Gomoku Strategy Learning with AlphaZero: MCTS and Integrating Neural Networks

Tianle Chen, Ruchik Jani,
MS in Robotics, Northeastern University
Boston, MA, USA, 02115
chen.tianle@northeastern.edu, jani.ru@northeastern.edu

Abstract

We present a approach to Gomoku strategy learning by leveraging the AlphaZero framework, combining Monte Carlo Tree Search (MCTS) with neural networks to achieve advanced gameplay intelligence. The proposed method integrates the search efficiency of MCTS with the predictive accuracy of neural networks, enabling adaptive decision-making and strategic improvements over time. Through extensive experiments, we demonstrate the system's ability to outperform traditional Gomoku algorithms, showcasing its capability to learn and generalize complex strategies autonomously. Our findings suggest that this approach has significant implications for reinforcement learning in strategic board games and other decision-making tasks. Along with the AlphaZero-based approach, we have also investigated the effects of applying Greedy Heuristics and DQN on the Gomoku environment.

Code - https://github.com/Tenshi0x0/alphazero-simple

Introduction

Games like Chess, Go, and Shogi have served as platforms to develop and evaluate algorithms, strategies, and computational intelligence. Historically, these domains have inspired innovations in algorithms and heuristics, but the resulting systems often relied heavily on handcrafted features and domain-specific knowledge, limiting their adaptability to other games or tasks.

While the AlphaGo Zero algorithm marked a significant milestone by achieving superhuman performance in Go using deep neural networks and reinforcement learning solely through self-play. This approach demonstrated that knowledge of the game could be encoded and refined without relying on pre-programmed strategies, paving the way for general-purpose learning systems.

In this paper, we extend these advancements to Gomoku, a classic strategy board game. By applying the AlphaZero framework—combining Monte Carlo Tree Search (MCTS) and neural networks—we aim to develop a system that learns optimal Gomoku strategies from scratch, leveraging only the rules of the game. This study highlights the potential of reinforcement learning and self-play in developing generalizable AI systems, while also contributing to the understanding of strategic decision-making in complex domains.

In this study, we also integrate a DQN-based algorithm with the Gomoku environment as an alternative to the AlphaZero framework, exploring its capability to learn effective strategies through experience. The DQN implementation leverages a convolutional neural network for approximating Q-values, enabling the agent to evaluate potential moves and adapt its strategy dynamically. By employing techniques such as prioritized experience replay and target networks, the model improves its stability and learning efficiency over successive iterations. Training is conducted through self-play, where the DQN competes against heuristic-based opponents, such as random and greedy players, to refine its policy and optimize decision-making. Experimental results highlight DQN's performance on the environment with steps made towards learning progression.

Related Work

Gomoku is a deterministic, multi-turn game with a large discrete state-action space which requires long-horizon decision-making. Solving Gomoku involves balancing computational efficiency and strategic depth. Various methods from the fields of reinforcement learning, heuristic search, and game theory could be applied to this problem. In this section, we explore some alternative approaches, their potential benefits, and reasons for choosing the methods in this project.

Minimax Algorithm: Minimax is a classical adversarial search algorithm used in deterministic games. It assumes both players play optimally and evaluates positions to select the best move. Alpha-beta pruning optimizes Minimax by reducing the number of nodes explored. This algorithm guarantees optimal play under perfect evaluation functions. The Alpha-beta pruning makes the approach computationally feasible for shallow searches. However, the Minimax algorithm becomes computationally infeasible on our environment due to the large branching factor of the 15x15 board size. Since it requires handcrafted evaluation functions, it may struggle with the strategic depth of the Gomoku game. Samuel's early work (Samuel, A.L. 1959) on Checkers highlighted the limitations of handcrafted evaluation functions and the potential of self-learning systems.

Graph Neural Networks (GNNs): GNNs have been applied to Go and Chess but remain an emerging area for board games like Gomoku (Battaglia et al. 2018). GNNs can represent the Gomoku board as a graph, where nodes correspond to board positions, and edges represent spatial relationships. GNNs are well-suited for games with complex spatial dependencies as they capture spatial relationships more explicitly than CNNs. They are scalable to larger board sizes. However, they are not suitable for real-time applications as GNNs require iterative message-passing steps, which can slow down inference, especially on larger boards or with deeper networks. This overhead becomes significant for real-time applications like Gomoku gameplay, where decisions must be made within milliseconds.

In this project, MCTS, DQN, and AlphaZero-based methods were chosen due to their scalability, effectiveness in handling large state-action spaces, and ability to incorporate learning from experience. MCTS offers a robust search-based framework that balances exploration and exploitation through simulations, providing a strong baseline for decision-making. DQN leverages deep learning to approximate value functions, enabling effective generalization and learning from sparse rewards, which is crucial in games like Gomoku. Finally, the AlphaZero-inspired MCTS + Neural Network approach synergizes the strengths of search and learning, allowing the agent to build intuition through self-play while efficiently guiding the search process with learned policy and value functions. These methods strike an effective balance between computational efficiency, learning capability, and strategic depth, making them well-suited for Gomoku.

Project Description

In this section, we will talk about the environment and the methods in detail.

Environment Dynamics:

The environment for this project is a digital implementation of Gomoku which we have developed from scratch, on a 15x15 grid. The objective is to place five consecutive markers (horizontally, vertically, or diagonally) before the opponent. The environment simulates all the rules and dynamics of the game, allowing players (agents) to interact with it through predefined actions.

Environment State: The environment state is represented by a 15×15 matrix, where each cell indicates the presence of a marker:

- 0 for an empty cell.
- 1 for Player 1's marker (black).
- 2 for Player 2's marker (white).

Action Space: The action space corresponds to the positions on the grid, represented as integers in the range $[0, 224]$ for a 15×15 board. Illegal actions, such as placing markers on occupied cells, are masked. Depending on the steps taken, the number of actions will be dynamically determined at each step based on the empty states and the legal moves left.

Reward Function: Rewards are sparse and assigned as follows:

- $+1$ for a win.
- -1 for a loss.
- 0 for a draw.

When a player selects an action:

- The marker is placed in the corresponding position.
- The environment checks for a win condition by evaluating connected markers in all directions (horizontal, vertical, and diagonal).
- If the board is filled without a winner, the game results in a draw.
- The turn alternates to the other player unless the game ends.

The environment provides a function to retrieve all legal moves, ensuring that agents can efficiently compute valid actions. The game ends when:

- A player achieves five consecutive markers.
- The board is filled with no winner, resulting in a draw.

Methods:

A. Monte Carlo Tree Search (MCTS):

Given a state s, the neural network predicts an initial policy $\tilde{p}_\theta$ and a value v_θ. To refine these predictions, we utilize Monte Carlo Tree Search (MCTS) to iteratively improve the action probabilities during training. In the search tree, each node corresponds to a unique board configuration, with directed edges representing valid transitions between states caused by specific actions. Starting with an empty tree, the search expands one node (state) at a time. If a new node is added, its value is directly derived from the neural network, bypassing traditional rollouts. This value is then propagated back up the path traversed during the simulation.

The MCTS maintains the following statistics for each state-action pair (s, a):

- $Q(s, a)$: The average value of taking action a in state s, representing the expected outcome of simulations.
- $N(s, a)$: The visit count for action a in state s, tracking the number of times this action was explored.
- $P(s, a)$: The prior probability of selecting action a in state s, as predicted by the policy network.

The action selection process uses an upper confidence bound $U(s, a)$ to balance exploration and exploitation:

$$U(s,a) = Q(s,a) + c_{\text{puct}} \cdot P(s,a) \cdot \frac{\sqrt{\sum_b N(s,b)}}{1 + N(s,a)} \qquad (1)$$

The main steps of MCTS are:

1. **Selection:** Starting from the root node, the algorithm selects child nodes based on $U(s, a)$ until a leaf node is reached.
2. **Expansion:** If the leaf node corresponds to a previously unvisited state, it is added to the tree.

3. **Backpropagation:** The value of the leaf node is propagated up the search path, updating $Q(s,a)$ for all visited nodes. Terminal states are assigned rewards of $+1$ (win) or -1 (loss).
4. **Policy Update:** After multiple simulations, the visit counts $N(s,a)$ at the root node form an improved policy:
$$\pi(a,s) = \frac{N(s,a)}{\sum_b N(s,b)} \tag{2}$$

In the absence of a neural network, we employ a pure Monte Carlo Tree Search (MCTS) approach that relies on a manually defined evaluation function for move selection. The core idea is to use game-specific heuristics to guide the search process, instead of leveraging a learned policy and value function from a neural network.

In this approach, when the search reaches a leaf node, the algorithm does not rely on neural network predictions but instead calculates a heuristic score for each possible move. Specifically, the valid moves are evaluated using a custom board evaluation function, which estimates the potential of each move based on the current state of the board. This evaluation function works by considering the board from the perspective of both the current player and the opponent. It takes into account various directions (horizontal, vertical, diagonal) from each potential move and evaluates the number of connected pieces for both the player and the opponent. The heuristic score incorporates these counts, penalizing the opponent's connected pieces while rewarding the player's strategic positioning.

Furthermore, the custom board evaluation function was also deployed in a **greedy AI agent**, which selects actions by greedily choosing the move with the highest heuristic score at each step. This greedy agent serves as a baseline to evaluate the performance of the pure MCTS approach. By conducting matches between the pure MCTS agent and the greedy AI agent, the effectiveness of the MCTS algorithm can be illustrated, highlighting its ability to balance exploration and exploitation while leveraging the heuristic evaluation function.

B. MCTS + Neural Network (AlphaZero inspired):

The MCTS + NN approach combines Monte Carlo Tree Search (MCTS) with a neural network to efficiently explore the game tree and improve decision-making. This hybrid strategy synergizes the search-based exploration of MCTS with the learning capability of neural networks, allowing the agent to approximate optimal play through self-play. (Silver et al. 2017)

Core Components:

1. **Neural Network (NN):**
   - The NN predicts two outputs for a given board state s:
     - **Policy π:** A probability distribution over actions, guiding the search.
     - **Value v:** An estimate of the expected outcome from state s (win/loss/draw).
   - Formally, $(\pi, v) = f_\theta(s)$, where θ are the NN parameters.
2. **Monte Carlo Tree Search (MCTS):**
   MCTS builds a search tree dynamically, balancing exploration (visiting less-explored actions) and exploitation (focusing on promising actions). The working is described in detail in the previous section on MCTS.

Training Process:

1. **Self-Play:**
   The agent plays games against itself. For each state s_t encountered during self-play, the tuple (s_t, π_t, z) is recorded:
   - π_t: Visit count-based policy from MCTS.
   - z: Game outcome (+1 for win, -1 for loss, 0 for draw).
2. **Neural Network Update:**
   The NN is trained using gradient descent to minimize the loss:
$$\mathcal{L} = \sum_t \left((z_t - v_t)^2 - \pi_t \cdot \log \pi_\theta(s_t)\right) \tag{3}$$
   where:
   - $(z_t - v_t)^2$: Mean squared error between predicted value and true outcome.
   - $-\pi_t \cdot \log \pi_\theta(s_t)$: Cross-entropy loss between predicted and target policy.

MCTS focuses on promising regions of the game tree, guided by the NN's predictions. The NN generalizes across states, improving play without exhaustive search. The approach handles large state-action spaces effectively by combining search with learned priors. This MCTS + NN framework represents a core component of AlphaZero-like algorithms, enabling the agent to achieve strong play through iterative self-improvement and strategic exploration. It is particularly suited for complex games like Gomoku, where exhaustive search is infeasible, and nuanced strategies emerge through self-play.

Neural Network Architecture:

The neural network used in this project is a convolutional architecture tailored to capture the spatial dependencies of the Gomoku board. It consists of four convolutional layers, each followed by batch normalization and ReLU activation, which extract hierarchical spatial features from the board state. These convolutional layers progressively reduce the board's spatial dimensions while learning local and global patterns critical for decision-making. The resulting feature maps are flattened and passed through two fully connected layers, each with batch normalization, ReLU activation, and dropout for regularization. Finally, the network branches into two output heads: one predicting the policy (π) as a probability distribution over actions using a fully connected layer with log-softmax, and the other predicting the value (v) of the current state using a single neuron with a tanh activation to constrain the output to $[-1, 1]$. This architecture was chosen because convolutional layers excel at recognizing

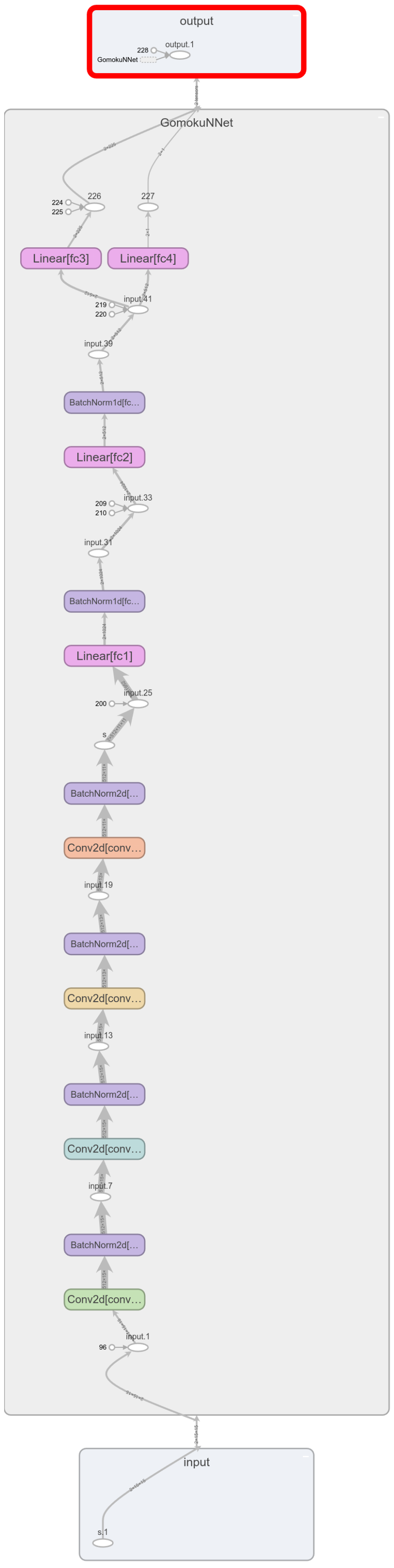


Figure 1: Neural Network Architecture Diagram for the AlphaZero-based approach showing the convolution layers along with batch normalization

spatial structures like winning lines or threats on a grid, making them ideal for Gomoku. Additionally, the dual-head design allows the network to simultaneously guide search (policy) and evaluate positions (value), which are both essential for the MCTS + NN approach. The inclusion of dropout and batch normalization ensures robustness and stability during training.

The policy head and value head in the neural network use distinct loss functions, tailored to their respective roles in guiding search and evaluating states in the MCTS + Neural Network framework. Cross-entropy loss measures the difference between the predicted policy $\hat{\pi}$ and the target policy π, which is based on MCTS visit counts. It penalizes the model when the predicted probabilities deviate from the ideal distribution suggested by MCTS. This loss ensures that the network learns to align its predictions with the improved strategy obtained through search, effectively guiding future exploration. Mean squared error (MSE) captures the deviation between the predicted value $\hat{v}$ and the true game outcome z. This choice reflects the continuous nature of the value head output, which represents the expected result of the game from a given state. By minimizing this loss, the network learns to accurately evaluate the desirability of states, helping MCTS make more informed decisions. MSE loss is suitable for regression tasks, where the objective is to predict continuous values like the game outcome. It ensures the network provides reliable evaluations of board states, which are crucial for backpropagating values in MCTS.

C. Deep Q-Network (DQN):

The DQN algorithm used in this project enables the agent to learn optimal policies for Gomoku by approximating the Q-value function, which estimates the expected cumulative reward of taking an action in a given state. Unlike the AlphaZero-based MCTS + NN method, which combines search and learning, DQN relies purely on learning through environment interaction, using reinforcement learning principles.

Core Components of DQN:

Q-value Function Approximation:

The Q-value $Q(s, a)$ represents the expected reward from taking action a in state s, followed by the optimal policy. DQN uses a neural network to approximate $Q(s, a)$:

$$Q(s, a; \theta) \approx E[R_t + \gamma \max_{a'} Q(s', a'; \theta)] \tag{4}$$

where:

- R_t: Immediate reward.
- γ: Discount factor, balancing immediate and future rewards.
- s', a': Next state and action.
- θ: Parameters of the neural network.

Experience Replay: Transitions $(s, a, r, s', \text{done})$ are stored in a replay buffer, allowing the model to learn from diverse past experiences. Sampling batches randomly from this buffer helps reduce correlation in training data, stabilizing learning.

Target Network: A separate target network $Q(s, a; \theta^-)$ is periodically updated to provide stable Q-value targets. The loss function is computed using the target:

$$y = r + \gamma \max_{a'} Q(s', a'; \theta^-) \tag{5}$$

The training minimizes the temporal difference (TD) error:

$$L(\theta) = E[(Q(s, a; \theta) - y)^2] \tag{6}$$

Epsilon-Greedy Exploration: The agent balances exploration and exploitation by choosing random actions with probability ϵ (decaying over time) and greedily selecting $\arg\max_a Q(s, a; \theta)$ otherwise.

DQN Algorithm:

State Representation:
The Gomoku board is represented as a 2D matrix, where each cell is 0 (empty), 1 (current player), or 2 (opponent). The input to the network is reshaped into a 4D tensor of shape: $[\text{batch}, 1, \text{board\_size}, \text{board\_size}]$.

Action Selection:
The agent uses epsilon-greedy exploration to choose actions:

- With probability ϵ, a random legal action is selected.
- Otherwise, the action maximizing $Q(s, a; \theta)$ is chosen.

Training:

1. After each step, the agent stores the transition in the replay buffer.
2. Mini-batches are sampled to compute the TD loss and update the network using gradient descent.
3. Periodically, the target network θ^- is synchronized with the main network θ.

Neural Network Architecture:
The DQN network differs slightly from the AlphaZero-based network in its structure and output:

Input: Similar to the AlphaZero architecture, the input is a single-channel 2D board tensor.

Convolutional Layers: The DQN uses three convolutional layers with increasing filter sizes and no batch normalization. This simpler design reflects the lack of a policy-value split in the output.

Fully Connected Layers: The flattened feature map is passed through two fully connected layers. The final layer outputs a single Q-value for each action (equal to the number of board positions).

Output: Unlike the AlphaZero-based network, which has separate policy and value heads, the DQN network has a single output head for $Q(s, a)$, making it purely value-driven.

By relying purely on reinforcement learning, DQN complements the MCTS + NN approach, providing a contrasting perspective on decision-making in Gomoku.

Experiments

We have carried out extensive experimentation in order to determine the best configurations possible for the AlphaZero and DQN approaches. The goal was to create an RL agent that could play on par with the pure MCTS and the greedy heuristics-based players.

A. MCTS + Neural Network (Alpha-Zero):

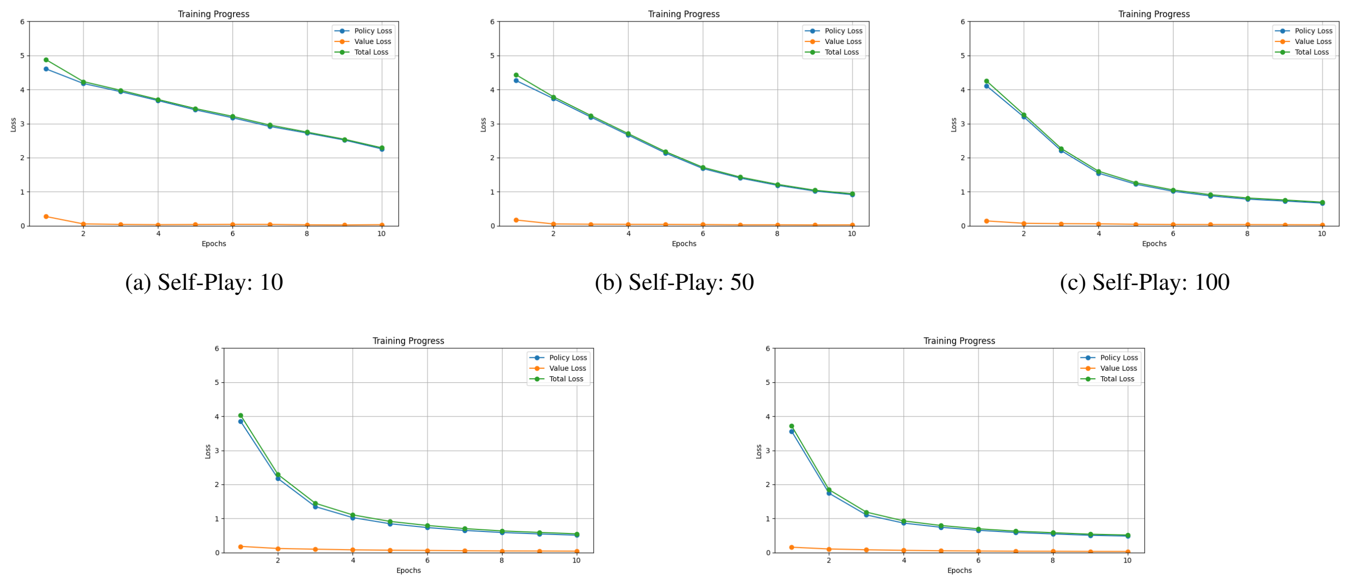
According to the original AlphaZero paper (Silver et al. 2017), they required a huge amount of compute power to train the AlphaZero model. The Google DeepMind team used 5000 first-generation TPUs to generate the self-play games and 64 second-generation TPUs to train the neural network. The training time for different games was given as: Chess (9 hours), Shogi (12 hours), and Go (34 hours). The model was initialized with random parameters and played millions of self-play games against itself to learn and improve it's game strategies.

Since we don't have such a high level of compute power or the time, we have reduced the parameters used by them to suit our hardware. We don't expect expert level game play from our trained AlphaZero model. However, we do plan to achieve some basic level of game play through our training methodology.

Neural Network Hyperparameters:
Learning Rate = 0.001
Dropout Rate (for regularization) = 0.3
No. of epochs = 10
Batch Size = 64
No. of channels in convolution layers = 512
Training device = GPU

The chosen hyperparameters are tailored to balance computational efficiency and model performance for the Gomoku application. A learning rate (0.001) allows steady weight updates, avoiding overshooting while converging effectively. The dropout rate (0.3) prevents overfitting, crucial given the limited diversity in game states. Using 512 channels in convolutional layers enables capturing rich spatial features on a 15x15 board. The batch size (64) strikes a balance between computational load and gradient estimation quality. Ten epochs allow sufficient training iterations without overfitting, leveraging GPU for faster computation. For most part of the experimentation prcoess, these hyperparameters remain the same.

Number of Self-Play Games:
Increasing the number of self-play games will increase the diversity of the training data generated for the Neural Network. However, this comes at a significant cost in terms of computation time. As self-play games increase, the time



(a) Self-Play: 10 (b) Self-Play: 50 (c) Self-Play: 100

(d) Self-Play: 200 (e) Self-Play: 300

Figure 2: Comparison of loss curves for different numbers of self-play games. The plots show the training curves for five settings: Self-Play 10, 50, 100, 200, and 300. Each plot illustrates the progression of Policy Loss (Blue), Value Loss (Orange), and Total Loss (Green).

required to generate training data from them increases. The time required to complete one epoch for the Neural Network also increases.

The goal of this experiment is to determine the optimum value of self-play games by increasing it in this order: 10, 50, 100, 200 and 300. We want to determine the point at which increasing the self-play games yields diminishing returns, i.e., there is no improvement in gameplay performance. Since the training time increases, and there is no significant improvement beyond a certain point, it doesn't make sense to keep on increasing the number of self-play games in hopes of improving the AlphaZero player. We will plot the loss curves: Policy, Value, and Total Loss (Policy + Value) to determine the optimum value.

The loss curves for different numbers of self-play games highlight the trade-off between training data and model performance. From figure 2, it is evident that when there are 10 self-play games (10 SP), the losses start high and decrease slowly, reflecting the challenges of training with limited data. The small number of self-play games results in insufficient diversity, causing the model to underfit and struggle to generalize across different board states. By contrast, the curve for 50 SP, shows steady and smoother loss reductions, indicating the benefits of a moderately sized dataset. With 50 games, the model begins to effectively optimize its policy and value functions, making notable progress in learning strategies while maintaining computational feasibility. The intital loss values in all the cases are decreasing with increase in self-play games which improves the inital predictions and training stability. The shape of the curves is also changing with this increase. A sharper decline and smoother flattening at higher values like 100, 200 and 300 suggest better learning and convergence, while gradual drops and a linear shape reflect the challenges caused by insufficient data at the lower values like 10, 50.

When the self-play training dataset is expanded further, as in the 100 SP, 200 SP, and 300 SP cases, the loss curves exhibit rapid declines and stabilize at lower values, signifying improved convergence and generalization. The 100 SP setup provides a balanced training environment, offering sufficient data diversity without excessive computational overhead. The 200SP and 300SP setups achieve even greater stability and lower loss values, with the latter representing the most comprehensive dataset. However, diminishing returns occur beyond 200 self-play games, where the computational cost rises significantly without proportional gains in model performance. This is confirmed by playing against the MCTS + NN player in our game environment.

This indicates that the increasing the number of self-play games beyond the range of 200-300 will not provide any performance improvements in our current setup. Unlike AlphaZero, which utilized millions of self-play games supported by Google's vast computational infrastructure, our setup operates under very tight resource constraints, making it infeasible to generate such extensive data. The neural network architecture that we are using in our project is not a very deep network and is very different from the one used in AlphaZero. Increasing the training data on such smaller networks increases the risk of overfitting to the training data.

Number of MCTS simulation moves:

This parameter in Monte Carlo Tree Search (MCTS) determines the number of simulations (i.e., the number of rollouts or expansions of the game tree) conducted at each de-

Black (P1	White (P2)	P1_Win	P2_Win	Draw
SIM_30	*SIM_100*	19	21	0
SIM_100	*SIM_30*	24	16	0

Table 1: Game Play results of AlphaZero with Self Play = 10 and Number of MCTS Simulations (SIM) = 30 and 100. P1 represents Player-1 (Black), similarly P2 represents Player-2 (White).

Black (P1	White (P2)	P1_Win	P2_Win	Draw
SIM_30	*SIM_100*	15	25	0
SIM_100	*SIM_30*	31	9	0

Table 2: Game Play results of AlphaZero with Self Play = 50 and Number of MCTS Simulations (SIM) = 30 and 100. P1 represents Player-1 (Black), similarly P2 represents Player-2 (White).

cision point during self-play. The Selection Phase is where this parameter is most relevant because it defines how many simulations are run to explore and refine the MCTS tree. While its impact cascades through all phases of MCTS (via more visits, improved statistics, and better policies), it is primarily a parameter for determining the depth and breadth of tree exploration during selection. Increasing this value has a significant impact on the learning process and overall effectiveness of the model. However, increasing the number of simulations beyond a certain point just increases the computation time and inference (gameplay) time. It does not provide any additional performance improvement. Through our experiment we determined that increasing this value beyond 100 will not improve the agent's learning process.

In this experiment, we are going to compare the performance of the AlphaZero agent with MCTS simulations 30 and 100. We have developed a code to play a custom number of matches between any two players. We used the code to determine the results of this experiment by having both the agents play 40 matches in each case. An agent gets to play equal number of matches as black and white, i.e., it plays 20 matches as black (Player-1) and 20 as white (Player-2) in each match. From the Tables 1 and 2, we can infer that with SIM_100, both Black and White demonstrate stronger play, as reflected in the higher win counts compared to SIM_30 irrespective of the number of self-play games. This indicates that additional simulations help players generate higher-quality self-play data, leading to better training and decision-making. More MCTS simulations enable the model to explore a larger portion of the game tree, leading to stronger policies. This reduces the likelihood of the model making suboptimal moves in critical game states. With 30 simulations, the tree exploration might miss optimal moves or critical opponent responses, resulting in less reliable data for training. With 100 simulations, MCTS better balances exploitation (following the best-known strategy) and exploration (discovering new strategies).

Player 1 (Black) consistently wins more games than Player 2 (White) in all scenarios. This is due to the inherent bias in Gomoku. Black (the first player) has a significant advantage because it gets the first move and can dictate the game's flow. In Gomoku, an optimal strategy allows the first player to control the board effectively, especially in high-skill games like this. Despite increased MCTS simulations, White struggles to match Black's performance. This is because even with better move evaluations, White is inherently reacting to Black's lead rather than setting the pace. This bias is deeply rooted in the game's mechanics due to the nature of its rules. This advantage cannot be overcome even when using advanced methods like AlphaZero.

In all the experiments, we observed that the value loss is very low and converges much faster than the policy loss. This discrepancy between the policy and value losses arises from the distinct nature of the learning objectives of these two loss functions. The value loss, being a regression task, involves predicting simple, bounded outcomes (win, loss, or draw) and converges quickly due to its binary nature and consistency during self-play. In contrast, the policy loss involves learning a probability distribution over a high-dimensional action space, requiring the network to balance exploration and exploitation for numerous legal moves, which is inherently more complex. While the value head works with straightforward scalar targets, the policy head must optimize multi-modal targets derived from MCTS simulations, which vary across board states. This imbalance in task complexity and data diversity explains why the policy loss is persistently higher during the training phase.

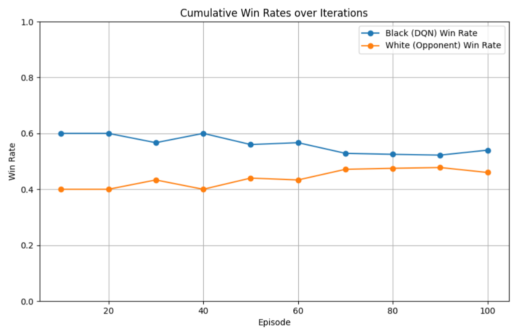
In order to obtain experimental data, we have limited the number of training iterations in AlphaZero appraoch to 1. The actual process involves training at hundreds or thousands of iterations in which the model is compared against a randomely initialized network in the first iteration to determine if the model should be retained or discarded for further iterations. In subsequent iterations, if the model trained during that particular iteration is better than the one from the last iteration, it is retained for training in further iterations. Since performing even 1 iteration takes a significant amount of time for a higher number of self-play games (about 2 hours for 100 SP), we had to train it for 1 iteration to obtain our learning curves.

B. Deep-Q Network (DQN:

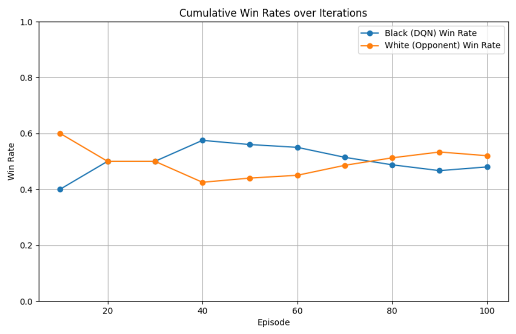
For training and evaluation of the DQN agent, we carry out two experiments and plot the rewards vs. episodes and the win rate curves for the DQN agent. We are even using a prioritized replay buffer to focus on the most relevant transitions in DQN training. Given the computational constraints, we decided to carry out both the experiments by training the DQN agent for 10 iterations with 10 episodes each (100 episodes total).

Effects of adding Greedy Player in training:

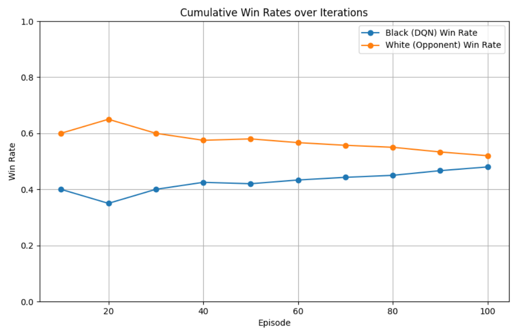
To provide a diverse training dataset to the DQN agent, we have developed our system to incorporate two different methods of training: one involving the greedy AI player and one without it. Since the greedy player is quite powerful, if the DQN agent is trained against it in the early stages of training, it will lose quickly and fail to learn any meaningful strategies. We explore the effects of adding the Greedy



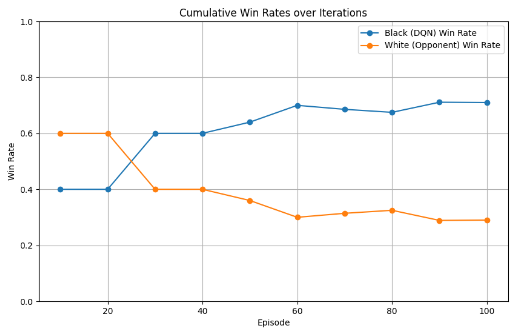
(a) Win rate curve with no greedy player (DQN - Black)



(b) Win rate curve with no greedy player (DQN - White)



(c) Win rate curve with greedy player (DQN - Black)



(d) Win rate curve with greedy player (DQN - White)

Figure 3: This figure shows how the cumulative win rate curves across 10 iterations for the DQN playing as the first player (black) and second player (white) against a combination of players with and without a greedy player. The black player is represented by the blue curve, and the white player is represented by the orange curve in all the cases.

Player in training. In our code, the DQN trains against three types of opponents: random player, DQN player (self-play), and greedy player. Without the greedy player, the opponent alternates between random and self-play. When the greedy player is included, the DQN plays against random, self-play, and greedy players in a repeating cycle, with the greedy player appearing in iterations that are multiples of 3.

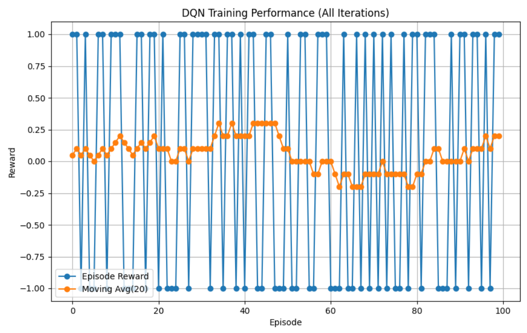
The Hyperparameters used in this experiment are:

1. Learning Rate = 0.0001
2. Epsilon Decay = 0.999
3. Target Network Update Frequency = 500
4. Batch Size = 128
5. Discount Factor = 0.99
6. Replay Buffer Size = 10,000

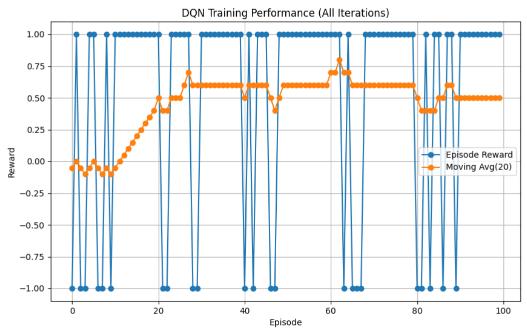
From Figure 3, we can infer that the win rate of DQN as black hovers around 60% throughout the training iterations in the first curve. This indicates that the DQN agent learns reasonably well against the random player and itself. The win rate shows only minor fluctuations, implying that the DQN agent is consistently outperforming the random player and itself. This suggests that the training process successfully instills a basic level of competency in the DQN agent. The same conclusions can be drawn about DQN as white player.

As soon as the Greedy Player is added in the training process, the win rate of the DQN agent drops significantly in both the cases. The Greedy Player is needed to train the DQN agent as the agent needs a diverse training dataset with different players to guide its learning process. However, adding the greedy player early introduces a much stronger opponent than random, which the DQN agent is not yet capable of competing against. This leads to consistent losses, reducing the agent's reward signal and making it harder for the agent to differentiate good actions from bad ones. The lack of early wins can destabilize learning by overwhelming the replay buffer with negative experiences. To improve, the agent should first learn basic strategies against easier opponents before facing tougher ones like the greedy player.

The performance of the DQN agent might be better in absence of the greedy player, but the win rate just plateaus around 50-60%, and doesn't significantly improve, which suggests that we might need further hyperparameter tuning to improve the model. This leads us to our second experiment on DQN.



(a) Rewards curve plotted with old hyperparameters (Set-1)



(b) Rewards curve plotted with new hyperparameters (Set-2)

Figure 4: This figure shows the difference between the difference between the rewards vs. episodes curves for different sets of Hyperparameters with the DQN player going first (Black). The blue lines indicate the raw rewards, and the orange curve is the moving average of the rwards.

Tweaking Hyperparameters:

The Hyperparameters used in this experiment are:

1. Learning Rate = 0.00005
2. Epsilon Decay = 0.9995
3. Target Network Update Frequency = 1000
4. Batch Size = 256
5. Discount Factor = 0.99
6. Replay Buffer Size = 50,000

The learning rate has been reduced to slow down the weight updates in order to reduce the risk of overshooting the optimal policy during training. This helps stabilize the training process, especially in environments with high variance rewards like Gomoku. The epsilon decay value has been changed to maintain the balance between exploration and exploitation. Increasing the update frequency reduces how often the target network is updated, leading to smoother training. This reduces instability caused by frequent updates and allows the current policy to converge more before the target policy shifts. A larger batch size of 256 increases the stability of gradient updates by reducing variance in each training step. This leads to better generalization and convergence in complex environments. Increasing the replay buffer size allows the agent to store more diverse transitions, providing richer training data. This helps prevent overfitting to recent episodes and improves learning.

The rewards being plotted are raw episode rewards collected during each episode of training. Since these just have 3 distinct values: 1, -1, and 0, it is very difficult to visualize the overall trends in learning progress with the raw rewards alone. The moving average smooths out fluctuations in the raw episode rewards, providing a clearer view of the overall learning trend. It helps identify whether the agent is improving over time, despite noisy episode-to-episode performance. Without the moving average, the high variance in episode rewards would make it challenging to assess the agent's progress.

From Figure 4, we can clearly see the difference between the learning progression of the DQN agent with hyperparameters used in the previous experiment (Set-1), versus the new ones (Set-2) used here. The moving average reveals a slow and inconsistent improvement, with large oscillations and a plateau in performance after about 50 episodes. This indicates unstable learning since the agent might be overfitting to recent experiences due to the lower replay buffer size. Other hyperparameters like higher learning rate and higher target network update frequency also contribute to this instability. In contrast, the episode reward exhibits fewer fluctuations in Set-2 as compared to Set-1, and the moving average steadily increases until around 50 episodes. Afterward, the performance stabilizes with a higher reward average compared to Set-1. The agent benefits from a more diverse replay buffer and controlled weight updates, leading to better generalization and smoother learning. We also observed that the win rate of the DQN agent was also consistently increasing with these new hyperparameters.

Throughout our experiments, we have avoided showing any comparison between the different players available: Greedy, MCTS, AlphaZero and DQN. This is because all these players are at completely different levels. MCTS is so powerful that it can easily beat all these players in all games. The Greedy player is also substantially powerful than the AlphaZero and DQN players. There will be no meaningful comparisons to show if all the players can't compete at the same level. Our work highlights how to design and improve the AlphaZero and DQN players through experimentation with different parameters.

Conclusion

Gomoku is a game with a large state space and requires strategic foresight to win. This creates several challenges for the RL agents we are trying to train.

The DQN agent suffers particularly from delayed and sparse rewards. The actions taken early in the game may have delayed consequences. Effective agents need to understand this delayed reward structure, which can be challenging for algorithms focused on short-term feedback like DQN. DQN struggles when rewards are infrequent or delayed, as it relies on temporal difference learning to propagate reward signals back through its value estimates. Effective strategies in Gomoku may require exploring sequences of moves that initially seem suboptimal but lead to long-term gains, something DQN struggles to handle without substantial engineering, such as advanced exploration strategies. Reward shaping is something that could be explored with DQN in which we could assign rewards to the DQN agent for moves that block the opponent's progress and penalize the agent for moves that allow the opponent to win in the next move. Our work here highlights how we can tailor the DQN agent to suit the environment's requirements. We already have a very good set of hyperparameters that show progress in learning. If we introduce the greedy player after substantial training iterations with the random player and self-play, we expect the model to improve significantly using the Set-2 hyperparameters. However, further modifications like different exploration strategies and reward shaping would likely be required to create a DQN player achieving a basic level of gameplay. It would require almost 1000 - 2000 episodes to train which requires a lot of time and compute power.

Our basic implementation of the AlphaZero algorithm involving MCTS + neural networks suffers from the lack of computational resources required for training the system. The pure MCTS player is quite powerful and pretty tough to beat as it uses pre-defined heuristics incorporating domain knowledge. However, the prime motivation behind the AlphaZero algorithm was to create a generalizable and scalable solution. Even though we could not achieve such high level of gameplay using our approach, the AlphaZero agent does have a basic sense of attacking and defending against its opponents in the inital stages of the game. This highlights the potential of self-play in model training. Given sufficient iterations and resources, our model would definitely improve and learn better game strategies.

References

Battaglia, P. W.; Hamrick, J. B.; Bapst, V.; Sanchez-Gonzalez, A.; Zambaldi, V.; Malinowski, M.; Tacchetti, A.; Raposo, D.; Santoro, A.; Faulkner, R.; Gulcehre, C.; Song, F.; Ballard, A.; Gilmer, J.; Dahl, G.; Vaswani, A.; Allen, K. R.; Nash, C.; Langston, V.; Dyer, C.; Heess, N.; Wierstra, D.; Kohli, P.; Botvinick, M.; Vinyals, O.; Li, Y.; Pascanu, R.; and Battaglia, P. 2018. Relational Inductive Biases, Deep Learning, and Graph Networks. arXiv preprint arXiv:1806.01261.

Samuel, A.L. 1959. Some Studies in Machine Learning Using the Game of Checkers. *IBM Journal of Research and Development*.

Silver, D.; Schrittwieser, J.; Simonyan, K.; Antonoglou, I.; Huang, A.; Guez, A.; Hubert, T.; Baker, L.; Lai, M.; Bolton, A.; Chen, Y.; Lillicrap, T.; Hui, F.; Sifre, L.; van den Driessche, G.; Graepel, T.; and Hassabis, D. 2017. Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm. *arXiv preprint arXiv:1712.01815*.