

QuaNAG: LLM-driven Memory-bounded Neural Architecture Generation via Quantization-branched Exploration

Yingqi Peng¹ Yifan Lin¹ Kaijie Gong² Yi Gao^{1*} Wei Dong^{1*}

¹State Key Laboratory of Blockchain and Data Security

Zhejiang Key Lab. of Accessible Perception and Intelligent Systems

College of Computer Science, Zhejiang University, Hangzhou, Zhejiang, China

²Unicom AI, China Unicom, Hangzhou, Zhejiang, China

{yingqipeng, 015, gaoyi, dongw}@zju.edu.cn, gongkj@chinaunicom.cn

Abstract

Human Activity Recognition (HAR) on memory-constrained edge devices demands heterogeneous, accurate, resource-adaptive architectures that are costly to design manually. Neural Architecture Search (NAS) provides an automated solution for model design, and its integration with quantization can further expose high-quality architectures under edge memory budgets. Recent NAS increasingly adopts LLM-based methods, including prompt-engineering and fine-tuning approaches. Prompt-engineering approaches suffer from 4-10 hours generation time, while fine-tuning approaches achieve minute-level generation but often sacrifice architecture quality due to the complexity of quantization. We propose QuaNAG, a Neural Architecture Generation (NAG) framework built on the insight that quantization is not an ordinary architecture attribute: it can change model feasibility, alter accuracy, and expand the feasible search space. QuaNAG therefore models quantization as branch-level search decisions and transfers quantization-aware search knowledge into an LLM through curriculum learning. Experiments on demanding HAR benchmarks over five baselines, including in-domain evaluations, out-of-domain evaluations, and on-device experiments, demonstrate that QuaNAG achieves 8.93% average accuracy improvement with several minutes generation time, satisfying all deployment requirements. Our code and model are available at <https://github.com/pyq623/QuaNAG>.

1 Introduction

Human Activity Recognition (HAR) systems are increasingly deployed across heterogeneous wearable and edge devices with diverse sensing configurations (Serpush et al., 2022; Jalal et al., 2014; Ranasinghe et al., 2016; Haresamudram et al.,

2025; Zhou et al., 2024). In practical HAR applications, models often need to be rapidly customized for different deployment scenarios and devices while simultaneously satisfying strict memory constraints and maintaining high recognition accuracy (Miao et al., 2024; Hong et al., 2024). However, manually designing such architectures for each dynamic deployment setting is costly and requires substantial expert effort. Neural Architecture Search (NAS) provides a natural solution by automatically discovering high-performance architectures under deployment constraints. Incorporating quantization further expands the feasible search space, enabling NAS to consider larger architectures that become deployable after quantization and may outperform smaller unquantized models (Peng et al., 2026).

Existing NAS approaches can be broadly divided into traditional methods and Large Language Model (LLM)-based methods. Traditional NAS methods, including RL-based NAS, DARTS, and OFA (Zoph and Le, 2017; Liu et al., 2019; Cai et al., 2020), can discover strong architectures but typically require several GPU days of search. Recently, LLM-based NAS methods have significantly reduced search time through either prompt engineering or fine-tuning. Prompt-engineering approaches (Chen et al., 2023; Zhou et al., 2025; Sun et al., 2025; El Zeinaty et al., 2025) iteratively generate architectures through retries and feedback, which typically requires hours of search. Moreover, during this iterative process, they frequently fail to satisfy memory constraints due to limited prior knowledge of architecture-memory relationships, making them less suitable for rapidly customized HAR deployment. Figure 1 shows that prompt-engineering methods exhibit intermediate candidate failure rates of up to 83.5% and 47.1% in search spaces without and with quantization, respectively.

In contrast, fine-tuning approaches, such as LM-

*Corresponding authors.

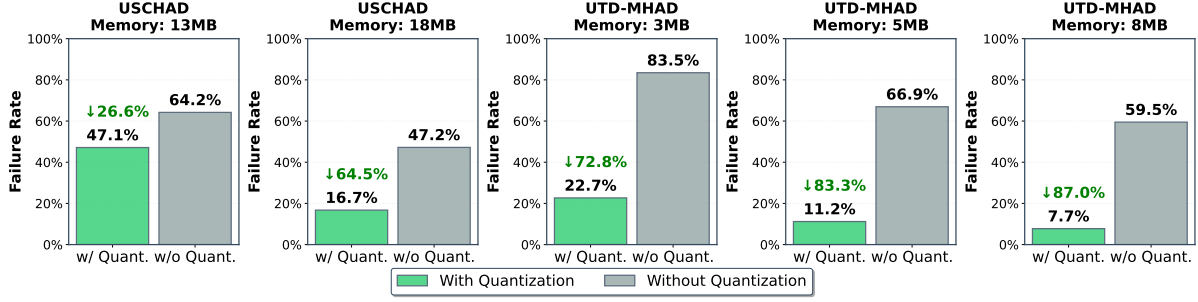


Figure 1: The intermediate candidate failure rates of prompt engineering approaches in the search space with quantization and without quantization across different memory constraint and datasets. All methods are executed 1,500 iterations to collect the final statistics.

Searcher (Hu et al., 2025), directly generate architectures in one shot within minutes, forming a fundamentally different paradigm: Neural Architecture Generation (NAG). NAG enables rapid architecture customization for edge HAR deployment under strict memory constraints. However, existing NAG methods struggle to achieve high-accuracy architectures with memory budgets in quantization search spaces, where quantization can enlarge the feasible search space by at least 6000 times (analyzed in Section C) and introduce complex accuracy degradation.

Therefore, we propose QuaNAG, a memory-bounded Neural Architecture Generation framework that jointly pursues fast generation and high accuracy. Our key insight is that quantization should not be treated as an ordinary architecture attribute. Existing LLM-based methods typically treat it in the same way as other architectural choices. However, unlike architectural choices such as convolution type, kernel size, depth, or others, quantization can reduce the accuracy of a high-performing model by over 20%, turn an originally over-budget model into a deployable one, and expand the feasible search space by thousands of times. Based on this insight, QuaNAG elevates quantization into branch-level search decisions, while keeping ordinary architectural choices as node-level decisions. It then transfers the resulting quantization-aware search knowledge into an LLM by curriculum learning for fast architecture generation. Achieving this goal requires overcoming two challenges.

First, incorporating quantization significantly enlarges the search space and complicates the memory-accuracy trade-off. Different quantization strategies may preserve accuracy for some architectures but severely degrade performance for others. To address this, we design a quantization-branched

Monte Carlo Tree Search (MCTS) algorithm that explicitly models quantization modes as macro-level search branches. By performing branch-aware exploration and backpropagation, QuaNAG separates quantization-specific search statistics from ordinary architectural choices, enabling more effective exploration and exploitation of quantized architecture spaces.

Second, NAG methods often struggle to achieve accuracy comparable to prompt-engineering approaches, because one-shot generation lacks iterative feedback and refinement. Worse still, they may generate architectures that violate memory constraints. To address this, we propose a constraint-to-quality curriculum pipeline that progressively teaches the LLM from constraint understanding to full architecture generation and final quality optimization.

We evaluate QuaNAG on five HAR benchmarks (Banos et al., 2014; Kong et al., 2019; Bruno et al., 2013; Zhang and Sawchuk, 2012; Chen et al., 2015). QuaNAG achieves average accuracy improvements of 9.11% and 8.93% in in-domain and out-of-domain memory settings, respectively, while reducing generation time by over 40 $\times$. In addition, quantization-branched MCTS improves search performance by up to 11.16% over prompt-engineering baselines (theoretical analysis in Section B), and on-device experiments demonstrate deployment efficiency and practical feasibility.

Our main contributions are summarized as follows: (1) We design a quantization-branched MCTS algorithm that elevates quantization into macro-level search branches, enabling more effective constrained exploration and exploitation of quantization-aware search knowledge. (2) We develop a constraint-to-quality curriculum pipeline that transfers constraint and architecture knowledge into an LLM, enabling one-shot architecture gen-

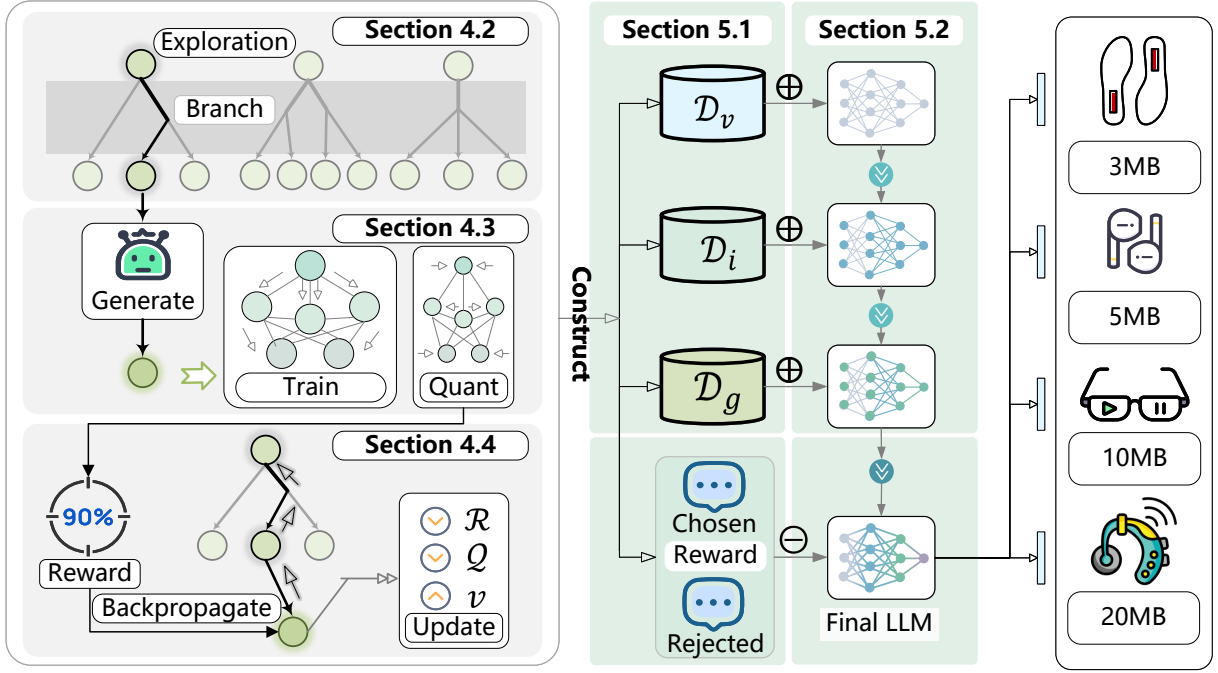


Figure 2: Overview of QuaNAG framework. The left panel shows quantization-branched MCTS (Section 4) for search knowledge acquisition, the middle panel shows constraint-to-quality curriculum (Section 5) for progressive LLM training, and the right panel shows deployment-oriented architecture generation for heterogeneous wearable devices under different memory budgets.

eration under memory constraints. (3) Extensive experiments on five HAR benchmarks demonstrate that QuaNAG achieves superior accuracy, generation efficiency, and memory compliance.

2 Related Work

2.1 Traditional NAS

Traditional NAS methods, including reinforcement learning-based NAS (Zoph and Le, 2017), differentiable NAS (Liu et al., 2019), and weight-sharing approaches such as OFA (Cai et al., 2020), have demonstrated strong performance in automated architecture design. However, these methods typically require expensive iterative search procedures and do not explicitly address memory-constrained architecture generation in quantization-expanded search spaces.

2.2 LLM-based NAS

Existing LLM-based NAS methods can be broadly divided into prompt-engineering approaches and fine-tuning approaches (Zheng et al., 2023; Nasir et al., 2024; Hu et al., 2025).

Prompt-engineering methods, including Evo-Prompting, LAPT, AutoIoT, LLM-TinyML, and RZ-NAS (Chen et al., 2023; Zhou et al., 2025; Shen et al., 2025; El Zeinaty et al., 2025; Sun et al., 2025), iteratively generate architectures through

prompting, retries, and feedback mechanisms. Although these approaches reduce search cost compared with traditional NAS and can incorporate memory constraints, they still require iterative search and often suffer from generation failures under memory constraints.

Fine-tuning approaches attempt to internalize architectural knowledge into LLMs for direct architecture generation. The recent LM-Searcher (Hu et al., 2025) demonstrates the potential of this paradigm through ranking-based reasoning over encoded architectures. However, existing fine-tuning methods still rely on search-based procedures and do not address quantization-aware memory-constrained generation.

3 Overview

The core idea of QuaNAG is to first acquire structured search knowledge through quantization-aware exploration, and then transfer this knowledge into an LLM for accurate one-shot architecture generation. To this end, QuaNAG consists of two coupled components: (1) quantization-branched MCTS, which organizes quantization modes as macro-level search branches to explore the memory-accuracy trade-off in enlarged search spaces; and (2) a constraint-to-quality curriculum, which converts the collected search knowledge into

progressive training signals that guide the LLM from constraint satisfaction to high-quality model generation.

Specifically, Figure 2 shows the search-to-generation process of QuaNAG. Quantization-branched MCTS first explores quantized architecture spaces through macro-branch exploration, generation, and backpropagation. The resulting search knowledge is then converted into curriculum datasets for progressive LLM training. After this transfer, the final LLM directly generates memory-compliant and high-quality architectures for heterogeneous wearable devices without iterative search.

4 Quantization-Branched MCTS

4.1 Problem Formulation

We consider memory-bounded neural architecture generation in a quantization-expanded search space. Let $\mathcal{S}$ denote the architecture search space, where each configuration $\theta \in \mathcal{S}$ contains both architectural attributes and a quantization mode $q \in \mathcal{Q} = \{\text{none}, \text{static}, \text{qat}, \text{qaft}\}$ (Jacob et al., 2018; Nagel et al., 2021; Jeon et al., 2025) (The detailed quantization mode is shown in Section F.1). The objective is to identify the optimal architecture under a memory constraint $M_{\max}$:

$$\theta^* = \arg \max_{\theta \in \mathcal{S}} R(\theta, \mathcal{D}) \quad \text{s.t.} \quad M(\theta) \leq M_{\max}, \quad (1)$$

where $R(\theta, \mathcal{D})$ denotes the model accuracy on dataset $\mathcal{D}$ and $M(\theta)$ represents the peak memory usage.

In this quantization-expanded setting, directly flattening q into ordinary architectural attributes entangles quantization effects with node-level choices. We therefore model quantization as a branch-level search structure.

4.2 Macro-Branch Exploration

As illustrated in Figure 3, we propose a quantization-branched MCTS that elevates quantization into macro branches. Specifically, each quantization mode $q \in \mathcal{Q}$ corresponds to a macro branch $\mathcal{B}_q$, which is visualized using different colors in Figure 3. All child nodes under the same branch share the corresponding quantization mode, and a macro branch may connect to either one or multiple child nodes. For example, root 1’s $\mathcal{B}_{\text{none}}$ has one none-quantized child node, whereas its $\mathcal{B}_{\text{QAFT}}$ has two QAFT-quantized child nodes.

Unlike standard MCTS, which have no explicit branches and only use edges to connect nodes, prior quantization-aware searches typically encode quantization as a node-level attribute together with convolution type, width, or depth.

As a result, the traversal process differs from standard MCTS. Instead of directly selecting the next child node, QuaNAG first selects a quantization branch and then traverses to child nodes associated with that branch. In the illustrated example, root node 1 first selects $\mathcal{B}_{\text{none}}$ to reach node 2. Node 2 then selects $\mathcal{B}_{\text{static}}$ and further chooses node i among nodes i, j, k . Since node i is a leaf node, the traversal selects $\mathcal{B}_{\text{QAFT}}$ to generate a new child node.

To implement this macro-branch decision, we adopt a branch-selection strategy based on Upper Confidence Bound applied to Trees (UCT):

$$\mathcal{B}_q^* = \arg \max_{q \in \mathcal{Q}} \left(Q_n^{\mathcal{B}_q} + c \cdot \sqrt{\frac{\ln \sum_{q' \in \mathcal{Q}} v_n^{\mathcal{B}_{q'}}}{v_n^{\mathcal{B}_q}}} \right), \quad (2)$$

where $Q_n^{\mathcal{B}_q}$ is the Q-value of node n in macro branch $\mathcal{B}_q$, $v_n^{\mathcal{B}_q}$ is the visit count of node n in macro branch $\mathcal{B}_q$, and c is the exploration weight. The numerator in the square root represents the total visits across all branches at node n .

This design explicitly separates exploration across different quantization modes, enabling the search process to better capture the heterogeneous memory-accuracy characteristics introduced by quantization.

The complete exploration process additionally includes root selection and node selection procedures, and the detailed formulations and traversal strategies are provided in Appendix F.2.

4.3 Generation and Evaluation

After reaching a leaf node or a maximum depth randomly sampled from a predefined range (detailed in Appendix F.2.3), we prompt the LLM to generate candidate architectures conditioned on the current search trajectory, memory constraints, the parent node and the selected macro branch.

During exploration, each search step is recorded as a triplet $(n_t, \mathcal{B}_{q_t}, n_{t+1})$, where n_t denotes the current node, $\mathcal{B}_{q_t}$ denotes the selected quantization branch, and n_{t+1} denotes the selected child node.

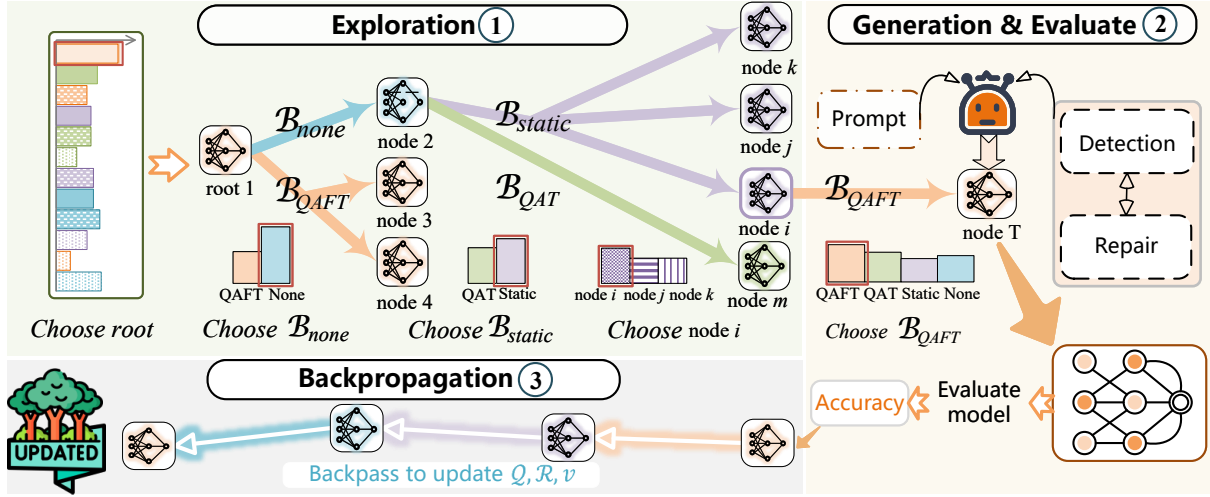


Figure 3: Overview of quantization-branched MCTS. The algorithm comprises three parts: (1) Macro-branch exploration: performs root selection, branch selection, and node selection. (2) Generation and Evaluation: generates a new node and then evaluates it. (3) Backpropagation: updates the entire trajectory.

The complete search trajectory is represented as:

$$\tau = [(n_0, \mathcal{B}_{q_0}, n_1), (n_1, \mathcal{B}_{q_1}, n_2), \dots, (n_{T-2}, \mathcal{B}_{q_{T-2}}, n_{T-1}), (n_{T-1}, \mathcal{B}_{q_{T-1}}, n_T)]. \quad (3)$$

The trajectory records both architecture transitions and quantization-aware exploration decisions, serving as the foundation for subsequent generation and backpropagation.

Each generated architecture is then trained and evaluated under its corresponding quantization mode. The evaluation result $R(\theta, \mathcal{D})$ is used as the reward for subsequent backpropagation.

4.4 Branch-aware Backpropagation

After evaluation, the reward is propagated backward through the search trajectory to update branch statistics. For each node-branch pair, we update the branch Q-value using:

$$Q_{n_t}^{\mathcal{B}_{q_t}} \leftarrow (1 - \alpha_t) Q_{n_t}^{\mathcal{B}_{q_t}} + \alpha_t \cdot \max \left(Q_{n_t}^{\mathcal{B}_{q_t}}, \max_{q' \in \mathcal{Q}} Q_{n_{t+1}}^{\mathcal{B}_{q'}} \right), \quad (4)$$

where α_t is an adaptive update rate, $Q_{n_t}^{\mathcal{B}_{q_t}}$ denotes the branch Q-value of node n_t under quantization branch $\mathcal{B}_{q_t}$, and $\max_{q' \in \mathcal{Q}} Q_{n_{t+1}}^{\mathcal{B}_{q'}}$ represents the maximum branch Q-value among all quantization branches of the child node n_{t+1} . This branch-aware update propagates the best quantization-aware exploration knowledge backward through the trajectory.

As mentioned in Section 4.2, the proposed branch-aware propagation explicitly maintains quantization-level statistics, allowing the search

process to accumulate exploration knowledge across different quantization modes. This mechanism improves constrained exploration efficiency and produces search trajectories that explicitly encode the relationships among memory constraints, quantization strategies, and architecture quality for subsequent LLM adaptation.

The complete backpropagation procedure additionally includes node-level updates, detailed reward propagation, and node statistics maintenance, which are provided in Appendix F.4. We also provide a theoretical analysis of the proposed algorithm in Appendix B.

5 Constraint-to-Quality Curriculum

The goal of LLM adaptation is to transfer the search knowledge collected by quantization-branched MCTS into an LM, enabling one-shot generation of architectures that satisfy memory constraints. Instead of directly training the LLM for end-to-end generation, we adopt a constraint-to-quality curriculum strategy that progressively decomposes memory-bounded neural architecture generation into multiple capability-learning stages.

5.1 Dataset Construction

Using the results collected during quantization-branched MCTS, we construct four complementary datasets, as illustrated in Figure 8 (Appendix G.1).

The validity dataset teaches the LLM to identify invalid architectures and satisfy memory constraints. The improvement dataset shows architecture refinement from parent-child trajectory pairs. The generation dataset trains the LLM to directly

Constraint	In-domain									Out-of-domain								
	Mhealth			MMAct			Wharf			Mhealth			MMAct			Wharf		
	12MB	15MB	20MB	3MB	5MB	8MB	3MB	5MB	8MB	10MB	13MB	18MB	2MB	4MB	7MB	2MB	4MB	7MB
QuaNAG (Ours)																		
Top10@best	98.75%	99.59%	99.39%	65.33%	66.50%	66.67%	69.85%	72.73%	74.24%	97.39%	99.18%	99.19%	62.89%	64.85%	66.01%	64.24%	72.27%	73.64%
Mode	static	static	static	none	none	none	static	static	none	static	static	static	qat	none	none	static	static	static
Time	5.86m	6.02m	7.14m	10.21m	9.10m	8.23m	7.04m	7.21m	5.50m	8.30m	6.93m	6.35m	13.10m	10.30m	8.00m	11.51m	8.12m	7.52m
LM-Searcher [EMNLP'25 Main] (Hu et al., 2025)																		
Top30@best	96.76%	97.78%	97.17%	56.41%	56.41%	61.00%	66.37%	66.82%	68.18%	97.39%	97.17%	97.17%	Failed	56.41%	60.15%	62.58%	65.76%	68.18%
Mode	static	static	static	static	static	static	static	static	static	Failed	static	static	Failed	static	static	static	static	static
Time	7.63m	7.78m	9.63m	23.02m	18.43m	17.91m	20.80m	19.64m	22.12m	10.29m	9.94m	8.76m	22.32m	21.68m	17.33m	33.59m	17.71m	38.82m
RZ-NAS(quant) [ICML'25] (Sun et al., 2025)																		
Accuracy	95.12%	96.34%	98.51%	58.87%	62.50%	62.03%	62.27%	68.64%	69.77%	94.51%	95.73%	98.39%	59.46%	60.63%	61.16%	61.36%	64.10%	66.27%
Mode	static	none	qat	qat	static	qat	static	static	static	static	qat	static	static	qat	static	qat	static	qat
Time	5.16h	4.74h	4.72h	6.16h	6.52h	6.40h	5.03h	5.28h	4.89h	5.42h	5.21h	5.22h	5.61h	5.48h	5.24h	5.01h	4.94h	4.68h
RZ-NAS(none) [ICML'25] (Sun et al., 2025)																		
Accuracy	96.95%	96.95%	97.34%	59.35%	62.13%	62.66%	62.03%	68.64%	67.73%	95.12%	96.00%	98.95%	57.91%	62.45%	63.30%	60.45%	60.00%	67.27%
Mode	none	none	none	none	none	none	none	none	none	none	none	none	none	none	none	none	none	none
Time	5.78h	4.80h	5.35h	6.76h	6.70h	6.64h	5.41h	5.38h	5.30h	5.49h	5.59h	5.63h	6.03h	5.65h	5.56h	6.03h	5.05h	4.80h
LLM-TinyML(quant) [arXiv'25] (El Zeinaty et al., 2025)																		
Accuracy	86.15%	88.54%	90.41%	57.59%	58.97%	58.66%	65.45%	64.09%	64.18%	79.56%	89.02%	90.24%	59.71%	59.44%	60.95%	50.00%	65.64%	63.18%
Mode	static	static	qat	static	static	static	static	static	static	qat	qat	qat	static	static	static	static	static	static
Time	7.09h	5.81h	5.72h	8.53h	8.79h	8.67h	7.81h	8.69h	7.72h	6.13h	7.24h	7.14h	7.42h	8.62h	8.83h	7.20h	7.41h	7.49h
LLM-TinyML(none) [arXiv'25] (El Zeinaty et al., 2025)																		
Accuracy	93.90%	96.34%	98.17%	63.14%	63.89%	64.48%	69.36%	70.73%	72.64%	92.68%	96.34%	99.07%	59.21%	61.81%	64.53%	62.96%	65.45%	72.73%
Mode	none	none	none	none	none	none	none	none	none	none	none	none	none	none	none	none	none	none
Time	5.72h	6.09h	6.02h	8.25h	9.06h	9.50h	7.53h	7.82h	6.95h	6.59h	6.73h	6.72h	7.97h	8.88h	9.34h	7.24h	7.37h	7.44h

Note: In the Time row, 'h' represents hours and 'm' represents minutes.

Table 1: In-domain and out-of-domain comparison on representative benchmarks under different memory constraints.

map target requirements into complete architecture configurations, while the comparison dataset supports preference learning between architectures with different memory and accuracy characteristics. Details of all datasets are in Appendix G.1.

Together, these datasets progressively teach the LLM constraint understanding, architecture refinement, direct generation, and preference judgment.

5.2 LLM Adaptation

Based on the progressive datasets, we employ a two-phase LLM adaptation strategy to transfer the search knowledge into the LM.

Phase 1: Progressive SFT Pipeline. We sequentially fine-tune the LLM on the validity, improvement, and generation datasets. This progressive strategy first establishes memory-aware constraint understanding, then teaches architecture refinement, and finally enables direct architecture generation.

Phase 2: Reward Preference Optimization (RPO). After SFT pipeline, we further refine the LLM through preference learning based on Direct Preference Optimization (DPO). The goal is to improve the model’s ability to judge architecture quality under memory constraints.

We first define a memory-aware reward function: $\mathbb{R}(\theta) = \begin{cases} \text{Acc}(\theta), & \text{if } \text{mem}(\theta) \leq M_{\max} \\ 0, & \text{otherwise} \end{cases}$, (5)

where $\text{Acc}(\theta)$ and $\text{mem}(\theta)$ denote the accuracy and memory consumption of architecture θ .

Using this reward, we construct preference pairs and further fine-tune the LLM with DPO. Details of training procedures, and optimization objectives are provided in Appendix G.2.

6 Evaluation

6.1 Experimental Setup

Implementation. All experiments are conducted on a workstation equipped with 4 NVIDIA GeForce RTX 4090 GPUs, dual Intel Xeon Gold 6430 processors, Ubuntu, and CUDA 12.8. Qua-NAG uses LLaMA-3.1-8B as the backbone LLM and adapts it with LoRA-based fine-tuning.

Baselines. We compare Qua-NAG against two prompt-engineering methods and one fine-tuning method, including RZ-NAS (Sun et al., 2025), LLM-TinyML (El Zeinaty et al., 2025), and LM-Searcher (Hu et al., 2025). For prompt-engineering baselines, we evaluate both quantized and non-quantized search spaces.

Benchmarks. We evaluate on five HAR benchmarks: Mhealth, MMAct, Wharf, USCHAD, and UTD-MHAD (Banos et al., 2014; Kong et al., 2019; Bruno et al., 2013; Zhang and Sawchuk, 2012; Chen et al., 2015). These datasets cover diverse sensing positions, activity categories, and memory budgets.

Metrics. We report accuracy and generation/search time. For fine-tuning approaches, we report the best result among multiple generated candidates (topk@best), and the adaptation cost is

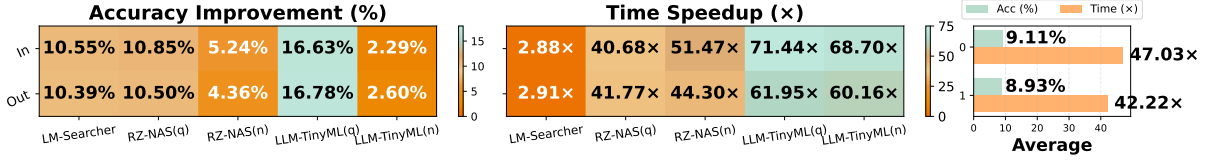


Figure 4: Performance improvements of QuaNAG over baselines in both in-domain and out-of-domain settings.

Our Search Method			Best Search Baseline			Dataset	Memory	QuaNAG			w/o RPO			w/o SFT		
Acc	Mode	Time	Method	Acc	Mode	Time		Acc	Mode	Time	Acc	Mode	Time	Acc	Mode	Time
98.78	none	2.95	RZ-NAS(none)	96.95	none	5.78	Mhealth	98.75	static	352	94.92	static	474	89.63	static	424
100.00	static	2.46	RZ-NAS(none)	96.95	none	4.80		99.59	static	361	98.58	static	461	94.10	static	342
100.00	static	4.01	RZ-NAS(quant)	98.51	qat	4.72		99.39	static	428	99.39	static	668	96.34	static	330
65.33	static	4.01	LLM-TinyML(none)	63.14	none	8.26	MMAct	65.33	none	614	65.05	none	878	61.06	none	1696
66.08	none	4.01	LLM-TinyML(none)	63.89	none	9.06		66.50	none	545	65.92	none	585	63.83	none	1057
70.03	none	4.01	LLM-TinyML(none)	64.48	none	9.51		66.67	none	492	66.46	none	578	64.44	none	752
79.09	static	4.02	LLM-TinyML(none)	69.36	none	7.54	Wharf	69.85	static	419	67.57	qat	418	65.76	static	589
77.73	none	4.01	LLM-TinyML(none)	70.73	none	7.82		72.73	static	431	73.39	qat	464	74.36	static	536
80.91	none	4.01	LLM-TinyML(none)	72.64	none	6.95		74.24	none	333	69.31	none	517	72.57	none	350

Table 2: Search evaluation and ablation study on three representative benchmarks. Left: quantization-branched MCTS versus the best prompt-engineering search baseline under each setting. Right: ablation study of the constraint-to-quality curriculum.

reported in Appendix H.5. For prompt-engineering methods, the reported time includes the iterative search process.

Memory calibration. $M(\theta)$ is a tensor-based estimate derived from parameter and intermediate activation tensors and is therefore platform independent for a fixed architecture, input size, and numerical precision. Before search and deployment, we calibrate this estimate to measured device peak memory using a small pilot set. The high correlations and low calibration errors observed on two hardware platforms support the cross-platform applicability of this memory model; detailed results are provided in Appendix E.

6.2 Main Results

Table 1 reports the in-domain and out-of-domain results on three representative benchmarks, while the complete results on all five benchmarks are provided in Appendix A.2 and A.3. Here, in-domain denotes evaluation under memory constraints used during training, while out-of-domain denotes evaluation under unseen memory constraints on the same benchmarks, testing whether the model generalizes beyond memorized training budgets.

Overall, QuaNAG consistently achieves the best performance across both in-domain and out-of-domain settings. For a conservative comparison with the fine-tuning baseline LM-Searcher, we report LM-Searcher with top30@best while using only top10@best for QuaNAG, giving LM-Searcher more generation attempts and no less total generation time. Even under this favorable

setting, QuaNAG still outperforms LM-Searcher by +10.55% and +10.39% accuracy in in-domain and out-of-domain settings, respectively, while being faster. More importantly, in the out-of-domain setting, LM-Searcher fails to generate valid architectures for 2 out of 15 unseen constraints, whereas QuaNAG successfully generates valid architectures across all configurations. This suggests that QuaNAG better generalizes to unseen memory budgets rather than merely memorizing training constraints.

Compared with prompt-engineering baselines, QuaNAG also achieves substantial accuracy and efficiency gains. In the in-domain setting, QuaNAG improves accuracy by +10.85%, +5.24%, +16.63%, and +2.29% over RZ-NAS(quant), RZ-NAS(none), LLM-TinyML(quant), and LLM-TinyML(none), respectively, while achieving 40-71 $\times$ speedup. In the out-of-domain setting, QuaNAG obtains corresponding accuracy gains of +10.50%, +4.36%, +16.78%, and +2.60%, with 42-62 $\times$ speedup. These results show that QuaNAG provides both higher generation quality and substantially better efficiency under memory constraints.

A notable observation is that prompt-engineering approaches often perform worse after incorporating quantization. For example, both RZ-NAS(quant) and LLM-TinyML(quant) generally underperform their non-quantized variants across benchmarks. This indicates that ordinary NAS pipelines struggle to effectively balance memory reduction and accuracy preservation in such a quantization-expanded search space. Additionally, Figure 4 summarizes the average gains and improvements across accu-

racy and efficiency of QuaNAG over all baselines.

Beyond the primary comparisons with LLM-based methods on HAR, we conduct two additional evaluations: a comparison with a conventional non-LLM quantization-aware NAS method (Appendix A.7), and a cross-domain comparison with the baselines on CIFAR-10 (Appendix A.8). QuaNAG outperforms the corresponding baselines in both, demonstrating advantages over conventional NAS and preliminary generalization beyond HAR.

6.3 Search Evaluation

To evaluate the contribution of the search module, we compare quantization-branched MCTS (QB-MCTS) with the strongest prompt-engineering search baseline under each benchmark and memory constraint. As shown in the left part of Table 2 (complete results in Table 7), our search consistently achieves higher accuracy than the best baseline, with improvements of up to 11.16%, while requiring less search time. These results demonstrate the effectiveness and efficiency of QB-MCTS for searching quantization-expanded architecture spaces under memory constraints.

To further examine how explicit quantization macro branches affect search behavior, we compare quantization-branched MCTS with a controlled Vanilla MCTS variant. Vanilla MCTS uses the same search space, backbone LLM, candidate-evaluation pipeline, and search budget as QB-MCTS, but treats quantization as an ordinary candidate attribute during generation and tree expansion. We measure four complementary quantities: H_{quant} , the normalized entropy of visits across quantization modes; P_{cross} , the fraction of parent-child transitions that change quantization mode; D_{arch} , the average normalized structural distance among evaluated architectures after excluding quantization mode; and N_{95} , the number of evaluations required to reach 95% of a common reference best accuracy.

Table 3 shows that QB-MCTS improves all three exploration metrics in every setting. On average, H_{quant} increases from 0.494 to 0.964, P_{cross} from 0.120 to 0.722, and D_{arch} from 0.348 to 0.405, while N_{95} decreases from 41.6 to 20.4. Complete results in Appendix D further show higher AUC and best feasible accuracy R^* in every setting. These empirical findings help explain why QB-MCTS outperforms Vanilla MCTS: explicit quantization macro branches enable broader explo-

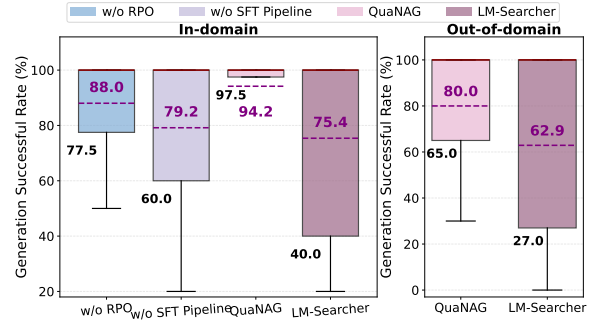


Figure 5: Comparison of generation successful rates across ablation variants and baselines. Boxes show the IQR, dashed lines denote means, and solid lines denote medians.

ration and faster discovery of high-quality feasible architectures.

6.4 Ablation Study

After validating quantization-branched MCTS in the search evaluation, we fix the search-generated data and ablate the components of the constraint-to-quality curriculum to avoid conflating search quality with curriculum learning. Specifically, w/o RPO removes reward preference optimization, and w/o SFT Pipeline removes the first two stages of the progressive SFT pipeline and retains only the generation stage. As shown in the right part of Table 2 (complete results in Table 8), QuaNAG outperforms w/o RPO and w/o SFT Pipeline by 1.53% and 2.99% in accuracy, respectively, while reducing generation time by 20.40% and 15.49%. These results confirm that both progressive SFT and reward preference optimization contribute to accurate and efficient memory-aware generation.

Figure 5 further compares generation successful rates, defined as the proportion of generated architectures satisfying all constraints. QuaNAG improves successful rates over w/o RPO and w/o SFT Pipeline by 6.2% and 15.0%, respectively, and surpasses LM-Searcher by 18.8% and 17.1% in in-domain and out-of-domain settings. These results show that the constraint-to-quality curriculum improves memory-constraint understanding rather than merely optimizing accuracy.

Complementing the curriculum ablations above, we isolate the contribution of the search method by replacing QB-MCTS with Vanilla MCTS, which removes explicit quantization macro branches, while keeping all other search and adaptation settings unchanged. As shown in Table 3, this replacement reduces the average generated accuracy from 74.80% to 70.08% (−4.72 points), showing that

Benchmark	Budget	Generation		Efficiency		Exploration Dynamics			
		Accuracy (%)		Δ		$N_{95} \downarrow$	$H_{\text{quant}} \uparrow$	$P_{\text{cross}} \uparrow$	$D_{\text{arch}} \uparrow$
Mhealth	15 MB	99.59	/ 96.02	+3.57		14 / 30	0.990 / 0.078	0.650 / 0.020	0.451 / 0.418
MMAct	3 MB	65.33	/ 60.25	+5.08		14 / 49	0.983 / 0.730	0.711 / 0.130	0.381 / 0.303
	5 MB	66.50	/ 62.53	+3.97		32 / 58	0.941 / 0.497	0.720 / 0.030	0.349 / 0.304
Wharf	3 MB	69.85	/ 62.27	+7.58		25 / 38	0.939 / 0.397	0.820 / 0.170	0.454 / 0.404
	5 MB	72.73	/ 69.35	+3.38		17 / 33	0.967 / 0.769	0.710 / 0.250	0.392 / 0.312
Average	—	74.80	/ 70.08	+4.72		20.4 / 41.6	0.964 / 0.494	0.722 / 0.120	0.405 / 0.348

Table 3: Search-knowledge ablation and exploration dynamics under identical search and adaptation settings. Except for Δ , paired entries report QB-MCTS / Vanilla MCTS; bold values correspond to QB-MCTS. Complete results, including R^* and AUC, are provided in Appendix D.

Scenario	Benchmark	Constraints	Method	End-to-End Pipeline				Model			Inference Latency	Accuracy	Acc Gap
				Request	Generation	Transfer	Total	Size	Mode	Server			
Exercise Monitoring	MMAct	8MB 500ms	QuaNAG	28ms	7.9min	0.13s	8.0min	4.04MB	none	66.77%	474ms	66.77%	0.00%
			LM-Searcher	28ms	18.1min	0.87s	18.2min	2.57MB	static	60.52%	<u>845ms</u>	60.58%	+0.06%
Industrial Operation	Wharf	10MB 150ms	QuaNAG	29ms	5.1min	0.32s	5.4min	7.71MB	none	76.36%	4.92ms	76.36%	0.00%
			LM-Searcher	29ms	15.3min	0.43s	15.8min	3.23MB	static	68.64%	75ms	69.55%	+0.91%
Health Tracking	Mhealth	12MB 50ms	QuaNAG	27ms	7.1min	0.10s	7.2min	8.23MB	static	98.78%	23ms	99.39%	+0.61%
			LM-Searcher	27ms	11.0min	0.22s	11.2min	11.24MB	static	96.34%	<u>66ms</u>	96.34%	0.00%

Table 4: Representative edge deployment comparison on Jetson devices across different HAR scenarios. Acc Gap is device-server accuracy difference.

QB-MCTS trajectories provide stronger supervision beyond the effect of the shared curriculum.

6.5 Edge Deployment Study

We further evaluate QuaNAG on NVIDIA Jetson devices under three representative HAR deployment scenarios with different memory and latency requirements. Table 4 reports representative results for one setting from each scenario, while detailed deployment configurations and complete results are provided in Appendix A.6.

Table 4 shows that QuaNAG achieves faster end-to-end deployment, higher device accuracy, and better latency compliance than LM-Searcher across three representative scenarios. QuaNAG reduces total deployment time from 18.2min to 8.0min, 15.8min to 5.4min, and 11.2min to 7.2min on MMAct, Wharf, and Mhealth, respectively, while improving device accuracy by 6.19%, 6.81%, and 3.05%. It also satisfies all latency constraints, whereas LM-Searcher violates the MMAct and Mhealth requirements.

7 Conclusions

We study memory-bounded NAG for HAR deployment on heterogeneous wearable and edge devices, where architectures must be rapidly customized under diverse sensing scenarios and memory budgets. We propose QuaNAG, a search-to-generation framework that uses quantization-branched MCTS to acquire quantization-aware search knowledge

and a constraint-to-quality curriculum to transfer such knowledge into an LLM for one-shot architecture generation. Extensive experiments show that QuaNAG improves accuracy, generation efficiency, and real-device deployment performance across both in-domain and out-of-domain settings.

Limitations

QuaNAG is evaluated mainly on HAR, with preliminary cross-domain evidence from CIFAR-10. Its effectiveness in other long-context domains, such as multi-turn dialogue and interactive language tasks, remains untested. For such larger-scale tasks, collecting and evaluating sufficiently diverse search trajectories to cover high-quality, constraint-satisfying architectures may incur substantial computational cost. Future work will evaluate QuaNAG on broader domains and larger-scale tasks, while further exploring and improving its scalability and transferability.

Acknowledgments

We thank all reviewers for their constructive comments. This work is supported by the National Natural Science Foundation of China under grant No. 92582114 and 62272407, the Major Program of Zhejiang NSF Grant No. LD26F020002. Yi Gao and Wei Dong are the corresponding authors.

References

- Oresti Banos, Rafael Garcia, Juan A Holgado-Terriza, Miguel Damas, Hector Pomares, Ignacio Rojas, Alejandro Saez, and Claudia Villalonga. 2014. mhealth-droid: a novel framework for agile development of mobile health applications. In *Ambient Assisted Living and Daily Activities: 6th International Work-Conference, IWAAL 2014, Belfast, UK, December 2-5, 2014. Proceedings* 6, pages 91–98. Springer.
- Barbara Bruno, Fulvio Mastrogiovanni, Antonio Sgorbissa, Tullio Vernazza, and Renato Zaccaria. 2013. Analysis of human behavior recognition algorithms based on acceleration data. In *2013 IEEE International Conference on Robotics and Automation*, pages 1602–1607. IEEE.
- Han Cai, Chuang Gan, Tianzhe Wang, Zhekai Zhang, and Song Han. 2020. Once for all: Train one network and specialize it for efficient deployment. In *International Conference on Learning Representations*.
- Angelica Chen, David M. Dohan, and David R. So. 2023. [EvoPrompting: Language models for code-level neural architecture search](#). In *Advances in Neural Information Processing Systems*, volume 36, pages 31484–31520, New Orleans, LA, USA. Curran Associates, Inc.
- Chen Chen, Roozbeh Jafari, and Nasser Kehtarnavaz. 2015. Utd-mhad: A multimodal dataset for human action recognition utilizing a depth camera and a wearable inertial sensor. In *2015 IEEE International conference on image processing (ICIP)*, pages 168–172. IEEE.
- Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, and 1 others. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.
- Christophe El Zeinaty, Wassim Hamidouche, Glenn Herrou, Daniel Menard, and Merouane Debbah. 2025. Can llms revolutionize the design of explainable and efficient tinymt models? *arXiv preprint arXiv:2504.09685*. Preprint.
- Harish Haresamudram and 1 others. 2025. [Past, present, and future of sensor-based human activity recognition using wearables: A surveying tutorial on a still challenging task](#). *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 9(2).
- Zhiqing Hong, Zelong Li, Shuxin Zhong, Wenjun Lyu, Haotian Wang, Yi Ding, Tian He, and Desheng Zhang. 2024. [CrossHAR: Generalizing cross-dataset human activity recognition via hierarchical self-supervised pretraining](#). *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 8(2).
- Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2022. Lora: Low-rank adaptation of large language models. In *International Conference on Learning Representations (ICLR)*.
- Yuxuan Hu, Jihao Liu, Ke Wang, Jinliang Zheng, Weikang Shi, Manyuan Zhang, Qi Dou, Rui Liu, Aojun Zhou, and Hongsheng Li. 2025. Lm-searcher: Cross-domain neural architecture search with llms via unified numerical encoding. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics. To appear.
- Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. 2018. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2704–2713.
- Ahmad Jalal, Shaharyar Kamal, and Daijin Kim. 2014. A depth video sensor-based life-logging human activity recognition system for elderly care in smart indoor environments. *Sensors*, 14(7):11735–11759.
- Hyesung Jeon, Yulhwa Kim, and Jae-Joon Kim. 2025. L4q: Parameter efficient quantization-aware fine-tuning on large language models. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2002–2024.
- Quan Kong, Ziming Wu, Ziwei Deng, Martin Klinkigt, Bin Tong, and Tomokazu Murakami. 2019. Mmact: A large-scale dataset for cross modal human action understanding. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 8658–8667.
- Alex Krizhevsky. 2009. [Learning multiple layers of features from tiny images](#). Technical report, University of Toronto.
- David A Levin and Yuval Peres. 2020. *Markov chains and mixing times*, 2nd edition. American Mathematical Society, Providence, RI. Available at <https://pages.uoregon.edu/dlevin/MARKOV/markovmixing.pdf>.
- Jonathan Light, Yue Wu, Yiyu Sun, Wenchao Yu, Yanchi Liu, Xujiang Zhao, Ziniu Hu, Haifeng Chen, and Wei Cheng. 2025. SFS: Smarter code space search improves LLM inference scaling. In *The Thirteenth International Conference on Learning Representations (ICLR)*.
- Hanxiao Liu, Karen Simonyan, and Yiming Yang. 2019. Darts: Differentiable architecture search. In *Proceedings of the 7th International Conference on Learning Representations*.
- Shenghuan Miao and 1 others. 2024. [Goat: A generalized cross-dataset activity recognition framework with natural language supervision](#). *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 8(4).

- Markus Nagel, Marios Fournarakis, Rana Ali Amjad, Yelysei Bondarenko, Mart Van Baalen, and Tijmen Blankevoort. 2021. A white paper on neural network quantization. *arXiv preprint arXiv:2106.08295*.
- Muhammad Umair Nasir, Sam Earle, Julian Togelius, Steven James, and Christopher Cleghorn. 2024. **LL-Matic: Neural architecture search via large language models and quality diversity optimization**. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1110–1118, Melbourne, VIC, Australia. ACM.
- Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, and 1 others. 2022. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35:27730–27744.
- Yingqi Peng, Wen Zhou, Kaijie Gong, Yang Liu, Yi Gao, and Wei Dong. 2026. **Edgegen: Efficient llm-empowered model generation with quantization-aware nas**. *Proceedings of the ACM Web Conference 2026*.
- Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. 2024. Direct preference optimization: Your language model is secretly a reward model. *Advances in Neural Information Processing Systems*, 36:53728–53741.
- Suneth Ranasinghe, Fadi Al Machot, and Heinrich C Mayr. 2016. A review on applications of activity recognition systems with regard to performance and evaluation. *International Journal of Distributed Sensor Networks*, 12(8):1550147716665520.
- Fatemeh Serpush, Mohammad Bagher Menhaj, Babak Karasfi, and Hoda Sagha Masoumi. 2022. Wearable sensor-based human activity recognition in the smart healthcare system. *Computational Intelligence and Neuroscience*, 2022:1391906.
- Leming Shen, Qiang Yang, Yuanqing Zheng, and Mo Li. 2025. Autoiot: Llm-driven automated natural language programming for aiot applications. In *Proceedings of the 31st Annual International Conference on Mobile Computing and Networking, MobiCom '25*, Hong Kong, China. ACM. Acceptance ratio: 41/240 = 17.1%.
- Yuxin Sun, Yiheng Tang, Yu Zhao, and 1 others. 2025. RZ-nas: Enhancing llm-guided neural architecture search via reflective zero-cost strategy. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *ICML '25*, Vancouver, Canada. PMLR.
- David van Son, Floran de Putter, Sebastian Vogel, and Henk Corporaal. 2023. **BOMP-NAS: Bayesian optimization mixed precision nas**. In *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*.
- Mi Zhang and Alexander A Sawchuk. 2012. Usc-had: A daily activity dataset for ubiquitous activity recognition using wearable sensors. In *Proceedings of the 2012 ACM conference on ubiquitous computing*, pages 1036–1043.
- Mingkai Zheng, Xiu Su, Shan You, Fei Wang, Chen Qian, Chang Xu, and Samuel Albanie. 2023. **Can GPT-4 perform neural architecture search?** *arXiv preprint arXiv:2304.10970*.
- Xun Zhou, Xingyu Wu, Liang Feng, Zhichao Lu, and Kay Chen Tan. 2025. **Design principle transfer in neural architecture search via large language models**. In *Proceedings of the 39th AAAI Conference on Artificial Intelligence*, volume 39, pages 23000–23008, Philadelphia, PA, USA. AAAI Press.
- Yexu Zhou, Tobias King, Haibin Zhao, Yiran Huang, Till Riedel, and Michael Beigl. 2024. **Mlp-har: Boosting performance and efficiency of har models on edge devices with purely fully connected layers**. In *Proceedings of the 2024 ACM International Symposium on Wearable Computers*, pages 133–139. Association for Computing Machinery.
- Barret Zoph and Quoc V Le. 2017. Neural architecture search with reinforcement learning. In *Proceedings of the European Conference on Computer Vision*.

A Detailed Evaluation

A.1 Detailed Setup

A.1.1 Baselines

Our experimental baselines consist of five methods, four of which are prompt engineering approaches and one is a fine-tuning approaches. The prompt engineering approaches: RZ-NAS(quant) (ICML'25), RZ-NAS(none) (ICML'25), LLM-TinyML(quant) (arXiv'25), and LLM-TinyML(none) (arXiv'25). The fine-tuning approach is represented by LM-Searcher (EMNLP'25 Main). RZ-NAS (quant) denotes RZ-NAS operating on a search space with quantization, while RZ-NAS (none) represents RZ-NAS operating on a search space without quantization. The same notation applies to LLM-TinyML (quant) and LLM-TinyML (none). As for LM-Searcher, it operates on a search space with quantization, identical to QuaNAG. For RZ-NAS and LLM-TinyML, we follow the hyperparameter settings from their original papers, such as 1,500 search iterations for RZ-NAS and 500 for LLM-TinyML. For LM-Searcher, the number of generated models used for training is 6-9 times larger than ours, as we use 3,000-4,500 models generated by quantization-branched MCTS per benchmark while it uses 27,000-300,000 models.

	Mhealth			MMAct			Wharf			USCHAD			UTD-MHAD		
Constraint	12MB	15MB	20MB	3MB	5MB	8MB	3MB	5MB	8MB	12MB	15MB	20MB	3MB	5MB	8MB
QuaNAG (Ours)															
Top10@best	98.75%	99.59%	99.39%	65.33%	66.50%	66.67%	69.85%	72.73%	74.24%	75.41%	76.12%	75.84%	68.37%	73.02%	72.87%
Mode	static	static	static	none	none	none	static	static	none	none	none	none	none	none	none
Time	5.86m	6.02m	7.14m	10.21m	9.10m	8.23m	7.04m	7.21m	5.50m	5.54m	5.52m	5.23m	5.61m	4.30m	4.24m
LM-Searcher(Hu et al., 2025)															
Top30@best	96.76%	97.78%	97.17%	56.41%	56.41%	61.00%	66.37%	66.82%	68.18%	56.50%	56.50%	59.50%	48.84%	52.33%	55.81%
Mode	static	static	static	static	static	static	static	static	static	static	static	static	static	static	static
Time	7.63m	7.78m	9.63m	23.02m	18.43m	17.91m	20.80m	19.64m	22.12m	29.73m	28.31m	25.20m	14.94m	18.12m	15.63m
RZ-NAS(quant)(Sun et al., 2025)															
Accuracy	95.12%	96.34%	98.51%	58.87%	62.5%	62.03%	62.27%	68.64%	69.77%	52.87%	52.72%	56.84%	45.05%	55.35%	55.07%
Mode	static	none	qat	qaft	static	qaft	static	static	static	qaft	qaft	qaft	qat	qat	qat
Time	5.16h	4.74h	4.72h	6.16h	6.52h	6.40h	5.03h	5.28h	4.89h	4.71h	4.79h	4.61h	5.15h	4.98h	4.92h
RZ-NAS(none)(Sun et al., 2025)															
Accuracy	96.95%	96.95%	97.34%	59.35%	62.13%	62.66%	57.05%	66.14%	67.73%	70.80%	70.82%	73.73%	60.00%	63.72%	70.77%
Mode	none	none	none	none	none	none	none	none	none	none	none	none	none	none	none
Time	5.78h	4.80h	5.35h	6.76h	6.70h	6.64h	5.41h	5.38h	5.30h	4.79h	4.96h	4.87h	5.46h	5.49h	5.31h
LLM-TinyML(quant)(El Zeinaty et al., 2025)															
Accuracy	86.15%	88.54%	90.41%	57.59%	58.97%	58.66%	65.45%	64.09%	64.18%	52.44%	53.04%	52.97%	32.09%	38.60%	42.09%
Mode	static	static	qat	static	static	static	static	static	static	qat	qat	qat	static	qat	qat
Time	7.09h	5.81h	5.72h	8.53h	8.79h	8.67h	7.81h	8.69h	7.72h	7.61h	7.86h	7.80h	7.32h	7.97h	7.80h
LLM-TinyML(none)(El Zeinaty et al., 2025)															
Accuracy	93.90%	96.34%	98.17%	63.14%	63.89%	64.48%	69.36%	70.73%	72.64%	70.06%	71.74%	75.47%	67.44%	70.49%	72.42%
Mode	none	none	none	none	none	none	none	none	none	none	none	none	none	none	none
Time	5.72h	6.09h	6.02h	8.25h	9.06h	9.50h	7.53h	7.82h	6.95h	9.01h	6.59h	6.81h	7.84h	7.21h	6.38h

Note: In the Time row, 'h' represents hours and 'm' represents minutes.

Table 5: In-domain comparison across benchmarks and methods under different memory constraints.

A.1.2 Benchmarks

QuaNAG is evaluated on five diverse and challenging HAR benchmarks spanning 12 to 35 activity classes across various body positions (wrist, hip, chest, ankle, etc.) and application scenarios: Mhealth (Banos et al., 2014), MMAct(Kong et al., 2019), Wharf(Bruno et al., 2013), USCHAD(Zhang and Sawchuk, 2012), and UTD-MHAD(Chen et al., 2015). These benchmarks feature varying data scales (from 569 to 7,843 samples) and sensor configurations (3-23 channels), demonstrating QuaNAG’s adaptability to both data-scarce and data-rich scenarios while covering activities from simple locomotion to complex sports movements and occupational tasks.

A.1.3 Metrics

For all prompt engineering baselines, the evaluation metrics are their accuracy and search time after completing all iterations. For fine-tuning approaches like LM-Searcher and QuaNAG, the metrics are their accuracy and generation time after 30 runs and 10 runs, respectively. Given that LM-Searcher’s generation time varies across benchmarks, outpacing our method on some while lagging on others, we adopt a conservative comparison strategy to ensure fairness. We allow LM-Searcher to generate 30 candidates and select the best one (top30@best), guaranteeing it has more generation

time than our method, which generates only 10 candidates (top10@best).

A.2 In-Domain Experiment

During model training, we impose memory constraints of 12, 15, and 20 MB for Mhealth; 3, 5, and 8 MB for MMAct; 3, 5, and 8 MB for Wharf; 12, 15, and 20 MB for USCHAD; and 3, 5, and 8 MB for UTD-MHAD. We define experiments conducted on these benchmarks with their corresponding training memory constraints as in-domain experiments.

As shown in Table 5, QuaNAG demonstrates strong performance across all benchmarks with an average accuracy improvement of +9.11% over all baseline methods. Our approach outperforms the fine-tuning baseline LM-Searcher by +10.55% while achieving $2.88\times$ faster generation times (6.45 minutes vs. 18.59 minutes on average). Against prompt engineering baselines, QuaNAG shows substantial advantages over quantization-enabled methods, with improvements of +10.85% over RZ-NAS(quant) and +16.63% over LLM-TinyML(quant), while remaining competitive with non-quantized variants (+5.24% vs. RZ-NAS(none) and +2.29% vs. LLM-TinyML(none)). Notably, QuaNAG achieves dramatic efficiency gains over prompt engineering approaches, being $40\text{-}71\times$ faster (requiring only min-

utes instead of hours), which makes our approach significantly more practical for real-world deployment scenarios where rapid model generation is essential.

A.3 Out-of-Domain Experiment

In contrast to in-domain experiments that use training memory constraints, out-of-domain experiments assess generalization to unseen memory constraints. Specifically, we evaluate with memory constraints of 10, 13, and 18 MB for Mhealth and USCHAD, and 2, 4, and 7 MB for MMAAct, Wharf, and UTD-MHAD, none of which were used during model training. Table 6 presents out-of-domain results, evaluating generalization to unseen memory constraints. QuaNAG maintains strong performance with an average accuracy improvement of +8.93% across all baselines, demonstrating robust generalization beyond its training distribution. Compared to LM-Searcher, QuaNAG achieves +10.39% higher accuracy while being $2.91\times$ faster (7.35 minutes vs. 21.38 minutes). Critically, LM-Searcher fails to generate valid architectures for 2 out of 15 configurations (Mhealth at 10 MB and MMAAct at 2 MB), highlighting its inability to extrapolate to memory constraints outside its training range. In contrast, QuaNAG successfully generates valid models across all configurations, demonstrating advantages of +10.50% over RZ-NAS(quant), +4.36% over RZ-NAS(none), +16.78% over LLM-TinyML(quant), and +2.60% over LLM-TinyML(none). The efficiency gains remain substantial, with QuaNAG being $41.77\times$ faster than RZ-NAS(quant), $44.30\times$ faster than RZ-NAS(none), $61.95\times$ faster than LLM-TinyML(quant), and $60.16\times$ faster than LLM-TinyML(none). These results validate that our memory-aware LLM fine-tuning enables genuine understanding of architecture-memory relationships, allowing effective generalization to novel memory constraints while maintaining dramatic computational efficiency.

We visualize the statistical results of in-domain and out-of-domain experiments in Figure 4 and attribute these results to several key factors. First, our quantization-branched MCTS effectively explores the expanded search space with quantization, discovering high-performance models that provide a strong foundation for subsequent fine-tuning. This is evidenced by the fact that all prompt engineering approaches perform worse in the search space with quantization compared to ones without quantiza-

tion, indicating that quantization improves search complexity that conventional methods struggle to handle. Even LM-Searcher, which uses significantly more training data, fails to match QuaNAG’s performance, further demonstrating the challenge in quantization-aware search. Importantly, our best models on benchmarks like Mhealth are quantized models that outperform models from the search space without quantization, revealing that appropriate quantization can preserve or improve accuracy while reducing memory, but only when guided by a sophisticated search strategy.

Second, our constraint-to-quality curriculum effectively internalizes the mapping from architecture configurations to memory constraints and accuracy into the LLaMA model. QuaNAG’s substantial improvements on USCHAD and UTD-MHAD (exceeding +15% over baselines) demonstrate that our training dataset construction and fine-tuning approach enable the model to genuinely understand and solve memory-constrained architecture generation. More compelling evidence comes from the out-of-domain experiments, where LM-Searcher fails to generate valid architectures, unable to produce models under 10 MB for Mhealth and under 2 MB for MMAAct. This reflects a fundamental limitation: LM-Searcher mimics patterns without truly understanding the memory-architecture relationship, whereas QuaNAG develops genuine reasoning capabilities through our fine-tuning strategy.

A.4 Search Evaluation

To better validate the effectiveness of our quantization-branched MCTS, we compare it separately against other prompt engineering approaches, with results shown in Table 7. The results demonstrate that our approach consistently outperforms all prompt engineering baselines, achieving improvements of up to +11.16%. This directly validates the effectiveness of our algorithm.

Moreover, according to Table 7, in most cases, search without quantization achieves better performance among the four prompt engineering approaches. This further demonstrates that without carefully designed algorithms to leverage quantization, it is more likely to produce adverse effects. This also highlights the advantage of our quantization-branched MCTS, as our algorithm outperforms all baselines in the search space with quantization, surpassing all the best results from the search space without quantization. Beyond experiments, we provide theoretical analysis of

Constraint	Mhealth			MMAct			Wharf			USCHAD			UTD-MHAD		
	10MB	13MB	18MB	2MB	4MB	7MB	2MB	4MB	7MB	10MB	13MB	18MB	2MB	4MB	7MB
QuaNAG															
Accuracy	97.39%	99.18%	99.19%	62.89%	64.85%	66.01%	64.24%	72.27%	73.64%	72.58%	73.66%	76.76%	66.36%	72.71%	73.49%
Mode	static	static	static	qaft	none	none	static	static	static	none	none	none	none	none	none
Time	8.30m	6.93m	6.35m	13.10m	10.30m	8.00m	11.51m	8.12m	7.52m	5.23m	5.31m	5.42m	5.40m	4.64m	4.19m
LM-Searcher															
Accuracy	Failed	97.17%	97.17%	Failed	56.41%	60.15%	62.58%	65.76%	68.18%	56.50%	56.58%	59.50%	48.84%	52.33%	58.64%
Mode	Failed	static	static	Failed	static	static	static	static	static	static	static	static	static	static	none
Time	10.29m	9.94m	8.76m	22.32m	21.68m	17.33m	33.59m	17.71m	38.82m	32.01m	33.54m	26.91m	17.04m	14.88m	15.87m
RZ-NAS(quant)															
Accuracy	94.51%	95.73%	98.39%	59.46%	60.63%	61.16%	61.36%	64.10%	66.27%	51.04%	51.49%	59.46%	45.12%	52.33%	56.74%
Mode	static	qat	static	static	qaft	static	qaft	static	qaft	qaft	qat	qat	qat	qat	qaft
Time	5.42h	5.21h	5.22h	5.61h	5.48h	5.24h	5.01h	4.94h	4.68h	4.97h	4.78h	4.59h	5.35h	5.25h	5.06h
RZ-NAS(none)															
Accuracy	95.12%	96.00%	98.95%	57.91%	62.45%	63.30%	60.45%	60.00%	67.27%	70.28%	70.65%	71.10%	57.67%	67.91%	70.81%
Mode	none	none	none	none	none	none	none	none	none	none	none	none	none	none	none
Time	5.49h	5.59h	5.63h	6.03h	5.65h	5.56h	6.03h	5.05h	4.80h	5.49h	4.96h	4.76h	5.66h	5.58h	5.18h
LLM-TinyML(quant)															
Accuracy	79.56%	89.02%	90.24%	59.71%	59.44%	60.95%	50.00%	65.64%	63.18%	51.85%	54.11%	57.59%	26.28%	36.74%	39.21%
Mode	qat	qat	qat	static	static	static	static	static	static	qat	qat	qat	static	qat	qat
Time	6.13h	7.24h	7.14h	7.42h	8.62h	8.83h	7.20h	7.41h	7.49h	7.41h	7.48h	7.51h	7.58h	8.15h	8.29h
LLM-TinyML(none)															
Accuracy	92.68%	96.34%	99.07%	59.21%	61.81%	64.53%	62.96%	65.45%	72.73%	69.37%	72.06%	73.30%	63.26%	71.49%	71.95%
Mode	none	none	none	none	none	none	none	none	none	none	none	none	none	none	none
Time	6.59h	6.73h	6.72h	7.97h	8.88h	9.34h	7.24h	7.37h	7.44h	6.08h	6.79h	6.40h	7.95h	7.46h	7.65h

Note: In the Time row, 'h' represents hours and 'm' represents minutes.

Table 6: Out-of-domain comparison across benchmarks and methods under different memory constraints.

quantization-branched MCTS in Appendix B, proving its superior diversity and faster convergence.

A.5 Ablation Study

To validate the effectiveness of our constraint-to-quality curriculum mechanism, we perform ablation experiments. Our ablation variants include: (1) w/o RPO, which removes reward preference optimization, and (2) w/o SFT Pipeline, which removes the first two stages of the progressive SFT pipeline and retains only the generation stage. QuaNAG surpasses w/o RPO and w/o SFT Pipeline by 1.53% and 2.99% in accuracy, respectively, while reducing generation time by 20.40% and 15.49%. These findings support the effectiveness of our fine-tuning mechanism.

In addition to the ablation experiment, we measure the generation successful rates of these two ablation variants in in-domain experiments, as well as the generation successful rates of QuaNAG and LM-Searcher in both in-domain and out-of-domain experiments. As shown in Figure 5, the generation successful rate refers to the proportion of complete model architectures generated that satisfy the constraint requirements. QuaNAG achieves generation successful rates 6.2% and 15.0% higher than w/o RPO and w/o SFT Pipeline in in-domain experiments, with Q1 values 20.0% and 37.5% higher, respectively. This consistent high performance (Q1

> 94%) demonstrates that constraint-to-quality curriculum substantially enhances memory constraint understanding. Additionally, QuaNAG surpasses LM-Searcher by 18.8% and 17.1% in generation successful rates for in-domain and out-of-domain experiments, respectively, evidencing that our fine-tuning mechanism promotes reasoning over memorization.

A.6 Edge Deployment Study

To evaluate our approach across diverse deployment scenarios, we configure edge device specifications tailored to three HAR scenarios with varying real-time and memory requirements (Figure 6). We design these configurations based on the real-time responsiveness demands of different application contexts: exercise monitoring scenarios employ single-core devices (806-1036MHz) with 7-8MB memory, 500ms latency, and low power consumption (Mode2, 15W); industrial operations utilize dual-core configurations (806-1036MHz) with 10MB memory, 150ms latency, and moderate power budget (Mode2, 15W); health tracking scenarios demand quad-core processors (1267-1574MHz), 12-20MB memory, 50ms latency, and higher power budget (Mode3, 25W). We assign the five benchmarks to these scenarios according to their activity profiles. MMAct, UTD-MHAD, and USCHAD focus on daily physical activities

Benchmark	Memory	Our Search Method			Best Search Baseline				Improvement	
		Acc (%)	Mode	Time (h)	Method	Acc (%)	Mode	Time (h)	$\Delta\text{Acc}\uparrow$	$\Delta\text{Time}\downarrow$
Mhealth	12MB	98.78	none	2.95	RZ-NAS(none)	96.95	none	5.78	+1.83	2.83
	15MB	100.00	static	2.46	RZ-NAS(none)	96.95	none	4.80	+3.05	2.34
	20MB	100.00	static	4.01	RZ-NAS(quant)	98.51	qat	4.72	+1.49	0.71
MMAct	3MB	65.33	static	4.01	LLM-TinyML(none)	63.14	none	8.26	+2.19	4.25
	5MB	66.08	none	4.01	LLM-TinyML(none)	63.89	none	9.06	+2.19	5.05
	8MB	70.03	none	4.01	LLM-TinyML(none)	64.48	none	9.51	+5.55	5.50
Wharf	3MB	79.09	static	4.02	LLM-TinyML(none)	69.36	none	7.54	+9.73	3.52
	5MB	77.73	none	4.01	LLM-TinyML(none)	70.73	none	7.82	+7.00	3.81
	8MB	80.91	none	4.01	LLM-TinyML(none)	72.64	none	6.95	+8.27	2.94
USCHAD	12MB	81.41	none	4.00	RZ-NAS(none)	70.80	none	4.79	+10.61	0.79
	15MB	79.98	none	4.01	LLM-TinyML(none)	71.74	none	6.59	+8.24	2.58
	20MB	82.12	none	4.01	LLM-TinyML(none)	75.47	none	6.81	+6.65	2.80
UTD-MHAD	3MB	78.60	none	4.01	LLM-TinyML(none)	67.44	none	7.84	+11.16	3.83
	5MB	79.53	none	4.01	LLM-TinyML(none)	70.49	none	7.21	+9.04	3.20
	8MB	80.00	none	4.02	LLM-TinyML(none)	72.42	none	6.38	+7.58	2.36

Table 7: Performance comparison between Quantization-Branched MCTS and best baseline methods across different benchmarks and memory constraints.

Dataset	Memory	QuaNAG			w/o RPO			w/o SFT Pipeline		
		Acc. (%)	Mode	Time (s)	Acc. (%)	Mode	Time (s)	Acc. (%)	Mode	Time (s)
Mhealth	12MB	98.75	static	<u>352</u>	94.92	static	474	89.63	static	424
	15MB	99.59	static	<u>361</u>	98.58	static	461	94.10	static	342
	20MB	99.39	static	428	99.39	static	668	96.34	static	<u>330</u>
MMAct	3MB	65.33	none	<u>614</u>	65.05	none	878	61.06	none	1696
	5MB	66.50	none	<u>545</u>	65.92	none	585	63.83	none	1057
	8MB	66.67	none	<u>492</u>	66.46	none	578	64.44	none	752
Wharf	3MB	69.85	static	419	67.57	qat	<u>418</u>	65.76	static	589
	5MB	72.73	static	<u>431</u>	73.39	qat	464	74.36	static	536
	8MB	74.24	none	<u>333</u>	69.31	none	517	72.57	none	350
USCHAD	12MB	75.41	none	<u>333</u>	75.08	none	397	76.15	none	359
	15MB	76.12	none	<u>328</u>	76.66	none	927	75.80	none	326
	20MB	75.84	none	<u>314</u>	75.56	none	1118	74.93	none	330
UTD-MHAD	3MB	68.37	none	337	67.77	none	<u>263</u>	64.65	none	359
	5MB	73.02	none	<u>260</u>	67.60	none	263	64.96	none	310
	8MB	72.87	none	<u>252</u>	68.53	none	259	71.32	none	303

Table 8: Ablation study comparing QuaNAG with two variants: w/o RPO (without Reward Preference Optimization) and w/o SFT Pipeline (without Progressive SFT pipeline). We report accuracy (%), quantization mode, and generation time (seconds). Ablation results are averaged over 3 runs. Best accuracy per row in **bold** with purple background ; shortest time per row underlined with blue background .

such as running, climbing stairs, and sit-ups, categorizing them under exercise monitoring scenarios. Wharf primarily monitors wrist-based occupational activities including cargo loading/unloading and forklift operations, placing it in the industrial operations category. Mhealth captures mobile health activities such as relaxation and sedentary behaviors, aligning it with health tracking scenarios. In addition, we select LM-Searcher as the primary baseline as it represents the sota fine-tuning approach with minute-level generation time, which is more aligned with practical deployment requirements than prompt engineering methods requiring hours. Our experiments are conducted on NVIDIA Jetson platforms with device specifications strictly configured according to Figure 6 to match the re-

source constraints of edge deployment scenarios.

The experimental results are summarized in Table 11. The edge device operates on ARM architecture, while the server uses AMD architecture. This architectural difference results in different quantization engines between device and server, leading to minor accuracy variations (denoted as "Acc Gap" in Table 11). Nonetheless, QuaNAG-generated models fully satisfy all latency and memory requirements while outperforming LM-Searcher. This reflects QuaNAG’s superior capability in generating deployment-ready models that balance performance constraints and accuracy under real-world conditions.

Specifically, QuaNAG achieves $2.0\text{-}3.8\times$ faster end-to-end deployment time compared to LM-

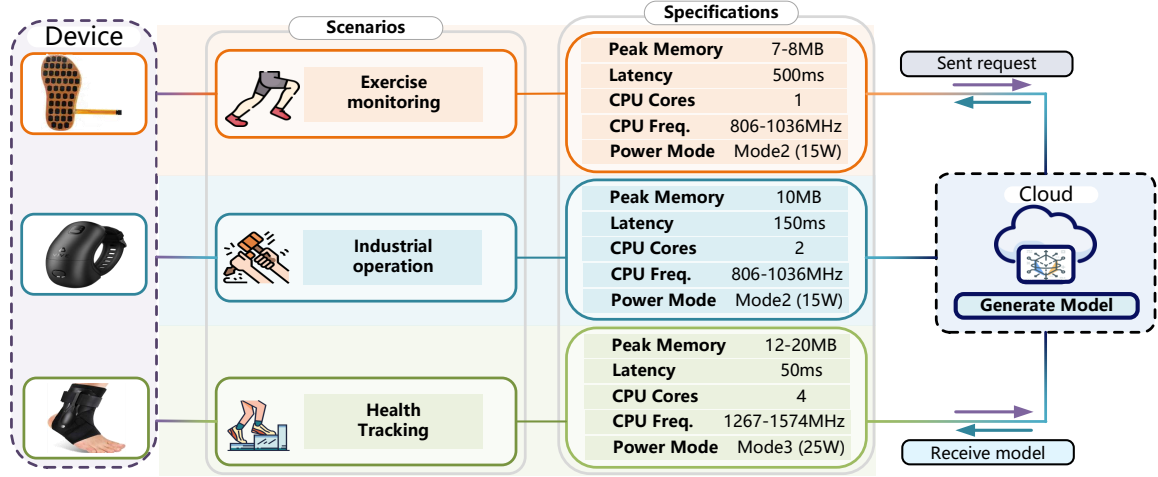


Figure 6: Overview of real-device experiments covering exercise monitoring, industrial operations, and health tracking scenarios. Experimental workflow: the edge device sends a request to the server, which then generates a model and returns the model file to the device for deployment and execution. All experiments are conducted on NVIDIA Jetson platforms with device specifications strictly configured to match the resource constraints of each scenario.

Searcher (5.7-8.0 minutes vs. 8.5-22.5 minutes). More importantly, QuaNAG consistently meets all latency constraints: all inference times remain well below the required thresholds (23-474ms vs. 50-500ms limits), whereas LM-Searcher violates latency requirements in 3 out of 7 cases (marked with red underlines). In terms of accuracy, QuaNAG surpasses LM-Searcher by 6.2-19.1% in exercise monitoring scenarios (e.g., 75.57% vs. 56.51% on USCHAD with 7MB constraint), 6.8% in industrial operation scenarios, 1.8%-3.1% in health tracking scenarios. Additionally, the accuracy gap between device and server deployment is negligible for QuaNAG (+0.61% on Mhealth with 12MB), indicating robust cross-platform consistency.

A.7 Comparison with Conventional Quantization-Aware NAS

To compare QuaNAG with a conventional non-LLM quantization-aware NAS approach, we implement BOMP-NAS (van Son et al., 2023), which combines Bayesian optimization with mixed-precision neural architecture search. We evaluate BOMP-NAS using the same HAR datasets, memory constraints, and candidate-evaluation protocol as QuaNAG. For BOMP-NAS, the reported runtime corresponds to its online architecture search, whereas for QuaNAG, it corresponds to online architecture generation.

As shown in Table 9, QuaNAG achieves higher accuracy than BOMP-NAS in all six settings, with improvements ranging from 2.45 to 4.88 percent-

age points and an average gain of 3.64 points. QuaNAG requires only 5.50–10.21 minutes for architecture generation, whereas BOMP-NAS requires 4.09–6.52 hours of search, corresponding to a 34.3–44.6 $\times$ speedup. These results indicate that the advantage of QuaNAG is not limited to generation efficiency: transferring quantization-aware search knowledge into the LLM also yields architectures with consistently higher final accuracy.

A.8 Generalization beyond HAR

To examine whether the proposed framework can extend beyond HAR, we conduct a preliminary experiment on CIFAR-10 (Krizhevsky, 2009), an image classification benchmark with a different modality and 2D architecture search space. We evaluate all methods under 1 MB and 3 MB memory constraints and report the resulting accuracy and architecture search or generation time. For CIFAR-10, QuaNAG is separately adapted to the corresponding 2D architecture search space, allowing us to evaluate whether the framework remains effective beyond HAR.

Table 10 shows that QuaNAG achieves the best accuracy under both memory constraints. Under 1 MB, QuaNAG reaches 85.54% accuracy, outperforming the strongest baseline, RZ-NAS (quant), by 3.22 percentage points. Under 3 MB, it achieves 89.98%, exceeding RZ-NAS (quant) by 2.41 points. QuaNAG requires only 5.61–5.83 minutes for generation, remaining substantially faster than the prompt-engineering baselines and slightly faster

Benchmark	Constraint	QuaNAG		BOMP-NAS		Δ Acc.	Speedup
		Accuracy (%)	Time	Accuracy (%)	Time		
Mhealth	15 MB	99.59	6.02 min	96.78	4.32 h	+2.81	43.1×
MMAct	3 MB	65.33	10.21 min	62.23	6.52 h	+3.10	38.3×
MMAct	5 MB	66.50	9.10 min	64.05	6.20 h	+2.45	40.9×
Wharf	3 MB	69.85	7.04 min	65.82	4.64 h	+4.03	39.5×
Wharf	5 MB	72.73	7.21 min	68.18	4.12 h	+4.55	34.3×
Wharf	8 MB	74.24	5.50 min	69.36	4.09 h	+4.88	44.6×
Average	–	74.71	7.51 min	71.07	4.98 h	+3.64	39.8×

Table 9: Comparison with BOMP-NAS, a conventional quantization-aware NAS method, under matched HAR settings. Accuracy reports the best feasible architecture under each memory constraint. QuaNAG time measures online architecture generation, whereas BOMP-NAS time measures online architecture search. Speedup is calculated after converting both times to minutes.

Method	1 MB		3 MB	
	Acc. (%)	Time	Acc. (%)	Time
QuaNAG	85.54	5.61 m	89.98	5.83 m
RZ-NAS	82.13	6.15 h	86.72	5.89 h
RZ-NAS (quant)	82.32	7.08 h	87.57	6.20 h
LM-Searcher	80.91	6.25 m	84.55	6.88 m
LM-Searcher (quant)	79.93	6.79 m	84.92	6.02 m

Table 10: Cross-domain evaluation on CIFAR-10. Time denotes iterative search for prompt-engineering methods and post-adaptation generation for QuaNAG and LM-Searcher; *m* and *h* denote minutes and hours.

than the fine-tuning baselines. These results provide initial evidence that the proposed search strategy and curriculum can be instantiated in a different task domain and architecture search space. Broader evaluations on large-scale vision, language, and multimodal tasks remain future work.

B Theoretical Analysis

Quantization-Branched MCTS algorithm integrates quantization into the algorithm by transforming quantization into a macro branch variable, which enhances the algorithm’s exploration capability. We provide a theoretical justification for why Quantization-Branched MCTS improves exploration compared to standard tree search methods. Our analysis adapts the Markov chain framework from optimization theory, demonstrating how diverse quantization macro branch enhance the conductance of the search process.

B.1 Markov Chain Formulation

We model the search process as a Markov chain over the space of model configurations. As mentioned above, $\mathcal{S}$ is the search space, each node has a model configuration θ . In the proof section, we

use state s to represent the configuration of the node. Each state $s \in \mathcal{S}$ represents a complete neural architecture configuration. The search strategy defines a transition kernel $P(s, s')$ representing the probability of generating configuration s' from current configuration s .

B.1.1 Standard Tree Search Transition Kernel

In standard tree search methods (including vanilla MCTS), the transition from state s to s' is realized by an LLM π conditioned on the current configuration s and feedback $f = F(s)$ from validation tests:

$$P_{\text{standard}}(s, s') = \pi_c(s' | s, F(s)), \quad (6)$$

where π_c denotes the LLM prompted to generate code configurations directly. As observed empirically, π_c tends to be highly concentrated, generating similar configurations even with temperature sampling.

B.1.2 Quantization-Branched MCTS Transition Kernel

With quantization as the branch direction, we introduce an intermediate step where the LLM first generates a quantization direction $\mathcal{B}_q \in \mathcal{Q}$ based on the current state and feedback. The transition kernel becomes:

$$P_{\text{QB-MCTS}}(s, s') = \sum_{\mathcal{B}_q \in \mathcal{Q}} \pi_d(\mathcal{B}_q | s, F(s)) \cdot \pi_c(s' | s, F(s), \mathcal{B}_q), \quad (7)$$

where:

- $\pi_d(\mathcal{B}_q | s, F(s))$ is the distribution over quantization directions (macro branches), which can be highly diverse

Scenario	Benchmark	Constraints	Method	End-to-End Pipeline				Model			Inference Latency	Accuracy	Acc Gap
				Request	Generation	Transfer	Total	Size	Mode	Server			
Exercise Monitoring	MMACT	8MB 500ms	QuaNAG	28ms	7.9min	0.13s	8.0min	4.04MB	none	66.77%	474ms	66.77%	0.00%
			LM-Searcher	28ms	18.1min	0.87s	18.2min	2.57MB	static	60.52%	<u>845ms</u>	60.58%	+0.06%
	USCHAD	7MB 500ms	QuaNAG	27ms	6.4min	0.08s	6.5min	4.23MB	none	75.57%	446ms	75.57%	0.00%
			LM-Searcher	27ms	17.7min	0.40s	18.1min	6.01MB	static	56.33%	<u>735ms</u>	56.51%	+0.18%
	USCHAD	8MB 500ms	QuaNAG	29ms	5.8min	0.03s	5.9min	4.78MB	none	76.16%	181ms	76.16%	0.00%
Industrial Operation	UTD-MHAD	8MB 500ms	QuaNAG	28ms	5.7min	0.10s	5.8min	4.67MB	none	72.56%	263ms	72.56%	0.00%
			LM-Searcher	28ms	12.1min	0.47s	12.7min	1.86MB	static	46.98%	450ms	45.58%	-1.40%
	Wharf	10MB 150ms	QuaNAG	29ms	5.1min	0.32s	5.4min	7.71MB	none	76.36%	4.92ms	76.36%	0.00%
			LM-Searcher	29ms	15.3min	0.43s	15.8min	3.23MB	static	68.64%	75ms	69.55%	+0.91%
	Mhealth	12MB 50ms	QuaNAG	27ms	7.1min	0.10s	7.2min	8.23MB	static	98.78%	23ms	99.39%	+0.61%
Health Tracking	Mhealth	15MB 50ms	QuaNAG	27ms	11.0min	0.22s	11.2min	11.24MB	static	96.34%	<u>66ms</u>	96.34%	0.00%
			LM-Searcher	29ms	6.5min	0.73s	6.6min	13.38MB	static	100.00%	26ms	100.00%	0.00%
	Mhealth	20MB 50ms	QuaNAG	29ms	9.2min	0.24s	9.5min	11.25MB	static	98.17%	11ms	98.17%	0.00%
Health Tracking	Mhealth	15MB 50ms	QuaNAG	29ms	5.6min	0.55s	5.7min	13.56MB	static	100.00%	31ms	100.00%	0.00%
			LM-Searcher	29ms	8.2min	0.25s	8.5min	11.74MB	static	98.19%	10ms	98.19%	0.00%
	Mhealth	20MB 50ms	QuaNAG	29ms	5.6min	0.55s	5.7min	13.56MB	static	100.00%	31ms	100.00%	0.00%

Note: *Constraints* denotes memory constraint and latency requirement. *Request* is network latency from device to server. *Generation* is server-side model generation time. *Transfer* is model file transfer time. *Total* is end-to-end deployment time. *Size* is actual model size. *Mode* indicates the quantization mode. *Server* is accuracy in server-side. *Inference Latency* is device forward pass time. *Accuracy* is the accuracy in device. *Acc Gap* is device-server accuracy difference caused by different quantization engines on ARM and AMD architectures. **Bold** indicates better performance. Underline indicates the violation of latency.

Table 11: Edge Deployment Performance Comparison on Jetson Device

- $\pi_c(s'|s, F(s), \mathcal{B}_q)$ is the LLM generating a new configuration conditioned on both the parent configuration and the selected quantization direction
- $\mathcal{Q} = \{\text{none, static, qat, qaft}\}$ is the set of quantization modes

Key distinction in branch direction: While some other MCTS (Light et al., 2025) uses coding style directions where each direction leads to a single child, our macro branches can connect to multiple child nodes. When $|\mathcal{C}_{s, \mathcal{B}_q}| > 1$, an additional node selection step (Eq. 30) determines which specific child $s' \in \mathcal{C}_{s, \mathcal{B}_q}$ to select via UCT.

B.2 Conductance Analysis

Following Markov chain mixing theory (Levin and Peres, 2020), we analyze how the transition kernel affects the **conductance** Φ of the search process, which measures how quickly the chain can escape local regions.

For a subset $S \subseteq \mathcal{S}$ of the state space, the conductance is defined as:

$$\Phi_P(S) = \frac{\sum_{s \in S, s' \notin S} \mu(s) P(s, s')}{\mu(S)}, \quad (8)$$

where μ is the stationary distribution and $\mu(S) = \sum_{s \in S} \mu(s)$.

B.2.1 Low Conductance of Standard Methods

Our algorithm uses branches as directions, while for standard tree search without direction, if the

search gets trapped in a local region S (e.g., configurations with similar quantization properties), we have:

$$P_{\text{standard}}(s, s') \approx 0 \quad \text{when } s \in S \text{ and } s' \notin S. \quad (9)$$

This occurs because $\pi_c(s'|s, F(s))$ concentrates on configurations similar to s , making it unlikely to generate s' outside the local region. Consequently:

$$\Phi_{P_{\text{standard}}}(S) \approx 0, \quad (10)$$

resulting in poor mixing and high probability of getting stuck in local optima.

B.2.2 Enhanced Conductance with Direction

With Quantization-Branched MCTS, even if π_c is concentrated, the diverse quantization directions in π_d provide pathways out of local regions:

$$\begin{aligned} P_{\text{QB-MCTS}}(s, s') &= \sum_{\mathcal{B}_q \in \mathcal{Q}} \pi_d(\mathcal{B}_q | s, F(s)) \\ &\quad \cdot \pi_c(s' | s, F(s), \mathcal{B}_q) \quad (11) \\ &> 0, \\ &\text{if some } \mathcal{B}_q \text{ are feasible.} \end{aligned}$$

Key insight: If the current configuration s uses one quantization mode (e.g., static PTQ), selecting a different macro branch $\mathcal{B}_q$ (e.g., QAFT) can lead to configurations s' in different regions of the search space, even if π_c individually is concentrated. This is because:

1. Different quantization modes fundamentally alter the model’s memory footprint and accuracy characteristics

2. The conditioning on $\mathcal{B}_q$ in $\pi_c(s'|s, F(s), \mathcal{B}_q)$ guides the LLM to generate configurations compatible with that quantization mode
3. Multiple children per branch ($|\mathcal{C}_{s, \mathcal{B}_q}| > 1$) further diversify exploration within each quantization mode

Therefore:

$$\Phi_{P_{\text{QB-MCTS}}}(S) > \Phi_{P_{\text{standard}}}(S). \quad (12)$$

B.3 Mixing Time Implications

By Cheeger’s inequality (Levin and Peres, 2020), higher conductance Φ improves the spectral gap γ of the Markov chain:

$$\gamma \geq \frac{\Phi^2}{2}. \quad (13)$$

The mixing time t_{mix} , which quantifies how quickly the chain approaches its stationary distribution, satisfies:

$$t_{\text{mix}} \leq \frac{\log(\mu_{\min}^{-1})}{\gamma}, \quad (14)$$

where $\mu_{\min} = \min_{s \in \mathcal{S}} \mu(s)$.

Conclusion: Since $\Phi_{P_{\text{QB-MCTS}}} > \Phi_{P_{\text{standard}}}$, we have:

$$t_{\text{mix}}^{\text{QB-MCTS}} < t_{\text{mix}}^{\text{standard}}. \quad (15)$$

This means Quantization-Branched MCTS mixes faster, reducing the likelihood of getting trapped in local regions and enabling more efficient exploration of the configuration space.

C Conservative Estimate of Quantization-Expanded Search Space

This section provides a conservative estimate of how quantization expands the feasible search space under memory constraints. Our search space is defined as follows:

- Expansion ratios: $\{1, 2, 3, 4\}$.
- Channels: $\{6, 12, 18, 24\}$.
- SE module choices: $\{\text{True}, \text{False}\}$.
- SE ratios: $\{0, 0.25, 0.5\}$.
- Quantization modes: $\{\text{none}, \text{static}, \text{qat}, \text{qatf}\}$.

Since each architecture contains at most four stages and each stage contains at most two blocks, the maximum number of blocks in one architecture is:

$$D_{\text{max}} = 4 \times 2 = 8. \quad (16)$$

Under a memory budget of $M_{\text{max}} = 5\text{MB}$, INT8-style quantization can approximately reduce model memory to one quarter of its FP32 memory. Thus, a quantized architecture satisfying the 5MB deployment budget may correspond to an FP32 architecture whose memory is up to approximately:

$$4M_{\text{max}} = 20\text{MB}. \quad (17)$$

This means that quantization does not simply add several categorical choices. Instead, it expands the feasible FP32-equivalent architecture region from M_{max} to approximately $4M_{\text{max}}$.

To provide empirical evidence for this estimate, we further randomly sample a small set of evaluated architectures from the USCHAD search records and examine whether the 20MB FP32-equivalent region exposes structurally larger models. Among the 24 sampled evaluated architectures, only 2 architectures fall within the $\leq 5\text{MB}$ region, with 1-2 blocks. In contrast, 18 architectures fall within the 5-20MB region, with an average of 4.44 blocks and up to 7 blocks. Considering all architectures within the $\leq 20\text{MB}$ region, the average number of blocks is 4.15 and the maximum number of blocks is 7. This empirical observation shows that the $4 \times$ FP32-equivalent memory region does expose substantially deeper architecture regions in practice.

However, to avoid overestimating the search-space expansion, we do not directly use the largest block count observed in the sampled records. Although the sampled USCHAD records show that the 5-20MB FP32-equivalent region contains architectures with up to 7 blocks, while the $\leq 5\text{MB}$ region contains only 1-2 blocks, we conservatively assume that the quantization-expanded region exposes only two additional blocks. This assumption is substantially smaller than the empirical difference observed in the sampled records.

- Number of stages: $\{1, 2, 3, 4\}$.
- Blocks per stage: $\{1, 2\}$.
- Convolution types: $\{\text{DWSepConv}, \text{MBConv}, \text{DpConv}, \text{SeSepConv}, \text{SeDpConv}\}$.
- Kernel sizes: $\{3, 5, 7\}$.
- Strides: $\{1, 2, 4\}$.
- Skip connections: $\{\text{True}, \text{False}\}$.
- Activations: $\{\text{ReLU6}, \text{LeakyReLU}, \text{Swish}\}$.

For a conservative lower-bound estimate, we only count three basic block-level choices: convolution type, kernel size, and activation function. This gives:

$$B = 5 \times 3 \times 3 = 45 \quad (18)$$

possible choices per block. We intentionally ignore other dimensions, including stride, skip connection, expansion ratio, channel width, SE module, and SE ratio.

Let D denote the maximum number of blocks in the original 5MB FP32-feasible region. Under the conservative assumption that quantization exposes only two additional blocks, the quantization-expanded region contains architectures with up to $D + 2$ blocks. The number of unique block-level architecture patterns without quantization is approximated by:

$$S_{\text{none}} = \sum_{d=1}^D B^d. \quad (19)$$

The number of unique block-level architecture patterns in the quantization-expanded region is approximated by:

$$S_{\text{quant}} = \sum_{d=1}^{D+2} B^d. \quad (20)$$

Therefore, the unique architecture-level expansion is:

$$\begin{aligned} G_{\text{unique}} &= \frac{S_{\text{quant}}}{S_{\text{none}}} \\ &= \frac{\sum_{d=1}^{D+2} 45^d}{\sum_{d=1}^D 45^d} \\ &\approx 45^2 \\ &= 2025. \end{aligned} \quad (21)$$

When the three quantized modes {static, qat, qaft} are counted as distinct configurations, the corresponding configuration-level expansion over the original non-quantized feasible space is:

$$\begin{aligned} G_{\text{config}} &= \frac{S_{\text{none}} + 3S_{\text{quant}}}{S_{\text{none}}} \\ &= 1 + 3G_{\text{unique}} \\ &\approx 1 + 3 \times 2025 \\ &= 6076. \end{aligned} \quad (22)$$

This estimate is deliberately conservative for three reasons. First, it assumes that the $4 \times$ FP32-equivalent memory expansion exposes only two

additional blocks, although the sampled USCHAD records show that the 5–20MB region already contains architectures with up to 7 blocks. Second, it only counts convolution type, kernel size, and activation function. Third, it ignores stride, skip connection, expansion ratio, channel width, SE module, and SE ratio. Even under this lower-bound estimate, quantization can expand the feasible configuration space by approximately 6.1×10^3 times. This supports our key insight that quantization should not be treated as an ordinary architecture attribute. Instead, quantization reshapes the feasible architecture region under memory constraints and should be modeled as a branch-level search decision.

D Detailed Search Dynamics Analysis

We compare Quantization-Branched MCTS (QB-MCTS) with Vanilla MCTS under strictly controlled settings. Both methods use the same search space, backbone LLM, candidate-evaluation pipeline, initialization procedure, and search budget. The only difference is that QB-MCTS represents quantization modes as explicit macro branches, whereas Vanilla MCTS handles quantization within ordinary candidate generation and tree expansion.

We evaluate exploration behavior using three metrics. Let p_q denote the proportion of visits assigned to quantization mode $q \in \mathcal{Q}$. Normalized quantization-visitation entropy is defined as

$$H_{\text{quant}} = -\frac{1}{\log |\mathcal{Q}|} \sum_{q \in \mathcal{Q}} p_q \log p_q, \quad (23)$$

where a larger value indicates more balanced exploration across quantization modes. Architectural diversity D_{arch} is the average normalized structural distance among evaluated architectures after excluding the quantization mode. The cross-quantization transition rate is

$$P_{\text{cross}} = \frac{\sum_{t=1}^T \mathbb{I}[q(n_t) \neq q(n_{t+1})]}{T}, \quad (24)$$

where T is the number of observed parent–child transitions. A larger P_{cross} indicates greater mobility across quantization regions.

We further report the area under the best-so-far feasible-accuracy curve (AUC), the best feasible accuracy R^* , and N_{95} , the number of evaluations required to reach 95% of the common reference

Benchmark	Budget	Method	Exploration Behavior			Search Outcome		
			$H_{\text{quant}} \uparrow$	$P_{\text{cross}} \uparrow$	$D_{\text{arch}} \uparrow$	AUC $\uparrow$	$R^* (\%) \uparrow$	$N_{95} \downarrow$
Mhealth	15 MB	QB-MCTS	0.990	0.650	0.451	96.92	99.69	14
		Vanilla MCTS	0.078	0.020	0.418	96.04	97.56	30
MMAct	3 MB	QB-MCTS	0.983	0.711	0.381	64.41	66.24	14
		Vanilla MCTS	0.730	0.130	0.303	64.36	63.44	49
MMAct	5 MB	QB-MCTS	0.941	0.720	0.349	65.26	67.20	32
		Vanilla MCTS	0.497	0.030	0.304	61.91	62.86	58
Wharf	3 MB	QB-MCTS	0.939	0.820	0.454	71.35	73.18	25
		Vanilla MCTS	0.397	0.170	0.404	70.17	70.91	38
Wharf	5 MB	QB-MCTS	0.967	0.710	0.392	71.36	74.55	17
		Vanilla MCTS	0.769	0.250	0.312	70.21	70.36	33
Average	–	QB-MCTS	0.964	0.722	0.405	73.86	76.17	20.4
		Vanilla MCTS	0.494	0.120	0.348	72.54	73.03	41.6

Table 12: Complete search-dynamics comparison between QB-MCTS and Vanilla MCTS. Both methods use identical search spaces, LLMs, candidate-evaluation pipelines, and search budgets. Higher values are better except for N_{95} .

best accuracy. Lower N_{95} indicates faster discovery of high-quality feasible architectures.

Across all five settings, QB-MCTS consistently improves quantization coverage, architectural diversity, and cross-quantization mobility. On average, H_{quant} increases from 0.494 to 0.964, P_{cross} from 0.120 to 0.722, and D_{arch} from 0.348 to 0.405. These improvements translate into better search outcomes: QB-MCTS increases average AUC from 72.54 to 73.86 and average best feasible accuracy from 73.03% to 76.17%, while reducing N_{95} from 41.6 to 20.4 evaluations. This approximately 51% reduction indicates substantially faster discovery of high-quality feasible architectures.

E Memory Calibration

Our memory constraint model distinguishes platform-independent tensor memory from device-specific peak memory. For an architecture θ , $M(\theta)$ is estimated from the shapes and numerical precision of its parameter tensors and intermediate activation tensors. Therefore, when the architecture, input size, and numerical precision are fixed, $M(\theta)$ does not depend on a particular hardware platform. In contrast, the peak memory observed during deployment may additionally be affected by the target hardware, runtime implementation, and memory-management overhead.

To account for these platform-specific effects without binding the architecture representation to a particular device, we perform a lightweight calibration before formal search and deployment. For each target platform h , we evaluate a small pilot

Platform	Pearson r	MRE
Raspberry Pi	0.924	6.65%
Jetson Nano	0.945	4.10%

Table 13: Calibration of the tensor-based memory estimate against measured device peak memory. MRE denotes mean relative error after platform-specific calibration.

set of architectures, record both $M(\theta)$ and the corresponding device-measured peak memory, and establish a calibration mapping g_h . Device peak memory is then predicted as

$$\widehat{M}_h(\theta) = g_h(M(\theta)). \quad (25)$$

This design preserves a common tensor-based memory representation across platforms while using g_h to capture systematic hardware- and runtime-specific differences. We evaluate the calibration using the Pearson correlation coefficient between $M(\theta)$ and measured peak memory, and the mean relative error (MRE) between calibrated predictions and device measurements.

As shown in Table 13, the tensor-based estimate is strongly correlated with measured peak memory on both platforms, with Pearson coefficients of 0.924 on Raspberry Pi and 0.945 on Jetson Nano. After calibration, the prediction errors are reduced to 6.65% and 4.10%, respectively. These results indicate that a platform-independent $M(\theta)$ can provide a stable basis for memory-constrained architecture generation, while a small platform-specific calibration set is sufficient to obtain accurate deployment-time memory predictions. Conse-

quently, the same adapted LLM can be applied to different hardware platforms without re-adaptation; only the lightweight calibration mapping needs to be updated.

F Detailed Quantization-Branched MCTS

This section provides the detailed implementation of the proposed quantization-branched MCTS, including the complete problem formulation, exploration procedure, architecture generation and repair mechanisms, quantization-aware evaluation, and detailed backpropagation updates.

F.1 Problem Formulation

Search Space and Model Configuration: We define the neural architecture search space as $\mathcal{S}$, where each model configuration $\theta \in \mathcal{S}$ represents a complete model architecture configuration including the number of stages, blocks per stage, convolution types, channel dimensions, and other architectural hyperparameters. The configuration also includes a quantization mode q selected from the set $\mathcal{Q} = \{\text{none}, \text{static}, \text{qat}, \text{qaft}\}$, which determines the quantization modes applied to the model. The four quantization modes in $\mathcal{Q}$ have specific meanings: none represents no quantization, static represents Static Post-Training Quantization (PTQ) (Jacob et al., 2018), qat represents Quantization-Aware Training (QAT) (Nagel et al., 2021), and qaft represents Quantization-Aware Fine-Tuning (QAFT) (Jeon et al., 2025).

Constraints and Objectives: The search is subject to constraints $\mathcal{C}$ which primarily include a memory constraint $M_{\max}$. Given a dataset $\mathcal{D}$, we aim to find the optimal configuration θ^* that maximizes the reward function $R(\theta, \mathcal{D})$ under $M_{\max}$, where the reward function $R(\theta, \mathcal{D})$ represents model accuracy after training and optional quantization. We formulate our objective as Equation 26.

$$\theta^* = \arg \max_{\theta \in \mathcal{S}} R(\theta, \mathcal{D}) \quad \text{s.t.} \quad M(\theta) \leq M_{\max}, \quad (26)$$

where $M(\theta)$ denotes the peak memory usage of configuration θ .

MCTS Formulation: To solve the optimization problem defined in Equation 1, we employ MCTS. We formulate the architecture search as a multi-root tree search, treating each model architecture as a tree node. Additionally, each node records

the architecture’s memory size, accuracy, and other metadata, enabling data updating for MCTS back-propagation.

Macro Branch: In the tree structure of MCTS, every parent node n_{parent} and its kid node n_{kid} is connected by a directed edge $d_{\langle \text{parent}, \text{kid} \rangle}$ from n_{parent} to n_{kid} . Unlike standard MCTS, we introduce a brand-new concept termed macro branch in quantization-branched MCTS, which is noted as $\mathcal{B}$. We define the four quantization options in $\mathcal{Q}$ as four macro branches, where each node connected to a macro branch adopts the corresponding quantization mode. For example, a node n_i may have three macro branches connecting it to multiple kid nodes, where kids connected via the static macro branch $\mathcal{B}_{\text{static}}$ adopt the static quantization mode. Additionally, a macro branch can connect to multiple child nodes. Thus, it can be visualized as multiple edges pointing in the same direction, like small twigs together forming a large branch.

Based on the formulation above, our quantization-branched MCTS operates in three main phases as illustrated in Figure 3: (1) Exploration phase for traversing the tree, (2) Generation and Evaluation phase for creating and assessing new configurations, and (3) Backpropagation phase for updating statistics. We detail each phase in the following subsections.

F.2 Macro-Branch Exploration

F.2.1 Root Selection

At each iteration, we select the most promising forest root node using the Upper Confidence Bound applied to Trees (UCT) formula:

$$i^* = \arg \max_{i \in F} \left(\mathbb{Q}_i + c \cdot \sqrt{\frac{\ln \sum_{j=1}^{|F|} v_j}{v_i}} \right), \quad (27)$$

where F is the set of all root nodes, $\mathbb{Q}_i$ is the Q-value for root node i , v_i is the visit count of root i , $\sum_{j=1}^{|F|} v_j$ is the total visits across all roots, and c is the exploration weight that balances exploitation versus exploration.

The Q-value $\mathbb{Q}_i$ for each root is computed as:

$$\mathbb{Q}_i = \begin{cases} \max_{q \in \mathcal{Q}} Q_i^{\mathcal{B}_q} & \text{if any } Q_i^{\mathcal{B}_q} > 0 \\ r_i & \text{otherwise} \end{cases}, \quad (28)$$

where $Q_i^{\mathcal{B}_q}$ denotes the Q-value of root i in macro branch $\mathcal{B}_q$ with $q \in \mathcal{Q}$, and r_i is the accuracy of root i itself. The second case handles initialization

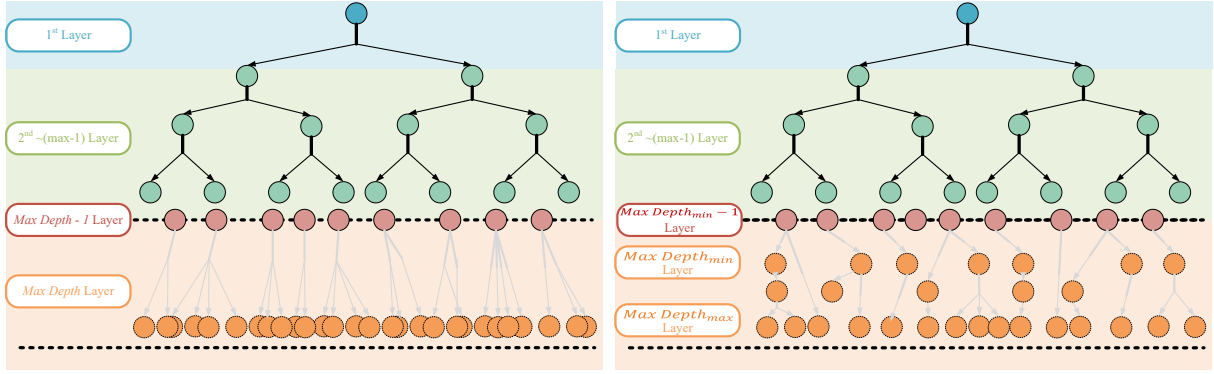


Figure 7: Comparison between fixed and randomized maximum search depth. Fixed depth causes terminal nodes to concentrate at the same layer, while randomized depth distributes terminal nodes across different layers, improving exploration diversity.

where all macro branch Q-values are zero, using the root's own accuracy as the initial Q-value. Note that we use $\mathbb{Q}$ to denote the Q-value of nodes, use Q to represent the macro branch's Q-value, and use $\mathcal{Q}$ to indicate the quantization set. Since the root is also a node, we use $\mathbb{Q}$ to denote the Q-value of the root.

As shown in Figure 3, after calculating the score for each root according to the formula, root 1 has the highest score, so root 1 is selected.

F.2.2 Branch Selection

After selecting a root node, the search traverses the tree by selecting macro branches at each node. At node n , we select macro branch $\mathcal{B}_q$ that maximizes the UCT value:

$$\mathcal{B}_q^* = \arg \max_{q \in \mathcal{Q}} \left(Q_n^{\mathcal{B}_q} + c \cdot \sqrt{\frac{\ln \sum_{q' \in \mathcal{Q}} v_n^{\mathcal{B}_{q'}}}{v_n^{\mathcal{B}_q}}} \right), \quad (29)$$

where $Q_n^{\mathcal{B}_q}$ is the Q-value of node n in macro branch $\mathcal{B}_q$, $v_n^{\mathcal{B}_q}$ is the visit count of node n in macro branch $\mathcal{B}_q$, and c is the exploration weight. The numerator in the square root represents the total visits across all branches at node n .

As illustrated in Figure 3, upon reaching root 1, the $\mathcal{B}_{none}$ is selected based on the Equation 29. Besides, a selected macro branch may contain one or multiple edges. If only a single edge exists, the search proceeds directly along this edge. However, when the search follows the $\mathcal{B}_{none}$ of root 1 and reaches node 2, since the Equation selects $\mathcal{B}_{static}$, and this branch has multiple edges, so it cannot directly traverse along one edge to the next node. When multiple edges are present, we employ a node selection strategy to determine which edge to traverse, as detailed in Appendix F.2.3.

F.2.3 Node Selection

When a selected macro branch $\mathcal{B}_q$ contains multiple edges connected to nodes, we choose the child node that maximizes the UCT value and the exploration traverses along the edge connected to the child node:

$$n_c^* = \arg \max_{n_{child} \in \mathcal{C}_n, \mathcal{B}_q} \left(\mathbb{Q}_{n_{child}} + c \cdot \sqrt{\frac{\ln \sum_{n_{child} \in \mathcal{C}_n, \mathcal{B}_q} v_{n_{child}}}{v_{n_{child}}}} \right), \quad (30)$$

where $\mathcal{C}_n$ denotes the children set of node n , and $\mathcal{C}_n, \mathcal{B}_q$ denotes the set of child nodes of n in branch $\mathcal{B}_q$, $\mathbb{Q}_{n_{child}}$ is the Q-value of child node n_{child} , $v_{n_{child}}$ is its visit count, and c is the exploration weight. The numerator in the square root sums visits across all children in the branch.

We record each step as a triplet $(n_t, \mathcal{B}_{q_t}, n_{t+1})$, where n_t is the current node, $\mathcal{B}_{q_t}$ is the selected branch, and $n_{t+1} \in \mathcal{C}_{n_t, \mathcal{B}_{q_t}}$ is the selected child. The complete trajectory τ from root to the terminal node is below:

$$\tau = [(n_0, \mathcal{B}_{q_0}, n_1), (n_1, \mathcal{B}_{q_1}, n_2), \dots, (n_{T-2}, \mathcal{B}_{q_{T-2}}, n_{T-1}), (n_{T-1}, \mathcal{B}_{q_{T-1}}, n_T)]. \quad (31)$$

The search process terminates under two conditions: encountering a leaf node or reaching the maximum depth. The maximum depth variable is set to a random number within a specified range, ensuring that terminal nodes are not predominantly concentrated at any single layer. Instead of using a fixed maximum depth, we randomly sample the maximum search depth d_{max} within a predefined range for each iteration. As illustrated in Figure 7, using a fixed depth causes terminal nodes to concentrate

on a single layer, making the exploration overly biased toward architectures with similar depths. In contrast, randomizing d_{max} distributes terminal nodes across different layers, enabling the search process to explore both shallow and deep architectures more uniformly. This depth randomization improves search diversity and reduces structural bias during trajectory sampling.

In the Figure 3, algorithm chooses the node i , and the terminal node is n_T which is a leaf node. n_T serves as the parent for generating new nodes. In addition, search computes the macro branch of the new node by Equation 29. As shown in Figure 3, macro branch of the new node is $\mathcal{B}_{QAFT}$, so the new node’s quantization mode must be qaft.

Subsequently, the terminal node n_T ’s configuration, along with a designed prompt, is provided to the LLM to generate candidate child nodes, as detailed in Section F.3.1.

In summary, the complete search process operates as follows: Starting from a root selected by Equation 27, we iteratively apply Equation 29 to select a macro branch at each node. If the branch contains a single edge, we traverse it directly; if multiple edges exist, we apply Equation 30 to select the optimal child node. This process continues until reaching either a leaf node or the maximum depth d_{max} .

F.3 Generation & Evaluation

F.3.1 Generation

Our prompt consists of a system prompt and a human prompt.

The system prompt provides global knowledge for architecture design, including definitions of five convolution types, dataset-specific memory consumption rankings from high to low (e.g., MBConv consumes the most memory while SeDpConv consumes the least), quantization mode descriptions, and global experience insights extracted from successful search branches.

The human prompt provides the current search context, including the parent node’s complete configuration and performance metrics, target quantization direction, memory constraints, and input-output specifications (HAR tasks’ time steps and channels), and guides the LLM to generate model configurations that comply with search space constraints through examples.

Additionally, we have three detection and repair mechanisms for model architectures. The configu-

ration validation mechanism automatically corrects constraint violations in LLM-generated configurations, and records all repair operations in the dialogue history. The memory auto-reduction mechanism, when configurations exceed memory limits, sequentially applies four-level strategies including intelligent channel reduction (selecting the closest valid value from the search_space), removing redundant blocks, deleting stages, and compressing expansion, ensuring the final configuration satisfies constraints. The fallback generation mechanism, when LLM calls fail or configurations cannot be repaired, activates a base configuration generator to randomly construct simple architectures that comply with constraints, guaranteeing the robustness of the search process.

F.3.2 Evaluation

After the LLM generates a model configuration that satisfies the constraints, this configuration is converted into an actual model, which is then subjected to corresponding training and quantization.

Each model architecture has a quantization mode $q \in \mathcal{Q}$. These four quantization modes are highly typical and representative. As shown in Table 14, after the models are generated, we train all of them for the same given number of epochs, and then move the models to the CPU to perform the corresponding quantization processing.

Among them, QAFT is the most special because it requires training twice. Therefore, to ensure fairness, it trains for $0.9 \times \text{given epochs}$ in the first stage and $0.1 \times \text{given epochs}$ in the second quantization training stage.

After the model evaluation is completed, backpropagation begins, as detailed in Section F.4.

F.4 Backpropagation

Since this new node is not None, its accuracy is based on the quantized accuracy. However, if this new node is None, the result is based directly on normal training. The resulting accuracy of model configuration θ is denoted as $R(\theta, \mathcal{D})$, where θ is the previously defined model architecture and $\mathcal{D}$ is the previously defined dataset, representing the accuracy of this model architecture on this dataset.

Upon obtaining $R(\theta, \mathcal{D})$ for the new node n_T , we backpropagate this reward through trajectory τ in reverse order, updating both Q-values, visit counts and other statistics.

Mode	Training & Quantization Process	Mode	Training & Quantization Process
1. none	Normal training for given epochs → No quantization → Output FP32 model	2. static	Normal training for given epochs → Post-training static INT8 quantization (try 6 configs, select best) → Output INT8 model
3. qat	Insert FakeQuantize → Quantization-aware training for given epochs → Convert to INT8 model	4. qaft	Pre-training for $0.9 \times$ given epochs → Insert FakeQuantize → Fine-tuning for $0.1 \times$ given epochs → Convert to INT8 model

Table 14: Four quantization modes and their training processes

F.4.1 Q-Value of Branch Update

We first initialize the Q-value for the newly generated node:

$$\mathbb{Q}_{n_T} = R(\theta_T, \mathcal{D}). \quad (32)$$

Then, we update the branch Q-value at the parent node n_{T-1} using an adaptive learning rate:

$$\begin{aligned} Q_{n_{T-1}}^{\mathcal{B}_{q_{T-1}}} &\leftarrow (1 - \alpha_{T-1}) Q_{n_{T-1}}^{\mathcal{B}_{q_{T-1}}} \\ &+ \alpha_{T-1} \cdot \max \left(Q_{n_{T-1}}^{\mathcal{B}_{q_{T-1}}}, R(\theta_T, \mathcal{D}) \right). \end{aligned} \quad (33)$$

where $Q_{n_{T-1}}^{\mathcal{B}_{q_{T-1}}}$ is the Q-value of the node n_{T-1} in the branch $\mathcal{B}_{q_{T-1}}$, α_{T-1} is the adaptive learning rate that decreases with visit count. And the Equation of the the adaptive learning rate is as follows:

$$\alpha_{T-1} = \frac{1}{v_{n_{T-1}}^{\mathcal{B}_{q_{T-1}}} + 1}. \quad (34)$$

For the remaining nodes along trajectory $\tau = [(n_0, \mathcal{B}_{q_0}, n_1), \dots, (n_{T-1}, \mathcal{B}_{q_{T-1}}, n_T)]$, we propagate Q-values backward from $t = T - 2$ to $t = 0$.

$$\begin{aligned} Q_{n_t}^{\mathcal{B}_{q_t}} &\leftarrow (1 - \alpha_t) Q_{n_t}^{\mathcal{B}_{q_t}} + \alpha_t \cdot \max \left(Q_{n_t}^{\mathcal{B}_{q_t}}, \right. \\ &\quad \left. \max_{q'_{t+1} \in \mathcal{Q}} Q_{n_{t+1}}^{\mathcal{B}_{q'_{t+1}}} \right). \end{aligned} \quad (35)$$

$Q_{n_t}^{\mathcal{B}_{q_t}}$ is the Q-value of node n_t in the macro branch $\mathcal{B}_{q_t}$. And $\max_{q'_{t+1} \in \mathcal{Q}} Q_{n_{t+1}}^{\mathcal{B}_{q'_{t+1}}}$ means choosing the maximum value among all the branch's Q-values of the child node n_{t+1} . This Bellman-style update takes the maximum between the current estimate and the best Q-value among all branches of the child node n_{t+1} . Simultaneously, we increment the visit count for each node-branch pair:

$$v_{n_t}^{\mathcal{B}_{q_t}} \leftarrow v_{n_t}^{\mathcal{B}_{q_t}} + 1. \quad (36)$$

F.4.2 Node Level Update

In addition to updating branch-specific Q-values, we maintain node-level reward statistics for guiding future exploration. The reward propagated to each node is computed recursively using a discount factor. For the terminal node n_T , we initialize:

$$r_T = R(\theta_T, \mathcal{D}). \quad (37)$$

For earlier nodes in the trajectory (from $t = T - 1$ to $t = 0$), the discounted reward is:

$$r_t = \bar{R}(n_{t+1}) \times \gamma, \quad (38)$$

where γ is the discount factor and $\bar{R}(n_{t+1}) = \frac{R_{\text{total}}(n_{t+1})}{v_{n_{t+1}}}$ is the average reward of the next node.

Using these discounted rewards, we update the node statistics:

$$v_{n_t} \leftarrow v_{n_t} + 1. \quad (39)$$

$$R_{\text{total}}(n_t) \leftarrow R_{\text{total}}(n_t) + r_t. \quad (40)$$

This discounting mechanism ensures that nodes closer to the terminal reward receive stronger updates while still allowing information to flow backward through the entire trajectory. The node-level average reward $\bar{R}(n_t)$ serves as a fallback Q-value when all branch Q-values are zero (as defined in Equation 28).

Additionally, this part also includes the update of the node's Q-value, which is updated similarly to the root node's update. The Q-value of each node is defined below:

$$\mathbb{Q}_{n_t} = \begin{cases} \bar{R}(n_t) & \text{if } \max_{q'_t \in \mathcal{Q}} Q_{n_t}^{\mathcal{B}_{q'_t}} = 0 \\ \max_{q'_t \in \mathcal{Q}} Q_{n_t}^{\mathcal{B}_{q'_t}} & \text{otherwise} \end{cases}, \quad (41)$$

where $\max_{q'_t \in \mathcal{Q}} Q_{n_t}^{\mathcal{B}_{q'_t}}$ is the maximum value of all the branch Q-values in node n_t , $\bar{R}(n_t)$ is the average reward of node n_t , and the visit count of node n_t is v_{n_t} . Equation 41 means: if all branch Q-values of the node n_t are 0, we use the node's average reward; otherwise, we use the maximum of all branch Q-values.

According to these equations, node n_T currently has no macro branches, so its $\max_{q'_t \in \mathcal{Q}} Q_{n_t}^{\mathcal{B}_{q'_t}} = 0$. Therefore, it uses the average reward $\bar{R}(n_T)$ as the node Q-value. And since it has no child nodes, its average reward is simply its own accuracy. Therefore, we can finally deduce that $\mathbb{Q}_{n_T} = R(\theta_T, \mathcal{D})$, which is Equation 32.

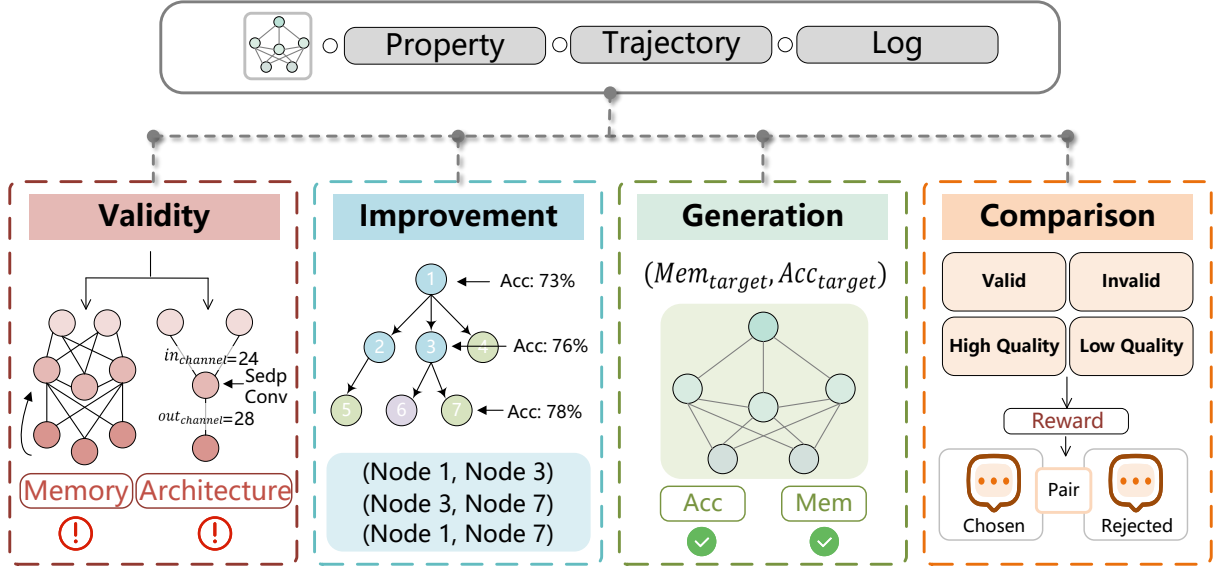


Figure 8: Illustration of four training datasets. The validity dataset focuses on detecting and correcting violations of memory rules and architecture rules. The improvement dataset focuses on enhancing model performance. The generation dataset focuses on generating complete models that satisfy the requirements. The comparison dataset focuses on constructing comparative pairs.

G Detailed Constraint-to-Quality Curriculum

This section provides detailed descriptions of the progressive dataset construction, training procedures, and implementation details for constraint-to-quality curriculum.

The effectiveness of neural architecture generation critically depends on how well the LLM internalizes the complex mappings between architectural choices, memory constraints, and performance outcomes. To enable the LLM to generate memory-compliant architectures with higher accuracy without time-intensive search, we construct a progressive training dataset from data collected during quantization-branched MCTS. And then these datasets are organized into two fine-tuning phases: progressive Supervised Fine-Tuning (SFT) pipeline and reward preference optimization. We detail the dataset construction in Section G.1 and the dual-phase training strategy in Section G.2.

G.1 Dataset Construction

We use the model configurations generated by quantization-branched MCTS, along with their associated model properties, search trajectories, and logs, as inputs to construct four complementary datasets. These four datasets are the validity, improvement, generation, and comparison datasets, which progressively decompose the memory-bounded NAG problem from different an-

gles to facilitate LLM learning.

G.1.1 Validity Dataset

The validity dataset trains the LLM to identify and correct architecture violations, which is fundamental for ensuring generated architectures comply with wearable devices’ memory constraints and architecture rules. Therefore, the validity dataset comprises two types of tasks: **validation** and **repair**. The validation task focuses on determining whether a model architecture violates constraints, while the repair task addresses how to correct invalid architectures.

The input of validation tasks is a complete model configuration, and the output is a structured judgment containing a validity flag and violation details. As shown in the validity section of Figure 8, validation tasks mainly include two types of violations: architecture violations and memory violations. Correspondingly, repair tasks take a violating configuration along with its structured violation information as input, and output the corrected configuration. Repair tasks are responsible for fixing the architecture and memory violations identified by validation tasks. Through this complementary design, validation tasks teach the LLM to identify errors, while Repair tasks teach it to correct them. We list the architecture rules and memory rules that model architectures must follow in Table 15.

Architecture rule include six types of breaches:

Algorithm 1: Quantization-Branched MCTS

Input: Search space $\mathcal{S}$, dataset $\mathcal{D}$, memory limit $M_{\max}$, iterations N

Output: Best configuration θ^*

```
1 Initialize forest  $F$  with  $K$  seed configurations;
2 for  $iter = 1$  to  $N$  do
3    $i^* \leftarrow \arg \max_{i \in F} UCT_{root}(i)$ ;
   // Eq. 27
4    $n \leftarrow root_{i^*}$ ,  $\tau \leftarrow []$ ;
   while  $n$  is not leaf and  $depth < d_{\max}$  do
5      $d \leftarrow random[d_{\min}, d_{\max}]$ ;
6      $\mathcal{B}_q^* \leftarrow \arg \max_q UCT_{branch}(n, \mathcal{B}_q)$ ;
       // Eq. 29
7     ;
8      $n_{child} \leftarrow \text{SelectChild}(n, \mathcal{B}_q^*)$ ;
       // Eq. 30
9      $\tau.append(n, \mathcal{B}_q^*, n_{child})$ ;
10     $n \leftarrow n_{child}$ ;
11   $\theta_{new} \leftarrow \text{LLM.generate}(n, \mathcal{B}_q^*, M_{\max})$ ;
   // Generation
12   $R \leftarrow \text{TrainAndQuantize}(\theta_{new}, \mathcal{D}, \mathcal{B}_q^*)$ ;
   // Evaluation
13   $\text{Backpropagate}(R, \tau, Q, \mathbb{Q}, v)$ ;
   // Update  $Q$ -values (Eq. 33-40)
14 return  $\theta^*$  with highest reward;
```

(1) unsupported skip connection in certain block types (e.g., DpConv and SeDpConv do not support skip connections); (2) DWSepConv expansion equivalence (DWSepConv with $expansion > 1$ is semantically equivalent to MBConv); (3) SeDpConv channel mismatch (SeDpConv requires $in_channels == out_channels$); (4) SE module inconsistency (has_se and se_ratio must be logically consistent); (5) DpConv SE incompatibility (DpConv does not support SE modules); (6) SeDpConv first-stage channel constraints (the first SeDpConv block must have channels equal to $input_channels$).

Memory rules include two primary keys: (1) For quantized models (i.e., models with quantization mode set to static, qat, or qat), memory is measured post-quantization; (2) All models must not exceed the given maximum memory budget, and any exceedance constitutes a violation.

Model configurations that violate architecture or memory rules are selected as samples for Validation tasks, while the repairs of these violated configurations serve as samples for repair tasks.

All these samples are then combined to form the validity dataset.

G.1.2 Improvement Dataset

The Improvement Dataset assists the LLM to learn performance optimization strategies by extracting successful architecture refinement trajectories. Specifically, we construct parent-child configuration pairs where the child node achieves higher accuracy than its parent. And the task type of this dataset is the **improvement** task.

For improvement tasks, the input consists of a parent configuration and its properties, while the output is an improved child configuration and its properties. Same as the improvement section of Figure 8, the child node achieves higher accuracy than its parent ($accuracy_{child} > accuracy_{parent}$) and satisfies the memory rules and architecture rules in each pair, ensuring that every sample represents a verified performance improvement.

G.1.3 Generation Dataset

The generation dataset helps the LLM to perform end-to-end architecture generation by learning direct mappings from accuracy and memory to complete model configurations. The input-output format aligns with the NAG problem equation: given user requirements, the LLM directly generates a model configuration that satisfies these constraints. Similar to previous datasets, the task type for this dataset is **generation**.

The input of generation tasks is a comprehensive instruction containing system prompts, task context (dataset information, memory constraints, search space), and crucially, the target accuracy acc_{target} and target memory mem_{target} , which is presented in the Figure 8. The output is a complete model configuration with all architectural parameters (stages, convolution types, channels, quantization modes, etc.) that satisfy the specified performance targets.

This design enables the LLM to learn the implicit mapping from performance specifications to architectural design choices. The generation dataset is also constructed across multiple memory constraints. The curated generation dataset enables the LLM to generate architecture across various memory.

G.1.4 Comparison Dataset

The comparison dataset aims the LLM to develop preference judgment capabilities by learning to distinguish between architectures of different quality

levels. This dataset is essential for reward preference optimization and reinforces the LM’s understanding of what constitutes a superior architecture design. In addition, the task type of the dataset is **comparison**. For each task, the input consists of: (i) a comprehensive instruction containing system prompts, task context, and search space constraints; (ii) two architecture configurations with their performance descriptions presented in randomized order to prevent positional bias; (iii) evaluation criteria specifying that configurations must first satisfy memory constraints ($\leq$ memory constraint), and higher accuracy is preferred among the configurations with validated memory. The output is structured into two parts: (i) the chosen configuration representing the superior design that satisfies memory constraints and achieves high accuracy, and (ii) the rejected configuration(s) representing inferior alternatives that either violate memory constraints or achieve lower performance.

As illustrated in the comparison section of Figure 8, the comparison pair construction employs two complementary strategies to capture diverse preference scenarios: (1) Valid vs invalid pairs: pairing high-quality valid configurations (accuracy $\geq$ chosen threshold, memory $\leq$ limit) with invalid configurations that achieve comparable or even higher accuracy but violate memory constraints, teaching the LLM that memory compliance is non-negotiable; (2) High-Quality vs lower-Quality pairs: pairing configurations both satisfying memory constraints but with significant accuracy differences (chosen accuracy $\geq$ chosen threshold, rejected accuracy $\leq$ chosen threshold), teaching the LLM to prefer higher-performance valid designs.

G.2 LLM Adaptation

With the progressive training dataset constructed, we employ a two-phase strategy to fine-tune LLaMA-3.1-8B (Dubey et al., 2024): the first phase uses progressive SFT pipeline to progressively establish the LM’s understanding of architecture rules and memory rules in section G.2.1, and the second phase applies reward preference optimization to refine the model’s judgment of architecture quality in section G.2.2. All training is conducted using LoRA (Low-Rank Adaptation) (Hu et al., 2022) for parameter-efficient fine-tuning.

G.2.1 Progressive SFT Pipeline

The progressive SFT pipeline employs a three-stage sequential fine-tuning strategy on the validity, im-

provement, and generation datasets, progressively teaching the LLM to generate architectures that satisfy memory constraints while achieving competitive accuracy. The training proceeds in the order: Validity $\rightarrow$ Improvement $\rightarrow$ Generation.

Stage 1: Validity Fine-Tuning. The first stage establishes the foundation by teaching the model constraint satisfaction and violation repair through two complementary task types: validation and repair.

For validation tasks, the model learns to identify violations in a given configuration. Each sample $(x_v^{\text{val}}, y_v^{\text{val}})$ consists of an input $x_v^{\text{val}} = (\text{instruction}, \theta)$ containing the task instruction and a model configuration θ , and a structured output $y_v^{\text{val}} = (\text{is_valid}, \mathcal{V})$ where $\mathcal{V} = \{\mathcal{V}_{\text{arch}}, \mathcal{V}_{\text{mem}}\}$ represents the set of architecture and memory violations.

The training objective for validation tasks is:

$$\mathcal{L}_{\text{val}} = -\mathbb{E}_{(\mathbf{x}_v^{\text{val}}, \mathbf{y}_v^{\text{val}}) \sim \mathcal{D}_{\text{val}}} [\log p_{\phi}(\mathbf{y}_v^{\text{val}} | \mathbf{x}_v^{\text{val}})] . \quad (42)$$

For repair tasks, the input is $x_v^{\text{rep}} = (\text{instruction}, \theta_{\text{invalid}}, \mathcal{V})$ containing a violating configuration and its structured violation information, with output $y_v^{\text{rep}} = \theta_{\text{fixed}}$ being the corrected configuration.

The training objective for repair tasks is:

$$\mathcal{L}_{\text{rep}} = -\mathbb{E}_{(\mathbf{x}_v^{\text{rep}}, \mathbf{y}_v^{\text{rep}}) \sim \mathcal{D}_{\text{rep}}} [\log p_{\phi}(\mathbf{y}_v^{\text{rep}} | \mathbf{x}_v^{\text{rep}})] . \quad (43)$$

The overall training objective combines both tasks:

$$\mathcal{L}_{\text{validity}} = \mathcal{L}_{\text{val}} + \mathcal{L}_{\text{rep}}, \quad (44)$$

where ϕ denotes the LoRA-adapted model parameters, $\mathcal{D}_{\text{val}}$ and $\mathcal{D}_{\text{rep}}$ are the Validation and Repair datasets respectively, and both are jointly sampled during training. After this stage, the parameters of model is denoted as ϕ_1 .

This joint training strategy enables the model to learn bidirectional constraint reasoning: identifying what makes a configuration invalid (validation) and knowing how to fix violations (repair). The model internalizes six types of architecture rules (e.g., skip connection support, channel constraints, SE module compatibility) and memory budget enforcement, which are essential for subsequent stages.

Stage 2: Improvement Fine-Tuning. Building upon the memory-bounded foundation from Stage 1, the second stage teaches the model performance

Architecture Rules			Memory Rules
A1 DpConv/SeDpConv $skip \equiv False$	A2 DWSepConv’s expansion $>1 \equiv \text{MBCConv}$	A3 $in_{channel} == out_{channel}$ in SeDpConv	M1 Quantized models measured post-quantization
A4 $has_se=True \leftrightarrow se_ratio>0$	A5 $has_se \equiv False$ in Dp-Conv	A6 SeDpConv 1st-stage channel = input_channels	M2 Memory $\leq$ maximum memory constraint

Table 15: Constraint validation rules for neural architecture.

optimization strategies by learning from successful architecture refinements in trajectories.

Each training sample (x_i, y_i) represents a parent-child improvement pair extracted from the search tree, where the input $x_i = (\text{instruction}, \theta_p, \mathcal{M}_p)$ contains the parent configuration θ_p and its performance metrics $\mathcal{M}_p = \{\text{acc}_p, \text{mem}_p, \text{quant}_p\}$, and the output $y_i = (\theta_c, \mathcal{M}_c)$ is the improved child configuration θ_c achieving higher accuracy $\text{acc}_c > \text{acc}_p$ while satisfying memory constraints.

The training objective is:

$$\mathcal{L}_{\text{improvement}} = -\mathbb{E}_{(x_i, y_i) \sim \mathcal{D}_{\text{improvement}}} \left[\log p_{\phi_2}(y_i | x_i; \phi_1) \right], \quad (45)$$

where $\mathcal{D}_{\text{improvement}}$ consists of trajectory pairs (θ_p, θ_c) satisfying $\text{acc}_c > \text{acc}_p$. By learning from these successful refinement trajectories, the model internalizes diverse optimization strategies including convolution type selection (e.g., DWSepConv $\rightarrow$ MBCConv for higher capacity), channel expansion for increased representational ability, and quantization-aware design for memory-accuracy trade-offs. The model learns to generate architectures that improve upon given parent architectures rather than starting from scratch, which is crucial for the iterative refinement capability required in Stage 3. **Stage 3: Generation Fine-Tuning.** The final stage trains the model for end-to-end architecture generation, enabling it to directly generate complete configurations without iterative search. Unlike Stages 1 and 2 which teach constraint satisfaction and incremental improvement, this stage consolidates these capabilities into direct generation.

Each training sample (x_g, y_g) consists of an input $x_g = (\text{instruction}, \mathcal{R}_{\text{target}})$ specifying the task context and target performance requirements $\mathcal{R}_{\text{target}} = \{\text{acc}_{\text{target}}, \text{mem}_{\text{target}}\}$, with an empty configuration input, and the output $y_g = \theta_{\text{complete}}$ is a complete model configuration achieving the specified targets.

The training objective is:

$$\mathcal{L}_{\text{generation}} = -\mathbb{E}_{(x_g, y_g) \sim \mathcal{D}_{\text{generation}}} [\log p_{\phi_3}(y_g | x_g; \phi_2)]. \quad (46)$$

This stage teaches the model to internalize the complete mapping from performance requirements to architectural instantiation, learning not just individual design choices but their holistic combination to achieve target performance levels. By training on successful MCTS discoveries, the model learns to generate architectures directly without search, effectively distilling the search process into the LLM’s parameters.

G.2.2 Reward Preference Optimization

After the SFT pipeline establishes foundational architecture generation capabilities, we apply reward preference optimization to refine the model’s judgment of architecture quality through preference learning, which is based on Direct Preference Optimization (DPO) (Rafailov et al., 2024). Unlike traditional Reinforcement Learning from Human Feedback (RLHF) (Ouyang et al., 2022) which requires training a separate reward model, DPO directly optimizes the LLM using preference pairs with an implicit reward function.

We define a performance-aware reward function $\mathbb{R}$ that encodes our optimization priorities:

$$\mathbb{R}(\theta) = \begin{cases} \text{Acc}(\theta) & \text{if } \text{mem}(\theta) \leq M_{\text{max}} \\ 0 & \text{otherwise} \end{cases}, \quad (47)$$

where $\text{Acc}(\theta)$ and $\text{mem}(\theta)$ denote the accuracy and memory consumption of configuration θ , and M_{max} is the memory budget. This reward formulation enforces a hard constraint on memory: violating configurations receive zero reward regardless of accuracy, while rewarding valid configurations based on their accuracy.

Critically, the prompt x in each comparison pair presents both configurations’ performance metrics in randomized order, without revealing which metrics correspond to which configuration.

$$x = (\text{instruction}, \text{rand}\{\{\text{acc}_c, \text{mem}_c, \text{quant}_c\}, \{\text{acc}_r, \text{mem}_r, \text{quant}_r\}\}). \quad (48)$$

This forces the model to analyze architectural features and infer their performance implications, thereby internalizing the implicit mappings between design choices and outcomes.

Phase	Stage/Task	Input	Output	Purpose
Phase 1: Progressive SFT Pipeline	Stage 1: Validation	Instruction, Configuration θ	$\{\text{is_valid}, \mathcal{V}\}$ $\mathcal{V} = \{\mathcal{V}_{\text{arch}}, \mathcal{V}_{\text{mem}}\}$	Identify architecture and memory violations
	Stage 1: Repair	Instruction, $\theta_{\text{invalid}}, \text{Violations } \mathcal{V}$	Fixed configuration θ_{fixed}	Learn to correct constraint violations
	Stage 2: Improvement	Instruction, Parent θ_p , Metrics $\mathcal{M}_p = \{\text{acc}_p, \text{mem}_p, \text{quant}_p\}$	Child θ_c , Metrics $\mathcal{M}_c$ ($\text{acc}_c > \text{acc}_p$)	Learn performance optimization strategies
	Stage 3: Generation	Instruction, Target $\mathcal{R}_{\text{target}} = \{\text{acc}_{\text{target}}, \text{mem}_{\text{target}}\}$	Complete configuration θ_{complete}	End-to-end architecture generation
Phase 2: RPO	Reward Preference Optimization	Instruction, rand $\{\{\text{acc}_c, \text{mem}_c, \text{quant}_c\},$ $\{\text{acc}_r, \text{mem}_r, \text{quant}_r\}\}$	Chosen: y_c ($R(\theta_c) > R(\theta_r)$) Rejected: y_r	Refine preference judgment for memory-compliant, high-accuracy architectures

Note: RPO denotes Reward Preference Optimization, and Progressive SFT denotes Progressive SFT Pipeline.

Table 16: Training sample formats in Dual-Phase Fine-Tuning. Each stage uses different input-output structures to progressively teach constraint satisfaction, performance optimization, direct generation, and preference judgment.

Given a preference pair (x, y_c, y_r) where y_c is the chosen configuration with $\mathbb{R}(\theta_c) > \mathbb{R}(\theta_r)$, the Reward Preference Optimization objective optimizes:

$$\mathcal{L}_{\text{DPO}}(\phi) = -\mathbb{E}_{(x, y_c, y_r) \sim \mathcal{D}_{\text{comparison}}} \left[\log \sigma \left(\beta \log p_{\phi}(y_c | x) - \beta \log p_{\text{ref}}(y_c | x) - \beta \log p_{\phi}(y_r | x) + \beta \log p_{\text{ref}}(y_r | x) \right) \right]. \quad (49)$$

where σ is the sigmoid function, β is the temperature parameter, p_{ϕ} is the current policy, and p_{ref} is the reference policy initialized from the Stage 3 SFT checkpoint. The log-probability ratios measure how much the policy prefers chosen over rejected responses relative to the reference policy.

Through this reward preference optimization, the model learns a nuanced decision boundary: it absolutely rejects memory-violating architectures ($\mathbb{R} = 0$) while making fine-grained quality distinctions among valid configurations based on accuracy.

H Responsible NLP Research Checklist

H.1 Benchmark Documentation

Table 17 summarizes the coverage of the HAR benchmarks used in this work, including activity classes, sample shapes, sensor channels, body positions, and representative scenarios.

H.2 Artifacts and licenses

We use publicly available HAR benchmarks, open-source baseline implementations when available, and LLaMA-3.1-8B as the backbone LLM. All external artifacts are used only for research and evaluation purposes, following their original access

conditions and licenses. The artifacts created in this work, including generated architecture configurations, training logs, and adapted models, are intended for research on memory-constrained architecture generation and edge HAR deployment.

H.3 Ethical Considerations: Privacy and Potential Risks

We do not collect new human-subject data. All experiments are conducted on existing public HAR benchmarks. The data used in this work consist of sensor-based activity samples rather than natural-language text, and we do not use names, identity labels, images, or other personally identifying information. We also do not attempt to infer individual identities from sensor traces.

QuaNAG targets HAR model generation for wearable and edge devices. Such models may be used in privacy-sensitive activity monitoring scenarios if deployed without appropriate safeguards. Our experiments are limited to public benchmarks and controlled edge-device evaluation, and the generated models are intended for research use rather than direct safety-critical or surveillance deployment.

H.4 Implementation Details and Hyperparameters

Table 18 reports the launcher-level hyperparameters used for quantization-branched MCTS. Dataset-specific memory budgets follow the in-domain and out-of-domain settings described in the experimental setup.

Table 19 summarizes the main hyperparameters used for the constraint-to-quality curriculum. QuaNAG uses LLaMA-3.1-8B as the backbone LM,

Benchmark	#Classes	Sample Shape	Body Position	Activity Coverage
Mhealth (Banos et al., 2014)	12	569×500×23	Chest, right wrist, and left ankle	Health-related daily activities, including standing, sitting, lying, walking, climbing stairs, forward bending, and jumping.
MMAct (Kong et al., 2019)	35	7843×250×9	Pants pocket	Daily and transportation-related actions, including walking, running, cycling, driving, static postures such as sitting/standing, and transition actions.
Wharf (Bruno et al., 2013)	14	1322×320×3	Wrist	Industrial and warehouse activities, including loading/unloading goods, forklift operation, walking, standing, and dock/yard operation behaviors.
USCHAD (Zhang and Sawchuk, 2012)	12	2236×1000×6	Right hip	Daily activities, including walking in different directions, running, going upstairs/downstairs, jumping, sitting, standing, lying, and taking an elevator up/down.
UTD-MHAD (Chen et al., 2015)	27	861×300×6	Right wrist or right thigh	Sports and interactive actions, including waving, punching, clapping, jumping, bending/squatting, and interaction actions such as throwing and catching.

Table 17: Documentation of HAR benchmarks used in this work. The sample shape denotes the number of samples, temporal length, and sensor channels.

and all SFT and DPO stages are performed with LoRA-based parameter-efficient fine-tuning.

H.5 Computational Details

QuaNAG uses LLaMA-3.1-8B as the backbone LM and performs both SFT and DPO with LoRA-based parameter-efficient fine-tuning. All experiments are conducted on a workstation equipped with 4 NVIDIA RTX 4090 GPUs, dual Intel Xeon Gold 6430 processors, Ubuntu, and CUDA 12.8. We report the main LoRA fine-tuning hyperparameters in Table 19 and the main search settings in Table 18. The LoRA-based fine-tuning pipeline takes approximately 5-9 hours in total on our workstation for each benchmark. Specifically, the validity SFT stage takes about 1.9-4.0 hours, the improvement SFT stage takes about 0.8-2.1 hours, the generation SFT stage takes about 0.6-2.1 hours, and the final DPO stage takes about 2.2-5.4 hours.

H.6 Software and packages

We implement QuaNAG using PyTorch 2.4.0 and HuggingFace Transformers 4.56.2. LoRA-based SFT and DPO are conducted with PEFT 0.17.1, TRL 0.9.6, and LLaMA-Factory 0.9.4.dev0. Dataset processing is implemented with Datasets 4.0.0, NumPy 1.26.4, Pandas 2.3.3, and Scikit-learn 1.6.1. The experiments use CUDA 12.8 with TorchVision 0.19.0 and TorchAudio 2.4.0. Static PTQ, QAT, and QAFT are implemented with PyTorch quantization utilities.

H.7 Use of AI assistants

AI assistants were used to support language polishing, code debugging, and writing refinement. All technical claims, experimental results, and final manuscript content were checked and verified by the authors.

Hyperparameter	Value	Description
Search iterations	50	Number of MCTS iterations
Initial seeds	4	Number of initial forest root configurations
GPUs	4	Number of GPUs used for parallel search
Training epochs	200	Training epochs for each generated architecture
Exploration weight	6.0	UCT exploration coefficient
Timeout	300s	Timeout for HAR search when no new result is returned
Maximum memory	Dataset-specific	Memory budget specified by each experiment
Task type	HAR	Automatically inferred from the dataset
Search space	Dataset-specific	Defined by the architecture search-space configuration
Quantization modes	none/static/qat/qaft	Candidate quantization branches
Checkpointing	Enabled	Save and resume search states during MCTS

Table 18: Main hyperparameters for quantization-branched MCTS.

Setting	Validity SFT	Improvement SFT	Generation SFT	DPO
Stage	SFT	SFT	SFT	DPO
Backbone LM	LLaMA-3.1-8B			
Dataset type	Validity	Improvement	Generation	Comparison
Cutoff length	4096	4096	4096	4096
LoRA rank	16	16	16	16
LoRA alpha	32	32	32	32
LoRA dropout	0.05	0.05	0.05	0.05
LoRA target	all	all	all	all
Learning rate	5.0×10^{-5}	1.0×10^{-5}	3.0×10^{-5}	5.0×10^{-7}
Epochs	3	6	3	3
Train batch size	2	2	2	2
Eval batch size	2	2	2	2
Gradient accumulation	8	8	8	8
Warmup steps	20	40	30	30
Validation split	0.1	0.1	0.1	0.05
Optimizer	adamw_torch	adamw_torch	adamw_torch	adamw_torch
LR scheduler	cosine	cosine	cosine	cosine
Max grad norm	1.0	1.0	1.0	1.0
Precision	fp16	fp16	fp16	fp16
Eval steps	100	100	100	100
Save steps	100	100	100	100
Logging steps	5	5	5	5
DPO beta	—	—	—	0.1
DPO loss	—	—	—	sigmoid

Table 19: Hyperparameters for the constraint-to-quality curriculum.