A Chess Engine Using MCTS with a Deep Value Network

Lorenzo Vittori
Matteo Comini

Academic Year 2024/2025

Contents

Chapter 1

Introduction

1.1 Background and Motivation

Reinforcement Learning (RL) has emerged as a powerful framework for enabling agents to learn complex decision-making strategies through interaction with uncertain or partially observable environments. By combining trial-and-error learning with function approximation, modern RL methods can tackle tasks ranging from simple grid-world navigation to board games and robotic control.

In this project, we explore the application of RL in the context of game playing, specifically focusing on chess. Chess serves as an excellent domain for studying AI and RL due to its rich strategic complexity and well-defined rules. We aim to develop a chess engine that can compete at a high level using Monte Carlo Tree Search (MCTS) combined with a deep value network.

1.2 Project Overview

This report presents two main projects: the first involves developing a Deep Q-Network (DQN) agent for the Frozen Lake environment, and the second focuses on creating a MiniChess engine using MCTS with a deep value network.

In the Frozen Lake project, we design and train a DQN that learns to navigate the Frozen Lake environment, a stochastic grid-world, by approximating the optimal action–value function. The agent learns through trial and error, receiving rewards for reaching the goal and penalties for falling into holes. We also integrate MCTS with the trained DQN to enhance the agent's performance by enabling look-ahead planning.

The MiniChess engine project involves developing a chess engine for Gardner's 5×5 variant of chess. Inspired by the AlphaZero paradigm, we integrate MCTS with a deep value network that evaluates board positions. The neural network provides fast, informed leaf evaluations during the MCTS search, drastically reducing the need for random rollouts and enabling competitive play with limited computational resources.

1.3 Report Structure

The report is structured as follows: After this introduction, Chapter 2 describes the Frozen Lake environment and the RL approach used to train the agent, including details on the DQN architecture, training pipeline, and MCTS integration. Chapter 3 presents the MiniChess engine, detailing the problem description, algorithmic approach, including MCTS and value network integration, frameworks and dependencies, engine architecture, and results. Chapter 4 discusses the issues encountered during the project and the solutions implemented. Finally, Chapter 5 concludes the report, summarizing the key findings and suggesting directions for future work.

Chapter 2

Frozen Lake

2.1 Environment and Problem Description

The *Frozen Lake* environment is formulated as a finite-horizon grid-world Markov decision process (MDP). The agent starts on the upper-left corner S and aims to reach the goal tile G situated on the opposite corner while avoiding holes H. Ordinary tiles F are safe, but when the map is declared *slippery* every intended move is perturbed with probability $\frac{1}{3}$ of veering $\pm 90°$, thus introducing stochastic transitions.

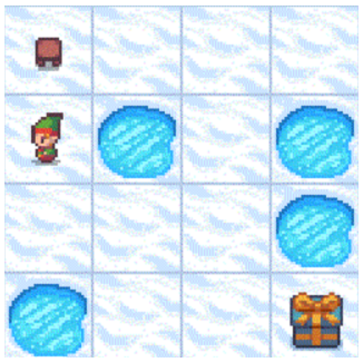


Figure 2.1: Enter Caption

- **State space** — Each grid position is encoded as a single integer $s = \text{row} \times n_{\text{col}} + \text{col}$, yielding the discrete observation space $\mathcal{S} = \{0, \ldots, n_S - 1\}$ with $n_S = n_{\text{row}}\, n_{\text{col}}$.
- **Action space** — Four actions `LEFT`, `DOWN`, `RIGHT`, `UP` constitute $\mathcal{A}$.
- **Map generation** — A helper `generate_random_map` draws an $n \times n$ binary matrix, fixes S and G, and repeatedly samples until a depth-first search (DFS) verifies that at least one viable path exists.
- **Transition tensor** $P(s, a)$ — During construction the environment enumerates every pair (s, a) (including slip branches) once; subsequent calls to `step` reduce to an $\mathcal{O}(1)$ table lookup.

Episodes terminate upon reaching G, falling into H, or after `MAX_EPISODE_STEPS` interactions.

2.2 Reinforcement Learning Approach

Learning proceeds in two separate phases:

1. **Offline value learning** — a Deep Q-Network (DQN) approximates the optimal action–value function Q_θ from gameplay experience gathered under an ε-greedy policy.

2. **Online planning** — at evaluation time the frozen DQN is embedded inside a Monte-Carlo Tree Search (MCTS) that performs look-ahead from the current root state before every move.

2.2.1 Reward shaping

To speed up convergence the environment returns dense rewards; constants are centralised in `config.py`:

Event	Reward
Goal reached	$+100$
Fall into hole	-100
Any other step	-1

Table 2.1: Reward shaping constants.

2.2.2 Deep Q-Network

The Q-function $Q_\theta(s, a)$ is parameterised by a fully-connected network defined in `deep_q_learning.py`:

$$\text{one\_hot}(s) \ \rightarrow \ \text{Linear}(128) \ \rightarrow \ \text{ReLU} \ \rightarrow \ \text{Linear}(128) \ \rightarrow \ \text{ReLU} \ \rightarrow \ \text{Linear}(|\mathcal{A}|).$$

Experience replay & target network A buffer of 10 000 transitions supplies IID minibatches (size 64). Parameters of an auxiliary target network are copied every 200 optimiser updates.

Optimisation The loss is the smooth L_1 distance between current predictions and bootstrapped targets

$$y = r + \gamma \max_{a'} Q_{\theta^-}(s', a'),$$

with Adam (learning rate 10^{-3}) and discount $\gamma = 0.99$.

Exploration Actions follow an ε-greedy policy with a decaying schedule

$$\varepsilon_t = \varepsilon_{\text{end}} + (\varepsilon_{\text{start}} - \varepsilon_{\text{end}})e^{-t/\tau}, \quad \varepsilon_{\text{start}} = 0.8,\ \varepsilon_{\text{end}} = 0.01,\ \tau = 500.$$

2.2.3 Training pipeline

1. **Initialisation** Fix an 8×8 random map with seed 48 for full reproducibility.
2. **Collect–update loop** During each episode the agent selects an action, stores the transition, and—once the buffer holds 500 samples—performs one gradient step per environment step.
3. **Checkpointing** Policy weights are periodically saved to disk and synchronised to the target network.

2.2.4 Monte-Carlo Tree Search executor

At evaluation time function `mcts()` uses the exact model extracted once from `env.P`.

Selection Children maximise

$$U(s, a) = \frac{Q}{N_{sa}} + c\sqrt{\frac{\ln N_s}{N_{sa}}}, \quad c = 1.0.$$

Expansion The first unvisited node along the path is added to the tree.

Roll-out From the leaf state the DQN's `greedy_action` is simulated up to `ROLLOUT_DEPTH = 50`; if unavailable, the policy defaults to uniform random.

Back-propagation Returns are discounted by γ^t and accumulated along the path.

Inference After `NUM_SIMULATIONS` trajectories (typically a few hundred) the action with the highest Q value is executed; normalised counts form a stochastic policy.

2.2.5 Performance

On a deterministic 4×4 board the hybrid DQN+MCTS agent achieves nearly 100% success, usually in the optimal number of moves. On slippery 8×8 layouts it exceeds 73% in only 1000 iterations while keeping per-move search latency in the millisecond range, and no planning overhead is incurred during DQN training.

2.3 Code Structure and Logic

Module / Script	Responsibility
`frozen_lake.py`	Environment implementation, map generator, Gym registration (discrete spaces, reward shaping, multiple render back-ends).
`deep_q_learning.py`	DQN architecture, replay buffer, ε-greedy policy; exposes `choose_action`/`greedy_action`.
`q_learning.py`	Tabular baseline compatible with the same planner interfaces.
`mcts_module.py`	Exact-model Monte-Carlo Tree Search using UCT; no environment calls inside simulations.
`main.py`	End-to-end experiment: trains agents then evaluates via MCTS (seeded for reproducibility).
`new_main.py`	Live visualisation of the search tree beside gameplay (matplotlib animation).

All constants reside in a dedicated `config.py` so alternative reward schedules or hyper-parameters can be changed without touching the algorithmic code.

2.3.1 Interactive Demo

The script `new_main.py` provides an animated demonstration of the trained agent interacting with the environment under Monte-Carlo Tree Search guidance. It combines policy roll-outs from a Deep Q-Network with graphical rendering of both the temporary planning tree and the actual decision path.

Purpose

This script is designed as an educational tool to visualise how MCTS selects actions and evolves its internal search tree over time. The board is fixed (a 4×4 deterministic map), and each decision is animated with a small delay for interpretability.

Features

- **Real-time tree visualisation** — Shows both the temporary search tree used for the current move and the full tree of actions taken so far.
- **Tile-aware color coding** — Goal nodes appear green, holes red, and ongoing expansions sky-blue.
- **Compact UCT planner** — The internal planning function `_mcts_for_visualization` implements UCT with 20 simulations.

This demo complements the numerical results by offering intuition into the agent's planning mechanism and how deep RL integrates with tree-based search.

2.4 Performance

Table 2.2: Evaluation of greedy agents (100 test episodes) on an 8×8 FrozenLake map under different training budgets and environment dynamics.

Run	**Slippery?**	**#Episodes**	**MCTS Sims**	**DQN Success** [%]	**Q-table Success** [%]	**Wall-time** [s]
Log 1	No	1000	5 000	100.0	100.0	85.7
Log 2	Yes	1000	5 000	69.0	73.0	117.4
Log 3	Yes	1000	15 000	56.0	62.0	122.6
Log 4	No	300	1 000	0.0	100.0	25.8

Discussion.

- **Deterministic vs. stochastic transitions.** In the non-slippery variant (Log 1) both agents eventually find deterministic, optimal paths; a single greedy rollout is always reliable, so Monte-Carlo Tree Search (MCTS) achieves a perfect 100 % success rate with only 5 000 simulations. When the surface becomes slippery (Logs 2–3) each action branches into multiple stochastic outcomes. This raises the planning horizon's variance: DQN drops to 69 % and 56 %, while the tabular policy—though slower to learn—degrades more gracefully (73 % and 62 %).

- **More simulations do not guarantee better control.** Increasing the MCTS budget from 5 k to 15 k (Log 3) barely improves success and even harms DQN. The extra samples average over many contradictory rollouts; without a stronger value function, the tree backs up noisy returns, so additional search depth adds diminishing—and sometimes negative—returns while inflating wall-time.

- **Training budget matters more than search budget for DQN.** Cutting the learning phase to 300 episodes (Log 4) removes 70 % of DQN's gradient updates. The network never converges, and greedy evaluation fails entirely (0 %). By contrast, Q-learning's lookup table still attains 36 % training success and 100 % greedy success because deterministic dynamics let it memorise state–action pairs even with few passes. Runtime shrinks to 26 s thanks to both fewer episodes and a smaller search budget (1 k sims).

Chapter 3

MiniChess Engine

3.1 Problem Description

In Gardner's 5×5 Minichess (on the leftmost five columns of a standard board), all standard pieces (rook, knight, bishop, queen, king, and pawns) are placed on a 5×5 board. This variant has a much smaller state space than full chess but still retains strategic complexity. The engine's goal is to choose strong moves in this game.

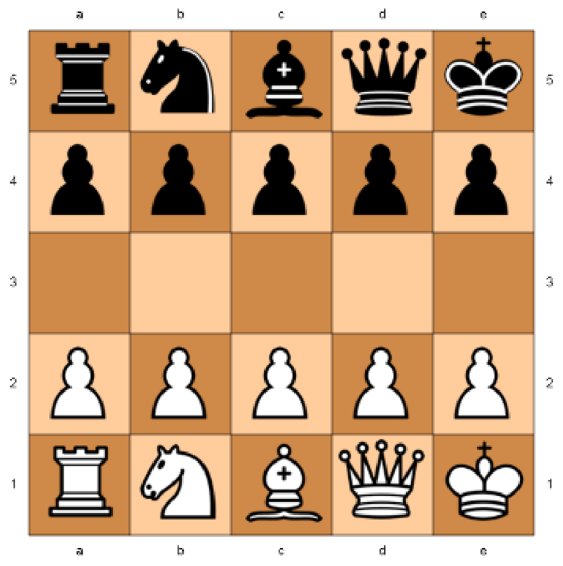


Figure 3.1: Image description.

3.2 Algorithmic Approach

The MiniChess engine is built on Monte Carlo Tree Search (MCTS) combined with a deep neural network value function. In practice, the code performs the following MCTS steps each time it needs to pick a move (see `mcts_pt.py`):

1. **Selection:** Starting from the root node (current position), recursively select child moves according to a PUCT policy (`_Node.ucb1` in `mcts_pt.py`) until reaching a leaf.
2. **Expansion:** If the leaf node is non-terminal and has untried children, expand the tree by adding one child node via `_Node.expand`.
3. **Evaluation:** Evaluate the new node by querying the deep value network (in `value_network.py`) to estimate $V(s)$. Encoding is done in `encode_state_as_tensor` (see `mcts_pt.py`).

4. **Backpropagation:** Propagate the obtained scalar value up the tree, updating visit counts and value estimates of ancestor nodes.

3.2.1 Reward shaping in MiniChess

To speed up learning in self-play, we apply a potential-based shaping with a step penalty (see `config.py`):

- Calculation of the material potential $\phi(s)$ in `train.py` via `compute_material_potential`, which uses the values in `PIECE_VALUES`.
- Shaped reward for the player to move m:
  $$r_m \;=\; \alpha\big(\gamma\,\phi(s') - \phi(s)\big) \;-\; \text{STEP\_COST},$$
  and for the opponent:
  $$r_{\text{opp}} \;=\; -\,\alpha\big(\gamma\,\phi(s') - \phi(s)\big).$$
  (See `compute_white_black_rewards` in `train.py`.)

3.3 Frameworks and Dependencies

The training pipeline in `train.py`, the MCTS engine in `mcts_pt.py`, the network definition in `value_network.py`, and the state representation in `minichess_state.py` make use of various libraries, divided here into three categories:

1. **Python Standard Libraries**
2. **Third-Party Libraries**
3. **Local Project Modules**

1. Python Standard Libraries

- `os`, `sys`: path management, dynamic insertion into `sys.path`, and runtime environment configuration.
- `math`, `random`: basic mathematical operations and random number generation for initialization and exploration strategies.
- `logging`, `csv`: structured logging of training events and saving results (metrics, losses, parameters).
- `collections.Counter`: counting occurrences of events (e.g., win/loss statistics) during training.
- `typing` (`List`, `Tuple`, `Optional`): annotations to improve readability and static type checking.
- `dataclasses.dataclass`: simplifies the definition of data classes (used in `mcts_pt.py` to structure nodes).

2. Third-Party Libraries

- **PyTorch** (`torch`, `torch.nn`, `torch.cuda.amp`, `torch.multiprocessing`, `torch.utils.data`):
  - `torch`: core for tensors and GPU/CPU operations.
  - `torch.nn / torch.nn.functional` (`Conv2d`, `BatchNorm2d`, `Linear`, `LeakyReLU`): defines the value network architecture.
  - `torch.cuda.amp` (`GradScaler`, `autocast`): enables mixed precision training to speed up training and reduce memory usage.
  - `torch.utils.data.Dataset`, `DataLoader`: implements the replay buffer as a dataset and handles automatic batching.
  - `torch.multiprocessing`: parallelism for data generation via self-play across multiple processes.
- **NumPy** (`numpy`): efficient array manipulation for state encoding into tensors.

3. Local Project Modules

- **config.py** Game settings and global parameters: defines `PIECE_VALUES`, `STEP_COST`, `ALPHA`, `GAMMA`, `INPUT_CHANNELS`, and other training and MCTS constants.
- **games.gardner.minichess_state.MiniChessState** Immutable representation of the game state. Main methods:
  - `board()`, `current_player()`, `get_legal_moves()`, `next_state()`, `is_terminal()`.
  - Definition of `Move = Tuple[int,int,int]` (piece, src_idx, dst_idx).
- **games.gardner.GardnerMiniChessLogic** Rules and helpers:
  - Calculation of legal moves, terminality checks, captures, castling, promotion.
  - `move_to_algebraic()`, `idx_to_square()`.
- **games.gardner.mcts_pt.MCTS** PUCT-MCTS with batching: `search()`, `_select()`, `_enqueue()`, `_flush()`, `_select_move()`.
- **games.gardner.value_network.ValueNetwork** CNN architecture for $V(s)$: three convolutional layers, batch-norm, LeakyReLU, flatten, and fully-connected layers.
- **games.gardner.ChessFunctions** GUI and demo in Pygame: board drawing, input handling, `demo.py`, `demo_dual.py`.
- **torch.multiprocessing** & **TrueSkill** Parallel self-play and Elo evaluations; metrics saved in `metrics.csv`.

3.4 Engine Architecture

Below are detailed descriptions of the main modules of the MiniChess engine. Each subsection reports the role of the file, the most relevant classes and functions, as well as the overall logical flow of each component.

3.4.1 minichess_state.py

This module encapsulates the representation of the game state and provides the fundamental operations to handle transitions and query terminal conditions.

- **Class** `MiniChessState`:

  `board(self) -> List[List[int]]` Returns a copy (or view) of the current board, already oriented with respect to the value network's perspective (White always "at the bottom").

  `current_player(self) -> int` Returns the player to move (+1 White, -1 Black).

  `get_legal_moves(self, player: int) -> List[Move]` Computes and returns the list of legal moves for the specified player (usually `self._player`). The `Move` class encapsulates:
  - starting square (row, column),
  - destination square (row, column),
  - possible pawn promotion,
  - flags for castling or en passant capture.

  `next_state(self, move: Move) -> MiniChessState` Applies the move to the current state and returns a new `MiniChessState` object updated. Internally updates:
  - `_board` by modifying piece placements,
  - `_player` (flips sign),
  - `_turns` incremented by 1,
  - `_repetition_count` checking if the new configuration has been repeated.

`is_terminal(self) -> bool` Checks if the position is terminal (checkmate, draw by repetition, draw by stalemate, maximum turn limit reached).

- **Design Notes**:
  - The class is immutable once initialized: every `apply_move` generates a new object.
  - Internally, utility functions (e.g., castling or promotion checks) are partly delegated to `GardnerMiniChessLogic.py`; the part in `minichess_state.py` is responsible for combining that information into a form suitable for the MCTS engine and the value network.

3.4.2 mcts_pt.py

This module implements a PUCT-based Monte Carlo Tree Search with a deep value network.

Data Structures: `_Node`

The `_Node` class (annotated with `@dataclass`) represents a tree node:

- `state: MiniChessState` — game state at this node.
- `parent: Optional[_Node]` — reference to parent node.
- `move: Optional[Move]` — move applied to reach this node.
- `wins: float` — cumulative value from simulations.
- `visits: int` — number of visits.
- `children: List[_Node]` — expanded child nodes.
- `untried_moves: List[Move]` — legal moves not yet explored.

Core Class: `MCTS`

`search(self, root_state)` $\rightarrow$ `Move` Executes `num_sims` iterations of the MCTS loop:

1. Initialize root: `root = _Node(root_state, None, None)`.
2. Add Dirichlet noise for exploration (AlphaZero-style): sample $\eta \sim \text{Dirichlet}(\alpha)$ over legal moves at the root and mix with network priors P_{net} using $P_{\text{root}} = (1-\varepsilon)P_{\text{net}} + \varepsilon\,\eta$.
3. Repeat `iterations_MCTS` times:
   (a) *Selection*: descend with `_select()` until a leaf with untried moves.
   (b) *Expansion*: pick one move from `leaf.untried_moves`, apply it, and add a child node for the new state.
   (c) *Evaluation*: use the deep value network to estimate the value of the new node.
   (d) *Backpropagation*: propagate the value up the tree, updating wins and visits counts.

3.4.3 Operation Pipeline of `mcts_pt.py`

This section rigorously describes the control flow in `mcts_pt.py`, showing how the selection, expansion, evaluation, and backpropagation phases are managed for each simulation, up to the final choice of the best move from the root node.

1. **Initial call**: The agent invokes

   ```
   mcts = MCTS(...)
   ```

   Here `root_state` is a `MiniChessState` object representing the current board position.

2. **Encoding of the root state**:

   - Inside `search()`, the root node is created:

     ```
     root = _Node(root_state, None, None).
     ```

3. **Simulation loop (for `num_sims` iterations)**:

   (a) **Selection** (`_select(root)`):

   - Start from `node = root` and recursively descend following the child that maximizes:

$$U(s,a) = \frac{\text{wins}_{s,a}}{\text{visits}_{s,a}} + c_{\text{puct}} \sqrt{\frac{\ln(\text{node.visits})}{\text{visits}_{s,a}}}.$$

   (b) **Expansion**:

   - Extract an untried move:
   - Apply `move` to `leaf.state`:

$$\texttt{next\_state} = \texttt{leaf.state.next\_state(move)}.$$

   - Create the new child node:

```
child = _Node(next_state, leaf, move)
```

   and add `child` to `leaf.children`.

   (c) **Evaluation / Rollout**:

   i. Invoke

$$\texttt{tensor} = \texttt{encode\_state\_as\_tensor(child.state)}.$$

   ii. Compute the value estimate with the network:

$$\texttt{value} = \texttt{value\_net(tensor).item()}.$$

   (d) **Backpropagation** (`backpropagate(child, value)`):

   - Starting from `node = child`, move up to the root, updating for each node `n` in the path:

$$\texttt{n.visits} \mathrel{+}= 1, \quad \texttt{n.wins} \mathrel{+}= \begin{cases} \texttt{value}, & \text{if } \textit{current\_player}() = \textit{player}, \\ -\texttt{value}, & \text{otherwise.} \end{cases}$$

   - The sign of `value` is flipped alternately as we ascend the tree, because the player to move alternates.

4. **Selecting the best move**: If `temperature` == 0 (as in arena), selection is deterministic based on the ratio of wins to visits; otherwise, if `temperature` != 0, a stochastic component is included to encourage exploration.

3.4.4 value_network.py

This module defines the architecture of the *Value Network*, used both during self-play for MCTS node evaluations and during supervised training.

1. Input tensor layout (12 × 5 × 5)

The network receives a *12-channel binary tensor* of shape $C = 12$, $H = W = 5$ that encodes the current Gardner Mini-Chess position from the *point of view of the side to move.*

Channel index	Piece type	Colour (relative)
0 – 5	Pawn, Knight, Bishop, Rook, Queen, King	*Friendly* (player to move)
6 – 11	Pawn, Knight, Bishop, Rook, Queen, King	*Enemy*

Table 3.1: Mapping from channel index to piece type and relative colour

The tensor is populated by the function `encode_state_as_tensor` according to:

$$\text{idx} = \text{PIECE\_TO\_IDX}[|p|] \quad \text{(gives } 0 \ldots 5 \text{ for the six piece kinds)},$$

$$\text{off} = \begin{cases} 0, & \text{if } p \times \text{mover} > 0 \quad \text{(same colour as mover)}, \\ 6, & \text{otherwise} \quad \text{(enemy)}. \end{cases}$$

$$t[\text{idx} + \text{off}, \, r, \, c] = 1.0 \quad \text{(one-hot per square)}$$

Because board planes are always oriented so that *friendly pieces appear in the lower half*, every position is presented in a canonical reference frame. The network therefore *never has to learn colour- or orientation-specific filters*, and weights can be fully re-used for both players. In summary,

$$\text{INPUT\_CHANNELS} = N_{\text{TYPES}} \times 2 = 6 \times 2 = 12.$$

2. Convolutional backbone

The `ValueNetwork` is a deep, purely convolutional encoder followed by a single fully-connected head:

```
Conv2d( 12 →  256, 3×3, padding=1)  + BatchNorm + LeakyReLU
Conv2d(256 →  512, 3×3, padding=1)  + BatchNorm + LeakyReLU
Conv2d(512 → 1024, 3×3, padding=1)  + BatchNorm + LeakyReLU
Flatten → Linear(1024·5·5 → 1)
```

- **Kernel size** 3×3 with unit padding preserves the 5×5 spatial resolution through all three layers, allowing the network to reason about local tactics while retaining global board context until the final flatten.
- **LeakyReLU** (slope = 0.01) avoids dead activations while maintaining low computational cost.
- **BatchNorm** on every convolution stabilizes training across self-play cycles.
- With HIDDEN_CHANNELS = 256, the model has approximately 5.9 million learnable parameters, balancing expressive power with fast inference during Monte-Carlo Tree Search (MCTS) rollouts.

3. Design rationale for the channel scheme

1. **Colour symmetry.** By separating friendly and enemy planes, but always numbering the current player's pieces in channels 0−5, identical filters can detect patterns such as "my pawn shield" or "opponent rook on open file" regardless of whether evaluating for White or Black. This effectively *doubles* the amount of training data per parameter.
2. **Type-specific processing.** Distinct planes for each piece type enable early convolutions to specialize (e.g., knight-fork motifs vs. bishop diagonals), while later layers can integrate information across types.
3. **Binary one-hot planes.** Using 0/1 features (instead of signed or scalar encodings) simplifies gradient flow and avoids biasing the network toward material counting. The value head learns strategic evaluation implicitly from self-play rewards.
4. **Small, fixed board.** Gardner Mini-Chess uses a 5×5 board. Keeping full resolution until the dense head means no information is lost via pooling, which is critical when a single square's occupancy can decide the game.

4. Output

The network outputs a single scalar $v \in R$, interpreted as the *expected discounted return* for the side to move. No tanh activation is applied inside the forward pass (the training script normalizes targets instead), leaving the value range unconstrained and letting the loss function enforce suitable scaling.

Together, this input-channel design and the lightweight yet expressive convolutional stack yield a value function that can be evaluated *approximately 200 times per move* within the MCTS without becoming a runtime bottleneck, while still capturing the rich tactical patterns of 5×5 Mini-Chess.

3.4.5 train.py

The `train.py` module manages the entire self-play pipeline and supervised training of the *Value Network*. Below are the essential components and the overall flow:

- **Internal utility functions**:

  `self_play_game(value_net: ValueNetwork)` Runs a full self-play game:

  1. Initializes `state = MiniChessState()` with the standard starting position.
  2. Creates an instance of `MCTS` for each player (White and Black) using the same `value_net`.
  3. While `state` is not terminal:
     - The current player calls `mcts.search(state)` to get the best move $\hat{a}$.
     - Applies $\hat{a}$ with `state = state.next_state(`$\hat{a}$`)`.
     - Records (encode_state_as_tensor($state$), z) where z is the target value:

$$z = \begin{cases} +1, & \text{if White wins at the end,} \\ -1, & \text{if Black wins,} \\ \mathit{Reward\_draw}, & \text{draw.} \end{cases}$$

  4. At the end, it returns the list of pairs (state_tensor, z) generated during the game.

  `train_value_network:` Updates the network weights given a batch of samples:

  1. Constructs a custom `Dataset` (`ChessValueDataset`) from $\{(s_i, z_i)\}$, where each `s_i` is a tensor $(12, 5, 5)$ and $z_i \in [-5, 5]$.
  2. Initializes a `DataLoader` with a predefined `batch_size` and `shuffle=True`.
  3. For `epoch = 1 ...EPOCHS`:
     - Iterates over the `DataLoader`, fetching each batch (X, Z):
       * `with autocast():` $\hat{Z} = \mathit{value\_net}(X)$ computes predictions in mixed precision.
       * Computes the Mean Squared Error:

$$\mathcal{L} = \frac{1}{|B|} \sum_{i \in B} (\hat{z}_i - z_i)^2.$$

     - Logs the average `loss` to `training.log`.

  `main()` Overall execution plan:

  1. Instantiates `ValueNetwork()` and moves it to `DEVICE`.
  2. For `cycle = 1,...,NUM_CYCLES`:
     (a) Launches 4 parallel processes (via `torch.multiprocessing`) each running `Games_per_cycle/4` self-play games using `self_play_game(...)` and collecting lists of (s_i, z_i).
     (b) Merges the collected data into a global set $\mathcal{D} = \bigcup \{(s_i, z_i)\}$.
     (c) Splits $\mathcal{D}$ into batches and calls `train_value_network(value_net, batch_D)` for each batch.
     (d) At the end of each cycle, evaluates the *value net* in arena games against the previous version: if `curr_net` achieves at least 55% wins, updates `best_net` and continues.

- **Key Points**:
  - The module unites data generation (self-play via MCTS) and network training in a continuous cycle, according to the "*Play* $\rightarrow$ *Collect Data* $\rightarrow$ *Train* $\rightarrow$ *Evaluate*" loop.
  - The use of `GradScaler` and `autocast` reduces GPU memory usage and speeds up training.
  - Synchronization of self-play processes occurs via shared queues in memory managed by `torch.multiprocessing` allowing the program generate more games simultaneously.

Chapter 4

Issues and Solutions

During the early stages of project development, the neural network was implemented with an architecture that included two separate outputs: one for estimating the state value from White's perspective and one for Black's. However, training results were unsatisfactory: the network could not learn stably nor converge to meaningful values.

To address the problem, three distinct approaches were explored:

4.1 Hyperparameter Optimization

The first attempt involved exploring the space of configuration parameters. In particular, we varied:

- the *learning rate*;
- the number of episodes and epochs;
- the rewards associated with pieces;
- the number of MCTS simulations and other parameters in `config.py`.

Despite careful hyperparameter tuning, the network's learning was still slow and unstable, especially during the arena matches against the previous model. In the logs, while the *loss* decreased, there was no significant and consistent increase in wins.

4.2 Verification of Data Flow Correctness

The second approach involved a thorough code review to verify that:

- data collection;
- cumulative reward computation;
- propagation of the value V within MCTS;

were all implemented correctly. Since the network had two outputs, the interaction between the MCTS algorithm and the model was more complex. After careful inspection, no significant errors were found in the code. This confirmed that the issue did not lie in data collection logic or value propagation but rather in the model's difficulty in learning effectively.

4.3 Network Architecture Reformulation

In the third attempt, we decided to simplify the network architecture by exploiting the fact that the value of a zero-sum game state can be represented by a single scalar output. The new network provides a single output $V(s)$, interpreted as the value from the perspective of the player to move. To make this formulation possible, each board state is rotated so that the active player is always White. In this way, the network is trained and predicts on consistently oriented states.

This solution significantly improved training performance: the network became more stable and faster at learning, even with fewer training examples. Furthermore, comparing the results with the two-output model, we observed that the latter can learn correctly but only with a much larger number of examples and training cycles.

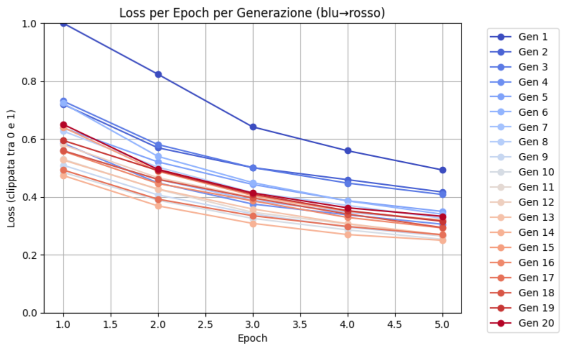
4.4 Conclusions

In summary, adopting a single-output representation with rotated boards yielded the best results in terms of learning speed and stability. This approach allowed us to simplify the network architecture and achieve better performance compared to the initial configurations.

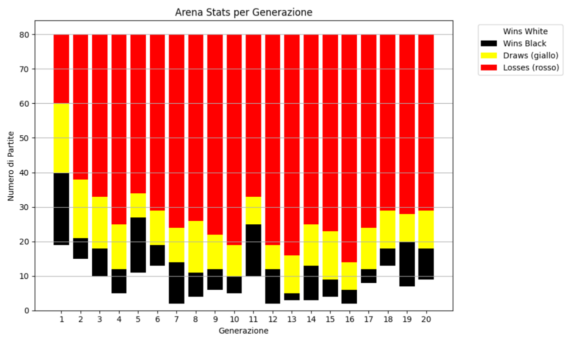
4.5 Experimental Results

This section compares four key training runs that highlight the impact of the architectural shift described in Section 3.4.4. Runs **29** and **31** use the *legacy dual-head, 13-channel* network, whereas Runs **02** and **03** employ the new *single-head, 12-channel* design with automatic board flipping.

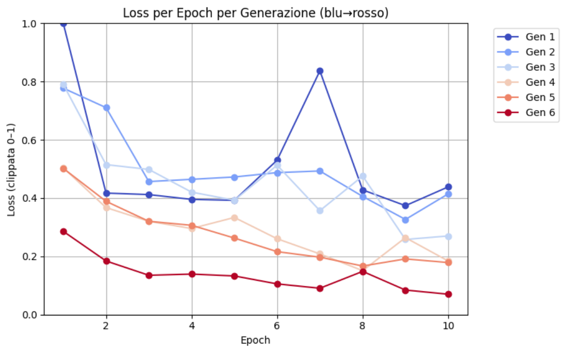
4.5.1 Legacy dual-head network (Runs 29 & 31)



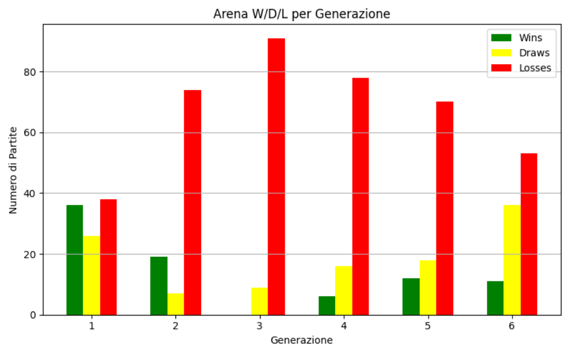
(a) Run 29 – loss per epoch



(b) Run 29 – arena W/D/L per generation



(c) Run 31 – loss per epoch



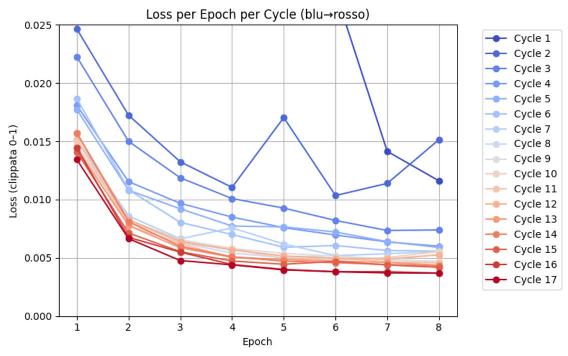
(d) Run 31 – arena W/D/L per generation

Figure 4.1: Performance of the legacy 2-output, 13-channel architecture. Loss values remain high and volatile; self-play arenas are dominated by losses (red) regardless of hyper-parameter tweaks (Run 31 versus Run 29).

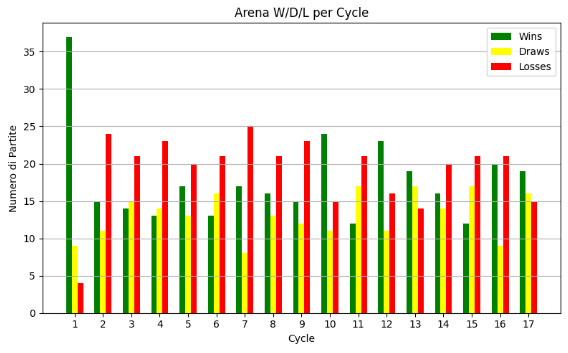
Observations.

- **Slow and erratic convergence.** Loss never falls below 0.25 and spikes re-appear after roughly five epochs (Figures 4.1a and 4.1c).
- **Collapse in self-play.** In both runs, the agent loses the vast majority of arena games (Figures 4.1b and 4.1d); draws rise but wins remain scarce.
- **Hyper-parameter changes are ineffective.** Despite a different learning rate and exploration constant in Run 31, the training dynamics hardly improve, suggesting the mismatch stems from the network design itself.

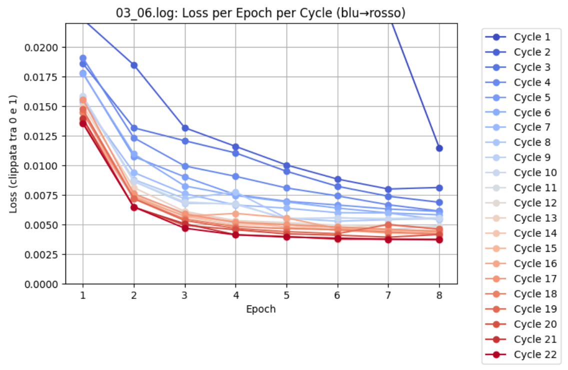
4.5.2 Single-head network with board flipping (Runs 02 & 03)



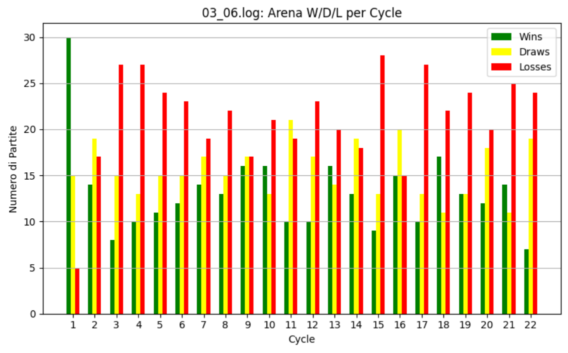
(a) Run 02 – loss per epoch



(b) Run 02 – arena W/D/L per cycle



(c) Run 03 – loss per epoch



(d) Run 03 – arena W/D/L per cycle

Figure 4.2: Performance of the new 1-output, 12-channel architecture. Loss plummets below 10^{-2} after the first epoch and stabilises; self-play quickly shifts towards wins (green) as training proceeds.

Observations.

- **Fast, smooth convergence.** After one epoch, loss is already an order of magnitude lower than the final loss in the legacy runs (Figures 4.2a and 4.2c).
- **Progressive dominance in self-play.** By cycle 10 (Run 02) and cycle 8 (Run 03), win counts overtake losses; draws also increase, indicating less erratic play (Figures 4.2b and 4.2d).
- **Effect of hyper-parameter tuning.** Run 03 uses a higher number of MCTS simulations and a slightly lower learning rate than Run 02. The result is a smoother loss curve and an earlier, more pronounced shift towards wins.

4.5.3 Key Take-aways

1. **Architectural change is decisive.** Removing the player-plane and predicting a single scalar value transformed an unstable optimisation problem into one that converges reliably within a few epochs.
2. **Hyper-parameter tuning matters *after* the fix.** Before the redesign, parameter sweeps had little effect. With the new architecture, modest tuning already yields measurable gains (Run 03 versus Run 02), suggesting ample room for further optimisation.
3. **Model can now outplay earlier checkpoints.** Runs 02 and 03 show consistent improvement in self-play, a prerequisite for Elo-based league training and eventual AlphaZero-style scaling.

Chapter 5

Conclusions and Future Work – MiniChess

The MiniChess 5×5 project demonstrated that **reinforcement learning combined with MCTS** can effectively tackle simplified yet strategically rich board games. Through iterative self-play and guided search, the agent progressively learned to recognize positional advantages and plan tactical sequences, even without any prior knowledge of the game's rules.

A key insight emerged from the architecture design: using a **single-output value network**, with board symmetry exploited to always present the position from White's perspective, led to **faster convergence and greater stability** than the traditional dual-output setup. Additionally, the **reward shaping based on material balance** proved essential to provide gradient signal during early learning stages, where full-game wins are rare and sparse.

Future Directions

To further improve performance and generality, several paths are worth exploring:

- **Full AlphaZero-style system:** adding a *policy network* to generate action priors would reduce search branching and allow for more informed exploration.
- **Deeper neural architectures** (e.g., ResNet, Transformer-style networks) could extract more abstract spatial features from the board, improving generalization and positional understanding.
- **Scaling self-play** using parallel environments or distributed GPU training would increase data throughput and speed up learning.
- **Curriculum learning** from MiniChess to more complex variants such as *Chess960* or full *8×8 Chess*, testing how well learned strategies transfer across board sizes.
- **Automated hyperparameter tuning** (e.g., via Optuna) could optimize the balance between search effort (MCTS depth) and network inference cost.
- **Game interface integration** (e.g., a web GUI or Lichess-compatible bot) would enable real-time play and public benchmarking against other engines or human players to collect more data for training.

In conclusion, this work validates the potential of self-play reinforcement learning even in small board games, laying the foundation for scalable approaches to more complex, human-level strategic tasks.