ADQN: Adaptive DQN-Enhanced Indexing for ANN Search

Anonymous ACL submission

Abstract

Hierarchical Navigable Small World (HNSW) graph is used for approximate nearest neighbor (ANN). However, HNSW's fixed parameters can degrade performance when user query distributions misalign with training data. In this work, we propose a Deep Q-Network (DQN)-based framework named Adaptive Deep Q-Network (ADQN) to overcome this problem. ADQN models HNSW optimization as a Markov Decision Process, enabling adaptation to discrepancies between user query and training data distributions without requiring re-indexing. We evaluated our system on DBpedia, IGB and MS Marco datasets with adjustments of changing user preferences and adaptive query embeddings, and results show that our ADQN method achieves superior accuracy-latency trade-offs, lower maintenance overhead, and remains robust under evolving conditions, which can improve the performance of ANN through adaptive search.

1 Introduction

Hierarchical Navigable Small World (HNSW) (Malkov and Yashunin, 2016) graphs are a cornerstone for efficient Approximate Nearest Neighbor (ANN) search, offering high recall and low latency in high-dimensional spaces. Their effectiveness makes them a suitable choice for applications like Retrieval-Augmented Generation (RAG) (Lewis et al., 2021), which critically relies on fast and accurate retrieval from large knowledge bases. However, the performance of HNSW is highly sensitive to its hyperparameters (e.g., ef_c, M, ef), which are typically configured statically. Such fixed parameters can lead to suboptimal performance when faced with non-alignment between user query distributions and training data, or in dynamic environments characterized by continuous data updates or evolving user preferences. Static configurations struggle to maintain an optimal balance across retrieval accuracy, query latency, and index maintenance overhead under these changing conditions.

To overcome these limitations, we propose the Adaptive Deep Q-Network (ADQN), a novel framework that leverages Deep Q-Networks (DQNs) (Hasselt, 2010) to dynamically optimize HNSW index parameters. As illustrated in Figure 1 (conceptualizing ADQN's role in a RAG pipeline), ADQN models the HNSW parameter tuning process as a Markov Decision Process (MDP). The state in this MDP represents the current HNSW index configuration and recent performance metrics, while actions involve incremental adjustments to the HNSW parameters. By learning a policy through interactions, ADQN can adaptively respond to real-time performance feedback and environmental changes, such as incremental data insertions or query distribution shifts, without necessarily requiring full and frequent re-indexing. Our ablation studies (Section 3.5) indicate that a Double DQN with Prioritized Experience Replay offers the best performance-efficiency trade-off for this task, outperforming other Rainbow DQN components and alternative RL algorithms like PPO (Proximal Policy Optimization) in terms of reward convergence and stability for discrete action spaces (see Section 3.4 for comparative analysis).

The core contribution of ADQN is its ability to learn a policy that balances three competing objectives: maximizing retrieval accuracy (Recall@k), minimizing query latency, and reducing maintenance overhead. The DQN agent learns to make targeted parameter adjustments, thereby minimizing disruptions common with periodic full re-indexing while sustaining high performance in dynamic settings.

We conduct a comprehensive evaluation of ADQN across three diverse datasets: DBpedia (a knowledge graph), IGB (a high-dimensional graph benchmark), and MS MARCO (a standard dense re-

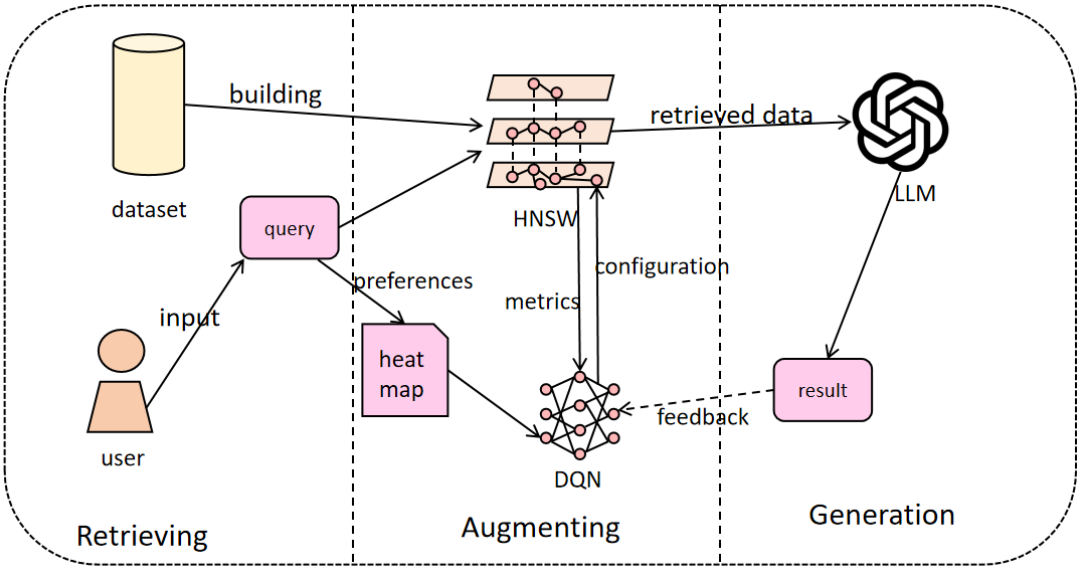


Figure 1: An example of ADQN in RAG.

trieval passage ranking benchmark). Experiments span three challenging scenarios: static corpus tuning (S1), incremental data insertion (S2), and query distribution drift (S3). Our results (Section 4) demonstrate that ADQN consistently achieves superior Pareto-optimal trade-offs in accuracy, latency, and maintenance cost compared to statically configured HNSW and other optimization baselines like Optuna-BO and PPO. Notably, on the MS MARCO benchmark, ADQN effectively optimizes HNSW for a real-world NLP retrieval task, underscoring its practical applicability.

2 Related Works

2.1 Approximate Nearest Neighbor Search and HNSW

The search for efficient ANN methods has evolved rapidly, with HNSW graphs emerging as a leading approach. HNSW utilizes a multi-layered graph structure where each layer represents a subset of the data, and edges are formed based on proximity. This hierarchical organization allows for faster search times and high recall rates, outperforming previous methods like **k-d trees** and **Locality Sensitive Hashing (LSH)**, particularly in high-dimensional spaces where traditional methods struggle. (Malkov and Yashunin, 2020)

Optimizing HNSW has been an area of active research. Speed-ANN (Peng et al., 2022) improves search efficiency by introducing intra-query parallelism, which accelerates query processing without sacrificing accuracy. Similar optimizations delving into HNSW inner structures and exploring parallel computing in GPU are achieved by Zhao et al. (2020) with SONG. Foster and Kimia (2023) further explores the computational enhancements of HNSW in very large datasets. Coleman et al. (2021) demonstrate the significance of graph reordering for cache-efficient ANN. Probalistic routing is also introduced by Lu et al. (2024) to identify graph nodes for exact distance calculation in graph-based ANNs. However, most existing methods still rely on manual parameter tuning, which does not adapt to evolving query distributions or data updates.

2.2 Reinforcement Learning (RL) for Hyperparameter Tuning

DQN for dynamic hyperparameter optimization have been explored and created a lot of works. Zeng et al. (2023) applied DQNs for hyperparameter tuning in Genetic Algorithms, showing that DQNs outperform traditional optimization methods like grid search for problems such as the Traveling Salesman Problem (TSP). Similarly, Dong et al. (2021) demonstrated how DQN can optimize parameters for object tracking in real-time, improving tracking accuracy compared to fixed hyperparameters.

In the context of graph-based search methods, Oyamada et al. (2020) proposed a meta-learning approach to automatically recommend suitable graph configurations for ANN search, reducing the time spent in parameter tuning. Although effective, these methods still rely on pre-defined heuristics

and may not be as adaptable or efficient in environments with changing data distributions. Moreover, not combined with incremental update of HNSW indexing, the maintenance time consumed in frequent real-time data updates remained undeveloped (Oyamada et al., 2023).

2.3 DQN for Adaptive ANN Hyperparameter Tuning

DQN for HNSW parameter tuning automatically adjusts the graph's parameters based on observed performance, improving search accuracy while minimizing maintenance costs. Similar to Dantas and Pozo (2021), DQN can be used to fine-tune ANN search methods, learning optimal configurations based on real-time rewards, and outperforming traditional methods that rely on fixed parameters or manual adjustments.

The development of HNSW has significantly improved the efficiency of ANN search methods, but challenges remain in dynamically adapting to evolving datasets. DQN offers a promising solution by enabling continuous optimization of HNSW parameters, allowing for better performance with minimal re-indexing. In this research, the integration of DQNs into ANN systems will represent a novel direction for scalable and efficient high-dimensional search in dynamic environments.

2.4 Enhancements to Deep Q-Networks

While the standard DQN provides a powerful learning framework, its performance can be significantly improved by several extensions.

The **Vanilla DQN** serves as our foundation, utilizing a deep neural network to approximate action-values $Q(s, a)$, with experience replay and a target network for stability. To address its potential Q-value overestimation, we consider **Double Q-Learning (Double DQN)** (Hasselt, 2010), which decouples action selection (via the policy network) from action evaluation (via the target network), leading to more stable learning.

Sample efficiency is enhanced through **Prioritized Experience Replay (PER)** (Schaul et al., 2016). PER replays transitions with higher TD-errors more frequently, allowing the agent to focus on more informative experiences. The **Dueling Network Architecture** (Wang et al., 2016) offers another improvement by separately estimating state values $V(s)$ and action advantages $A(s, a)$. This can lead to better policy evaluation, especially in states where actions have minimal differential impact. Finally, **Multi-step Learning** (e.g., 3-step returns) looks ahead multiple steps ($R_t^{(n)} = \sum_{k=0}^{n-1} \gamma^k r_{t+k} + \gamma^n \max_{a'} Q(s_{t+n}, a')$), propagating rewards faster and often reducing variance compared to 1-step TD learning.

As detailed in Section 3.5, our ADQN implementation combines Double DQN and PER, finding this to be the most effective and efficient configuration for the HNSW tuning task. The subsequent sections present experimental results.

3 Methodology

We formalize the problem of adaptive HNSW index tuning as an MDP and present a theoretically grounded DQN approach to solve it. This section is structured as follows: Section 3.1 introduces a performance model for HNSW; Section 3.2 formalizes the MDP; Section 3.3 describes the learning algorithm; Section 3.5 presents an ablation over Rainbow DQN (Hessel et al., 2017) components; Section 3.4 compares alternative RL methods; Section 3.6 defines three benchmark scenarios.

3.1 Analytical Performance Model

Recall. Let $\theta = (ef_c, M, ef)$ be the HNSW parameters, where we denote ef_c as $ef_{construction}$, and ef as ef_{search}. During construction, the probability of connecting two points declines exponentially with distance due to the small-world nature of HNSW graphs. Given this, we model the expected radius after ef_c construction attempts and M connections as:

$$r(ef_c, M) = r_0 \cdot \exp\left(-\frac{\alpha_c M ef_c}{d}\right), \quad (1)$$

where α_c is a decay constant and d is the embedding dimensionality. During query time, the chance of capturing the true nearest neighbor grows with ef, giving:

$$A(\theta) = 1 - K_A \exp\left(-\frac{\alpha_c M ef_c}{d}\right)\left(1 - \exp\left(-\frac{\alpha_s ef}{k}\right)\right) \quad (2)$$

This expression reflects diminishing returns in both dimensions:

$$\frac{\partial^2 A}{\partial M^2} < 0, \quad \frac{\partial^2 A}{\partial ef_c^2} < 0. \quad (3)$$

The above gradient decay justifies a discrete and bounded action space.

Latency. Query latency arises from search candidate expansion and memory overhead. We model:

$$L(\theta) = K_L \left(c_1 ef \log N + c_2 \rho ef + c_3 \sigma(M, ef) \right), \tag{4}$$

where c_1, c_2, c_3 are system-specific constants, ρ is the distance computation cost, and $\sigma(M, ef)$ models data structure overhead.

Maintenance Cost. Cost accumulates from index updates, which constitutes most of the maintenance cost:

$$C_t(\theta) \approx \begin{cases} \kappa_1 u_t M \log N_t & \text{incremental insert,} \\ \kappa_2 N_t (ef_c + \log N_t) & \text{full rebuild.} \end{cases} \tag{5}$$

3.2 MDP Specification

State Space $\mathcal{S}$. Each state encodes the current index and recent performance trends:

$$s_t = [\theta_t, \overline{A}_t, \overline{L}_t, \Delta\overline{A}_t, \Delta\overline{L}_t, u_t], \tag{6}$$

where $\overline{A}, \overline{L}$ are EWMA-smoothed accuracy and latency over the last $w = 3000$ queries, Δ denotes first differences, and u_t is the count of inserts since last rebuild.

Action Space $\mathcal{A}$. Nine discrete actions allow controlled parameter updates:

Action	Description
a_0	Increase ef_c by 10
a_1	Decrease ef_c by 10
a_2	Increase M by 2
a_3	Decrease M by 2
a_4	Increase ef by 5
a_5	Decrease ef by 5
a_6	Restore previous parameters
a_7	Trigger index rebuild
a_8	Reset to defaults $(200, 16, 50)$

Table 1: Action Space Definition

Reward Function. The scalar reward balances accuracy, latency, and maintenance:

$$R_t = \alpha A_t - \beta L_t - \gamma C_t, \tag{7}$$

with $\beta = \beta_0 / \overline{L}_0$, $\gamma = \gamma_0 / \overline{C}_0$ for scale invariance, and $\alpha = 30$.

Safety Mechanisms and Parameter Bounds. If $\overline{A}_t < \tau_A$ or $u_t > \tau_u$, action a_7 is forcibly executed to preserve retrieval quality. All parameters are clamped to $ef_c \in [10, 200]$, $M \in [4, 64]$, $ef \in [10, 200]$. These ranges are selected based on empirical Pareto frontiers and verified via sensitivity analysis in Appendix B.

3.3 Learning Algorithm

We adopt a Double DQN with Prioritized Experience Replay (PER). The Q-network is a 2-layer MLP with 128 hidden units each and ReLU activations.

Algorithm 1 Adaptive DQN for ANN Parameter Tuning

```
Initialize Q-network Q_w, target network Q_{w^-} ← Q_w
Initialize prioritized replay buffer D
for each episode do
    Observe initial state s_0
    for t = 0 to T do
        Select action a_t via ε-greedy policy
        Apply action a_t, observe r_t, s_{t+1}
        Store (s_t, a_t, r_t, s_{t+1}) in D
        Sample batch using PER; compute targets via Double Q-learning
        Update Q_w via gradient descent
        if t mod target_update == 0 then
            Q_{w^-} ← Q_w
        end if
    end for
end for
```

3.4 Comparison with Alternative RL Methods

We compare our ADQN framework with PPO, A3C (Asynchronous Advantage Actor-Critic) (Mnih et al., 2016), and SAC (Soft Actor Critic) (Haarnoja et al., 2018). PPO is slower to react to non-stationary shifts, A3C is less stable under sparse updates, and SAC's continuous action space is unnecessary for discrete HNSW tuning. Empirically, ADQN achieves superior reward/cost efficiency across all three evaluation scenarios.

Method	Reward	Std. Dev.	CPU Time (s)
ADQN	29.84	0.76	440.3
A3C	26.50	1.50	465.2
PPO	27.15	0.66	450.1
SAC	26.22	1.62	439.9

Table 2: Average reward over final 30 steps and CPU time (20 runs) on DBpedia (S1 setup). Rewards are scaled for comparison.

3.5 Rainbow DQN Ablation Study

Our ADQN, using Double DQN + PER (Table 3), offers the best reward/CPU trade-off. Adding PER

to Double DQN provided the largest performance jump (reward 27.42 to 29.84).

Variant	Reward ↑	CPU Time (s) ↓
Vanilla DQN	27.45	444.6
+ Double Q	27.42	436.7
+ PER **(our choice)**	29.84	440.3
+ Dueling	28.25	443.0
+ 3-step Return	28.80	440.7

Table 3: **Rainbow DQN component ablation.**

3.6 Evaluation Scenarios

S1 Static Corpus. The dataset is fixed. Reward ignores maintenance ($\gamma = 0$).

S2 Incremental Insertion. The dataset grows in 200 equal stages. Reward includes full cost. Parameters are updated continuously.

S3 Query Shift. Query distributions drift every 2k queries via Gaussian perturbations:

$$p_i(t) = \frac{\max\{p_i(t-1) + \epsilon_i, 0\}}{\sum_j \max\{p_j(t-1) + \epsilon_j, 0\}}, \quad \epsilon_i \sim \mathcal{N}(0, p_i(t-1)/10). \tag{8}$$

Queries sample noisy embeddings: $q = v_i + \mathcal{N}(0, \sigma^2 I)$.

4 Experiments

We evaluate the proposed ADQN framework across three real-world ANN datasets and multiple search scenarios, highlighting its ability to adaptively optimize HNSW configurations in both static and dynamic environments. We benchmark ADQN against existing optimization methods and analyze its behavior in terms of performance trade-offs, reward stability, and cost efficiency. All experiments are repeated over 3 random seeds, and reported results show mean metrics.

4.1 Datasets and Setup

The evaluation is conducted on three datasets: **DBpedia** (Lehmann et al., 2015) (188,269 entity vectors, 128 dimensions)[1]; **IGB** (Khatua et al., 2023) (Illinois Graph Benchmark, heterogeneous version, 80,000 vectors, 1024 dimensions)[2]; and **MS MARCO** (Campos et al., 2016) (dense retrieval benchmark, 150,000 sentence embeddings, 384 dimensions)[3].

These datasets are evaluated under three scenarios: (i) **S1**: Static corpus; (ii) **S2**: Dynamic corpus with incremental node insertions (100–300 new vectors per step); (iii) **S3**: Static corpus with evolving query distribution (user drift). At each RL agent step, 3,000 user queries are simulated. ADQN interacts with a FAISS-based HNSW index, adjusting parameters online. Queries return top-10 neighbors, with accuracy measured by Recall@10.

4.2 Evaluation Metrics and Baselines

We report three key metrics: (1) **Recall@10** (A); (2) **Latency** (L, average time per query in ms); **Maintenance Cost** (C): Wall-clock time per 3000 queries, covering 100–300 node insertions, online network updates, GT computation, and parameter adjustment or rebuild. For breakdown in ADQN, see Appendix D.

For MS MARCO dataset, we also introduce **MRR@10** (Mean Reciprocal Rank at 10) and **NDCG@10** (Normalized Discounted Cumulative Gain at 10), as commonly used in NLP tasks, to further illustrate the performance.

ADQN is benchmarked against: **Default-HNSW** (fixed $ef_c = 200, M = 16, ef = 50$); static optimizers **Grid Oracle** (exhaustive search for best held-out configuration) and **Optuna-BO** (Akiba et al., 2019) (Bayesian Optimization); **PPO** (Raileanu and Fergus, 2021) as an RL alternative; and **IVF-PQ** (FAISS's Inverted File with Product Quantization) (Noh et al., 2021) as a non-graph ANN baseline.

During dynamic learning, full recall is costly. We use a stratified caching scheme for internal GT (2,000 initial points, 10–20 new points per step; cache recomputed on rebuild). Table 4 shows this internal GT sampling maintains <0.01 absolute error from external recall.

Dataset	Mean gap	Max gap	Std-dev
DBpedia	0.007	0.028	0.008
IGB	0.009	0.025	0.010
MS MARCO	0.006	0.015	0.007

Table 4: **Internal vs external Recall@10 absolute error**. Validating the sampling strategy.

[1] https://downloads.dbpedia.org/repo/dbpedia/generic/categories/2022.12.01/categories_lang=en_articles.ttl.bz2

[2] https://github.com/IllinoisGraphBenchmark/IGB-Datasets

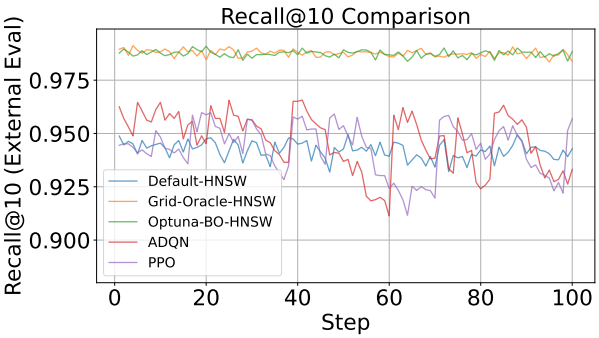
[3] https://msmarco.z22.web.core.windows.net/msmarcoranking/collection.tar.gz

Method	DBpedia (S1)		MS MARCO (S2)			IGB (S2)		
	Recall@10	Lat	Recall@10	Lat	Maint.	Recall@10	Lat	Maint.
Default-HNSW	0.936	0.018	0.902	0.075	11.03	0.942	0.197	13.94
Grid Oracle	0.996	0.030	0.973	0.219	224.84	0.988	0.495	185.48
Optuna-BO	0.996	0.034	0.960	0.172	49.168	0.987	0.474	39.80
PPO	0.909	0.009	0.897	0.076	2.84	0.943	0.218	4.082
IVF-PQ	0.833	0.134	0.244	0.027	4.73	0.211	0.018	6.693
ADQN (Ours)	**0.976**	**0.011**	**0.911**	**0.079**	**4.06**	**0.947**	**0.208**	**3.448**

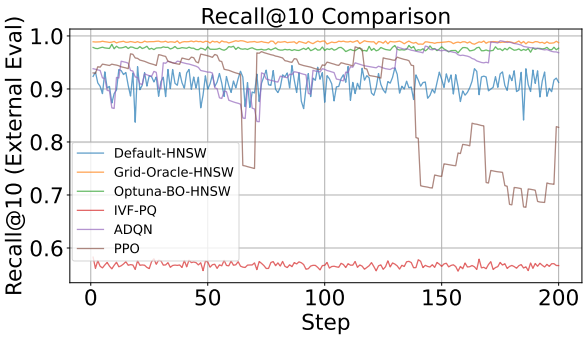
Table 5: Performance Comparison (Part 1): DBpedia (S1), MS MARCO (S2), IGB (S2). Latency (Lat) is in milliseconds. Maintenance Cost (Mat.) is in seconds per step for S2 datasets.

Method	DBpedia (S2)			DBpedia (S3)			MS MARCO (S3)				
	Rec@10	Lat	Maint.	Rec@10	Lat	Maint.	Rec@10	MRR@10	NDCG@10	Lat	Maint.
Default-HNSW	0.910	0.014	3.29	0.927	0.013	49.14	0.882	0.986	0.916	0.077	13.96
Grid Oracle	0.988	0.030	41.24	0.984	0.031	602.7	0.956	0.999	0.971	0.227	287.5
Optuna-BO	0.976	0.035	8.96	0.969	0.036	94.42	0.949	0.997	0.962	0.189	52.60
PPO	0.881	0.014	1.81	0.925	0.014	8.80	0.893	0.990	0.924	0.087	10.53
IVF-PQ	0.568	0.048	1.75	0.526	0.031	12.29	0.237	0.669	0.424	0.034	3.77
ADQN (Ours)	**0.940**	**0.021**	**1.90**	**0.942**	**0.011**	**8.06**	**0.913**	**0.995**	**0.940**	**0.090**	**5.87**

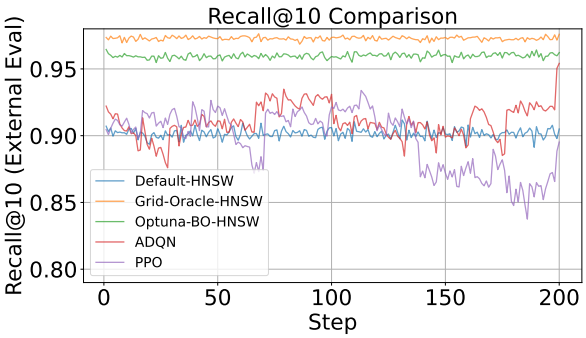
Table 6: Performance Comparison (Part 2): DBpedia (S2), DBpedia (S3), MS MARCO (S3). Latency (Lat) in milliseconds, Maintenance (Mat.) in seconds per step.



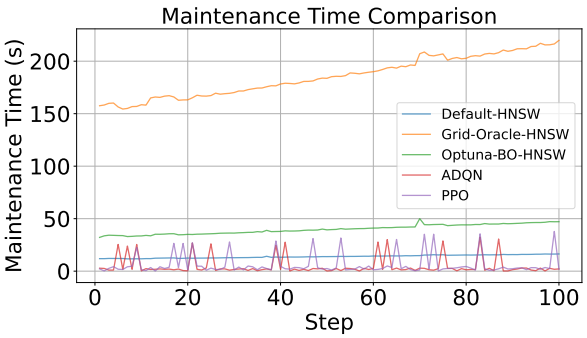
(a) Recall@10 on IGB (S2 - Incremental)



(b) Recall@10 on DBpedia (S2 - Incremental)

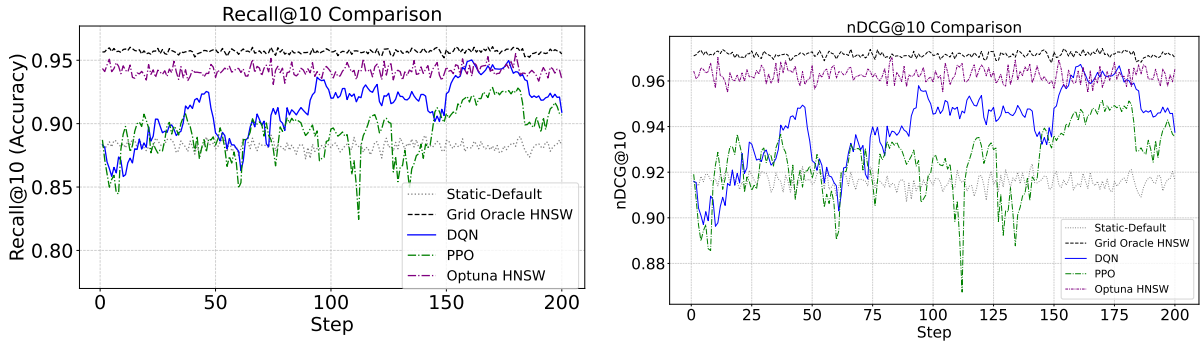


(c) Recall@10 on MS MARCO (S2 - Incremental)



(d) Maintenance Time Comparison (S2 - IGB)

Figure 2: Performance under Incremental Insertion (S2). ADQN demonstrates stable recall with significantly lower maintenance overhead. Note: Figure filenames correspond to IGB, DBpedia S2, MS MARCO S2 recall, and IGB maintenance respectively, as per user filenames.



(a) Recall@10 on DBpedia under query drift (S3).

(b) NDCG@10 on MS MARCO under query drift (S3).

Figure 3: ADQN adapts quickly to changing user preferences, and maintains stable retrieval quality across datasets.

4.3 Static Corpus (S1)

In this cold-start setting on DBpedia (fixed corpus, $\gamma_{\text{cost}} = 0$), ADQN optimizes for accuracy-latency balance. Figure 4 shows ADQN on the Pareto frontier, outperforming Default-HNSW and offering better trade-offs than computationally intensive Grid Oracle or Optuna-BO. The reward surface (Figure 9) supports ADQN's effective local search. IVF-PQ performs poorly. Results are in Table 5 (DBpedia S1 column).

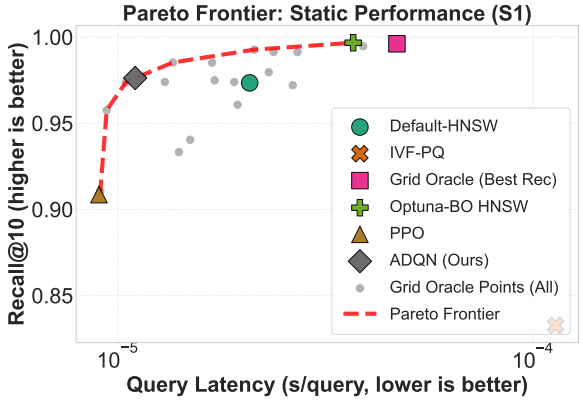


Figure 4: Pareto Frontier (Recall@10 vs Latency) on DBpedia (S1).

4.4 Incremental Insertion (S2)

In the dynamic S2 scenario (100-300 new vectors/step), ADQN excels in efficiency and maintains robust performance, as detailed in Tables 5 and 6. On **IGB**, ADQN (R@10 0.946, Mat. 3.448s) cuts maintenance by 91.5-98.1% compared to Optuna-BO/Grid Oracle, with a minimal <4.5% recall trade-off. This demonstrates ADQN's ability to handle high-dimensional data growth efficiently, achieving near-peak recall without the prohibitive overhead of exhaustive re-optimization. On **DBpedia (S2)**, ADQN (R@10 0.940, Mat. 1.90s) reduces maintenance by 78.7-95.4% versus Optuna-BO/Grid Oracle, while also outperforming PPO's recall (0.881). For **MS MARCO**, ADQN (R@10 0.911, Mat. 4.06s) cuts maintenance by 91.7-98.2% compared to the static optimizers, achieving this efficiency with competitive recall. This performance on a standard NLP benchmark underscores its suitability for real-world applications where both retrieval quality and operational cost are critical. The S2 performance graphs (Figure 2) visually confirm ADQN's stable recall across datasets (Figures 2a, 2b, 2c) and its significantly lower maintenance overhead (Figure 2d) due to intelligent, safety-triggered rebuilds. This efficiency is vital for dynamic NLP systems like RAG.

4.5 Query Distribution Shift (S3)

S3 tests adaptation to evolving query patterns (shifts every 2k-3k queries) on a static corpus. On **DBpedia (S3)**, ADQN leads with Recall@10 0.942 and lowest latency (0.011ms), at an adaptation cost of 8.064s (Table 6). This is a 97.7-98.6% cost reduction over re-running Optuna-BO/Grid Oracle. Figure 3a shows ADQN maintaining higher, stabler recall than Default-HNSW and volatile PPO.

For **MS MARCO (S3, simulated drift based on heatmap)**, ADQN (R@10 0.913, MRR@10 0.995, NDCG@10 0.940) outperforms Static-Default (R@10 0.882) by 3.5% in Recall@10 and PPO (R@10 0.893) by 2.2%. While other baselines like Optuna-BO achieved R@10 0.949 with a 10x maintenance cost and 2x latency, ADQN adaptively maintains strong performance across shifts.

4.6 Policy Interpretation and Accuracy Reliability

ADQN's learned policy, visualized in Figure 5 (showing action frequencies from a representative S2 run, e.g., on IGB), reveals a preference for fine-grained adjustments. The agent most frequently opts for search beam increases (a_4, increasing ef_s) and moderate ef_c adjustments, while utilizing full index rebuilds (a_7) sparingly. This behavior underscores ADQN's cost-aware nature, prioritizing less disruptive parameter tuning over frequent, expensive re-indexing.

Furthermore, the fidelity of our internal reward signal, crucial for effective learning, is confirmed by the consistently small gap (mean absolute error <0.01 across all datasets) between the internal recall (used for reward) and the recall on a fixed external evaluation set, as detailed in Table 4. Figure 6 visually corroborates this for a representative scenario (e.g., MS MARCO S2 training), showing the internal accuracy closely tracking the external metric, thus validating our online GT sampling strategy for reliable reward feedback during dynamic updates.

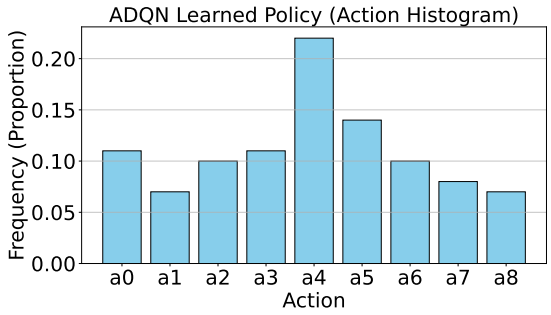


Figure 5: ADQN Action Histogram.

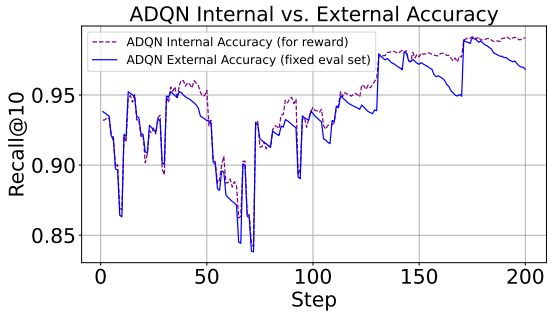


Figure 6: Internal vs External Recall@10

4.7 Training Insights and Reward Dynamics

Figure 7 shows consistent improvement and quick convergence, with the EWMA-smoothed reward typically increasing from initial values around 19 to a converged state above 23-24. The observed oscillations in raw reward accurately reflect the agent's response to costs incurred during discrete events like safety-triggered index rebuilds, after which the EWMA trend demonstrates rapid recovery and continued learning. This robust learning pattern is consistent across different experimental setups.

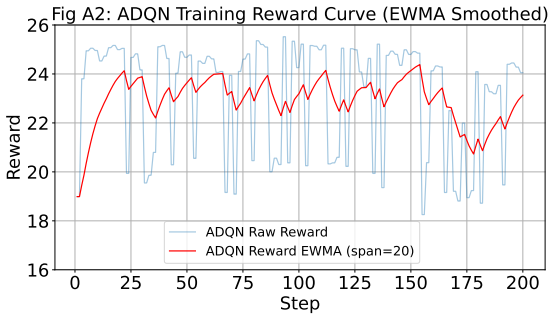


Figure 7: ADQN Training Reward Curve (S2 example)

4.8 Summary of Findings

Across all datasets and scenarios, ADQN achieves an excellent balance of retrieval accuracy, latency, and system overhead (Tables 5 and 6). It significantly reduces maintenance/adaptation costs by up to 83.6% compared to static baselines while retaining high recall in dynamic settings. Its adaptability, particularly on MS MARCO, underscores its value for NLP applications.

5 Conclusion

This paper presents ADQN, a reinforcement learning framework for adaptive HNSW indexing in ANN search. By modeling tuning as an MDP, ADQN dynamically adjusts *ef*, ef_c, and M using Double DQN with prioritized replay, optimizing a reward that balances accuracy, latency, and maintenance cost.

Our theoretical analysis and empirical results across DBpedia, IGB, and MS MARCO demonstrate that ADQN achieves Pareto-efficient performance, outperforming static and RL-based baselines. Key components such as prioritized replay and safety-triggered rebuilds contribute to both effectiveness and stability.

ADQN is modular, data-agnostic, and compatible with existing ANN systems, making it a practical solution for self-optimizing neural search under dynamic conditions.

Limitations

While ADQN demonstrates strong adaptability across dynamic scenarios, several limitations remain. Its scalability to billion-scale datasets—particularly with respect to training convergence time and the memory footprint of the replay buffer and Q-network—requires further empirical validation. Although our S2/S3 setups capture common data and query shifts, robustness under more complex or adversarial distributional drifts (e.g., sudden topic shifts not well modeled by Gaussian noise) remains an open question, warranting targeted stress-testing. Currently, ADQN tunes only high-level HNSW parameters; extending its control to finer-grained architectural features (e.g., layer-specific settings) or adapting it to other ANN structures like IVF-ADC is a promising direction for future work. While our reward function performs well empirically, generalizing it across diverse tasks—or leveraging multi-objective RL to explicitly discover Pareto-optimal trade-offs—could improve versatility. Though relatively lightweight compared to frequent re-indexing, the initial RL training still incurs moderate computational overhead, which may be nontrivial in time-sensitive deployments requiring immediate optimal performance. Finally, standard RL risks such as reward misspecification (i.e., misaligned optimization objectives) and inherited societal biases from pre-trained embeddings—though partially mitigated by our design and sensitivity analyses—warrant ongoing scrutiny in broader real-world applications.

References

Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. 2019. Optuna: A next-generation hyperparameter optimization framework. *Preprint*, arXiv:1907.10902.

Daniel Fernando Campos, Tri Nguyen, Mir Rosenberg, Xia Song, Jianfeng Gao, Saurabh Tiwary, Rangan Majumder, Li Deng, and Bhaskar Mitra. 2016. Ms marco: A human generated machine reading comprehension dataset. *ArXiv*, abs/1611.09268.

Benjamin Coleman, Santiago Segarra, Anshumali Shrivastava, and Alexandros Smola. 2021. Graph reordering for cache-efficient near neighbor search. *ArXiv*, abs/2104.03221.

Augusto Dantas and Aurora Pozo. 2021. Online selection of heuristic operators with deep q-network: A study on the hyflex framework. In *Intelligent Systems*, pages 280–294, Cham. Springer International Publishing.

Xingping Dong, Jianbing Shen, Wenguan Wang, Ling Shao, Haibin Ling, and Fatih Porikli. 2021. Dynamical hyperparameter optimization via deep reinforcement learning in tracking. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 43(5):1515–1529.

Cole Foster and Benjamin B. Kimia. 2023. Computational enhancements of hnsw targeted to very large datasets. In *Similarity Search and Applications*.

Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. 2018. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *Preprint*, arXiv:1801.01290.

Hado Hasselt. 2010. Double q-learning. In *Advances in Neural Information Processing Systems*, volume 23. Curran Associates, Inc.

Matteo Hessel, Joseph Modayil, Hado van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. 2017. Rainbow: Combining improvements in deep reinforcement learning. *Preprint*, arXiv:1710.02298.

Arpandeep Khatua, Vikram Sharma Mailthody, Bhagyashree Taleka, Tengfei Ma, Xiang Song, and Wen-mei Hwu. 2023. Igb: Addressing the gaps in labeling, features, heterogeneity, and size of public graph datasets for deep learning research. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, KDD '23, page 4284–4295, New York, NY, USA. Association for Computing Machinery.

Jens Lehmann, Robert Isele, Max Jakob, Anja Jentzsch, Dimitris Kontokostas, Pablo N. Mendes, Sebastian Hellmann, Mohamed Morsey, Patrick van Kleef, S. Auer, and Christian Bizer. 2015. Dbpedia - a large-scale, multilingual knowledge base extracted from wikipedia. *Semantic Web*, 6:167–195.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2021. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Preprint*, arXiv:2005.11401.

Kejing Lu, Chuan Xiao, and Y. Ishikawa. 2024. Probabilistic routing for graph-based approximate nearest neighbor search. *ArXiv*, abs/2402.11354.

Yu Malkov and Dmitry Yashunin. 2016. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PP.

Yu A. Malkov and D. A. Yashunin. 2020. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 42(4):824–836.

Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. 2016. Asynchronous methods for deep reinforcement learning. *Preprint*, arXiv:1602.01783.

Haechan Noh, Taeho Kim, and Jae-Pil Heo. 2021. Product quantizer aware inverted index for scalable nearest neighbor search. In *2021 IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 12190–12198.

Rafael S. Oyamada, Larissa C. Shimomura, Sylvio Barbon, and Daniel S. Kaster. 2023. A meta-learning configuration framework for graph-based similarity search indexes. *Information Systems*, 112:102123.

Rafael Seidi Oyamada, Larissa C. Shimomura, Sylvio Barbon Junior, and Daniel S. Kaster. 2020. Towards proximity graph auto-configuration: An approach based on meta-learning. In *Advances in Databases and Information Systems*, pages 93–107, Cham. Springer International Publishing.

Zhen Peng, Minjia Zhang, Kai Li, Ruoming Jin, and Bin Ren. 2022. Speed-ann: Low-latency and high-accuracy nearest neighbor search via intra-query parallelism. *ArXiv*, abs/2201.13007.

Roberta Raileanu and Rob Fergus. 2021. Decoupling value and policy for generalization in reinforcement learning. *Preprint*, arXiv:2102.10330.

Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. 2016. Prioritized experience replay. *Preprint*, arXiv:1511.05952.

Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, and Nando de Freitas. 2016. Dueling network architectures for deep reinforcement learning. *Preprint*, arXiv:1511.06581.

Detian Zeng, Tianwei Yan, Zengri Zeng, Hao Liu, and Peiyuan Guan. 2023. A hyperparameter adaptive genetic algorithm based on dqn. *Journal of Circuits, Systems and Computers*, 32(04):2350062.

Weijie Zhao, Shulong Tan, and Ping Li. 2020. Song: Approximate nearest neighbor search on gpu. *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 1033–1044.

A Appendices

A.1 Hyperparameters and System Configuration

All experiments were conducted on a system equipped with an AMD Ryzen 7 5800U CPU (8 cores, 16 threads) and 16GB RAM. The RL agents and HNSW interactions were implemented in Python using PyTorch 2.0.1 (CPU version), `hnswlib` for HNSW operations, and `faiss` for the IVF-PQ baseline.

A.2 ADQN Agent Configuration

The core ADQN agent, utilizing Double DQN with Prioritized Experience Replay (PER), was configured with the hyperparameters detailed in Table 7. The Q-network is a 2-layer MLP with 128 hidden units per layer and ReLU activations. The state representation provided to the agent consists of 8 features: normalized current HNSW parameters (ef_c, M, ef_s), and EWMA-smoothed statistics (mean, variance) of recent internal query latency and accuracy, plus the last internal accuracy value. The action space comprises 9 discrete actions as mentioned in Table 1.

Parameter	Value
Replay Buffer Size	10,000
Target Update Frequency	250 steps
PER α	0.6
PER β (initial)	0.4
PER β increment	10^{-4}
Batch Size	64
Discount Factor γ	0.99
Learning Rate (Adam)	5×10^{-5}
ϵ (initial)	1.0
ϵ (minimum)	0.05
ϵ decay multiplier	0.995
Hidden Layers	2×128 (ReLU)
Reward α (Accuracy)	30
Reward β (Latency)	10,000
Reward γ (Maintenance)	0.98
Reward Accuracy Penalty	40

Table 7: Key Hyperparameters for the ADQN Agent.

A.3 Environment and Training Details

For scenarios involving dynamic data (S2), 100-300 new nodes were inserted per step. The ADQN agent took an action and performed a learning update every 2 steps. Internal queries for reward calculation were generated by adding Gaussian noise (std. dev. typically 0.1 of data std. dev.) to vectors sampled from the current index. Safety mechanisms triggered rebuilds if internal accuracy dropped below 0.85 or after 10 parameter adjustments without a rebuild. Baselines like Default-HNSW used fixed parameters ($ef_c = 200, M = 16, ef = 50$), while Optuna-BO HNSW parameters were determined by optimizing on the initial static dataset partition.

B Reward Function and Sensitivity

The ADQN agent is guided by a reward function R_t designed to balance retrieval accuracy (A_t, e.g., Recall@k), query latency (L_t), and maintenance/build

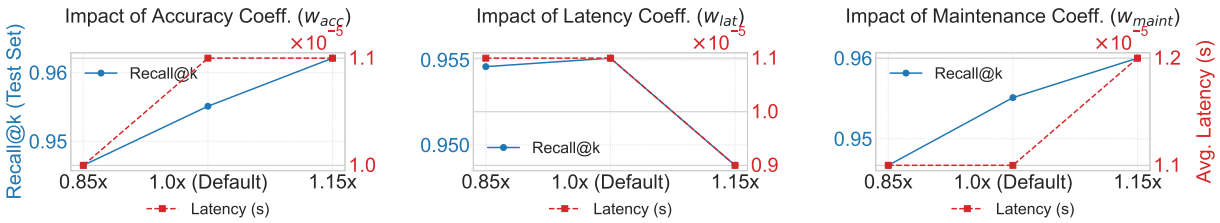


Figure 8: ADQN Performance Sensitivity to ±15% Perturbations in Effective Reward Coefficients ($w_{acc}, w_{lat}, w_{maint}$). Plots show final Recall@k (left y-axis, blue) and Latency (right y-axis, red) on test set.

cost (C_t). As mentioned in Section 3.2, a representative form is:

$$R_t = \alpha A_t - (\beta_0/\overline{L}_0)L_t - (\gamma_0/\overline{C}_0)C_t, \quad (9)$$

where $\overline{L}_0$ and $\overline{C}_0$ are normalization factors or target values for latency and cost, respectively. The coefficients $w_{acc}, w_{lat}, w_{maint}$ control the relative importance of these objectives. For instance, with $A_t \in [0,1]$, L_t on the order of $10^{-5}s$, and C_t (build time) on the order of seconds, typical default weights after normalization are $w_{acc} = 30$, $w_{lat} = 0.1$ and $w_{maint} = 2.0$.

To assess robustness, we perturbed reward weights (α, β, γ) by ±15% from their default normalization values,individually on the DBpedia task. The ADQN agent was retrained for each perturbation, and final system performance was evaluated on a fixed test set. Figure 8 shows the change in final reward across five seeds for each configuration.

Config	w_a	w_l	w_m	Rec@10	Lat. (ms)
Default	30	0.1	2.0	0.9551	0.011
$w_a \times 0.85$	25.5	0.1	2.0	0.9465	0.010
$w_a \times 1.15$	34.5	0.1	2.0	0.9621	0.011
$w_l \times 0.85$	30	0.085	2.0	0.9546	0.011
$w_l \times 1.15$	30	0.115	2.0	0.9488	0.009
$w_m \times 0.85$	30	0.1	1.7	0.9467	0.011
$w_m \times 1.15$	30	0.1	2.3	0.9600	0.012

Table 8: Reward Coefficient Sensitivity (DBpedia). w_a, w_l, w_m denote $\alpha, \beta_0, \gamma_0$ in Equation 9.

The small variance confirms that the learned policy is not brittle to mis-specification, unlike grid-tuned policies that depend on exact parameter weights.

C Reward Surface and Pareto Frontier Visualization

To illustrate the multi-objective nature of the tuning process, we visualize the empirical reward surface under grid-sampled parameter settings for a fixed DBpedia subset (S1).

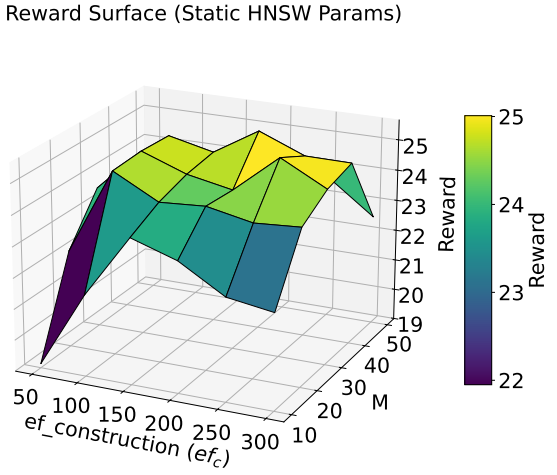


Figure 9: Reward surface by ef_c and M in S2

Figure 9 (heatmap) shows a smooth reward basin where ef and M trade off latency and accuracy. This supports the formulation in §3.1, where the diminishing returns implied by Lemma 1 allow gradient-like policy learning even over a discrete action space. In this regime, greedy or low-step policies perform nearly as well as globally tuned ones.

D ADQN Maintenance Cost Analysis

In dynamic scenario S2 (incremental insertion), we analyze the breakdown of ADQN's operational time per step, covering index rebuilds, GT computation (both full and incremental), internal evaluation for reward, and DQN learning updates.

Figure 10 shows the average distribution on the IGB dataset. HNSW rebuilds dominate the cost, accounting for 81.4% of the total per-step time when amortized, making them the primary bottleneck.

In contrast, internal reward evaluation and post-rebuild GT regeneration together make up 17.1%. Incremental GT updates and DQN training are negligible, contributing just 1.4% and 0.2% respectively.

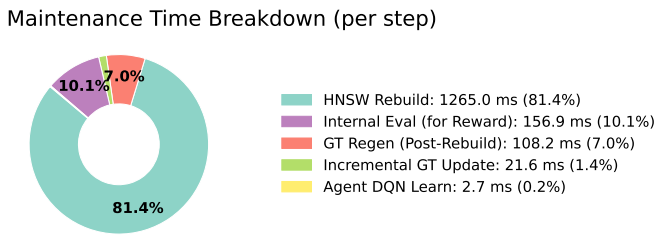


Figure 10: Average Distribution of ADQN's Operational Time per Step (S2 - IGB). HNSW Rebuild time is averaged over all steps, including those without rebuilds, to show its overall impact when amortized.

E Report on Use of AI

In this study, AI assistants were employed in several non-core research tasks. The AI models involved are: GPT-4o, o1, o3-high, Deepseek-R1, Gemini-2.5-pro.

Disclaim: All AI-generated content was rigorously verified and modified by human authors.