Self-Learning Chess AI

Leonardo Plazzogna

September 2025

Contents

Audience & prerequisites **3**

Introduction **3**
Overview 5

1 The Environment **6**
1.1 Design Goals & State Model 6
1.2 Move Lifecycle 8
1.3 Performance, Extensions & Tooling 10

2 MCTS Agent **13**
2.1 Overview & Design Principles 13
2.2 Core Search Mechanics 13
2.3 Performance, Caching & Robustness 17

3 Encoder **18**
3.1 Overview and Shape 18
3.2 Plane Layout and Encoding Mechanics 19
3.3 Normalization, Invariants, Contracts, and Extensibility 20

4 Policy–Value Neural Network (ResNet) **22**
4.1 Inputs & Backbone (Stem → Residual Trunk) 23
4.2 Policy & Value Heads (Shapes & Semantics) 23
4.3 Initialization, Numerics, and Training/Integration 24

5 NN Adapter **26**
5.1 Purpose and Interface 26
5.2 Feature Encoding and Move Indexing 27
5.3 Inference Paths (Policy and Value) 28
5.4 Environment Contract and Operational Considerations 29

6 Training the Network **31**
6.1 Dataset & Loading 32
6.2 Model outputs & device handling 32
6.3 Objective, optimization & checkpointing 33
6.4 Contracts 34
6.5 Efficiency & extensibility 35

7 Self-Play **36**
7.1 Orchestration and Phases 36
7.2 Performance, Parallelism, and Portability 38
7.3 Inference and Search Behavior 38
7.4 Data, Training, and Evaluation 39

8 Simulation & Results **42**
8.1 simulation Parameters 42
8.2 Results & Comments 42

9 Scripts **43**
9.1 Referenced scripts 43
9.2 Most recent scripts 43

References & Suggested Readings **44**

License & Citation **44**

Audience & prerequisites

For readers familiar with Python and basic PyTorch/Tensors. Prior knowledge of Monte-Carlo Tree Search and supervised loss functions helps, but the notebook adds gentle reminders where needed.

Introduction

Chess has long been considered as an incredibly complex game, with countless openings, intricate tactics, and an almost overwhelming number of possible moves at each turn. This complexity is beyond human capacity to fully calculate, and despite significant advancements in technology, even the most powerful computers are far from playing perfect chess. In fact, there are approximately 10^{47} possible positions in a chess game. The first engine to defeat a world champion in a classical chess match was IBM's Deep Blue, which achieved this milestone in 1997. Since then, chess engines have evolved significantly, surpassing human capabilities.

Today, two main types of chess engines dominate the landscape:

- **Traditional Engines:** Engine like Stockfish rely on brute-force game-tree search: they systematically explore huge numbers of move sequences (minimax/negamax with iterative deepening), prune provably inferior lines via alpha–beta and heuristics (move ordering, transposition tables), and score leaf positions with a handcrafted evaluation.

- **Neural Network-based Engines:** Exemplified by AlphaZero, these engines incorporate machine learning techniques, particularly deep reinforcement learning, to enhance their performance.

I find the second approach particularly fascinating because these engines learn from their own experience, much like humans do, which inspired me to develop my own AlphaZero-like engine. The goal of this project is to explore the potential of such a system and gain a deeper understanding of how it learns and adapts.

The first step in my project is to create the environment—board representation, legal-move generation, and rules—then to implement the agent and its supporting modules (MCTS, policy–value network, data pipeline), and finally to write the simulation script that orchestrates self-play and evaluation. The environment is a standard chess setup, featuring classical chess boards, pieces, and rules, offering a fully functional playing field.

Regarding the agent, I initially considered a range of approaches, from complex deep learning models to simpler rule-based strategies. Ultimately, I opted for a Monte Carlo Tree Search (MCTS) Agent. This choice strikes a balance between complexity and feasibility, allowing for a functional implementation within my current skill set.

At the core of this self-learning engine is MCTS, which facilitates decision-making by balancing exploration (trying promising but uncertain moves) and exploitation (leaning on moves already estimated to be strong)—a central trade-off in reinforcement learning. In practice, the search selects actions using a rule that favors high estimated value while granting an "uncertainty bonus" to less-visited moves, so the tree both deepens good lines and keeps discovering new ones. In my setup, exploration at the root is increased with a higher policy temperature and Dirichlet noise injected into the network priors to diversify self-play; exploitation strengthens as visit counts and value estimates concentrate on consistently good candidates. This tuning lets the agent probe broadly early on and then commit more decisively as evidence accumulates, with full details deferred to later sections.

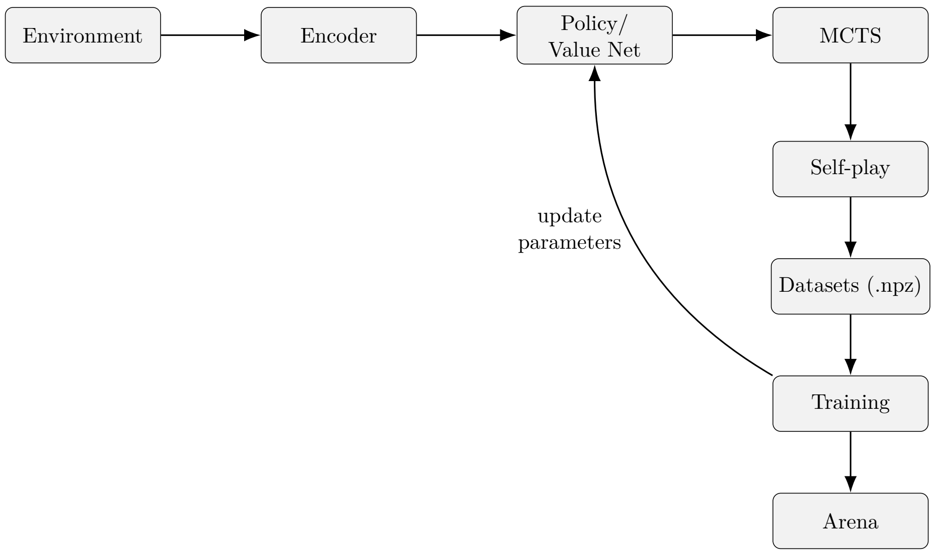
As the engine develops, the neural network refines its decision-making process. Using a deep convolutional model, the policy and value network predicts move probabilities and evaluates board positions. This model is trained through self-play, allowing the engine to learn from its mistakes and improve over time. By combining MCTS with a neural network, the engine gains the ability to search deeply into positions while continuously learning from experience—similar to how human players improve through practice.

The next phase of this project involves fine-tuning the neural network through self-play games, utilizing reinforcement learning techniques. The data generated in these cycles will be used to update the policy and value network, steadily improving the engine's performance. Though computationally intensive, this iterative learning process promises to yield a stronger, more adaptable agent.

Through this project, I aim to explore the intersection of game theory, machine learning, and computational efficiency. By developing a fully functional self-learning chess engine, I hope to enhance the engine's playing strength while gaining a deeper understanding of how machines can autonomously learn and adapt through reinforcement learning.

End-to-End System Overview

The scheme that follows illustrates the overall operation of the AI in a simplified form.



1 The Environment

1.1 Design Goals & State Model

This **chess environment** is designed with a clear focus on simplicity, search-friendliness, and ease of extension, making it ideal for both traditional engines and reinforcement-learning experiments. The guiding philosophy is centered around clarity, ensuring that all **states are explicit** and the **rules are implemented directly**, rather than through opaque caches. This approach allows for seamless integration with **tree-search** methods, such as **Monte Carlo Tree Search (MCTS)**, without introducing hidden side effects or desynchronization. In support of efficient tree search, the implementation maintains an explicit 64-bit Zobrist hash of the position and a lightweight transposition table (TT); these are transparent caches that never determine rules or legality and introduce no hidden side effects.

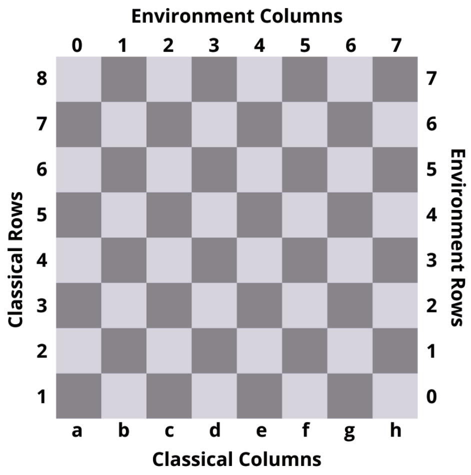


Figure 1.1(a): Board Indexing

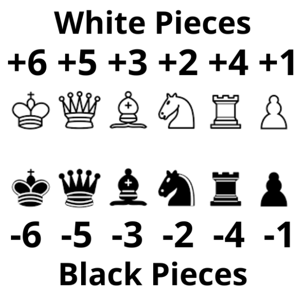


Figure 1.1(b): Piece Codes

The environment employs a straightforward board model, represented as an **8x8 integer matrix** (as shown in Figure 1.1). Each square is encoded as either zero (empty), a positive value (white pieces), or a negative value (black pieces). **Piece codes** are: pawn = 1, knight = 2, bishop = 3, rook = 4, queen = 5, king = 6. The rows are indexed from 0 (White's back rank) to 7 (Black's back rank), while the columns are arranged from the queenside to the kingside. Moves are represented as simple **four-tuples** specifying the source and destination coordinates. This compact and numerically efficient representation mirrors typical reinforcement learning tensor layouts, making the piece logic clear and predictable. For example, the bishop move f1→c4 (Figure 1.2) is (5, 0, 2, 3): from column 5, row 0 to column 2, row 3.

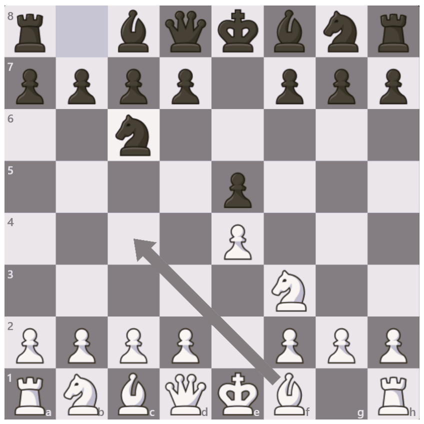


Figure 1.2: Moves Representation

In terms of **state management**, the environment maintains a variety of essential elements: the current turn, a terminal flag, the en-passant[1] square resulting from the most recent double pawn push, castling rights, a half-move clock enforcing the fifty-move rule[2], and a repetition counter based on a canonical signature of the position. The game initializes from the **standard setup**, with all castling rights available, no en-passant target, counters reset, and a strict validation pass to ensure positional consistency before play begins.

For search and deduplication, each position carries a Zobrist hash that combines: (i) a 12×64 piece–square table (P, N, B, R, Q, K for White then Black), (ii) side-to-move, (iii) castling rights (KQkq toggles), and (iv) a canonical en-passant file that is included only when a capture onto the en-passant square is actually possible for the side to move. By contrast, threefold repetition is tracked via a canonical position signature built from

[1]En-passant is a special chess move where a pawn, after moving two squares forward from its starting position, can be captured by an opponent's pawn as though it had only moved one square forward. This move must be done immediately after the two-square move.

[2]The fifty-move rule in chess states that if fifty consecutive moves are made by both players without any pawn movement or capture, either player can claim a draw.

the board array, side to move, castling rights, and the same canonical en-passant condition, aligning with rule-level equality of positions.

A key feature of the environment is its **reversible transition system**. Each move records precisely what is needed to undo it, including the pieces moved or captured, the prior en-passant square, castling rights, the half-move clock, and the rook hops during castling. Promotion[3] is automatically handled as an **auto-queen**, though this can be generalized later. The precise and reversible updates allow search algorithms to explore branches deeply without the need for deep copies, ensuring that the environment can always be returned to its prior state with bitwise accuracy. Reversible updates symmetrically maintain the position hash while recording only minimal undo facts; position-repetition counters are updated only when advancing the official game state, so search explorations do not pollute gameplay bookkeeping.

1.2 Move Lifecycle

Move generation is performed in two stages. First, the environment generates pseudo-legal moves, considering all piece rules, including special cases like double pawn pushes, en-passant captures, knight jumps, and sliding moves for bishops, rooks, and queens. Second, these candidates are filtered for legality by temporarily making each move and testing whether the moving side's king would be in check. This check ensures that only legal moves are applied, using a streamlined mechanism that works efficiently with the push/pop[4] machinery. The en-passant rule follows FIDE[5] semantics, with the en-passant square being valid for exactly one reply after a double pawn push. Both move generation and execution ensure that the captured pawn is properly removed. For simplicity, promotion is automatically handled as a queen upon reaching the last rank, but this can be adjusted for underpromotion if needed for specific curricula or agent behaviors.

[3]Pawn promotion occurs when a pawn reaches the opponent's back rank and is then converted into a queen, rook, bishop, or knight.

[4]The push/pop mechanism is a technique used in move generation and execution where the game state is temporarily modified (push) by making a move, and then reverted (pop) if the move is invalid or after checking for legality. This ensures that the environment can quickly test multiple move possibilities without permanently altering the game state.

[5]FIDE (Fédération Internationale des Échecs) is the international governing body for chess, responsible for overseeing chess competitions and establishing the official rules of the game.

Game termination is evaluated after each move. The environment checks for checkmate (if the opponent is in check and has no legal moves), stalemate (if the opponent has no legal moves and is not in check), and draws due to the fifty-move rule or threefold repetition. Special adjudication logic ensures that positions with insufficient material (such as king vs. king, or king vs. king with a knight or bishop) are recognized as draws, while any position containing pawns, rooks, or queens is considered potentially decisive.

To ensure robust king safety, **attack maps** are generated directly from the board, allowing for precise and reliable check detection. The environment has dedicated attack helpers for pawns, knights, and kings, while sliding pieces trace rays until blocked, including the first opposing blocker square. This separation of concerns ensures that legality checks remain robust and easy to reason about.

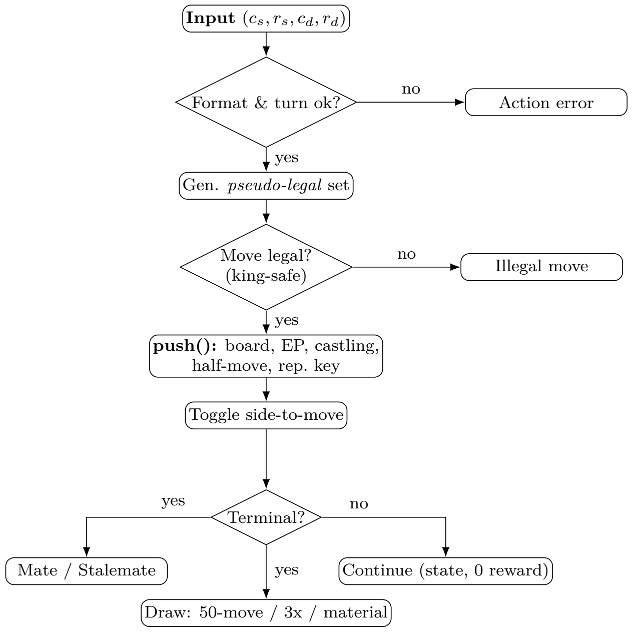


Figure 1.3: One-move lifecycle.

The environment's **step function** validates moves, ensuring that they are correctly formatted, fall within bounds, and adhere to turn ownership and legal move rules. After executing a valid move, the system updates special cases, such as en-passant, castling rights, and the half-move clock[6]. It then

[6]The half-move clock tracks the number of moves made by each player without a pawn

toggles the side to move and updates the repetition count. Nonterminal steps return the updated board with a neutral reward, while terminal outcomes return a result tag for logging and training purposes.

To maintain the integrity of the game state, a compact **exception hierarchy** ensures that early and informative failures occur when invariants are violated. Action errors cover issues such as malformed moves, out-of-bounds coordinates, or attempting to move out of turn. State errors address issues like missing or multiple kings, inconsistent castling rights, or malformed en-passant data. Validators enforce consistency for the board, side-to-move flag, and the half-move clock, ensuring the environment remains resilient during intensive search or exploratory reinforcement learning.

Rewards are neutral (0) on nonterminal steps. Terminal outcomes return +1 for checkmate (from the perspective of the side that just delivered mate) and 0 for draws. Draws (stalemate, fifty-move, threefold repetition, insufficient material) are adjudicated automatically rather than merely being claimable, which is suited to reinforcement-learning workflows.

Figure 1.3 summarizes the **one-move lifecycle**. A candidate move—specified by source and destination coordinates—is first checked for correct format and turn ownership; failures at this stage are rejected as action errors. Next, the set of pseudo-legal moves is generated and filtered by king-safety to retain only legal actions; non-passing candidates are rejected as illegal moves. When a move is accepted, the position is updated while recording just enough information to reverse it later (piece changes, en-passant target, castling rights, half-move clock, and repetition components), and the side to move is switched. The resulting position is then assessed for termination: checkmate or stalemate, or a draw by the fifty-move rule, threefold repetition, or insufficient material, yield a terminal outcome; otherwise, play proceeds with the new state and a neutral reward.

1.3 Performance, Extensions & Tooling

The environment is designed with **performance** and **extension** in mind. While it recomputes attacks on demand and eschews bitboards[7] by default,

move or capture.

[7]Bitboards are a data structure used in chess engines to represent the state of the board using bitmaps. Each piece type and color is represented as a 64-bit integer, where each bit corresponds to a square on the chessboard.

this implementation includes incremental Zobrist hashing[8] and a minimal transposition table to accelerate tree search and position deduplication; bitboards and incremental attack updates can be introduced later if higher throughput is required.

For testing and developer experience, the environment supports **FEN**[9] and **EPD**[10] import/export, enabling easy seeding of arbitrary positions for curricula, regression tests, and edge-case validation.

It also supports **SAN**[11] and **PGN**[12] formats for human-readable tracing and external tool interoperability. A **perft(depth) harness** is included for validating move generation in complex scenarios, ensuring that edge cases like en-passant, castling through check, and rook captures on home squares are handled correctly. The environment also includes a **render helper** for debugging, a string-based renderer, and an optional tensor exporter for integration with learning pipelines.

Lastly, the environment is designed with reinforcement learning in mind. The state is compact and easy to manipulate, the **push/pop** mechanism supports deep exploration without excessive memory usage, and the rules are comprehensive enough to prevent agents from developing illegal or brittle strategies.

The reward and result interface can be extended with additional features, such as a perspective-invariant terminal value or explicit winner indicator, to streamline training loops. The architecture is clear and easy to extend, allowing for straightforward modifications like under-promotion support, richer dead-position logic, and the inclusion of transposition tables. This results

[8]Zobrist hashing is a technique used in chess engines to uniquely represent board positions using a hash value. Each square on the board and each piece type is assigned a random number, and the position's hash is calculated by combining the corresponding values of all pieces on the board.

[9]FEN (Forsyth-Edwards Notation) is a standard notation used to represent a chess position.

[10]EPD (Extended Position Description) is a format used to represent chess positions, often used in chess databases and software. It is an extension of FEN, adding additional fields for position metadata such as evaluation, depth, and search statistics.

[11]SAN (Standard Algebraic Notation) is a widely used notation for recording chess moves. It specifies the piece moved, the destination square, and any captures, checks, or special moves in a concise format.

[12]PGN (Portable Game Notation) is a standard format for storing chess game data. It includes the game's moves in SAN, along with metadata such as player names, event details, and the result of the game.

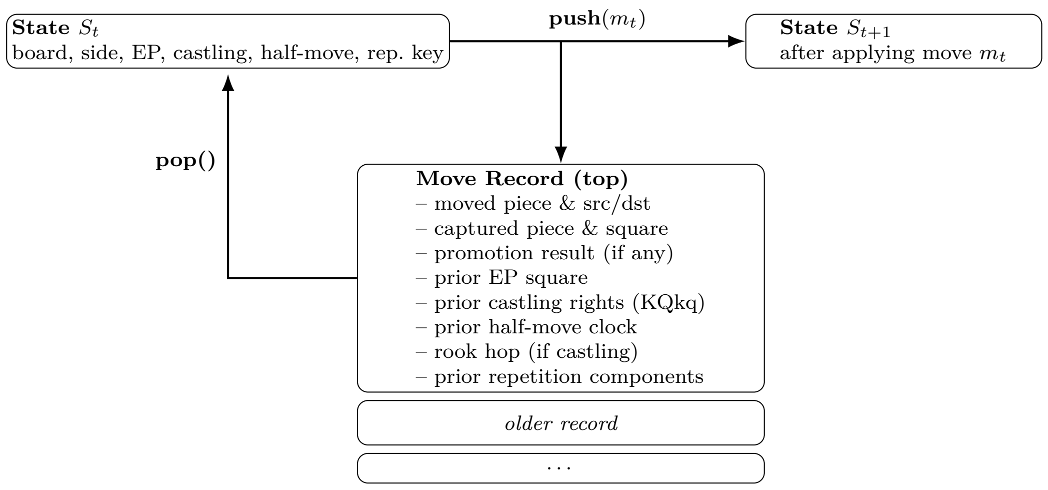


Figure 1.4: Reversible transitions without deep copies: **push** applies m_t and saves a compact record; **pop** restores S_t exactly.

in a minimal yet complete chess environment that is predictable, easy to reason about, and ready for future performance enhancements and feature expansions.

2 MCTS Agent

2.1 Overview & Design Principles

After wiring up the environment, I wanted an **agent** that could actually think over positions rather than just play the first legal move it finds. I went with an AlphaZero-style **Monte Carlo Tree Search (MCTS)** because it strikes a sweet balance: it's agnostic to how positions are evaluated (hand-crafted, classical, or neural) yet scales gracefully as the policy/value model improves. The design is deliberately modular: a thin core that reasons only in terms of legal moves, applying and undoing state changes, and evaluating positions, plus a set of controls that let me tune exploration and compute according to the game phase or training schedule. It **reuses** the search tree across moves, injects noise during training to keep play creative, processes leaf evaluations in small batches to reduce overhead, and enforces strict legality checks so it won't desynchronize from the environment when edge cases (castling, en-passant) appear. When a factored policy head is used, the agent consumes 64×64 from/delta logits with masked softmaxover legal factors; a legacy flat policy head is also supported. The agent uses an incremental Zobrist key to share priors and values across transpositions; cached priors are then renormalized to the current legal set before use.

2.2 Core Search Mechanics

At the start of each **search** the agent prepares the root. If the previous root's children remain legal and the same side is to move, it **reuses that subtree** and preserves statistics; if children become illegal or the environment has been reset, it rebuilds the root from scratch. Root priors are refreshed every move; in training mode, **Dirichlet noise**[13] is mixed into those priors to broaden exploration and avoid deterministic, repetitive self-play. A quick mate-in-one check runs at the root to opportunistically finish the search in clearly winning positions. This reuse policy is conservative by design: if any previously cached child became illegal, the root is rebuilt rather than salvaging a subset of children, avoiding subtle desynchronizations.

[13]Dirichlet noise is added during MCTS training to promote exploration by introducing randomness to the prior probabilities of moves. It is a form of noise generated from a Dirichlet distribution and is typically applied to the root node to diversify the search and prevent deterministic behavior in self-play.

Selection traverses the tree using a **PUCT[14]-style score** that balances prior belief and accumulated value. To avoid deterministic tie-breaking, children are shuffled before choosing the best candidate. Progressive widening limits how many children can be introduced until a node has enough visits, and optional **top-k/ε-tail filters**[15] focus effort when the policy assigns a long tail of tiny probabilities. At the root during training, a small repetition penalty can be applied to discourage immediate twofold repetitions. Throughout, illegal actions are excluded from consideration and removed from the priors; before making any move, the agent rechecks legality against the current set of legal moves.

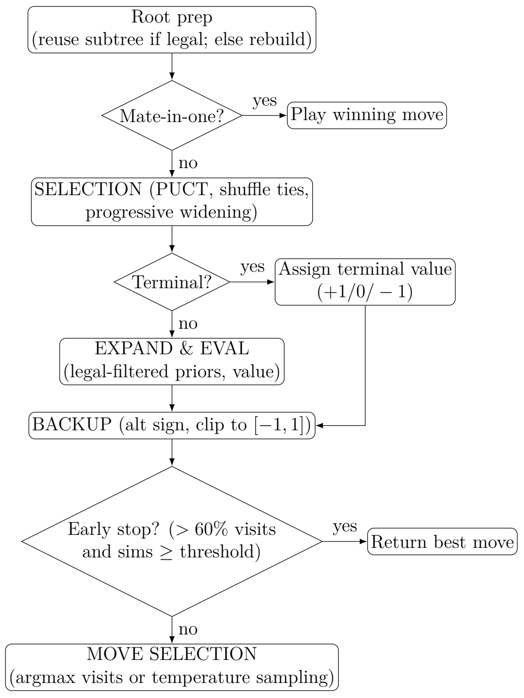

Root prep
(reuse subtree if legal; else rebuild)
Mate-in-one?
yes
Play winning move
no
SELECTION (PUCT, shuffle ties, progressive widening)
Terminal?
yes
Assign terminal value (+1/0/ − 1)
no
EXPAND & EVAL (legal-filtered priors, value)
BACKUP (alt sign, clip to [−1, 1])
Early stop? (> 60% visits and sims ≥ threshold)
yes
Return best move
no
MOVE SELECTION (argmax visits or temperature sampling)

Figure 2.1: End-to-end MCTS search flow.

[14]PUCT (Partially Observable Upper Confidence bounds applied to Trees) is an algorithm used in MCTS to balance exploration and exploitation during tree traversal. It incorporates a policy network to guide the search and modifies the traditional UCT algorithm by adjusting the value of each node based on prior probabilities.

[15]Top-k/ϵ-tail filters are techniques in MCTS used to limit the number of moves considered at each node. The topk filter restricts the search to the k most probable moves, while the -tail filter excludes moves with probabilities below a threshold ϵ, focusing computational resources on the most promising moves.

In Figure 2.1, the end-to-end MCTS flow is summarized: root preparation (with subtree reuse), a mate-in-one guard, selection via PUCT with tie shuffling and progressive widening, expansion & evaluation with legal-move filtering and transposition lookups, signed backup with clipping, an early-stop rule, and final move selection (argmax or temperature sampling).

Two simulation engines execute playouts. The **preferred engine** updates the live game state and then reverses those updates, minimizing allocations and recovering cleanly from exceptions by restoring the root state. The **fallback engine** works on a separate copy of the game state and advances step by step. Both engines share the same logic for expanding nodes, detecting terminal positions, and evaluating leaves; they differ only in how they interact with the environment, with the first offering lower overhead.

Expansion and **evaluation** follow a uniform template. Upon reaching an unexpanded node, the agent obtains the current legal moves, derives priors and a scalar value for the position, and initializes children with those priors restricted to the legal set and renormalized. With a factored head, priors come from 64×64 from/delta logits (auto-queen promotions only), combined with masked softmax over legal factors; with a flat head, a move$\rightarrow$logit map is used. If the policy output is degenerate (for example, mass assigned to non-legal moves), a uniform prior over the legal set is used. **Terminal nodes** (checkmate, stalemate, draws, or other environment-signaled endings) skip both expansion and model evaluation: they receive predefined values ($+1/-1/0$ from the root's perspective), and when there is a single forced terminal action, the prior is set as a delta on that action for consistency. Whenever a transposed position is encountered, cached priors and values are reused via the Zobrist key, with priors renormalized to the current legal set.

Backup proceeds from leaf to root, incrementing visit counts and accumulating value with alternating signs to keep the evaluation in the root player's perspective: if the node's player to move matches the root's, the value is added; otherwise it is subtracted. This enforces a consistent perspective across alternating turns and is equivalent to a signed mean over visits and rewards. All node values remain within $[-1, 1]$ after backup. Building on the flow in Figure 2.1 and the mechanics above (root prep, selection with PUCT, legal-move filtering and TT reuse, and signed backup), the pseudocode below operationalizes a **single MCTS simulation** and notes the training-time Dirichlet noise and temperature-based move choice.

One simulation in MCTS (pseudocode)

```
def mcts_simulation(root):
    node = root
    path = [node]

    # SELECT
    while node.is_expanded():
        node = argmax_child_ucb(node)     # policy priors + UCB/
    PUCT
        path.append(node)

    # EXPAND + EVALUATE
    state = node.state
    policy_logits, value = f_theta(encode(state))   # network
    call
    node.expand(legal_moves(state), policy_logits)

    # BACKUP
    for n in reversed(path):
        n.visit_count += 1
        if n.player_to_move == root.player_to_move:
            n.value_sum += value
        else:
            n.value_sum -= value
```

Dirichlet noise (root, training only): $P' = (1-\varepsilon)P + \varepsilon\,\mathrm{Dir}(\alpha)$. Typical $\varepsilon \in [0.1, 0.3]$; choose α by branching factor. Apply to legal moves and renormalize. **Temperature (move choice):** sample $\pi(a) \propto N(a)^{1/\tau}$; use higher $\tau \approx 1$ early for diversity, then lower $\tau \to 0$ (argmax) later/evaluation.

Move selection at the end of search is based on **visit counts**. By default, the move with the highest visit count is chosen, with random tie-breaking to avoid determinism. When a **temperature parameter** is active, the agent samples from a distribution proportional to the visit counts raised to the power $1/\tau$, enabling more exploration early in the game and greedier behavior later. Early-stop criteria watch the visit distribution: if one move surpasses a dominance threshold (for example, more than 60% of visits after a minimum number of simulations), the search halts and returns that move. Dynamic budgets modulate the number of simulations based on branching factor and game phase—allocating more compute to narrow, tactical positions and less to wide, quiet ones—with a mild opening discount to keep self-play brisk.

2.3 Performance, Caching & Robustness

Leaf evaluation is batched. The agent accumulates leaves up to a configurable batch size, consults a transposition table for cached priors and values, and—if the cache misses—prefers evaluating features or full positions in batches before falling back to single items. After each batch, results are cached and immediately backed up along their paths. Values are clipped to the range $[-1, 1]$ to maintain numerical stability and respect the model's calibration range.

The **transposition table**[16] is keyed by a position hash that respects side-to-move and encodes all information relevant for legality and repetition (board layout, castling rights, en-passant, and similar details). When available, **Zobrist hashing** is used (or planned) to produce fast, collision-resistant keys. Because legal move sets can change between calls, cached priors are renormalized to the current legal set. Cached values and priors reduce redundant work and accelerate repeated or transposed positions.

Robustness and guards are built in. The root is always a legal position; illegal actions are excluded from the outset. Terminal nodes are never expanded. Backup uses alternating signs along the path. The transposition table is keyed in a way that respects side-to-move. Values are clipped to $[-1, 1]$. Before any state change, moves are re-validated against the current legal set; if an illegal child is encountered (for example, due to a desynchronization or stale cache), that child is pruned and the search continues without crashing.

Performance is driven by two levers: first, the **low-allocation engine** that updates and then reverses state changes; second, consolidated model calls via **batched evaluation** (with micro-batches when accelerated hardware is present). The architecture is intentionally extensible: richer terminal detection, joint policy–value networks, parallel MCTS with virtual loss and per-thread trees, improved hashing with Zobrist keys, and exporting self-play trajectories for learning are all natural extensions.

[16]A transposition table is a caching mechanism used in search algorithms to store the results of previously evaluated positions. It allows for quick retrieval of information on previously encountered positions, avoiding redundant calculations and speeding up the search process.

3 Encoder

With the environment and search agent in place, the next piece of the puzzle is the **encoder**—the small but mighty step that turns a raw chess position into a neat stack of feature planes for the model. This is about alignment: the network expects a fixed set of channels with clear meaning, and the encoder guarantees exactly that. It performs a single pass over the board, sets global flags for side-to-move, castling, and en-passant, and yields a compact [**18,8,8**] block that mirrors the training and self-play pipelines while remaining easy to extend (e.g., history or counters) without disrupting the base layout.

The output is a single **unbatched** feature tensor of shape [**18, 8, 8**] with strictly binary base channels. Batching and device placement are intentionally handled outside this step, so the core stays portable and side-effect free. Features are assembled in a standard array format and then adapted to the model's tensor format at the boundary; this adaptation is typically **zero-copy** where supported, remaining lightweight.

3.1 Overview and Shape

The encoder emits an unbatched feature tensor of shape $\mathbf{18 \times 8 \times 8}$ with strictly binary entries for all base channels. It mirrors the self-play model's input convention so the policy/value network receives consistent planes without scattered preprocessing; batching and device placement, when needed, are handled outside. Internally, features are kept as **float32**[17] (optionally stored as **uint8**[18] for memory, then cast to **float32** at the boundary without semantic change). Temporal stacking, if used, wraps these as [**T,C,8,8**] with the same per-plane semantics. Where the array→tensor boundary supports it, the conversion preserves a shared view until moved or modified.

[17]float32 refers to a 32-bit floating-point format used to represent real numbers. It provides a balance between range and precision, commonly used in machine learning and numerical applications where memory usage and computational efficiency are important.

[18]uint8 is an unsigned 8-bit integer type, capable of representing integers in the range [0, 255]. It is often used in contexts where memory efficiency is important, such as in image processing or when storing binary feature planes in neural networks.

3.2 Plane Layout and Encoding Mechanics

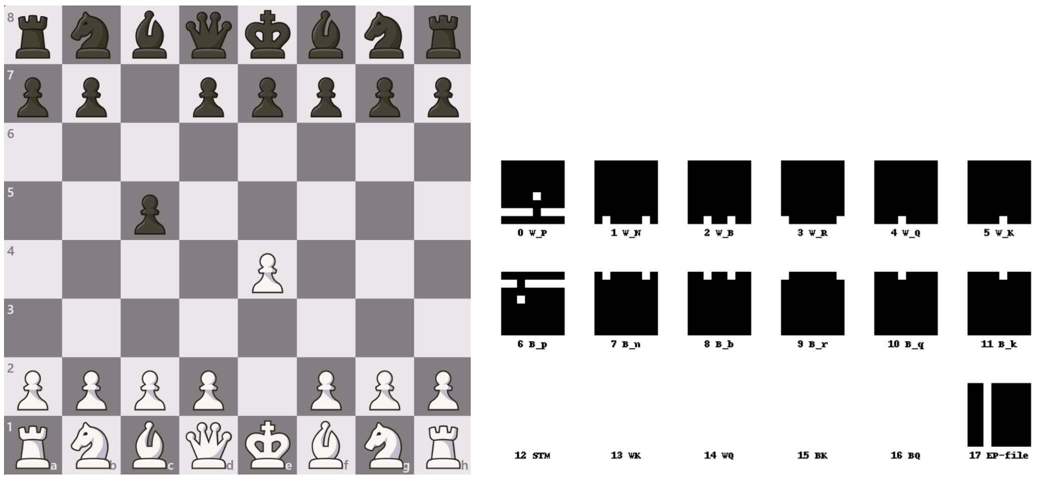
The **18 base planes** follow a fixed layout (as shown in Table 3.1 below): indices **0-5** are White piece occupancy **P,N,B,R,Q,K**, **6-11** are Black **p,n,b,r,q,k**, each as one-hot bits on the squares they occupy. **Plane 12** is the side-to-move as a full-board mask (all ones when White moves, all zeros otherwise), preserving the usual AlphaZero-style perspective without relying on parity or history. **Planes 13-16** encode castling availability—white kingside, white queenside, black kingside, black queenside—as full-board binary flags (ones when available, zeros otherwise); a compact four-tuple of rights is preferred when present, with a graceful fallback to per-side booleans. **Plane 17** encodes en-passant with a "file-hot" mask: if an EP target exists, every square in that column is set to one; otherwise the plane is zero. Optional counters, when used, are reserved in **indices 18-20** (for example, half-move clock, repetition hint, ply modulo[19]), and the optional history is stacked along a time axis as [T, C, 8,8] without changing the semantics per plane.

Table 3.1: Input planes layout

Index range	**Plane(s)**	**Description**
0–5	White piece planes	$\{P, N, B, R, Q, K\}$ one/hot
6–11	Black piece planes	$\{p, n, b, r, q, k\}$ one/hot
12	Side/to/move	1 for white, 0 for black (or vice versa)
13–16	Castling rights	WK, WQ, BK, BQ as binary planes
17	En/passant file	one/hot file indicator if applicable
18–20	Move counters	e.g., half/move clock, repetition hint, ply modulo
...	(Optional history)	past N positions stacked

Fig. 3.2 provides a concrete, visual walk-through of the encoder. Panel (a) shows the board after 1.e4 c5 (White to move, all castling rights intact, en-passant square c6). Panel (b) then depicts the encoder's response: 18 binary feature planes that disentangle this board state into machine-readable layers.

[19]Ply modulo is a term used in chess engines to track the number of half-moves (plies) in a game. The modulo operation can be used to reset or cycle the counter at certain intervals, such as a specific number of half-moves, to help manage game state or detect patterns.



(a) Chess position (board view). (b) 18-plane encoder mosaic.

Figure 3.2

Piece placement uses a small mapping from absolute piece IDs 1..6 to indices 0..5 for White and the same mapping with a +6 offset for Black; during a single scan of the 64 squares, each nonzero signed entry (positive for White, negative for Black) sets exactly one bit in its corresponding plane, while empty cells are skipped. Optional extras (beyond the base 18) can include move counters (e.g., half-move clock, repetition hint, ply modulo) or history windows; when present, counters are appended as additional planes and history is stacked along a time axis.

3.3 Normalization, Invariants, Contracts, and Extensibility

All base channels are binary one-hot indicators of presence, availability, or eligibility; there is no scaling, counts, or distances in the core design. Optional counters are normalized into **[0,1]** (e.g.,**min(counter,cap)/cap)**[20] to keep magnitudes consistent with binary planes. The key invariant is per-square exclusivity across piece planes: for any square, at most one piece bit is 1. The

[20]min(counter, cap)/cap is a normalization technique used to scale a counter value. The counter is first limited to a maximum value (cap), and then the ratio is computed by dividing the counter by the cap. This ensures that the counter is represented as a value between 0 and 1, making it consistent with binary feature planes in the model.

environment contract is simple and explicit: the board is an **8×8 grid** of signed integers with $|\cdot| \in \boldsymbol{\{1,\ldots,6\}}$; the side-to-move indicator uses +1 for White and a different value (typically -1) for Black; any **EP** file, when present, is a valid column index; castling rights are exposed either as a four-tuple or via per-side booleans. Error handling in the hot path is minimal by intent: zeros are skipped; unknown piece IDs are not expected; if the castling tuple is missing, per-right booleans are used; there is no explicit bounds check on the **EP file** index, trusting upstream validators[21] for consistency. **Dtypes** are kept simple: features are float32 end-to-end; memory-sensitive builds may store planes as uint8 internally and cast to float32 at the **array→tensor** boundary without semantic change.

Performance is **cache-friendly** and negligible relative to search or a forward pass: one linear scan of 64 squares plus constant-time writes for the global planes. The scheme is deliberately extensible—additional planes (e.g., recent-capture markers, a one-hot EP square, or counters) can be appended without disturbing the first 18; most importantly, adherence to the 18-plane convention keeps training, inference, and self-play aligned and prevents off-by-one-channel errors.

[21]Upstream validators refer to checks or validation steps that occur earlier in the processing pipeline, before the data reaches the encoder. These validators ensure that the input data is consistent and valid, such as checking for legal move conditions or ensuring that the game state adheres to the rules.

4 Policy–Value Neural Network (ResNet)

Once the board is encoded and the search is humming, the last pillar is the **model** that turns raw planes into chess intuition. This **policy–value network** follows the AlphaZero tradition: a slim **ResNet**[22] built only from standard **Conv[23]/BN[24]/ReLU[25]/Linear components** and sized to scale cleanly by increasing channel width and block depth.

In Figure 4.1, an intuitive scheme represents how the policy–value neural network operates. Encoded positions **[B,18,8,8]** first pass through a $\mathbf{3 \times 3}$ **stem** (conv + normalization + ReLU) and then a constant-width **residual trunk**[26] that preserves the $\mathbf{8 \times 8}$ spatial structure (width C, n post-activation blocks). The backbone then branches: the policy head compresses channels, flattens to **1024**, and maps to **4096** raw logits over the flat $\mathbf{64 \times 64}$ move space; legality masking and probability normalization are applied outside the network. In parallel, the value head compresses, flattens to **1024**, and uses a small **MLP** with final tanh to produce a bounded scalar $[\boldsymbol{B}, \mathbf{1}] \in [\mathbf{-1}, \mathbf{1}]$ interpreted from the side to move. This layout highlights the clean split between shared spatial features and the two task-specific outputs.

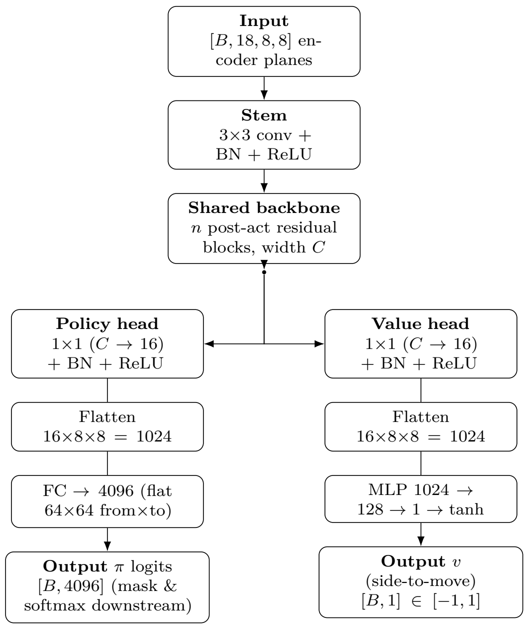
[22]ResNet (Residual Network) is a type of deep neural network architecture that includes residual connections, which allow gradients to flow more easily through the network. These connections skip one or more layers, helping to alleviate the vanishing gradient problem and enabling the training of much deeper networks.

[23]Conv (Convolutional layer) is a neural network layer that applies a filter (kernel) over the input data to detect patterns such as edges or textures. In the context of a chess model, it helps process spatial data, like piece arrangements on the board, by sliding the kernel across the input feature planes.

[24]BN (Batch Normalization) is a technique to normalize the activations of neurons in a neural network by adjusting the mean and variance of the input layer during training. It accelerates training and improves performance by reducing the risk of vanishing/exploding gradients.

[25]ReLU (Rectified Linear Unit) is an activation function commonly used in neural networks, defined as $f(x) = \max(0, x)$. It introduces non-linearity while being computationally efficient and mitigating the vanishing gradient problem in deep networks.

[26]The residual trunk refers to the sequence of residual blocks in a ResNet, where each block consists of two convolutions with skip connections. These blocks allow the network to learn an identity function, which helps in training deeper models by enabling easier gradient flow.



Input
$[B, 18, 8, 8]$ encoder planes
Stem
3×3 conv + BN + ReLU
Shared backbone
n post-act residual blocks, width C
Policy head
1×1 ($C \to 16$) + BN + ReLU
Value head
1×1 ($C \to 16$) + BN + ReLU
Flatten 16×8×8 = 1024
Flatten 16×8×8 = 1024
FC → 4096 (flat 64×64 from×to)
MLP 1024 → 128 → 1 → tanh
Output π logits $[B, 4096]$ (mask & softmax downstream)
Output v (side-to-move) $[B, 1] \in [-1, 1]$

Figure 4.1

4.1 Inputs & Backbone (Stem → Residual Trunk)

The network ingests batches of encoded positions [**B, 18, 8, 8**] and lifts them with a **3×3 convolution** (padding 1) followed by normalization and ReLU. A constant-width **residual trunk** (C default 64) with n post-activation blocks (default 6) balances capacity and speed on the **8×8 grid**. Each block applies two 3×3 convolutions without biases, each followed by normalization; **ReLU** follows the first normalization, the skip connection is added after the second, and a final ReLU closes the block. No projection is needed because channel dimensions remain constant throughout the trunk, yielding a shared backbone of shape [**B, C, 8, 8**].

4.2 Policy & Value Heads (Shapes & Semantics)

From the shared features, the **policy head** is parameterized in a factored form: a from-square head produces 64 logits and, conditioned on the from-square, a to-square head produces 64 logits, yielding tensors [**B, 64**] and [**B, 64, 64**]; joint **64×64** logits are formed downstream as a sum of from and conditional to logits; legality masking and probability normalization are applied outside the network.

The **value head** mirrors the early structure (1×1 to 16 + normalization + ReLU), flattens to **1024**, then uses a small **MLP**[27] (1024 → 128 → 1) with ReLU between layers and a final tanh[28] to produce $[\boldsymbol{B}, \mathbf{1}] \in [-\mathbf{1}, \mathbf{1}]$, representing the side-to-move evaluation (positive favors that side, negative disfavors it).

To summarize these two heads at a glance, Table 4.2 condenses their outputs and key operations.

Table 4.2: Heads: outputs and brief ops.

Head	**Output**	**Ops & Notes**
Policy	$[B, 4096]$	1×1 Conv → Norm → ReLU → Flatten → Linear. Logits are unnormalized; legality mask & normalization happen downstream.
Value	$[B, 1] \in [-1, 1]$	1×1 Conv → Norm → ReLU → Flatten → MLP (1024→128→1) → tanh.

4.3 Initialization, Numerics, and Training/Integration

Weights use He/Kaiming[29] initialization suited to ReLU activations (normal for convolutions; uniform for linear layers with a small nonlinearity parameter), and convolutions omit biases where followed by normalization; final logit 1×1 layers retain bias (no BN follows).

The model runs robustly in **FP32**[30] and supports mixed precision; nor-

[27]MLP (Multilayer Perceptron) is a type of fully connected neural network that consists of multiple layers of neurons, where each neuron is connected to every neuron in the next layer. It is typically used for tasks like regression or classification, where each layer learns progressively more complex features.

[28]tanh (hyperbolic tangent) is an activation function that maps input values to a range between -1 and 1. It is used in neural networks to introduce non-linearity and is particularly useful when the model requires outputs with a centered range.

[29]He/Kaiming initialization is a method for initializing the weights in deep neural networks. It is specifically designed for ReLU activation functions and helps maintain the variance of activations throughout the network, making training more stable and improving convergence.

[30]FP32 (32-bit floating-point) refers to a data type used to represent real numbers with 32 bits of precision. It is widely used in machine learning models for storing weights and

malization layers behave appropriately in training versus inference modes.

The design stays lean—no attention, squeeze–excitation, dilations, or dropout—keeping compute light for inner-loop MCTS and self-play.

Capacity scales via C (the number of channels of the neural network, determining the depth of the feature maps) and n (the number of post-activation blocks in the residual trunk, controlling the depth of the network) without changing the rest of the pipeline; the total parameter count grows accordingly and can be queried at runtime if needed.

Typical training pairs **cross-entropy** (or KL)[31] on the policy against the root-visit distribution with **MSE** (or Huber[32]) on the value against outcomes in **[-1, 1]**. With the factored policy, the joint CE equals CE on the from-square marginal plus the expected CE on the to-square conditional; targets are formed by marginalizing and conditioning the MCTS root-visit distribution. The clean encoder $\rightarrow$ network $\rightarrow$ agent split (input [B, 18, 8, 8] $\rightarrow$ backbone [B, C, 8, 8] $\rightarrow$ heads [B, 4096], [B, 1]) reduces shape bugs, leaves legality handling to downstream components, and makes width/depth scaling straightforward.

activations due to its balance between precision and memory requirements.

[31]Cross-entropy and KL (Kullback-Leibler) divergence are loss functions used in training neural networks. Cross-entropy measures the difference between the predicted probability distribution and the true distribution, while KL divergence quantifies the difference between two probability distributions, often used to guide training in generative models.

[32]Huber loss is a variant of MSE that combines the benefits of MSE and absolute error, being less sensitive to outliers by applying a quadratic penalty for small errors and a linear penalty for large ones.

5 NN Adapter

With the encoder producing clean feature planes and the Policy–Value network ready to score positions, the last connector is the **NN adapter**—a thin wrapper that speaks both languages. Its job is pragmatic: take a live environment, build the exact **[18, 8, 8] tensor** the model expects, run a forward pass, and convert the outputs into objects the MCTS agent can consume. It hides device management, evaluation mode, and minor schema differences across environments, and it returns just what search needs: **unnormalized policy scores** for legal moves and a perspective-correct scalar value for the side to move. The result is a dependable, dependency-light bridge that keeps your training and self-play code crisp.

Figure 5.1 summarizes the end-to-end **inference path**: the environment feeds the 18×8×8 encoder planes to the NN adapter, which runs a single forward pass on the Policy–Value net and returns (i) legal-only, unnormalized policy scores and (ii) a side-to-move value in [-1,1]. The boxed note beneath the adapter highlights legality masking and the "exp+ϵ" step, with normalization deliberately deferred to MCTS.

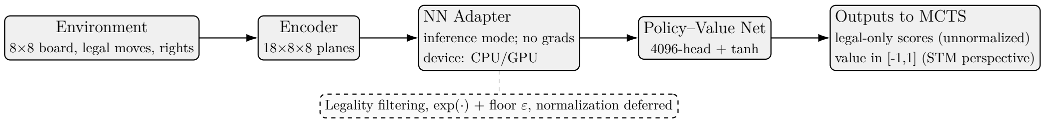

Environment
8×8 board, legal moves, rights
Encoder
18×8×8 planes
NN Adapter
inference mode; no grads
device: CPU/GPU
Policy–Value Net
4096-head + tanh
Outputs to MCTS
legal-only scores (unnormalized)
value in [-1,1] (STM perspective)
Legality filtering, exp(·) + floor ε, normalization deferred

Figure 5.1: End-to-end inference path

5.1 Purpose and Interface

The **NN adapter** is a thin, dependable bridge between a live chess environment and the Policy–Value network. It takes the current position, assembles the exact **18×8×8 feature tensor** the model expects, performs a single forward pass, and hands back precisely what search needs: **unnormalized policy scores** for the currently legal moves and a **scalar value** that is always "from the side to move." It hides device placement, runs the network strictly in inference mode with gradient tracking[33] disabled to keep memory and latency

[33]Gradient tracking refers to the process of recording and storing gradients during the backpropagation step of training a neural network. This allows the model to update its weights in response to the error gradient, ensuring that it learns from the training data over time.

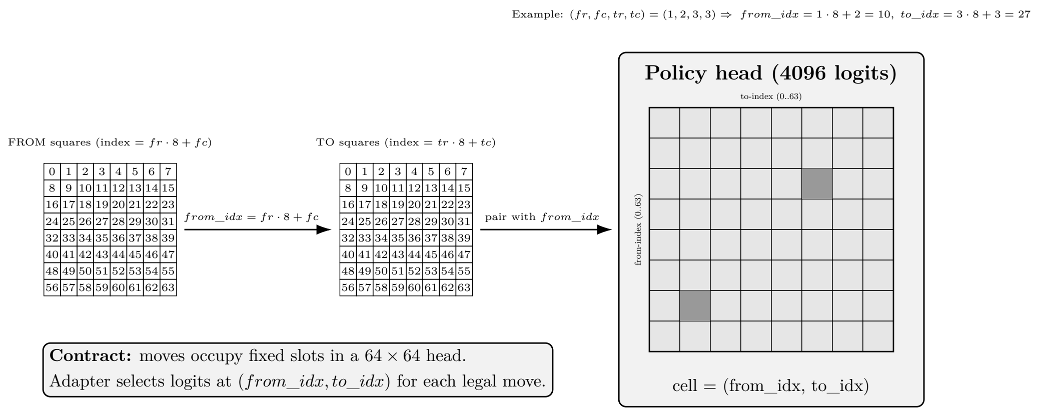
predictable, smooths over minor schema differences across environments, and keeps dependencies light so the rest of the training and self-play stack stays crisp.

Conceptually, the adapter exposes two results per position. First, a **mapping** from legal move tuples (fr, fc, tr, tc) to floating-point scores that serve as priors. Second, a single **evaluation** in the interval **[-1, 1]** representing the position from any requested perspective; taken "as is" for White's point of view and sign-flipped for Black, this guarantees a consistent convention for expansion and leaf evaluation in MCTS. The network is placed on the requested compute device (CPU by default, GPU/CUDA[34] if specified) and kept in stable inference mode so batch-normalization statistics are fixed during evaluation.

5.2 Feature Encoding and Move Indexing

Before every inference, the adapter builds the **18-plane** encoding directly from the environment. **Planes 0–5** contain White's piece occupancies (P, N, B, R, Q, K); **planes 6–11** mirror the same for Black. **Plane 12** encodes the side to move as an all-ones board for White and zeros otherwise. **Planes 13–16** replicate castling rights across all squares in the order (wK, wQ, bK, bQ), accepting either a compact "rights" tuple or a set of individual booleans (e.g., white/black $\times$ king/queen side). **Plane 17** marks the en-passant file as a column of ones when such a file exists; if no en-passant square is available, this plane remains all zeros. Empty squares are skipped during population, and the adapter assumes the board uses signed piece IDs with absolute values in 1..6.

[34]CUDA (Compute Unified Device Architecture) is a parallel computing platform and API model developed by NVIDIA. It enables developers to utilize the power of GPUs (Graphics Processing Units) for general-purpose computing tasks, significantly accelerating operations like matrix multiplications and training deep neural networks.



Example: $(fr, fc, tr, tc) = (1, 2, 3, 3) \Rightarrow$ $from\_idx = 1 \cdot 8 + 2 = 10$, $to\_idx = 3 \cdot 8 + 3 = 27$
FROM squares (index $= fr \cdot 8 + fc$)
$from\_idx = fr \cdot 8 + fc$
TO squares (index $= tr \cdot 8 + tc$)
pair with $from\_idx$
Policy head (4096 logits)
to-index (0..63)
from-index (0..63)
cell = (from_idx, to_idx)
Contract: moves occupy fixed slots in a 64×64 head.
Adapter selects logits at $(from\_idx, to\_idx)$ for each legal move.

Figure 5.2: Move Indexing Map

Moves are indexed in a flat **64×64 policy space**. As shown in Figure 5.2, a move **(fr,fc,tr,tc)** maps to a unique index by first computing the from-square $\boldsymbol{fr \times 8 + fc}$ and the to-square $\boldsymbol{tr \times 8 + tc}$, then composing those into a stable slot in the **4,096-dimensional policy head**. This contract ensures the adapter picks exactly the logits corresponding to the environment's legal actions. For visual clarity, Figure 5.2 depicts an 8×8 coarse grid as a proxy for the true 64×64 lattice; each shown cell stands in for a block of indices, while the network actually emits one logit per (from_idx,to_idx) pair across all 4,096 entries.

5.3 Inference Paths (Policy and Value)

On the **policy path**, the adapter performs a single-position inference, reads the 4,096-length logit vector, filters it down to the legal moves reported by the environment, exponentiates each selected logit to obtain a positive score, and applies a tiny floor to avoid zeros. Normalization is intentionally deferred: returning legal-only, unnormalized scores lets downstream search normalize over the current legal set and mix in exploration noise without double-normalizing.

The **policy path** takes the full 64×64 logit head, gathers only the entries at legal indices $\mathcal{L}$, exponentiates each selected logit to obtain positive scores $s_i = \exp(\ell_i)$, and applies a tiny floor $s_i \leftarrow \max\{s_i, \varepsilon\}$ to avoid zeros; normalization is deliberately deferred so MCTS can normalize over the current legal set and mix exploration noise without double-normalizing.

On the **value path**, the same single-position inference yields a scalar already bounded in [-1, 1] (via a final tanh). Because evaluation is defined from the side to move, the adapter returns this scalar directly for White and with opposite sign for Black, making the convention explicit and reliable for the search code.

5.4 Environment Contract and Operational Considerations

The environment contract is deliberately minimal: an 8×8 board of signed integers; a flag for the current player (+1 for White); a way to list legal moves as (fr, fc, tr, tc) tuples; and fields that communicate en-passant and castling rights in one of the supported forms. With these in place, integration is effectively plug-and-play. If the environment reports no legal moves—or cannot produce them—the adapter simply returns an empty mapping for the policy, allowing the caller to treat the position as terminal. The en-passant plane is populated only when a valid file index exists; otherwise it is safely ignored.

Simplicity is a design choice: there is no batching[35], caching[36], or central inference queue[37]. Each call performs a compact **array-to-tensor** conversion, one forward pass, and light post-processing. This keeps behavior easy to profile and avoids surprising interactions with any micro-batching the agent might implement elsewhere. The adapter assumes the model emits a full 4,096-dimensional policy head aligned with the 64×64 indexing and that these outputs are logits; any normalization and legality masking beyond the initial legal-move filtering is the responsibility of downstream components. It also assumes the side-to-move channel follows the shared +1/-1 convention across

[35]Batching refers to the process of grouping multiple data samples together into a single batch for processing during training or inference. This allows for more efficient use of computational resources and often leads to faster convergence in training due to better utilization of vectorized operations.

[36]Caching involves storing intermediate results of computations in a memory store to avoid redundant recalculations. In neural network models, caching can be used to store computed activations or feature maps, speeding up subsequent accesses and improving performance during both training and inference.

[37]A central inference queue is a system architecture pattern where incoming inference requests are placed in a queue and processed by available computational resources. This setup can help efficiently manage workload distribution across multiple workers, improving throughput and scalability in large systems.

the stack.

Performance per call is dominated by the single forward pass on a very small 8×8 input; the remaining work—array ops and assembling the move-score mapping—is lightweight. When needed, extending the design is straightforward: one can switch the policy path to a softmax over legal moves to return normalized priors, add **FP16**[38] or automatic mixed precision for GPUs, or introduce batching when many evaluations are requested at once. Because the encoding mirrors the self-play feature planes, these variations remain drop-in compatible as you extend the feature set or scale the network.

[38]FP16 (16-bit floating-point) is a numerical data type that uses 16 bits to represent real numbers. It offers reduced memory usage and faster computation compared to FP32 while sacrificing some precision.

6 Training the Network

With the environment generating positions, the agent exploring them, and the encoder–model pair turning boards into scores, we still need a simple way to learn from self-play. The **training** script loads **.npz shards**[39] (with factored policy targets), builds a plain **PyTorch**[40] Dataset/DataLoader, and runs a short **AdamW**[41] loop that optimizes **cross-entropy** on a factored policy (from-square marginal plus per-from conditional deltas) and **SmoothL1**[42] on value; it stays on CPU by default, with an option to select an accelerator and move batches per step. The focus is clarity and reproducibility: one file, cleanly separated responsibilities, and pieces that are easy to extend when you're ready to scale. A lightweight moving-average of parameters and a cosine learning-rate schedule with a brief warmup are included for stable training, while keeping the baseline compact.

Figure 6.1 shows the end-to-end training flow we use after self-play: a **forward pass** across the top row, then the **learning/update** on the bottom row. **(1)** .npz shards supply X, PF/PD (factored policy), and Z; **(2)** a deterministic loader makes mini-batches; **(3)** the policy–value net outputs a factored policy (from-square distribution and per-from delta conditionals, yielding an implied 4096-move space) and a bounded scalar value. Then **(4)** the loss combines CE(factored policy) + SmoothL1(value) with optional weight decay; **(5)** AdamW backpropagates and updates with data-weighted epoch means; **(6)** weights are saved as a state dict. CPU is the default; batches move to a chosen device inside the step; the baseline stays minimal to simplify later extensions.

[39].npz shards are compressed binary files used by NumPy to store large arrays efficiently. In the context of training, they contain policy targets, feature planes, and game outcomes, and are loaded and concatenated during training to form mini-batches.

[40]PyTorch is an open-source deep learning framework developed by Facebook's AI Research lab. It provides tools for building and training neural networks.

[41]AdamW (Adaptive Moment Estimation with decoupled weight decay) is an optimizer that keeps per-parameter adaptive learning rates via first- and second-moment gradient estimates, while applying weight decay as a separate L_2 regularizer rather than through the gradient update; this preserves the intended regularization and often improves generalization and training stability when weight decay is used.

[42]SmoothL1 (Huber) loss behaves like L_2 for small errors and like L_1 for large ones, providing stable gradients with improved robustness to outliers compared to pure MSE.

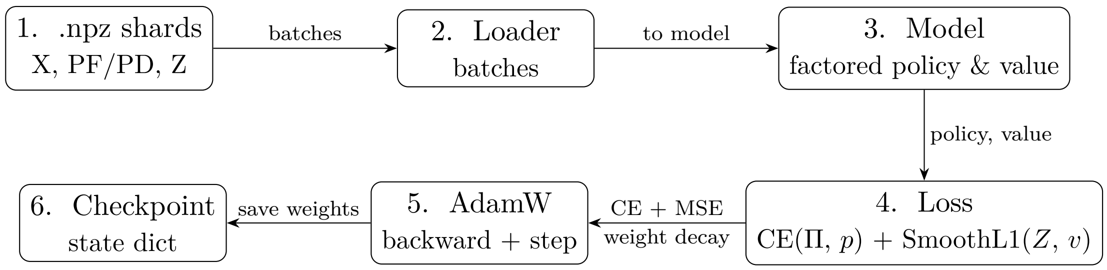


Figure 6.1: End-to-end training flow

6.1 Dataset & Loading

The training pipeline is designed as a clear, portable baseline. Training data lives in one or more `.npz` shards, each containing arrays `X`, `PF`, `PD` and `Z`. All shards are loaded and concatenated along the batch axis so that samples are exposed as triplets plus factored policy: feature planes, factored policy targets, and scalar values.

Features X use the model's [18,8,8] encoding; **policy targets** are factored: PF is a normalized from-square marginal over 64 squares, and PD provides per-from delta conditionals over 64 deltas; and **values Z** are game outcomes from the side-to-move's perspective with $Z \in [-1, 1]$, shaped to broadcast against a [B,1] value head.

A standard data loader shuffles and yields mini-batches, uses a single worker for portability, and can enable pinned host memory when an accelerator is selected to keep behavior predictable across machines. Everything resides in a single file; responsibilities are separated cleanly for easy extension. Deterministic behavior is achieved by fixing seeds and using single-worker loading.

6.2 Model outputs & device handling

A unified network produces two outputs per batch: a **policy head** and a bounded scalar **value head**. By convention, the policy is factored: the first part yields a distribution over from-squares, and the second yields conditional distributions over deltas per from-square; log-softmax is applied along the appropriate axes before computing the loss when raw logits are produced. The model remains on **CPU** by default. When a device is specified, batches are moved to it inside the training step, localizing device handling and making a later switch to hardware accelerators frictionless.

6.3 Objective, optimization & checkpointing

The objective combines a policy term, a value term, and an optional regularization term. Moves are indexed by (i, j), where $i \in \{1, \ldots, 64\}$ is the from-square and $j \in \{1, \ldots, 64\}$ is a delta conditioned on i. The policy distribution factorizes by the chain rule,

$$\pi_\theta(i,j) \;=\; p_{\text{from}}(i)\, p_{\text{delta}}(j \mid i). \tag{1}$$

Given a flat target $\Pi(i, j)$, define the marginal–conditional pair

$$\begin{aligned} PF(i) &= \sum_j \Pi(i,j), \\ PD(j \mid i) &= \begin{cases} \Pi(i,j)/PF(i), & PF(i) > 0, \\ \text{any prob. row}, & PF(i) = 0\ , \end{cases} \end{aligned} \tag{2}$$

so that $\Pi(i, j) = PF(i)\, PD(j \mid i)$, with PF summing to 1 and each row $PD(\cdot \mid i)$ summing to 1. The flat 4096-way cross-entropy then rearranges exactly into two terms:

$$\begin{aligned} -\sum_{i,j} \Pi(i,j) \log \pi_\theta(i,j) = &-\sum_i PF(i) \log p_{\text{from}}(i) \\ &-\sum_i PF(i) \sum_j PD(j \mid i) \log p_{\text{delta}}(j \mid i). \end{aligned} \tag{3}$$

In words, the factored policy loss is a cross-entropy on the from-square marginal plus a PF-weighted sum of cross-entropies on the per-from conditional deltas, yielding an implied distribution over the full move space. For value regression, with targets $Z \in [-1, 1]$ and prediction $v_\theta(X)$, we use the SmoothL1 (Huber) penalty

$$\rho_\delta(e) \;=\; \begin{cases} \frac{1}{2}e^2, & |e| \le \delta, \\ \delta\Big(|e| - \frac{1}{2}\delta\Big), & |e| > \delta, \end{cases} \qquad e \;=\; v_\theta(X) - Z, \tag{4}$$

and the total objective

$$L \;=\; L_{\text{policy}} \;+\; \lambda_v\, \mathbb{E}\Big[\rho_\delta\Big(v_\theta(X) - Z\Big)\Big]\,. \tag{5}$$

A small weight-decay term helps generalization; in practice we apply decoupled L_2 weight decay directly in the parameter update as an additive shrinkage,

$$\theta_{t+1} \;=\; \theta_t \;-\; \eta\, g_t \;-\; \eta\, \lambda_{\mathrm{wd}}\, \theta_t, \tag{6}$$

where g_t is the adaptive step from first- and second-moment estimates. Optionally, targets can be temperature-adjusted as $\tilde{q}_k = q_k^{\alpha} / \sum_{\ell} q_{\ell}^{\alpha}$ (with $\alpha < 1$ flattening and $\alpha > 1$ sharpening) before computing cross-entropy, which acts as a controlled regularizer. Parameters are optimized with AdamW; a short cosine learning-rate schedule with a brief warmup helps the first epochs settle, and a simple moving-average of parameters provides a stable set of weights for evaluation/export. Loss is accumulated as a data-weighted mean, so the reported **epoch loss**[43] is independent of batch size. Inputs are read from an explicit list of `.npz` paths, and learned weights are written as a lightweight state-dictionary checkpoint (e.g., `policy_value_net.pt`) for downstream use.

6.4 Contracts

Simplicity and readability are deliberate: the dataset only loads and serves tensors; the training routine handles a single epoch of optimization; and the driver wires the model, optimizer, data, and loop. We avoid external logging frameworks and heavyweight scheduling, relying only on compact stability aids: decoupled weight decay, optional gradient clipping, a short cosine schedule with brief warmup, and an exponential moving average (EMA[44])of parameters.

A few contracts are assumed: the factored policy output must align with the loss semantics (from-square marginal and per-from delta conditionals, with log-probabilities along the appropriate axes); PF and each row $PD(\cdot \mid i)$ are normalized (together implying a distribution over the 4096-move space); $Z \in [-1, 1]$ and broadcasts to the value head's shape; and all shards share consistent shapes and `dtype`s such that concatenating them in memory is acceptable for the target machine.

[43]Epoch loss is the average loss over all mini-batches in a training epoch, providing an overall measure of performance after one complete pass through the dataset.

[44]EMA (Exponential Moving Average) is a technique used to smooth the training process of a model by averaging the model's weights over time. It is particularly useful for improving the stability of evaluation metrics and generating more reliable predictions after training.

6.5 Efficiency & extensibility

Efficiency favors determinism over bells and whistles: we use single-worker loading and full precision; accelerators can be used when available, but mixed precision is off by default. When using an accelerator, pinned host memory may be enabled to improve host-to-device transfer. Tensors move to the chosen device each iteration to avoid mismatches, and per-batch loss normalization keeps gradients comparable across batch sizes—failure modes stay obvious and easy to debug. For reproducibility, set global seeds and backend flags; with single-worker loading this yields repeatable epochs.

Progress reporting is intentionally minimal—one scalar (average loss per epoch) printed to the console. When richer observability is needed, it is easy to add CSV[45] or TensorBoard[46] logging without touching the core loop.

Because responsibilities are cleanly separated, extensions slot in naturally: adjust temperature smoothing, enable gradient clipping or automatic mixed precision for speed, tune the learning-rate schedule, or switch the parameter moving-average to an *exponential moving average (EMA)* for smoother evaluation. The dataset already supports multiple shards and can be augmented with weighting, filtering, or curriculum logic without a rewrite.

Portability and reproducibility are strong: the stack relies only on **NumPy** and a mainstream deep-learning framework, and runs the same on Linux, macOS, and Windows. While global seeding and backend flags are not set by default, a brief reproducibility preamble (seeds, single-worker loading) offers consistent runs. Memory usage remains modest beyond the concatenated dataset and standard training buffers—no extra caches or staging areas—so scaling batch size or widening the model is straightforward when hardware allows.

[45]CSV: comma-separated text tables.

[46]TensorBoard: widely used DL visualization; usable from PyTorch.

7 Self-Play

At the top level, this script runs a complete AlphaZero-style loop: it **explores** positions via self-play to create data, **trains** the policy–value network on that data, **evaluates** the fresh network against the exploration backend in an **arena** with a sound statistical test, and then **evaluates** again for logging and datasets. As shown in Figure 7.1, those four phases—EXPLORE → TRAIN → ARENA → EVAL—repeat for a configurable number of cycles, and each phase has its own outputs, schedules, and performance knobs. Every cycle emits per-game and combined PGNs, sparse or dense datasets as .npz, CSV summaries for the arena, a line-oriented JSON[47] log of all games, and a manifest that captures the full configuration used in that run.

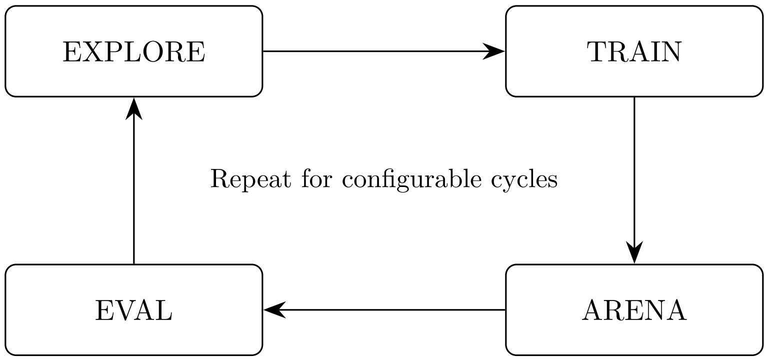


Figure 7.1: AlphaZero-style loop

7.1 Orchestration and Phases

This system is the conductor for a full self-play pipeline: it coordinates efficient data generation, safe and reproducible training, principled evaluation and promotion, and exhaustive logging—while remaining portable and kind to your hardware.

Play can be driven by either a lightweight heuristic backend or a neural-network backend, chosen independently for the exploration and evaluation

[47]JSON (JavaScript Object Notation) is a lightweight, text-based data format used for storing and exchanging structured data. It is human-readable and commonly used for configuration files, logs, and data interchange between systems, such as in the logging and configuration management of self-play cycles.

phases. The arena then pits the current evaluation network against the current exploration backend with alternating colors for fairness. Each phase writes its own artifacts to phase-specific directories for quick inspection, and a manifest records seeds, devices, configurations, and thresholds to make every run reproducible.

Schedules and annealing are explicit and traceable: as shown in Figure 7.2, root exploration noise (Dirichlet and mix-in) shrinks by cycle, the temperature for policy sampling cools across opening plies, and only a controlled fraction of exploration games use MCTS versus policy-only play.

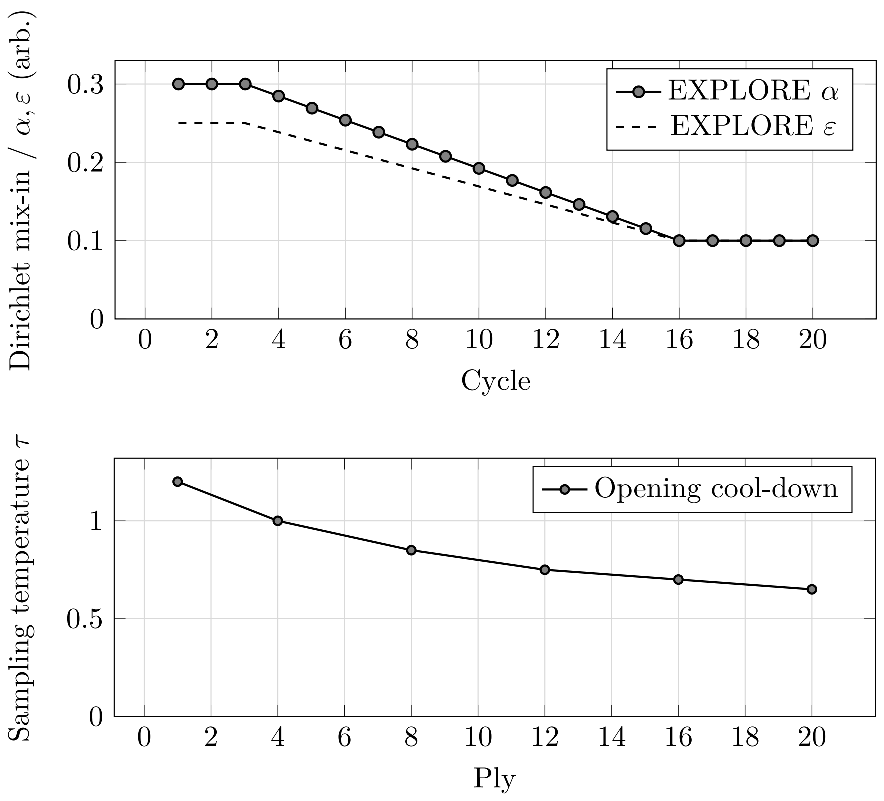


Figure 7.2: EXPLORE noise by cycle (both α and ε) and policy temperature by ply.

Key schedules are recorded in the per-cycle manifest. Configuration remains tidy and centralized: all knobs—seeds, counts, devices, inference behavior, training hyperparameters, and per-phase limits—live in one clear configuration; outputs are separated by phase (explore, eval, arena, data, logs); components are cleanly separated so parts can be swapped without untangling the whole loop.

7.2 Performance, Parallelism, and Portability

Self-play is parallelized through an OS-aware process pool that stays stable on Linux, macOS, and Windows. Worker counts are capped sensibly relative to available cores, and chunking is tuned so progress bars advance per game even under concurrency. If a platform rejects the preferred multiprocessing start method, execution falls back seamlessly to a sequential path. To prevent oversubscription, BLAS[48] thread pools are limited (e.g., via `OMP_NUM_THREADS` and `MKL_NUM_THREADS`), and hot paths avoid deep copies by relying on stack-style push/pop state updates, resulting in fast, predictable throughput with clear terminal feedback.

Portability and safety are built in: the system detects Windows and notebook environments, chooses safe start methods, calls *freeze_support*, and wraps parallel sections in robust fallbacks. All filesystem writes are atomic, directories are created up front, and the inference and data paths behave well on machines without GPUs or with constrained thread pools.

7.3 Inference and Search Behavior

When a GPU is available, a separate central inference process can serve neural evaluations to all workers. Requests flow through queues and are micro-batched, with adaptive flushing every few milliseconds or when a maximum batch size is reached. On non-CPU devices the model runs in FP16, applies the softmax only over the legal policy indices, and returns compact priors. A light LRU[49] cache—keyed by BLAKE2b[50] hashes of the $18 \times 8 \times 8$ state

[48]BLAS (Basic Linear Algebra Subprograms) is a set of standardized low-level routines used for performing common linear algebra operations such as matrix multiplication, vector addition, and dot products.

[49]LRU (Least Recently Used) is a cache eviction algorithm that removes the least recently accessed items first when the cache reaches its limit. This helps maintain an efficient use of memory by ensuring that frequently used data stays in the cache while older, less used data is discarded.

[50]Blake2b is a cryptographic hash function designed for high-speed hashing and security. It produces a fixed-size hash value (e.g., 256 bits) and is used in scenarios requiring fast, collision-resistant hashing, such as in the LRU cache for deduplicating queries in the neural

planes plus the legal-index tuple or the evaluation perspective—deduplicates in-flight and repeated queries. If centralized inference is disabled, workers fall back to local, device-aware inference.

Search behavior is configured in three profiles for evaluation, exploration, and arena play. Each supports dynamic simulation budgets based on branching factor and game phase; early stopping once the visit distribution becomes sufficiently concentrated; PUCT selection with optional top-k expansion; and root Dirichlet noise that anneals by cycle for exploration; arena uses near-deterministic settings with a very small or zero mix-in. In openings, a configurable number of plies use policy-only play to diversify lines before handing control to MCTS.

When the neural backend is off—or by design in some runs—a handcrafted policy supplies fast, usable priors, rewarding center control, profitable captures, castling, promotions, and in endgames driving the opposing king to the edge while coordinating one's own king. A very cheap SEE-lite[51] checks attack and defense on the destination square to avoid blunders in exchanges. The value estimate blends bounded material balance with in-check adjustments, mobility differences, and rapid mate-in-one detection. Game logic and encoding follow a compact, learning-friendly design: positions use 18 planes of size 8×8; moves use a 64×64 indexing of (from, to) pairs; SAN/PGN generation handles disambiguation, checks, castling, promotions, and en-passant.

7.4 Data, Training, and Evaluation

During self-play, every position is recorded with its encoded state, the player to move, and a factored policy representation with two arrays—PF (from-square probabilities over 64) and PD (per-from delta probabilities over 64×64). This factored representation is compact and training-friendly and scales cleanly. All shard writes are atomic—written to a temporary path, flushed, and moved into place—and shards are concatenated correctly even when rows are ragged.

Once exploration ends, training can begin immediately on the freshly collected dataset. The pipeline uses AdamW, a cosine learning-rate schedule with warmup, gradient clipping, and an exponential moving average of

network inference pipeline.

[51]SEE-lite is a simplified version of the SEE (Static Exchange Evaluation) algorithm used in chess engines to assess the value of a potential exchange. It performs basic checks to evaluate captures or exchanges, helping the engine avoid blunders and quickly estimate the value of a move in exchange situations.

weights. Sparse policy labels are consumed with a collation step that gathers the appropriate log-probabilities per row. The value loss is MSE against scalar targets; the policy loss is cross-entropy against the smoothed target distribution; and the relative weight of the value term is configurable. EMA weights are saved alongside the regular checkpoint and then loaded back before exporting the final file.

Targets and adjudication are practical and training-aware: terminal outcomes map to $+1/-1$ for checkmates or resignations from the side-to-move perspective and to 0 for other terminals. Draws receive a small penalty; when the final position is materially better for one side, a slight nudge biases draws toward that side. Policy targets are lightly smoothed toward the uniform over legal moves. Games may end early by resignation after a minimum move count if evaluations stay below a threshold, or by "quickdraw" logic based on a configurable halfmove clock. Standard terminal mapping covers checkmate, resignation, stalemate, repetition, the fifty-move rule, insufficient material, and quickdraw. The arena evaluates and promotes candidates rigorously: matches alternate colors; a SPRT[52] evaluates the hypothesis that the evaluation network is meaningfully better (e.g., p = 0.55 vs p = 0.50) with controlled type-I and type-II error rates. The system computes score, an approximate Elo[53], and a 95% confidence interval, writes per-game PGNs and summary CSVs, and—upon acceptance—promotes the evaluation network to become the new exploration policy for the next cycle. Operational hygiene supports auditability: a live-logging utility tees output to timestamped files, enables a fault handler, periodically `fsyncs`[54], renders progress bars correctly in IDE[55] terminals, and writes a per-cycle JSON manifest capturing seeds,

[52]SPRT (Sequential Probability Ratio Test) is a statistical method used for hypothesis testing. It is applied in the self-play arena to evaluate whether one network is significantly better than another by testing the hypothesis of a higher win probability, controlling type-I and type-II errors during evaluation.

[53]Elo is a rating system used to calculate the relative skill levels of players in two-player games like chess. It is based on the idea that the performance of a player is proportional to the rating difference between them and their opponent. The system is used in the self-play arena to evaluate and track the performance of different models.

[54]fsync is a system call used to flush data to disk, ensuring that any changes made to files are written to persistent storage. This is crucial for ensuring data integrity, particularly in logging and checkpointing systems to avoid data loss during unexpected failures.

[55]IDE (Integrated Development Environment) is a software application that provides comprehensive facilities for software development. It typically includes a code editor, debugger, and build automation tools. IDEs are commonly used for programming, debugging,

devices, configurations, quickdraw thresholds, and SPRT parameters.

and running machine learning workflows in a more efficient and user-friendly manner.

8 Simulation & Results

8.1 simulation Parameters

8.2 Results & Comments

9 Scripts

9.1 Referenced scripts

The exact scripts used in this presentation are available here together with a runnable build of the trained network.

9.2 Most recent scripts

The project evolves: the latest code, likely differing from what is presented here, is available here along with a runnable build of the current trained model.

References & Suggested Readings

- Silver et al. (2017) **Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm.**
- Silver et al. (2018) **A General Reinforcement Learning Algorithm that Masters Chess, Shogi, and Go through Self-Play.**
- Sutton & Barto (2018) **Reinforcement Learning: An Introduction (2nd ed.).**
- Gelly & Silver (2011) **Monte-Carlo Tree Search and Rapid Action Value Estimation.**

License & Citation