Can LLMs Revolutionize the Design of Explainable and Efficient TinyML Models?

Christophe El Zeinaty[1], Wassim Hamidouche[2], Glenn Herrou[1], Daniel Menard[1] and Merouane Debbah[2]
[1]Univ. Rennes, INSA Rennes, CNRS, IETR - UMR 6164, Rennes, France.
[2]KU 6G Research Center, Khalifa University, Abu Dhabi, UAE.
Emails: firstname.lastname@insa-rennes.fr and firstname.lastname@ku.ac.ae

***Abstract*—This paper introduces a novel framework for designing efficient neural network architectures specifically tailored to tiny machine learning (TinyML) platforms. By leveraging large language models (LLMs) for neural architecture search (NAS), a vision transformer (ViT)-based knowledge distillation (KD) strategy, and an explainability module, the approach strikes an optimal balance between accuracy, computational efficiency, and memory usage. The LLM-guided search explores a hierarchical search space, refining candidate architectures through Pareto optimization based on accuracy, multiply-accumulate operations (MACs), and memory metrics. The best-performing architectures are further fine-tuned using logits-based KD with a pre-trained ViT-B/16 model, which enhances generalization without increasing model size. Evaluated on the CIFAR-100 dataset and deployed on an STM32H7 microcontroller (MCU), the three proposed models, LMaNet-Elite, LMaNet-Core, and QwNet-Core, achieve accuracy scores of 74.50%, 74.20% and 73.00%, respectively. All three models surpass current state-of-the-art (SOTA) models, such as MCUNet-in3/in4 (69.62% / 72.86%) and XiNet (72.27%), while maintaining a low computational cost of less than 100 million MACs and adhering to the stringent 320 KB static random-access memory (SRAM) constraint. These results demonstrate the efficiency and performance of the proposed framework for TinyML platforms, underscoring the potential of combining LLM-driven search, Pareto optimization, KD, and explainability to develop accurate, efficient, and interpretable models. This approach opens new possibilities in NAS, enabling the design of efficient architectures specifically suited for TinyML. To facilitate further research and development in this field, the proposed framework and the best-performing architectures are made publicly available at Link.**

***Index Terms*—TinyML, IoT, Large Language Models, Neural Architecture Search, Knowledge Distillation, Explainable AI.**

I. Introduction

The proliferation of internet of things (IoT) devices built on tiny hardware platforms, such as microcontrollers (MCUs), has underscored the demand for deploying deep learning (DL) models in resource-constrained environments [1]. Despite their potential to democratize artificial intelligence (AI), these devices face unique challenges that differ significantly from mobile DL, particularly due to stringent memory limitations. For instance, the typical MCU has a static random-access memory (SRAM) of less than 512kB, which poses a substantial obstacle for running conventional DL models [2]. Numerous lightweight deep neural network (DNN) architectures, such as MobileNet [3], have demonstrated the utility of handcrafted model design for mobile applications. Yet, when adapted for MCUs, the limited memory resources pose significant

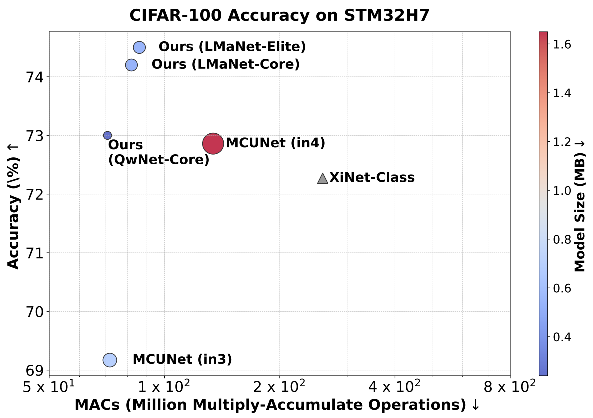

Fig. 1. Our proposed architectures balance accuracy and efficiency, reducing model size and computational cost compared to baselines. Marker size and color indicate model size, XiNet-Class is shown as a triangle due to missing size information.

hurdles. Even compact models often exceed the memory capacities of MCUs, requiring external memory or reliance on cloud-based solutions. To overcome these challenges, various optimization techniques, such as quantization, pruning, and knowledge distillation (KD), have been extensively studied to reduce the size and complexity of neural networks [4]–[6]. A landmark contribution in this domain is the deep compression framework proposed by Han *et al.* [7], which combines pruning and quantization to significantly lower both memory usage and computational requirements. Recent works, such as MCUNet [8], [9], showcased advanced optimization strategies like patch-based inference and receptive field redistribution. These approaches significantly reduce peak memory usage, enabling deployment on tiny devices, but they rely on system co-design neural architecture search (NAS) with their custom inference engine. XiNet [10], on the other hand, adopts a manual design approach coupled with hardware aware scaling (HAS) to develop efficient convolutional (Conv) architectures for tiny machine learning (TinyML). Instead of relying on traditional NAS, XiNet prioritizes energy and memory efficiency through manual fine-tuning of architectural parameters. This approach demonstrates significant improvements in performance-energy trade-offs, achieving state-of-the-art (SOTA) results on platforms like the STM32H7. However, manual design inherently lacks the scalability and automation provided by modern NAS methodologies.

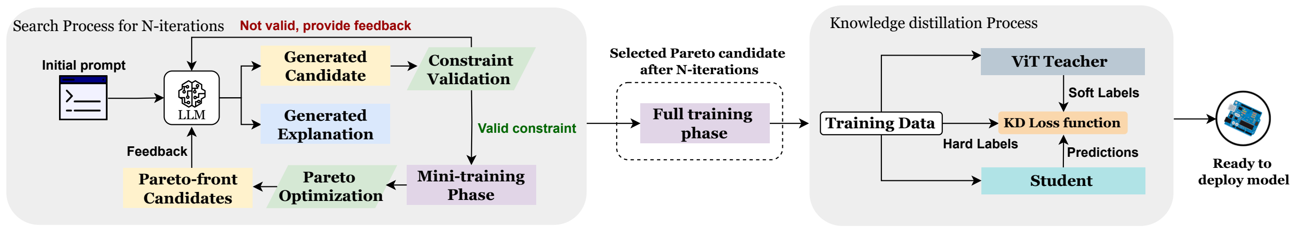

Fig. 2. Overview of the LLM-guided architecture search process with integrated explainability. The LLM generates candidate architectures, which are validated against MAC and SRAM constraints. Valid candidates undergo lightweight training, followed by Pareto optimization to evaluate trade-offs between accuracy, computational cost, and memory usage. During the process, the LLM provides explanations for its design choices, enhancing interpretability and guiding further iterations. The best candidates are fully trained and refined using ViT-based knowledge distillation to produce the final deployable model.

Recent advancements in large language models (LLMs) provide a new paradigm for architecture search [11], [12]. By leveraging their reasoning and generative capabilities, LLMs streamline the design process, offering flexible and efficient solutions. However, these prior works primarily focus on general-purpose NAS tasks without specific deployment constraints. In this work, we bridge this gap by proposing a novel LLM-guided NAS framework tailored specifically for TinyML platforms. Throughout this study, we aim to answer the research question of whether LLMs can revolutionize the design of explainable and efficient TinyML models. Our approach leverages open-source LLMs, such as Llama [13] and Qwen [14], to efficiently generate candidate architectures under stringent resource constraints. We integrate a Pareto-guided feedback loop to balance accuracy, computational cost, and memory footprint. Furthermore, our method incorporates KD [6], using pretrained vision transformer (ViT) models [15] as teacher networks to enhance the performance of student architectures. As illustrated in Fig. 1, our proposed architectures achieve SOTA accuracy on the CIFAR-100 dataset [16] while maintaining low computational costs. In addition to optimizing neural architectures, our framework explores the explanatory potential of LLMs for explainable artificial intelligence (XAI), providing insights into the decision-making process during architecture generation. This aspect not only improves interpretability but also paves the way for more transparent and trustworthy deployment solutions in resource-constrained environments. The contributions of this paper are as follows:

- A novel LLM-guided NAS framework designed specifically for TinyML, demonstrating significant reductions in search time compared to traditional methods.
- Pareto-guided optimization for balancing accuracy, computational cost, and memory usage, validated on the STM32H7 platform.
- An analysis of LLM-generated architectures, emphasizing the impact of model family and size on architectural efficiency.
- Exploration of LLM-driven explanations for XAI, enhancing transparency in architecture generation.

The remainder of this paper is as follows. Section II provides an overview of related works. Section III introduces the proposed framework, including its core components and workflow. Section IV outlines the experimental results, followed by an ablation study in Section V, and a discussion of the XAI module in Section VI. Finally, Section VII presents the conclusions and future research directions.

II. Related Works

A. Advancements in Large Language Models

The development of LLMs has significantly advanced the field of AI, enabling remarkable capabilities in tasks such as natural language processing, summarization, reasoning, and content generation [17]. Commercial models, including OpenAI's GPT series [18], Google's Gemini [19], and Anthropic's Claude [20], have demonstrated the transformative potential of LLMs across various domains, such as code generation [21], scientific discovery [22], and interactive problem-solving. Concurrently, open-weight LLMs like Llama [13], Qwen [14], Phi [23] and Falcon [24] have emerged, offering accessible alternatives for researchers. These models provide the flexibility to explore LLM applications, fostering innovation and enabling greater transparency and control in the design and deployment of advanced AI solutions. Building on these advancements, this study leverages open-weight LLM to guide NAS. Specifically, Llama3.2-3B-Instruct, Llama3.1-8B-Instruct [13] and Qwen2.5-3B-Instruct [14] models are employed to generate efficient neural architectures tailored for TinyML platforms. Unlike traditional NAS methods, which often rely on computationally intensive strategies such as reinforcement learning or evolutionary algorithms [25], [26], LLMs enable a more rapid and flexible exploration of the design space through their inherent reasoning and generative capabilities [11], [12].

B. Evolution of Neural Architecture Search (NAS)

NAS represents a significant advancement in the automation of neural network design. Initial NAS approaches employed reinforcement learning to navigate vast search spaces [25], while subsequent methods incorporated evolutionary algorithms [26] and Bayesian optimization [27] to refine the search process. However, the computational demands of these early methods were often prohibitive, leading to the development of more efficient strategies. For example, DARTS [28] introduced a gradient-based approach, significantly streamlining the search process. Further research has considerably expanded the scope of NAS, with a focus on techniques that balance computational efficiency with the generation of high-performing architectures [29]. These advancements have facilitated the transition of NAS from theoretical exploration to practical deployment across a range of applications.

C. LLMs for NAS: Current Approaches

Recent advancements in leveraging LLMs for NAS have demonstrated their potential to streamline the search process while maintaining competitive performance. GENIUS [11] and LLMatic [12] are two notable studies exploring this synergy, each offering distinct methodologies but sharing certain limitations when applied to deployment-specific scenarios. GENIUS employs a prompting mechanism with GPT-4 to iteratively refine neural architectures. The process involves querying the LLM with architecture design tasks, followed by empirical evaluation and feedback integration to guide subsequent iterations. While GENIUS significantly reduces the computational cost compared to traditional NAS methods, its primary focus remains on optimizing for broad performance metrics such as accuracy. Moreover, the absence of a multi-objective framework and explainability limits its applicability in deployment-specific scenarios, particularly for resource-constrained environments. In contrast, LLMatic utilizes CodeGen-6.1B [30], an LLM specialized in code generation, to generate and refine neural architectures. Starting with simple architectures, LLMatic applies quality-diversity (QD) optimization techniques to explore a diverse set of solutions across different metrics, including parameter count and depth. This hierarchical evolution strategy ensures a variety of architectural candidates while maintaining a low computational footprint. However, LLMatic focuses more on architectural diversity than on deployment-specific constraints, and its reliance on code-generation LLMs may limit its generalizability. Additionally, like GENIUS, it does not incorporate explainability into the architecture generation process.

These studies highlight the emerging role of LLMs in automating and accelerating NAS. However, their limitations underscore the need for approaches that can address deployment-specific requirements, such as memory and computational constraints, while also enhancing transparency and interpretability.

D. Bridging the Gap for TinyML Applications

This work addresses the challenge by demonstrating the effectiveness of LLMs in generating neural network architectures optimized for resource-constrained environments. Through an analysis of Pareto-optimal candidates, we validate the adaptability of LLMs to such scenarios, highlighting their utility in tasks beyond general-purpose NAS. By integrating explainability into the search process, our approach not only bridges the gap between general-purpose and deployment-specific requirements but also enhances trust and usability through XAI. This framework underscores the potential of LLMs to revolutionize NAS by making it faster, more efficient and interpretable, particularly for real-world deployments with stringent constraints.

III. Methodology

This section presents our novel approach to designing efficient TinyML models by integrating LLMs for NAS and ViT-based KD. As shown in Fig. 2, the goal is to optimize architecture for computational efficiency and accuracy within TinyML constraints. The method combines LLM-driven search, Pareto optimization [31], and logits-based KD with a pretrained ViT [15], while also incorporating an explainability module to provide insights into the LLM's design choices, as discussed in Section VI.

A. LLm-Guided Search Space Exploration

Search space. The architecture search process is structured around a predefined skeleton composed of N sequential stages, as shown in Fig. 3 for $N = 5$. Each stage ST_i consists of a stack of Conv Blocks $l_{i,j}$, where $j \in \{1, \ldots, L_i\}$, and all blocks $l_{i,j}$ within a stage ST_i share the same configuration. This skeleton provides a consistent framework for generating lightweight and efficient DNN architectures tailored to TinyML platforms. The value of N defines the overall depth of the network and provides a consistent framework while enabling the LLM to optimize the architecture holistically. Unlike traditional approaches that configure each stage independently, this method allows the LLM to propose the entire N stage architecture in a single step. This global search approach ensures that the relationships and dependencies over the stages are considered, leading to more cohesive and contextually optimized architectures. At the core of each stage lies a generalized operation, repeated L_i times, defined as:

$$\mathbf{x}' = \text{ConvBlock}_i(\mathbf{x}) = \underbrace{f\big(f\big(\ldots f(\mathbf{x})\ldots\big)\big)}_{L_i \text{ times}}, \qquad (1)$$

where $\mathbf{x}$ is the input tensor to the (i)-th stage, $\mathbf{x}'$ is the resulting output tensor, f represents a single Conv block, and L_i indicates the number of Conv Blocks within this stage. The function f refers to the convolutional operation block, which can be either a depthwise separable convolution (DWSepConv) block (Fig. 3(a)) or a mobile inverted bottleneck convolution (MBConv) block [3] (Fig. 3(b)), determining the computational structure of the block. To provide a more detailed analysis of the generalized formulation, each component of Eq. (1) is expanded and examined below based on its specific operation and role within the Conv Block.

In the case of a DWSepConv block, the convolution operation processes each input channel independently, significantly reducing computational cost while preserving spatial relationships. This is achieved through the depthwise convolution (DWConv) ($\text{dwconv}_{K\times K}$), followed by batch normalization (BN) and an activation function (A), such as ReLU6, Leaky ReLU, or Swish, which introduces non-linearity, enabling the model to capture complex patterns. The result is then passed through an optional SE block, denoted $SEBlock(\mathbf{x})$, which recalibrates the importance of the channel by highlighting relevant characteristics at an intermediate step of the block. The inclusion of the SE block is controlled by the binary

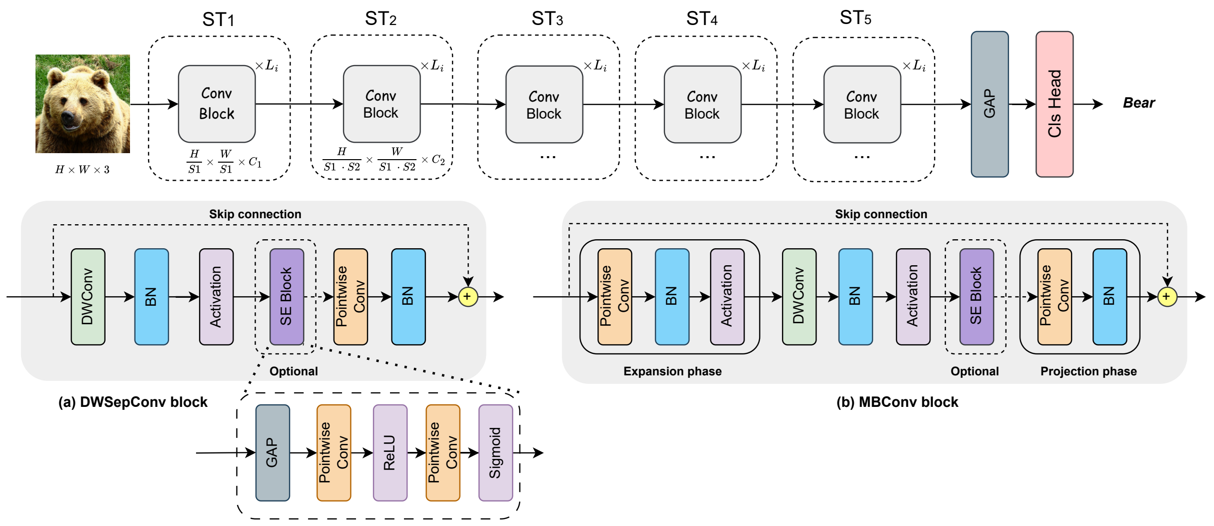

Fig. 3. The LLM defines the configuration parameters for each stage, including layers L_i, kernel size K, stride S, activation A, convolution type $Conv\ Block$, expansion E, and output channels C_{out}, along with the use of building blocks like DWSepConv and MBConv, and controls the inclusion of the SE block and skip connections.

parameter δ_{se}, where $\delta_{\text{se}} = 1$ denotes the inclusion of the SE block within the Conv Block, and $\delta_{\text{se}} = 0$ denotes its absence. Additionally, the degree of channel reduction is governed by the SE ratio, selected from $\{0.25, 0.5\}$, ensuring a balance between feature emphasis and computational efficiency. The intermediate tensor $\mathbf{z}$ (prior to the SE block) is given by:

$$\mathbf{z} = A\Big(\text{BN}(\text{dwconv}_{K\times K}(\mathbf{x}))\Big) \tag{2}$$

The optional SE block computes the output tensor $\tilde{\mathbf{z}}$ as follows:

$$\tilde{\mathbf{z}} = \delta_{\text{se}} \left[\mathbf{z} \odot \text{SEBlock}(\mathbf{z})\right] + (1 - \delta_{\text{se}})\, \mathbf{z} \tag{3}$$

where the SEBlock operations for an input tensor $\mathbf{z}$ is defined as:

$$\text{SEBlock}(\mathbf{z}) = \mathbf{z} \odot \sigma\left(\text{Conv}_{1\times1}\left(\text{ReLU}\left(\text{Conv}_{1\times1}\left(\text{GAP}(\mathbf{z})\right)\right)\right)\right) \tag{4}$$

In this expression, GAP denotes global average pooling (GAP), σ is the sigmoid activation, $\odot$ denotes element-wise multiplication, and ReLU is the activation function.

Subsequently, a pointwise convolution (PWConv) is applied, followed by batch normalization (BN). These operations adjust the number of channels and ensure the output tensor is ready for subsequent steps of the network. The resulting intermediate tensor after application of the PWConv and batch normalization is defined as:

$$\mathbf{z}_{\text{pw}} = \text{BN}\Big(\text{PWConv}(\tilde{\mathbf{z}})\Big) \tag{5}$$

In the absence of a skip connection $SC = 0$, the output simplifies to $\mathbf{x}' = \mathbf{z}_{\text{pw}}$. Conversely, when a skip connection is employed, it facilitates gradient flow and stabilizes training, particularly in deeper architectures. In this case, the resulting output, $\mathbf{x}'$, is computed as follows:

$$\mathbf{x}' = \mathbf{x} + \mathbf{z}_{\text{pw}} \tag{6}$$

where $\mathbf{x}$ is the input tensor and $\mathbf{z}_{\text{pw}}$ is the output tensor after the block's transformations.

Alternatively, when the Conv Block corresponds to a MBConv block, the convolution operation follows a structure similar to that of the DWSepConv block, but with an additional expansion phase at the beginning. This phase improves the model's expressiveness while maintaining computational efficiency. Initially, the input tensor undergoes expansion through a PWConv, which increases the number of channels by a factor E, selected from $\{3, 4, 6\}$. The expanded tensor is then passed through batch normalization BN and an activation function A, as follows:

$$\mathbf{z} = A\Big(\text{BN}\Big(\text{PWConv}(\mathbf{x}, E)\Big)\Big) \tag{7}$$

After the expansion, the tensor is processed through the same operations as in the DWSepConv block, resulting in the intermediate tensor $\mathbf{z}_{\text{pw}}$, which is given by:

$$\mathbf{z}_{\text{pw}} = \text{DWSepConvBlock}(\mathbf{z}) \tag{8}$$

It is important to note that although a DWSepConv block is employed here, its internal skip connection is not utilized. Instead, the skip connection is implemented as defined for the MBConv block. In the absence of a skip connection ($SC = 0$), the output simplifies to $\mathbf{x}' = \mathbf{z}_{\text{pw}}$. Conversely, when a skip connection is active, the original input tensor $\mathbf{x}$ is combined with the processed tensor $\mathbf{z}_{\text{pw}}$ using a residual connection, as detailed in Eq. (6).

Each Conv Block has searchable parameters selected by the LLM. These include the convolution kernel size K chosen from $\{3, 5, 7\}$, which defines the receptive field, the number of stacked blocks L_i selected from $\{1, 2, 3, 4, 6\}$, which controls the stage's depth and representational capacity, the number of output channels C_{out} selected from $\{16, 24, 32, 48, 64, 96, 128, 160\}$, which governs the stage's

Example LLM Prompt for Architecture Generation

Role: You are a neural architecture design algorithm that exclusively outputs configurations in JSON format.

Task: Generate a lightweight neural network architecture tailored for CIFAR-100 image classification.

Objective: Achieve at least 70% accuracy while minimizing computational cost and memory usage.

Constraints:

- Minimize RAM usage by reducing intermediate activation size.
- Prioritize stride=2 in early blocks for downsampling.
- Use smaller expansion_factor and output_channels in early blocks.
- Limit SE blocks and their ratios to reduce activation memory.
- Ensure total MACs $\leq$ 350M.
- Image size: 160$\times$160.

Search Space: Use only values from the hierarchical search space: json_search_space.

Fig. 4. Example of initial prompt used to guide the LLM for architecture generation.

width and feature extraction capability, and the stride S, set to either 1 or 2, which manages spatial downsampling.

As presented in Table I, the search space for a single stage is further defined by the expansion factor E, the presence of a SE block δ_{se}, the convolution block type (Conv Block), the presence of a skip connection, and the activation function A. These parameters (E, δ_{se}, Conv Block, SC, A) along with K, L_i, C_{out}, and S from the previous paragraph, offer 3, 2, 2, 2, 3, 3, 5, 8, and 2 choices, respectively, yielding approximately 17,280 unique configurations per stage. All blocks within a stage share the same configuration. Thus, L_i acts as a discrete parameter in the search space rather than increasing individual stage complexity. Empirical evaluations indicated that a skeleton architecture with $N = 5$ stages provides an optimal balance between accuracy and efficiency, making it well-suited for resource-constrained environments. This configuration ensures the generated architectures meet stringent memory and computational constraints while maintaining competitive performance. Consequently, with $N = 5$ independently configured stages, the total search space encompasses approximately $17,280^5$ possible architectures.

Search algorithm. The proposed search algorithm leverages the capabilities of a pre-trained LLM to explore the search space described earlier effectively. Instead of fine-tuning, the LLM is guided through carefully crafted prompts designed to encode architectural constraints and optimization objectives. These prompts allow the LLM to generate candidate architectures that adhere to the stringent requirements of TinyML platforms, such as constraints on memory, the maximum allowed multiply-accumulate operations (MACs), and peak SRAM usage. An example of the prompt is provided in Fig. 4. To further improve efficiency, a series of validation checks is applied to each generated candidate before training. First, the estimated MACs of the architecture must lie within the defined range, MMACs_range = [min, max]. If a candidate exceeds or falls below these thresholds,

TABLE I

CONFIGURATION OPTIONS FOR THE ARCHITECTURE SEARCH SPACE.

Parameter	Description	Choices
C_{out}	Output channels	{16, 24, 32, 48, 64, 96, 128, 160}
K	Kernel size	{3, 5, 7}
S	Stride	{1, 2}
E	Expansion factor	{3, 4, 6}
δ_{se}	Enable the SE block	{True, False}
Conv Block	Convolution type	{DWConv, MBConv}
SC	Skip connection	{True, False}
A	Activation function	{ReLU6, Leaky ReLU, Swish}
L_i	Number of layers within the block	{1, 2, 3, 4, 6}
Total per stage	$\approx 17,280$ configurations	

it is rejected, and feedback is provided to the LLM through the prompts, specifying the current value and the desired range. Similarly, the architecture's estimated peak SRAM must not exceed a threshold for a quantized `int8` model. Candidates that violate this constraint are also rejected, with similar feedback provided to the LLM. Additionally, to avoid redundant computations, architectures that have already been generated in previous iterations are skipped, ensuring only unique candidates proceed to the training phase. By applying these checks, the search algorithm significantly reduces the time and resources spent training invalid or redundant candidates. Only architectures that pass all validation checks are considered legitimate for training, forming the basis for subsequent refinement through iterative feedback.

B. Feedback Loop With Pareto Optimization

To iteratively refine the architectures generated by the LLM, we implement a feedback loop guided by pareto optimization. Each valid candidate architecture proposed by the LLM undergoes a mini-training phase to evaluate its performance. Specifically, the candidate is trained on the CIFAR-100 [16] dataset for 30 epochs, providing an efficient yet consistent estimation of the key metrics required for feedback. These metrics include test accuracy, the number of MACs, and the total number of model parameters. Mini-training ensures that meaningful performance feedback is obtained without incurring excessive computational overhead, enabling a fast and iterative refinement process. Once the performance metrics are obtained, pareto optimization is applied to analyze the trade-offs between the competing objectives. Pareto optimization is a multi-objective optimization approach that identifies solutions achieving optimal trade-offs, where improving one objective would lead to the degradation of another. A solution is considered *Pareto optimal* if no other solution in the search space outperforms it across all metrics simultaneously. The collection of such solutions forms the *Pareto front*, representing the set of non-dominated candidates that balance the conflicting objectives.

In this work, we evaluate candidate architectures using three critical metrics: test accuracy, the number of MACs, and the total number of parameters. Accuracy reflects the classification performance on the test dataset. MACs quantify computational cost, while the total number of parameters influences the model's memory footprint, a key constraint for deployment on resource-limited devices. These objectives are inherently conflicting: improving accuracy may require increasing model complexity, while reducing computational cost and memory often comes at the expense of performance.

To navigate these trade-offs, we employ pareto dominance as the decision criterion. A solution A is said to dominate a solution B if it performs no worse than B across all metrics and is strictly better on at least one metric. While dominance suggests eliminating clearly inferior solutions, pareto optimization deliberately retains all non-dominated candidates to ensure that trade-offs between accuracy, computational cost, and memory usage are fully explored. For instance, one candidate might achieve superior accuracy but at the cost of higher MACs, while another may offer a more resource-efficient configuration with slightly lower accuracy. By maintaining the pareto front, we preserve this diversity, ensuring flexibility when selecting architectures for different deployment constraints. The pareto front is dynamically updated as new candidates are evaluated. Each new architecture is compared to the current pareto front: solutions dominated by the new candidate are removed, while non-dominated candidates are retained. If the new architecture itself is non-dominated, it is added to the pareto front. This ensures that only candidates representing optimal trade-offs remain. Additionally, we track the single architecture achieving the highest test accuracy, even if it does not meet resource constraints. This ensures that the architecture with the best overall performance is preserved, complementing the pareto-optimal solutions.

The updated pareto front and the best accuracy candidate are then used to generate structured feedback for the LLM. This feedback provides a summary of the current set of high-quality candidates, including their accuracy, MACs, and total number of parameters. Additionally, trends across the pareto front, such as the average accuracy or computational cost, are analyzed to guide the search process. Targeted suggestions are included to refine future candidates. For example, the feedback may encourage reducing MACs or parameters in solutions that exceed predefined thresholds, or it may focus on improving accuracy while maintaining resource efficiency.

By iteratively evaluating, filtering, and refining candidate architectures, the feedback loop enables the LLM to progressively converge toward solutions that achieve an optimal balance between predictive performance and computational efficiency. Once a candidate demonstrates strong performance in the pareto front or achieves the best accuracy, it is selected for full training. In the full training phase, the architecture is trained for an extended number of epochs to fully exploit its learning capacity. This final model is subsequently passed to the ViT-based KD phase, where additional refinements are applied to enhance accuracy further without increasing the model size or computational overhead.

C. *ViT-Based Knowledge Distillation (KD)*

KD is a model compression technique where a smaller student model learns to approximate the behavior of a larger teacher model. In this work, we employ logits-based KD, which focuses on aligning the output probabilities of the student and teacher models. This approach is particularly suitable for resource-constrained platforms, as it enhances accuracy without increasing model size or computational cost. Logits-based KD leverages softened output probability distributions to transfer knowledge from the teacher to the student. Given teacher logits z_t and student logits z_s, the softened probabilities are defined as:

$$p_t = \text{softmax}\left(\frac{z_t}{T}\right), \quad p_s = \text{softmax}\left(\frac{z_s}{T}\right), \tag{9}$$

where $T > 1$ is the temperature parameter. A higher T produces smoother distributions, emphasizing the relative probabilities of non-target classes. This softening enables the student model to capture inter-class relationships, improving generalization.

The distillation loss $\mathcal{L}_{\text{KD}}$ is expressed as the Kullback-Leibler (KL) divergence [32] between the teacher and student probabilities:

$$\mathcal{L}_{\text{KD}} = T^2 \cdot \text{KL}\left(p_t \parallel p_s\right) = T^2 \sum_{i=1}^{C} p_t^{(i)} \log\left(\frac{p_t^{(i)}}{p_s^{(i)}}\right), \tag{10}$$

where C is the total number of classes. The scaling factor T^2 ensures appropriate gradient magnitudes during optimization. The overall loss function combines the KD loss with the cross-entropy loss $\mathcal{L}_{\text{CE}}$, based on ground-truth labels y:

$$\mathcal{L} = \alpha\, \mathcal{L}_{\text{CE}} + (1-\alpha)\, \mathcal{L}_{\text{KD}}, \tag{11}$$

where α controls the relative contribution of each loss component. To ensure an effective balance between ground-truth supervision and distillation, we adopt an adaptive scheduling strategy for α. Initially, the student model relies more heavily on the cross-entropy loss to establish a strong understanding of the task. As training progresses, the influence of the distillation loss increases gradually. This is achieved by updating α at each epoch using:

$$\alpha = \alpha + (\alpha_{\text{final}} - \alpha)\left(\frac{\text{epoch}}{\text{num\_epochs}}\right), \tag{12}$$

where α_{final} is the desired final value. This gradual transition avoids overwhelming the student with complex teacher knowledge early in training and ensures that it benefits fully from the softened logits in later stages. We set a high temperature in our experiments, as higher values effectively capture fine-grained class relationships. For instance, softened logits can encode similarities between visually similar classes, such as "cat" and "tiger," which are otherwise challenging for the student to learn from hard labels alone.

The ViT [15] is employed as the teacher model in this work. Unlike convolution neural networks (CNNs) that focus on local spatial features, ViTs use self-attention mechanisms to capture global contextual relationships across the input image. This allows the teacher to produce enriched output probabilities that better reflect subtle relationships between classes. The student model benefits from this knowledge transfer by learning a richer representation of the dataset, leading to improved generalization.

IV. Experimental Results

A. Experimental setup

1) LLM Models: The neural NAS process is driven by open-weight LLMs to ensure accessibility and reduce local execution costs. Specifically, Llama3.2-3B-Instruct and Llama3.1-8B-Instruct [13], along with Qwen2.5-3B-Instruct [14], are employed for their efficiency, with parameter sizes up to 8 billion. This parameter range represents a suitable balance between performance and computational cost. The use of their instruction-tuned versions ensures more coherent outputs. Furthermore, leveraging open-source models enhances the transparency and reproducibility of the NAS workflow.

2) Dataset: The CIFAR-100 dataset [16] is employed to evaluate the generated architectures. This dataset comprises 60,000 images distributed across 100 classes, with a training set of 50,000 images and a test set of 10,000 images. For the experiments, images are resized to 160×160 pixels and standard preprocessing techniques, including normalization, are applied. CIFAR-100 was chosen for its balanced class distribution, which allows the evaluation to focus on the representational capacity of the generated backbone architectures.

3) Training Details: The training process is structured into three distinct phases: a lightweight mini-phase for rapid initial evaluation of candidate architectures, a full training phase to optimize the final selected model, and a KD phase for further fine-tuning. Each phase is tailored with specific data augmentations and hyperparameter settings to balance computational efficiency and model performance.

Mini-Phase Training. In this initial phase, each candidate architecture generated by the LLM undergoes a lightweight evaluation to provide rapid feedback for Pareto optimization while minimizing computational overhead. The candidates are trained for 30 epochs on the CIFAR-100 dataset with a batch size of 128. Training begins with a learning rate of 0.5, which is progressively reduced by a factor of 0.1 every 10 epochs using a step learning rate scheduler. A warm-up period of 10 epochs is employed to ensure gradient stability during the initial stages of training. The stochastic gradient descent (SGD) optimizer is used with Nesterov momentum set to 0.9 and weight decay set to 10^{-4}. Standard data augmentations, including image resizing to 160×160 pixels and AutoAugment [33], are applied to improve generalization. Advanced augmentations, such as MixUp, are excluded to avoid introducing unnecessary complexity during the limited 30-epoch training phase. This streamlined setup ensures both a fair comparison and efficient convergence across candidate architectures.

Full-Phase Training. The architecture selected from the Pareto front undergoes full training to optimize its performance. During this phase, the model is trained for 120 epochs with a batch size of 128, using the SGD optimizer with Nesterov momentum set to 0.9 and weight decay set to 10^{-4}. The learning rate follows a warmup-cosine annealing schedule: it begins at 0.0, increases linearly to 0.5 during the first 20 epochs (the warmup phase), and then decays smoothly following a cosine function for the remainder of the training. To improve generalization, AutoAugment [33] is employed alongside MixUp [34] with an alpha value of 0.2. These augmentations enhance robustness while preventing overfitting, ensuring that the final model achieves optimal performance.

KD Phase Training. In this final fine-tuning phase, logits-based KD is applied to further refine the model. A pretrained ViT-B/16[1] serves as the teacher model, transferring its knowledge to the student model (the selected architecture). The distillation process uses a temperature $T = 10$ to soften the teacher's outputs, allowing the student to learn inter-class relationships. The total loss combines cross-entropy and distillation losses, weighted by an adaptive alpha scheduling strategy ($\alpha = 0.4$, $\alpha_{\text{final}} = 0.8$) as defined in (12). Since this phase focuses on fine-tuning rather than extensive retraining, no advanced data augmentations, such as MixUp or AutoAugment, are applied. Only basic preprocessing, including image resizing to 160×160 pixels and normalization, is performed to maintain consistency with earlier phases. The KD phase runs for 50 epochs, allowing the student to refine its performance efficiently while adhering to computational constraints.

4) Hardware: The models are trained on an NVIDIA RTX-8000 GPU server equipped with 4608 CUDA cores and 48 GB of VRAM, running a Linux-based operating system. For inference evaluation, the models are profiled on an STM32H743 MCU, which features 2 MB of flash memory and 512 KB of SRAM. Although the STM32H743 has 512 KB of SRAM, the models are specifically tested to meet the more stringent constraint of 320 KB SRAM, a common limitation in many TinyML applications. Furthermore, the minimum and maximum MMACs are set to 70 and 350, respectively, to ensure the models operate within the defined computational limits. This setup reflects the real-world constraints of TinyML platforms, ensuring that the designed architectures are both efficient and suitable for deployment in resource-constrained environments.

5) Baselines and Evaluation Metrics: The effectiveness of the proposed method is demonstrated by comparing our optimized architectures LMaNet-Core, QwNet-Core, and LMaNet-Elite which are generated using Llama3.2-3B-Instruct, Qwen2.5-3B-Instruct, and Llama3.1-8B-Instruct, respectively, against SOTA baselines for resource-constrained platforms. These baselines include lightweight models such as XiNet [10] and various versions of MCUNet [8], known for their balance of efficiency and accuracy. Model performance is evaluated using four key metrics: classification accuracy on the CIFAR-100 test dataset, number of MACs to assess computational complexity, total model parameters to evaluate memory usage, and peak SRAM usage during inference to

[1]https://huggingface.co/google/vit-base-patch16-224-in21k

TABLE II
COMPARISON OF CIFAR-100 CLASSIFICATION PERFORMANCE WITH SOTA BASELINES FOR 160×160 INPUT RESOLUTION.

Models	Accuracy (%)↑	MACs (M)↓	Flash (MB)↓	#Parameters (M)↓	Search cost (days)↓
MCUNet-in4 [9] (NeurIPS21)	72.86	134	1.65	1.73	12.5
MCUNet-in3 [9] (NeurIPS21)	69.62	72	0.70	0.74	12.5
XiNet-Class [10] (ICCV23)	72.27	259	–	–	manual
Ours (LMaNet-Core)	74.20	82	0.53	0.51	2.5
Ours (QwNet-Core)	73.00	**71**	**0.24**	**0.18**	3.5
Ours (LMaNet-Elite)	**74.50**	86	0.54	0.44	**1.5**

ensure hardware compatibility. These metrics provide a comprehensive assessment of accuracy, computational cost, and resource efficiency.

B. Results and Analysis

1) Accuracy, Computational Complexity, and Model Size Trade-offs: Table II presents a detailed comparison of our proposed models with SOTA baselines in terms of accuracy on the CIFAR-100 dataset, MACs, model size and search cost. Among the proposed models, LMaNet-Elite achieves the highest accuracy at 74.50%, surpassing the MCUNet-in4 model by 1.64% and XiNet-Class by 2.23%. LMaNet-Elite also demonstrates superior efficiency, with a computational cost of 86 million MACs, significantly lower than MCUNet-in4 with 134 million MACs. Furthermore, LMaNet maintains a compact model size of 0.54 MB, making it an ideal choice for TinyML platforms. While MCUNet-in3 and XiNet-Class models also achieve competitive accuracy, they exhibit trade-offs. MCUNet-in3 achieves a lower accuracy of 69.62% with a computational cost of 72 million MACs, whereas XiNet-Class incurs significantly higher computational costs (259 million MACs), making it less efficient than LMaNet-Elite despite its competitive accuracy of 72.27%. LMaNet-Core and QwNet-Core also surpass the SOTA models, achieving competitive accuracies of 74.20% and 73.00%, respectively, while using smaller model sizes of 0.51MB (LMaNet-Core) and 0.24MB (QwNet-Core). These models strike an optimal balance between performance and memory efficiency, with QwNet-Core being especially suitable for highly resource-constrained devices due to its compact model size of 0.24MB. In addition, our models significantly outperform MCUNet in search cost. LMaNet-Elite, LMaNet-Core, and QwNet-Core require only 1.5, 2.5, and 3.5 days, respectively, over 4 times faster than MCUNet's 12.5 days. This demonstrates the efficiency of the LLM-guided methodology in reducing search time while adhering to the stringent resource constraints of the STM32H743 MCU.

2) SRAM Usage and Memory Efficiency: Figure 5 provides a visual comparison of the SRAM usage across different models, emphasizing their suitability for deployment in memory-constrained environments. The LMaNet-Elite model shows the lowest SRAM usage at 165KB, well below the 320KB limit, making it ideal for highly resource-constrained devices. In contrast, the MCUNet-in4 model, with a peak SRAM usage of 456KB, exceeds the 320KB limit, limiting its deployment to systems with higher memory capacities.

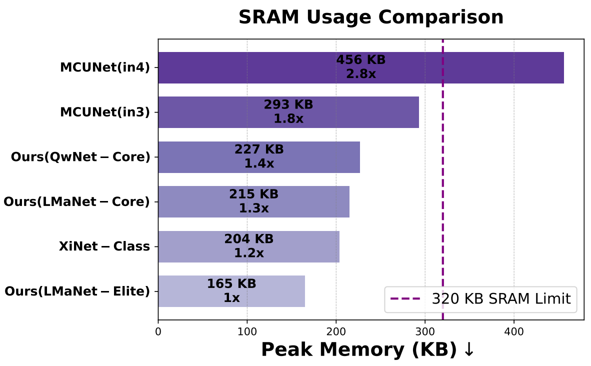

Fig. 5. SRAM usage comparison of different models for deployment on STM32H7. The purple dashed line indicates the 320 KB SRAM constraint. Each bar shows the SRAM usage (in KB) of the models, with comparative factors relative to **LMaNet-Elite**, which achieves the lowest peak memory consumption.

The LMaNet-Core and QwNet-Core models also fit within the 320KB SRAM limit, with usages of 215KB and 227KB, respectively. These models offer a strong balance between memory efficiency and performance, making them suitable for environments with stringent memory constraints. Notably, all models fit comfortably under the more relaxed 512KB SRAM limit, ensuring compatibility with platforms like the STM32H743 MCU. While both MCUNet-in3 and XiNet-Class meet the 320KB SRAM limit, they come with trade-offs in other areas as discussed earlier. MCUNet-in3 uses 293KB of SRAM but sacrifices accuracy. On the other hand, XiNet-Class, despite using only 204KB of SRAM, incurs significantly higher computational costs, making it less efficient than our proposed models. These results demonstrate the strength of our LLM-guided framework in generating models that balance high accuracy with low computational cost and minimal memory usage, pushing the boundaries of TinyML for real-world applications.

V. ABLATION STUDY

This ablation study investigates the impact of search efficiency with LLMs, focusing specifically on the performance of Llama3.2-3B-Instruct, Llama3.1-8B-Instruct, and Qwen2.5-3B-Instruct. Furthermore, the study explores the influence of the proposed ViT-KD method in further improving model accuracy, examining how it enhances the performance achieved through the configuration search phase.

A. Search efficiency with LLMs

In the experiments, SRAM is treated as a hard constraint, while accuracy, MACs, and the number of parameters serve as soft constraints optimized by the LLM during the search process to update the Pareto front. To ensure a fair comparison among the three LLMs, the same training configuration is used across the mini-phase training, full training phase, and ViT-KD phase. Regarding the LLM configuration parameters, the decoding temperature is set to $T = 1.5$ and the minimum probability threshold (`min_p`) is set to 0.1. A higher temperature promotes exploration by increasing randomness in the

TABLE III
PERFORMANCE METRICS FOR MODEL GENERATION ACROSS LLM MODELS IN THE MINI-PHASE (30 EPOCHS), PRESENTED IN THE FORMAT [MIN, MAX] AVG. TC: TOTAL CANDIDATES, PC: PARETO CANDIDATES.

Model	**TC**	**PC**	**Accuracy (%)↑**	**MACs (M)↓**	**#Parameters (M)↓**
Llama3.2-3B	127	27	[38.68, 68.58] 60.50	[71.17, 255.86] 112.01	[0.04, 0.73] 0.15
Llama3.1-8B	100	29	[45.33, 67.87] 61.50	[72.05, 322.36] 130.47	[0.03, 0.59] 0.18
Qwen2.5-3B	213	43	[48.41, 70.31] 61.90	[70.19, 327.28] 140.05	[0.05, 0.77] 0.25

TABLE IV
IMPACT OF VIT-KD ON ACCURACY. THE TABLE SHOWS THE ACCURACY BEFORE AND AFTER APPLYING THE VIT-KD METHOD.

Model	**w/o ViT-KD (%)**	**w/ ViT-KD (%)**	**Gain (%)↑**
LMaNet-Core	72.88	74.20	+1.32
QwNet-Core	71.60	73.00	+1.40
LMaNet-Elite	73.15	74.50	+1.35

output, which helps the LLM generate diverse configurations. Simultaneously, the `min_p` threshold prevents the selection of excessively low-probability tokens, thereby reducing the likelihood of incoherent or invalid outputs. This balance between exploration and control is crucial for effectively navigating the hierarchical search space while adhering to task constraints. By fixing these variables and maintaining the same configuration, the choice of LLM becomes the sole factor in the experiment. The experiment is conducted over 500 iterations, where all three models adhere to the SRAM hard constraint. However, significant differences are observed in the time spent on the search, the final model accuracy, and SRAM usage. Table III presents several key observations regarding the efficiency and effectiveness of the different LLMs in identifying suitable configurations. For each model, the total number of generated candidates, the number of Pareto candidates, and the average values for accuracy, MACs, and number of parameters of the Pareto candidates are reported. Llama 3B generated a total of 127 candidates, of which 27 qualified as Pareto candidates. The average accuracy of the Pareto candidates was 60.50%, with an average of 112.01 million MACs and 0.15 million parameters. Despite having a smaller search space relative to the other models, Llama 3B still identified viable configurations that satisfied the SRAM constraint, demonstrating its efficiency in terms of both search time and candidate generation. The search process for this model required 2.5 days as indicated in Table II. Llama 8B produced 100 total candidates, with 29 Pareto candidates. The average accuracy of the Pareto candidates was 61.50%, with an average of 130.47 million MACs and 0.18 million parameters. The search process for Llama 8B was completed in 1.5 days (see Table II), achieving higher average accuracy compared to Llama 3B. This suggests that larger LLMs, may be more effective at optimizing for both accuracy and SRAM usage. The enhanced search capability of Llama 8B likely contributes to its superior performance. On the other hand Qwen 3B, which generated 213 candidates, yielded 43 Pareto candidates. The average accuracy for the Pareto candidates was 61.90%, with an average of 140.05 million MACs and 0.25 million parameters. Despite the larger search space and requiring 3.5 days as seen in Table II for the search process, Qwen 3B achieved only marginally better results in terms of average accuracy compared to the other models. The extended search time did not translate into significantly improved performance, indicating diminishing returns with an increased search space. These findings underscore the importance of search efficiency in configuration optimization. While the size of the LLM and the number of generated candidates are significant factors, the search efficiency remains a critical determinant of identifying the best-performing configurations. In particular, Llama 8B provided the best balance between search time and accuracy, achieving superior Pareto average accuracy in the shortest search time cost. Qwen 3B, despite generating a larger number of candidates, did not exhibit substantial improvements in performance and required more time for the search, reinforcing the notion that search efficiency can outweigh the breadth of the search space.

B. *Impact of ViT-KD on Accuracy*

The results presented in Table IV demonstrate the effectiveness of the proposed ViT-based KD method in enhancing the accuracy of our models. Before applying KD, LMaNet-Core achieved an accuracy of 72.88%, QwNet-Core 71.60%, and LMaNet-Elite 73.15%. Following the KD phase, which employed a ViT teacher model with 93% accuracy, all student models exhibited notable improvements in accuracy. Specifically, LMaNet-Core's accuracy increased by 1.32%, QwNet-Core by 1.40%, and LMaNet-Elite by 1.35%. These results indicate that the ViT-based KD method contributes significantly to improving the accuracy of the student models, showcasing the effectiveness of leveraging a high-performing teacher model to enhance the performance of smaller, more efficient student models in resource-constrained environments.

VI. EXPLANATORY POTENTIAL OF LLM FOR XAI

The LLM-guided NAS approach generates efficient architectures, but the rationale behind the design choices often remains opaque. Leveraging the explanatory capabilities of LLMs aligns with XAI principles, enabling researchers to query the LLM for insights into why specific configurations were chosen. This approach enhances interpretability and fosters trust in automated architecture generation. For instance, Figure 6 illustrates the explanation provided by the LLM for the design of LMaNet-Elite. The LLM justifies the strategic use of kernel sizes (3×3, 5×5, and 7×7) for spatial feature extraction, the fixed expansion factor of 3 to balance representation and computational cost, and the selective inclusion of SE block to enhance channel attention in critical layers. These explanations bridge the gap between automation and human understanding, transforming the NAS process into a more interpretable and collaborative endeavor. While LLMs provide valuable insights, their reasoning can sometimes be overly generalized or imprecise, requiring critical evaluation of the generated explanations. Nonetheless, this integration of XAI principles represents a significant step toward making

Explain why this design was chosen for the current iteration. Highlight the reasoning behind these choices.

The selected architecture leverages a combination of depthwise and MBConv blocks to balance performance and computational efficiency:

- **Kernel sizes:** 3×3, 5×5, and 7×7 are strategically used to optimize computational efficiency and enhance spatial feature extraction in deeper layers.
- **Expansion factors:** Fixed at 3 across all blocks to provide a balanced growth of feature map capacity while minimizing computational overhead.
- **Stride:** Downsampling with stride 2 is applied in early and middle blocks to reduce spatial dimensions and computational cost, while stride 1 in later blocks preserves spatial fidelity for final feature aggregation.
- **SE Ratio:** SE blocks with a ratio of 0.5 are incorporated selectively in critical blocks to enhance channel attention and improve representational power.
- **Activation functions:** `ReLU6` is utilized for efficiency in early and intermediate layers, while `LeakyReLU` in deeper blocks improves non-linear representation.
- **Skip operations:** Residual connections in selected layers were used to enhance optimization and stability.

Fig. 6. LLM Explanation response for the design choices behind LMaNet-Elite. The response highlights how the configuration balances resource constraints with performance, emphasizing the strategic reasoning of the LLM.

automated architecture generation more transparent and trustworthy.

VII. Conclusion

In this paper, we have introduced a novel framework for designing efficient neural architectures for resource-constrained platforms. Our approach integrates LLM-guided NAS, ViT-based KD, and an explainability module. The results showed that LLMs can revolutionize TinyML by achieving SOTA performance on CIFAR-100, surpassing existing baselines in accuracy while adhering to stringent constraints. This framework not only improves performance but also reduces search cost by streamlining the design process. Through Pareto optimization and adaptive fine-tuning, it effectively balances accuracy, efficiency, and deployability. Additionally, the explainability module enhances transparency by providing valuable insights into the design decisions made by the LLM. Future work will focus on exploring larger datasets, leveraging advanced LLM capabilities, and assessing the scalability of the approach on even more resource-constrained hardware.

References

[1] Christophe El Zeinaty, Glenn Herrou, Wassim Hamidouche, and Daniel Menard, "Dicetrack: Lightweight dice classification on resource-constrained platforms with optimized deep learning models," in *ICASSP 2024 - 2024 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2024, pp. 51–55.

[2] Ji Lin, Ligeng Zhu, Wei-Ming Chen, Wei-Chen Wang, and Song Han, "Tiny machine learning: Progress and futures [feature]," *IEEE Circuits and Systems Magazine*, vol. 23, no. 3, pp. 8–34, 2023.

[3] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2018, pp. 4510–4520.

[4] Markus Nagel et al., "A white paper on neural network quantization," *arXiv preprint arXiv:2106.08295*, 2021.

[5] Yann Le Cun, John S. Denker, and Sara A. Solla, "Optimal brain damage," in *Proceedings of the 3rd International Conference on Neural Information Processing Systems*, Cambridge, MA, USA, 1989, NIPS'89, p. 598–605, MIT Press.

[6] Geoffrey Hinton, "Distilling the knowledge in a neural network," *arXiv preprint arXiv:1503.02531*, 2015.

[7] Song Han, Huizi Mao, and William J Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv preprint arXiv:1510.00149*, 2015.

[8] Ji Lin et al., "Mcunet: tiny deep learning on iot devices," in *Proceedings of the 34th International Conference on Neural Information Processing Systems*, Red Hook, NY, USA, 2020, NIPS '20, Curran Associates Inc.

[9] Ji Lin et al., "Mcunetv2: memory-efficient patch-based inference for tiny deep learning," in *Proceedings of the 35th International Conference on Neural Information Processing Systems*, Red Hook, NY, USA, 2024, NIPS '21, Curran Associates Inc.

[10] Alberto Ancilotto, Francesco Paissan, and Elisabetta Farella, "Xinet: Efficient neural networks for tinyml," in *2023 IEEE/CVF International Conference on Computer Vision (ICCV)*, 2023, pp. 16922–16931.

[11] Mingkai Zheng et al., "Can gpt-4 perform neural architecture search?," *arXiv preprint arXiv:2304.10970*, 2023.

[12] Muhammad Umair Nasir et al., "Llmatic: Neural architecture search via large language models and quality diversity optimization," in *Proceedings of the Genetic and Evolutionary Computation Conference*, New York, NY, USA, 2024, GECCO '24, p. 1110–1118, Association for Computing Machinery.

[13] Hugo Touvron et al., "Llama: Open and efficient foundation language models," *arXiv preprint arXiv:2302.13971*, 2023.

[14] Jinze Bai et al., "Qwen technical report," *arXiv preprint arXiv:2309.16609*, 2023.

[15] Bichen Wu, Xu, et al., "Visual transformers: Token-based image representation and processing for computer vision," *arXiv preprint arXiv:2006.03677*, 2020.

[16] Alex Krizhevsky, "Learning multiple layers of features from tiny images," 2009.

[17] Jean Kaddour et al., "Challenges and applications of large language models," *arXiv preprint arXiv:2307.10169*, 2023.

[18] Josh Achiam et al., "Gpt-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

[19] Rohan Anil et al., "Gemini: a family of highly capable multimodal models," *arXiv preprint arXiv:2312.11805*, 2023.

[20] Anthropic, "The claude 3 model family: Opus, sonnet, haiku," .

[21] Juyong Jiang et al., "A survey on large language models for code generation," *arXiv preprint arXiv:2406.00515*, 2024.

[22] Microsoft Research AI4Science and Microsoft Azure Quantum, "The impact of large language models on scientific discovery: a preliminary study using gpt-4," *arXiv preprint arXiv:2311.07361*, 2023.

[23] Marah Abdin et al., "Phi-4 technical report," *arXiv preprint arXiv:2412.08905*, 2024.

[24] Ebtesam Almazrouei et al., "The falcon series of open language models," *arXiv preprint arXiv:2311.16867*, 2023.

[25] B Zoph, "Neural architecture search with reinforcement learning," *arXiv preprint arXiv:1611.01578*, 2016.

[26] Esteban Real et al., "Large-scale evolution of image classifiers," in *Proceedings of the 34th International Conference on Machine Learning - Volume 70*. 2017, ICML'17, p. 2902–2911, JMLR.org.

[27] Kirthevasan Kandasamy, Willie Neiswanger, Jeff Schneider, Barnabás Póczos, and Eric P. Xing, "Neural architecture search with bayesian optimisation and optimal transport," in *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, Red Hook, NY, USA, 2018, NIPS'18, p. 2020–2029, Curran Associates Inc.

[28] Hanxiao Liu, Karen Simonyan, and Yiming Yang, "Darts: Differentiable architecture search," *arXiv preprint arXiv:1806.09055*, 2018.

[29] Han Cai, Ligeng Zhu, and Song Han, "Proxylessnas: Direct neural architecture search on target task and hardware," *arXiv preprint arXiv:1812.00332*, 2018.

[30] Erik Nijkamp et al., "Codegen: An open large language model for code with multi-turn program synthesis," *arXiv preprint arXiv:2203.13474*, 2022.

[31] Kalyanmoy Deb and Kalyanmoy Deb, *Multi-objective Optimization*, pp. 403–449, Springer US, Boston, MA, 2014.

[32] Fernando Perez-Cruz, "Kullback-leibler divergence estimation of continuous distributions," in *2008 IEEE International Symposium on Information Theory*, 2008, pp. 1666–1670.

[33] Ekin D. Cubuk, Barret Zoph, Dandelion Mané, Vijay Vasudevan, and Quoc V. Le, "Autoaugment: Learning augmentation strategies from data," in *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019, pp. 113–123.

[34] H Zhang, M Cisse, Y Dauphin, and D Lopez-Paz, "mixup: Beyond empirical risk management," in *6th Int. Conf. Learning Representations (ICLR)*, 2018, pp. 1–13.