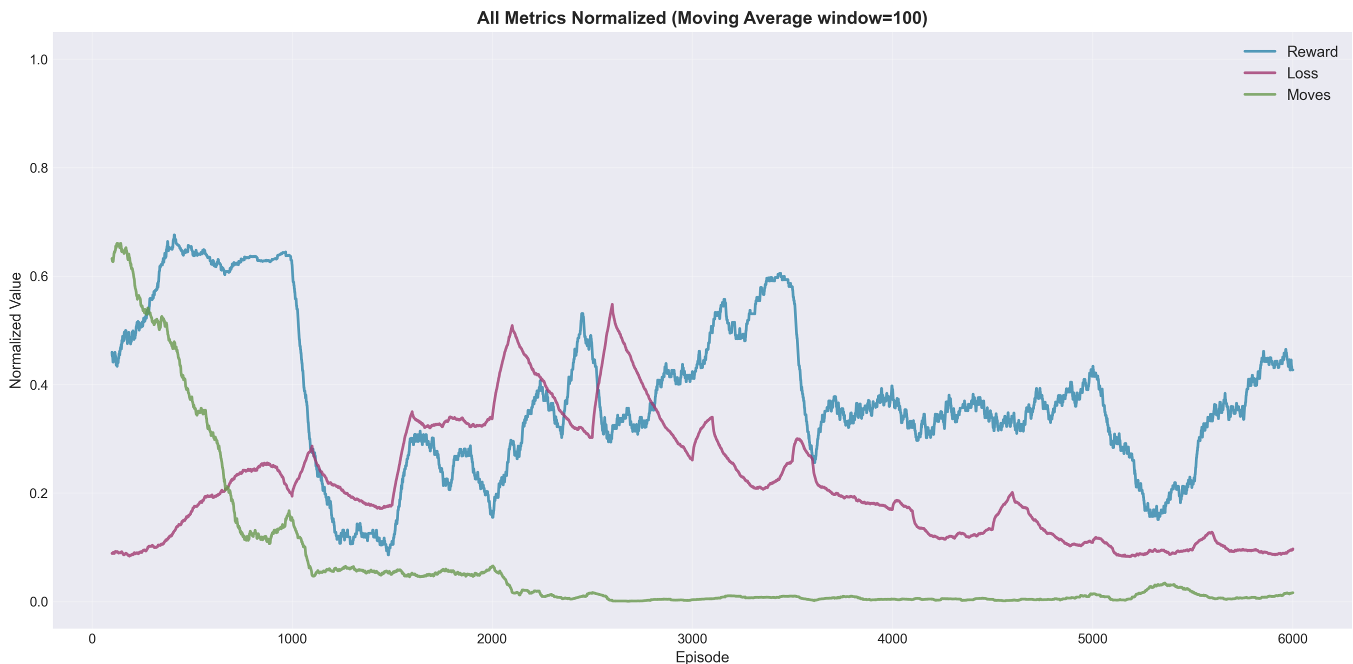


(d) AI White

Figure 8: Figures showing certain interesting moves the DQN agent has taken. This was the second iteration of the DQN, with rewards structured in Table 2.



(a) Figure showing the rolling average outcome of the agent during its training. A win is 1, a loss is -1.



(b) Figure showing the rolling average normalised metrics for reward (blue), loss (red) and moves (green) over its training

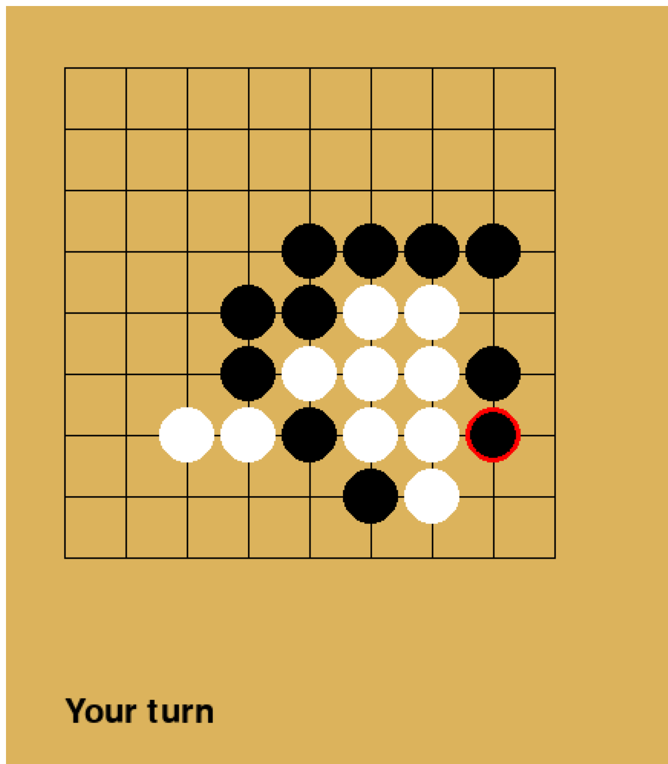
Figure 9: Figures showing the rolling average outcome (left) and the rolling average normalised metrics of reward, loss and moves (right). Both are for the second DQN iteration

To try and address the problems from the second iteration, we decided to try some more foundational tricks. We made all the rewards to be between -1 and 1, this is to try and ensure a better training of the Convolutional Neural Networks in the DQN agent. With a more stably trained model, hopefully the agent gets better at all points in the game. We also increased the rewards for three and four in a row to try and force the agent to make more actions that create chains. The reward for a draw was increased from 0.1 to 0.2 to allow the agent to think a draw was more acceptable and take the games longer.

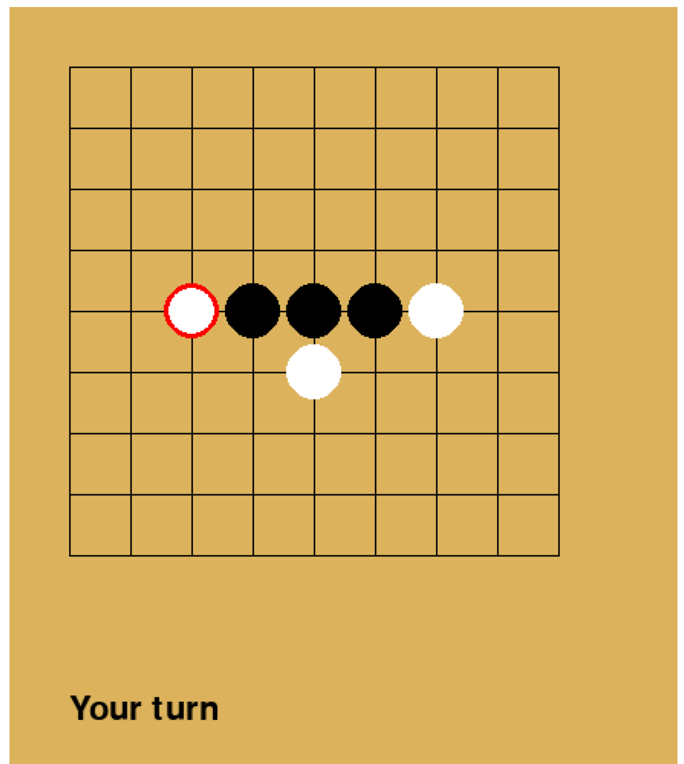
Variable	Previous Reward	New reward
OPEN_TWO	0.01	0.01
HALF_THREE	0.02	0.02
OPEN_THREE	0.2	0.4
HALF_FOUR	0.3	0.7
OPEN_FOUR	0.3	0.7
BLOCK_THREE	0.2	0.62
BLOCK_FOUR	0.9	0.9
WIN_REWARD	10.0	1.0
LOSS_REWARD	-2.0	-1.0
DRAW_REWARD	0.1	0.2
STEP_PENALTY	0.0	0.0

Table 3: Rewards for third training iteration

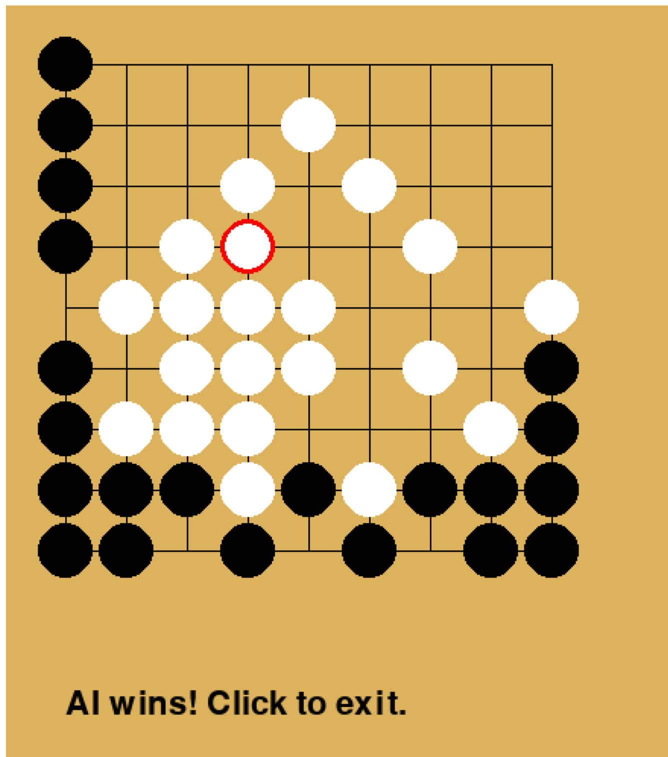
The progress from the second agent is difficult to quantify. This new agent is significantly more aggressive at shutting down 3 in a row from opponents, Figure 10b, and it successfully blocks 5 in a row when the 5th piece would not be on the edge. However, it is still really bad at blocking 5 in a row where the 5th tile is placed on the end, Figures 10a & 10d. In Figure 10c we can see that the agent is not just placing 5 in a row at the start like the previous iteration, it is trying to control the centre of the board and then eventually wins with 5 in a row, coincidentally avoiding winning with 5 in a row that was placed on the end. The outcome, Figure 11a followed the same pattern that the previous two DQN agents followed when training, converging on a 50% win rate in self play, what we would expect when it is playing itself and reaches the best strategy it can. The number of moves, Figure 11b (green), increasing as training goes on is presumably a positive sign that the agent is getting better at defending and overall strategy and not just stuck on one strategy like in the previous iteration. However, more moves could be a result of the agent being unable to win and struggling to finish the game off, after playing against the agent we think it is a combination of both reasons as the agent can defend but it does struggle to place the fifth tile in a row.



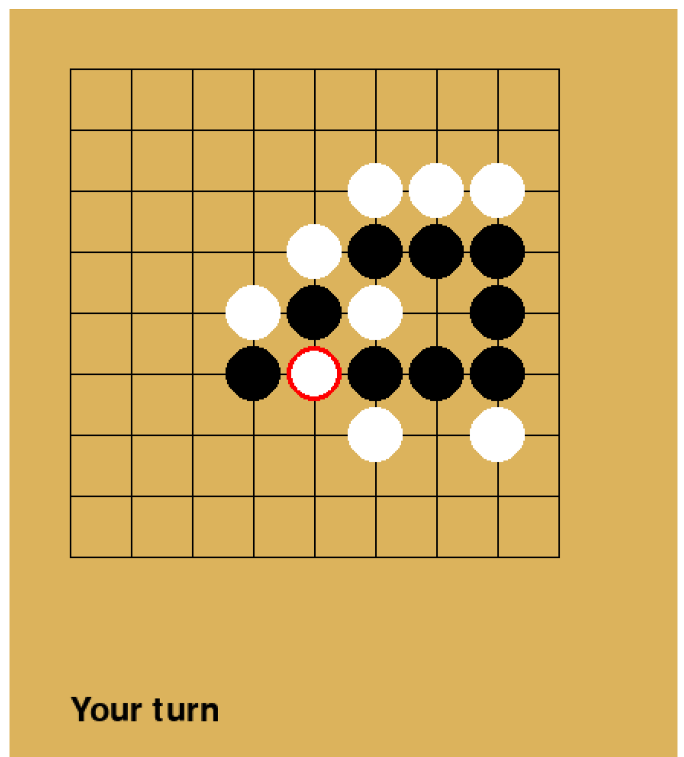
(a) AI Black



(b) AI white

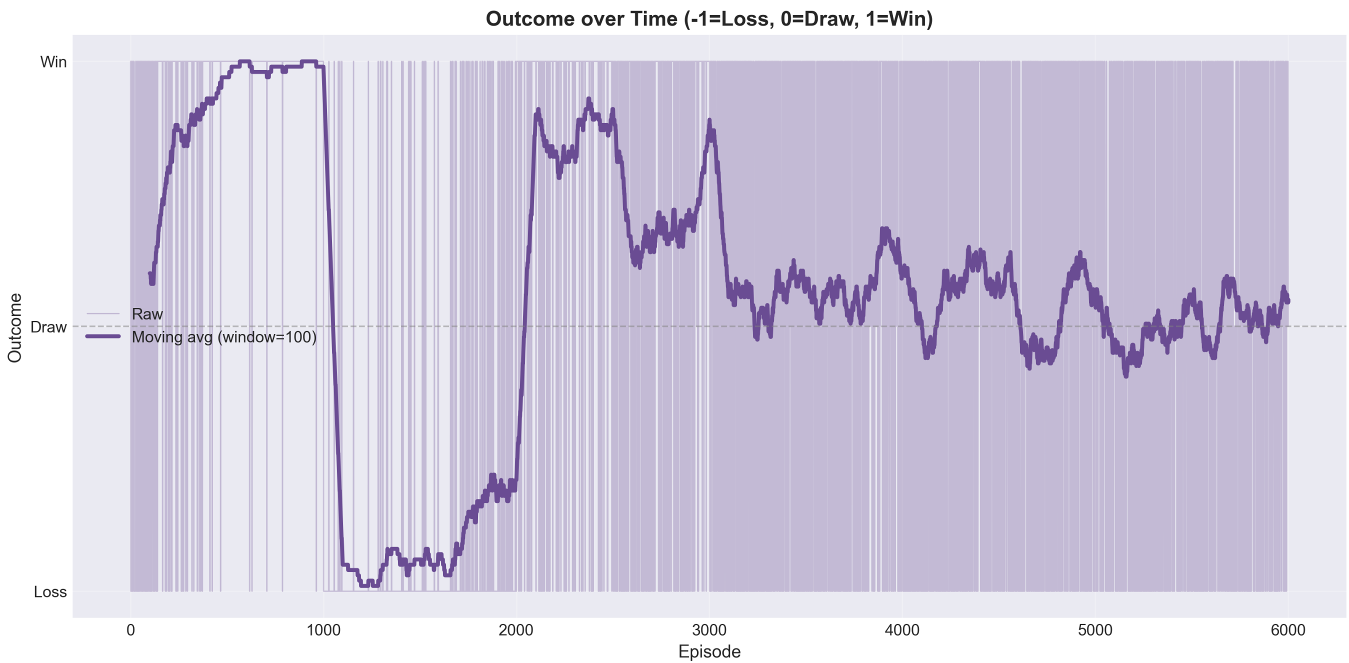


(c) AI White

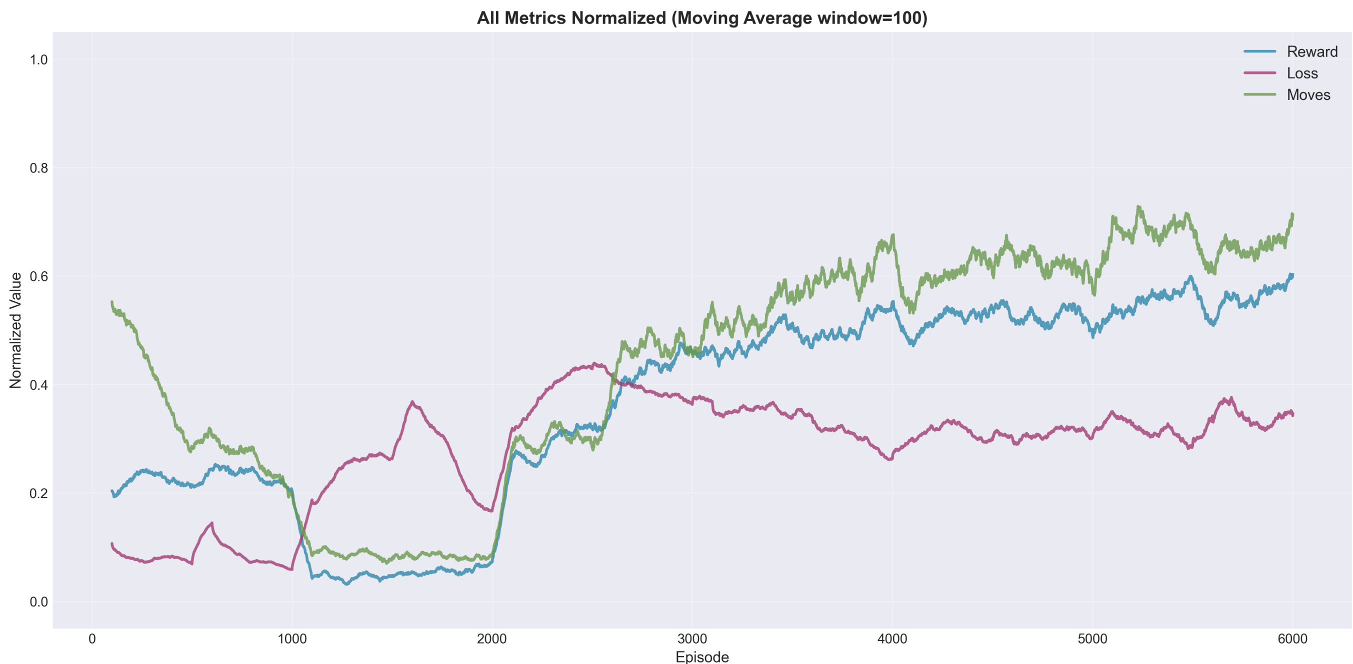


(d) AI White

Figure 10: Figures showing certain interesting moves the DQN agent has taken. This was the third iteration of the DQN, with rewards structured in Table 3.



(a) Figure showing the rolling average outcome of the agent during its training. A win is 1, a loss is -1.



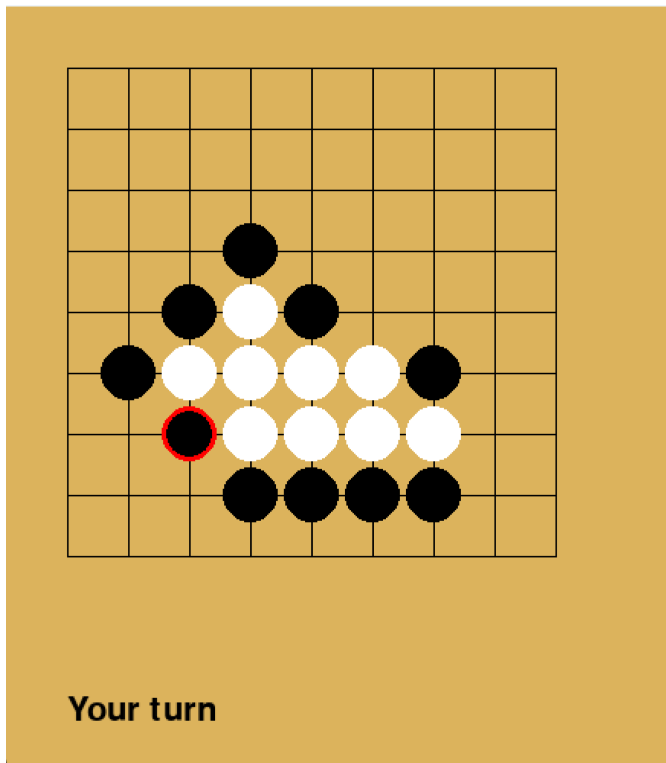
(b) Figure showing the rolling average normalised metrics for reward (blue), loss (red) and moves (green) over its training

Figure 11: Figures showing the rolling average outcome (left) and the rolling average normalised metrics of reward, loss and moves (right). Both are for the third DQN iteration

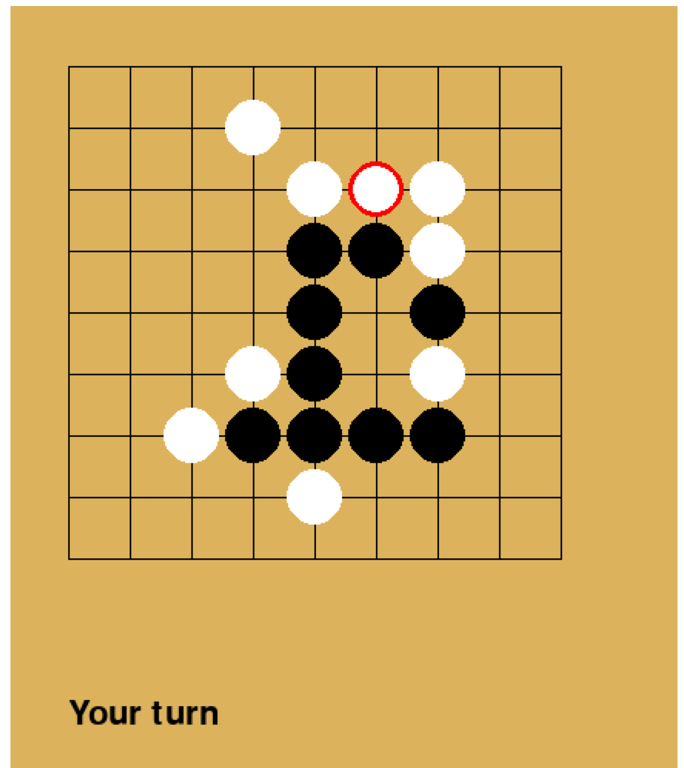
For our fourth iteration of the DQN agent, we decided to keep the rewards the same as we were not unhappy with the strategy. Instead, we increased the number of games it trains off, we kept the number of games against random and heuristic bots the the same, but we doubled the number of games against itself, going from 4000 to 8000. This was in an attempt to get the agent to develop more nuanced strategy and by playing more games should develop a better baseline strategy to play with.

When playing against this agent, we see the same strategies that the previous iteration employed

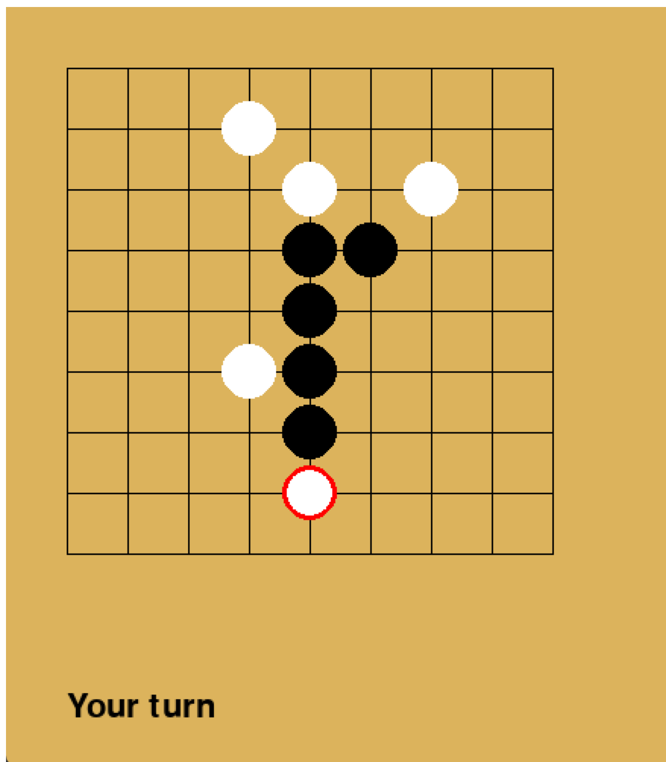
but with more focus on those same strategies. A notable improvement is that the AI is much better at blocking straight 5 in a row, Figures 12b & 12c, but then could not block the 5 in a row diagonal in Figure 12d. A very interesting move was in Figure 12a when instead of winning, it blocked me winning, demonstrating a clear preference to defend over attacking, a stark difference from the second agent. We think this difference in behaviour comes from the reward for winning not being too much higher than the defensive rewards, and then in training being exposed to blocking more than winning it has built up a false idea that blocking is more valuable than winning. The graphs, Figure 13, demonstrate the same trends as the previous iteration, although loss does start to increase by the end of the training, potentially a result of training too long and over-fitting to past data. Overall, the agent did perform marginally better against a human but will not achieve superhuman abilities from this reward structure. A major weakness is still the diagonals, both when attacking and defending, we are not sure if this is caused by the kernel in the CNN being too small or a small bug in the reward system, but from these errors we decided to test a new algorithm and see if it could perform better than the DQN agent at Gomoku



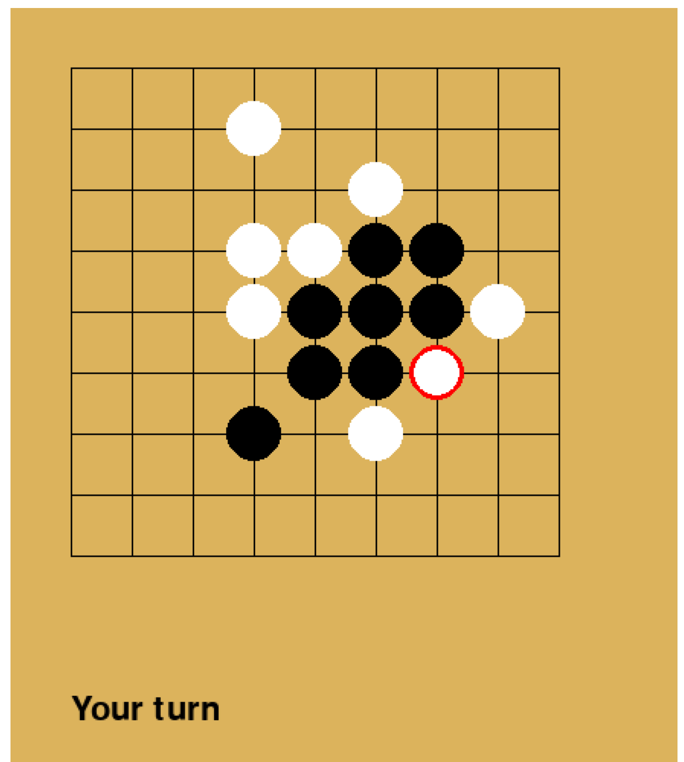
(a) AI Black



(b) AI white

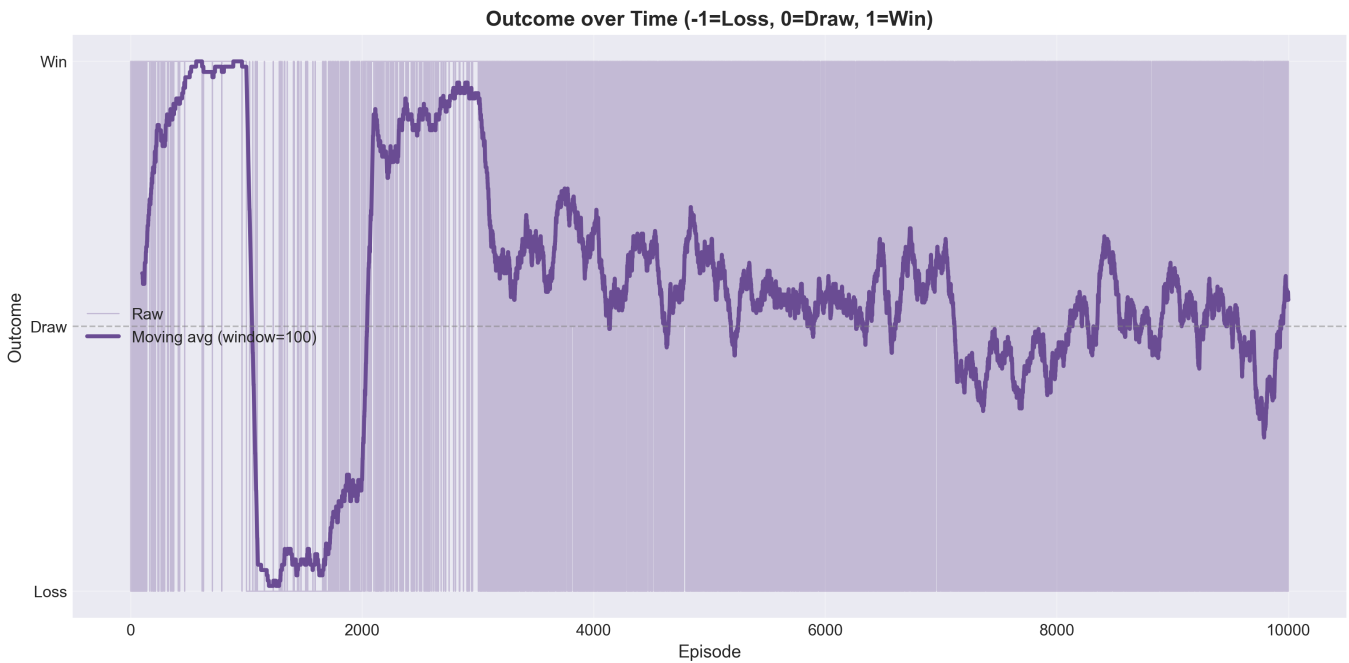


(c) AI White

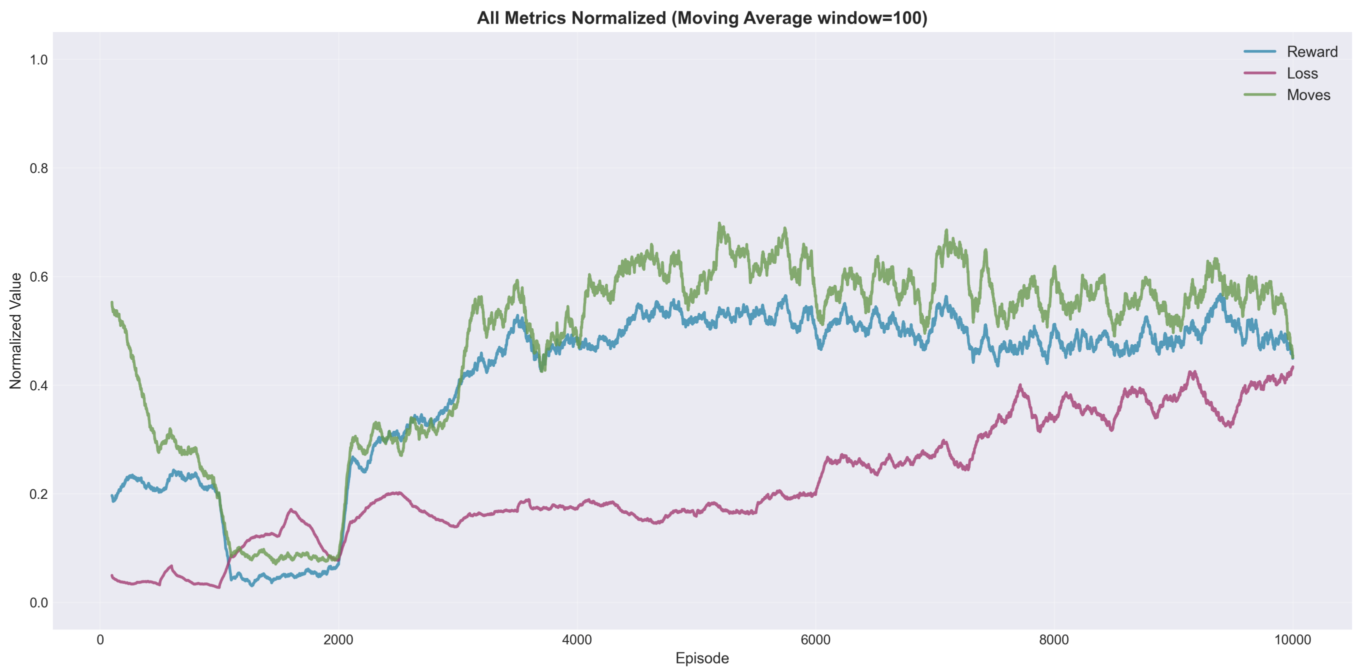


(d) AI White

Figure 12: Figures showing certain interesting moves the DQN agent has taken. This was the fourth iteration of the DQN, with rewards structured in Table 3.



(a) Figure showing the rolling average outcome of the agent during its training. A win is 1, a loss is -1.



(b) Figure showing the rolling average normalised metrics for reward (blue), loss (red) and moves (green) over its training

Figure 13: Figures showing the rolling average outcome (left) and the rolling average normalised metrics of reward, loss and moves (right). Both are for the fourth DQN iteration

4 Approach 2: AlphaZero

4.1 Algorithm Overview

AlphaZero [3] is a model-based reinforcement learning algorithm that learns entirely from self-play, with no human knowledge beyond the rules of the game. It couples a deep residual neural network f_θ with Monte-Carlo Tree Search (MCTS) in a closed loop: the network guides the tree search by providing move probabilities and position evaluations, while the search results serve as improved training targets for the network.

At each position s, the network outputs a policy vector $\mathbf{p}$ over legal moves and a scalar value $v \in [-1, 1]$ estimating the expected outcome:

$$(\mathbf{p}, v) = f_\theta(s) \tag{3}$$

During self-play, MCTS uses the network's outputs to build a search tree. Each simulation traverses the tree by selecting actions that maximise an upper confidence bound:

$$a_t = \arg\max_a \left[Q(s,a) + c_{\text{puct}} \cdot P(s,a) \cdot \frac{\sqrt{\sum_b N(s,b)}}{1 + N(s,a)} \right] \tag{4}$$

where $Q(s,a)$ is the mean action-value, $P(s,a)$ is the prior from the network, $N(s,a)$ is the visit count, and $c_{\text{puct}} = 1.5$ controls the exploration–exploitation trade-off. When a leaf node is reached, the network evaluates it and the value is backpropagated up the tree, with the sign negated at each level to account for alternating players.

After all simulations complete, the visit count distribution $\boldsymbol{\pi}$ at the root serves as an improved policy target. The network is then trained to minimise:

$$\mathcal{L} = (z - v)^2 - \boldsymbol{\pi}^\top \log \mathbf{p} + c\|\theta\|^2 \tag{5}$$

where $z \in \{-1, 0, +1\}$ is the game outcome from the perspective of the player at that position and $c = 10^{-4}$ is the L2 regularisation coefficient.

Unlike value-based methods such as DQN, AlphaZero does not use reward shaping or discount factors. The only reward signal is the terminal game outcome, and the training target for the value head is this undiscounted result. The policy improves through the MCTS "policy improvement" step: the search produces better move distributions than the raw network, and the network learns to match these improved distributions.

4.2 Adaptation for 9×9 Gomoku

The original AlphaZero was trained on 5,000 TPUs across chess, shogi, and Go. We adapted the algorithm for 9×9 Gomoku on a single NVIDIA RTX 4090. Table 4 summarises the key changes.

Table 4: Adaptations from the AlphaZero paper for single-GPU 9×9 Gomoku.

Paper (Chess/Go)	**Our Setting**	**Reason**
5,000 TPUs	1× RTX 4090	Hardware constraint
20 res. blocks, 256 filters	10 res. blocks, 128 filters	Sufficient for 9×9
19×19×17 input	9×9×3 input	No ko/repetition rules
800 MCTS simulations	400–800 (tuned)	Smaller game tree
Batch size 4,096	Batch size 256	Single GPU memory
LR = 0.2	LR = 0.05	Scaled for smaller batch

The input representation uses three 9×9 binary planes: current player's stones, opponent's stones, and a colour plane (all ones if Black to move, all zeros if White). Unlike Go or chess, Gomoku has no ko or repetition rules, so history planes are unnecessary. Gomoku also possesses 8-fold dihedral symmetry (4 rotations $\times$ 2 reflections), which we exploit for data augmentation, effectively multiplying the training data by $8\times$ at no additional self-play cost.

The network architecture consists of an input convolution (3$\rightarrow$128 filters, 3$\times$3, BatchNorm, ReLU), followed by 10 residual blocks (each containing two 3$\times$3 convolutions with BatchNorm and a skip connection), a policy head (1$\times$1 conv $\rightarrow$ 2 channels $\rightarrow$ FC $\rightarrow$ 81 outputs), and a value head (1$\times$1 conv $\rightarrow$ 1 channel $\rightarrow$ FC(256) $\rightarrow$ tanh). The total parameter count is 2,992,794. We use SGD with momentum 0.9, weight decay 10^{-4}, and mixed-precision training (FP16 via PyTorch AMP).

4.3 Batched MCTS for GPU Efficiency

A naïve MCTS implementation performs one neural network forward pass per simulation per game, resulting in hundreds of thousands of sequential GPU calls per training iteration. On an RTX 4090, the Python$\rightarrow$GPU round-trip latency dominates wall-clock time.

We implemented **batched parallel MCTS**, where all N self-play games run in lockstep. Each simulation step proceeds as follows:

1. **Select**: traverse each of the N trees from root to leaf.
2. **Batch**: collect all N leaf states into a single tensor.
3. **Evaluate**: one batched GPU forward pass for all N states.
4. **Expand & Backpropagate**: distribute results back to each tree.

This reduces the number of GPU calls from $N \times S$ (where S is the number of simulations) to just S calls of batch size N, yielding a measured **4$\times$ speedup** on the RTX 4090. Games that finish before others are removed from the active batch, and their training examples are stored immediately.

4.4 The Self-Play Distribution Gap

After an initial overnight training run (400 simulations, 200 iterations, 6.5 hours), the agent achieved a loss of 3.17 and a 100% win rate against a random opponent. However, when tested against a human player, it failed to block a trivial vertical column attack—even with 1,600 MCTS simulations at play time.

The root cause is a **training distribution gap** inherent to pure self-play at limited compute. During training, both players use the network's own policy. Since the early network does not play simple linear attacks, neither side produces them. The network never encounters positions such as "opponent has 3 stones in a column on a sparse board" because such positions never arise in self-play. The network learns to play well *against itself* but not against strategies outside its self-play distribution.

At the scale of the original paper (millions of games on thousands of TPUs), this gap closes naturally through sheer volume of exploration. On a single GPU, we addressed it through a targeted intervention described in the following section.

4.5 Tactical Threat Detection in Training MCTS

To bridge the distribution gap, we integrated a rule-based threat detector into the MCTS *during training only*. This detector overrides the network's prior at nodes where an immediate tactical

response is required, producing sharper policy targets that teach the network to handle threats it would otherwise never encounter.

The detector operates on a three-tier priority system:

1. **Immediate win** (always active): if the current player can complete 5-in-a-row, play the winning move.
2. **Immediate block** (always active): if the opponent can complete 5-in-a-row on the next move, block it.
3. **Open three detection** (active 50% of the time): if the opponent has an open three pattern, block it before it becomes an unblockable open four.

Three open three patterns are detected using a sliding 6-cell window across all four directions:

- Consecutive: `.OOO.` — three in a row with both ends empty
- Broken A: `.O.OO.` — gap between the first and second stone
- Broken B: `.OO.O.` — gap between the second and third stone

When a threat is detected, 90% of the MCTS prior probability is placed on the forced move, with 10% spread across remaining legal moves. This preserves some exploration while ensuring the search overwhelmingly visits the tactically critical response. For consecutive open threes, a centrality heuristic (Manhattan distance to the board centre) selects the more strategically valuable of the two blocking ends.

Crucially, the threat detector is **not used at play time**. The goal is for the network to internalise the blocking patterns through training so that it produces correct responses from its own policy.

4.5.1 The 50% Randomisation

An initial version of the threat detector fired on all three tiers at 100% probability. This caused every self-play game to end in a draw after 81 moves (the entire board filled), because both sides blocked every threat perfectly via the heuristic. The resulting training data contained only dense full-board positions, and the network never learned to block threats on sparse, early-game boards—precisely the positions it faced against human opponents.

The solution was to fire the open three detection (tier 3 only) with 50% probability at each node expansion. Tiers 1 and 2 (immediate win/block) always fire because those moves are unconditionally forced. This produces a mixture of training data:

- When the check fires: threats are blocked early, producing defensive positions and drawn games.
- When the check is skipped: threats develop on sparse boards, one side fails to block, and the game produces decisive outcomes with exactly the sparse-board blocking positions the network needs.

This 50% randomisation was the single most important design decision in the project. Without it, the network achieved a policy loss of 2.45 but could not block a straight line at play time. With it, the first successful model blocked column attacks and won against a human player.

4.6 Training and Hyperparameter Analysis

4.6.1 Learning Rate

The paper uses a learning rate of 0.2 with SGD, but this was calibrated for a batch size of 4,096. With our batch size of 256, 0.2 caused the loss to flatline at 5.39 (the network failed to learn). A learning rate of 0.01 learned but was too slow to converge within a practical number of iterations. The geometric mean, **0.05**, worked immediately and was used for all subsequent runs. The learning rate is dropped by a factor of 10 at 50% and 75% of total iterations, following the paper's schedule.

4.6.2 Replay Buffer Sizing

The replay buffer stores training examples as $(s, \boldsymbol{\pi}, z)$ tuples. Each self-play game produces approximately $m \times 8$ examples (where m is the number of moves and the factor of 8 comes from symmetry augmentation). The buffer should retain the last 50–75 iterations of data to provide a sufficiently diverse training distribution without diluting recent high-quality examples. For 30 games per iteration with an average of $\sim$55 moves, this corresponds to $\sim$13,200 examples per iteration, giving a target buffer size of $\sim$1,000,000.

4.6.3 Training Epochs per Iteration

Self-play is the computational bottleneck (CPU-bound MCTS tree traversal), while gradient updates on the GPU are fast. We found that increasing training epochs from 10 to 20 per iteration improved network learning at negligible wall-clock cost (seconds of additional GPU time per iteration versus minutes of self-play time).

4.6.4 MCTS Simulations

With the threat detection system providing sharp priors, 400 simulations per move during training proved sufficient. The threat check ensures the correct tactical move receives high visit counts even with relatively shallow search. At play time, we use 800 simulations for stronger performance, where the additional depth helps the network evaluate complex positional trade-offs.

4.7 Ablation: Comparing Two Models

We compare two successful training configurations to analyse the effect of increased compute. Both use the same architecture, threat detection system, and 50% open three randomisation.

Table 5: Configuration and final metrics for the two compared models.

	Model A (400/30/180)	**Model B** (800/40/200)
MCTS simulations	400	800
Games per iteration	30	40
Training iterations	180	200
Replay buffer size	1,000,000	1,500,000
Training time	13 hours	26 hours
Final total loss	2.48	1.22
Final policy loss	2.06	1.09
Final value loss	0.46	0.13
Centre move prob. (empty board)	0.941	0.248 (per adj. square)
Value estimate (empty board)	0.494	0.445
Win rate vs. random	100% (40 games)	100% (40 games)

Policy loss. Model B achieves a policy loss of 1.09, nearly half that of Model A (2.06). This indicates significantly sharper policy predictions—the network concentrates its probability mass more tightly on the moves selected by MCTS. At play time, this manifests as visit distributions of 700–800 out of 800 simulations on the top move, compared to more diffuse distributions in Model A.

Value loss. Model B's value loss (0.13) is roughly a quarter of Model A's (0.46), indicating substantially more accurate position evaluation. This improvement is driven by the deeper MCTS (800 vs. 400 simulations), which produces more informative value targets during training. When MCTS searches deeper, it more frequently encounters terminal states, providing direct win/loss signals rather than relying solely on the network's own value estimates.

Opening policy. An interesting difference appears in the empty-board policy. Model A places 94.1% probability on the centre square (4,4), while Model B distributes probability roughly equally among the four centre-adjacent squares ($\sim$24% each on (3,4), (4,3), (4,5), (5,4)). The latter is arguably more sophisticated—it reflects the understanding that the exact centre square and its immediate neighbours are strategically equivalent in the opening.

Play-test results. Both models achieve 100% win rate against a random opponent. Against a human amateur player, both models successfully block vertical and diagonal column attacks. Model B demonstrates stronger counter-attacking ability: it consistently builds its own winning threats while defending, rather than purely reacting to the opponent's moves.

Figure 14: Model A (400/30/180): Policy loss over training iterations.

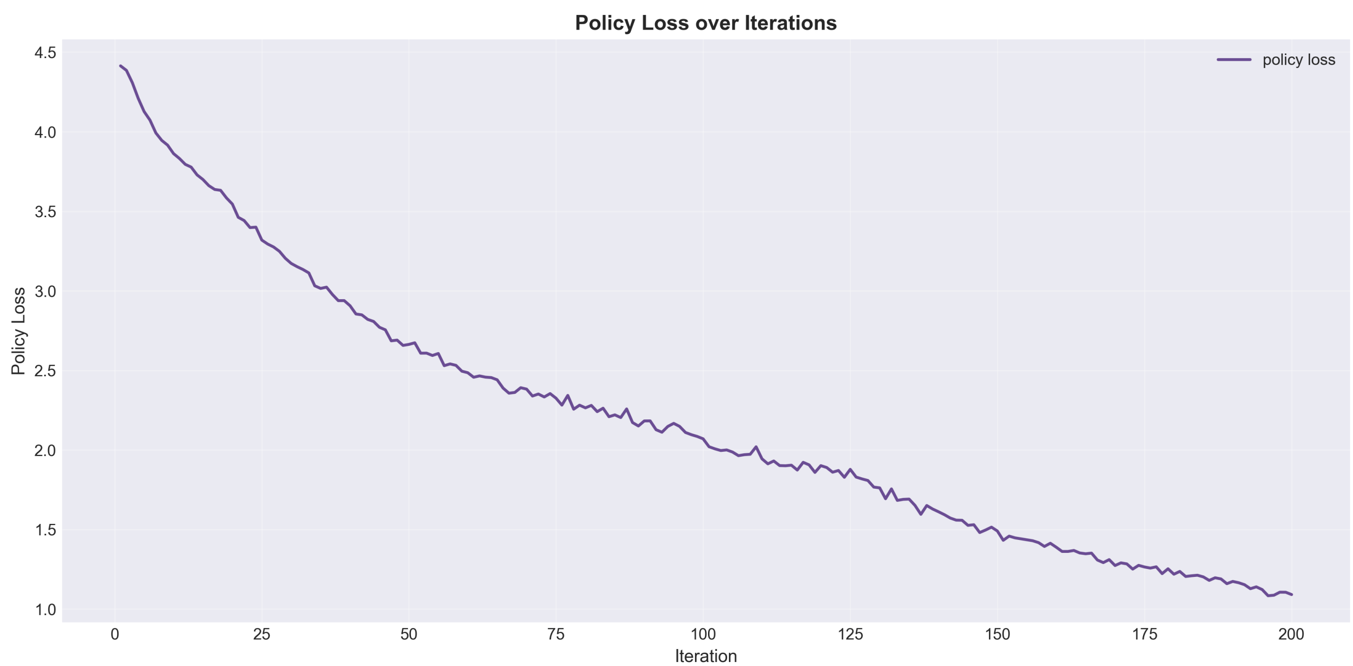


Figure 15: Model B (800/40/200): Policy loss over training iterations. Both models show steady decrease with visible drops at LR reduction points. Model B reaches a substantially lower final value (1.09 vs. 2.06).

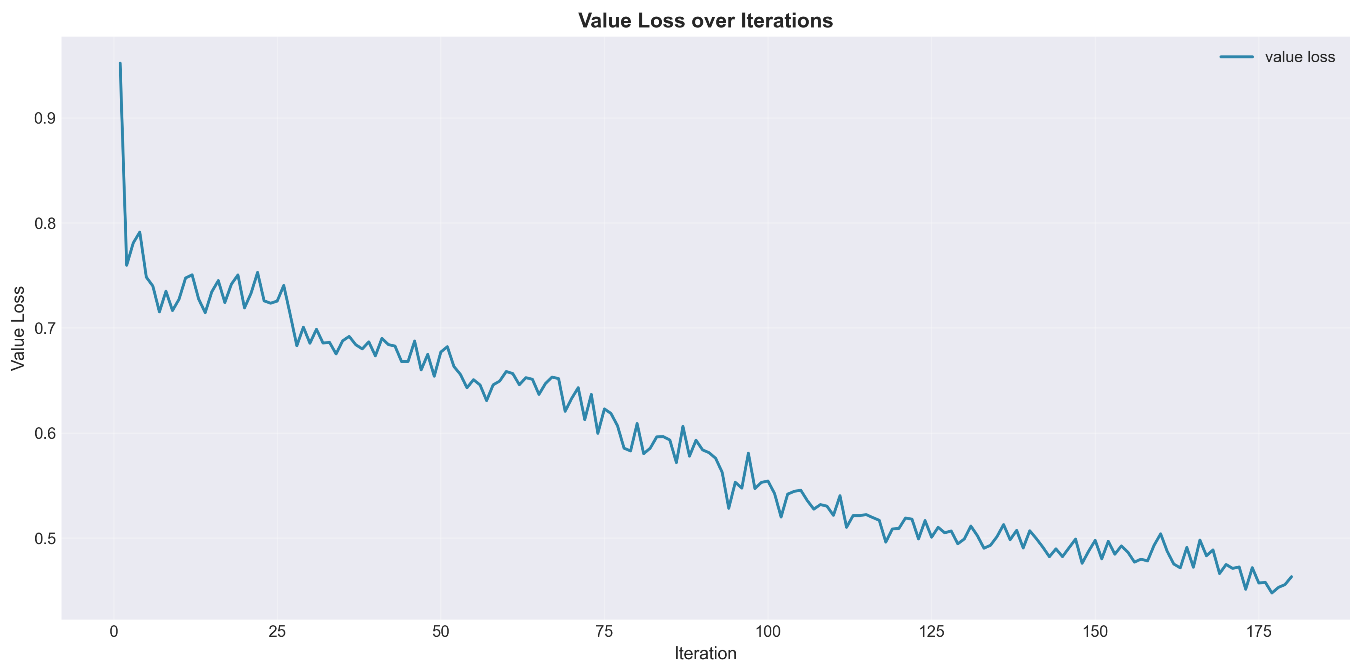


Figure 16: Model A (400/30/180): Value loss over training iterations.

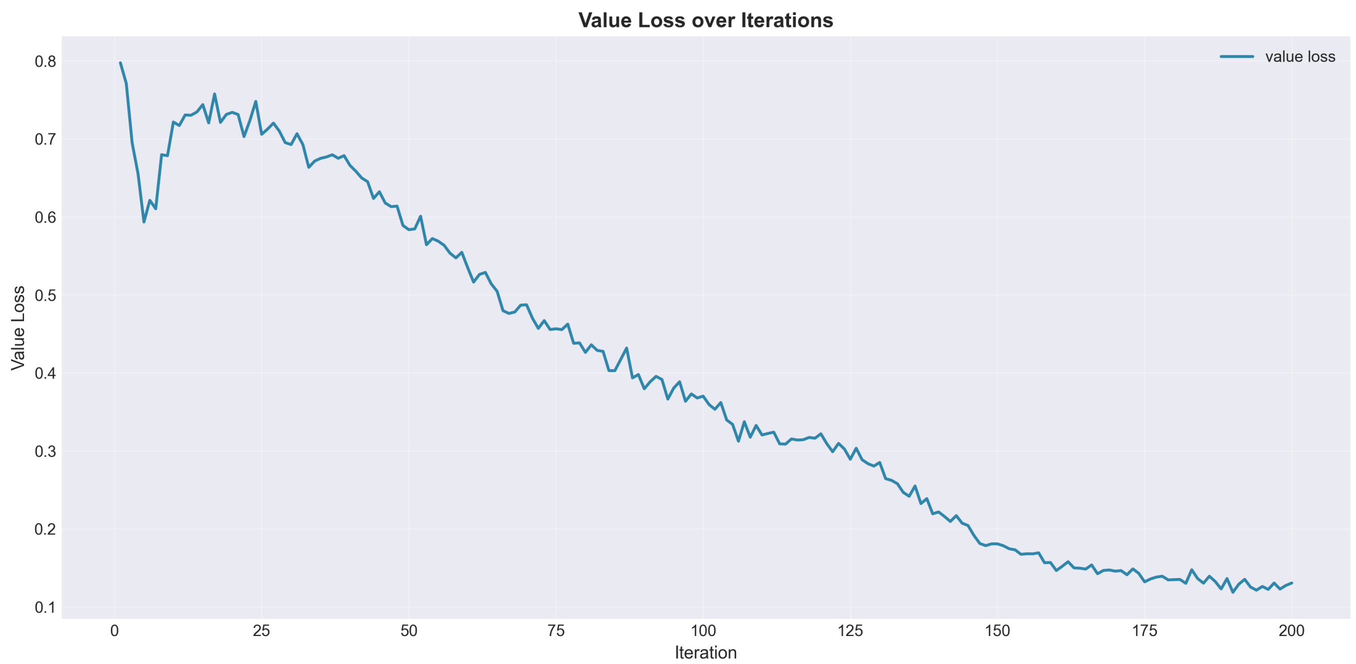


Figure 17: Model B (800/40/200): Value loss over training iterations. Model B's deeper MCTS produces more informative value targets, resulting in a significantly lower final value loss (0.13 vs. 0.46).

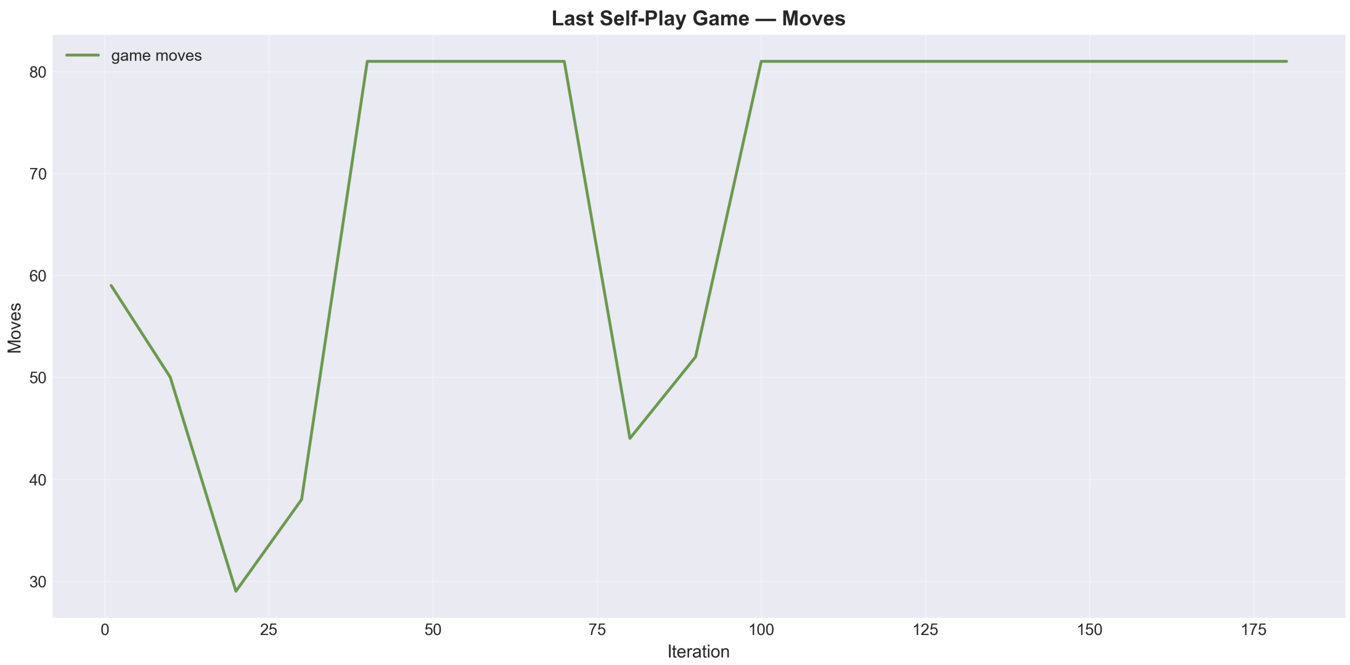


Figure 18: Model A (400/30/180): Self-play game length over training.

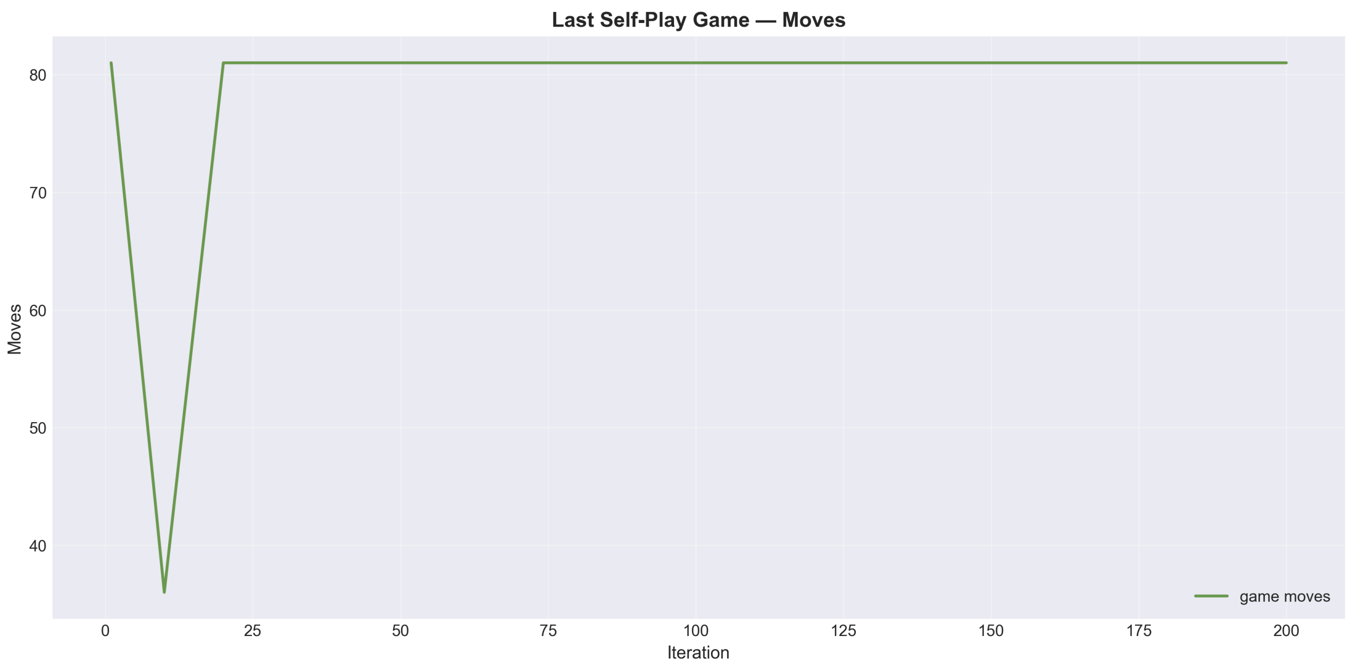


Figure 19: Model B (800/40/200): Self-play game length over training. Both models show a mix of short decisive games and full-board draws, reflecting the 50% threat check randomisation.

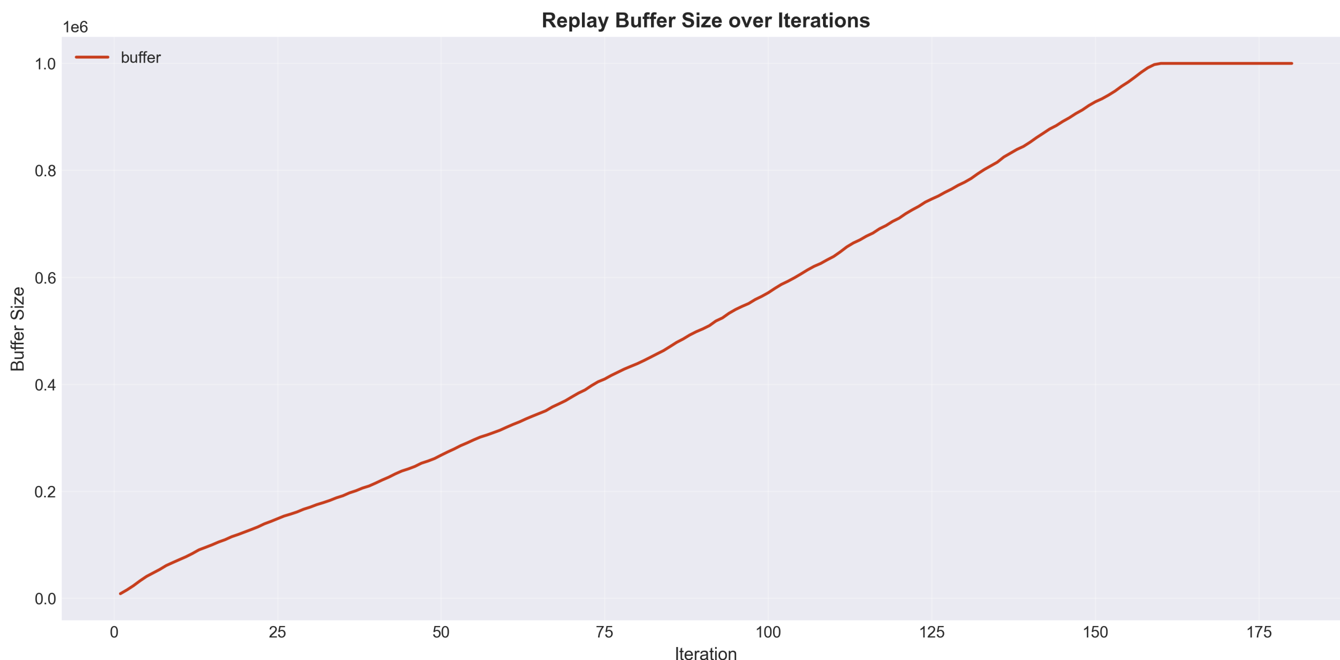


Figure 20: Model A (400/30/180): Replay-buffer capacity over training. Saturates around iteration 155.

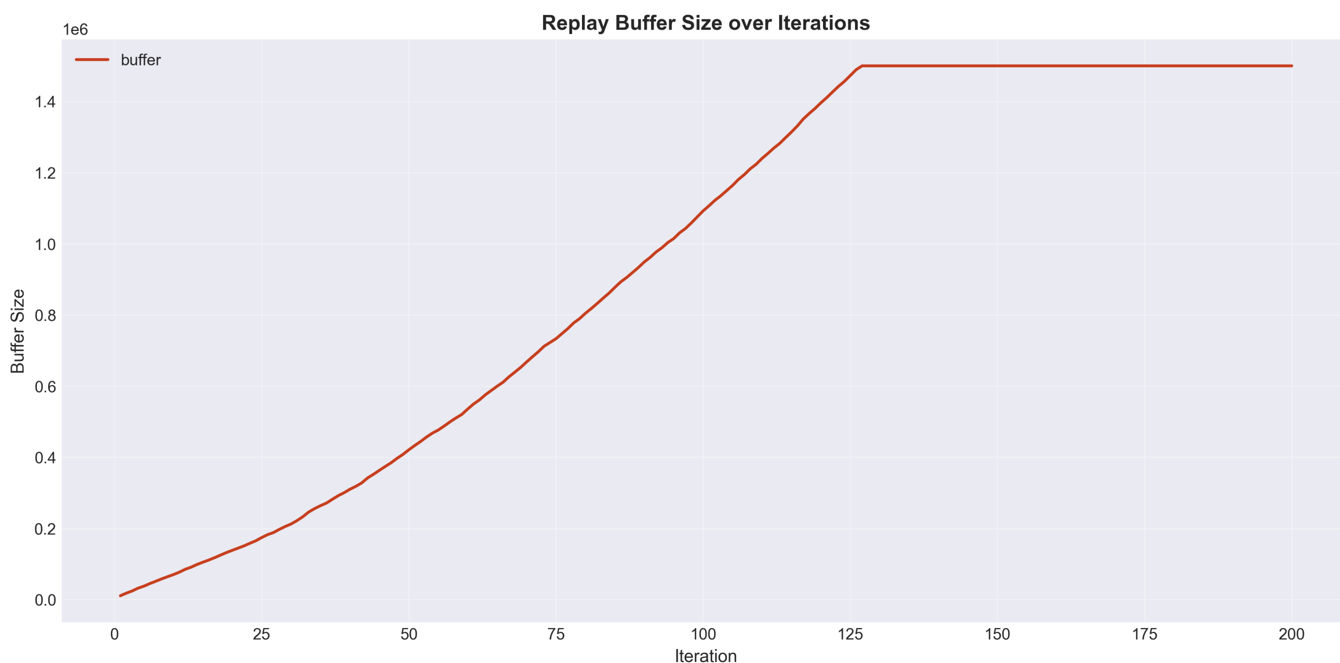


Figure 21: Model B (800/40/200): Replay-buffer capacity over training. Saturates earlier than Model A even with a higher capacity, at around iteration 125.

4.8 Limitations

The agent was trained on a single consumer GPU in 13–26 hours, compared to the thousands of TPU-hours used in the original paper. Despite this ∼1,000× reduction in compute, it blocks standard tactical threats and defeats amateur human players. It can still be beaten by strong players who construct complex multi-threat forks, and the threat detection during training is limited to three specific open three patterns; more advanced tactical motifs are left for the network to learn implicitly.

5 Benchmarking: Round Robin

To compare all trained agents on equal footing, we built a standalone evaluation harness with a neutral game engine independent of either training codebase. The neutral engine uses the same 9×9 board and five-in-a-row win condition, but carries no dependency on the DQN or AlphaZero repositories. Each agent is wrapped behind a common interface that accepts a board state and returns a move, with internal translation handling the differences in board encoding between the two approaches (the DQN repo uses 0/1/2 for empty/black/white, while the AlphaZero repo uses $0/1/-1$).

Five agents were entered into the tournament:

DQN-A (third reward iteration) was trained with a curriculum of 1,000 random games, 1,000 heuristic games, and 4,000 self-play games (checkpoint interval of 500, producing 8 self-play checkpoints). Its reward structure normalised all shaping signals to $[-1, 1]$, with offensive rewards of 0.4 for open threes and 0.7 for open/half fours, and defensive rewards of 0.62 for blocking threes and 0.9 for blocking fours.

DQN-B (fourth reward iteration) used the same reward values as DQN-A but doubled the checkpoint interval to 1,000, giving 8,000 self-play episodes (10,001 total). The longer self-play phase with fewer, more spaced-out opponent pool updates was intended to produce a more stable late-stage policy by giving the agent more games against each frozen opponent before moving on.

Both DQN agents share the same architecture (4 residual blocks, 64 channels, Double DQN with action masking) and the same core hyperparameters (ϵ-decay 0.995, learning rate 10^{-4}, target update every 2,000 steps, buffer capacity 50,000). At evaluation time, both play greedily ($\epsilon = 0$) with a single forward pass per move.

AlphaZero Model A (400/30/180) was trained for 13 hours with 400 MCTS simulations per move, 30 self-play games per iteration, and 180 training iterations, using a replay buffer of 1,000,000 examples. **AlphaZero Model B** (800/40/200) doubled the simulation budget to 800, increased games per iteration to 40, extended training to 200 iterations with a 1,500,000 replay buffer, and trained for 26 hours. Both use the same 10-block, 128-filter residual network (2.99M parameters), the same threat detection system with 50% open-three randomisation, and SGD with learning rate 0.05. At evaluation time, both use 800 MCTS simulations per move with greedy (argmax) move selection and no Dirichlet noise.

The fifth agent, **Heuristic**, is a rule-based baseline that checks for immediate wins, blocks opponent four-in-a-row, blocks open threes, and otherwise plays near the board centre with light randomisation among the top candidates.

Each pair of agents played 10 games with strictly alternating colours (agent 1 plays Black in odd games, White in even games), giving 100 games across all 10 matchups.

5.1 Win Rate Matrix

Table 6 shows each agent's win rate against every other agent. A clear hierarchy emerges: AlphaZero Model B is undefeated, followed by Model A, then the Heuristic, and finally the two DQN agents.

Table 6: Win rate matrix where the agent on the ROW label is what the win rate measures, e.g. AZ-A on the row means that row is its win rates against the other four agents. Each cell is the row agent's win percentage over 10 games. (E.g. In the row for AZ-B, AZ-B has a 100% win rate against DQN-A, 100% win rate against DQN-B, and so on)

	DQN-A	**DQN-B**	**AZ-A**	**AZ-B**	**Heuristic**
DQN-A	—	50%	0%	0%	30%
DQN-B	50%	—	0%	0%	10%
AZ-A	100%	100%	—	0%	90%
AZ-B	100%	100%	50%	—	90%
Heuristic	70%	90%	0%	0%	—

Both AlphaZero models achieve a perfect 100% win rate against both DQN agents, winning every game in an average of just 13–17 moves. The DQN agents cannot win a single game against either AlphaZero model, regardless of colour. Against the Heuristic, both AlphaZero models win 9 out of 10 games with one draw, while the DQN agents lose the majority of their games to the same Heuristic. The DQN-A vs DQN-B matchup splits evenly at 5–5.

5.2 Per-Agent Summary

Table 7 aggregates each agent's performance across all 40 games played (10 against each of the other four agents).

Table 7: Per-agent aggregate results across all matchups (40 games each).

Agent	**Wins**	**Losses**	**Draws**	**Win%**	**Wins as Black**	**Wins as White**
AZ-B	34	0	6	85.0%	20	14
AZ-A	29	5	6	72.5%	15	14
Heuristic	16	22	2	40.0%	9	7
DQN-A	8	32	0	20.0%	8	0
DQN-B	6	34	0	15.0%	5	1

AZ-B finishes the tournament with zero losses and a win rate of 85%, with the remaining 15% being draws (all against AZ-A). AZ-A follows at 72.5%, its only losses coming from AZ-B. The most striking result is the colour breakdown for the DQN agents: DQN-A won 8 games as Black and 0 as White; DQN-B won 5 as Black and only 1 as White. This indicates that the DQN agents have learned to exploit the first-move advantage but have not learned to play defensively as the second player.

5.3 Game Length and Decisiveness

Table 8 shows the average game length per matchup.

Table 8: Average game length (moves) and result breakdown per matchup.

Matchup	**Avg Moves**	**Black Wins**	**White Wins**	**Draws**
DQN-A vs DQN-B	53.0	10	0	0
DQN-A vs AZ-A	16.5	5	5	0
DQN-A vs AZ-B	15.5	5	5	0
DQN-A vs Heuristic	45.4	8	2	0
DQN-B vs AZ-A	14.5	5	5	0
DQN-B vs AZ-B	13.5	5	5	0
DQN-B vs Heuristic	46.8	4	6	0
AZ-A vs AZ-B	56.0	5	0	5
AZ-A vs Heuristic	30.0	5	4	1
AZ-B vs Heuristic	29.6	5	4	1

The AlphaZero agents defeat DQN opponents in an average of 13–17 moves, suggesting that the DQN agents fail to block basic threats from the opening. By contrast, AZ-A vs AZ-B averages 56 moves per game—the longest matchup in the tournament—with half the games ending in draws. This reflects two strong defensive players engaging in deep positional play rather than early tactical blunders.

5.4 First-Player Advantage

Across all 100 games, Black (the first player) won 57%, White won 36%, and 7% were draws. This is consistent with the known first-player advantage in Gomoku. However, the strength of this advantage varies sharply by agent quality. In the DQN-A vs DQN-B matchup, Black won all 10 games—neither agent could overcome the first-move disadvantage. In the AlphaZero vs Heuristic matchups, the AlphaZero agents won 4 out of 5 games as White, demonstrating that a sufficiently strong agent can overcome the first-player disadvantage. In AZ-A vs AZ-B, all 5 decisive games were won by Black (AZ-B), with the remaining 5 drawn, suggesting that at the highest level of play in our tournament, the first-move advantage becomes more pronounced.

A Team Contributions

Name	Contributions
Kishan	• AlphaZero adaptations for Gomoku • Developed game environment
Helitha	• Initial DQN/DDQN implementations and training • Tuned DQN parameters to refine models
Xavier	• Reward tuning of DQN • Produced the video • Created scripts for training analysis
Yash	• Training AlphaZero models via UCL's remote workstation service • Report writing and formatting lead

Table 9: Team Contributions

B Program Code

GitHub Repository DQN:
https://github.com/yash-joshi5379/Gomoku-rl/tree/best-train

GitHub Repository Alphazero:
https://github.com/yash-joshi5379/Gomoku-rl/tree/alphazero-9x9

C Video

YouTube video link:
https://youtu.be/lpM9TbTGkxs

D Bibliography

References

[1] Murray Campbell, A.Joseph Hoane, and Feng-hsiung Hsu. "Deep Blue". In: *Artificial Intelligence* 134.1 (2002), pp. 57–83. ISSN: 0004-3702. DOI: https://doi.org/10.1016/S0004-3702(01)00129-1. URL: https://www.sciencedirect.com/science/article/pii/S0004370201001291.

[2] David Silver et al. "Mastering the game of Go with deep neural networks and tree search". In: *nature* 529.7587 (2016), pp. 484–489.

[3] David Silver et al. "Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm". In: *arXiv preprint arXiv:1712.01815* (2017). URL: https://arxiv.org/abs/1712.01815.

[4] Hado van Hasselt, Arthur Guez, and David Silver. *Deep Reinforcement Learning with Double Q-learning*. 2015. arXiv: 1509.06461 [cs.LG]. URL: https://arxiv.org/abs/1509.06461.