AutoTailor: Automatic and Efficient Adaptive Model Deployment for Diverse Edge Devices

Mengyang Liu
Southeast University
University of Macau

Chenyu Lu
HKUST (GZ)

Haodong Tian
Tsinghua University

Fang Dong*
Southeast University

Ruiting Zhou
Southeast University

Wei Wang
HKUST

Dian Shen
Southeast University

Guangtong Li
Southeast University

Ye Wan
Southeast University

Li Li
University of Macau

Abstract

On-device machine learning (ML) has become a fundamental component of emerging mobile applications. Adaptive model deployment delivers efficient inference for heterogeneous device capabilities and performance requirements through customizing neural architectures. SuperNet-based approaches offer a promising solution by generating numerous model variants from a pre-trained ML model. However, applying SuperNet in existing frameworks requires tedious model-aware development and time-consuming hardware-aware profiling, which limits their practical adoption.

We present AutoTailor, the first framework to enable automated, end-to-end SuperNet-based adaptive model deployment for edge devices. Unlike manual SuperNet construction, AutoTailor employs a computation graph-guided compilation approach to automatically transform user-provided ML models into SuperNets. To support efficient specialization, AutoTailor incorporates learning-free latency and accuracy predictors, enabling low-cost yet accurate performance prediction. Our extensive evaluations demonstrate that AutoTailor reduces the lines of code for SuperNet construction by 10.8–34.5×, reduces predictor profiling time by 2.2–11.5×, and achieves up to 15.60% absolute accuracy improvement and 60.03% latency reduction compared to state-of-the-art approaches across diverse models and devices.

CCS Concepts: • **Computing methodologies** → **Neural networks**; • **Computer systems organization** → **Embedded systems**.

Keywords: adaptive model deployment, supernets, neural architecture adaptation, edge computing

*Corresponding author: fdong@seu.edu.cn.



This work is licensed under a Creative Commons Attribution 4.0 International License.

SIGOPS ATC 2026, Hong Kong, China

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2818-1/2026/11
https://doi.org/10.1145/3828523.3857084

ACM Reference Format:
Mengyang Liu, Chenyu Lu, Haodong Tian, Fang Dong, Ruiting Zhou, Wei Wang, Dian Shen, Guangtong Li, Ye Wan, and Li Li. 2026. AutoTailor: Automatic and Efficient Adaptive Model Deployment for Diverse Edge Devices. In *The 2026 ACM SIGOPS Annual Technical Conference (SIGOPS ATC 2026), November 15-18, 2026, Hong Kong, China.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3828523.3857084

1 Introduction

Deep neural networks (DNNs) are increasingly executed on edge devices to provide private intelligence for a wide range of mobile applications, including augmented reality [30], video analytics [29], and robot navigation [34]. Static DNN deployment often fails to meet the diverse requirements of applications (e.g., real-time responsiveness or energy efficiency) on heterogeneous edge devices (e.g., industrial GPU boards or personal mobile phones). To solve this problem, adaptive model deployment [14, 23, 69] has emerged as a widely adopted strategy. It first generates multiple model variants with different latency-accuracy trade-offs from a pre-trained DNN and then deploys the most suitable one according to device capabilities and runtime conditions.

The number and granularity of model variants determine the performance of adaptive model deployment. Early work adopted coarse-grained uniform filter pruning [14] to generate variants with different widths. LegoDNN [23] refined this approach by applying non-uniform pruning, providing more variants with improved overall performance. AdaptiveNet [69] further extended the design space by varying not only width but also depth. These methods generate model variants through *block scaling*, where each block has multiple weight-independent variants, and complete model variants are constructed by combining them. However, in block scaling approaches, each block requires fine-tuning to preserve accuracy, which limits scalability in producing a larger number of variants and achieving higher performance.

To overcome these limitations, *SuperNets* [3, 62, 67] offer a scalable alternative through weight sharing. Instead of training model variants independently, SuperNets jointly train multiple variants (SubNets) that share parameters. As shown in Fig. 1, adopting SuperNets for adaptive deployment achieves up to 34.67% latency reduction with the same

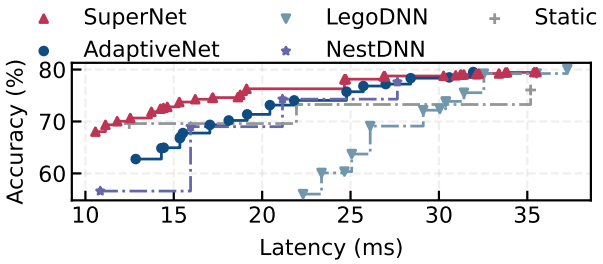


Figure 1. Latency-accuracy tradeoff of adaptive ResNet-50 on the Exynos 1380 SoC's big cores using the ImageNet1k dataset across different methods. SuperNet consistently achieves state-of-the-art performance. Static uses ResNet-18 and ResNet-34 as smaller model variants.

accuracy and 7.89% absolute accuracy improvement under the same latency over state-of-the-art adaptive deployment methods for the ResNet50 DNN [24].

Unfortunately, applying SuperNets in practice incurs substantial development and deployment costs due to their high flexibility. On the development side, users must manually transform a static DNN into a dynamic one that supports multiple architectural modifications. This process not only bloats the codebase but also demands deep expertise in DNN architecture design. For example, ElasticViT [62] transforms LeViT [20] with 4× more lines of code than its original implementations (1638 vs. 406) to make each module transformable. Retiarii [79] alleviates this burden through the Mutator abstraction, eliminating the need to manually implement transformable operators. Users can annotate target operators with predefined Mutators and specify candidate configurations, enabling automatic model variant generation. Nevertheless, this process still requires architectural expertise to identify suitable operators and ensure configuration compatibility, particularly to prevent tensor shape mismatches. Thus, SuperNet construction remains challenging and labor-intensive, especially for non-experts.

On the deployment side, optimizing model architectures for specific devices requires profiling candidate variants, which is highly time-consuming. The vast number of variants within a SuperNet makes exhaustive offline profiling infeasible, while online profiling during optimization introduces significant inefficiency. To mitigate this, existing approaches [16, 77] train operator-level latency predictors and aggregate their outputs to estimate model-level latency, thereby simplifying predictor construction. However, the accuracy of these learning-based methods fundamentally depends on the quantity and quality of training data. Consequently, building such predictors commonly demands hours of data collection (e.g., 1 hour of profiling to build a predictor with only 50% accuracy as shown in Fig. 7) for each model-platform pair, yet the resulting predictors still suffer from non-negligible prediction errors.

Our goal is to enable automatic and efficient SuperNet deployment, thereby advancing adaptive model deployment in practice. Achieving this goal presents two key challenges. First, automating SuperNet construction requires decoupling users from the DNN programs, leaving the framework unaware of which modules are transformable or what candidate choices to assign. Second, the vast number of possible model variants makes accurate predictor training difficult: the larger the design space, the more profiling data are required to train the predictor effectively, resulting in prohibitive time costs.

We observe two fundamental characteristics of SuperNets that can be leveraged to address these challenges. First, SuperNets exhibit **graph structural consistency**: they generate variants by applying architectural modifications to a base architecture, which preserves the computation graph of the original DNN. This property enables transformable modules to be automatically identified and replaced directly from the computation graph, eliminating the need for manual annotation. Second, SuperNets exhibit **cross-SubNet architectural similarity**: although they contain many SubNets, most operators are repeated. Our analysis (Section 2.4) shows a high repetition ratio (96–98%), meaning the number of unique operators is significantly small. This property enables learning-free predictors that profile only unique operators, eliminating costly sampling and predictor training.

We present AutoTailor, an end-to-end adaptive model deployment framework for edge devices. AutoTailor automatically compiles supported static-graph DNNs into SuperNets represented by TailorIR, a novel intermediate representation. The resulting SuperNet is then fine-tuned to preserve the accuracy of its model variants. To enable efficient latency prediction, AutoTailor identifies unique operators across the SuperNet by eliminating redundant configurations through modification dependency analysis and execution-free tensor shape inference. Operator-level latency measurements are then aggregated to predict model-level latency. For accuracy estimation, AutoTailor applies a similar approach to improve prediction efficiency. Finally, guided by these predictors and user-specified requirements, AutoTailor searches for the optimal model variant and generates deployable DNNs tailored to the target edge platform.

The key contributions of AutoTailor are as follows:

- To the best of our knowledge, AutoTailor is the first framework to provide automated, end-to-end adaptive model deployment for edge devices.
- We propose a computation graph-guided compilation method and a method for building learning-free predictors to reduce development and deployment overheads in applying SuperNets.
- We extensively evaluate AutoTailor across diverse DNNs and edge devices, demonstrating that it reduces

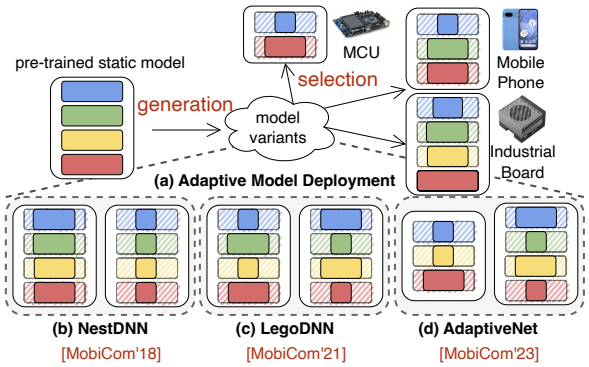


Figure 2. An illustration of adaptive model deployment and different model variant generation methods. A four-layer DNN is used as a running example, where each color denotes a distinct layer.

engineering effort by 10.8–34.5× and reduces predictor profiling time by 2.2–11.5×, while achieving up to 15.60% absolute accuracy improvement and 60.03% latency reduction compared to state-of-the-art approaches.

2 Background and Motivation

2.1 Adaptive Model Deployment for Edge Devices

Deploying powerful deep neural networks (DNNs) on edge devices to enable private and personalized intelligence has become a rapidly growing trend in mobile applications [68]. Traditional static model deployment, placing a single fixed-architecture model on the device for inference, proves inadequate for edge platforms due to two key characteristics of edge devices. First, edge devices exhibit wide hardware diversity, ranging from high-performance NVIDIA Orin boards [47] to commodity smartphones [52] and resource-constrained microcontrollers (MCUs) [60]. Second, performance requirements vary across edge applications: some real-time processing tasks demand high-frequency DNN inference [29, 30, 40, 80], while persistent background applications [17, 18] require energy-efficient, low-power DNN inference. To better align model execution with heterogeneous resources (e.g., compute capacity) and application requirements (e.g., accuracy and latency), adaptive model deployment has emerged as a promising approach [14, 23, 69]. This process consists of two steps as shown in Fig. 2(a): (1) generating multiple model variants with different architectures, accuracy, and efficiency; and (2) selecting the most suitable variant for a given device and runtime requirement.

The effectiveness of adaptive model deployment depends on the diversity and granularity of available model variants. Fig. 2(b-d) illustrates representative variants generated by existing adaptive deployment methods. Early work such as NestDNN [14] relied on uniform filter pruning [37], applying the same sparsity ratio across all layers. LegoDNN [23] extended this idea by enabling non-uniform

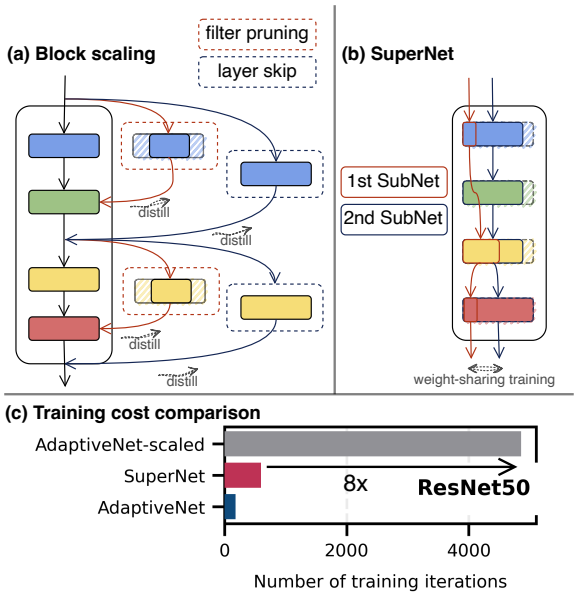


Figure 3. Comparison between block scaling and SuperNet-based model variant generation.

layer-wise pruning, thereby expanding the design space with greater flexibility. Moving beyond width-only modifications,AdaptiveNet [69] further introduced a layer-skipping strategy, which substantially enlarges the search space and refines the granularity of architectural adaptation. Collectively, these efforts reveal a consistent trend toward expanding the model variant design space to improve adaptive deployment performance.

2.2 SuperNet-based Adaptive Model Deployment

LegoDNN [23] and AdaptiveNet [69] adopt a block scaling approach to generate model variants. As shown in Fig. 3(a), block scaling constructs multiple block variants, and different model variants are formed by selecting different combinations of these block choices. Since each new block must be independently constructed and retrained with additional supervision (e.g., block-wise distillation [69]), the overhead of memory consumption and training grows linearly with the number of blocks. Such overhead fundamentally limits scalability, restricting both the number of supported model variants and the granularity of modifications.

To overcome these limitations, we find that the SuperNet [3] is a more scalable and efficient model variant generation mechanism. Unlike block scaling, SuperNet can represent a vast number of SubNets in a weight-sharing manner as shown in Fig. 3(b). Different SubNets activate distinct subsets of layers or weights within the same SuperNet, enabling the reuse of parameters across variants. SuperNet training is performed by randomly sampling multiple SubNets per batch, with the total loss aggregated over their outputs. This weight-sharing design significantly reduces memory and

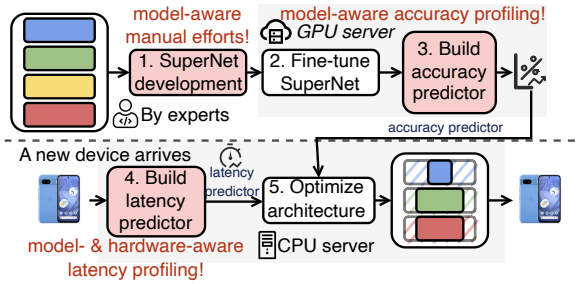


Figure 4. An illustration of the SuperNet workflow.

training overhead, while supporting a much larger and finer-grained variant space. Scaling AdaptiveNet to match the architectural design space of the SuperNet (excluding input modifications) requires 27× more independent blocks and incurs 27× more training iterations, which is also 8× more than SuperNets, as shown in Fig. 3(c). Consequently, SuperNet provides a more practical foundation for adaptive model deployment for resource-constrained and dynamic edge environments.

Therefore, we advance SuperNet as the model variant generation method to push the performance frontier of adaptive model deployment. As shown in Fig. 1, our experimental results demonstrate the advantages of the SuperNet-based approach compared with existing methods. Specifically, the SuperNet-based method achieves up to 60.03% latency reduction with the same accuracy and 15.6% absolute accuracy improvement under the same latency constraint compared with AdaptiveNet [69] benefiting from its $1,000,000,000\times$ larger design space.

2.3 Problems in Applying SuperNet

Despite the significant performance improvements demonstrated by SuperNet-based adaptive deployment, its adoption in real-world systems remains costly. Fig. 4 illustrates the conventional SuperNet workflow and its inefficiencies. First, experts manually design and develop a SuperNet (Step 1), requiring significant model-aware expertise. The SuperNet is then fine-tuned on a GPU server (Step 2) and used to build an accuracy predictor (Step 3). When a new device arrives, a latency predictor must be built through extensive hardware profiling (Step 4), which incurs high device-specific costs. Finally, the system combines both predictors to optimize the architecture on an edge server (which can be CPU-only) (Step 5) and deploy the optimized architecture to the target device. This workflow suffers from two major inefficiencies: heavy reliance on expert manual efforts for SuperNet development and expensive profiling for each new model and device.

On the development side, supporting fine-grained and large-scale variant generation requires complex and manual modifications to the original DNN. Existing implementations typically extend each layer with the dynamic module and implement its exploration method too as shown in Fig. 5(b). This implementation not only inflates the codebase, but also strongly relies on DNN architecture design expertise. Retiarii [79] attempts to mitigate the issue via abstractions like Mutator as shown in Fig. 5(c), which not only keeps the SuperNet code clean, but also reduces repetitive engineering efforts in implementing model exploration required by SuperNet training and architecture optimization through system-level mutator management. However, developers must still rely heavily on model-aware manual efforts in specifying Mutators.

Table 1. Profiling costs for deploying SuperNets. Latency profiled on Samsung A54 big cores and accuracy profiled on an NVIDIA RTX 3070 GPU with a batch size of 128.

Model	ResNet50	MobileNetv3
Number of variants	2×10^{13}	7×10^{22}
Latency profiling (s/variant)	0.2 ~ 1.75	0.4 ~ 3.85
Accuracy profiling (s/variant)	31 ~ 47	31 ~ 43

On the profiling side, searching for optimal variants requires performance evaluation across the vast SuperNet design space. Offline profiling is practical for prior block scaling-based methods with 10,000–20,000 model variants. However, as illustrated in Table 1, offline profiling is impractical for a SuperNet with 10^{13}–10^{22} model variants. The other way is online profiling, which incurs unacceptable online overhead in the optimization process. A common alternative is to build predictors trained from partial profiling data. For example, OFA [3] trains MLP-based predictors that encode subnet architectures and output accuracy or latency estimates as shown in Fig. 6(a). However, model-level prediction remains difficult due to the high dimensionality of the architecture space, necessitating substantial profiling to achieve sufficient accuracy. To reduce cost, operator-level latency predictors have been proposed as shown in Fig. 6(b), such as nn-Meter [77], which predicts operator-level latencies and aggregates them to estimate end-to-end latency. LitePred [16] further improves predictor robustness against out-of-distribution sampling.

Nevertheless, two fundamental issues persist. First, predictor accuracy still depends on extensive profiling, often requiring hours of operator measurements. Second, training predictors introduces its own complexity, demanding careful design choices and hyperparameter tuning. As shown in Fig. 7, existing latency or accuracy predictor building methods commonly require hours of data collection to achieve nearly usable predictors.

2.4 Opportunity

To address the development and deployment bottlenecks of SuperNet-based adaptive model deployment, we aim to automate SuperNet construction by isolating users from DNN programs and to devise an efficient method for building latency predictors. However, two challenges arise: (1) how can

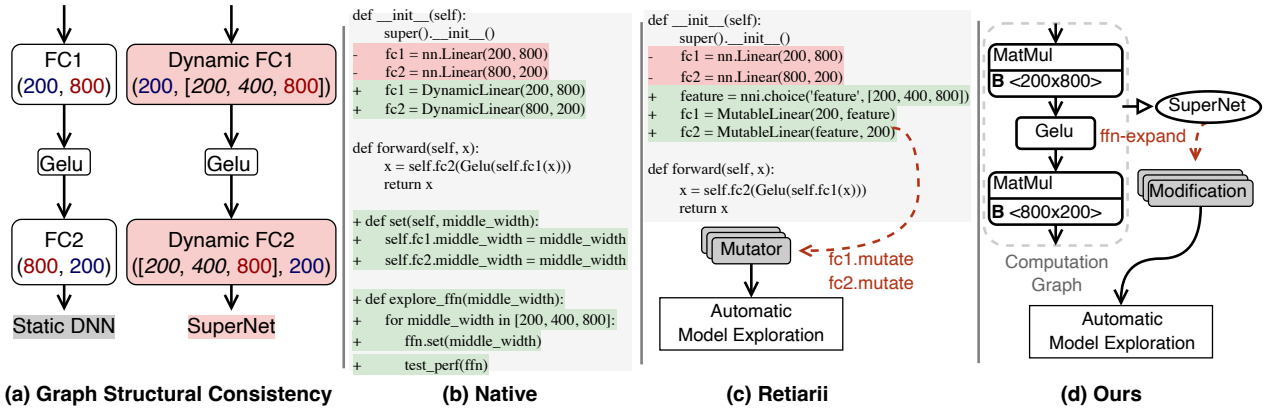


Figure 5. Illustration of the graph structural consistency property and comparison of SuperNet development methods.

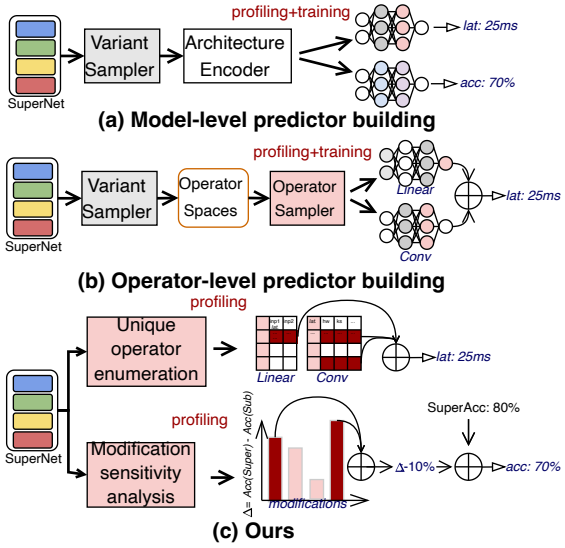


Figure 6. Comparison of predictor building methods.

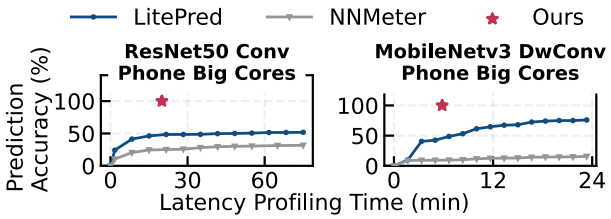


Figure 7. Existing predictor building methods rely on time-consuming profiling for collecting training data.

the framework automatically identify transformable modules and their candidate choices without user intervention? (2) how can accurate predictors be trained in the vast SuperNet design space with limited profiling data?

We identify two exploitable characteristics to address these challenges. **(1) Graph Structural Consistency:** a SuperNet is derived from a static DNN, where its largest SubNet matches the original architecture. As a result, the overall computation graph remains consistent with the original DNN (Fig. 5(a)). **(2) Cross-SubNet Architectural Similarity:** analysis of 1,000 randomly sampled SubNets (Fig. 8) shows that the number of unique operators quickly converges as sampling increases, while many operators appear repeatedly (96–98% redundancy), often at high frequencies (up to 414–3590 times). This indicates strong structural redundancy across SubNets, suggesting that the set of unique operators is small and feasible to enumerate.

These two properties create promising opportunities for overcoming existing bottlenecks:

O#1. Automated SuperNet construction. Leveraging graph structural consistency, static modules can be automatically replaced with dynamic modules during compilation as shown in Fig. 5(d), eliminating the manual effort required for SuperNet construction.

O#2. Enumerating unique operators to build learning-free predictors. Cross-SubNet similarity enables the enumeration and profiling of all unique operators, which in turn supports efficient construction of lookup-table (LUT)–based predictors as shown in Fig. 6(c), eliminating the need for costly learning-based models.

3 System Overview

AutoTailor advances adaptive model deployment on edge devices by leveraging SuperNets to generate model variants. It achieves automatic and efficient deployment through two key techniques: (1) a graph compilation method (§4.1) that automatically replaces computation graph components without manual coding, and (2) a learning-free predictor construction method (§4.2) that rapidly builds latency lookup tables (LUTs) without costly sampling or training. Given a DNN model and deployment requirements, AutoTailor automatically outputs optimized architectures that deliver state-of-the-art performance on target edge platforms.

Challenges and Approaches. Despite the opportunities, making automatic and efficient deployment practical requires addressing the following challenges.

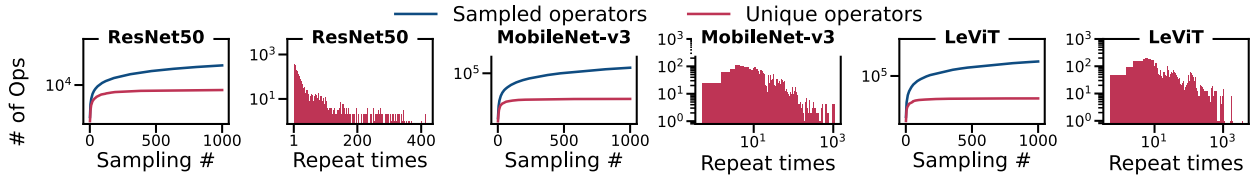


Figure 8. Illustration of cross-SubNet architectural similarity in different models.

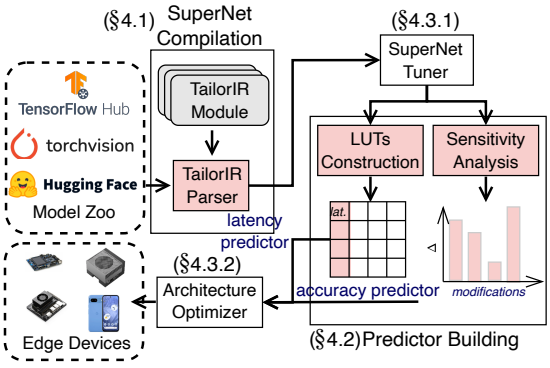


Figure 9. System architecture.

C#1. Lack of abstraction. Constructing SuperNets from static DNNs raises a key challenge: how to represent them in a scalable, model-agnostic way. Retiarii [79] wraps dynamic modules (MUTATOR) and SuperNets (MODELSPACE) around the original DNN (e.g., PyTorch *nn.Module* [51]). While simple, this approach has two drawbacks: (1) modifications are tightly coupled with the model structure, limiting scalability, and (2) reliance on the original DNN restricts cross-platform support and requires framework-specific versions.

We design TailorIR, a model-agnostic abstraction for automatic SuperNet construction (Section 4.1.1). SuperNets are organized as computation graphs of TailorIR modules, each managing its architecture and generating executable versions. This decouples modifications from structure, supports scaling, and ensures traceability. Similar to Retiarii, TailorIR enables automatic model exploration, but with broader applicability and cleaner integration.

C#2. High engineering effort in dynamic module development. Supporting diverse DNNs requires many dynamic modules, but existing systems like Retiarii make building these libraries labor-intensive.

To this end, TailorIR introduces two optimizations (Section 4.1.2): (1) bypassing non-variable modules and auto-generating executables with conversion tools; (2) using templated module designs to maximize reuse across models.

C#3. Inefficiency in enumerating unique operators. Constructing LUTs for learning-free latency prediction requires extracting parameters of unique operators (e.g., kernel size, tensor shapes). Although their number is limited (Fig. 8), operator architectures are only fixed after model-level modifications, making brute-force sampling infeasible.

To this end, we propose a rapid LUT construction method for our latency predictor (Section 4.2.1) that prunes unnecessary sampling by analyzing independent modifications. Using TailorIR's architecture traceability, AutoTailor extracts operator parameters without real inference, greatly speeding up enumeration.

C#4. High accuracy profiling cost. Accuracy predictors usually require full model profiling, which is costly since each candidate must be evaluated on GPUs. Unlike latency, accuracy cannot be decomposed to the operator level.

To this end, we propose a modification-level accuracy predictor (Section 4.2.2) based on SuperNet modification sensitivities. Relative accuracy differences between SubNets can be approximated from these sensitivities, allowing efficient prediction without full profiling.

System architecture and workflow. Fig. 9 shows the system architecture. The workflow starts with an input DNN, either from a model zoo or a user implementation, represented as a computation graph. The graph is parsed into a TailorIR-based SuperNet (Section 4.1), where static modules are replaced with TailorIR modules, and then fine-tuned to preserve SubNet accuracy (Section 4.3.1). AutoTailor next builds learning-free predictors (Section 4.2): the latency predictor profiles unique operators to build device-specific LUTs (Section 4.2.1), while the accuracy predictor samples SubNets using modification sensitivity to estimate accuracy (Section 4.2.2). Finally, given deployment requirements, AutoTailor searches for the best architecture (Section 4.3.2) and deploys it to the target device.

4 Detailed Design

4.1 Computation Graph-guided Compilation

4.1.1 TailorIR Abstraction. Fig. 10 shows the design of TailorIR using an FFN block as an example. Unlike Retiarii [79], which mainly wraps PyTorch *nn.Module*, TailorIR is an independent abstraction that directly manages architectures. Specifically: (1) each TailorIR module maintains a *Feature* describing both meta and active architectures; (2) the `transform` function updates features after modifications; (3) the `update` function ensures consistency and legality; (4) the `build` function generates an executable PyTorch module. In a SuperNet, operators, blocks, and the full model are all encapsulated as TailorIR modules, each managing its own state to ensure consistent modifications across different levels.

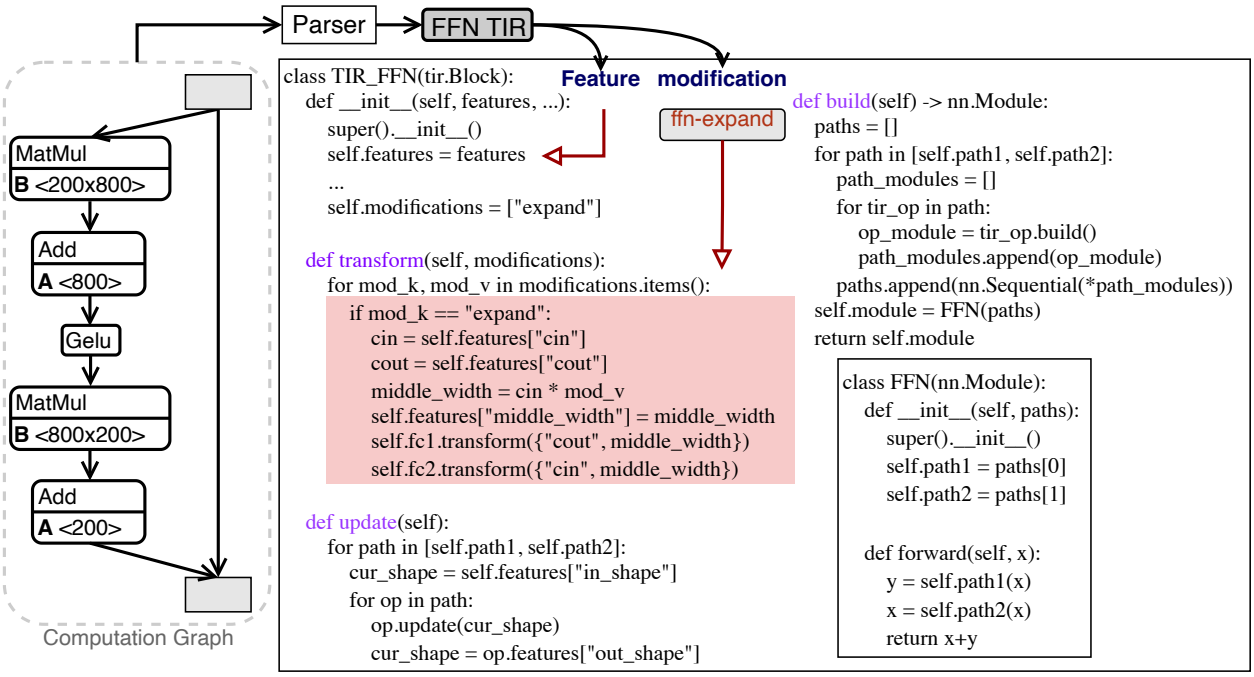


Figure 10. TailorIR overview.

Decoupled design. TailorIR separates architecture from modifications. A module's *Feature* encodes its architecture and is turned into an executable via `build`, while `transform` and `update` handle modifications. This decoupling improves readability and scalability, as new modifications only require changes to `transform` without touching `build`.

Architecture traceability. A key advantage of TailorIR is architecture traceability, improving module generality and scalability. Unlike conventional approaches that require manually hardcoding parameters, TailorIR uses a one-time deduction rule in `update`, so any `transform` automatically keeps dependent features consistent.

4.1.2 Module Development Optimizations.

Automatic construction requires static graphs covered by TailorIR modules and parser rules. Unsupported operators or architectural patterns require module or parser development; these extensions are reusable across models. Our module library reduces this effort through two optimizations:

Static module bypassing. As in existing SuperNet methods, not all layers need to be transformed into dynamic modules. In AutoTailor, since the original program is not directly accessible, we design dedicated TailorIR modules to wrap static layers (Fig. 11(a)). These modules maintain a general *Feature* with a default deduction rule in the *update* function to automatically infer attributes from inputs, which works well for non-parameterized and element-wise operators. During the `build` stage, model conversion tools (e.g., onnx2torch [12]) generate the corresponding executable PyTorch modules from the computation graph.

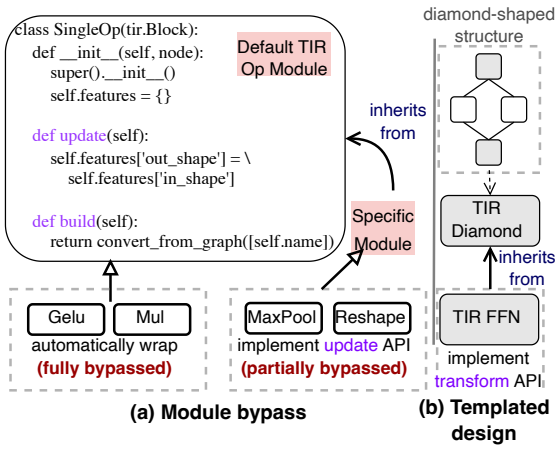


Figure 11. Optimizations in TailorIR module development.

Templated module design. TailorIR enables automatic SuperNet construction and reuses dynamic modules across diverse DNNs. Although DNNs differ in structure, many share common architectural patterns [25, 33]. To leverage this, we adopt a templated module design that encourages generic, reusable TailorIR modules. Using templated modules and the *Path* abstraction (Fig. 11(b)), TailorIR decouples structure from specific operators, allowing a single design to adapt across networks and reducing redundant engineering.

4.1.3 Automatic SuperNet Construction.

We design a bottom-up three-step parsing process that incrementally transforms an input DNN into a TailorIR-based SuperNet.

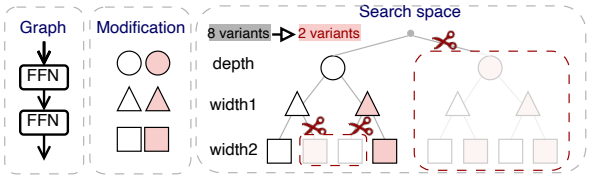


Figure 12. An example of space pruning.

Step 1: Operator parsing. The parser traverses the computation graph, mapping each operator to its TailorIR counterpart. Static operators are bypassed (Section 4.1.2). For multi-output operators, the parser traces output paths until they converge, grouping them into default TailorIR blocks. **Step 2: Block parsing.** The parser then matches subgraphs with templated TailorIR blocks based on topological patterns, replacing default blocks with structured block modules. **Step 3: Stage division.** Finally, the model is partitioned into stages (e.g., by resolution or output width), enabling higher-level modifications such as adjusting stage depth. For supported graphs, these steps construct a SuperNet without manual model rewriting.

4.2 Learning-free Predictor Building

We propose a learning-free predictor that avoids the overhead and errors of learning-based methods. For latency, AutoTailor profiles only unique operators to build operator-level LUTs, keeping costs low and eliminating generalization errors. To identify these operators efficiently, we prune the SubNet enumeration space using modification dependencies. We extend this idea to accuracy with *modification sensitivity*, which measures the impact of architectural changes and enables efficient modification-level prediction.

4.2.1 Operator-level LUT-based Latency Predictor.

AutoTailor accelerates LUT construction by pruning redundant operator enumerations through analysis of modification dependencies.

Pruning the enumeration space. As shown in Fig. 12, depth modifications determine only the number of stacked blocks within a stage. Because these blocks are structurally identical, the corresponding operator features remain unchanged regardless of depth, making such modifications independent. Similarly, modifications confined to a specific stage or block are also independent of other choices. By exploiting such independence, AutoTailor prunes redundant operator combinations from the enumeration space, thereby avoiding unnecessary profiling.

Beyond the large enumeration space, the cost of extracting operators from a single SubNet is also non-negligible. Tensor shapes are essential operator features for constructing LUTs, yet obtaining them typically requires inference of the entire SubNet due to the dynamic graph design in existing deep learning frameworks. AutoTailor alleviates this overhead by leveraging TailorIR's architectural traceability,

Table 2. Extraction cost of unique operators. In the normal setting, the cost is estimated by multiplying the number of profiling runs by the average cost of a single profiling run.

Models	normal (GPU hours)	+ space pruning (GPU hours)	+ TailorIR (CPU minutes)
ResNet50	1.25E+09	10.8	11.8
MobileNetv2	872000	0.0382	2.2
ConvNext	8.08978E+15	0.0259	4.8
LeViT	1.02E+41	57.44	46.5

which automatically infers the tensor shape of each module without full SubNet execution. As shown in Tab. 2, this optimization makes model-aware enumeration both efficient and practical.

Latency predictor building. Once the LUT is built, each device profiles its unique operators. For a given SubNet, AutoTailor extracts each operator's *Feature*, looks up its latency in the device-specific LUT, and sums them to get the model's total latency. This design ensures accuracy and easy transfer across devices. Operator fusion [46] merges multiple operators into one kernel, breaking the assumption that total latency equals the sum of individual latencies [77]. To handle this, AutoTailor extends TailorIR to support fusion: multiple operators can be combined into a single TailorIR module based on user-defined fusion rules. During LUT construction, each fused operator is built and profiled as a single module, so predictors capture the true execution cost. Section 6.4 shows that this fusion-aware approach achieves accurate latency prediction.

4.2.2 Modification-level Accuracy Predictor.

The primary goal of our accuracy predictor is to identify high-performing SubNets through approximate ranking rather than precisely predict their absolute accuracy. Fig. 13 evaluates the effectiveness of this design choice. Across the evaluated budgets, Top-5 accuracy is only 0.64% lower than Top-1 accuracy on average for ResNet50 and 0.92% lower for ViT. These results suggest that approximate ranking is sufficient to identify competitive SubNets, reducing the need for highly precise accuracy estimates.

To enable efficient ranking, we leverage the observation that all SubNets inherit weights from the same trained SuperNet and differ only in their architectural modifications. This shared foundation allows us to approximate the accuracy impact of combined modifications using the sensitivity of individual modifications, avoiding exhaustive evaluation of their combinations. Specifically, let A_0 denote the accuracy of the unmodified SuperNet and $A(S)$ the measured accuracy after applying a set of modifications S. AutoTailor profiles each modification m individually on the training dataset and estimates the accuracy of a candidate SubNet as

$$\Delta_m = A_0 - A(\{m\}), \qquad \widehat{A}(S) = A_0 - \sum_{m \in S} \Delta_m. \tag{1}$$

Here, Δ_m captures the accuracy sensitivity of modification m, while summing individual sensitivities approximates their

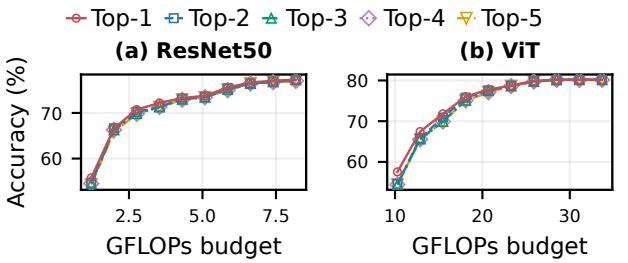


Figure 13. Accuracy of top-ranked SubNets across FLOPs budgets. Top-k denotes the minimum measured accuracy among the k highest-ranked candidates in each search.

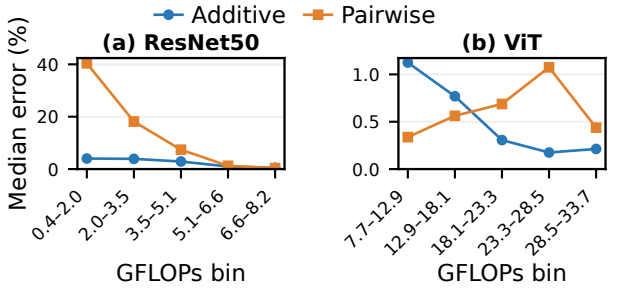


Figure 14. Median accuracy error of additive and pairwise predictors across FLOPs bins, with 100 SubNets per bin.

combined impact without explicitly modeling interactions. This additive approximation enables efficient, architecture-aware SubNet ranking without the overhead of evaluating every candidate.

Fig. 14 examines the effectiveness of our additive approximation across different computational scales. For each model, we sample 100 distinct SubNets from each of five FLOPs bins. For ResNet50, the median prediction error decreases from 4.03% to 0.49% across bins, while for ViT, it ranges from 0.18% to 1.13%.

A limitation of the additive approximation is that it ignores interactions among architectural modifications. To assess whether modeling such interactions improves prediction accuracy, we extend the predictor with a pairwise correction term, $A(\{a,b\}) - A(\{a\}) - A(\{b\}) + A_0$, for each co-occurring pair of modifications. Surprisingly, explicitly modeling pairwise interactions does not consistently improve prediction accuracy. For ResNet50, the median errors range from 40.39% to 0.35% across bins, while for ViT, they range from 0.34% to 1.08%, despite requiring 33× and 14.8× more profiling configurations, respectively. This result suggests that pairwise corrections do not necessarily capture the combined effects of architectural modifications, as higher-order interactions remain unmodeled and individual correction terms may introduce additional estimation errors. Overall, these findings support additive sensitivity as a practical approximation that balances prediction accuracy and profiling overhead.

4.3 Automatic Model Exploration

Model exploration samples SubNets from the search space and is crucial for (1) fine-tuning the SuperNet and (2) architecture optimization, where the best candidates are selected using the fine-tuned SuperNet and predictors. AutoTailor automates this by collecting modification abstractions from each TailorIR module; users only specify modifications and candidate choices in a simple configuration file:

```
title = "Example"
[arch]
blocks = ["FFNBlock"]
[var.global_vars]
resolution = [128, 160, 192, 224]
[var.stage_vars]
reduce_depth = [-3, -2, -1, 0]
[var.block_vars]
FFNBlock.expand_ratio = [2, 3, 4]
```

4.3.1 SuperNet Fine-tuning. Fine-tuning is necessary to ensure that modified SuperNets preserve competitive accuracy. AutoTailor incorporates a *Tuner* component to execute this process, providing seamless integration with existing SuperNet-based model exploration strategies [56, 62]. Given a TailorIR-represented SuperNet, the Tuner samples SubNets by applying modifications and fine-tunes them accordingly.

4.3.2 Architecture Optimization. The *Optimizer* in AutoTailor efficiently searches for the best architecture to deploy fine-tuned SuperNets on edge devices using the constructed predictors. To handle this complex task, AutoTailor uses genetic algorithms [3, 62] and accelerates the search with a beam search-based initialization for large design spaces.

5 Implementation

We implement AutoTailor on PyTorch [50] in 12500+ LoC of Python, including 4700+ LoC for the library of TailorIR modules to support all SuperNets mentioned in this paper. We use ONNX [49] and ONNX Graph Surgeon [48] tools for parsing the model. The tuner component is implemented on PyTorch DistributedDataParallel (DDP) for supporting distributed SuperNet tuning. The optimized architecture will be translated into NCNN executable format by the PNNX [78] tool for deploying on mobile phones. The AutoTailor artifact is publicly available at https://doi.org/10.5281/zenodo.22984628.

6 Evaluation

6.1 Experimental Setup

Testbed. We evaluate AutoTailor on four edge setups: a Samsung A54 smartphone (Exynos 1380) using its Big- and Little-cores, and NVIDIA Jetson Orin Nano and AGX Orin GPUs. AutoTailor runs on a local server with NVIDIA 3070 GPUs and Intel Xeon Gold 6230R CPUs, offloading SuperNet tuning to the cloud. NCNN [63] is used for on-device

inference (FP16 precision), except for ViT on Jetson devices, where PyTorch [50] performs better.

Models and datasets. Tab. 3 lists the SuperNets and their original models used in our experiments. All SuperNets support width, depth, and input modifications. We use ImageNet1K [55] for both fine-tuning and evaluating adapted models. Following CompOFA [56], each SuperNet is fine-tuned for 600 iterations (150 epochs) with 4 SubNets sampled per batch.

6.2 Adaptive Deployment Performance

We first evaluate AutoTailor's adaptive deployment performance by searching for optimal architectures across different models, edge devices, and requirements. For each case, we generate 40–50 latency constraints and report only the Pareto frontier of the resulting latency–accuracy trade-offs. Our adaptation uses our proposed learning-free predictors for providing both latency and accuracy guidance in the architecture optimization. The hyperparameter settings of the genetic search algorithm used in architecture optimization follow those in OFA [3].

Baselines. We compare against AdaptiveNet [69] and LegoDNN [23] using their official implementations. AdaptiveNet uses its released weights, while LegoDNN trains new blocks on pre-trained DNNs from TIMM [70]. For NestDNN, whose code is unavailable, we approximate its performance using Torch-Pruning [15] and denote it as NestDNN*. Pruned variants are fine-tuned for 60 epochs on ResNet50 and MobileNetV2, and 120 epochs on ConvNeXt and ViT. Since AdaptiveNet and LegoDNN do not support Transformers, we compare them only against NestDNN* for these models. We also report static DNN performance for reference.

Results. Due to space constraints, we select one representative model per runtime, pairing larger models with stronger devices and smaller models with weaker ones. As shown in Fig. 15, AutoTailor supports a wider range of latency constraints and consistently delivers higher accuracy under 92% of the evaluated latency conditions than existing methods.

On ResNet50, AutoTailor achieves an average accuracy gain of 2.27% and a 10.37% latency reduction over AdaptiveNet. On MobileNetV2, the improvements are larger, with 4.12% higher accuracy and 21.48% lower latency. These gains are most pronounced on mobile phones, which have fewer parallel resources than Jetson GPUs and thus require more compact models. AutoTailor generates such compact variants more effectively than existing methods. In contrast, LegoDNN and NestDNN* perform poorly under tight latency budgets, especially on weaker devices, due to their lack of depth modifications. For larger latency budgets (8%), AutoTailor is slightly less competitive, which we attribute to limited SuperNet training. We expect this gap can be closed with stronger tuning strategies, such as OFA's progressive training [3] (3× more iterations) or AlphaNet's self-distillation [65] (1.25× more iterations).

On Transformer-style models, AutoTailor improves accuracy by 5.12% and reduces latency by 29.62% on ConvNeXt, and achieves 3.36% higher accuracy with 15.04% lower latency on Vision Transformer compared with NestDNN*. These results demonstrate AutoTailor's effectiveness on advanced architectures and its potential for emerging systems such as vision foundation models [30] and vision–language models [41], where ConvNeXt and Vision Transformer serve as common image encoders.

We also report hardware-agnostic results by using FLOPs instead of latency (Fig. 16). Unlike hardware-aware results, the FLOPs-guided ones show different trends, underscoring the need for accurate hardware-specific predictors. Across most FLOPs constraints, AutoTailor still achieves state-of-the-art performance. For Vision Transformer, width pruning performs well, likely due to its higher architectural complexity, which may benefit from more SuperNet fine-tuning.

Overhead analysis. Next, we analyze the overhead of applying SuperNets. Fig. 17 compares the extra cost from AutoTailor's larger design space with AdaptiveNet and a scaled version (AdaptiveNet-scaled) that matches our search space. The official AdaptiveNet validates accuracy on only a data subset, which reduces reliability. For fairness, we validate all 10,609 variants and include this cost in our analysis.

Our sensitivity-based predictor needs only one hour of profiling. Latency profiling takes 12.5× longer than AdaptiveNet but only 20 minutes in absolute time, which is practical. AdaptiveNet requires 60 training iterations with 3 sampled blocks per batch, while our method incurs 3.3× more iterations to fine-tune SuperNets. The AdaptiveNet-scaled comparison highlights our better scalability, even without input modifications that would further raise its overhead.

Table 3. Illustration of SuperNets in this paper.

Original DNN	# of Modifications	# of Variants
ResNet50 [24]	100	2×10^{13}
MobileNetv2 [58]	85	8×10^{10}
MobileNetv3 [27]	160	7×10^{22}
ProxylessNet [4]	166	6×10^{23}
ConvNext-Tiny [42]	163	2×10^{23}
ViT-Base [11]	103	1×10^{12}
LeViT [20]	324	1×10^{45}

6.3 Effectiveness of Computation-graph Guided Compilation

Engineering Effort Reduction. We quantify the engineering effort required for SuperNet construction by measuring lines of code (LoC) changes relative to the original DNN using `git diff`. We evaluate four SuperNets (ResNet50, MobileNetV3, ProxylessNet, and LeViT), using their official

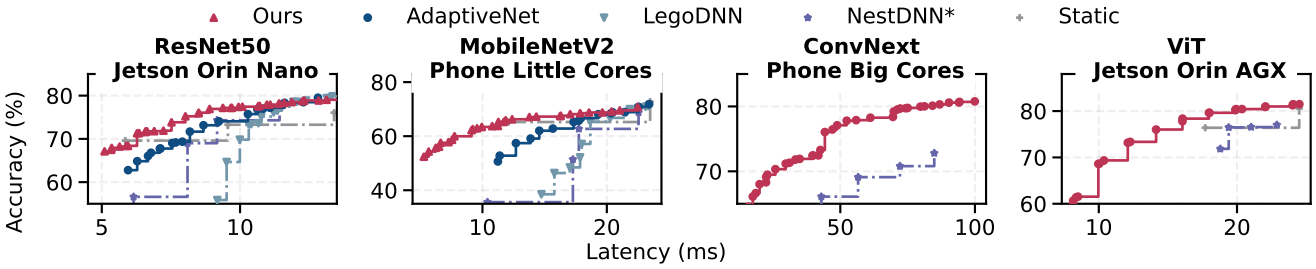


Figure 15. Performance comparison of hardware-aware adaptive model deployment.

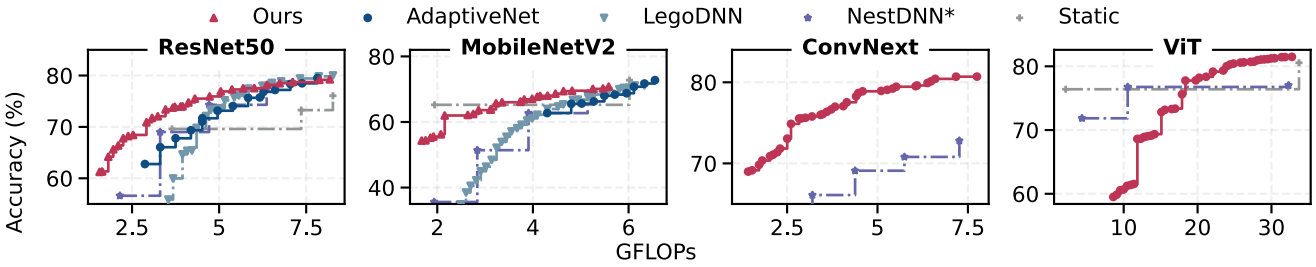


Figure 16. Performance comparison of hardware-agnostic adaptive model deployment.

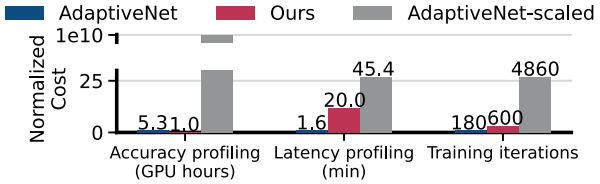


Figure 17. Overhead analysis of ResNet50 adaptive model on phone big cores.

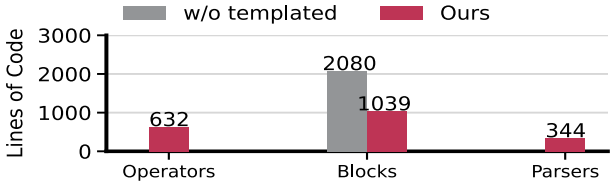


Figure 19. Analysis of TailorIR Modules.

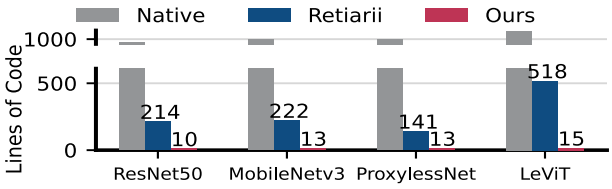


Figure 18. Comparison of LoC in constructing SuperNets.

OFA [3] and ElasticViT [62] implementations as native baselines. The Retiarii baseline is implemented by us using their open-source framework, NNI [45]. With AutoTailor, users only need to provide a simple configuration file specifying modifications (Section 4.3). As shown in Fig. 18, AutoTailor reduces LoC by 10.8–34.5× compared with Retiarii, thanks to automatic graph compilation. While Retiarii eases model exploration, it still requires manual replacement of static modules with mutable ones, which demands both detailed architecture knowledge and significant engineering. In contrast, AutoTailor eliminates such model-aware coding by letting users define SuperNets through concise, easy-to-write configuration files.

Effectiveness of Development Optimization. Fig. 19 analyzes the LoC distribution of core functions, including `build`, `transform`, `update`, and `init` functions, across TailorIR modules developed for the SuperNets of Tab. 3. About 51% of the effort goes into 12 templated blocks, with nearly 50% saved through LoC reuse. Operator development takes 632 LoC for 21 operators (10 partially bypassed), out of 28 total operators (7 fully bypassed). Parser development accounts for about 20% of the effort, as it maps sub-graphs to block templates. To simplify and avoid conflicts, users can specify block templates directly in the config file (Sec. 4.3).

6.4 Effectiveness of Learning-free Predictors

Baselines. For latency prediction, we compare AutoTailor with LitePred [16], extending both to support additional operators. Each SuperNet uses 2000 sampled SubNets to build predictors. For accuracy prediction, we compare with OFA's MLP-based method [3].

Metrics. For latency prediction, we report error rates at the 10% level on 1000 randomly sampled SubNets following [16, 77]. For accuracy prediction, we use top-5 ranking accuracy: 100 groups of 200 candidates are ranked by predicted accuracy. We adopt ranking accuracy instead of absolute values to better match the optimization process, which selects the best SubNets within each group.

Predictor profiling cost reduction. As shown in Fig. 7, existing learning-based predictors struggle with accuracy and efficiency. Our learning-free method profiles all unique

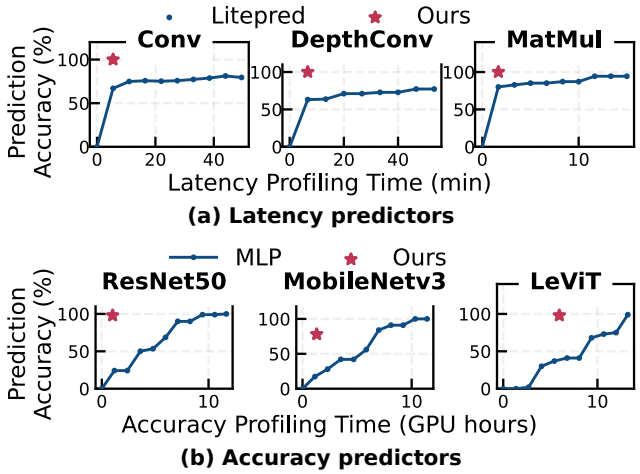


Figure 20. Profiling cost of (a) latency predictors for LeViT operators and (b) accuracy predictors across models.

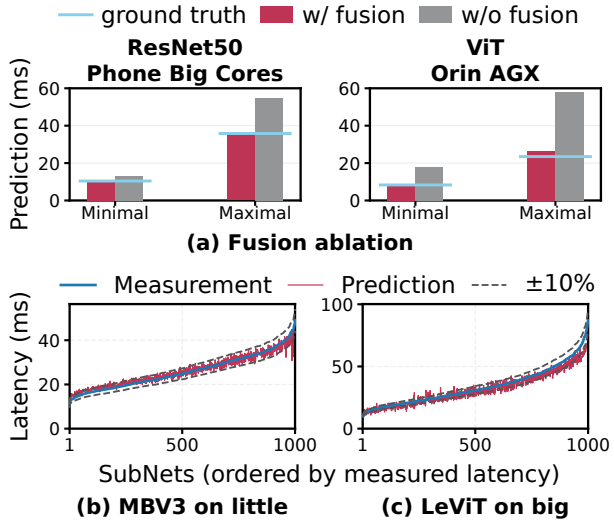


Figure 21. Model-level latency prediction: (a) fusion ablation; (b,c) measured and predicted latency for 1000 SubNets per setting on Samsung A54. Dashed lines mark ±10% of measured latency.

operators for precise latency estimation, taking only 20 minutes for ResNet50's convolutions and 40 minutes for MobileNetV3's depth-wise convolutions, which are their main latency contributors. As shown in Tab. 3, LeViT has the most model variants, making predictor construction challenging. Fig. 20(a) compares our method with LitePred [16] on three major operators. Our method reduces latency-predictor profiling time by 8–9× compared with the largest LitePred budgets shown, while achieving higher prediction accuracy. For accuracy prediction, Fig. 20(b) shows that our method reduces predictor construction time by 2.2–11.5× compared to the full MLP profiling runs shown while achieving 78–98% top-5 ranking accuracy. Our method offers an efficient alternative for rapidly constructing predictors.

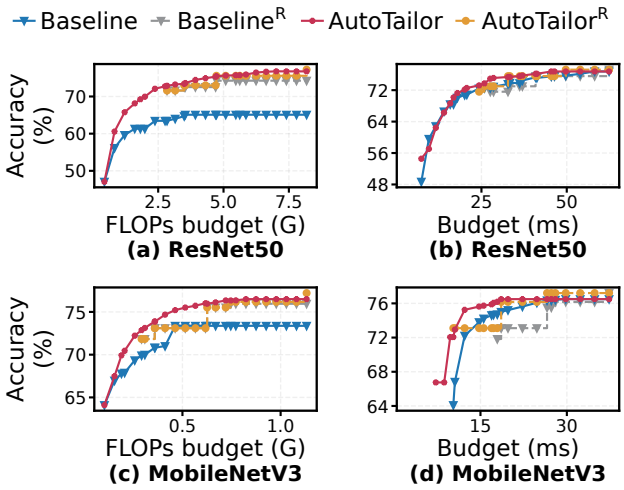


Figure 22. Ablation study on computation-quality trade-offs. Baselines: MLP in (a,c) and MLP+LitePred in (b,d). R denotes the restricted design space.

Model-level latency prediction. We compare predicted and measured latencies of the smallest and largest SubNets, with and without fusion. As shown in Fig. 21(a), ignoring fusion causes errors up to 1.47×, especially in larger SubNets with more fused operators. While perfect accuracy is difficult, our fusion-aware method keeps deviations within an acceptable range. We further choose MobileNetV3 and LeViT to evaluate memory-intensive execution. We evaluate predicted and measured model-level latency for 1000 randomly sampled SubNets of each model on the Samsung A54's big and little CPU cores. Fig. 21(b,c) shows that most predictions lie within the ±10% bounds in both settings.

6.5 Ablation Study

To evaluate how AutoTailor's design space and predictors contribute to better computation-quality trade-offs, we ablate both components in Fig. 22. The full space provides 44.9% and 22.5% wider observed FLOPs ranges than the five restricted presets (R) for ResNet50 and MobileNetV3, respectively. Within the full space, we compare AutoTailor's accuracy predictor with MLP in (a,c), and its accuracy-latency predictor pair with MLP+LitePred in (b,d). Baseline^R often outperforms Baseline, suggesting that searching the larger space amplifies baseline ranking errors and makes reliable predictors essential to realizing its benefits. Our predictors improve accuracy by 10.43%/2.90% at matched FLOPs and 11.36%/3.14% at matched summed A54 operator latency for ResNet50/MobileNetV3, respectively.

6.6 Generality Case Studies

To evaluate the generality of AutoTailor, we extend the framework to support object detection and autoregressive language modeling.

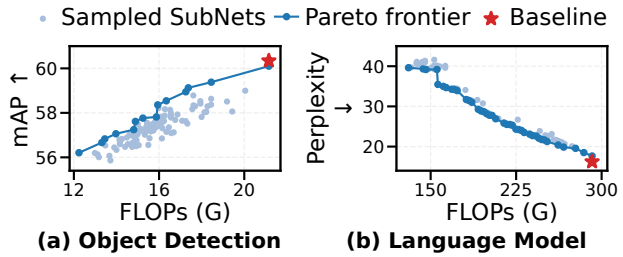


Figure 23. Computation–quality trade-offs for object detection and language modeling.

Setup. (1) Object detection. We use the HardHat [43] dataset, split into 4000 training and 1000 validation images. We construct the detector SuperNet by combining a ResNet-style TailorIR backbone with a YOLOv8 [64] detector. Only the backbone is transformable, while the neck and head architectures are fixed. We jointly train the backbone, neck, and heads for 70 additional epochs at a resolution of (640×640). Each batch trains the largest and smallest SubNets together with two randomly sampled SubNets. We measure detection quality using mean average precision (mAP) [13]. (2) Autoregressive language modeling. We train a GPT-2 [53] SuperNet with 4–12 Transformer blocks and feed-forward expansion ratios of 2–4 on WikiText-103 [44] for 20000 iterations following nanoGPT [31]. We measure language-model quality using perplexity [2]. For both tasks, we select SubNets using CompOFA [56] presets and FLOPs-stratified random sampling. The baseline for each task is the corresponding full-size model fine-tuned on the same dataset using the same training procedure.

Results. Fig. 23 shows the computation-quality trade-offs for both tasks. (1) Object detection. The Pareto frontier spans 12.24–21.15 GFLOPs and 56.21–60.10% mAP. (2) Autoregressive language modeling. The GPT-2 SubNets span 130.6–291.6 GFLOPs per sequence and achieve perplexities of 17.66–41.62. Object detection reuses existing parser support and block templates with zero additional LoC; GPT-2 requires 134 LoC for parser support and 261 LoC for templated blocks (395 LoC total). Together, these results demonstrate AutoTailor's extensibility across diverse model architectures and tasks with modest engineering effort.

7 Related Work

SuperNet Architectures and Training. Weight-sharing SuperNets support adaptive architectures across vision [3, 8, 57, 62, 75], speech [59], and language tasks [5, 6, 26, 67]. Industrial-scale foundation models also adopt this paradigm, including Google's Gemma 3n [10, 19] and NVIDIA's Nemotron Elastic [61]. Their high training cost motivates algorithmic optimizations, including search-space restriction [1, 56, 74, 76] and knowledge distillation [28, 62, 65, 66, 72], and system-level parallelization [73, 81].

Adaptive Inference Systems. Using model variants as a control knob is a widely adopted strategy for meeting diverse edge deployment requirements [14, 23, 69] and adapting to fluctuating cloud serving workloads [32, 54]. For edge deployment, frameworks support model exploration and porting [22, 79], while adaptation mechanisms use early exits [36] or pruning-derived model variants [14, 23]. For cloud serving, systems optimize model selection, resource allocation, and request scheduling to meet latency and accuracy requirements [21, 54]. More recently, SuperServe [32] extends adaptive serving to weight-sharing SuperNets by dynamically selecting SubNets in response to changing workloads. AutoTailor complements these systems by automating SuperNet construction and enabling efficient SubNet selection, thereby is also extensible to support adaptive cloud serving.

Optimizations for On-Device ML Inference. Beyond model architecture, inference efficiency can be further improved through system-level optimizations, including accelerator utilization [7, 30, 35, 71], runtime memory management [38], and kernel generation and fusion [9, 39, 46]. While these techniques optimize the execution of a given model on a target device, AutoTailor focuses on tailoring model architectures to device-specific characteristics. We therefore hold the inference execution stack fixed for each model–device pair, as these system-level optimizations are orthogonal to AutoTailor.

8 Conclusion

AutoTailor makes SuperNet-based adaptive deployment practical across diverse edge devices. By automating SuperNet construction and SubNet selection with learning-free performance prediction, AutoTailor reduces the cost and complexity of architecture specialization. Across evaluated models and devices, AutoTailor reduces construction code by 10.8–34.5× and reduces predictor profiling time by 2.2–11.5×, while achieving better latency–accuracy trade-offs than state-of-the-art approaches.

Acknowledgments

We thank our shepherd and reviewers for their invaluable feedback. This work was supported in part by the National Natural Science Foundation of China under Grant 62232004, Jiangsu Provincial Frontier Technology Research and Development Program under Grant BF2024070, Shenzhen Science and Technology Program under Grant KJZD20240903100814018, Jiangsu Provincial Key Laboratory of Network and Information Security under Grant BM2003201, 2024 Nansha District Science and Technology Program under Grant 2024ZD005, and the Science and Technology Development Fund of Macau (0061/2025/RIB2, 0107/2024/RIA2). We also thank the Big Data Computing Center of Southeast University for providing the experiment environment and computing facility.

References

[1] Aditya Annavajjala, Alind Khare, Animesh Agrawal, Igor Fedorov, Hugo Latapie, Myungjin Lee, and Alexey Tumanov. 2024. DєpS: Delayed ϵ-Shrinking for Faster Once-for-All Training. In *Proceedings of the European Conference on Computer Vision (ECCV '24)*. 315–331.

[2] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. 2003. A Neural Probabilistic Language Model. *Journal of Machine Learning Research* 3 (2003), 1137–1155.

[3] Han Cai, Chuang Gan, Tianzhe Wang, Zhekai Zhang, and Song Han. 2020. Once-for-All: Train One Network and Specialize it for Efficient Deployment. In *International Conference on Learning Representations (ICLR '20)*.

[4] Han Cai, Ligeng Zhu, and Song Han. 2019. ProxylessNAS: Direct Neural Architecture Search on Target Task and Hardware. In *International Conference on Learning Representations (ICLR '19)*.

[5] Ruisi Cai, Saurav Muralidharan, Greg Heinrich, Hongxu Yin, Zhangyang Wang, Jan Kautz, and Pavlo Molchanov. 2024. Flextron: Many-in-One Flexible Large Language Model. In *Proceedings of the International Conference on Machine Learning (ICML '24)*. 5298–5311.

[6] Ruisi Cai, Saurav Muralidharan, Hongxu Yin, Zhangyang Wang, Jan Kautz, and Pavlo Molchanov. 2025. LLaMaFlex: Many-in-one LLMs via Generalized Pruning and Weight Sharing. In *International Conference on Learning Representations (ICLR '25)*.

[7] Le Chen, Dahu Feng, Erhu Feng, Yingrui Wang, Rong Zhao, Yubin Xia, Pinjie Xu, and Haibo Chen. 2025. Characterizing Mobile SoC for Accelerating Heterogeneous LLM Inference. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP '25)*. 359–374.

[8] Minghao Chen, Houwen Peng, Jianlong Fu, and Haibin Ling. 2021. AutoFormer: Searching Transformers for Visual Recognition. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV '21)*. 12270–12280.

[9] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: an automated end-to-end optimizing compiler for deep learning. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI '18)*. 579–594.

[10] Devvrit, Sneha Kudugunta, Aditya Kusupati, Tim Dettmers, Kaifeng Chen, Inderjit Dhillon, Yulia Tsvetkov, Hannaneh Hajishirzi, Sham Kakade, Ali Farhadi, and Prateek Jain. 2024. MatFormer: Nested Transformer for Elastic Inference. In *Advances in Neural Information Processing Systems (NeurIPS '24)*. 140535–140564.

[11] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. 2021. An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale. In *International Conference on Learning Representations (ICLR '21)*.

[12] ENOT developers, Igor Kalgin, Arseny Yanchenko, Pyoter Ivanov, and Alexander Goncharenko. 2021. onnx2torch: Convert ONNX models to PyTorch. https://enot.ai/ Accessed: 2026-10-14.

[13] Mark Everingham, Luc Van Gool, Chris K. I. Williams, John Winn, and Andrew Zisserman. 2010. The PASCAL Visual Object Classes (VOC) Challenge. *International Journal of Computer Vision* 88, 2 (2010), 303–338.

[14] Biyi Fang, Xiao Zeng, and Mi Zhang. 2018. NestDNN: Resource-Aware Multi-Tenant On-Device Deep Learning for Continuous Mobile Vision. In *Proceedings of the ACM International Conference on Mobile Computing and Networking (MobiCom '18)*. 115–127.

[15] Gongfan Fang, Xinyin Ma, Mingli Song, Michael Bi Mi, and Xinchao Wang. 2023. DepGraph: Towards Any Structural Pruning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR '23)*. 16091–16101.

[16] Chengquan Feng, Li Lyna Zhang, Yuanchi Liu, Jiahang Xu, Chengruidong Zhang, Zhiyuan Wang, Ting Cao, Mao Yang, and Haisheng Tan. 2024. LitePred: Transferable and Scalable Latency Prediction for Hardware-Aware Neural Architecture Search. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI '24)*. 1463–1477.

[17] Francesco Fraternali, Bharathan Balaji, Dhiman Sengupta, Dezhi Hong, and Rajesh K. Gupta. 2020. Ember: energy management of batteryless event detection sensors with deep reinforcement learning. In *Proceedings of the ACM Conference on Embedded Networked Sensor Systems (SenSys '20)*. 503–516.

[18] Graham Gobieski, Brandon Lucia, and Nathan Beckmann. 2019. Intelligence Beyond the Edge: Inference on Intermittent Embedded Systems. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*. 199–213.

[19] Google DeepMind. 2025. Gemma 3n. https://deepmind.google/models/gemma/gemma-3n/ Accessed: 2026-10-14.

[20] Benjamin Graham, Alaaeldin El-Nouby, Hugo Touvron, Pierre Stock, Armand Joulin, Hervé Jégou, and Matthijs Douze. 2021. LeViT: A Vision Transformer in ConvNet's Clothing for Faster Inference. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV '21)*. 12259–12269.

[21] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. 2020. Serving DNNs like Clockwork: Performance Predictability from the Bottom Up. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*. 443–462.

[22] Peizhen Guo, Bo Hu, and Wenjun Hu. 2021. Mistify: Automating DNN Model Porting for On-Device Inference at the Edge. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI '21)*. 705–719.

[23] Rui Han, Qinglong Zhang, Chi Harold Liu, Guoren Wang, Jian Tang, and Lydia Y. Chen. 2021. LegoDNN: block-grained scaling of deep neural networks for mobile vision. In *Proceedings of the ACM International Conference on Mobile Computing and Networking (MobiCom '21)*. 406–419.

[24] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR '16)*. 770–778.

[25] Zicong Hong, Jian Lin, Song Guo, Sifu Luo, Wuhui Chen, Roger Wattenhofer, and Yue Yu. 2024. Optimus: Warming Serverless ML Inference via Inter-Function Model Transformation. In *Proceedings of the European Conference on Computer Systems (EuroSys '24)*. 1039–1053.

[26] Lu Hou, Zhiqi Huang, Lifeng Shang, Xin Jiang, Xiao Chen, and Qun Liu. 2020. DynaBERT: Dynamic BERT with Adaptive Width and Depth. In *Advances in Neural Information Processing Systems (NeurIPS '20)*.

[27] Andrew Howard, Ruoming Pang, Hartwig Adam, Quoc V. Le, Mark Sandler, Bo Chen, Weijun Wang, Liang-Chieh Chen, Mingxing Tan, Grace Chu, Vijay Vasudevan, and Yukun Zhu. 2019. Searching for MobileNetV3. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV '19)*. 1314–1324.

[28] Jeimin Jeon, Youngmin Oh, Junghyup Lee, Donghyeon Baek, Dohyung Kim, Chanho Eom, and Bumsub Ham. 2025. Subnet-Aware Dynamic Supernet Training for Neural Architecture Search. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR '25)*. 30137–30146.

[29] Shiqi Jiang, Zhiqi Lin, Yuanchun Li, Yuanchao Shu, and Yunxin Liu. 2021. Flexible high-resolution object detection on edge devices with tunable latency. In *Proceedings of the ACM International Conference on Mobile Computing and Networking (MobiCom '21)*. 559–572.

[30] Chanyoung Jung, Jeho Lee, Gunjoong Kim, Jiwon Kim, Seonghoon Park, and Hojung Cha. 2025. ARIA: Optimizing Vision Foundation

Model Inference on Heterogeneous Mobile Processors for Augmented Reality. In *Proceedings of the ACM International Conference on Mobile Systems, Applications, and Services (MobiSys '25)*. 249–262.

[31] Andrej Karpathy. 2023. nanoGPT. https://github.com/karpathy/nanoGPT Accessed: 2026-10-14.

[32] Alind Khare, Dhruv Garg, Sukrit Kalra, Snigdha Grandhi, Ion Stoica, and Alexey Tumanov. 2025. SuperServe: Fine-Grained Inference Serving for Unpredictable Workloads. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI '25)*. 739–758.

[33] Fan Lai, Yinwei Dai, Harsha V. Madhyastha, and Mosharaf Chowdhury. 2023. ModelKeeper: Accelerating DNN Training via Automated Training Warmup. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI '23)*. 769–785.

[34] Jeho Lee, Chanyoung Jung, Jiwon Kim, and Hojung Cha. 2024. Panopticus: Omnidirectional 3D Object Detection on Resource-constrained Edge Devices. In *Proceedings of the ACM International Conference on Mobile Computing and Networking (MobiCom '24)*. 1207–1221.

[35] Jeho Lee, Gunjoong Kim, Chanyoung Jung, Jaehee Kim, Seonghoon Park, and Hojung Cha. 2026. viNPU: Optimizing Vision Transformer Inference on Mobile NPUs. In *Proceedings of the European Conference on Computer Systems (EuroSys '26)*. 1812–1828.

[36] En Li, Liekang Zeng, Zhi Zhou, and Xu Chen. 2020. Edge AI: On-Demand Accelerating Deep Neural Network Inference via Edge Computing. *IEEE Transactions on Wireless Communications* 19, 1 (2020), 447–457.

[37] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. 2017. Pruning Filters for Efficient ConvNets. In *International Conference on Learning Representations (ICLR '17)*.

[38] Xiangyu Li, Yuanchun Li, Yuanzhe Li, Ting Cao, and Yunxin Liu. 2024. FlexNN: Efficient and Adaptive DNN Inference on Memory-Constrained Edge Devices. In *Proceedings of the ACM International Conference on Mobile Computing and Networking (MobiCom '24)*. 709–723.

[39] Rendong Liang, Ting Cao, Jicheng Wen, Manni Wang, Yang Wang, Jianhua Zou, and Yunxin Liu. 2022. Romou: rapidly generate high-performance tensor kernels for mobile GPUs. In *Proceedings of the ACM International Conference on Mobile Computing and Networking (MobiCom '22)*. 487–500.

[40] Neiwen Ling, Xuan Huang, Zhihe Zhao, Nan Guan, Zhenyu Yan, and Guoliang Xing. 2022. BlastNet: Exploiting Duo-Blocks for Cross-Processor Real-Time DNN Inference. In *Proceedings of the ACM Conference on Embedded Networked Sensor Systems (SenSys '22)*. 91–105.

[41] Haotian Liu, Chunyuan Li, Qingyang Wu, and Yong Jae Lee. 2023. Visual instruction tuning. In *Advances in Neural Information Processing Systems (NeurIPS '23)*. 34892–34916.

[42] Zhuang Liu, Hanzi Mao, Chao-Yuan Wu, Christoph Feichtenhofer, Trevor Darrell, and Saining Xie. 2022. A ConvNet for the 2020s. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR '22)*. 11966–11976.

[43] Make ML. 2024. Hard Hat Dataset. https://makeml.app/datasets/hard-hat-workers Dataset mirrored at https://huggingface.co/datasets/Voxel51/hard-hat-detection; Accessed: 2026-10-14.

[44] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. 2017. Pointer Sentinel Mixture Models. In *International Conference on Learning Representations (ICLR '17)*.

[45] Microsoft NNI Team. 2022. NNI: Neural Network Intelligence. https://github.com/microsoft/nni Accessed: 2026-10-14.

[46] Wei Niu, Jiexiong Guan, Yanzhi Wang, Gagan Agrawal, and Bin Ren. 2021. DNNFusion: accelerating deep neural networks execution with advanced operator fusion. In *Proceedings of the ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21)*. 883–898.

[47] NVIDIA. 2024. NVIDIA Jetson Orin. https://www.nvidia.com/en-sg/autonomous-machines/embedded-systems/jetson-orin/ Accessed: 2026-10-14.

[48] NVIDIA Team. 2021. NVIDIA ONNX GraphSurgeon. https://docs.nvidia.com/deeplearning/tensorrt/latest/api/onnx-graphsurgeon-api.html Accessed: 2026-10-14.

[49] ONNX Team. 2021. Open Neural Network Exchange. https://onnx.ai/ Accessed: 2026-10-14.

[50] PyTorch Team. 2016. PyTorch Deep Learning Framework. https://pytorch.org/ Accessed: 2026-10-14.

[51] PyTorch Team. 2025. PyTorch nn.Module. https://docs.pytorch.org/docs/stable/generated/torch.nn.Module.html Accessed: 2026-10-14.

[52] Qualcomm. 2007. Snapdragon Overview. https://www.qualcomm.com/snapdragon/overview Accessed: 2026-10-14.

[53] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. *Language Models are Unsupervised Multitask Learners*. Technical Report. OpenAI. https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf

[54] Francisco Romero, Qian Li, Neeraja J. Yadwadkar, and Christos Kozyrakis. 2021. INFaaS: Automated Model-less Inference Serving. In *Proceedings of the USENIX Annual Technical Conference (ATC '21)*. 397–411.

[55] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael S. Bernstein, Alexander C. Berg, and Li Fei-Fei. 2015. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision* 115, 3 (2015), 211–252.

[56] Manas Sahni, Shreya Varshini, Alind Khare, and Alexey Tumanov. 2021. CompOFA: Compound Once-For-All Networks for Faster Multi-Platform Deployment. In *International Conference on Learning Representations (ICLR '21)*.

[57] Yuiko Sakuma, Masato Ishii, and Takuya Narihira. 2023. DetOFA: Efficient Training of Once-for-All Networks for Object Detection using Path Filter. In *Proceedings of the IEEE/CVF International Conference on Computer Vision Workshops (ICCVW '23)*. 1325–1334.

[58] Mark Sandler, Andrew G. Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. MobileNetV2: Inverted Residuals and Linear Bottlenecks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR '18)*. 4510–4520.

[59] Yuan Shangguan, Haichuan Yang, Danni Li, Chunyang Wu, Yassir Fathullah, Dilin Wang, Ayushi Dalmia, Raghuraman Krishnamoorthi, Ozlem Kalinli, Junteng Jia, Jay Mahadeokar, Xin Lei, Mike Seltzer, and Vikas Chandra. 2024. TODM: Train Once Deploy Many Efficient Supernet-Based RNN-T Compression For On-Device ASR Models. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP '24)*. 10216–10220.

[60] STMicroelectronics. 2007. STM32 Microcontrollers (MCUs). https://www.st.com/en/microcontrollers-microprocessors/stm32-32-bit-arm-cortex-mcus.html Accessed: 2026-10-14.

[61] Ali Taghibakhshi, Sharath Turuvekere Sreenivas, Saurav Muralidharan, Ruisi Cai, Marcin Chochowski, Ameya Sunil Mahabaleshwarkar, Yoshi Suhara, Oluwatobi Olabiyi, Daniel Korzekwa, Mostofa Patwary, Mohammad Shoeybi, Jan Kautz, Bryan Catanzaro, Ashwath Aithal, Nima Tajbakhsh, and Pavlo Molchanov. 2025. Nemotron Elastic: Towards Efficient Many-in-One Reasoning LLMs. arXiv:2511.16664

[62] Chen Tang, Li Lyna Zhang, Huiqiang Jiang, Jiahang Xu, Ting Cao, Quanlu Zhang, Yuqing Yang, Zhi Wang, and Mao Yang. 2023. ElasticViT: Conflict-aware Supernet Training for Deploying Fast Vision Transformer on Diverse Mobile Devices. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV '23)*. 5829–5840.

[63] Tencent Team. 2019. Tencent NCNN. https://github.com/Tencent/ncnn/ Accessed: 2026-10-14.

[64] Ultralytics. 2023. Ultralytics YOLOv8. https://github.com/ultralytics/ultralytics Accessed: 2026-10-14.

[65] Dilin Wang, Chengyue Gong, Meng Li, Qiang Liu, and Vikas Chandra. 2021. AlphaNet: Improved Training of Supernets with Alpha-Divergence. In *Proceedings of the International Conference on Machine Learning (ICML '21)*. 10760–10771.

[66] Dilin Wang, Meng Li, Chengyue Gong, and Vikas Chandra. 2021. AttentiveNAS: Improving Neural Architecture Search via Attentive Sampling. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR '21)*. 6418–6427.

[67] Hanrui Wang, Zhanghao Wu, Zhijian Liu, Han Cai, Ligeng Zhu, Chuang Gan, and Song Han. 2020. HAT: Hardware-Aware Transformers for Efficient Natural Language Processing. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL '20)*. 7675–7688.

[68] Xubin Wang, Zhiqing Tang, Jianxiong Guo, Tianhui Meng, Chenhao Wang, Tian Wang, and Weijia Jia. 2025. Empowering Edge Intelligence: A Comprehensive Survey on On-Device AI Models. *Comput. Surveys* 57, 9, Article 228 (2025), 39 pages.

[69] Hao Wen, Yuanchun Li, Zunshuai Zhang, Shiqi Jiang, Xiaozhou Ye, Ye Ouyang, Yaqin Zhang, and Yunxin Liu. 2023. AdaptiveNet: Post-deployment Neural Architecture Adaptation for Diverse Edge Environments. In *Proceedings of the ACM International Conference on Mobile Computing and Networking (MobiCom '23)*. 28:1–28:17.

[70] Ross Wightman. 2019. PyTorch Image Models. https://github.com/rwightman/pytorch-image-models Accessed: 2026-10-14.

[71] Daliang Xu, Hao Zhang, Liming Yang, Ruiqi Liu, Gang Huang, Mengwei Xu, and Xuanzhe Liu. 2025. Fast On-device LLM Inference with NPUs. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '25)*. 445–462.

[72] Jin Xu, Xu Tan, Kaitao Song, Renqian Luo, Yichong Leng, Tao Qin, Tie-Yan Liu, and Jian Li. 2022. Analyzing and Mitigating Interference in Neural Architecture Search. In *Proceedings of the International Conference on Machine Learning (ICML '22)*. 24646–24662.

[73] Ying Xu, Long Cheng, Xuyi Cai, Xiaohan Ma, Weiwei Chen, Lei Zhang, and Ying Wang. 2023. Efficient Supernet Training Using Path Parallelism. In *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture (HPCA '23)*. 1249–1261.

[74] Shan You, Tao Huang, Mingmin Yang, Fei Wang, Chen Qian, and Changshui Zhang. 2020. GreedyNAS: Towards Fast One-Shot NAS With Greedy Supernet. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR '20)*. 1996–2005.

[75] Jiahui Yu and Thomas S. Huang. 2019. Universally Slimmable Networks and Improved Training Techniques. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV '19)*. 1803–1811.

[76] Jiahui Yu, Pengchong Jin, Hanxiao Liu, Gabriel Bender, Pieter-Jan Kindermans, Mingxing Tan, Thomas S. Huang, Xiaodan Song, Ruoming Pang, and Quoc Le. 2020. BigNAS: Scaling up Neural Architecture Search with Big Single-Stage Models. In *Proceedings of the European Conference on Computer Vision (ECCV '20)*. 702–717.

[77] Li Lyna Zhang, Shihao Han, Jianyu Wei, Ningxin Zheng, Ting Cao, Yuqing Yang, and Yunxin Liu. 2021. nn-Meter: Towards Accurate Latency Prediction of Deep-Learning Model Inference on Diverse Edge Devices. In *Proceedings of the ACM International Conference on Mobile Systems, Applications, and Services (MobiSys '21)*. 81–93.

[78] Nihui Zuo Zhang. 2023. PNNX: PyTorch Neural Network eXchange. https://github.com/pnnx/pnnx Accessed: 2026-10-14.

[79] Quanlu Zhang, Zhenhua Han, Fan Yang, Yuge Zhang, Zhe Liu, Mao Yang, and Lidong Zhou. 2020. Retiarii: A Deep Learning Exploratory-Training Framework. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*. 919–936.

[80] Wuyang Zhang, Zhezhi He, Luyang Liu, Zhenhua Jia, Yunxin Liu, Marco Gruteser, Dipankar Raychaudhuri, and Yanyong Zhang. 2021. Elf: accelerate high-resolution mobile deep vision with content-aware parallel offloading. In *Proceedings of the ACM International Conference on Mobile Computing and Networking (MobiCom '21)*. 201–214.

[81] Shixiong Zhao, Fanxin Li, Xusheng Chen, Tianxiang Shen, Li Chen, Sen Wang, Nicholas Zhang, Cheng Li, and Heming Cui. 2022. NASPipe: High Performance and Reproducible Pipeline Parallel Supernet Training via Causal Synchronous Parallelism. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22)*. 374–387.