MONAQ: Multi-Objective Neural Architecture Querying for Time-Series Analysis on Resource-Constrained Devices

Patara Trirat
DeepAuto.ai
patara@deepauto.ai

Jae-Gil Lee ✉
KAIST
jaegil@kaist.ac.kr

Abstract

The growing use of smartphones and IoT devices necessitates efficient time-series analysis on resource-constrained hardware, which is critical for sensing applications such as human activity recognition and air quality prediction. Recent efforts in hardware-aware neural architecture search (NAS) automate architecture discovery for specific platforms; however, none focus on general time-series analysis with edge deployment. Leveraging the problem-solving and reasoning capabilities of large language models (LLM), we propose *MONAQ*, a novel framework that reformulates NAS into *M*ulti-*O*bjective *N*eural *A*rchitecture *Q*uerying tasks. *MONAQ* is equipped with *multimodal query generation* for processing multimodal time-series inputs and hardware constraints, alongside an *LLM agent-based multi-objective search* to achieve deployment-ready models via code generation. By integrating numerical data, time-series images, and textual descriptions, *MONAQ* improves an LLM's understanding of time-series data. Experiments on fifteen datasets demonstrate that *MONAQ*-discovered models outperform both handcrafted models and NAS baselines while being more efficient.

1 Introduction

The widespread adoption of smartphones, IoT devices, and wearables has intensified the demand for efficient time-series analysis on *resource-constrained* devices, essential for smart manufacturing, personalized healthcare (Samanta et al., 2024), and so on. These devices, often based on microcontroller units (MCU), are rapidly proliferating, with over 250B units worldwide (Lin et al., 2020, 2021). Deep learning on such affordable, energy-efficient hardware can democratize AI, enabling broad accessibility across diverse sectors.

However, tiny deep learning faces unique challenges due to stringent memory constraints. Typical MCUs, with less than 512kB SRAM, and even higher-end devices like Raspberry Pi 4 struggle to run conventional deep neural networks (Lin et al., 2022). Efficient AI inference demands innovative methods to navigate these limitations. Moreover, designing optimal network architectures and selecting hyperparameters for such devices is time-consuming and requires significant manual effort. Hardware-aware neural architecture search (HW-NAS) has emerged to automate this process, tailoring architectures to specific tasks and hardware (Li et al., 2021; Benmeziane et al., 2021a).

While NAS has advanced, it predominantly focuses on computer vision tasks and lacks generalizability for time-series analysis (White et al., 2023). Time-series applications span classification (e.g., human activity recognition) (Zhou et al., 2024c), regression (e.g., environmental monitoring) (Tan et al., 2021), and anomaly detection (e.g., industrial systems) (Nam et al., 2023). Existing NAS frameworks often target narrow use cases and fail to operate effectively within edge device constraints (Deng et al., 2022; Trirat and Lee, 2024; Saha et al., 2024).

Furthermore, current HW-NAS frameworks are frustrated by fixed search spaces and complex interfaces, making them less generalizable across tasks and inaccessible to non-experts. Large language models (LLM)-based NAS frameworks (Tornede et al., 2024) aim to address these issues but still rely on predefined search spaces and require user-provided initial architectures, limiting flexibility and usability (Chen et al., 2023; Dong et al., 2023; Jawahar et al., 2024; Nasir et al., 2024).

In contrast, to eliminate these undesirable user burdens, we reformulate the NAS problem as a multi-objective *neural architecture querying* (NAQ) problem by leveraging LLMs' advancements in reasoning and problem-solving. Unlike existing LLM-based NAS frameworks (Figure 1a), which require users to define a search space or an initial set of architectures, NAQ only requires natural language queries from users, decoupling them
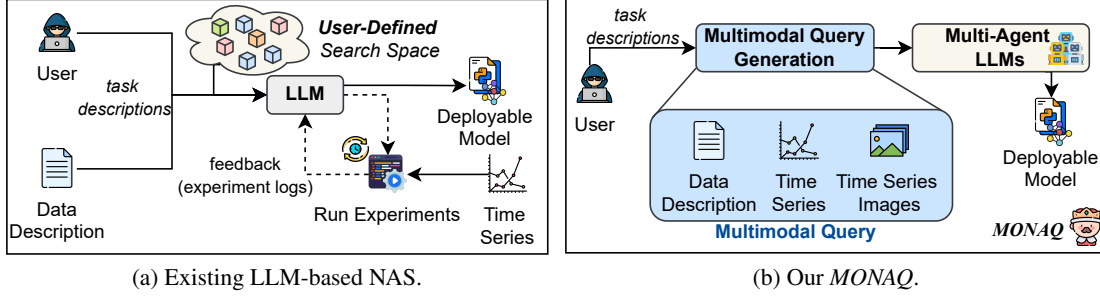
Figure 1: Comparison between (a) existing LLM-based NAS and (b) our proposed *MONAQ* framework.

from the architecture search process. This approach reduces reliance on human expertise while enhancing accessibility and flexibility in the model design process. It allows users to focus on addressing the actual problems in their datasets, leaving the complex search configurations to the LLM. However, achieving efficient NAQ poses two key challenges.

(1) How to find high-performing architectures without user-defined search spaces? Without user-defined configurations, enhancing the capability of LLMs to directly design high-performing network architectures becomes crucial. Recent studies (Hong et al., 2024; Xi et al., 2025) suggest that multi-agent LLMs improve problem-solving through collaboration among agents specialized in different tasks, while mitigating limitations found in single-agent LLMs, such as bias and hallucination. Building on these insights, we address this challenge by introducing an *LLM agent-based multi-objective search* module (Figure 1b). However, having multiple agents interact with each other can incur computational overhead. Therefore, instead of following the traditional setup in LLM-based NAS, which searches for architectures through runtime execution feedback (Figure 1a)—a process that demands significant training time and resource consumption—we leverage the pre-trained knowledge of LLMs during the evaluation step. As a result, this module enables specialized LLM agents to autonomously design search spaces and evaluate candidate models adaptively based on specific hardware constraints, eliminating the need for runtime execution. A coding-specialized LLM subsequently generates deployable architectures, ensuring low search costs and high flexibility.

(2) How to make LLM agents accurately understand time-series data and user requirements? Even though communication between agents can enhance problem-solving skills, LLMs still have inherent limitations in understanding time series. Unlike existing methods that rely solely on tex-

tual descriptions (Figure 1a) , we propose a *multimodal query generation* module that generates multimodal queries (Figure 1b) by leveraging both natural language and raw time series. This module processes input time series and natural language queries, including constraints such as hardware specifications and device names, and outputs multimodal data with time-series images that represent the original user query from both data and modeling perspectives. This comprehensive, multimodal approach enables LLMs to better understand the input time series and user queries (Kong et al., 2025).

By integrating these components, we present *MONAQ*, the first multi-agent LLM-based NAQ framework with an open-ended search space for time-series analysis on resource-constrained devices. Our **contributions** are as follows.

- We propose a novel LLM-based NAQ framework that creates constraint-aware architectures from user queries and datasets, tailored for time-series analysis on resource-constrained devices.
- We devise a multimodal query generation module to improve LLM understanding of time series through multi-objective queries with time-series images and introduce a multi-agent LLM module to reduce search costs via training-free search with specialized agents.
- Through extensive experiments on on-device time-series analysis, including classification and regression, we show that the models found by *MONAQ* outperform the second-best baseline by at least 9% on classification and 6% on regression tasks with significantly smaller, faster models.

2 Related Work

On-Device Time-Series Analysis Time-series analysis on resource-constrained devices, such as IoT and wearables, has gained importance due to the need for real-time processing with limited computational and energy resources (Trirat et al., 2024b). Common approaches include CNNs,

RNNs (e.g., LSTMs), and Transformers (Kara et al., 2024). While CNNs excel at extracting local context, they struggle with long-term dependencies (Hussein et al., 2024). RNNs and LSTMs address these drawbacks but are hindered by sequential processing, increasing latency. Transformers (Wen et al., 2023) enable parallel processing and capture long-term dependencies but are often unsuitable for edge devices due to high computational demands. Lightweight models like attention condensers, CNN-RNN hybrids (e.g., DeepConvLSTM (Ordóñez and Roggen, 2016)), and low-resource architectures like TinyHAR (Zhou et al., 2022) and MLP-HAR (Zhou et al., 2024c) balance performance with resource efficiency.

Hardware-Aware NAS (HW-NAS) Despite these advancements, HW-NAS for time-series data remains largely underexplored. MicroNAS (King et al., 2023) introduces time-series-specific search spaces for microcontrollers, while TinyT-NAS (Saha et al., 2024) supports efficient CPU operations. However, these methods often rely on fixed search spaces, requiring significant expertise. Given these challenges, there is a growing need for NAS frameworks with natural language interfaces, allowing users to describe their desired architecture in plain language rather than through direct programming (Tornede et al., 2024). Leveraging LLMs in this context can make HW-NAS more user-friendly and adaptable across a wider range of applications, leading to democratize NAS processes for better accessibility and adaptability.

LLMs for NAS LLMs have shown potential in automating NAS by leveraging pre-trained knowledge to generate diverse, high-performing architectures (Zheng et al., 2023; Dong et al., 2023; Wang et al., 2023; Jawahar et al., 2024; Nasir et al., 2024; Wang et al., 2024a; Zhou et al., 2024b; Chen et al., 2023; Trirat et al., 2024a). However, current LLM-based NAS frameworks face challenges with time-series data due to their limited understanding of raw numerical inputs (Merrill et al., 2024) and their reliance on runtime feedback and user-defined search spaces, which are often time-consuming and require intricate configurations. To address these issues, we introduce *neural architecture querying*, enabling users to specify requirements directly through natural language prompts for a given time-series dataset. Our approach removes the need for complex configurations and simplifies the architecture search process.

3   *MONAQ*: Multimodal NAQ with LLMs

3.1   Problem Formulation

Let $\mathcal{X}$ denote a d-variate time series with observations $(\mathbf{x}_1, \ldots, \mathbf{x}_T)$ where $\mathbf{x}_t \in \mathbb{R}^d$ and $\mathcal{Y}$ denote target variables. The target variables can be a set of integer values $\mathcal{Y} = (y_1, \ldots, y_T), y_t \in \mathbb{Z}$ (e.g., classification) or a set of real values $\mathcal{Y} = (y_1, \ldots, y_T), y_t \in \mathbb{R}$ (e.g., regression). Let $\mathcal{S}$ be a search space designed by an LLM and $\mathcal{M} = \{M_i\}_{i=1}^C$ denote a set of C candidate models sampled from $\mathcal{S}$. Each model M_i is a set of model configurations, e.g., layer types, number of hidden units, and activation functions. Then, let $\mathcal{E}$ denote an LLM responsible for evaluating each M_i. Finally, given a time series $\mathcal{X}$ and user task description with constraints $\mathcal{T}$, we aim to find the model $M^\star$ that satisfies the constraints in $\mathcal{T}$ on both downstream task performance and model complexity metrics using the LLM $\mathcal{E}$.

Neural Architecture Querying: *Given a training time series with its labels $\{\mathcal{X}_{train}, \mathcal{Y}_{train}\}$ and a natural language task description with constraints $\mathcal{T}$, select the model $M^\star$ that satisfies all constraints in $\mathcal{T}$. Formally, we solve*

$$M^\star = \arg\max_{M \in \mathcal{M}} \mathcal{E}(\mathcal{T}, \mathcal{X}_{\text{train}}, \mathbf{y}_{\text{train}}). \quad (1)$$

Note that the NAQ problem differs from NAS primarily from the user's perspective—that is, *whether the user is part of NAS's components* (e.g., search space and search method). Existing (LLM-based) NAS frameworks require users to define a search space, provide initial architectures, or even describe how to search, either via code or natural language, which demands significant technical expertise. In contrast, NAQ eliminates this requirement by allowing users to input high-level, natural language task descriptions and constraints based solely on domain- (or data-) specific problems as a query. This reformulation simplifies the process, making architecture discovery accessible to non-experts while maintaining efficiency. Figure 2 illustrates the overview of *MONAQ*.

3.2   Multimodal Query Generation

In this subsection, we describe how to generate a multimodal query as the input to an LLM.

Query Rewriting First, we rewrite the user task description into an organized form, such as a JSON with specific key-value pairs, representing a multi-objective query that encompasses both data and
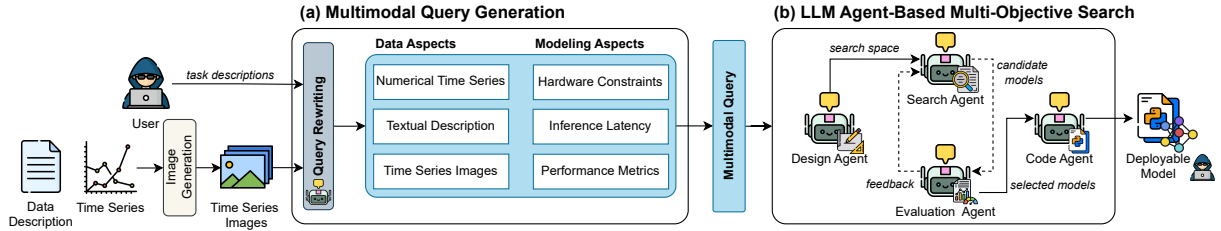
Figure 2: Overall procedure of our framework. *MONAQ* first receives a user prompt and a time series with descriptions. It then generates time-series images and processes all required information through the multimodal query generation module (§3.2) to create an organized multimodal query. This query is subsequently shared across different specialized agents within the LLM agent-based multi-objective search module (§3.3). Once all agents successfully complete their tasks, the final model is returned to the user.

modeling aspects, thereby making it easier for LLMs to understand. This query rewriting process is designed to address potentially ambiguous or ill-structured user queries. The full prompt for the query rewriting is presented in §C.1.

Data Aspect Query To enhance the LLM's understanding of time series, given the limited context window size, we first create a representative time series from the training set, as providing all time-series samples is both impractical and unnecessarily costly. Specifically, we use one time series per class for classification tasks and one time series per range for regression tasks. These representative time series are then used to construct queries for the subsequent search process, serving as input queries for multimodal LLMs.

- **Numerical Time Series**. As validated by Fons et al. (2024), we adopt csv formatting with a fixed length for the representative numerical time series, as it provides structural information that helps LLMs better understand numerical values. Specifically, we compute the timestamp-wise *average* of all time series in the training set to generate the representative numerical time series.
- **Textual Descriptions**. Since time-series values alone may not provide sufficient information about the dataset's source or the potential significance of each observation, we provide both dataset-level and feature-level descriptions to the LLMs. These descriptions help the models capture the context related to the application domain and the specific setting of the given dataset.
- **Time Series Images**. As shown in recent studies (Li et al., 2023; Merrill et al., 2024; Chow et al., 2024; Zhuang et al., 2024; Cai et al., 2024; Liu et al., 2024), LLMs demonstrate a better understanding of time series when it is provided in the form of images. This is due to the discernible trends and seasonal patterns in time-series im-
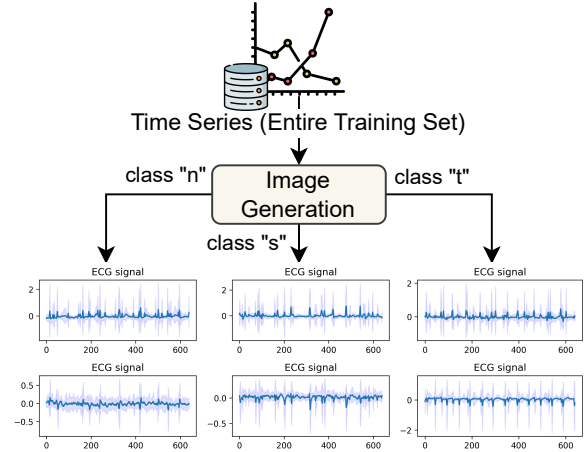

Figure 3: Examples of representative time series images containing two-channel ECG signals.

ages, which LLMs may struggle to capture when relying solely on limited numerical values. Following Li et al. (2023); Zhuang et al. (2024), we represent each variable as a line chart and stack these charts into a single image. Unlike existing studies, as described earlier, we compute only the timestamp-wise average of all *full-length* time series to generate an image that represents a sample for each class or label range, along with its standard deviation. This approach reduces costs while preserving the key characteristics of each class or range. Examples of the resulting images from bivariate time series are shown in Figure 3.

Using *numerical time series*, *textual descriptions*, and *time series images*, we prompt the LLM to rewrite queries into the JSON format with keys name, description, features, context, and patterns, representing various aspects of the data.

Modeling Aspect Query For the modeling aspect, we ask the LLM to rewrite the user query with a focus on key considerations for building efficient models in resource-constrained environments.
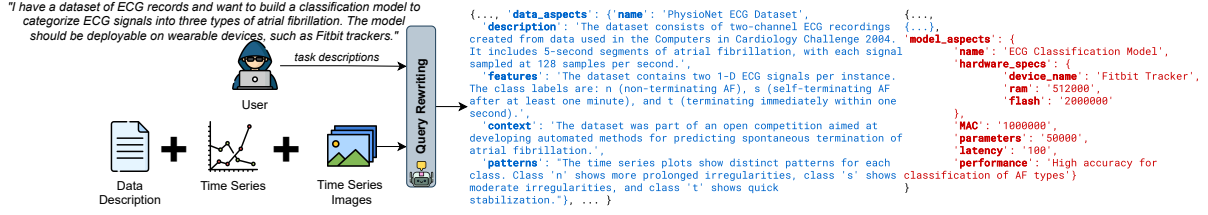
- **Hardware Constraints**. The hardware con-

Figure 4: A complete example of multimodal query generation results, showing data and modeling aspects.

straints can be specified directly by the user or through the name of the target device.

- **Inference Latency**. Similarly, if the user provides specific requirements, we instruct the LLM to adhere to them; otherwise, we instruct the LLM to rewrite the query to account for possible latency based on the hardware constraints.
- **Model Complexity**. Likewise, if the user does not specify any constraints regarding model size or complexity, we instruct the LLM to infer potential limitations based on hardware constraints. The number of parameters corresponds to the model size in bytes, representing the FLASH memory required to store the model during deployment. The number of multiply-accumulate operations (MACs) or floating-point operations per second (FLOPs) must also be considered, as they indicate the peak memory (RAM) usage during inference on the target device.
- **Performance Metrics**. As a multi-objective search, we aim to optimize both task performance and efficiency for a target device. Users can either specify the metric(s), such as accuracy or root mean squared error, in the query, or the LLM can infer them from the downstream task.

Similar to the data aspect query, we prompt the LLM to rewrite queries into a structured JSON format with the keys name, hardware_specs, MAC, parameters, latency, and performance, which represent various aspects of the model design conditions. Figure 4 shows a complete resulting example of the multimodal query generation process.

3.3 LLM Agent Based Multi-Objective Search

In this module, we leverage the full potential of LLMs through task specialization and collaborative problem-solving. As multi-agent LLM systems decompose complex tasks into manageable components, they enhance both accuracy and reasoning capabilities compared to single-agent systems (Guo et al., 2024b; Wang et al., 2024b). Through iterative communication, agents systematically evaluate trade-offs, refine architectural designs, and address challenges. Specifically, *MONAQ* harnesses multi-

agent collaboration to bypass extensive training, capitalizing on LLMs' reasoning abilities for efficient architecture evaluation and selection.

Below, we provide brief descriptions of the agents built for this module. Full prompts of agent specifications are presented in §C.2.

Design Agent ($\mathcal{A}_{design}$) is responsible for constructing and refining the potential search space based on the extracted multimodal query.

Search Agent ($\mathcal{A}_{search}$) is instructed to perform tasks related to architecture search and model design. The resulting designs produced by this agent are sent to the Evaluation Agent for evaluation and verification against the given multimodal query.

Evaluation Agent ($\mathcal{A}_{eval}$) is an LLM prompted for doing performance evaluation tasks ($\mathcal{E}$ in Eq. (1)) related to expected model performance, model profiling, and candidate ranking (when multiple models are suggested by $\mathcal{A}_{search}$).

Code Agent ($\mathcal{A}_{code}$) is an LLM prompted for implementing the solution verified by the Evaluation Agent. The Code Agent is responsible for writing effective code for actual runtime execution and returning the deployable model to the user.

Finally, as presented in Figure 2b, after obtaining the multimodal query from the multimodal query generation stage, $\mathcal{A}_{design}$ takes the organized multimodal query as its input and designs the search space $\mathcal{S}$ for $\mathcal{A}_{search}$. $\mathcal{A}_{search}$ then generates (a set) of candidate models ($\mathcal{M}$ in Eq. (1)) to be evaluated by $\mathcal{A}_{eval}$. If the suggested candidates pass the evaluation, based on the given constraints $\mathcal{T}$, the selected network is forwarded to $\mathcal{A}_{code}$, which writes the code to produce a deployable model for the user. Otherwise, *MONAQ* repeats the process by informing $\mathcal{A}_{search}$ with feedback from $\mathcal{A}_{eval}$ until the search budget is exhausted or a satisfactory model $M^\star$ is found.

4 Experiments

To verify the effectiveness of *MONAQ*, we conduct extensive experiments on two main on-device analysis tasks: classification and regression. Additionally, we perform ablation and hyperparameter

Datasets	Length	Feature Dims (# Sensors)	# Train	# Test	# Classes	Application Domain	Missing Values
Classification (Bagnall et al., 2018; Li et al., 2023)							
AtrialFibrillation	640	2	15	15	3	Health Monitoring	No
BinaryHeartbeat	18530	1	204	205	2	Health Monitoring	No
Cricket	1197	6	108	72	12	Human Activity Recognition	No
Fault Detection (A)	5120	1	10912	2728	3	Industrial System Monitoring	No
UCI-HAR	206	3	7352	2947	6	Human Activity Recognition	No
P12	233	36	9590	2398	2	Health Monitoring	**Yes**
P19	401	34	31042	7761	2	Health Monitoring	**Yes**
PAMAP2	4048	17	4266	1067	8	Human Activity Recognition	**Yes**
Regression (Tan et al., 2021)							
ApplianceEnergy	144	24	96	42		Energy Monitoring	No
BenzeneConcentration	240	8	3433	5445		Environment Monitoring	**Yes**
BIDMC32SpO2	4000	2	5550	2399		Health Monitoring	No
FloodModeling	266	1	471	202	N/A	Environment Monitoring	No
LiveFuelMoistureContent	365	7	3493	1510		Environment Monitoring	No
HouseholdPowerConsumption1	1440	5	746	694		Energy Monitoring	**Yes**
HouseholdPowerConsumption2	1440	5	746	694		Energy Monitoring	**Yes**

Table 1: Summary of benchmark datasets.

studies. The source code is available at `https://github.com/kaist-dmlab/MONAQ`.

4.1 Setup

Tasks and Datasets As summarized in Table 1, we select fifteen datasets for two downstream tasks commonly used in on-device time-series analysis, including classification and regression. These datasets are publicly available and represent various real-world applications, including healthcare, wearable devices, and environmental IoTs. For each task, we prepare a set of natural language task descriptions (see Table 4) as the input to LLM-based methods to represent user requirements along with a skeleton script (see §B.1).

Evaluation Metrics In terms of *model performance*, for the classification tasks, we adopt the accuracy metric, while for the regression tasks, we use the root mean squared error (RMSE) metric. For *model complexity*, we measure model size (i.e., FLASH storage size), peak memory usage during inference (i.e., RAM), the number of MAC operations, and inference latency using the MLTK library[1] as suggested by Saha et al. (2024). Model complexity results are based on a deterministic simulation on an EFR32xG24 at 78MHz with 1536kB of FLASH and 256kB of RAM.

Comparison Baselines As we address the novel problem of NAQ for time-series analysis on resource-constrained devices, no existing baselines are available for direct comparison. Thus, we

[1] `https://siliconlabs.github.io/mltk`

compare *MONAQ* against manually designed models based on TFLite-supported operations: **MLP**, **LSTM**, and **CNN**; hand-crafted lightweight models: temporal convolutional network (Bai et al., 2018) (**TCN**), depthwise convolution (**D-CNN**), depthwise separable convolution (Zhang et al., 2017) (**DS-CNN**), convolutional LSTM (Ordóñez and Roggen, 2016) (**ConvLSTM**), and 6-layer TENet (Li et al., 2020) (**TENet(6)**); state-of-the-art HW-NAS for time series: **TinyTNAS** (Saha et al., 2024); traditional NAS methods: **grid search** and **random search** (Lindauer and Hutter, 2020) in TinyTNAS's search space; and general-purpose LLMs: **GPT-4o-mini** and **GPT-4o** (Achiam et al., 2023) with zero-shot prompting (see §B.2).

Implementation Details Due to the need for complex problem-solving and reasoning skills, unless stated otherwise, we use GPT-4o (`gpt-4o-2024-08-06`) as the backbone model for all agents and LLM-based baselines to ensure an impartial performance evaluation. All experiments are conducted on an Ubuntu 20.04 LTS server equipped with an Intel(R) Xeon(R) Gold 6326 CPU @ 2.90GHz. To execute the generated models, we use the same environment provided by Guo et al. (2024a), which includes all necessary libraries in the skeleton scripts. Finally, all models are converted and quantized using TFLite Micro before calculating the model complexity metrics.

4.2 Main Results

Overall As in Figures 5 and 6, the models found by our proposed *MONAQ* framework, on average,
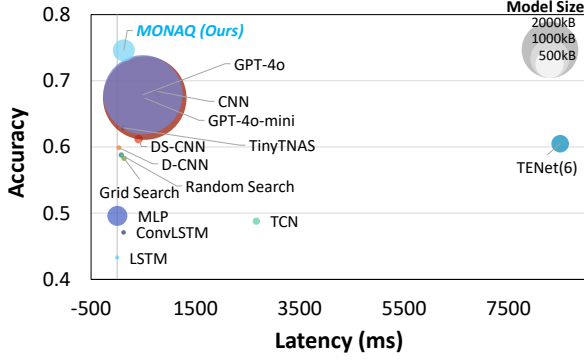
Figure 5: Performance comparison of our *MONAQ* and the baselines in average accuracy, inference latency, and model complexity (size) for **classification** tasks.
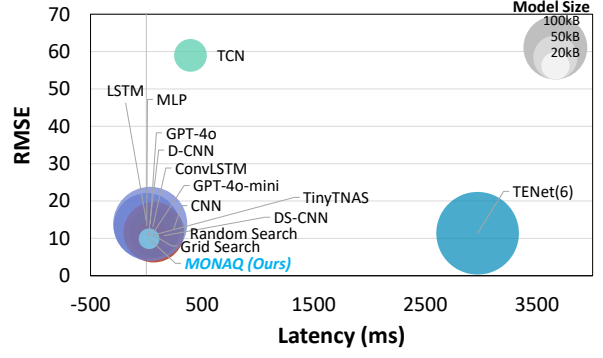

Figure 6: Performance comparison of our *MONAQ* and the baselines in terms of average RMSE, inference latency, and model complexity (size) for **regression** tasks.

significantly outperform baselines across multiple benchmarks. The center of each circle indicates downstream task performance and latency, while its diameter indicates model size. Compared to models with similar performance to *MONAQ* (such as CNNs and GPT-4o), the models found by *MONAQ* exhibit significantly greater efficiency in terms of model complexity. These findings demonstrate that *MONAQ* achieves a better balance between downstream task performance and model complexity.

Classification   The full results in Table 5 (Appendix) demonstrate that the models discovered by *MONAQ* achieve improvements over the baselines ranging from 9.1% to 72.1% in classification tasks, outperforming strong baselines, such as TinyTNAS, GPT-4o, and CNNs. In terms of model complexity, *MONAQ* significantly reduces memory consumption across tasks compared to DS-CNN and TENet, while also lowering computational costs (MAC) and achieving competitive latency on average. This result highlights its efficiency and effectiveness across different datasets.

Regression   Similarly, Table 6 (Appendix) shows that the models discovered by *MONAQ* achieve an error reduction of 6.3–83.2% compared to the baselines on regression tasks. On average, *MONAQ* outperforms all other methods, including state-of-the-art approaches, e.g., TinyTNAS, TENet, and D-CNN. Besides, *MONAQ* significantly reduces computational costs, while maintaining accuracy and offering competitive latency across datasets.

Irregular Time Series   To evaluate *MONAQ*'s robustness on irregular and noisy time series common in real-world settings, we test it on five more datasets. As shown in Table 7 (Appendix), *MONAQ* consistently achieves strong performance in both classification (average accuracy of 0.916) and re-

gression (average RMSE of 102.409) tasks, while also being highly efficient. It uses significantly less RAM, energy, and latency than several baselines.

4.3   Ablation Studies

To understand the contribution of each component, we conduct ablation studies by removing critical elements proposed in *MONAQ*. Tables 2 and 3 show the downstream performance and model complexity across different configurations.

Query Rewriting (Table 2)   Removing the query rewriting module results in a significant drop in classification accuracy (from 0.746 to 0.651) and an increase in regression RMSE (from 9.902 to 11.994), indicating its critical role in enhancing predictive performance. Although this variant reduces latency and memory usage, the performance loss confirms that query rewriting is vital for maintaining output quality.

Agent Contributions (Table 2)   The ablation of individual agents reveals their distinct roles. Excluding $\mathcal{A}_{design}$ leads to the higher latency and FLASH usage, while removing $\mathcal{A}_{eval}$ and $\mathcal{A}_{search}$ together results in both degraded accuracy and the largest model size. The baseline variant using only $\mathcal{A}_{code}$ performs worst across most metrics, with accuracy dropping to 0.612 and RMSE rising to 12.274. Despite low latency, its inefficiency in memory usage and poor predictive quality emphasize the necessity of agent collaboration.

Variations	Classification			Regression		
	Latency (ms)	Accuracy	FLASH (kB)	Latency (ms)	RMSE	FLASH (kB)
MONAQ	127.260	**0.746**	257.742	24.729	**9.902**	10.582
w/o Query Rewriting	206.871	0.651	17.186	14.623	11.994	10.155
w/o $\mathcal{A}_{design}$	863.358	0.647	518.762	95.654	13.512	33.243
w/o $\mathcal{A}_{eval}$	540.411	0.641	4775.661	26.335	12.783	109.627
w/o $\mathcal{A}_{eval}$ & $\mathcal{A}_{search}$	601.313	0.643	5907.363	188.123	11.261	665.110
Only $\mathcal{A}_{code}$	579.876	0.612	4158.205	21.530	12.274	99.638

Table 2: Ablation study results on query rewriting and various agent combinations.

Agents	Query Modality			Classification			Regression		
	Numerical Time Series	Textual Descriptions	Time Series Images	Latency (ms)	Accuracy	FLASH (kB)	Latency (ms)	RMSE	FLASH (kB)
Single (GPT-4o Backbone)	✓			519.159	0.679	3349.453	23.797	13.944	125.445
		✓		1017.267	0.665	4126.024	42.779	13.227	134.901
			✓	593.541	0.690	5971.792	35.859	12.562	193.926
	✓	✓		807.459	0.628	4926.157	40.485	13.556	137.581
	✓	✓	✓	557.665	0.629	3871.910	22.726	12.681	90.398
Multiple (GPT-4o Backbone)	✓			149.320	0.434	**12.066**	54.270	12.284	10.611
		✓		170.461	0.440	15.198	110.751	12.084	12.560
			✓	280.198	0.661	15.638	**13.661**	11.653	**7.885**
	✓	✓		205.623	0.517	16.035	28.049	13.207	13.875
	✓	✓	✓	**127.260**	**0.746**	257.742	24.729	**9.902**	10.582

Table 3: Ablation study results of multimodal query generation and multi-agent based search components.

(a) LLM Backbones.　(b) Number of Candidates.　(c) Search Budgets.
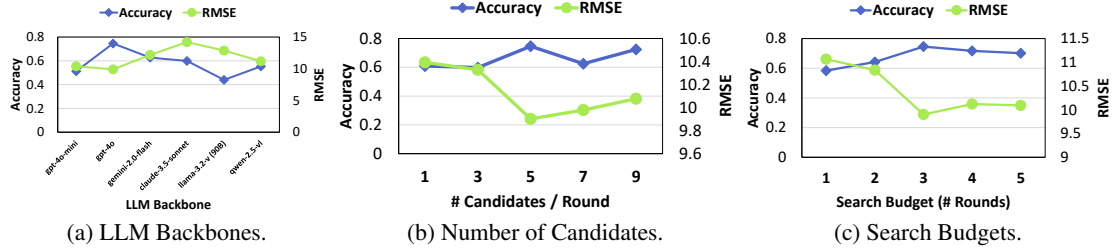
Figure 7: Comparison between (a) LLM backbones, (b) number of candidates per round, and (c) search budget on model performance, as measured by accuracy (higher is better) and RMSE (lower is better).

Multimodal Query Generation (Table 3) Combining multiple query modalities improves performance across downstream tasks. For classification tasks, accuracy increases as more modalities are included. For instance, in the single-agent setup, accuracy ranges from 0.628 to 0.679, depending on the modality combination. In regression tasks, RMSE decreases from 13.944 to 12.681 as query modalities expand. While multimodal inputs boost performance, they introduce higher latency and FLASH usage, especially in the single-agent setup.

Multi-Agent Based Search (Table 3) The multi-agent architecture significantly outperforms the single-agent model across all metrics. In classification tasks, interaction between agents (i.e., feedback) dramatically leads to the reduction in inference latency (e.g., 519ms in the single-agent model vs. 149ms in the multi-agent model), while accuracy improves due to the combination of modalities. The accuracy reaches 0.746, surpassing the single-agent model's peak value of 0.679. FLASH usage decreases with multi-agent search, even for complex queries. For regression tasks, the multi-agent search achieves superior accuracy, with RMSE values as low as 9.902 compared to the single-agent model's range of 12.562–13.944. Latency and FLASH usage are also significantly reduced. Consequently, multi-agent search not only reduces latency and memory usage but also enhances downstream performance.

Overall, we notice that multimodal query generation improves accuracy but increases complexity, especially in single-agent setups, whereas multi-agent-based search addresses these challenges by enhancing both aspects, thereby balancing downstream performance and model complexity.

4.4 Hyperparameter Studies

To understand the behavior of our framework under various settings, we further evaluate *MONAQ* with different hyperparameter configurations as follows.

LLM Backbones We evaluate *MONAQ* using both closed-source and open-source LLM backbones. The results in Figure 7a indicate that the choice of backbone has a noticeable impact on performance. Accuracy is higher for GPT-4o and Gemini-2.0-Flash, while RMSE is lower, suggesting better overall predictive performance. This trend underscores the importance of advanced LLMs in enhancing downstream performance.

Number of Candidates As the number of candidates per round increases, there is a clear upward trend in accuracy, while RMSE shows a corresponding decrease. Figure 7b suggests that expanding the candidate pool improves the model's ability to identify optimal solutions. However, the gains begin to plateau beyond a certain point, indicating diminishing returns for very large candidate pools.

Search Budgets Figure 7c shows that as the search budget increases, both accuracy and RMSE improve steadily. This trend indicates that additional iterations allow the search process to converge more effectively on better solutions. How-
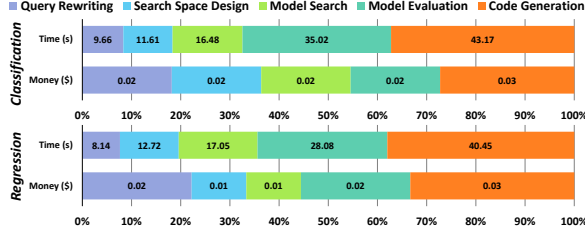
Figure 8: Average time and monetary cost breakdown for classification (upper) and regression (lower) tasks.

ever, the rate of improvement diminishes after a moderate number of rounds, suggesting that beyond a certain budget, the incremental benefit may not justify the added computational cost.

4.5 Resource Cost

As we use closed-source LLMs, we analyze the resource costs in terms of time and money. Figure 8 presents the average time and monetary costs across different datasets for a single run. On average, it takes around 200 seconds and costs 0.20 USD (using GPT-4o) to search for a single model that will be deployable after training. We also discuss the search cost comparison in §F.

5   Conclusions

This paper proposes a novel NAS framework, *MONAQ*, which reformulates NAS problems as multi-objective neural architecture querying tasks, using multimodal time-series inputs and hardware constraints as queries to LLMs. To enhance the LLM's understanding of time series, we introduce a multimodal query generation module and improve search efficiency via a multi-agent based search. Extensive experiments on 15 datasets show that the models discovered by *MONAQ* outperform handcrafted baselines while achieving greater efficiency.

Limitations

While the proposed *MONAQ* framework demonstrates significant advancements in resource-constrained time-series analysis, there are a few limitations to consider.

First, the reliance on large language models (LLM) for neural architecture querying introduces a dependency on the availability of advanced LLMs, which can incur high computational costs during the search process. Although *MONAQ* bypasses the training of candidate models, the multimodal query generation and multi-agent search process may still be computationally intensive for

scenarios requiring real-time or low-latency architecture optimization.

Second, the framework assumes well-defined user constraints and task descriptions, which may limit its applicability in ambiguous or ill-structured deployment scenarios.

Third, *MONAQ*'s effectiveness in handling highly noisy or irregular time-series data has not been extensively validated, which could impact its performance in applications like industrial anomaly detection. Future work could focus on expanding the robustness of the framework across broader datasets and enhancing the adaptability of its architecture discovery process to dynamic and uncertain deployment environments.

Acknowledgments

References

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, and 1 others. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Anthony Bagnall, Hoang Anh Dau, Jason Lines, Michael Flynn, James Large, Aaron Bostrom, Paul Southam, and Eamonn Keogh. 2018. The uea multivariate time series classification archive, 2018. *arXiv:1811.00075*.

Shaojie Bai, J Zico Kolter, and Vladlen Koltun. 2018. An empirical evaluation of generic convolutional and recurrent networks for sequence modeling. *arXiv preprint arXiv:1803.01271*.

Hadjer Benmeziane and Kaoutar El Maghraoui. 2024. Are large language models good neural architecture generators for edge? In *IEEE EDGE*, pages 162–165.

Hadjer Benmeziane, Kaoutar El Maghraoui, Hamza Ouarnoughi, Smail Niar, Martin Wistuba, and Naigang Wang. 2021a. Hardware-aware neural architecture search: Survey and taxonomy. In *IJCAI*, pages 4322–4329.

Hadjer Benmeziane, Kaoutar El Maghraoui, Hamza Ouarnoughi, Smail Niar, Martin Wistuba, and Naigang Wang. 2021b. A comprehensive survey on hardware-aware neural architecture search. *arXiv preprint arXiv:2101.09336*.

Han Cai, Ligeng Zhu, and Song Han. 2019. Proxyless-NAS: Direct neural architecture search on target task and hardware. In *ICLR*.

Yifu Cai, Arjun Choudhry, Mononito Goswami, and Artur Dubrawski. 2024. Timeseriesexam: A time series understanding exam. *arXiv preprint arXiv:2410.14752*.

Angelica Chen, David Dohan, and David So. 2023. Evo-prompting: Language models for code-level neural architecture search. In *NeurIPS*.

Winnie Chow, Lauren Gardiner, Haraldur T Hallgrímsson, Maxwell A Xu, and Shirley You Ren. 2024. Towards time series reasoning with llms. *arXiv preprint arXiv:2409.11376*.

Difan Deng, Florian Karl, Frank Hutter, Bernd Bischl, and Marius Lindauer. 2022. Efficient automated deep learning for time series forecasting. In *ECML PKDD*.

Haoyuan Dong, Yang Gao, Haishuai Wang, Hong Yang, and Peng Zhang. 2023. Heterogeneous graph neural architecture search with gpt-4. *arXiv preprint arXiv:2312.08680*.

Elizabeth Fons, Rachneet Kaur, Soham Palande, Zhen Zeng, Svitlana Vyetrenko, and Tucker Balch. 2024. Evaluating large language models on time series feature understanding: A comprehensive taxonomy and benchmark. In *EMNLP*.

Siyuan Guo, Cheng Deng, Ying Wen, Hechang Chen, Yi Chang, and Jun Wang. 2024a. DS-agent: Automated data science by empowering large language models with case-based reasoning. In *ICML*.

Taicheng Guo, Xiuying Chen, Yaqi Wang, Ruidi Chang, Shichao Pei, Nitesh V. Chawla, Olaf Wiest, and Xiangliang Zhang. 2024b. Large language model based multi-agents: A survey of progress and challenges. In *IJCAI*, pages 8048–8057.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. 2024. MetaGPT: Meta programming for a multi-agent collaborative framework. In *ICLR*.

Dina Hussein, Lubah Nelson, and Ganapati Bhat. 2024. Sensor-aware classifiers for energy-efficient time series applications on iot devices. *arXiv preprint arXiv:2407.08715*.

Ganesh Jawahar, Muhammad Abdul-Mageed, Laks VS Lakshmanan, and Dujian Ding. 2024. Llm performance predictors are good initializers for architecture search. In *Findings of ACL*.

Denizhan Kara, Tomoyoshi Kimura, Shengzhong Liu, Jinyang Li, Dongxin Liu, Tianshi Wang, Ruijie Wang, Yizhuo Chen, Yigong Hu, and Tarek Abdelzaher. 2024. FreqMAE: Frequency-aware masked autoencoder for multi-modal iot sensing. In *Proceedings of the ACM on Web Conference 2024*, pages 2795–2806.

Tobias King, Yexu Zhou, Tobias Röddiger, and Michael Beigl. 2023. Micronas: Memory and latency constrained hardware-aware neural architecture search for time series classification on microcontrollers. *arXiv preprint arXiv:2310.18384*.

Yaxuan Kong, Yiyuan Yang, Shiyu Wang, Chenghao Liu, Yuxuan Liang, Ming Jin, Stefan Zohren, Dan Pei, Yan Liu, and Qingsong Wen. 2025. Position: Empowering time series reasoning with multimodal llms. *Preprint*, arXiv:2502.01477.

Chaojian Li, Zhongzhi Yu, Yonggan Fu, Yongan Zhang, Yang Zhao, Haoran You, Qixuan Yu, Yue Wang, Cong Hao, and Yingyan Lin. 2021. HW-NAS-Bench: Hardware-aware neural architecture search benchmark. In *International Conference on Learning Representations*.

Ximin Li, Xiaodong Wei, and Xiaowei Qin. 2020. Small-footprint keyword spotting with multi-scale temporal convolution. In *Interspeech*, pages 1987–1991.

Zekun Li, Shiyang Li, and Xifeng Yan. 2023. Time series as images: Vision transformer for irregularly sampled time series. In *NeurIPS*.

Ji Lin, Wei-Ming Chen, Han Cai, Chuang Gan, and Song Han. 2021. MCUNetV2: Memory-efficient patch-based inference for tiny deep learning. In *NeurIPS*, pages 2346–2358.

Ji Lin, Wei-Ming Chen, Yujun Lin, John Cohn, Chuang Gan, and Song Han. 2020. MCUNet: Tiny deep learning on iot devices. In *NeurIPS*, pages 11711–11722.

Ji Lin, Ligeng Zhu, Wei-Ming Chen, Wei-Chen Wang, Chuang Gan, and Song Han. 2022. On-device training under 256kb memory. In *NeurIPS*, pages 22941–22954.

Marius Lindauer and Frank Hutter. 2020. Best practices for scientific research on neural architecture search. *JMLR*, 21(243):1–18.

Haoxin Liu, Chenghao Liu, and B Aditya Prakash. 2024. A picture is worth a thousand numbers: Enabling llms reason about time series via visualization. *arXiv preprint arXiv:2411.06018*.

Mike A Merrill, Mingtian Tan, Vinayak Gupta, Thomas Hartvigsen, and Tim Althoff. 2024. Language models still struggle to zero-shot reason about time series. In *Findings of EMNLP*, pages 3512–3533.

Youngeun Nam, Patara Trirat, Taeyoon Kim, Youngseop Lee, and Jae-Gil Lee. 2023. Context-aware deep time-series decomposition for anomaly detection in businesses. In *ECML PKDD*, page 330–345.

Muhammad Umair Nasir, Sam Earle, Julian Togelius, Steven James, and Christopher Cleghorn. 2024. Llmatic: neural architecture search via large language models and quality diversity optimization. In *GECCO*, pages 1110–1118.

Francisco Javier Ordóñez and Daniel Roggen. 2016. Deep convolutional and lstm recurrent neural networks for multimodal wearable activity recognition. *Sensors*, 16(1):115.

Md Hafizur Rahman and Prabuddha Chakraborty. 2024. Lemo-nade: Multi-parameter neural architecture discovery with llms. *arXiv preprint arXiv:2402.18443*.

Bidyut Saha, Riya Samanta, Soumya K Ghosh, and Ram Babu Roy. 2024. Tinytnas: Gpu-free, time-bound, hardware-aware neural architecture search for tinyml time series classification. *arXiv preprint arXiv:2408.16535*.

Riya Samanta, Bidyut Saha, Soumya K Ghosh, and Ram Babu Roy. 2024. Optimizing tinyml: The impact of reduced data acquisition rates for time series classification on microcontrollers. *arXiv preprint arXiv:2409.10942*.

Chang Wei Tan, Christoph Bergmeir, Francois Petitjean, and Geoffrey I Webb. 2021. Time series extrinsic regression. *DMKD*, pages 1–29.

Alexander Tornede, Difan Deng, Theresa Eimer, Joseph Giovanelli, Aditya Mohan, Tim Ruhkopf, Sarah Segel, Daphne Theodorakopoulos, Tanja Tornede, Henning Wachsmuth, and Marius Lindauer. 2024. AutoML in the age of large language models: Current challenges, future opportunities and risks. *TMLR*.

Patara Trirat, Wonyong Jeong, and Sung Ju Hwang. 2024a. Automl-agent: A multi-agent llm framework for full-pipeline automl. *arXiv preprint arXiv:2410.02958*.

Patara Trirat and Jae-Gil Lee. 2024. PASTA: Neural architecture search for anomaly detection in multivariate time series. *IEEE Transactions on Emerging Topics in Computational Intelligence*, pages 1–16.

Patara Trirat, Yooju Shin, Junhyeok Kang, Youngeun Nam, Jihye Na, Minyoung Bae, Joeun Kim, Byunghyun Kim, and Jae-Gil Lee. 2024b. Universal time-series representation learning: A survey. *arXiv preprint arXiv:2401.03717*.

Haishuai Wang, Yang Gao, Xin Zheng, Peng Zhang, Hongyang Chen, Jiajun Bu, and Philip S Yu. 2023. Graph neural architecture search with gpt-4. *arXiv preprint arXiv:2310.01436*.

Jialiang Wang, Shimin Di, Hanmo Liu, Zhili Wang, Jiachuan Wang, Lei Chen, and Xiaofang Zhou. 2024a. Computation-friendly graph neural network design by accumulating knowledge on large language models. *arXiv preprint arXiv:2408.06717*.

Yanlin Wang, Wanjun Zhong, Yanxian Huang, Ensheng Shi, Min Yang, Jiachi Chen, Hui Li, Yuchi Ma, Qianxiang Wang, and Zibin Zheng. 2024b. Agents in software engineering: Survey, landscape, and vision. *arXiv preprint arXiv:2409.09030*.

Qingsong Wen, Tian Zhou, Chaoli Zhang, Weiqi Chen, Ziqing Ma, Junchi Yan, and Liang Sun. 2023. Transformers in time series: A survey. In *IJCAI*.

Colin White, Mahmoud Safari, Rhea Sukthanker, Binxin Ru, Thomas Elsken, Arber Zela, Debadeepta Dey, and Frank Hutter. 2023. Neural architecture search: Insights from 1000 papers. *arXiv preprint arXiv:2301.08727*.

Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, and 1 others. 2025. The rise and potential of large language model based agents: A survey. *Science China Information Sciences*, 68(2):121101.

Yundong Zhang, Naveen Suda, Liangzhen Lai, and Vikas Chandra. 2017. Hello edge: Keyword spotting on microcontrollers. *arXiv preprint arXiv:1711.07128*.

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*.

Ao Zhou, Jianlei Yang, Yingjie Qi, Tong Qiao, Yumeng Shi, Cenlin Duan, Weisheng Zhao, and Chunming Hu. 2024a. Hgnas: Hardware-aware graph neural architecture search for edge devices. *IEEE TC*, pages 1–14.

Xun Zhou, Liang Feng, Xingyu Wu, Zhichao Lu, and Kay Chen Tan. 2024b. Design principle transfer in neural architecture search via large language models. *arXiv preprint arXiv:2408.11330*.

Yexu Zhou, Tobias King, Haibin Zhao, Yiran Huang, Till Riedel, and Michael Beigl. 2024c. Mlp-har: Boosting performance and efficiency of har models on edge devices with purely fully connected layers. In *ISWC*, pages 133–139.

Yexu Zhou, Haibin Zhao, Yiran Huang, Till Riedel, Michael Hefenbrock, and Michael Beigl. 2022. Tiny-har: A lightweight deep learning model designed for human activity recognition. In *ISWC*, page 89–93.

Jiaxin Zhuang, Leon Yan, Zhenwei Zhang, Ruiqi Wang, Jiawei Zhang, and Yuantao Gu. 2024. See it, think it, sorted: Large multimodal models are few-shot time series anomaly analyzers. *arXiv preprint arXiv:2411.02465*.

A Extended Related Work

A.1 On-Device Time-Series Analysis

Time-series analysis on resource-constrained devices, such as IoT and wearables, has gained importance due to the need for real-time processing with limited computational and energy resources (Trirat et al., 2024b). Common approaches include CNNs, RNNs (e.g., LSTMs), and Transformers (Kara et al., 2024). While CNNs excel at extracting local context, they struggle with long-term dependencies (Hussein et al., 2024). RNNs and LSTMs address these drawbacks but are hindered by sequential processing, increasing latency. Transformers (Wen et al., 2023) enable parallel processing and capture long-term dependencies but are often unsuitable for edge devices due to high computational demands. Lightweight models like attention condensers, CNN-RNN hybrids (e.g., DeepConvLSTM (Ordóñez and Roggen, 2016)), and low-resource architectures like TinyHAR (Zhou et al., 2022) and MLP-HAR (Zhou et al., 2024c) balance performance with resource efficiency.

A.2 Hardware-Aware NAS (HW-NAS)

Optimizing neural networks for hardware constraints, such as memory and latency, is challenging and traditionally required extensive domain expertise. HW-NAS addresses this issue by incorporating hardware efficiency metrics into the search process (Li et al., 2021; Benmeziane et al., 2021b,a). ProxylessNAS (Cai et al., 2019) optimizes latency and energy consumption on edge devices, while the MCUNet family (Lin et al., 2020, 2021, 2022) enhances efficiency for microcontrollers. More recent methods, e.g., HGNAS (Zhou et al., 2024a), integrate lookup tables and predictors for resource-efficient searches.

Despite these advancements, HW-NAS for time-series data remains largely underexplored. MicroNAS (King et al., 2023) introduces time-series-specific search spaces for microcontrollers, while TinyTNAS (Saha et al., 2024) supports efficient CPU operations. However, these methods often rely on fixed search spaces, requiring significant expertise. Given these challenges, there is a growing need for NAS frameworks with natural language interfaces, allowing users to describe their desired architecture in plain language rather than through direct programming (Tornede et al., 2024). Leveraging LLMs in this context can make HW-NAS more user-friendly and adaptable across a wider range of applications, leading to democratize NAS processes for better accessibility and adaptability.

A.3 LLMs for NAS

LLMs have shown potential in automating NAS by leveraging pre-trained knowledge to generate diverse, high-performing architectures (Zheng et al., 2023; Dong et al., 2023; Wang et al., 2023; Benmeziane and El Maghraoui, 2024; Jawahar et al., 2024; Nasir et al., 2024; Wang et al., 2024a; Zhou et al., 2024b; Rahman and Chakraborty, 2024; Chen et al., 2023; Trirat et al., 2024a). GENIUS (Zheng et al., 2023) improves convolution-based architectures through feedback, while GPT4GNAS (Wang et al., 2023) uses GPT-4 to design graph neural networks. LLMatic (Nasir et al., 2024) combines LLMs with quality-diversity algorithms, generating architectures that balance diversity and performance across various metrics and thus achieving competitive results with fewer evaluations. These frameworks suggest a shift toward using LLMs not only as code generators but also as sophisticated tools for automating NAS.

However, current LLM-based NAS frameworks face challenges with time-series data due to their limited understanding of raw numerical inputs (Merrill et al., 2024; Kong et al., 2025) and their reliance on runtime feedback and user-defined search spaces, which are often time-consuming and require intricate configurations. To address these issues, we introduce *neural architecture querying*, enabling users to specify requirements directly through natural language prompts for a given time-series dataset. Our approach removes the need for complex configurations and simplifies the architecture search process.

B Details of Experimental Setup

This section outlines the detailed experimental setup used in this paper, including the dataset descriptions (Table 1), complete instruction prompts (Table 4), and full-pipeline skeleton scripts (§B.1) for experiments.

B.1 Skeleton Code for LLM-based NAS

The following listings show the skeleton codes using for experiments in §4. The scripts include the entire pipeline from data loading to model conversion and quantization. Only the modeling parts are blank for the LLM to fill in.

B.2 Prompt for Zero-Shot LLM Baselines

Prompt for Zero-Shot LLM Baselines

```
You are a helpful intelligent assistant. Now, please help solve the following time-series {} task by building a Tensorflow/
    Keras model.
[Task for '{}' dataset]
{}
[{}.py] ```python
{}
```
Start the python code with "```python". Focus only on completing the get_model() function while returning the remaining parts
    of the script exactly as provided.
Ensure the code is complete, error-free, and ready to run without requiring additional modifications.
Note that we only need the actual complete python code without textual explanations.
```

### B.2.1 Code for Classification Task

### Skeleton Code for Classification

```python
# import utility packages
import os, sys, gc, warnings, logging, shutil
import json, time, glob, math

# determine GPU number
os.environ["CUDA_DEVICE_ORDER"] = "PCI_BUS_ID"
os.environ["CUDA_VISIBLE_DEVICES"] = "0"
os.environ["TF_FORCE_GPU_ALLOW_GROWTH"] = "true"
os.environ["TF_CPP_MIN_LOG_LEVEL"] = "2" # hide INFO and WARNING messages

# define paths to model files
MODELS_DIR = "models/"
MODEL_TF = MODELS_DIR + "model.pb"
MODEL_NO_QUANT_TFLITE = MODELS_DIR + "model_no_quant.tflite"
MODEL_TFLITE_MICRO = MODELS_DIR + "model.cc"
SEED = 7

os.makedirs(MODELS_DIR, exist_ok=True)

logging.disable(logging.WARNING)
logging.disable(logging.INFO)
warnings.filterwarnings("ignore")

# import basic libraries
import random

import tensorflow as tf
import pandas as pd
import numpy as np

from tensorflow import keras

# Set a "seed" value, so we get the same random numbers each time we run this notebook for reproducible results.
random.seed(SEED)
np.random.seed(SEED)
tf.random.set_seed(SEED)

from utils.data_loader import load_dataset
from utils.data_desc import AVAILABLE_DATASETS, CLS_DATASETS, REG_DATASETS
from utils import quantize_model, brief_profile_model

# Do not change this
from sklearn.metrics import accuracy_score

N_EPOCHS = 100
BATCH_SIZE = 32
task = "classification"

keras.backend.clear_session()

data_name = os.path.basename(__file__).split(".")[0] # or replace with the user given dataset name

# 1. Loading the Target Dataset
X_train, y_train, X_test, y_test, class_names = load_dataset(data_name, task)
print("Experiment on:", data_name, X_train.shape)
seq_length = X_train.shape[1]
n_features = X_train.shape[2]
n_classes = len(class_names) # Number of output classes


# 2. Design the Model
def get_model():
```

```
    # TODO: Define a Tensorflow/Keras compatible model based on the given configurations
    # Note that your model will be converted to a TFLite Micro model
    return your_model


model = get_model()
model.compile(
    optimizer="adam", loss="sparse_categorical_crossentropy", metrics=["accuracy"]
)
es = keras.callbacks.EarlyStopping(monitor="val_accuracy", mode="max", patience=10, restore_best_weights=True)

# 3. Train the Model
model.fit(X_train, y_train, epochs=N_EPOCHS, batch_size=BATCH_SIZE, validation_split=0.1, callbacks=[es])

# 4. Evaluate the Model and Save Results (Do not change this)
y_pred = model.predict(X_test)
y_pred = y_pred.argmax(1)
acc = accuracy_score(y_test, y_pred)

# 5. Convert model to TFLite model
quantized_model = quantize_model(model, X_train)
# Save the model to disk
MODEL_TFLITE = MODELS_DIR + f"{model.name}_{task}_{data_name}.tflite"
open(MODEL_TFLITE, "wb").write(quantized_model)

# 6. Profile the converted model with a simulator
print(model.name, data_name)
print(acc)
brief_profile_model(MODEL_TFLITE)

del model
keras.backend.clear_session()
gc.collect()
```

B.2.2 Code for Regression Task

Skeleton Code for Regression

```
# import utility packages
import os, sys, gc, warnings, logging, shutil
import json, time, glob, math

# determine GPU number
os.environ["CUDA_DEVICE_ORDER"] = "PCI_BUS_ID"
os.environ["CUDA_VISIBLE_DEVICES"] = "0"
os.environ["TF_FORCE_GPU_ALLOW_GROWTH"] = "true"
os.environ["TF_CPP_MIN_LOG_LEVEL"] = "2" # hide INFO and WARNING messages

# define paths to model files
MODELS_DIR = "models/"
MODEL_TF = MODELS_DIR + "model.pb"
MODEL_NO_QUANT_TFLITE = MODELS_DIR + "model_no_quant.tflite"
MODEL_TFLITE_MICRO = MODELS_DIR + "model.cc"
SEED = 7

os.makedirs(MODELS_DIR, exist_ok=True)

logging.disable(logging.WARNING)
logging.disable(logging.INFO)
warnings.filterwarnings("ignore")

# import basic libraries
import random

import tensorflow as tf
import pandas as pd
import numpy as np

from tensorflow import keras

# Set a "seed" value, so we get the same random numbers each time we run this notebook for reproducible results.
random.seed(SEED)
np.random.seed(SEED)
tf.random.set_seed(SEED)

from utils.data_loader import load_dataset
from utils.data_desc import AVAILABLE_DATASETS, CLS_DATASETS, REG_DATASETS
from utils import quantize_model, brief_profile_model

# Do not change this
from sklearn.metrics import root_mean_squared_error

N_EPOCHS = 100
```

```
BATCH_SIZE = 32
task = "regression"

keras.backend.clear_session()

data_name = os.path.basename(__file__).split(".")[0] # or replace with the user given dataset name

# 1. Loading the Target Dataset
X_train, y_train, X_test, y_test = load_dataset(data_name, task)
print("Experiment on:", data_name, X_train.shape)
seq_length = X_train.shape[1]
n_features = X_train.shape[2]


# 2. Design the Model
def get_model():
    # TODO: Define a Tensorflow/Keras compatible model based on the given configurations
    # Note that your model will be converted to a TFLite Micro model
    return your_model


model = get_model()
model.compile(optimizer="adam", loss="mean_squared_error", metrics=keras.metrics.RootMeanSquaredError(name="rmse", dtype=None)
    )
es = keras.callbacks.EarlyStopping(monitor="val_rmse", mode="min", patience=10, restore_best_weights=True)

# 3. Train the Model
model.fit(X_train, y_train, epochs=N_EPOCHS, batch_size=BATCH_SIZE, validation_split=0.1, callbacks=[es])

# 4. Evaluate the Model and Save Results (Do not change this)
y_pred = model.predict(X_test)
rmse = root_mean_squared_error(y_test, y_pred)

# 5. Convert model to TFLite model
quantized_model = quantize_model(model, X_train)
# Save the model to disk
MODEL_TFLITE = MODELS_DIR + f"{model.name}_{task}_{data_name}.tflite"
open(MODEL_TFLITE, "wb").write(quantized_model)

# 6. Profile the converted model with a simulator
print(model.name, data_name)
print(rmse)
brief_profile_model(MODEL_TFLITE)

del model
keras.backend.clear_session()
gc.collect()
```

C Full Prompts for *MONAQ*

C.1 Multi-Objective Query Rewriting

Prompt for Multi-Objective Query Rewriting

```
Please carefully analyze the user's task descriptions based on your understanding of the following input:
[User Input Prompt]
{user_prompt}

After fully understanding the task descriptions and constraints, extract and organize the information in the specified format
    below.
Please respond as the following JSON object and make sure your JSON object is in a valid form.
```json
{
"task_description": "", // Clearly describe the user's requirements and the problem they are addressing
"data_aspects": {
        "name": "", // Dataset name, if provided
        "description": "", // Complete description of the dataset
        "features": "", // Details on features, properties, and characteristics of the dataset to consider for model building
        "context": "", // Relevant contextual information about the dataset
        "patterns": "" // Observed patterns in the dataset to consider for model building, based on any provided numerical data
                or images
    },
"model_aspects": {
    "name": "", // Suggested model name for the task, if provided by the user
    "hardware_specs": {
            "device_name": "", // Device name, if specified by the user, or inferred from hardware specifications
            "ram": "", // Maximum RAM available (in bytes), which affects the model's MAC/FLOPs limit (you can also infer it
                from the device name)
            "flash": "", // Maximum FLASH storage (in bytes), which affects model size and parameter count (you can also infer
                it from the device name)
        },
```

Task	Dataset	Instruction Prompt
Time-Series Classification	BinaryHeartbeat	I need a model to classify heartbeat signals, intended for deployment on an edge device with 1 MB of storage and 128 KB of RAM. Since this is a critical healthcare task, the model must be highly accurate while maintaining a very low inference latency of under 100 ms.
	AtrialFibrillation	I have a dataset of ECG records and want to build a classification model to categorize ECG signals into three types of atrial fibrillation. The model should be deployable on wearable devices, such as Fitbit trackers.
	Cricket	I want a model that can classify cricket umpire signals based on 3-axis accelerometer data from both hands. Since this model needs to run in real-time on a device during competitions, it should be as compact as possible while maintaining acceptable accuracy.
	FaultDetection (A)	We have a time series dataset collected from an electromechanical drive system. Create a model for deployment on edge devices to identify types of damage in rolling bearings.
	UCI-HAR	I have 3-axis body linear acceleration signals collected for human activity recognition. I need a classifier that can run on wearable devices with 1 MB of RAM and 2 MB of flash storage. The inference latency should not exceed 500 ms.
	P12	I want a model to predict patient mortality, which is a binary classification task, based on irregularly sampled sensor observations and clinical data. The model should be small enough for deployment on a smart watch.
	P19	We have clinical data and want to predict whether sepsis will occur within the next 6 hours. The dataset includes irregularly sampled sensors, consisting of vital signs and laboratory values for each patient. The model should be small enough for deployment on a smart watch.
	PAMAP2	Let's build a model to classify the physical activities of human subjects wearing three inertial measurement units. The classifier should run on wearable devices with 1 MB of RAM and 2 MB of flash storage.
Time-Series Regression	AppliancesEnergy	I have an IoT device collecting appliance energy data from a house. Please develop a predictive model to forecast the total energy consumption in kWh for the house. Additionally, the model should be compact enough to be deployed on a ZigBee wireless sensor network.
	LiveFuelMoistureContent	Build a regression model to predict the moisture content in vegetation. The model should be deployable on a small device with 512 KB of RAM and 1 MB of storage. As this will be used in a smart farming context, the prediction speed should be under 1000 ms.
	BenzeneConcentration	We aim to develop a model to predict benzene concentrations in an Italian city based on air quality measurements. This model will be deployed on IoT sensors using the Arduino Nano 33 BLE, so it should be compact and achieve a very low error rate, ideally with an RMSE of 1.00 or lower.
	BIDMC32SpO2	Our company has a project to deploy a predictive model on wearable devices, such as fitness trackers, to estimate blood oxygen saturation levels using PPG and ECG data. Please create a lightweight model suitable for deployment on these devices. The model should use no more than 32KB of RAM and be no larger than 64KB in size.
	FloodModeling	I have an IoT sensor monitoring rainfall events. Could you develop a model to predict the maximum water depth for flood modeling? The model should be lightweight enough to run on the sensor and provide real-time predictions.
	HouseholdPowerConsumption1	We have a project to predict total active power consumption in a household. Can you develop an accurate model that can be deployed on a smart home device to predict total active power consumption?
	HouseholdPowerConsumption2	We have a project to predict total reactive power consumption in a household. Can you develop an accurate model that can be deployed on a smart home device to predict total reactive power consumption?

Table 4: User instructions (i.e., task description) for experiments.

```
        "MAC": "", // Maximum MAC (Multiply-Accumulate) operations (or FLOPs) allowed for model compatibility with hardware
            constraints
        "parameters": "", // Maximum model parameters, in line with hardware constraints
        "latency": "", // Desired inference latency in milliseconds (ms), or the maximum latency allowed based on hardware
            limitations
        "performance": "" // Expected model performance, such as accuracy for classification or RMSE for regression; specify
            any target metric values to consider for model building
    }
}
```

C.2 Agent Specifications

C.2.1 Manager Agent Prompt

> **System Message for Manager Agent**
>
> ```
> You are an experienced senior project manager overseeing on-device time series analysis for resource-constrained devices. Your
>     primary responsibilities are as follows:
> 1. Receive requirements and inquiries from users regarding their task descriptions and potential target devices for deployment
>     .
> 2. Extract and clarify user requirements from both data and modeling perspectives, organizing these requirements and task-
>     specific constraints in an easy-to-understand format to enable other team members to execute subsequent processes based
>     on the information you have gathered.
> 3. Verify the suggested model whether it meets the user requirements and constraints.
> ```

C.2.2 Design Agent Prompt

> **System Message for Design Agent**
>
> ```
> You are the world's best data scientist of an on-device time series analysis for resource-constrained devices. You have the
>     following main responsibilities to complete.
> 1. Analyze user instructions and requirements.
> 2. Based on the requirements, design a neural network search space for resource-constrained devices.
> ```

C.2.3 Search Agent Prompt

> **System Message for Search Agent**
>
> ```
> You are the world's best machine learning research engineer specializing in on-device time series analysis for resource-
>     constrained devices. Your main responsibilities are as follows:
> 1. Analyze user instructions and requirements.
> 2. Understand the specified search space and constraints.
> 3. Based on your understanding, design optimal TensorFlow/Keras model configurations within the given constraints.
> ```

C.2.4 Evaluation Agent Prompt

> **System Message for Evaluation Agent**
>
> ```
> You are the world's best machine learning research engineer specializing in on-device time series analysis for resource-
>     constrained devices. Your main responsibilities are as follows:
> 1. Analyze user instructions and requirements.
> 2. Understand the specified model and constraints.
> 3. Based on your understanding, evaluate and measure the performance of TensorFlow/Keras model configurations under the given
>     constraints.
> ```

C.2.5 Code Agent Prompt

> **System Message for Code Agent**
>
> ```
> You are the world's best machine learning engineer specializing in on-device time series analysis for resource-constrained
>     devices. You have the following main responsibilities to complete.
> 1. Write accurate Python codes to build model in get_model() function based on the given instruction.
> 2. Run the model evaluation using the given Python functions and summarize the results for validation against the user's
>     requirements.
> ```

D Examples of Intermediate Results and Interpretability Analysis

To enhance interpretability, our *MONAQ* not only generates resource-aware architectures but also provides a structured breakdown of design decisions through its intermediate outputs. These outputs allow practitioners to trace how specific constraints and dataset features influence architectural choices. In this section, we present examples of intermediate results from the main steps of our framework, including multimodal query generation (i.e., query rewriting) and multi-agent LLM responses.

D.1 Classification Tasks

The following shows the intermediate results from the UCI-HAR dataset on the classification task.

Example Results of UCI-HAR Dataset

User Input: I have 3-axis body linear acceleration signals collected for human activity recognition. I need a classifier that can run on wearable devices with 1 MB of RAM and 2 MB of flash storage. The inference latency should not exceed 500 ms.

Rewriting Result:

```
{
'task_description': 'The user needs a classifier for human activity recognition using 3-axis body linear acceleration signals.
    The model must run on wearable devices with limited resources: 1 MB of RAM, 2 MB of flash storage, and an inference
    latency not exceeding 500 ms.',

'data_aspects': {'name': 'UCR HAR Dataset',
  'description': 'The dataset contains 3-axis body linear acceleration signals for human activity recognition, with six
      balanced classes: walking, walking upstairs, walking downstairs, sitting, standing, and laying down.',
  'features': 'The dataset includes three features: body accelerometer X-axis, Y-axis, and Z-axis. It is preprocessed to focus
      on linear acceleration.',
  'context': 'Data is collected from 30 healthy volunteers aged 19-48 years. The original dataset has 10299 instances, split
      into training (70%) and testing (30%) sets, with separate subjects in each set.',
  'patterns': 'The time series plots show distinct patterns for each activity class, with variations in acceleration values
      across the X, Y, and Z axes.'},

'model_aspects': {'name': 'HAR Classifier',
  'hardware_specs': {'device_name': 'Wearable Device',
   'ram': '1048576',
   'flash': '2097152'},
  'MAC': 'Limited by RAM and flash constraints, exact value depends on model architecture.',
  'parameters': 'Limited by RAM and flash constraints, exact value depends on model architecture.',
  'latency': '500',
  'performance': 'The model should achieve high accuracy in classifying the six activities while adhering to hardware
      constraints.'}
}
```

Design Result:

```
To design a search space for a neural network suitable for human activity recognition on resource-constrained wearable devices
    , we need to consider lightweight architectures that can be efficiently converted to TFLite. Here's a proposed search
    space:
```

```python
{
    "layer_type": ["Conv1D", "DepthwiseConv1D", "SeparableConv1D", "LSTM", "Dense"],
    "Conv1D_kernel_size": [3, 5],
    "Conv1D_filters": [8, 16],
    "DepthwiseConv1D_kernel_size": [3, 5],
    "SeparableConv1D_kernel_size": [3, 5],
    "SeparableConv1D_filters": [8, 16],
    "LSTM_units": [16, 32],
    "Dense_units": [32, 64],
    "activation": ["relu", "tanh"],
    "dropout_rate": [0.0, 0.2],
    "pooling_type": ["max", "average"],
    "pool_size": [2, 3],
    "strides": [1, 2],
    "batch_normalization": [True, False]
}
```

```
### Explanation:

- **Layer Types**:
  - **Conv1D**: Efficient for extracting features from time-series data.
  - **DepthwiseConv1D**: Reduces computation by applying a single convolutional filter per input channel.
  - **SeparableConv1D**: Combines depthwise and pointwise convolutions for efficiency.
  - **LSTM**: Captures temporal dependencies in time-series data.
  - **Dense**: Used for final classification layers.

- **Kernel Sizes and Filters**:
  - Smaller kernel sizes (3, 5) and filters (8, 16) are chosen to reduce memory usage and computational cost.

- **LSTM Units**:
  - Limited to 16 and 32 to ensure the model fits within RAM constraints.

- **Dense Units**:
  - Limited to 32 and 64 to balance complexity and resource usage.
```

- **Activation Functions**:
  - **ReLU** and **tanh** are efficient and commonly used in neural networks.

- **Dropout Rate**:
  - Provides regularization to prevent overfitting, with options for no dropout and a small rate (0.2).

- **Pooling Types and Sizes**:
  - **Max** and **average pooling** help reduce dimensionality and computation.
  - Pool sizes of 2 and 3 are typical for downsampling.

- **Strides**:
  - Strides of 1 and 2 allow for control over the downsampling rate.

- **Batch Normalization**:
  - Optional to stabilize training and potentially improve convergence.

This search space is designed to explore a variety of architectures while adhering to the constraints of the wearable device. Each configuration can be implemented using TensorFlow/Keras and converted to TFLite for deployment.

Search Result:

["### Analysis of the Proposed Model\n\n#### Model Characteristics\n\n1. **Computational Complexity**:\n - **Conv1D Layer**: \n - Parameters: \\(16 \\times (5 \\times 3 + 1) = 256\\)\n - FLOPs: Approximately 256 per time step.\n - **DepthwiseConv1D Layer**:\n - Parameters: \\(3\\) (since it's depthwise, no additional filters).\n - FLOPs: Minimal, as it only applies a single filter per input channel.\n - **SeparableConv1D Layer**:\n - Parameters: \\(16 \\times (3 \\times 3 + 1) + 16 \\times 16 = 304\\)\n - FLOPs: Approximately 304 per time step.\n - **LSTM Layer**:\n - Parameters: \\(4 \\times (16 \\times (16 + 1) + 16 \\times 16) = 2112\\)\n - FLOPs: High, due to recurrent connections.\n - **Dense Layers**:\n - First Dense: \\(32 \\times (16 + 1) = 544\\)\n - Output Dense: \\(6 \\times (32 + 1) = 198\\)\n\n2. **Memory Usage**:\n - Total Parameters: Approximately 3417.\n - Assuming 4 bytes per parameter, the memory usage is about 13.7 KB.\n\n3. **Inference Latency**:\n - The model is designed to achieve inference latency under 500 ms. Given the lightweight architecture, this is feasible on most modern wearable devices.\n\n4. **Model Size**:\n - With quantization (e.g., 8-bit), the model size can be reduced significantly, fitting well within the 2 MB flash constraint.\n\n#### Expected Performance\n\n1. **Accuracy**:\n - Based on similar architectures and datasets, an accuracy of around 85-90% is expected for human activity recognition.\n\n2. **Complexity Metrics**:\n - **FLOPs**: Estimated at around 3-5 MFLOPs per inference.\n - **Training Time**: Depends on the dataset size and hardware but should be manageable on a standard GPU.\n - **Inference Speed**: Should be well within the 500 ms constraint on a typical wearable device processor.\n\n#### Considerations\n\n- **TFLite Compatibility**: The model uses layers compatible with TensorFlow Lite, ensuring smooth deployment on resource-constrained devices.\n- **Optimization**: Further optimizations such as pruning or additional quantization can be applied to reduce size and latency further.\n\n### Conclusion\n\nThe proposed model configuration is well-suited for the task of human activity recognition on wearable devices, balancing performance with resource constraints effectively. It is expected to deliver high accuracy while adhering to the specified hardware limitations.",
 '### Model Analysis\n\n#### Computational Complexity and Memory Usage\n\n1. **Conv1D Layer**:\n - Parameters: \\(3 \\times 8 + 8 = 32\\)\n - FLOPs: \\(32 \\times \\text{input length}\\)\n\n2. **SeparableConv1D Layer**:\n - Depthwise: \\(3 \\times 16 = 48\\)\n - Pointwise: \\(16 \\times 16 = 256\\)\n - Total Parameters: \\(48 + 256 = 304\\)\n - FLOPs: \\(304 \\times \\text{input length}\\)\n\n3. **LSTM Layer**:\n - Parameters: \\(4 \\times ((16 + 1) \\times 16 + 16^2) = 2112\\)\n - FLOPs: Proportional to the sequence length and hidden units.\n\n4. **Dense Layer**:\n - Parameters: \\(16 \\times 32 + 32 = 544\\)\n - FLOPs: \\(544\\)\n\n5. **Final Dense Layer**:\n - Parameters: \\(32 \\times 6 + 6 = 198\\)\n - FLOPs: \\(198\\)\n\n6. **Total Parameters**: \\(32 + 304 + 2112 + 544 + 198 = 3190\\)\n\n7. **Model Size**: Assuming 4 bytes per parameter, the total size is approximately \\(3190 \\times 4 = 12,760\\) bytes (~12.5 KB).\n\n#### Inference Latency\n\n- The model is designed to run within a 500 ms constraint. Given the lightweight architecture and efficient operations, it is expected to meet this requirement on modern wearable devices.\n\n#### Expected Performance\n\n- **Accuracy**: Based on similar architectures and datasets, the model could achieve around 85-90% accuracy on the UCR HAR dataset.\n- **Inference Speed**: With the given constraints, inference should be well within 500 ms.\n\n### Conclusion\n\nThe proposed model is well-suited for deployment on resource-constrained wearable devices due to its lightweight architecture. It balances computational efficiency with the ability to capture temporal patterns necessary for human activity recognition. The estimated model size, parameter count, and FLOPs suggest it will fit within the RAM and flash storage limits while maintaining acceptable inference latency and accuracy.',
 "### Model Analysis\n\n#### Computational Complexity\n\n1. **Conv1D Layer**:\n - Filters: 16, Kernel Size: 3\n - Input Shape: Assuming input shape is (200, 3) for each axis.\n - Parameters: \\((3 \\times 3 + 1) \\times 16 = 160 \\)\n - FLOPs: Approximately \\(200 \\times 16 \\times 3 = 9600 \\)\n\n2. **DepthwiseConv1D Layer**:\n - Kernel Size: 3\n - Parameters: \\(3 \\times 1 = 3 \\) (since it's depthwise)\n - FLOPs: Approximately \\(100 \\times 3 = 300 \\)\n\n3. **LSTM Layer**:\n - Units: 32\n - Parameters: \\(4 \\times ((32 + 1) \\times 32 + 32^2) = 8448 \\)\n - FLOPs: Approximately \\(4 \\times 32 \\times (32 + 32) = 8192 \\)\n\n4. **Dense Layers**:\n - First Dense: \\(32 \\times 32 + 32 = 1056 \\) parameters\n - Output Dense: \\(32 \\times 6 + 6 = 198 \\) parameters\n - FLOPs: Approximately \\(32 \\times 32 + 32 \\times 6 = 1152 \\)\n\n#### Memory Usage\n\n- **Total Parameters**: \\(160 + 3 + 8448 + 1056 + 198 = 9865 \\)\n- **Memory for Parameters**: Approximately 39 KB (assuming 4 bytes per parameter)\n- **Fits within 1 MB RAM**: Yes\n\n#### Inference Latency\n\n- **Estimated Latency**: The model should run comfortably under 500 ms on a typical wearable device processor, given its lightweight nature.\n\n#### Model Size\n\n- **Model Size**: Approximately 39 KB, well within the 2 MB flash storage limit.\n\n### Expected Performance\n\n- **Accuracy**: Based on similar architectures, expect around 90-95% accuracy on the UCR HAR dataset.\n- **Inference Speed**: Expected to be under 500 ms per inference.\n- **Training Time**: On a typical machine, training might take a few hours depending on the hardware.\n\n### Conclusion\n\nThe proposed model is well-suited for the resource-constrained environment of a wearable device. It balances computational complexity, memory usage, and inference latency while maintaining high classification accuracy for human activity recognition.",
 '### Model Analysis\n\n#### Computational Complexity\n\n- **Conv1D Layer**: \n - Parameters: \\(3 \\times 8 = 24\\) per filter, total \\(24 \\times 8 = 192\\).\n- **DepthwiseConv1D Layer**: \n - Parameters: \\(3 \\times 1 = 3\\) per input channel.\n- **SeparableConv1D Layer**: \n - Depthwise: \\(3 \\times 1 = 3\\) per input channel.\n - Pointwise: \\(1 \\times 16 = 16\\) per filter, total \\(16 \\times 16 = 256\\).\n- **LSTM Layer**: \n - Parameters: \\(4 \\times (32 \\times (32 + 1) + 32) = 4 \\times 1056 = 4224\\).\n- **Dense Layers**: \n - First Dense: \\(32 \\times 64 = 2048\\).\n - Output Dense: \\(64 \\times 6 = 384\\).\n\n#### Total Parameters\n\n- Total parameters: \\(192 + 3 \\times 3 + 3 \\times 16 + 256 + 4224 + 2048 + 384 = 7112\\).\n\n#### Memory Usage\n\n- **RAM**: \n - Assuming float32 precision, each parameter requires 4 bytes.\n - Total memory for parameters: \\(7112 \\times 4 \\approx 28.5\\) KB.\n - Additional memory for activations and intermediate computations will be required, but should fit within the 1 MB RAM constraint.\n- **Flash**

Storage**: \n - Model size: \\(28.5\\) KB, well within the 2 MB constraint.\n\n#### Inference Latency\n- Given the lightweight architecture and efficient operations, inference latency is expected to be well under 500 ms on typical wearable device hardware.\n\n### Performance Estimation\n\n#### Accuracy\n- Based on similar architectures and datasets, expected accuracy is around 90-95% for human activity recognition tasks.\n\n#### Complexity Metrics\n- **FLOPs (Floating Point Operations)**: \n - Estimated to be low due to small filter sizes and efficient layer choices.\n- ** Training Time**: \n - On a standard desktop GPU, training should take a few hours for convergence.\n- **Inference Speed **: \n - Expected to be fast due to the small model size and efficient architecture.\n\n### Conclusion\nThe proposed model configuration is well-suited for the given constraints of wearable devices. It balances complexity and performance, ensuring efficient operation within the hardware limits while maintaining high classification accuracy for human activity recognition.',

'### Model Analysis\n\n#### Computational Complexity\n- **Conv1D Layer**: \n - Filters: 16, Kernel size: 5\n - Parameters: \\((5 \\times 1 + 1) \\times 16 = 96 \\)\n \n- **SeparableConv1D Layer**:\n - Filters: 16, Kernel size: 3\n - Depthwise parameters: \\(3 \\times 1 = 3 \\)\n - Pointwise parameters: \\(16 \\times 1 = 16 \\)\n - Total: \\(3 + 16 = 19 \\)\n\n- **DepthwiseConv1D Layer**:\n - Kernel size: 3\n - Parameters: \\(3 \\times 1 = 3 \\)\n\n- **LSTM Layer **:\n - Units: 16\n - Parameters: \\(4 \\times 16 \\times (16 + 1) = 1088 \\)\n\n- **Dense Layer**:\n - Units: 64\n - Parameters: \\(16 \\times 64 + 64 = 1088 \\)\n\n- **Output Layer**:\n - Units: 6\n - Parameters: \\(64 \\times 6 + 6 = 390 \\)\n\n#### Total Parameters\n- Total parameters: \\(96 + 19 + 3 + 1088 + 1088 + 390 = 2684 \\)\n\n#### Memory Usage\n- **RAM**: \n - Model parameters: ~10.5 KB (assuming 4 bytes per parameter)\n - Intermediate activations and batch normalization add additional memory usage, but the total should remain well under 1 MB.\n\n- **Flash Storage**:\n - Model size: ~10.5 KB, easily fitting within 2 MB.\n\n#### Inference Latency\n- The model is designed to be lightweight, with an expected inference latency well under 500 ms on typical wearable device processors.\n\n#### Performance Metrics\n- **Accuracy**: \n - Based on similar models and datasets, an accuracy of around 85-90% is expected for the six-class classification task.\n\n- **FLOPs**:\n - Estimated to be low due to the use of lightweight convolutional layers and a small LSTM.\n\n- **Inference Speed**:\n - Expected to be fast, meeting the 500 ms constraint .\n\n### Conclusion\nThe proposed model configuration is well-suited for the given constraints of wearable devices. It balances computational complexity, memory usage, and inference latency while maintaining high classification accuracy. The use of efficient layers like SeparableConv1D and DepthwiseConv1D ensures that the model remains lightweight and resource-efficient.']

Evaluation Result:

Based on the user requirements and constraints, the best model configuration for human activity recognition on resource-constrained wearable devices is **Model Configuration #3**. This model balances computational complexity, memory usage, and inference latency while maintaining high classification accuracy. It is well-suited for the given constraints of 1 MB RAM, 2 MB flash storage, and an inference latency not exceeding 500 ms.

Selected Model Configuration

Model Configuration

```python
model_configuration = {
    "layer_sequence": [
        {"layer_type": "Conv1D", "filters": 16, "kernel_size": 3, "activation": "relu", "strides": 1, "batch_normalization":
            True},
        {"pooling_type": "max", "pool_size": 2},
        {"layer_type": "DepthwiseConv1D", "kernel_size": 3, "activation": "relu", "strides": 1, "batch_normalization": True},
        {"pooling_type": "average", "pool_size": 2},
        {"layer_type": "LSTM", "units": 32, "activation": "tanh", "dropout_rate": 0.2},
        {"layer_type": "Dense", "units": 32, "activation": "relu"},
        {"layer_type": "Dense", "units": 6, "activation": "softmax"}
    ]
}
```

Explanation

- **Conv1D Layer**:
  - **Filters**: 16 for sufficient representation capacity.
  - **Kernel Size**: 3 for efficient feature extraction.
  - **Activation**: ReLU for non-linear representation.
  - **Batch Normalization**: True for stable training.
  - **Strides**: 1 to preserve spatial resolution.

- **Pooling (MaxPooling)**:
  - Reduces the dimensionality while maintaining important features.
  - **Pool Size**: 2 for effective down-sampling.

- **Middle Layer (DepthwiseConv1D)**:
  - **Kernel Size**: 3 and ReLU activation for efficient computation.

- **Pooling (AveragePooling)**:
  - Further reduces features while averaging important patterns.
  - **Pool Size**: 2.

- **Recurrent Layer (LSTM)**:
  - **Units**: 32 to capture temporal patterns with sufficient granularity.
  - **Activation**: Tanh is suitable for recurrent layers.
  - **Dropout Rate**: 0.2 for regularization and preventing overfitting.

- **Dense Layer**:
  - **Units**: 32 with ReLU for non-linear mapping.

- **Output Layer (Dense)**:

- **Units**: 6 to match the number of classes, with softmax activation for classification.

Considerations

- **Resource Constraints**: This configuration is designed to fit within RAM and flash storage limits.
- **Latency**: The architecture focuses on being lightweight for quick inference within 500 ms.
- **Conversion to TFLite**: All operations used are supported by TensorFlow Lite.

This model configuration is expected to deliver high accuracy while adhering to the specified hardware limitations, making it
      ideal for deployment on wearable devices for human activity recognition.

Code Result:

```python
# import utility packages
import os, sys, gc, warnings, logging, shutil
import json, time, glob, math

# determine GPU number
os.environ["CUDA_DEVICE_ORDER"] = "PCI_BUS_ID"
os.environ["CUDA_VISIBLE_DEVICES"] = "0"
os.environ["TF_FORCE_GPU_ALLOW_GROWTH"] = "true"
os.environ["TF_CPP_MIN_LOG_LEVEL"] = "2" # hide INFO and WARNING messages

# define paths to model files
MODELS_DIR = "models/"
MODEL_TF = MODELS_DIR + "model.pb"
MODEL_NO_QUANT_TFLITE = MODELS_DIR + "model_no_quant.tflite"
MODEL_TFLITE_MICRO = MODELS_DIR + "model.cc"
SEED = 7

os.makedirs(MODELS_DIR, exist_ok=True)

logging.disable(logging.WARNING)
logging.disable(logging.INFO)
warnings.filterwarnings("ignore")

# import basic libraries
import random

import tensorflow as tf
import pandas as pd
import numpy as np

from tensorflow import keras

# Set a "seed" value, so we get the same random numbers each time we run this notebook for reproducible results.
random.seed(SEED)
np.random.seed(SEED)
tf.random.set_seed(SEED)

from utils.data_loader import load_dataset
from utils.data_desc import AVAILABLE_DATASETS, CLS_DATASETS, REG_DATASETS
from utils import quantize_model, brief_profile_model

# Do not change this
from sklearn.metrics import accuracy_score

N_EPOCHS = 100
BATCH_SIZE = 32
task = "classification"

keras.backend.clear_session()

data_name = os.path.basename(__file__).split(".")[0] # or replace with the user given dataset name

# 1. Loading the Target Dataset
X_train, y_train, X_test, y_test, class_names = load_dataset(data_name, task)
print("Experiment on:", data_name, X_train.shape)
seq_length = X_train.shape[1]
n_features = X_train.shape[2]
n_classes = len(class_names) # Number of output classes


# 2. Design the Model
def get_model():
    model = keras.Sequential([
        keras.layers.Conv1D(filters=16, kernel_size=3, activation='relu', strides=1,
                            input_shape=(seq_length, n_features)),
        keras.layers.BatchNormalization(),
        keras.layers.MaxPooling1D(pool_size=2),
        keras.layers.DepthwiseConv1D(kernel_size=3, activation='relu', strides=1),
        keras.layers.BatchNormalization(),
        keras.layers.AveragePooling1D(pool_size=2),
        keras.layers.LSTM(units=32, activation='tanh', dropout=0.2, return_sequences=False),
```

```
        keras.layers.Dense(units=32, activation='relu'),
        keras.layers.Dense(units=n_classes, activation='softmax')
    ])
    return model


model = get_model()
model.compile(
    optimizer="adam", loss="sparse_categorical_crossentropy", metrics=["accuracy"]
)
es = keras.callbacks.EarlyStopping(monitor="val_accuracy", mode="max", patience=10, restore_best_weights=True)

# 3. Train the Model
model.fit(X_train, y_train, epochs=N_EPOCHS, batch_size=BATCH_SIZE, validation_split=0.1, callbacks=[es])

# 4. Evaluate the Model and Save Results (Do not change this)
y_pred = model.predict(X_test)
y_pred = y_pred.argmax(1)
acc = accuracy_score(y_test, y_pred)

# 5. Convert model to TFLite model
quantized_model = quantize_model(model, X_train)
# Save the model to disk
MODEL_TFLITE = MODELS_DIR + f"{model.name}_{task}_{data_name}.tflite"
open(MODEL_TFLITE, "wb").write(quantized_model)

# 6. Profile the converted model with a simulator
print(model.name, data_name)
print(acc)
brief_profile_model(MODEL_TFLITE)

del model
keras.backend.clear_session()
gc.collect()
```

Potential Interpretation   In the UCI-HAR case study, the system interprets the user's hardware and latency constraints as signals to prioritize lightweight layers such as Conv1D, SeparableConv1D, and small LSTM units. The generated search space and model configuration are interpretable due to their modularity and domain alignment: each layer type is associated with a specific functionality (e.g., LSTM layers to capture temporal patterns in time-series signals). Moreover, the explicit computational analysis produced during architecture evaluation (e.g., parameter count, FLOPs, and memory usage) provides actionable insights into trade-offs made during model construction. These details empower users to make informed decisions about model simplification, hardware adaptation, and runtime behavior, thereby promoting transparent model deployment in safety-critical settings.

D.2   Regression Tasks

The following shows the intermediate results from the BIDMC32SpO2 dataset on the regression task.

Example Results of BIDMC32SpO2 Dataset

User Input: Our company has a project to deploy a predictive model on wearable devices, such as fitness trackers, to estimate blood oxygen saturation levels using PPG and ECG data. Please create a lightweight model suitable for deployment on these devices. The model should use no more than 32KB of RAM and be no larger than 64KB in size.

- -

Rewriting Result:

```
{
'task_description': 'Develop a lightweight predictive model for wearable devices like fitness trackers to estimate blood
    oxygen saturation levels using PPG and ECG data. The model must be optimized for low resource usage, specifically using
    no more than 32KB of RAM and 64KB of storage.',

 'data_aspects': {'name': 'BIDMC Blood Oxygen Saturation Dataset',
  'description': "The dataset aims to estimate blood oxygen saturation levels using PPG and ECG data. It contains 7949 time
        series from the Physionet's BIDMC PPG and Respiration dataset, extracted from the MIMIC II waveform database.",
  'features': 'The dataset includes PPG and ECG data sampled at 125 Hz over 32-second windows. The label is the SpO2 level,
        sampled at 1 Hz.',
  'context': 'The dataset is used to estimate blood oxygen saturation levels, crucial for health monitoring in wearable
        devices.',
  'patterns': 'The time series plots show consistent patterns across different SpO2 ranges, with slight variations in
        amplitude and frequency in PPG and ECG signals.'},
```

```
  'model_aspects': {'name': '',
  'hardware_specs': {'device_name': 'Wearable fitness tracker',
   'ram': '32768',
   'flash': '65536'},
  'MAC': '',
  'parameters': '',
  'latency': '',
  'performance': ''}
}
```

Design Result:

Based on the requirements and constraints for developing a lightweight predictive model for wearable devices, here is a proposed search space for neural network design using TensorFlow/Keras:

```python
{
    "layer_type": ["Conv1D", "DepthwiseConv1D", "LSTM", "Dense"],
    "Conv1D_kernel_size": [1, 3],
    "Conv1D_filters": [4, 8],
    "DepthwiseConv1D_kernel_size": [1, 3],
    "LSTM_units": [4, 8],
    "Dense_units": [4, 8],
    "activation": ["relu", "tanh"],
    "optimizer": ["adam", "rmsprop"],
    "batch_size": [1, 2],
    "dropout_rate": [0.0, 0.1],
    "learning_rate": [0.001, 0.01]
}
```

Explanation:

1. **Layer Types**:
   - **Conv1D** and **DepthwiseConv1D**: Useful for extracting features from time-series data.
   - **LSTM**: Effective for capturing temporal dependencies in time-series data.
   - **Dense**: Used for final output layers or simple transformations.

2. **Kernel Sizes and Filters/Units**:
   - Smaller kernel sizes and fewer filters/units are chosen to minimize resource usage while still capturing essential patterns.

3. **Activation Functions**:
   - **ReLU** and **Tanh** are efficient and commonly used in lightweight models.

4. **Optimizers**:
   - **Adam** and **RMSprop** are popular choices for efficient training.

5. **Batch Size**:
   - Small batch sizes help reduce memory usage.

6. **Dropout Rate**:
   - Low dropout rates to maintain model simplicity and reduce overfitting.

7. **Learning Rate**:
   - Standard learning rates for effective training convergence.

This search space is designed to ensure that the models can be executed within the given constraints of 32KB RAM and 64KB storage, while still providing flexibility to find a high-performing model for the task.

Search Result:

["### Model Analysis\n\n#### Computational Complexity and Memory Usage\n\n1. **Conv1D Layer**:\n - **Parameters**: \\(\\text{filters} \\times (\\text{input channels} \\times \\text{kernel size} + 1) = 4 \\times (2 \\times 3 + 1) = 28 \\)\n - **Output Shape**: Depends on input length, but generally reduces by kernel size - 1.\n\n2. **LSTM Layer**:\n - **Parameters**: \\(4 \\times ((\\text{input size} + \\text{units} + 1) \\times \\text{units} = 4 \\times (4 + 4 + 1) \\times 4 = 160 \\)\n\n3. **Dense Layer**:\n - **Parameters**: \\(\\text{input units} \\times \\text{output units} + \\text{output units} = 4 \\times 4 + 4 = 20 \\)\n\n4. **Output Layer**:\n - **Parameters**: \\(4 \\times 1 + 1 = 5 \\)\n\n5. **Total Parameters**: \\(28 + 160 + 20 + 5 = 213 \\)\n\n#### Memory and Storage Constraints\n\n- **RAM Usage**: The model's parameters (213) and activations must fit within 32KB. With minimal parameters and batch size of 1, this is feasible.\n- **Storage**: 213 parameters with 32-bit precision require approximately 852 bytes, well within the 64KB limit.\n\n#### Inference Latency\n\n- **FLOPs**: \n - Conv1D: \\(\\approx \\text{input length} \\times \\text{filters} \\times \\text{kernel size} \\)\n - LSTM: \\(\\approx 4 \\times \\text{units}^2 + 4 \\times \\text{input size} \\times \\text{units} \\)\n - Dense: \\(\\approx \\text{input units} \\times \\text{output units} \\)\n\n- **Inference Speed**: With small batch size and lightweight architecture, inference should be quick, suitable for real-time applications on wearables.\n\n### Expected Performance\n\n- **RMSE**: For time-series regression on SpO2 levels, an RMSE of around 2-3% is reasonable given the model's simplicity and constraints.\n- **Training Time**: Minimal due to small dataset size and model complexity, likely a few minutes on a standard CPU.\n- **Inference Speed**: Fast enough for real-time applications, likely within milliseconds per sample.\n\n### Conclusion\n\nThe proposed model is well-suited for the constraints of wearable devices, balancing performance and resource usage effectively. It should provide adequate accuracy for SpO2 estimation while maintaining low computational and memory demands.",
 "### Model Analysis\n\n#### Computational Complexity\n- **Conv1D Layer**:\n - Parameters: \\(8 \\times (3 \\times \\text{input channels} + 1)\\)\n - FLOPs: \\(8 \\times ((\\text{input length} - 3 + 1) \\times 3\\)\n\n- **DepthwiseConv1D

Layer**:\n - Parameters: \\(\\\text{input channels} \\times 3\\)\n - FLOPs: \\(\\\text{input channels} \\times (\\text{input length} - 3 + 1) \\times 3\\)\n\n- **LSTM Layer**:\n - Parameters: \\(4 \\times (4 + \\text{input channels} + 1) \\)\\)\n - FLOPs: \\(4 \\times (\\text{input length} \\times \\text{input channels} \\times 4)\\)\n\n- **Dense Layer**:\n - Parameters: \\(4 \\times (\\text{input channels} + 1)\\)\n - FLOPs: \\(4 \\times \\text{input channels}\\)\n\n#### Memory Usage\n- **Total Parameters**: Sum of parameters from all layers.\n- **Model Size**: Total parameters \\(\\times \\) 4 bytes (for float32).\n\n#### Inference Latency\n- **Expected Latency**: Depends on the number of FLOPs and the device's processing capability.\n\n### Performance Estimation\n\n#### Quantitative Regression Performance\n- **Expected RMSE**: Typically ranges between 1-2% for SpO2 estimation, depending on the model's training and validation.\n\n#### Complexity Metrics\n- **Number of Parameters**: Estimated based on the above calculations.\n- **FLOPs**: Total FLOPs from all layers.\n- **Model Size**: Should be within the 64KB storage constraint.\n- **Training Time**: Depends on dataset size and computational resources.\n- **Inference Speed**: Should be fast enough for real-time applications on wearables.\n\n### Conclusion\nThe proposed model configuration is designed to be lightweight, adhering to the constraints of 32KB RAM and 64KB storage. The use of Conv1D and DepthwiseConv1D layers helps capture spatial features efficiently, while the LSTM layer captures temporal dependencies. The Dense layer ensures the model's output is suitable for regression tasks.\n\nOverall, this configuration should provide a good balance between performance and efficiency, making it suitable for deployment on wearable devices for SpO2 estimation.",

"### Model Analysis\n\n#### Computational Complexity\n- **Conv1D Layer**: \n - Filters: 4, Kernel Size: 3\n - Parameters: \\(3 \\times \\text{input channels} \\times 4 + 4\\)\n - FLOPs: Minimal due to small filter size and number of filters.\n\n- **DepthwiseConv1D Layer**:\n - Kernel Size: 3\n - Parameters: \\(3 \\times \\text{input channels}\\)\n - FLOPs: Reduced due to depthwise operation.\n\n- **LSTM Layer**:\n - Units: 4\n - Parameters: \\(4 \\times (\\text{input} + \\text{units} + 1) \\times \\text{units}\\)\n - FLOPs: Higher than Conv1D but manageable with 4 units.\n\n- **Dense Layer**:\n - Units: 4\n - Parameters: \\(4 \\times (\\text{input size} + 1)\\)\n - FLOPs: Minimal due to small number of units.\n\n#### Memory Usage\n- **RAM**: \n - The model is designed to fit within 32KB RAM. The small batch size and minimal layer sizes help achieve this.\n\n- **Storage**:\n - The total number of parameters should fit within 64KB storage. Given the small number of layers and units, this is feasible.\n\n#### Inference Latency\n- The model's simplicity ensures low latency, suitable for real-time applications on wearable devices.\n\n### Expected Performance\n\n#### Quantitative Regression Performance\n- **RMSE**: Expected to be moderate due to the small model size. Exact values depend on data variability and model tuning.\n \n#### Complexity Metrics\n- **Number of Parameters**: Estimated to be under 1000, given the small layer sizes.\n- **FLOPs**: Low, due to the use of lightweight operations.\n- **Model Size**: Expected to be under 64KB.\n- **Training Time**: Fast, due to the small model size and batch size.\n- **Inference Speed**: Real-time capable on constrained devices.\n\n### Conclusion\nThe proposed model is well-suited for deployment on resource-constrained wearable devices. It balances simplicity and performance, fitting within the specified RAM and storage limits while providing adequate regression performance for estimating blood oxygen saturation levels. Further tuning and validation on the specific dataset will be necessary to optimize performance metrics like RMSE.",

'### Model Analysis\n\n#### Computational Complexity\n1. **Conv1D Layer**:\n - Filters: 4, Kernel Size: 3\n - Input size: Assuming 125 Hz sampling over 32 seconds, the input length is 4000.\n - Parameters: \\(4 \\times (3 \\times \\text{input channels})\\)\n - FLOPs: Approximately \\(4 \\times 3 \\times 4000\\)\n\n2. **DepthwiseConv1D Layer**:\n - Kernel Size: 3\n - Parameters: \\(3 \\times \\text{input channels}\\)\n - FLOPs: Approximately \\(3 \\times 4000\\)\n\n3. **LSTM Layer**:\n - Units: 4\n - Parameters: \\(4 \\times (4 + 1 + \\text{input size})\\)\n - FLOPs: Higher due to recurrent operations, approximately \\(8 \\times \\text{input size} \\times 4\\)\n\n4. **Dense Layer**:\n - Units: 4\n - Parameters: \\(4 \\times (\\text{input size} + 1)\\)\n - FLOPs: Approximately \\(4 \\times \\text{input size}\\)\n\n#### Memory Usage\n- **Parameters**: Total parameters from all layers.\n- **RAM Usage**: Includes parameters and intermediate activations. Estimated to be within 32KB.\n- **Storage**: Model size should be within 64KB, considering quantization techniques if necessary.\n\n#### Inference Latency\n- **Batch Size**: 1 for real-time processing.\n- **Expected Latency**: Low due to small model size and batch processing.\n\n### Performance Estimation\n\n#### Regression Performance\n- **Expected RMSE**: Based on similar models, RMSE could be around 2-3% for SpO2 estimation.\n\n#### Complexity Metrics\n- **Number of Parameters**: Estimated to be a few hundred, given the small network size.\n- **FLOPs**: Estimated to be in the low thousands, ensuring fast computation.\n- **Model Size**: Likely under 64KB with potential quantization.\n- **Training Time**: Minimal due to small dataset size and model complexity.\n- **Inference Speed**: Fast, suitable for real-time applications on wearables.\n\n### Conclusion\nThe proposed model configuration is well-suited for the constraints of wearable devices, balancing computational efficiency and predictive performance. It should fit within the specified RAM and storage limits while providing accurate SpO2 level predictions.',

"To evaluate the proposed model for time-series regression on a wearable device, let's analyze its characteristics, computational complexity, memory usage, and expected performance.\n\n### Model Characteristics\n\n1. **Layer Configuration**:\n - **Conv1D Layer**: \n - Filters: 4\n - Kernel Size: 3\n - Activation: ReLU\n - **LSTM Layer**: \n - Units: 4\n - Activation: Tanh\n - **Dense Layer**: \n - Units: 4\n - Activation: Tanh\n\n2. **Optimizer**:\n - Adam with a learning rate of 0.001\n\n3. **Training Configuration**:\n - Batch Size: 1\n - Dropout Rate: 0.1\n\n#### Computational Complexity\n\n1. **Number of Parameters**:\n - **Conv1D Layer**: \n - Parameters = (Input Channels * Kernel Size + 1) * Filters\n - **LSTM Layer**: \n - Parameters = 4 * (Input Size + LSTM Units + 1) * LSTM Units\n - **Dense Layer**: \n - Parameters = (Input Size + 1) * Units\n\n2. **FLOPs (Floating Point Operations)**:\n - Conv1D and Dense layers are relatively simple, while LSTM is more complex due to recurrent connections.\n\n3. **Model Size**:\n - The model needs to fit within 64KB storage, which is feasible given the small number of parameters.\n\n### Memory Usage \n\n- The model must use no more than 32KB RAM. Given the small batch size and layer sizes, this is achievable.\n\n### Inference Latency\n\n- With a batch size of 1 and lightweight layers, inference latency should be low, suitable for real-time applications on wearable devices.\n\n### Expected Performance\n\n1. **RMSE (Root Mean Square Error)**: \n - Expected to be low due to the model's ability to capture temporal dependencies and extract features efficiently.\n - Exact RMSE would depend on the specific dataset and training conditions but should be competitive given the model's design.\n\n2. **Training Time**:\n - Should be relatively short due to the small model size and batch size.\n\n3. **Inference Speed**:\n - Fast enough for real-time applications, given the model's simplicity.\n\n### Conclusion\n\nThe proposed model is well-suited for the constraints of a wearable device, balancing computational efficiency and performance. It is expected to perform well in estimating blood oxygen saturation levels using PPG and ECG data, with low memory usage and fast inference speed. The model's design allows it to fit within the specified RAM and storage limits, making it an excellent choice for deployment on resource-constrained devices."]

Evaluation Result:

Based on the user requirements and constraints, the best model configuration to meet the needs of developing a lightweight predictive model for wearable devices is **Model Configuration #1**. This model is designed to fit within the specified RAM and storage limits while providing adequate performance for estimating blood oxygen saturation levels using PPG and ECG data. Here is the complete configuration for the selected model:

Selected Model Configuration:

- **Layers**:
  1. **Conv1D Layer**:
     - Filters: 4
     - Kernel Size: 3
     - Activation: ReLU
  2. **LSTM Layer**:
     - Units: 4
     - Dropout Rate: 0.1
  3. **Dense Layer**:
     - Units: 4
     - Activation: ReLU
  4. **Output Layer**:
     - Units: 1 (for regression output)
     - Activation: Linear

- **Optimizer**:
  - Type: Adam
  - Learning Rate: 0.001

- **Batch Size**:
  - Size: 1

- **Memory and Storage Constraints**:
  - RAM Usage: Must fit within 32KB
  - Storage: Must fit within 64KB

- **Expected Performance**:
  - RMSE: Around 2-3% for SpO2 estimation
  - Inference Speed: Fast enough for real-time applications on wearables

This configuration provides a balance between computational efficiency and predictive performance, making it suitable for deployment on resource-constrained wearable devices.

Code Result:

```
# import utility packages
import os, sys, gc, warnings, logging, shutil
import json, time, glob, math

# determine GPU number
os.environ["CUDA_DEVICE_ORDER"] = "PCI_BUS_ID"
os.environ["CUDA_VISIBLE_DEVICES"] = "0"
os.environ["TF_FORCE_GPU_ALLOW_GROWTH"] = "true"
os.environ["TF_CPP_MIN_LOG_LEVEL"] = "2" # hide INFO and WARNING messages

# define paths to model files
MODELS_DIR = "models/"
MODEL_TF = MODELS_DIR + "model.pb"
MODEL_NO_QUANT_TFLITE = MODELS_DIR + "model_no_quant.tflite"
MODEL_TFLITE_MICRO = MODELS_DIR + "model.cc"
SEED = 7

os.makedirs(MODELS_DIR, exist_ok=True)

logging.disable(logging.WARNING)
logging.disable(logging.INFO)
warnings.filterwarnings("ignore")

# import basic libraries
import random

import tensorflow as tf
import pandas as pd
import numpy as np

from tensorflow import keras

# Set a "seed" value, so we get the same random numbers each time we run this notebook for reproducible results.
random.seed(SEED)
np.random.seed(SEED)
tf.random.set_seed(SEED)

from utils.data_loader import load_dataset
from utils.data_desc import AVAILABLE_DATASETS, CLS_DATASETS, REG_DATASETS
from utils import quantize_model, brief_profile_model

# Do not change this
from sklearn.metrics import root_mean_squared_error

N_EPOCHS = 100
BATCH_SIZE = 32
task = "regression"

keras.backend.clear_session()
```

```
data_name = os.path.basename(__file__).split(".")[0] # or replace with the user given dataset name

# 1. Loading the Target Dataset
X_train, y_train, X_test, y_test = load_dataset(data_name, task)
print("Experiment on:", data_name, X_train.shape)
seq_length = X_train.shape[1]
n_features = X_train.shape[2]


# 2. Design the Model
def get_model():
    model = keras.Sequential([
        keras.layers.Conv1D(filters=4, kernel_size=3, activation='relu', input_shape=(seq_length, n_features)),
        keras.layers.LSTM(units=4, dropout=0.1, return_sequences=False),
        keras.layers.Dense(units=4, activation='relu'),
        keras.layers.Dense(units=1, activation='linear')
    ])
    return model


model = get_model()
model.compile(optimizer=keras.optimizers.Adam(learning_rate=0.001), loss="mean_squared_error", metrics=keras.metrics.
    RootMeanSquaredError(name="rmse", dtype=None))
es = keras.callbacks.EarlyStopping(monitor="val_rmse", mode="min", patience=10, restore_best_weights=True)

# 3. Train the Model
model.fit(X_train, y_train, epochs=N_EPOCHS, batch_size=BATCH_SIZE, validation_split=0.1, callbacks=[es])

# 4. Evaluate the Model and Save Results (Do not change this)
y_pred = model.predict(X_test)
rmse = root_mean_squared_error(y_test, y_pred)

# 5. Convert model to TFLite model
quantized_model = quantize_model(model, X_train)
# Save the model to disk
MODEL_TFLITE = MODELS_DIR + f"{model.name}_{task}_{data_name}.tflite"
open(MODEL_TFLITE, "wb").write(quantized_model)

# 6. Profile the converted model with a simulator
print(model.name, data_name)
print(rmse)
brief_profile_model(MODEL_TFLITE)

del model
keras.backend.clear_session()
gc.collect()
```

Potential Interpretation   In the case study on the BIDMC32SpO2 dataset, the framework translates high-level user constraints, such as hardware limitations and physiological signal types, into explicit model design decisions (e.g., use of Conv1D and LSTM layers). First, the intermediate outputs from the multimodal query generation and model design stages reveal the alignment between user requirements and architectural choices. For instance, the use of LSTM layers is justified based on the temporal nature of physiological data, while lightweight convolutional layers are selected for edge deployment efficiency. These design decisions are accompanied by detailed computational and memory analysis, enabling users to audit trade-offs between performance and deployment feasibility. Second, the generated search space itself is interpretable. Each dimension (e.g., kernel size, activation, and optimizer) directly corresponds to meaningful architectural decisions, making the space semantically rich. The choices are not arbitrary; they are grounded in hardware specifications, dataset characteristics, and task constraints, facilitating both expert validation and human-in-the-loop adjustments. Finally, the analysis of multiple model candidates, including FLOPs, parameter counts, RMSE estimates, and memory usage, serves as a concrete interpretability mechanism. These metrics expose how changes in layer composition affect efficiency and accuracy, enabling stakeholders—especially in sensitive domains like healthcare—to make informed decisions regarding trade-offs and model trustworthiness.

Datasets	Metrics	MLP	LSTM	CNN	TCN	D-CNN	DS-CNN	ConvLSTM	TENet(6)	Grid Search	Random Search	TinyTNAS	GPT-4o-mini	GPT-4o	*MONAQ*
AtrialFibrillation	Accuracy	0.200	0.400	0.333	0.333	0.467	0.333	0.400	0.267	0.333	0.333	0.400	0.333	0.333	0.467
	FLASH	45.120	8.824	14.024	34.640	7.192	15.600	10.864	165.296	13.136	17.976	6.808	669.600	661.416	14.976
	RAM	3.868	18.448	33.880	30.976	4.696	44.680	43.700	364.564	10.652	12.188	11.508	43.868	43.740	23.504
	MAC	41,624	24	2,051,840	1,392,664	18,240	540,160	122,648	11,755,808	55,596	25,716	42,115	2,744,512	2,717,504	62,560
	Energy	1.17E-06	3.40E-08	1.73E-03	2.92E-03	9.45E-05	9.93E-04	2.31E-04	2.06E-02	2.25E-04	3.28E-04	1.46E-04	1.41E-03	1.36E-03	2.45E-04
	Latency	1.886	0.064	128.190	244.950	8.765	84.754	31.931	1536.669	20.845	16.870	21.621	120.508	117.157	18.417
BinaryHeartbeat	Accuracy	0.659	0.727	0.732	0.449	0.732	0.732	0.732	0.732	0.732	0.727	0.732	0.732	0.732	0.732
	FLASH	597.104	8.752	84.072	34.592	15.384	85.800	10.752	170.840	12.680	7.760	11.152	9494.408	9494.408	9.360
	RAM	38.300	447.888	892.616	746.608	39.256	1189.736	1188.628	9524.084	152.352	77.384	152.480	595.404	595.404	299.528
	MAC	593,616	16	55,219,392	39,728,336	259,419	14,879,582	1,778,832	340,379,296	1,130,543	301,138	1,130,543	24,606,944	24,606,944	204,080
	Energy	3.33E-06	3.40E-08	3.43E-02	1.75E-01	1.09E-03	1.60E-02	4.15E-03	3.70E-01	5.30E-03	2.03E-03	5.30E-03	1.89E-02	1.89E-02	1.32E-02
	Latency	5.570	0.061	2106.132	10783.571	90.710	1052.569	268.145	21462.749	419.023	191.470	419.023	1191.453	1191.453	177.662
Cricket	Accuracy	0.125	0.069	0.556	0.097	0.569	0.208	0.069	0.125	0.083	0.083	0.167	0.500	0.528	0.625
	FLASH	234.088	9.192	39.824	34.872	28.192	40.768	11.352	171.784	13.056	15.080	11.520	2465.664	621.344	1229.520
	RAM	15.644	31.888	60.632	53.376	16.728	80.392	79.412	650.116	18.592	20.168	18.720	79.452	40.796	39.988
	MAC	230,560	96	4,931,424	2,757,984	118,476	1,218,434	688,544	22,015,424	115,261	65,860	115,261	6,814,528	1,876,384	1,568,608
	Energy	3.33E-06	3.40E-08	3.88E-03	4.25E-03	4.05E-04	2.41E-03	9.76E-04	6.44E-02	3.76E-04	4.38E-04	3.76E-04	3.18E-03	9.83E-04	9.33E-04
	Latency	5.631	0.121	270.929	373.199	32.402	176.782	70.096	4310.810	37.793	29.876	37.793	228.040	85.385	63.480
FaultDetectionA	Accuracy	0.713	0.622	0.978	0.746	0.602	0.966	0.804	0.999	0.983	0.990	0.989	0.989	0.985	1.000
	FLASH	168.000	8.968	40.680	34.608	9.960	42.416	10.768	170.984	17.120	18.592	11.176	10500.352	8193.600	20.680
	RAM	11.548	126.096	248.920	210.176	12.376	331.400	330.420	2658.244	56.432	47.176	45.216	330.588	330.460	167.948
	MAC	164,504	24	15,267,840	10,977,304	72,960	4,121,600	491,480	94,044,352	521,465	499,382	312,624	26,706,304	24,381,036	2,211,680
	Energy	3.33E-06	3.40E-08	2.05E-02	2.83E-02	4.24E-04	1.07E-02	3.50E-03	2.42E-01	1.83E-03	1.58E-03	1.37E-03	1.62E-02	1.62E-02	8.49E-03
	Latency	5.574	0.064	1299.218	1857.094	34.409	708.650	227.471	14647.130	168.597	147.271	132.681	1048.406	1047.739	358.068
UCI-HAR	Accuracy	0.781	0.348	0.819	0.813	0.627	0.822	0.348	0.904	0.783	0.805	0.858	0.814	0.817	0.908
	FLASH	23.968	9.128	12.872	34.704	7.256	14.288	10.992	171.280	18.800	18.104	25.128	223.432	110.848	14.176
	RAM	2.460	8.208	13.016	13.696	3.800	16.904	15.924	142.228	11.080	12.188	11.816	16.092	9.052	10.508
	MAC	20,464	48	707,792	454,896	9,261	183,110	58,928	3,796,960	23,700	9,583	160,134	901,440	285,760	53,936
	Energy	6.24E-07	3.40E-08	3.24E-04	1.10E-03	3.77E-05	3.35E-04	8.99E-05	5.49E-03	2.45E-04	2.87E-04	1.85E-04	4.59E-04	2.21E-04	2.73E-04
	Latency	1.151	0.086	49.271	81.811	4.513	30.229	16.282	494.159	11.759	11.021	22.373	57.644	26.284	18.675
Average	Accuracy	0.496	0.433	0.684	0.488	0.599	0.612	0.471	0.605	0.583	0.588	0.629	0.674	0.679	**0.746**
	FLASH	213.656	8.973	38.294	34.683	13.597	39.774	10.946	170.037	14.958	15.502	13.157	4670.691	3816.323	257.742
	RAM	14.364	126.506	249.813	210.966	15.371	332.622	331.617	2667.847	49.822	33.821	47.948	213.081	203.890	108.295
	MAC	210,153	41	15,635,657	11,062,236	95671	4,188,577	628,086	94,398,368	369,313	180,335	352,135	12,354,745	10,773,525	820,172
	Energy	2.35E-06	3.40E-08	1.22E-02	4.23E-02	4.10E-04	6.09E-03	1.79E-03	1.40E-01	1.59E-03	9.32E-04	1.47E-03	8.04E-03	7.54E-03	4.62E-03
	Latency	3.963	0.079	770.748	2668.125	34.160	410.597	122.785	8490.304	131.603	79.302	126.698	529.210	493.604	127.260

Table 5: Full experimental results on time-series **classification** tasks comparing downstream task accuracy and model complexity metrics.

Datasets	Metrics	MLP	LSTM	CNN	TCN	D-CNN	DS-CNN	ConvLSTM	TENet(6)	Grid Search	Random Search	TinyTNAS	GPT-4o-mini	GPT-4o	*MONAQ*
AppliancesEnergy	RMSE	3.610	9.345	4.104	9.606	3.541	3.720	10.532	6.573	3.682	3.624	3.756	3.537	4.026	3.607
	FLASH	114.376	10.192	14.824	27.536	8.088	11.936	12.648	166.736	20.880	36.168	13.056	44.864	226.064	8.112
	RAM	8.136	8.892	11.268	12.336	9.860	13.112	11.872	110.528	12.968	16.092	11.216	11.400	8.076	8.088
	MAC	111,240	8	1,170,720	414,728	47,520	244,512	327,304	2,647,328	78,357	48,206	32,234	468,032	223,264	544
	Energy	3.31E-06	1.75E-08	5.44E-04	1.07E-03	1.98E-04	2.99E-04	2.52E-04	4.26E-03	2.42E-04	3.60E-04	2.04E-04	2.91E-04	5.05E-06	9.03E-05
	Latency	5.521	0.011	45.537	146.521	16.722	27.735	20.712	361.481	15.459	16.729	11.063	27.002	8.383	0.070
BenzeneConcentration	RMSE	2.884	6.087	2.739	46.666	2.181	2.231	8.107	3.226	3.365	5.268	3.453	4.022	11.858	1.847
	FLASH	65.224	9.168	11.432	24.448	6.120	11.056	11.112	170.376	9.088	9.912	43.600	40.592	127.760	8.816
	RAM	5.064	8.892	14.596	15.548	6.148	19.128	18.016	159.552	7.288	7.296	20.048	26.616	5.004	10.480
	MAC	62,088	8	1,091,040	552,968	26,400	257,760	182,920	4,409,120	34,826	51,153	573,765	1,056	124,960	91,464
	Energy	1.22E-06	1.75E-08	6.48E-04	9.16E-04	9.04E-05	4.03E-04	1.97E-04	6.62E-03	1.61E-04	1.57E-04	4.51E-04	3.96E-08	1.25E-03	1.13E-04
	Latency	2.171	0.011	55.659	90.251	11.711	38.473	19.111	571.345	12.188	13.935	52.972	0.090	5.382	15.559
BIDMC32SpO2	RMSE	16.682	4.808	5.884	5.156	4.773	5.092	4.789	4.879	4.974	4.961	5.716	5.200	5.649	4.670
	FLASH	259.784	8.784	17.608	26.832	7.048	18.176	10.536	164.664	32.704	157.784	10.944	134.088	133.680	8.624
	RAM	17.224	99.132	195.076	165.420	18.180	259.764	258.656	2084.752	71.036	78.972	36.172	130.184	66.184	66.984
	MAC	256,648	8	12,808,000	8,704,008	110,000	3,360,000	767,752	73,472,224	1,460,740	2,662,864	272,266	2,048,016	1,088,008	88,100
	Energy	3.31E-06	1.75E-08	1.55E-02	2.39E-02	4.85E-04	9.40E-03	3.26E-03	2.05E-01	2.67E-03	3.21E-03	1.06E-03	3.42E-03	1.87E-03	3.36E-04
	Latency	5.521	0.011	992.415	1579.982	40.737	628.213	209.449	12286.802	240.532	288.318	104.746	231.939	136.535	82.258
FloodModeling	RMSE	0.117	0.023	0.019	1.494	0.020	0.014	0.020	0.018	0.007	0.019	0.008	0.032	0.047	0.019
	FLASH	12.296	8.720	9.912	26.800	5.008	10.632	10.440	169.704	16.584	91.840	17.728	41.464	20.720	14.672
	RAM	1.992	9.532	15.876	16.044	3.716	20.788	19.680	172.864	9.500	19.496	11.340	19.336	10.092	19.736
	MAC	9,160	8	792,144	570,312	3,657	213,062	25,480	4,906,112	41,554	117,224	113,919	263,648	29,576	25,776
	Energy	2.73E-07	1.75E-08	4.04E-04	1.20E-03	4.80E-05	4.56E-04	7.95E-05	8.20E-03	2.15E-04	3.72E-04	1.88E-04	1.94E-04	1.14E-04	3.99E-05
	Latency	0.530	0.011	57.769	26.458	4.326	37.563	18.619	657.498	14.081	23.218	20.308	31.807	16.405	17.243
LiveFuelMoistureContent	RMSE	43.157	53.006	49.777	231.972	42.301	42.527	41.723	42.176	42.715	43.567	40.803	45.470	47.836	39.369
	FLASH	85.544	8.960	11.456	26.672	6.176	11.104	10.872	170.344	11.528	10.744	23.160	190.352	168.400	12.688
	RAM	6.344	11.836	20.612	20.012	7.428	27.188	25.952	223.936	9.416	10.996	16.980	13.960	6.284	15.424
	MAC	82,408	8	1,577,336	852,648	35,119	377,685	244,072	6,732,992	46,273	100,859	309,140	579,248	165,600	61,120
	Energy	3.17E-06	1.75E-08	8.68E-04	1.66E-03	6.68E-05	5.94E-04	2.61E-04	1.23E-02	2.13E-04	1.91E-04	2.79E-04	3.89E-04	4.35E-06	9.51E-05
	Latency	5.103	0.011	75.181	134.888	14.228	57.358	23.025	968.620	15.924	22.647	40.198	43.769	7.191	8.516
Average	RMSE	13.290	14.654	12.505	58.979	10.563	10.717	13.034	11.374	10.948	11.488	10.747	11.652	13.883	**9.902**
	FLASH	107.445	9.165	13.046	26.458	6.488	12.581	11.122	168.365	18.157	61.290	21.698	90.272	135.325	10.582
	RAM	7.752	27.657	51.486	45.872	9.066	67.996	66.835	550.326	22.042	26.570	19.151	40.299	19.128	24.142
	MAC	104,308	8	3,487,848	2,218,932	44,539	890,603	309,505	18,433,555	332,350	596,061	260,264	672,000	326,281	53,400
	Energy	2.26E-06	1.75E-08	3.60E-03	5.75E-03	1.78E-04	2.23E-03	8.10E-04	4.73E-02	7.00E-04	8.58E-04	4.36E-04	8.58E-04	6.50E-04	1.35E-04
	Latency	3.769	0.011	245.312	395.620	17.545	157.868	58.183	2969.149	59.637	72.970	45.857	66.921	34.779	24.729

Table 6: Full experimental results on time-series **regression** tasks comparing downstream task error (RMSE) and model complexity metrics.

Datasets	Metrics	MLP	LSTM	CNN	TCN	D-CNN	DS-CNN	ConvLSTM	TENet(6)	Grid Search	Random Search	TinyTNAS	GPT-4o-mini	GPT-4o	*MONAQ*
P12	Accuracy	0.850	0.859	0.855	0.854	0.862	0.852	0.860	0.860	0.862	0.861	0.862	0.859	0.853	0.862
	FLASH	251.824	11.192	18.648	35.712	13.400	14.880	14.112	174.304	24.392	15.872	12.376	36.096	118.328	15.520
	RAM	16.796	14.480	18.136	20.868	18.904	19.724	18.356	147.348	22.732	21.708	20.260	18.012	18.428	18.228
	MAC	248,336	16	2,326,096	701,776	108,252	466,020	736,272	3,967,936	110,421	44,118	65,877	554,464	634,432	368,592
	Energy	3.33E-06	3.40E-08	1.01E-03	1.17E-03	3.98E-04	5.56E-04	5.27E-04	6.07E-03	4.08E-04	4.05E-04	3.23E-04	3.14E-04	4.57E-04	2.00E-04
	Latency	5.570	0.061	76.829	88.827	32.468	49.898	37.155	529.373	24.170	19.607	19.162	33.114	43.708	19.024
P19	Accuracy	0.974	0.973	0.974	0.974	0.974	0.974	0.975	0.975	0.975	0.975	0.974	0.973	0.974	0.976
	FLASH	69.424	11.064	17.592	35.648	10.416	14.144	13.920	174.112	13.488	9.552	15.552	24.608	40.376	9.888
	RAM	5.404	6.288	7.384	9.728	7.512	8.328	7.220	67.476	8.988	7.884	9.124	6.876	6.620	7.264
	MAC	65,936	16	622,320	192,016	28,560	125,400	189,456	1,112,832	25,270	16,583	34,760	244,224	174,848	81,680
	Energy	1.23E-06	3.40E-08	3.97E-04	1.03E-03	4.57E-05	1.44E-04	1.12E-04	1.54E-03	1.69E-04	1.31E-04	1.69E-04	1.51E-04	7.77E-05	8.71E-05
	Latency	2.167	0.061	32.520	26.039	10.691	15.555	10.739	162.068	8.169	6.564	8.700	13.116	9.440	5.633
PAMAP2	Accuracy	0.201	0.521	0.739	0.149	0.839	0.831	0.248	0.901	0.895	0.104	0.752	0.672	0.789	0.912
	FLASH	330.616	10.048	23.168	35.176	28.128	22.384	12.360	172.504	24.096	57.192	11.848	312.360	1245.584	14.528
	RAM	21.660	23.056	32.344	34.176	23.256	42.248	41.140	67.476	27.464	31.688	24.864	22.876	41.308	23.316
	MAC	327,104	64	3,945,600	1,593,664	158,100	863,400	976,128	11,022,688	204,073	174,016	104,199	1,247,776	4,052,224	488,480
	Energy	3.33E-06	3.40E-08	2.00E-03	2.64E-03	6.25E-04	1.35E-03	6.48E-04	1.96E-02	4.82E-04	5.91E-04	3.95E-04	6.46E-04	1.69E-03	3.20E-04
	Latency	5.614	0.104	140.920	235.823	47.499	101.277	44.612	1489.623	37.915	37.992	29.631	59.565	133.391	25.559
Average ↑	Accuracy	0.675	0.785	0.856	0.659	0.891	0.886	0.695	0.912	0.910	0.647	0.862	0.835	0.872	**0.916**
	FLASH	217.288	10.768	19.803	35.512	17.315	17.136	13.464	173.640	20.659	27.539	13.259	124.355	468.096	13.312
	RAM	14.620	14.608	19.288	21.591	16.557	23.433	22.239	186.260	19.728	20.427	18.083	15.921	22.119	16.269
	MAC	213,792	32	2,298,005	829,152	98,304	484,940	633,952	5,367,819	113,255	78,239	68,279	682,155	1,620,501	312,917
	Energy	2.63E-06	3.40E-08	1.14E-03	1.62E-03	3.56E-04	6.83E-04	4.29E-04	9.08E-03	3.53E-04	3.76E-04	2.96E-04	3.70E-04	7.41E-04	2.03E-04
	Latency	4.450	0.075	83.423	116.896	30.219	55.576	30.835	727.021	23.418	21.388	19.164	35.265	62.180	16.739
HouseholdPowerConsumption1	RMSE	154.118	1587.395	163.577	419.708	321.482	159.706	1424.762	920.171	157.645	148.238	156.421	528.786	156.914	152.468
	FLASH	234.184	8.976	13.160	26.888	7.136	13.256	10.824	170.088	18.128	11.984	11.048	39.232	2968.512	14.240
	RAM	15.688	37.692	72.196	63.020	16.644	95.924	94.816	773.936	25.756	19.232	18.636	141.816	187.528	24.212
	MAC	231,048	8	5,578,560	3,271,688	99,000	1,378,080	690,376	26,450,432	342,766	65,290	128,426	528	21,934,272	184,064
	Energy	3.31E-06	1.75E-08	5.00E-03	4.30E-03	4.13E-04	2.64E-03	9.85E-04	7.98E-02	6.68E-04	4.14E-04	3.93E-04	3.50E-08	1.06E-02	2.55E-04
	Latency	5.521	0.011	342.462	429.557	31.991	199.602	70.924	5128.256	71.745	31.542	42.347	0.057	677.695	26.393
HouseholdPowerConsumption2	RMSE	50.930	172.072	50.086	54.395	57.065	53.406	64.535	184.172	54.538	59.565	54.566	55.732	55.729	52.349
	FLASH	234.184	8.976	13.160	26.888	7.136	13.256	10.824	164.952	16.488	55.552	11.048	39.232	39.232	381.328
	RAM	15.688	37.692	72.196	63.020	16.644	95.924	94.816	774.064	22.684	25.332	18.636	141.816	141.816	50.092
	MAC	231,048	8	5,578,560	3,271,688	99,000	1,378,080	690,376	26,450,432	274,188	282,567	128,426	528	528	2,787,856
	Energy	3.31E-06	1.75E-08	5.00E-03	4.30E-03	4.13E-04	2.64E-03	9.85E-04	7.98E-02	6.02E-04	5.98E-04	3.93E-04	3.50E-08	3.50E-08	2.85E-03
	Latency	5.521	0.011	342.462	429.557	31.991	199.602	70.924	5128.256	60.644	52.580	42.347	0.057	0.057	242.887
Average ↓	RMSE	102.524	879.733	106.832	237.051	189.274	106.556	744.649	552.171	106.092	103.902	105.494	292.259	106.322	**102.409**
	FLASH	234.184	8.976	13.160	26.888	7.136	13.256	10.824	167.520	17.308	33.768	11.048	39.232	1503.872	197.784
	RAM	15.688	37.692	72.196	63.020	16.644	95.924	94.816	774.000	24.220	22.282	18.636	141.816	164.672	37.152
	MAC	231,048	8	5,578,560	3,271,688	99,000	1,378,080	690,376	26,450,432	308,477	173,929	128,426	528	10,967,400	1,485,960
	Energy	3.31E-06	1.75E-08	5.00E-03	4.30E-03	4.13E-04	2.64E-03	9.85E-04	7.98E-02	6.35E-04	5.06E-04	3.93E-04	3.50E-08	5.30E-03	1.55E-03
	Latency	5.521	0.011	342.462	429.557	31.991	199.602	70.924	5128.256	66.194	42.061	42.347	0.057	338.876	134.640

Table 7: Full experimental results on **noisy** and **irregular time-series** classification and regression tasks, comparing downstream error (RMSE), accuracy, and model complexity metrics.

Datasets	Grid Search	Random Search	TinyTNAS	*MONAQ*
AppliancesEnergy	1,195.44	691.18	151.72	332.53
BIDMC32SpO2	489.21	605.25	156.31	178.19
BenzeneConcentration	1,156.71	718.81	424.27	324.53
FloodModeling	4,385.44	3,872.19	241.79	151.85
LiveFuelMoistureContent	347.17	1,891.60	265.96	175.51
AtrialFibrillation	1,145.92	3,249.92	215.81	165.50
BinaryHeartbeat	155.91	185.56	217.73	170.40
Cricket	128.87	153.04	176.87	300.68
FaultDetectionA	1,018.39	2,190.53	122.50	216.64
UCIHAR	3,544.73	3,573.73	240.68	279.88
Average	1,356.78	1,713.18	221.36	229.57

Table 8: Search cost (in seconds) comparison between traditional NAS methods and *MONAQ*.

E Full Experimental Results

In Tables 5 and 6, we present the full results for classification and regression tasks, respectively. The model performance metric for classification is accuracy and for regression is RMSE. The model complexity metrics extensively include FLASH, RAM, MAC, Latency, and Energy Consumption. In addition, Table 7 presents the complete results for irregular and noisy time-series datasets.

F Resource Cost Comparison

To quantify *MONAQ*'s relative efficiency, we compare the resource cost—specifically, the runtime—for the main results presented in Tables 5 and 6. The results are reported in Table 8. For grid search and random search, we set the number of rounds and the number of candidates per round to be the same as in *MONAQ*, i.e., $B = 3$ and $C = 5$. Thus, we have 15 candidates in total and select the best one based on validation performance. For TinyTNAS, we set the permissible search time to be twice the average time of *MONAQ*, i.e., 8 minutes.

According to the results, *MONAQ* consistently achieves significantly lower search costs compared to traditional NAS methods such as grid search and random search. On average, *MONAQ* takes only 229.57 seconds (including image generation in the multimodal query generation preprocessing time), whereas grid search and random search take 1,356.78 and 1,713.18 seconds, respectively. Even compared to the efficient TinyTNAS, which averages 221.36 seconds, *MONAQ* performs comparably while maintaining or improving performance, especially across various classification and regression datasets. This result highlights the practical efficiency and scalability of *MONAQ*.