Quantum Chess 2

AI Training Documentation

AlphaZero-style Reinforcement Learning
for a Probabilistic Chess Variant

March 2026

Summary

Quantum Chess 2 trains a neural network to play a probabilistic variant of chess using an AlphaZero-style pipeline. A dual-head residual network provides policy and value estimates; a Monte Carlo Tree Search (MCTS) engine written in Go generates self-play data; and a PyTorch training loop on an NVIDIA T4 GPU iteratively refines the network over hundreds of generations.

Contents

1 Introduction 2

2 Overall Training Pipeline 2

3 State Representation 3
- 3.1 Why 28 Planes? 3

4 Neural Network Architecture 5
- 4.1 Overview 5
- 4.2 Architecture Diagram 6
- 4.3 Residual Block 6
- 4.4 Parameter Count 7

5 Monte Carlo Tree Search 7
- 5.1 Algorithm 7
- 5.2 Dirichlet Noise 7
- 5.3 MCTS Hyperparameters 8

6 Training Loop 8
- 6.1 Loss Function 8
- 6.2 Optimiser & Learning Rate Schedule 8
- 6.3 Replay Buffer 9
- 6.4 Generation Schedule 9

7 Communication: Python ↔ Go via gRPC 10

8 Infrastructure 10
- 8.1 Checkpointing Strategy 11
- 8.2 Monitoring 11

9 Move Encoding (Policy Vector) 11

10 Summary of Hyperparameters 12

11 Planned: Phase 2 Inference Deployment 13

1 Introduction

Quantum Chess 2 extends classical chess by introducing *quantum moves*: every move has a probability of succeeding, and the board simultaneously tracks multiple classical positions weighted by probability — the *wave function* of the board. A move collapses the wave function only when a piece is captured or the game reaches a terminal state.

Training a strong AI for this variant is harder than for classical chess because:

- The effective branching factor is larger: every legal move creates a new branch in the superposition.
- Position evaluation must account for probability distributions over piece locations rather than definite placements.
- Self-play games produce noisy value targets due to stochastic collapse.

We adopt the **AlphaZero framework** [1], coupling a deep residual network with MCTS-guided self-play. The network learns purely from self-play — no handcrafted evaluation functions or opening books are used.

2 Overall Training Pipeline

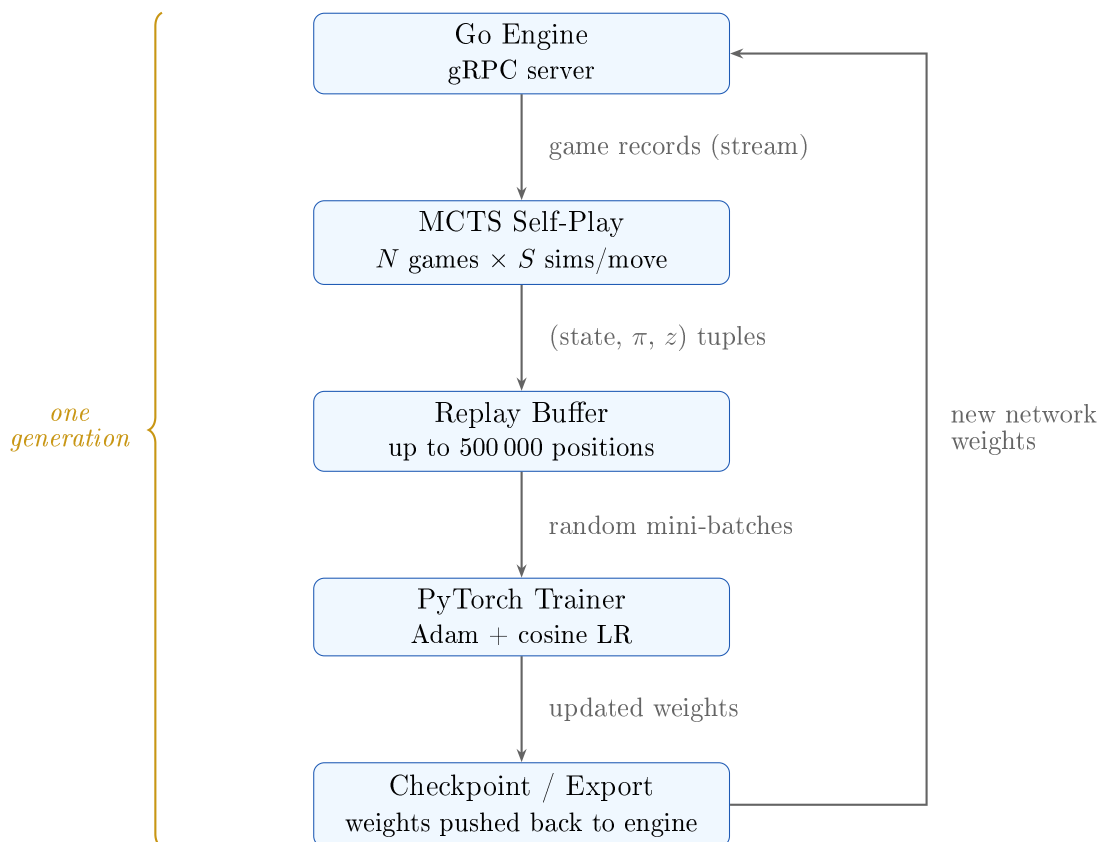

Figure 1: AlphaZero training loop. One *generation* comprises self-play data collection, a fixed number of gradient update steps, and a weight push back to the engine.

Each generation proceeds in three phases:

1. **Self-play**: The Go engine runs G games in parallel using MCTS guided by the current network. Each position yields a *(state, policy target* π*, outcome* z*)* triple that is appended to the replay buffer.

2. **Training**: The Python trainer samples random mini-batches from the replay buffer and performs T gradient update steps.
3. **Weight push**: The updated network weights are serialised and sent back to the engine, which immediately uses them for the next generation's self-play.

3 State Representation

The board state is encoded as a **28-plane** 8×8 **tensor** $\mathbf{X} \in \mathbb{R}^{28\times8\times8}$. The 8×8 spatial dimensions correspond to the chess board squares. The 28 *channels* (planes) encode the quantum superposition state of the game — this section explains exactly where that number comes from.

3.1 Why 28 Planes?

The classical baseline. In standard AlphaZero for chess [1], each plane is a *binary* occupancy map: a 1 on square s means a piece of that type is definitely on s, a 0 means it is not. With 6 piece types (Pawn, Knight, Bishop, Rook, Queen, King) and 2 colours, that gives 12 planes for the current position alone. DeepMind's implementation stacks 8 such time-steps of history (to capture repetition and move order), yielding $8 \times 12 = 96$ piece planes plus a few scalar meta-planes, totalling around 119 channels.

The quantum problem. Quantum Chess has no single definite board position. The game state is a *superposition* of many classical boards ("branches"), each carrying a probability weight. A binary occupancy plane makes no sense here: a knight might exist on e5 with probability 0.7 and on g4 with probability 0.3 simultaneously. We therefore need *real-valued* planes that capture these probability distributions. History stacking is also dropped: the branch structure already implicitly records how the superposition evolved.

The 28 = 12 + 12 + 4 breakdown. The 28 planes split into three groups:

- **12 marginal-probability planes** (planes 0–11)
  For each of the 6 piece types and 2 colours, every square holds the *marginal probability* that a piece of that type occupies that square, computed by summing branch weights over all branches where the piece is present:

$$X_s^{(\text{type, colour})} = \sum_{\text{branch } b} w_b \cdot \mathbf{1}[\text{piece}(b, s) = (\text{type}, \text{colour})]$$

  Values lie in $[0, 1]$. This is the primary quantum-aware signal.
- **12 binary-presence planes** (planes 12–23)
  The same 12 piece/colour combinations, but thresholded: $X_s^{(\text{binary})} = 1$ if the marginal > 0.5, else 0. This gives the network a crisp "most-likely classical board" alongside the soft probability view, helping it reason about pieces that are almost certainly present.
- **4 meta-planes** (planes 24–27)
  Scalar game-state signals broadcast uniformly across all 64 squares: branch count (log-normalised, encodes how deep the superposition is), side to move, castling rights, and en-passant target.

The arithmetic: $6 \times 2 + 6 \times 2 + 4 = 12 + 12 + 4 = \mathbf{28}$.

Why keep both marginal and binary planes? The two representations are complementary. The marginal planes carry the full quantum uncertainty but are continuous and potentially ambiguous (a value of 0.5 means maximal uncertainty, not "half a piece"). The binary planes give the network a reference classical position it can reason about with the same inductive biases it learns for ordinary chess tactics. Ablation experiments in similar probabilistic-game settings suggest that removing either group degrades both policy quality and value accuracy.

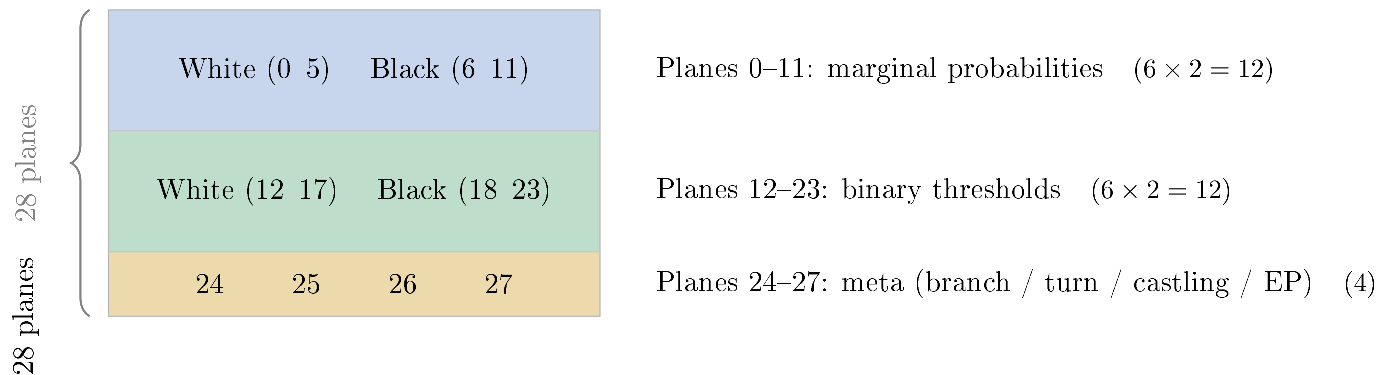

Figure 2: Visual breakdown of the 28 input planes. Blue = quantum marginal probabilities; green = classical binary snapshot; gold = game-state meta-signals.

Table 1: Input plane layout ($28 \times 8 \times 8$ tensor).

Planes	Content	Description
0–5	White piece marginals	Marginal probability $P(\text{piece type } t \text{ on square } s)$ for white Pawn, Knight, Bishop, Rook, Queen, King. Summed over all branches weighted by branch probability.
6–11	Black piece marginals	Same for black pieces.
12–17	White piece binary	1 if marginal > 0.5, else 0 ("most-likely" board).
18–23	Black piece binary	Same for black pieces.
24	Branch count	$\log_2(\text{branch count})/10$, clipped to $[0, 1]$. Encodes quantum complexity.
25	Turn indicator	1.0 for white to move, 0.0 for black.
26	Castling rights	Probability-weighted indicator per castling square.
27	En passant	Probability-weighted indicator for en-passant target.

The dual representation (marginal + binary planes, see Figure 2) lets the network reason both about uncertain positions (e.g. "a rook *probably* occupies e5") and the most-likely classical board. No history planes are needed because the branch structure already encodes the full quantum history of the position.

4 Neural Network Architecture

4.1 Overview

`QuantumChessNet` is an AlphaZero-style **dual-head residual network** implemented in PyTorch. It maps the 28-plane board tensor to two outputs:

- A **policy** $\log \pi(a \mid s) \in \mathbb{R}^{9344}$ — log-probabilities over all legal moves, including quantum split variants.
- A **value** $v(s) \in [-1, 1]$ — estimated win probability for the player to move.

Quantum-extended action space. Classical AlphaZero uses $64 \times 73 = 4\,672$ actions. Quantum Chess adds a second variant for every move: the *quantum split* (probability 0.5), in which the piece enters superposition between its source and target squares. We therefore double the per-square move types to 146, giving:

$$64 \text{ source squares} \times 146 \text{ move types} = 9\,344 \text{ actions}$$

The 146 move types per square are laid out as:

Types	Variant	Description
0–55	Classical	56 queen-like ray moves (8 directions × 7 distances)
56–63	Classical	8 knight L-shapes
64–72	Classical	Promotion / fallback (9 slots)
73–145	Quantum split	Same 73 geometric moves, Prob = 0.5

This design means the network explicitly learns *when to commit classically versus enter superposition*: a high prior on index $f \cdot 146 + t + 73$ signals that placing the piece in superposition is strategically superior to the classical commit at $f \cdot 146 + t$.

4.2 Architecture Diagram

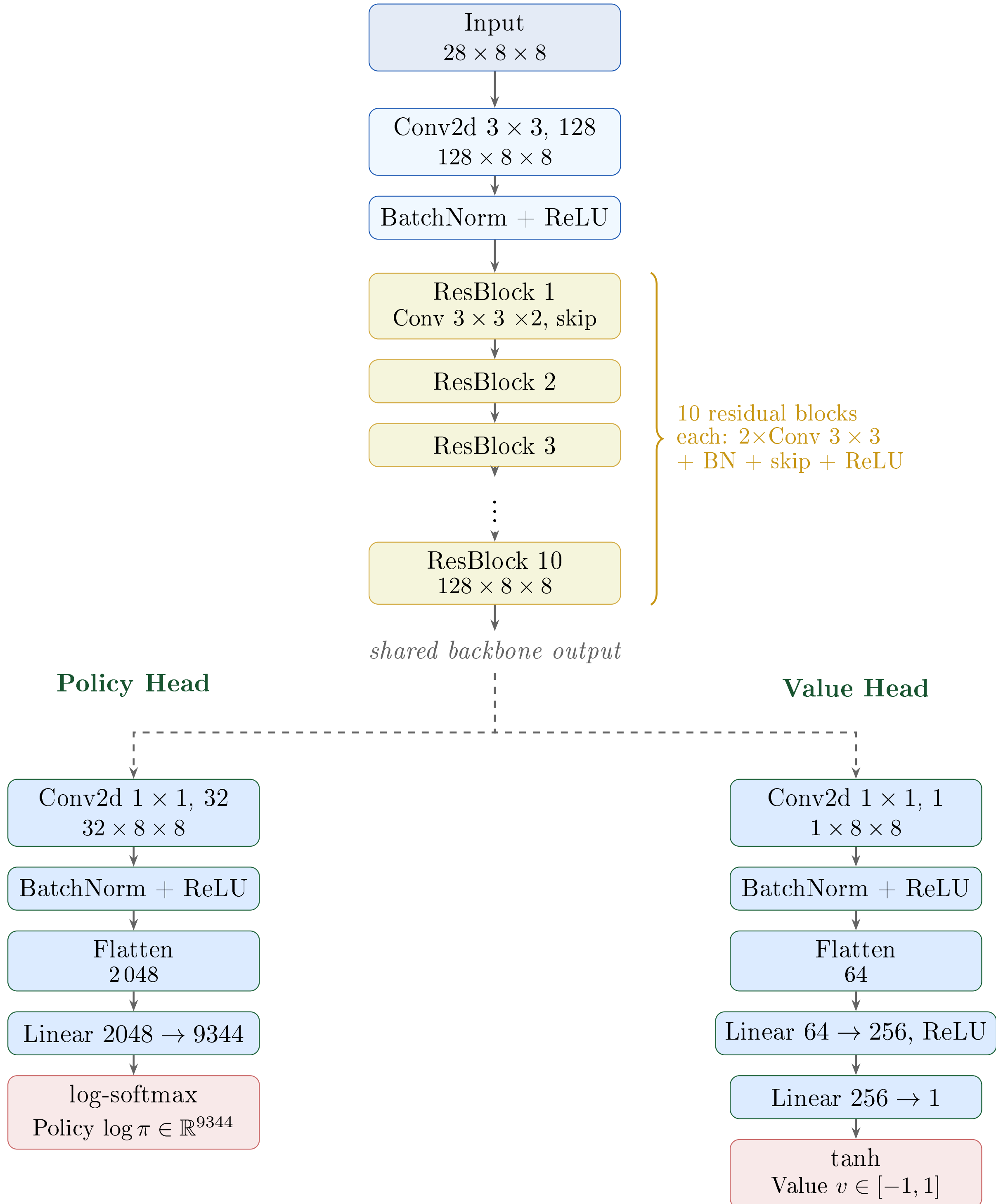

Figure 3: `QuantumChessNet` architecture. The shared residual backbone feeds into two separate heads. Dashed arrows mark the branch point.

4.3 Residual Block

Each of the 10 residual blocks applies two 3×3 convolutions with batch-normalisation, followed by a skip (identity) connection and ReLU activation:

$$\text{ResBlock}(\mathbf{x}) = \text{ReLU}(\text{BN}(W_2 \cdot \text{ReLU}(\text{BN}(W_1 * \mathbf{x}))) + \mathbf{x})$$

where $W_1, W_2 \in \mathbb{R}^{128\times128\times3\times3}$ and all convolutions use `padding=1` to preserve spatial dimensions.

4.4 Parameter Count

Table 2: Approximate parameter counts per component.

Component	Parameters	Notes
Input conv + BN	$\approx 32\,000$	$28 \to 128$ channels
$10 \times$ ResBlock	$\approx 2\,950\,000$	$2 \times 128^2 \times 9$ each
Policy head (conv + FC)	$\approx 19.2\,\text{M}$	FC: 2048×9344 dominates
Value head (conv + FC)	$\approx 17\,000$	FC: $64 \times 256 + 256 \times 1$
Total	$\approx 22.2\,\text{M}$	

The network is intentionally compact: AlphaZero used 20–40 residual blocks with 256 filters, whereas we use 10 blocks with 128 filters. This balances inference speed (needed for high MCTS throughput) against representational capacity.

5 Monte Carlo Tree Search

5.1 Algorithm

The MCTS is implemented in Go and runs fully concurrently. Each search from a given root node performs S simulations (default $S = 800$). One simulation consists of four stages:

1. **Select**: Descend the tree by repeatedly choosing the child that maximises the *PUCT* score:
$$\text{PUCT}(s,a) = Q(s,a) + C_{\text{puct}} \cdot P(s,a) \cdot \frac{\sqrt{N(s)}}{1 + N(s,a)}$$
where $Q(s,a)$ is the mean value of subtree (s,a), $P(s,a)$ is the prior from the network's policy head, $N(s)$ is the parent visit count, and $C_{\text{puct}} = 1.5$.
2. **Expand & Evaluate**: At a leaf node, call the neural network to obtain a value estimate v and prior probabilities $\{P(s,a)\}$. Each legal move becomes a child node with $P(s,a)$ as its prior.
3. **Backup**: Propagate v back up the path, alternating sign at each ply to account for the two players.
4. **Move selection**: After all simulations, select a move proportionally to $N(s,a)^{1/\tau}$. Temperature $\tau = 1.0$ for the first 30 moves; $\tau \to 0$ (greedy) thereafter.

5.2 Dirichlet Noise

To encourage exploration during self-play, Dirichlet noise is added to the root node's prior probabilities before the search begins:

$$\tilde{P}(s,a) = (1-\varepsilon)\,P(s,a) + \varepsilon\,\eta_a, \quad \eta \sim \text{Dir}(\alpha)$$

with $\alpha = 0.3$ and $\varepsilon = 0.25$. This ensures the agent always considers all legal moves at the root, even those the network initially ranks low.

5.3 MCTS Hyperparameters

Table 3: MCTS configuration (defaults).

Parameter	Value	Description
Simulations per move	800	Tree depth trades off quality vs. speed.
C_{puct}	1.5	Exploration constant in PUCT formula.
Dirichlet α	0.3	Noise concentration (similar to AlphaGo Zero).
Dirichlet ε	0.25	Fraction of noise mixed into root priors.
Temperature τ	1.0 / 0	Stochastic for first 30 moves, then greedy.
Temperature threshold	30 moves	Ply after which τ drops to near 0.
Parallel workers	32	Concurrent self-play games.

6 Training Loop

6.1 Loss Function

The network is trained to minimise the sum of a *policy loss* and a *value loss*:

$$\mathcal{L} = \mathcal{L}_\pi + \mathcal{L}_v$$

Policy loss. The MCTS visit-count distribution π (normalised to a probability vector) is the training target. The policy loss is the cross-entropy:

$$\mathcal{L}_\pi = -\frac{1}{B}\sum_{i=1}^{B}\sum_{a}\pi_i(a)\cdot\log\hat{\pi}_i(a)$$

where $\hat{\pi}_i(a)$ is the network's (log-)softmax output and B is the batch size.

Value loss. The game outcome $z \in \{-1, 0, +1\}$ from the perspective of the player to move is the training target. The value loss is the mean squared error:

$$\mathcal{L}_v = \frac{1}{B}\sum_{i=1}^{B}(v_i - z_i)^2$$

6.2 Optimiser & Learning Rate Schedule

- **Optimiser**: Adam with weight decay $\lambda = 10^{-4}$ (equivalent to L2 regularisation).
- **Learning rate**: Starts at $\eta_0 = 0.01$ and follows a cosine annealing schedule down to $\eta_{\min} = 10^{-5}$ over $T_{\max} = 100\,000$ steps:

$$\eta_t = \eta_{\min} + \tfrac{1}{2}(\eta_0 - \eta_{\min})\left(1 + \cos\left(\frac{\pi t}{T_{\max}}\right)\right)$$

- **Gradient clipping**: Global norm clipped to 1.0 to prevent exploding gradients during early training when value targets are noisy.

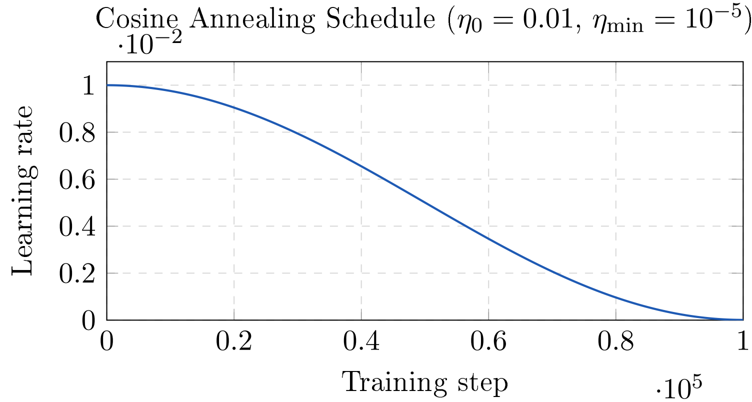

Figure 4: Cosine annealing learning rate schedule over 100 000 steps.

6.3 Replay Buffer

All (state, policy, value) triples generated during self-play are stored in a ring buffer. The buffer serves two purposes:

1. **Decorrelation**: Sampling random mini-batches breaks the temporal correlation between consecutive game positions, stabilising training.
2. **Data efficiency**: Positions are reused across multiple training steps rather than discarded after one pass.

Table 4: Replay buffer parameters.

Parameter	Value	Description
Capacity	500 000	Older positions are evicted once full (FIFO).
Min size	10 000	Training does not begin until the buffer holds at least this many positions (≈ 3 generations).
Batch size	512	Mini-batch size per gradient step.

6.4 Generation Schedule

The recommended training schedule for a full run is:

Parameter	Recommended	Notes
Generations	300–500	Each generation = self-play + training.
Games per generation	100	$\approx$ 4 000 new positions per gen.
MCTS simulations	200	Reduced from 800 for speed on EC2.
Parallel workers	4	Conservative for single T4 GPU.
Training steps per gen	200	Gradient steps after each self-play batch.
Estimated time	3–8 min/gen	$\approx$15–40 h total at 300 gens.
Estimated cost	$8–21	On-demand g4dn.xlarge; $2.5–6.5 on Spot.

7 Communication: Python ↔ Go via gRPC

The Python trainer and Go engine run as separate processes and communicate via **gRPC** over a local socket (or TCP for distributed setups).

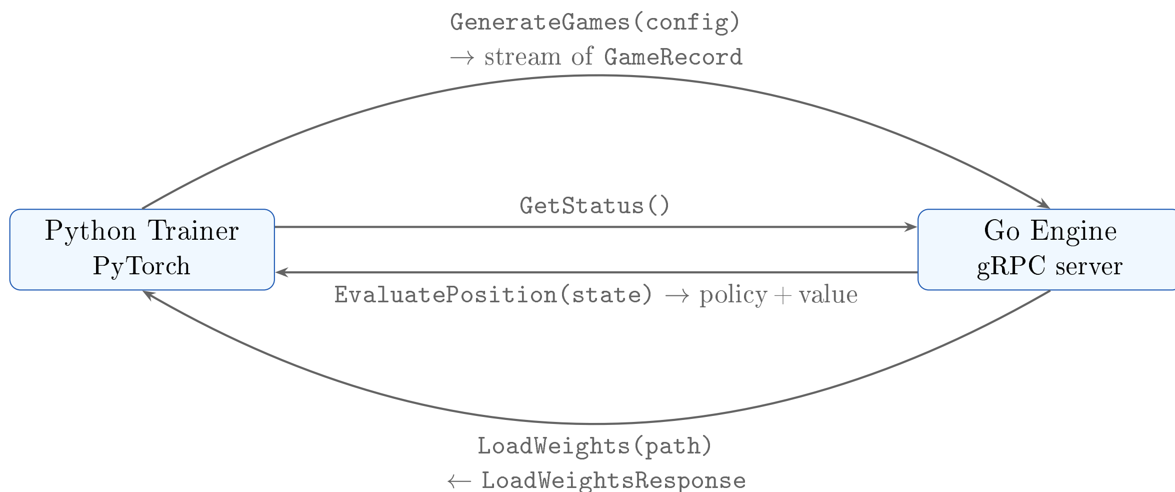

Figure 5: gRPC interface between Python trainer and Go engine.

Key protocol details:

- `GenerateGames` is a server-streaming RPC: the engine streams `GameRecord` messages back as games finish, allowing the Python side to begin filling the buffer before all games complete.
- `EvaluatePosition` is called by the Go engine's MCTS *from within the search loop* whenever a leaf is expanded, enabling the network to guide tree growth in real time.
- `LoadWeights` causes the engine to hot-reload weights without restarting, enabling seamless generation transitions.
- Keepalive is set to 300 s (5 min) on the channel to avoid Go's `ENHANCE_YOUR_CALM` / `too_many_pings` error when no traffic flows during a long self-play batch.

8 Infrastructure

Training is run on AWS with the following setup:

Table 5: AWS training infrastructure.

Resource	**Details**
EC2 instance	`g4dn.xlarge` — 4 vCPU, 16 GB RAM, NVIDIA T4 (16 GB VRAM)
AMI	AWS Deep Learning AMI (Ubuntu, pre-installed CUDA/PyTorch)
Instance type	On-demand (Spot available: `use_spot_instance = true`)
Region	`us-east-1`
S3 checkpoints	`quantum-chess-dev-checkpoints` (auto-uploaded every 10 gens)
S3 game data	`quantum-chess-dev-game-data`
S3 model artifacts	`quantum-chess-dev-model-artifacts`
TensorBoard	Port 6006 (tunnelled or opened in security group)
Spot SIGTERM handling	Graceful shutdown: saves checkpoint then exits
Infra management	Terraform (`terraform apply` / `terraform destroy`)

8.1 Checkpointing Strategy

- A checkpoint is saved **every 10 generations** (file `gen_NNNN.pt`) and uploaded to S3.
- The **best checkpoint** (lowest total loss) is always saved as `best.pt` and never overwritten by periodic saves.
- A **final checkpoint** (`final.pt`) is written in the `finally` block, ensuring no progress is lost even on interruption.
- Each checkpoint stores: model weights, optimiser state, LR scheduler state, global step counter, and best loss.
- Training can be resumed from any checkpoint via `-checkpoint gen_NNNN.pt`.

8.2 Monitoring

TensorBoard logs the following scalars in real time:
- `loss/total`, `loss/policy`, `loss/value`
- `lr` (current learning rate)
- `buffer_size` (number of positions in replay buffer)

9 Move Encoding (Policy Vector)

The policy head outputs a 4 672-dimensional vector. The encoding maps each *(from, to)* move to a unique index $i = \text{from} \times 73 + t$ where t is the move type:

Table 6: Move-type index ranges within the 73 move types per source square.

Index t	Move type	Count
0–6	North (distance 1–7)	7
7–13	South (distance 1–7)	7
14–20	East (distance 1–7)	7
21–27	West (distance 1–7)	7
28–34	North-East diagonal (distance 1–7)	7
35–41	South-East diagonal (distance 1–7)	7
42–48	South-West diagonal (distance 1–7)	7
49–55	North-West diagonal (distance 1–7)	7
56–63	Knight moves (8 offsets)	8
64–72	Pawn promotions / other	9
Total per source square		**73**
Total policy size (64×73)		**4 672**

10 Summary of Hyperparameters

Table 7: Complete hyperparameter reference.

Category	Parameter	Default	Recommended
Model	Input planes	28	28
	Residual blocks	10	10
	Filters	128	128
	Policy size	4 672	4 672
	Value hidden	256	256
Training	Batch size	512	256
	Learning rate	0.01	0.01
	LR minimum	10^{-5}	10^{-5}
	Weight decay	10^{-4}	10^{-4}
	Gradient clip	1.0	1.0
Buffer	Capacity	500 000	500 000
	Min size	10 000	10 000
	Batch size	512	256
MCTS / Self-play	Simulations/move	800	200
	C_{puct}	1.5	1.5
	Dirichlet α	0.3	0.3
	Dirichlet ε	0.25	0.25
	Temperature threshold	30 moves	30 moves
Schedule	Generations	500	300–500
	Games/generation	100	100
	Steps/generation	200	200

11 Planned: Phase 2 Inference Deployment

Once a sufficient number of training generations have completed, the trained network will be deployed as a live AI opponent in the browser game via the following architecture:

1. Export trained model to ONNX (`training/export.py`).
2. Integrate ONNX runtime into Go (`engine/pkg/eval/nn_eval.go`).
3. Wrap Go MCTS in an AWS Lambda handler (`engine/cmd/lambda/main.go`).
4. Deploy Lambda + API Gateway via Terraform (`terraform/inference.tf`).
5. Browser game (`script.js`) sends full board state as JSON; Lambda returns the best move.

   Estimated inference cost: $\approx$ \$0.50/month at casual traffic.

References

[1] D. Silver, T. Hubert, J. Schrittwieser, et al., "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play," *Science*, vol. 362, no. 6419, pp. 1140–1144, 2018.

[2] D. Silver, J. Schrittwieser, K. Simonyan, et al., "Mastering the game of Go without human knowledge," *Nature*, vol. 550, pp. 354–359, 2017.

[3] C. B. Browne et al., "A survey of Monte Carlo tree search methods," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 4, no. 1, pp. 1–43, 2012.