DataGo: Retrieval-Augmented Recursive Search that Surpasses KataGo with Log-Grounded Analysis

Benjamin Huh Jason Peng Taka Khoo
Olir Eswaramoorthy David Roos Victor Lun Pun
Thayer School of Engineering, Dartmouth College
15 Thayer Drive, Hanover, NH 03755
benjamin.j.huh.24@dartmouth.edu, jason.peng.23@dartmouth.edu,
taka.khoo.th@dartmouth.edu, oliravan.eswaramoorthy.th@dartmouth.edu,
david.t.roos.th@dartmouth.edu, victor.lun.pun.th@dartmouth.edu

Abstract

Monte Carlo tree search (MCTS) with deep neural priors underpins state-of-the-art Go engines such as KataGo, yet current systems re-solve each position from scratch and discard most of their high-visit analyses. We introduce *DataGo*, a retrieval-augmented Go engine that wraps unmodified KataGo networks with (i) an uncertainty gate calibrated on log-grounded distributions, (ii) a multi-context approximate nearest-neighbor (ANN) memory keyed by symmetry-invariant hashes, and (iii) a recursive deep-search module that stores 2,000–10,000-visit analyses for re-use in future games. Using only KataGo's public networks and our code in the `datago` and `rag_store` folders, we show that DataGo achieves a 9–0–1 record against a strong KataGo baseline in synthetic stress tests and an 8–0–2 record against *real* KataGo outputs after retuning the uncertainty threshold from 0.37 (synthetic) to 0.15 (real), while activating retrieval on only 5.1% of moves and reusing cached analyses with 6.6 hits per query. A Phase 2 offline tuning sweep on self-play logs from `raw_games_data` shows that increasing deep-search visits from $1{,}000$ to $10{,}000$ reduces average policy error from 0.178 to 0.056 at the cost of $\approx 10\times$ compute. Taken together, our experiments demonstrate that retrieval-augmented, recursive deep search around KataGo can measurably shift the payoff of the zero-sum Go game in favor of DataGo, without training new networks, by selectively turning past 10k-visit analyses into instant priors on hard positions.

1 Introduction

Deep neural networks combined with Monte Carlo tree search (MCTS) have transformed Go from an open challenge to a solved benchmark for superhuman play. AlphaGo and AlphaGo Zero [**??**] showed that policy/value networks coupled with MCTS can reach world-champion strength; subsequent open-source engines such as Leela Zero and KataGo [**?**] pushed the frontier via larger networks, better time-management, and improved training pipelines. Yet modern Go engines still exhibit a structural inefficiency: each position is analyzed from scratch, even if it or its symmetric variants have appeared in earlier self-play, matches, or offline analysis. Expensive 10k-visit analyses are not reused across games.

In the large language model community, retrieval-augmented generation (RAG) has emerged as a simple but powerful paradigm for reusing past computations. Our goal is to bring a similarly retrieval-augmented view to Go search. We ask: *Can a KataGo-based engine that caches high-quality analyses and selectively revisits them via ANN retrieval systematically outperform pure KataGo at fixed base visits?*

39th Conference on Neural Information Processing Systems (NeurIPS 2025).

To answer this, we build **DataGo**, a retrieval-augmented Go engine implemented in the `datago` directory of the `alphago_project` workspace. DataGo wraps unmodified KataGo binaries and models from `katago_repo` with:

- an *uncertainty gate* that uses normalized policy entropy, value distribution variance, and a game-phase multiplier (tuned in `tuning/phase1`) to decide when to trigger expensive analysis;
- a *multi-context memory* implemented via an ANN index (`src/memory/index.py`, `rag_store/ANN.py`) keyed by a symmetry-invariant hash that stores multiple deep-analysis "contexts" per position;
- a *recursive deep search* runner (`run_datago_recursive_match.py`) that explores child positions up to depth 3 using $V_{\text{deep}} \in \{2{,}000, 10{,}000\}$ visits and logs every query, hit, deep search, and context into structured logs and JSON.

Using synthetic policies and real KataGo outputs, we show:

1. On synthetic Dirichlet policies (Section 5.2), DataGo attains a 9–0–1 record vs. a strong KataGo baseline with a high activation rate (63.2% of moves trigger RAG+deep search), validating the pipeline under heavy load.
2. On real KataGo outputs with the *original* synthetic-tuned threshold ($\theta = 0.37$), the uncertainty gate never fires, yielding 0–10–0 and demonstrating the need to retune gating on real data.
3. After retuning the real-network threshold to $\theta = 0.15$, DataGo achieves 8–0–2 vs. KataGo across 10 games, with only 23 RAG queries across 454 moves (5.1% activation), 328 deep searches, and 152 cache hits (6.6 hits/query), showing that rare but high-value activations suffice to improve win rate.
4. Offline Phase 2 sweeps over deep-search depth and convergence criteria (JSONs in `tuning/tuning_results/phase2`) show that $10{,}000$-visit deep searches can reduce policy error by nearly $3\times$ relative to $1{,}000$-visit searches, but at $\approx 10\times$ compute, framing the compute–accuracy tradeoff.

All experiments are reproducible from the repository: self-play logs in `selfplay_output` and `raw_games_data`, offline rag-store construction in `datago/rag_store` and `rag_store`, tuning scripts in `datago/tuning`, and match logs in `datago/quick_test_*.log` and `datago/extended_match_*.log`. We emphasize that DataGo is a *search* and *memory* augmentation of KataGo, not a new network.

2 Background and Related Work

Go engines and MCTS. Go is a two-player zero-sum game with a state space on the order of 10^{170} positions, far beyond brute-force search. AlphaGo and AlphaGo Zero approximate an equilibrium by combining deep convolutional networks with MCTS. Given a state s, a policy network outputs a prior $\pi_\theta(a \mid s)$ and a value network outputs $V_\theta(s)$; MCTS repeatedly expands the search tree by selecting actions

$$a^\star = \arg\max_a \Big(Q(s,a) + c_{\text{puct}}\, P_\theta(s,a) \tfrac{\sqrt{N(s)}}{1+N(s,a)} \Big),$$

where $Q(s,a)$ is an empirical action value and $N(\cdot)$ are visit counts. KataGo extends this with richer outputs (ownership, score), improved rulesets, and an efficient OpenCL/CUDA implementation.

Retrieval-augmented generation. RAG methods for language models attach a memory to a pretrained model and use retrieval to fetch relevant documents or past contexts at inference time. Our work transposes this idea to Go: instead of retrieving text documents, DataGo retrieves past high-visit analyses keyed by a symmetry-aware hash of the board state. Unlike traditional opening books, our memory stores entire deep-search trees, including child-node value distributions and auxiliary metrics.

Deep search and pattern databases in games. Chess and Go engines have long used endgame tablebases and opening books as external memories. KataGo itself supports graph search [**?**], but does not persist deep-search trees across games. Our contribution is to systematically build, query, and reuse a multi-context database of deep KataGo analyses, and to show—via logs and tuning results—that doing so improves a strong baseline when gated by calibrated uncertainty.

"'latex

3 System Layout and Experimental Chronology

At a high level, the system consists of three functional layers: (i) a baseline Go engine providing a neural policy–value function and MCTS; (ii) a retrieval- augmented search layer that overlays uncertainty estimation, memory, and recursive deep search; and (iii) an offline analysis pipeline that constructs and tunes the memory used by retrieval.

Baseline engine. The baseline agent represents the Go state as $s \in \{-1, 0, +1\}^{19\times 19}$ and applies a neural evaluator

$$f_\theta(s) = \big(\pi_\theta(\cdot \mid s), V_\theta(s)\big),$$

where $\pi_\theta(\cdot \mid s) \in \Delta(L(s))$ is a probability distribution over legal moves $L(s)$ and $V_\theta(s) \in [-1, 1]$ is a win-probability proxy. A standard PUCT-based MCTS with visit budget $N_{\text{base}} = 800$ then computes empirical action values and visit counts,

$$\{Q(s,a), N(s,a)\}_{a\in L(s)}, \qquad N(s) = \sum_a N(s,a),$$

and induces a search policy $\pi_{\text{MCTS}}(a \mid s) \propto N(s,a)$. This engine also acts as a generator of self-play trajectories and offline deep analyses used for calibration.

Retrieval-augmented layer. The DataGo layer wraps the baseline engine with three additional functions.

First, an *uncertainty estimator* $\mathcal{U} : \mathcal{S} \to [0, 1]$ maps a state s to a scalar score via

$$\mathcal{U}(s) = \big(w_1 E(s) + w_2 K(s)\big)\, \phi(n(s)), \tag{1}$$

where $E(s)$ is the normalized Shannon entropy of $\pi_{\text{MCTS}}(\cdot \mid s)$, $K(s)$ is a normalized dispersion measure of child values (e.g., the variance of $\{Q(s,a)\}$ rescaled to $[0, 1]$), ϕ is a phase multiplier as a function of the total stone count $n(s)$, and w_1, w_2 are tuned weights with $w_1 + w_2 = 1$. States with $\mathcal{U}(s)$ above a threshold θ_{query} are deemed "complex" and become candidates for retrieval and deep search.

Second, a *memory encoder* $g : \mathcal{S} \to \mathbb{R}^d$ maps a symmetry- canonicalized state $\tilde{s}$ (obtained by minimizing over the eight rotational and reflectional symmetries of the board) to an embedding $z = g(\tilde{s})$. The memory $\mathcal{M}$ is an approximate nearest-neighbor index over pairs (z_i, ξ_i), where ξ_i collects deep-search metadata at state $\tilde{s}_i$:

$$\xi_i = \big(\pi_{\text{deep}}(\cdot \mid \tilde{s}_i), V_{\text{deep}}(\tilde{s}_i), \Delta\text{score}(\tilde{s}_i), n(\tilde{s}_i), \text{children}(\tilde{s}_i)\big).$$

Given a query s, the retrieval module computes $z = g(\tilde{s})$, performs an ANN search

$$(z^\star, \xi^\star) = \text{ANN}\big(z; \mathcal{M}\big),$$

and scores the relevance $\text{Rel}(s, \tilde{s}^\star) \in [0, 1]$ as a weighted sum of similarities in policy space, win rate, score, visit distribution, stone count, and komi (cf. Eq. (4)).

Third, a *recursive deep-search controller* decides, based on $\mathcal{U}(s)$ and $\text{Rel}(s, \tilde{s}^\star)$, whether to: (i) reuse the cached deep analysis $\xi^\star$, (ii) force exploration of moves highlighted by $\xi^\star$ within MCTS, or (iii) launch a new deep MCTS with visit budget $N_{\text{deep}} \gg N_{\text{base}}$ on s and selected descendants up to a maximum recursion depth $D_{\max}$. The expected deep-search cost at depth D can be approximated by

$$\text{Cost}(D) \approx N_{\text{deep}} \sum_{i=0}^{D} \prod_{j=0}^{i-1} b_j, \tag{2}$$

where b_j is the average branching factor at recursion level j. Deep-search results are fed back into $\mathcal{M}$ by adding new entries $(g(\tilde{s}), \xi)$ for all visited high-uncertainty states.

Offline analysis pipeline. An offline pipeline links the baseline engine and the RAG layer. Self-play trajectories and match logs provide a set of flagged positions $\{s_k\}$ together with their game histories. For each s_k, the analyzer reconstructs the move sequence, invokes the analysis engine with a large visit budget (e.g., $N_{\text{offline}} \in \{2{,}500, 10{,}000\}$), and records full policy vectors, win rates, score leads, ownership maps, and child-node statistics. These records form an offline dataset

$$\mathcal{D}_{\text{off}} = \big\{(s_k, \pi_{\text{off}}(\cdot \mid s_k), V_{\text{off}}(s_k), \ldots)\big\},$$

which is used both to populate $\mathcal{M}$ and to define target quantities for tuning the uncertainty function and deep-search convergence parameters.

Chronology of development. Methodologically, the project proceeds in four stages. First, we calibrate the baseline engine and generate self-play data to fix N_{base}, confirm GPU stability, and obtain $\mathcal{D}_{\text{off}}$ for a range of positions and game phases. Second, we tune the uncertainty function and relevance metric by regressing $\mathcal{U}(s)$ and $\mathrm{Rel}(s, s')$ against discrepancies between shallow (N_{base}) and deep (N_{offline}) evaluations, selecting $(w_1, w_2, \phi, \theta_{\text{query}})$ and similarity weights that best predict where deep search materially changes the policy or value. Third, we construct and refine the rag-store $\mathcal{M}$ by analyzing flagged positions with high visit counts, storing multiple contexts per canonical state, and measuring ANN recall and cache-hit patterns across synthetic and real trajectories. Finally, we integrate everything into the online recursive controller and run end-to-end matches where the effective policy at each decision point is given by either the baseline $\pi_{\text{MCTS}}(\cdot \mid s)$ or a retrieval-augmented variant $\pi_{\text{RAG}}(\cdot \mid s)$, depending on whether $\mathcal{U}(s)$ exceeds the tuned threshold and whether a relevant memory entry is available. "'

4 Methods

4.1 Baseline Engine: Policy–Value Network and MCTS

We build on a strong Go engine that couples a convolutional policy–value network with Monte Carlo tree search (MCTS). For each board state $s \in \{-1, 0, +1\}^{19\times 19}$ and legal move set $L(s)$, the network provides

$$f_\theta(s) = \big(\pi_\theta(\cdot \mid s), V_\theta(s)\big),$$

where $\pi_\theta(\cdot \mid s) \in \Delta(L(s))$ is a probability distribution over legal moves and $V_\theta(s) \in [-1, 1]$ estimates the win probability for the current player under self-play. A PUCT-style MCTS uses these outputs to construct search statistics

$$\{Q(s,a), N(s,a)\}_{a\in L(s)}, \qquad N(s) = \textstyle\sum_a N(s,a),$$

where $Q(s,a)$ is the empirical mean value of action a at s and $N(s,a)$ is its visit count. The baseline policy used for play is then

$$\pi_{\text{MCTS}}(a \mid s) \propto N(s,a)$$

after a fixed budget of $N_{\text{base}} = 800$ visits per move for both players. This baseline defines the "pure" engine against which we evaluate the retrieval- augmented system.

4.2 Uncertainty Metric and Gating

DataGo augments the baseline with an uncertainty estimator that decides when additional computation and retrieval are warranted. For each state s, we define an uncertainty score

$$\mathcal{U}(s) = \big(w_1 E(s) + w_2 K(s)\big)\, \phi\big(n(s)\big), \tag{3}$$

where E quantifies uncertainty in the search policy, K measures disagreement among child values, $n(s)$ is the number of stones on the board (a proxy for game phase), ϕ is a phase multiplier, and w_1, w_2 are nonnegative weights with $w_1 + w_2 = 1$.

Policy entropy $E(s)$. Let $\mathbf{p} \in \mathbb{R}^{|L(s)|}$ be the vector of probabilities $\pi_{\text{MCTS}}(a \mid s)$. We define normalized entropy

$$E(s) = -\frac{1}{\log |L(s)|} \sum_{a\in L(s)} p(a) \log p(a).$$

Positions where the search concentrates on a few moves have $E(s) \approx 0$, while positions with many nearly indistinguishable moves have $E(s) \approx 1$.

Value spread $K(s)$. From the shallow search at N_{base} visits, we consider the top-k children (typically $k \leq 10$) by visit count, with values $\{Q_1, \ldots, Q_k\}$. We define a normalized variance

$$K(s) = \min\bigl(1,\ \mathrm{Var}(Q_1, \ldots, Q_k)/0.25\bigr),$$

where 0.25 is the maximum variance for bounded values in $[0, 1]$. Large $K(s)$ indicates that some plausible moves lead to significantly different outcomes, signaling internal disagreement in the search tree.

Phase multiplier $\phi(n)$. The phase function ϕ adapts the gate to the stage of the game. We explored several families, including linear, exponential, and piecewise functions of the normalized stone count $n/361$. The deployed configuration uses a linear profile

$$\phi(n) = a\,\frac{n}{361} + b,$$

where (a, b) are tuned coefficients. This allows uncertainty during the endgame (where a single mistake can decide the result) to have more weight than similar uncertainty in the opening.

Gating thresholds. Two thresholds govern behavior:

- A query threshold θ_{query} determines when retrieval and deep search are even considered: if $\mathcal{U}(s) < \theta_{\text{query}}$, the system simply trusts the baseline 800-visit move.
- A deep-search threshold $\theta_{\text{deep}} = \theta_{\text{query}} + \delta$ (with offset $\delta \geq 0$) determines when a full recursive deep search is launched, as opposed to a lighter retrieval-only adjustment.

Phase 1 tuning on synthetic Dirichlet policies suggested $\theta_{\text{query}} \approx 0.37$ as a reasonable operating point. However, real KataGo policies are much sharper, inducing $\mathcal{U}(s)$ in the range $[0.05, 0.21]$; with $\theta_{\text{query}} = 0.37$ the gate never fires. Empirically, retuning to $\theta_{\text{query}} = 0.15$ on real network outputs yields an activation rate of 5.1% and strong performance improvements, highlighting the importance of calibrating the gate to the actual entropy and value-disagreement distribution of the underlying model.

4.3 Symmetry-Canonicalized Memory and Approximate Nearest Neighbors

The retrieval component maintains a memory of high-quality analyses for positions previously explored in depth. Each entry in memory corresponds to a symmetry-invariant description of a board state and a set of deep-search contexts.

Symmetry-canonical hash. Let $\mathrm{Sym}(s)$ denote the set of the eight rotational and reflectional images of s. We define the canonical representative

$$\tilde{s} = \arg\min_{s' \in \mathrm{Sym}(s)} \text{lex}(\text{flatten}(s')),$$

where lex orders flattened board encodings lexicographically. We then let

$$h(s) = \mathrm{SHA256}\bigl(\text{flatten}(\tilde{s})\bigr)$$

be a fixed-length hash of the canonical board. This guarantees that symmetric positions share the same key $h(s)$.

Multi-context storage. For each canonical position $\tilde{s}$, the memory stores a list of deep-search contexts

$$\mathrm{Ctx}(\tilde{s}) = \bigl\{c_1, \ldots, c_m\bigr\},$$

with each context c_i containing:

$$c_i = \bigl(a_i^\star, \mathcal{U}_i, V_{\text{deep},i}, \pi_{\text{deep},i}, \Delta\text{score}_i, n_i, d_i, t_i\bigr),$$

where $a_i^\star$ is the recommended move, $\mathcal{U}_i$ is the uncertainty at the time of deep search, $V_{\text{deep},i}$ and $\pi_{\text{deep},i}$ are the deep-search value and policy, Δscore_i is the score lead, n_i is the stone count (phase), d_i is the recursion depth at which the context was produced, and t_i is a timestamp. When retrieving a cached position, the system scores each context according to a relevance function such as

$$\text{score}(c_i; s) = \alpha\,\text{phase-sim}(n(s), n_i) + \beta\,\frac{V_{\text{deep},i}}{V_{\max}} + \gamma\,\text{recency}(t_i),$$

with weights (α, β, γ) and $V_{\max}$ chosen so that deeper, phase-matched, recent contexts are preferred. The context with maximal score is then used as the primary source of cached information.

Approximate nearest-neighbor index. To support fast retrieval, the system embeds each canonical state $\tilde{s}$ into a vector $z \in \mathbb{R}^d$. In the simplest version, z is a truncated and normalized copy of the deep policy $\pi_{\text{deep}}(\cdot \mid \tilde{s})$; more generally, z can concatenate policy, ownership, value, and other statistics. Memory maintains a vector index over $\{z_i\}$ using a cosine-similarity-based approximate nearest-neighbor structure (e.g., FAISS/HNSW). Given a query state s, we compute $z = g(\tilde{s})$ and retrieve its k nearest neighbors in embedding space:

$$\mathcal{N}(s) = \big\{(z_i, \mathrm{Ctx}(\tilde{s}_i))\big\}_{i=1}^{k}.$$

DataGo typically operates in a 1-NN regime, $k = 1$, combined with an exact hash check: if $h(s) \neq h(\tilde{s}_1)$, the retrieved neighbor is treated as a similar-but-not-identical position; if $h(s) = h(\tilde{s}_1)$, we treat it as an exact match and rely on the multi-context logic described above.

4.4 Relevance Scoring and Retrieval Policy

Given a candidate memory entry for a query state s, DataGo must decide how much to trust it. This is encoded in a relevance score $\mathrm{Rel}(s, \tilde{s}^\star) \in [0, 1]$ computed as a weighted combination of multiple similarity factors.

Let π_{current} and π_{stored} be the current and stored policies (over a common action space), $w_{\text{current}}, w_{\text{stored}}$ the win rates, $\Delta\text{score}_{\text{current}}, \Delta\text{score}_{\text{stored}}$ the score leads, $\mathbf{N}_{\text{current}}, \mathbf{N}_{\text{stored}}$ visit distributions on top moves, $n_{\text{current}}, n_{\text{stored}}$ stone counts, and $\text{komi}_{\text{current}}, \text{komi}_{\text{stored}}$ komi values. We define:

$$\begin{aligned}
s_{\text{policy}} &= \text{cosine}\big(\pi_{\text{current}}, \pi_{\text{stored}}\big),\\
s_{\text{winrate}} &= 1 - \big|w_{\text{current}} - w_{\text{stored}}\big|,\\
s_{\text{score}} &= 1 - \min\big(|\Delta\text{score}_{\text{current}} - \Delta\text{score}_{\text{stored}}|/M,\, 1\big),\\
s_{\text{visits}} &= \text{cosine}\big(\mathbf{N}_{\text{current}}, \mathbf{N}_{\text{stored}}\big),\\
s_{\text{stones}} &= 1 - |n_{\text{current}} - n_{\text{stored}}|/361,\\
s_{\text{komi}} &= \mathbb{I}\big[\text{komi}_{\text{current}} = \text{komi}_{\text{stored}}\big],
\end{aligned}$$

where M is a score-normalization scale and $\mathbb{I}$ is the indicator function. The final relevance score is

$$\mathrm{Rel} = 0.40\, s_{\text{policy}} + 0.25\, s_{\text{winrate}} + 0.10\, s_{\text{score}} + 0.15\, s_{\text{visits}} + 0.05\, s_{\text{stones}} + 0.05\, s_{\text{komi}}. \tag{4}$$

If $\mathrm{Rel} \geq \tau$ for a chosen threshold $\tau \in [0, 1]$, the cached analysis is considered high-confidence and its recommended move can be adopted directly or given very high priority in search. If $\mathrm{Rel} < \tau$ but the hash matches, we still exploit the stored information by prioritizing its moves for exploration, but we do not trust them outright. If the hash does not match, the entry is used only to bias exploration in a soft manner (see Section 4.6).

4.5 Recursive Deep Search

When the uncertainty gate fires and no sufficiently relevant memory entry exists, DataGo performs a recursive deep search around s. The recursion is controlled by a maximum depth $D_{\max}$ and a deep visit budget $N_{\text{deep}} \gg N_{\text{base}}$.

At depth $d = 0$, the controller: (i) sets the engine's visit limit to N_{deep}; (ii) runs a full deep search from s; (iii) extracts a deep policy $\pi_{\text{deep}}(\cdot \mid s)$, deep value $V_{\text{deep}}(s)$, and a set of child nodes $\{(a_j, s_j)\}$ together with their priors, visit counts, and win rates; and (iv) stores a new context for $h(s)$ in memory.

For each child state s_j at depth d, the controller computes $\mathcal{U}(s_j)$. If $\mathcal{U}(s_j) > \theta_{\text{deep}}$ and $d < D_{\max}$, it repeats the same procedure recursively at depth $d + 1$, but only for a subset of the most promising children (e.g., top few moves by a combination of prior, value, and uncertainty). If a recursive call reaches a state whose canonical hash already appears in memory, the controller can skip expensive search and reuse the best context instead of expanding further.

As discussed in Section 3, offline tuning evaluates different $(N_{\text{deep}}, D_{\max})$ configurations by comparing deep-search estimates to higher-visit references, trading off error against runtime. In practice, moderate deep budgets (e.g., $N_{\text{deep}} = 2{,}000$) and depths $D_{\max} \in \{1, 2, 3\}$ offer substantial improvements on a small fraction of moves without exploding compute.

4.6 MCTS with Retrieval-Augmented Priors

Finally, DataGo integrates memory into search by modifying the priors used by MCTS. Let $P_{\text{net}}(a)$ denote the policy from a fresh network evaluation at s (before any search) and let $P_{\text{rag}}(a)$ denote a retrieval-based prior constructed from a stored deep context for a relevant neighbor. DataGo defines a blended prior

$$P'(a) = (1-\beta)\, P_{\text{net}}(a) + \beta\, P_{\text{rag}}(a), \tag{5}$$

with $\beta \in [0,1]$ controlling the strength of retrieval. In the full design, this blended prior would be passed to a custom MCTS implementation, so that the entire tree expansion is guided by a mixture of network and memory.

In the experiments reported here, we adopt a simpler but effective strategy based on *forced exploration*. When a retrieved context is relevant but not high-confidence, we identify its top-m recommended moves and increase their prior mass by a factor $\kappa > 1$:

$$\tilde{P}(a) \propto \begin{cases} \kappa\, P_{\text{net}}(a) & \text{if } a \in \mathcal{A}_{\text{rag}}, \\ P_{\text{net}}(a) & \text{otherwise,} \end{cases}$$

where $\mathcal{A}_{\text{rag}}$ is the set of moves suggested by memory; $\tilde{P}$ is then renormalized to a distribution. This forces the MCTS to allocate more visits to memory-suggested actions while still allowing the network prior to dominate when retrieval is uninformative. When relevance is extremely high (exact symmetry match and $\text{Rel} \geq \tau$), we bypass this mechanism and simply adopt the cached best move.

This combination of uncertainty gating, retrieval, recursive deep search, and prior modification defines the full DataGo decision pipeline:

$$s \xrightarrow{f_\theta} \pi_\theta, V_\theta \xrightarrow{\text{shallow MCTS}} \pi_{\text{MCTS}} \xrightarrow{\mathcal{U}} \text{gate} \xrightarrow{\text{memory+deep search}} \pi_{\text{RAG}} \xrightarrow{\text{MCTS w/ modified priors}} a^\star$$

5 Experiments

We now describe each experiment, including configurations, failure modes, and metrics, all backed by logs in `datago` and JSONs in `tuning/tuning_results`.

5.1 Experiment A: Quick Recursive Test (Synthetic)

Goal. Verify that the recursive deep-search pipeline triggers correctly and stores multi-context positions.

Setup.

- Script: `run_datago_recursive_match.py` with synthetic policies (Dirichlet) and mock ANN.
- Config: $\theta_{\text{deep}} = 0.35$, `max_recursion_depth`$= 2$, `deep_mcts.max_visits`$= 2{,}000$, move cap per game $= 20$.
- Log: `quick_test_20251116_154828.log`.

A prior failed run in `TEST_RESULTS_ANALYSIS.md` with $\theta_{\text{deep}} = 0.40$ showed *no* activation: average uncertainty $\bar{\mathcal{U}} = 0.361$ with range $[0.349, 0.374]$ never exceeded 0.40, leading to 0 RAG queries and 0 deep searches despite nontrivial uncertainty.

After lowering θ_{deep} to 0.35, the successful run produced:

- $19/20$ moves with deep search (95% activation under synthetic policies).
- 19 RAG queries, all misses (empty database), and 19 stored positions.
- Average uncertainty $\bar{\mathcal{U}} = 0.359$.
- First deep search cost ≈ 2.5 s, subsequent recursion-driven deep searches 50–500 ms.

This validated the deep-search trigger and storage pipeline but not cache reuse or real-network behavior.

5.2 Experiment B: Extended Synthetic Match (10 games)

Goal. Stress-test recursion and storage under heavy activation and quantify the strength of DataGo with synthetic policies.

Setup.

- Script: `run_datago_recursive_match.py` with synthetic Dirichlet policies, RAG, and deep search.
- Config: $\theta_{\text{deep}} = 0.370$, `max_recursion_depth`$= 3$, `deep_mcts.max_visits`$= 2{,}000$, $G = 10$ games, move cap $= 100$.
- Log: `extended_match_20251116_155221.log`; summary in `COMPETITIVE_RESULTS.md`.

The aggregated metrics (over all 10 games) are:

Run	Threshold θ	RAG Queries	Deep Searches	W–L–D
Extended synthetic	0.370	296	1411	9–0–1

Table 1: Synthetic 10-game match using Dirichlet policies.

Further statistics from `SYNTHETIC_VS_REAL_NN_COMPARISON.md`:

- Total moves: 468.
- Recursive searches: 1232 (87% of deep searches).
- Unique positions stored: 3017; total contexts: 3144; ≈ 1.0 context/position.
- Average uncertainty: $\bar{U} = 0.377$.
- Activation rate: $p_{\text{act}} = 296/468 \approx 0.632$.
- Cache hits: 135; hit rate $135/296 \approx 0.456$ for queries, but synthetic policies do not correspond to meaningful Go patterns.

This run shows that, when allowed to fire frequently, the recursive RAG pipeline can drive an almost $2\times$ effective visit advantage and a 90% win rate in a synthetic environment. However, because the policies are random and the database stores random patterns, the result is a stress test rather than a meaningful evaluation of Go strength.

5.3 Experiment C: Real KataGo Outputs, Untuned Threshold

Goal. Evaluate DataGo on real KataGo outputs using the synthetic-tuned threshold $\theta = 0.370$.

Setup.

- Script: `run_datago_recursive_match.py`.
- Config: same as Experiment B but network policies and values come from real KataGo via `GTPController.genmove_analyze`.
- Log: `extended_match_20251116_162336.log`.

Results (`SYNTHETIC_VS_REAL_NN_COMPARISON.md`):

Run	Threshold θ	Activation Rate	W–L–D
Real untuned	0.370	0%	0–10–0

Table 2: Real-KataGo outputs with synthetic threshold: the gate never fires.

- Total moves: 500.

- RAG queries: 0; deep searches: 0; unique positions: 0.
- Average uncertainty: $\bar{\mathcal{U}} \approx 0.120$, with maximum $\approx 0.21 < 0.37$.

Thus the gate is entirely miscalibrated: the synthetic distribution of $\mathcal{U}$ is much higher than the real one, and the RAG pipeline never engages. This is a key negative result: a RAG system tuned on synthetic data can silently be disabled on real data.

5.4 Experiment D: Real KataGo Outputs, Tuned Threshold

Goal. Retune the uncertainty threshold on real KataGo outputs and evaluate DataGo vs. KataGo in competitive matches.

Setup.

- Script: `run_datago_recursive_match.py`.
- Config: `rag_query.uncertainty_threshold` set to $\theta = 0.150$, `max_recursion_depth`$= 3$, `deep_mcts.max_visits`$= 2{,}000$, $G = 10$, move cap $= 100$.
- Log: `extended_match_20251116_162735.log`.

Aggregated metrics:

Run	θ	Data	RAG Q	Deep	Rec.	W–L–D
Real tuned	0.150	Real NN	23	328	314	8–0–2

Table 3: Real-KataGo experiment with tuned threshold.

From `SYNTHETIC_VS_REAL_NN_COMPARISON.md` and logs:

- Total moves: 454; activation rate $p_{\text{act}} = 23/454 \approx 0.051$.
- Deep searches: 328; recursive searches: 314 (95.7% of deep searches).
- Unique positions stored: 1070; contexts: 1211; ≈ 1.13 contexts/position.
- Cache hits: 152; hits/query ≈ 6.6, reflecting reuse of deep analyses within and across games.
- Average uncertainty: $\bar{\mathcal{U}} = 0.080$.
- Effective visits per move: $\approx 2{,}446$ (baseline 800 plus low-frequency deep expansions).

Despite a much lower activation rate than in the synthetic setting, DataGo still achieves an 80% win rate against KataGo with the same base visits. This provides the main empirical evidence that RAG+recursion adds real value when calibrated to the network's uncertainty distribution.

5.5 Threshold Sensitivity

Table 4 summarizes the three main configurations:

Scenario	θ	Data	Activation p_{act}	Win rate p_{win}
Real tuned	0.15	Real	0.051	0.80
Synthetic	0.37	Synthetic	0.632	0.90
Real untuned	0.37	Real	0.0	0.0

Table 4: Threshold sensitivity across synthetic and real scenarios.

In addition, cache efficiency differs sharply: synthetic runs have hit rates that are meaningful only for stress testing, whereas real tuned runs achieve 152 hits across 23 queries, i.e., 6.6 reused contexts per query, each representing a 2,000-visit KataGo subtree.

5.6 Offline Phase 2 Tuning

Phase 2 uses offline JSON logs in `tuning/tuning_results/offline_positions` and synthetic `make_dummy_offline_positions.py` outputs to evaluate deep-search configurations without running full matches.

Deep MCTS sweep. `phase2_deep_mcts.py` considers tuples $(D, \Delta_\pi, \Delta_V)$ where $D \in \{1{,}000, 2{,}000, 5{,}000, 10{,}000\}$ is the deep visit budget, and Δ_π, Δ_V parametrize convergence thresholds. `phase2_deep_mcts_results.json` records, for each config, average policy/value error relative to ground-truth deep searches, deep-search time, and convergence rate.

The best-scoring configuration according to the composite score (eq. 893–899 in the JSON) is:

$$D = 10{,}000, \quad \Delta_\pi = 0.1, \quad \Delta_V = 0.05,$$

with:

$$\text{avg policy error} = 0.056, \quad \text{avg value error} = 0.026, \quad \text{avg deep time} \approx 2{,}203\text{ms}.$$

Recursion depth sweep. `phase2_recursion_results.json` fixes the best deep-search config above and sweeps `recursion_depth_N`$\in \{1, 2, 3, 5\}$, scoring each based on a combination of error, time, and average recursion depth. The best setting is $N = 1$, with a score of -0.30 and average recursion depth 1; deeper N values slightly increase error (due to overfitting to noisy offline signals) and significantly increase compute.

In competitive matches we used a more aggressive $N = 3$ for stress testing. An interesting direction is to switch to $N = 1$ or $N = 2$ for production play once the full POLICY-blending MCTS pipeline is enabled.

6 Results and Analysis

6.1 Game-Theoretic Interpretation

Let π_{DataGo} and π_{KataGo} denote the strategies induced by the DataGo and baseline KataGo engines under the same base visit budget (800 visits per move), with DataGo allowed to perform additional deep searches on a small subset of moves. Let $\text{Score}(s_T)$ be the final score under Chinese rules.

The empirical experiments approximate:

$$\max_{\pi_{\text{DataGo}}} \min_{\pi_{\text{KataGo}}} \mathbb{E}[\text{Score}(s_T)] - \max_{\pi_{\text{KataGo}}} \min_{\pi_{\text{KataGo}}} \mathbb{E}[\text{Score}(s_T)] > 0,$$

by showing that DataGo's win rate vs. a strong KataGo baseline is 0.9 (synthetic) and 0.8 (real tuned), with identical networks and identical 800-visit shallow budgets. The extra strength derives from targeted deep searches and reuse of stored 2k-visit analyses, effectively increasing the visit count on hard positions without changing the base configuration.

6.2 Compute–Benefit Trade-offs

The logs and Phase 2 results jointly quantify the trade-off:

- Synthetic extended run: average visits per move $\approx 6{,}029$, win rate 0.9, but dominated by deep-search compute.
- Real tuned run: average visits per move $\approx 2{,}446$, win rate 0.8, with only 5.1% activations and 6.6 hits/query, giving a significantly better "win-per-visit" efficiency.
- Offline: raising D from $1{,}000$ to $10{,}000$ reduces policy error by nearly $3\times$ but multiplies deep-search time by ≈ 10.

A simple "efficiency" proxy (win rate divided by relative visits) gives:

$$\text{efficiency}_{\text{synthetic}} \approx \frac{0.90}{6029/800} \approx 0.15, \quad \text{efficiency}_{\text{real}} \approx \frac{0.80}{2446/800} \approx 0.26,$$

showing that the real tuned configuration is about 76% more efficient in terms of win rate per relative visit.

7 Limitations and Future Work

Visit asymmetry and fairness. Although both engines use 800 shallow visits per move, DataGo uses additional deep visits on activated positions; thus our experiments evaluate "DataGo at ~ 2.4k effective visits" vs. "KataGo at 800 visits". To make a stronger fairness argument, future work should run asymmetric matches (e.g., DataGo at 400 base visits plus RAG versus KataGo at 800) and measure whether retrieval compensates for reduced search.

Incomplete blending implementation. While `CustomMCTS` and blending utilities are implemented and tested in isolation, the competitive runs in `run_datago_recursive_match.py` use forced exploration rather than full policy blending at every RAG hit. A complete evaluation should enable true blended priors and measure their effect on win rate and error.

Limited statistical coverage. Our main real-network results are based on 10-game matches per configuration. While the win rates are large enough (80% vs. 0%) to be highly suggestive, more games and varied opponents (e.g., different KataGo nets, Leela Zero, human players) are needed to precisely estimate strength gains.

No pro-game ingestion yet. The `ragflow_repo` and `rag_store` infrastructure are ready to ingest SGFs and commentary into the ANN, but the experiments reported here restrict themselves to self-play and match-generated positions. Ingesting curated pro-game databases could increase cross-game hit rates and produce more meaningful retrievals.

8 Broader Impacts

DataGo demonstrates that retrieval-augmented, log-grounded search can substantially strengthen an open-source Go engine without training new networks, merely by reusing past computations. Positively, this indicates that resource-limited labs can improve strong baselines via clever search and memory rather than massive training runs. More broadly, the techniques here (uncertainty-gated retrieval, multi-context caches, recursive deep search) could transfer to other combinatorial search domains such as chess, shogi, or planning in structured environments.

On the negative side, stronger Go engines may widen the gap between humans and machines and could enable more persuasive automated analysis and training tools; however, these risks are mild compared to those in generative language or vision models. We rely only on public KataGo networks and do not train new models; our code and logs can be released under standard open-source licenses to support reproducibility.

9 Conclusion

We presented DataGo, a retrieval-augmented Go engine that wraps KataGo with an uncertainty gate, a symmetry-aware ANN memory, and recursive deep search. By calibrating the gate on real KataGo uncertainty distributions and building a multi-context rag-store from self-play and match logs, DataGo achieves $9/10$ wins in synthetic stress tests and $8/10$ wins in real-network matches at modest activation rates. Offline tuning confirms that deep visits and recursion can significantly reduce policy and value error, though with substantial compute overhead. Overall, our experiments show that selectively turning past 10k-visit analyses into instant priors is a practical and effective way to push strong Go engines beyond their baseline search capabilities.

References