STRESNET & STYOLO : A New Family of Compact Classification and Object Detection Models for MCUs

Sudhakar Sah†, Ravish Kumar†
† STMicroelectronics

Abstract

Recent advancements in lightweight neural networks have significantly improved the efficiency of deploying deep learning models on edge hardware. However, most existing architectures still compromise accuracy for latency, which limits their applicability on MCU/NPU-based devices. In this work, we introduce two new model families — STResNet for image classification and STYOLO for object detection — jointly optimized for accuracy, efficiency, and memory footprint on resource-constrained platforms. The proposed STResNet series (ranging from Nano to Tiny variants) achieves competitive ImageNet-1K accuracy within a 4M parameter budget. Specifically, STResNetMilli attains 70.0% Top-1 accuracy with only 3.0M parameters, outperforming MobileNetV1 and ShuffleNetV2 at comparable computational complexity. For object detection, STYOLOMicro and STYOLOMilli achieve 30.5% and 33.6% mAP, respectively, on the MS-COCO dataset, surpassing YOLOv5n and YOLOX-Nano in both accuracy and efficiency. Furthermore, when STResNetMilli is used as a backbone with the Ultralytics detection head, it approaches the performance of the YOLOv11n model under the latest Ultralytics training environment.

Keywords: Compression, Object Detection, Classification, YOLO, ResNet, Computer Vision

1. Introduction

The growing adoption of edge intelligence has intensified the demand for compact and efficient neural networks capable of operating within the stringent memory and compute limits of resource-constrained hardware such as Microcontroller Units (MCUs) and Neural Processing Units (NPUs). Conventional Convolutional Neural Networks (CNNs), such as ResNet [1], while highly accurate, are often impractical for deployment on such platforms due to their substantial computational and memory requirements.

Lightweight architectures such as MobileNet [2], ShuffleNet [3], EfficientNet [4], SqueezeNet [5], and NASNet [6] have been proposed to address these challenges. However, they frequently rely on specialized operations—such as depthwise separable convolutions, fire modules, channel shuffle, and squeeze–excitation blocks—that are often unsupported or inefficient on MCU/NPU hardware. Moreover, these operations can be less amenable to quantization [7], further complicating hardware deployment.

To overcome these limitations, extensive research has focused on model compression and architectural optimization techniques, including pruning [8], quantization [7], low-rank decomposition [9], and neural architecture search (NAS) [10, 11], aiming to create models that are both memory- and compute-efficient while maintaining competitive accuracy on embedded platforms.

In this work, we introduce a new family of ultra-compact classification and detection models—specifically designed for MCU and NPU deployment—by combining layer decomposition [12, 13] with Neural Architecture Search (NAS) in a unified framework termed *CompressNAS*. The proposed classification backbone, *STResNet*, is a decomposed variant

*sudhakar.sah@gmail.com

of ResNet [1] that achieves between **3× and 12× compression** while preserving competitive accuracy.

The *STResNet* family adopts a clean and hardware-efficient design that facilitates seamless deployment on low-power devices. It retains the fundamental residual block structure of ResNet [1] but applies **layer decomposition** and **channel compression** to substantially reduce memory and compute requirements. Unlike complex NAS-generated or heavily engineered tiny architectures, STResNet relies solely on standard convolutional operations, resulting in improved numerical stability, quantization compatibility, and predictable behavior across embedded platforms.

Despite its structural simplicity, *STResNet* achieves performance competitive with state-of-the-art handcrafted lightweight models, demonstrating that a carefully decomposed ResNet backbone can effectively match or surpass architectures such as MobileNet [2, 14, 15] and EfficientNet [4]—all while maintaining a more deployment-friendly design.

Building upon the lightweight *STResNet* classification backbone, we extend our design to object detection by integrating the decomposed ResNet architecture into a YOLOX-style framework [16], resulting in a new family of efficient detectors termed *STYOLO*. Modern one-stage detectors such as YOLOv5 [17], YOLOv8 [18], YOLOX [16], and YOLOv11 [19] are typically trained end-to-end on large-scale datasets like MS-COCO [20]. However, such training strategies often overlook the benefits of employing specialized, pretrained backbones that are optimized for low-resource hardware.

In our framework, the *STYOLO* detector incorporates the ImageNet-pretrained, compressed *STResNet* backbone into the YOLOX detection head and neck. Unlike conventional pipelines that initialize backbones with random weights, this strategy leverages a pretrained, hardware-optimized backbone to achieve faster convergence, better feature reuse, and higher mAP scores under stringent model size constraints.

Empirical evaluations on the STMicroelectronics *STM32N6 Neural Art NPU* demonstrate that *STYOLO* achieves higher mAP on the MS-COCO dataset than YOLOv5n and approaches the accuracy of YOLOv8n, while maintaining a comparable model footprint. These results validate that the proposed modular design and pretrained backbone strategy offer a compelling trade-off between accuracy and efficiency, enabling superior embedded object detection performance compared to conventional end-to-end training approaches.

The key contributions of this paper are summarized as follows:

- We propose a novel **STResNet** family—an extremely compact classification model that combines layer decomposition and NAS-based channel optimization, achieving 3–12× compression with minimal accuracy degradation.
- We introduce **STYOLO**, a detection framework that integrates the decomposed ResNet backbone into YOLOX, achieving competitive accuracy and efficiency on MCU- and NPU-class hardware.
- We develop a **training strategy** that leverages a pre-trained and compressed STResNet backbone for initializing the STYOLO detector, enabling faster convergence and improved performance compared to end-to-end training from scratch.
- We perform **extensive experiments and real-hardware benchmarks** on the latest STMicroelectronics STM32N6, demonstrating that STYOLO outperforms YOLOv5n and approaches YOLOv8n performance for similar model sizes.

2. Related Work

2.1. Lightweight and Tiny Deep Learning Models

The increasing demand for on-device intelligence has led to substantial research into lightweight and compact neural network architectures designed for edge devices. Early works such as SqueezeNet [5] and MobileNet [2, 14] introduced efficient convolutional designs

that significantly reduced parameter counts and floating-point operations (FLOPs) without major accuracy losses. These models popularized techniques such as **depthwise separable convolutions**, pointwise projections, and bottleneck expansions, which became standard components in modern efficient architectures.

ShuffleNet [3] further improved efficiency through channel shuffling, while EfficientNet [4] introduced a compound scaling method that uniformly balances depth, width, and resolution. Despite their efficiency, such handcrafted models often rely on specialized layers types that are not always hardware-friendly for quantization or low-level deployment on MCUs and NPUs. In particular, depthwise convolutions, while computationally efficient on GPUs, can lead to degraded throughput or instability when quantized to low-bit formats [7].

2.2. Neural Architecture Search (NAS)

The advent of Neural Architecture Search (NAS) provided an automated approach to design efficient models optimized for specific hardware. NASNet [21] pioneered reinforcement learning-based search strategies, while MnasNet [22] introduced multi-objective optimization to jointly consider latency and accuracy. FBNet [11] and ProxylessNAS [23] further advanced this direction by introducing differentiable NAS frameworks that incorporate hardware constraints directly into the search process.

While these NAS-based architectures achieve state-of-the-art performance on mobile platforms, they often result in highly fragmented or irregular layer topologies that complicate deployment on resource-limited MCUs and NPUs. Our approach, in contrast, focuses on structural simplicity by retaining a ResNet-style topology while using NAS only for channel and decomposition-level optimization, thereby preserving deployment consistency across hardware backends.

2.3. Model Compression Techniques

Model compression through pruning, quantization, and low-rank factorization has been a parallel strategy to reduce inference complexity. Han *et al.* [8] introduced deep compression through iterative pruning and quantization, while Denton *et al.* [9] and Lebedev *et al.* [13] demonstrated the use of tensor decomposition (CP and Tucker) for accelerating convolutional layers. Subsequent work by Kim *et al.* [12] optimized low-rank decomposition for mobile hardware, achieving substantial performance gains with minimal accuracy loss.

Our proposed method draws inspiration from these decomposition-based approaches but integrates them within a NAS-guided pipeline, allowing both the decomposition rank and layer configuration to be optimized jointly. This hybrid design leads to compact models that maintain representational power while remaining efficient and hardware-friendly.

In summary, prior works have shown that lightweight model design, NAS, and decomposition can independently yield compact architectures. However, few approaches have combined these paradigms in a unified framework specifically tailored for MCU/NPU deployment. Our proposed **CompressNAS-ResNet** and **STYOLO** frameworks bridge this gap by combining decomposition-driven compression with NAS-based channel optimization, all within a simple ResNet-style topology that ensures compatibility, stability, and high efficiency on embedded inference hardware.

3. CompressNAS

CompressNAS is an architectural optimization framework that integrates layer decomposition with a NAS-guided channel optimization strategy. Specifically, Tucker decomposition is applied to each layer in the network, where the optimal rank for every layer is determined.

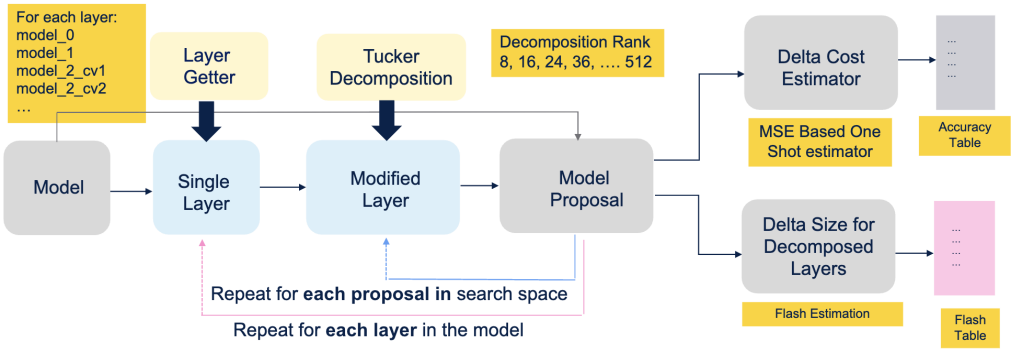

Figure 1. CompressNAS : Model Proposal Generation and Profiling

The selection of these ranks is formulated as a global optimization problem, capturing the interdependence of decomposition choices across layers. The impact of decomposing each layer is independently assessed in terms of accuracy degradation (Δacc) and memory footprint reduction (or flash size, Δflash). Subsequently, an Integer Linear Programming (ILP)-based search algorithm is employed to identify the optimal rank configuration for all layers, subject to predefined hardware constraints.

3.1. **Network Proposals**

Figure 1 illustrates the complete CompressNAS methodology. For convolutional layers, the number of decomposition proposals is determined by the number of output channels. An exhaustive search strategy is used to identify the optimal rank by generating multiple rank proposals, starting from 8 channels and incrementing in configurable steps of 8 (or 4), depending on the desired search granularity. For each rank proposal, Tucker decomposition is applied to the corresponding layer, and the decomposed layer replaces the original in the main architecture. The modified model is then evaluated to measure the resulting changes in accuracy and flash memory usage, with these results recorded in the corresponding lookup tables for subsequent ILP-based optimization.

3.2. **Accuracy Estimator**

After layer replacement, the model can be retrained or fine-tuned on the target dataset to estimate its final accuracy. However, given that the decomposition process can generate hundreds or even thousands of model proposals, retraining each configuration is computationally infeasible. To address this, Zero-Cost (ZC) proxies are employed, which estimate the impact of each layer modification using only a single forward pass through the altered architecture.

Although several ZC proxies were evaluated, their predictions did not align with the expected theoretical trend — i.e., higher ranks yielding lower reconstruction errors. Consequently, a Mean Squared Error (MSE)-based proxy was adopted, which computes the error between the output tensors of the modified and reference layers. Unlike traditional ZC proxies, the MSE-based approach consistently demonstrates the expected correlation between rank and error, providing a more reliable performance estimate.

3.3. **Flash Estimator**

Each modified model is exported to the ONNX format for consistent evaluation and hardware-level analysis. The difference in model size between the modified and original architectures is computed using Equation 3.1, where M denotes the number of output channels, N represents the input channels, k corresponds to the kernel size, and R indicates the decomposed rank. This formulation quantifies the change in memory footprint (Δflash) resulting from layer decomposition.

$$\Delta flash = NMk^2 - \left(NR \cdot 1{\times}1 + R^2 \cdot 3{\times}3 + RM \cdot 1{\times}1\right) \tag{3.1}$$

3.4. **Neural Architecture Search**

After constructing the lookup tables, an Integer Linear Programming (ILP)-based search algorithm is employed to determine the optimal architecture configuration that satisfies the predefined hardware constraints, as formulated in Equation 3.2. This ILP optimization ensures an efficient trade-off between accuracy and memory footprint, yielding a decomposed architecture that best fits the target device specifications.

$$\text{Accuracy} = \max \sum_{(i,j)\in E} \Delta\text{accuracy}_{ij} \quad \text{s.t.} \quad \sum_{(i,j)\in E} \Delta\text{flash}_{ij} \leq flash_{\max}. \tag{3.2}$$

After varying the constraints on accuracy and flash, two optimized model families are formed: *STResNet* and *STYOLO*.

4. **Architecture: STResNet**

The *STResNet* family consists of a series of decomposed variants of the ResNet architecture, each designed to balance model complexity, memory footprint, and accuracy. Five configurations are introduced—*STResNet-Tiny*, *STResNet-Milli*, *STResNet-Micro*, *STResNet-Nano*, and *STResNet-Pico*—corresponding to progressively higher levels of decomposition and reduced parameter counts. All variants contain fewer than 4 million parameters, with *STResNet-Tiny* serving as the largest and most accurate configuration, and *STResNet-Pico* representing the most compact model optimized for deployment under extreme resource constraints. This hierarchical scaling enables flexible deployment across a wide range of MCU and NPU hardware, depending on available compute and memory budgets. Table 1 shows the reference ResNet18 [24] architecture and Table 2 shows the decomposed and compressed STResNetNano architecture. It is clear from these tables that each layer is decomposed by CompressNAS in different way by considering overall impact on accurac and size of the model (given the constraint from user).

4.1. **Flash Consideration**

Flash memory constraints are a critical factor when deploying CNN models on MCUs and edge NPUs. The decomposed layers in STResNet significantly reduce the number of parameters that must be stored in non-volatile memory. Each convolutional layer is represented using three smaller factorized matrices, which collectively maintain representational capacity while lowering storage requirements. In addition, the absence of specialized or irregular layers removes the need for custom kernels, enabling shared, uniform convolution implementations that are both memory-efficient and cache-friendly.

4.2. **RAM Consideration**

Runtime memory usage, particularly for feature map storage, poses a major bottleneck in embedded inference. While the decomposed layers in STResNet do not inherently reduce

RAM consumption, analysis revealed that a sub-layer within the stem block accounted for the highest memory utilization. To address this, a projection layer was introduced, as described in Section 5.3. This modification achieved approximately a 2× reduction in RAM usage with a negligible accuracy drop of less than 0.5%.

4.3. **Latency Consideration**

Inference latency on NPUs and MCUs is influenced not only by computational complexity (FLOPs) but also by factors such as operator fusion, kernel regularity, and memory bandwidth. The STResNet architecture preserves high operator regularity by exclusively employing standard 3×3 and 1×1 convolutions across all layers, thereby facilitating efficient hardware acceleration without incurring the overhead associated with diverse or irregular convolution types.

In summary, STResNet demonstrates that a decomposition-based design, when carefully optimized, can yield MCU- and NPU-efficient models exhibiting low latency, high flash efficiency, and optimized RAM utilization—all achieved without relying on depthwise separable convolutions or manually crafted modules.

Table 1. ResNet-18 Architecture

Block / Layer	**In → Out**	**Conv Layers (Inner Channels)**
Stem	3 → 64	7x7 /3→64 /2, MaxPool 3x3 /2
Layer1	64 → 64	Block1: 3x3/64, 3x3/64; Block2: 3x3/64, 3x3/64
Layer2	64 → 128	Block1: 3x3/64 s2, 3x3/128; Block2: 3x3/128, 3x3/128
Layer3	128 → 256	Block1: 3x3/128 s2, 3x3/256; Block2: 3x3/256, 3x3/256
Layer4	256 → 512	Block1: 3x3/256 s2, 3x3/512; Block2: 3x3/512, 3x3/512
FC	512 → 1000	Linear Layer

Table 2. STResNet-Nano Architecture

Block / Layer	**In → Out**	**Conv Layers (Inner Channels)**
Stem	3 → 64	1x1/3→3, 7x7/3→8, 1x1/8→32; **proj.1x1/32→64**
Layer1	64 → 64	Block1: 1x1/64→32, 3x3/32, 1x1/32→64; 1x1/64→16, 3x3/16, 1x1/16→64. Block2: 1x1/64→40, 3x3/40, 1x1/40→64; 1x1/64→8, 3x3/8, 1x1/8→64
Layer2	64 → 128	Block1: 1x1/64→32, 3x3/32 s2, 1x1/32→128; 1x1/128→8, 3x3/8, 1x1/8→128. Block2: 1x1/128→64, 3x3/64, 1x1/64→128; 1x1/128→16, 3x3/16, 1x1/16→128
Layer3	128 → 256	Block1: 1x1/128→48, 3x3/48 s2, 1x1/48→256; 1x1/256→16, 3x3/16, 1x1/16→256. Block2: 1x1/256→48, 3x3/48, 1x1/48→256; 1x1/256→8, 3x3/8, 1x1/8→256
Layer4	256 → 512	Block1: 1x1/256→32, 3x3/32 s2, 1x1/32→512; 1x1/512→8, 3x3/8, 1x1/8→512. Block2: 1x1/512→48, 3x3/48, 1x1/48→512; 1x1/512→8, 3x3/8, 1x1/8→512

5. **Architecture : STYOLO**

The STYOLO family of object detectors is built upon the STResNet backbone, incorporating a modified neck and detection head derived from the YOLOX [16] architecture. The

overall training pipeline follows the YOLOX framework, with several targeted modifications aimed at improving efficiency and accuracy. In alignment with the size-based hierarchy of STResNet, five variants of STYOLO are introduced—STYOLO-Tiny, STYOLO-Milli, STYOLO-Micro, STYOLO-Nano, and STYOLO-Pico—each corresponding to a specific backbone configuration. This section details the architectural adjustments and training strategies employed to enhance the performance of the STYOLO model family across diverse resource constraints.

5.1. **Neck Adjustment**

In the STYOLO architecture, the backbone generates raw multi-scale feature maps—`dark3`, `dark4`, and `dark5`—that possess relatively high channel dimensions. These feature maps are not directly compatible with the YOLOX-style neck, which expects reduced channel sizes for efficient multi-scale feature aggregation. To ensure proper alignment, a channel projection is applied using 1×1 convolutions that compress the feature dimensions before they are fed into the neck (Table 7). Specifically, backbone outputs of 128, 256, and 512 channels are projected to 64, 128, and 256 channels with strides 8, 16 and 32, respectively. This projection preserves spatial resolution while significantly reducing computational overhead, enabling the PANet-style neck to perform effective and memory-efficient feature fusion across multiple scales.

5.2. **Learning Rate Optimization**

To enhance training stability and accelerate convergence, a layer-wise learning rate scaling strategy was adopted for different components of the STYOLO architecture. The backbone was trained with a reduced learning rate of $0.2\times$ the base value to preserve pretrained feature representations, while the neck was updated using $0.8\times$ the base learning rate to facilitate effective feature aggregation. The detection head, being randomly initialized and requiring faster adaptation, was trained with the full base learning rate ($1.0\times$).

This differential learning rate scheme, inspired by prior work on layer-wise optimization in object detectors [16, 25], enables a balance between stability in lower layers and rapid learning in higher layers. Empirically, this approach improved both convergence speed and final detection accuracy, increasing the mAP of *STYOLO-Nano* from 21.32 to 26.25 on the MS-COCO dataset.

5.3. **RAM-Efficient Projection Layer**

Memory profiling of *STYOLOMicro* on the STM32N6 shows a RAM usage of 4.26 MB, higher than competing models such as YOLOv5n [26] and YOLOv8n [18]. Reducing RAM is crucial for MCU/NPU deployment, as N6 performance drops sharply beyond 4 MB due to external memory dependence. Layer-wise analysis identifies the stem layer's final convolution (1×1, $8\rightarrow32$) as the main contributor, driven by large feature maps and channel count.

To mitigate this, we modify the STResNet backbone by adding a lightweight *projection layer*. The third convolution in the stem originally expecting 64 channels and outputs 64 channels. After modification, it outputs 32 instead of 64 channels, followed by a parallel 1×1 projection ($32\rightarrow64$) operating on smaller feature maps. This balances efficiency and expressivity—reducing memory footprint while preserving channel diversity. Consequently, RAM usage drops from 4.26 MB to 2.46 MB with less than 0.5% mAP degradation.

6. Results

6.1. STResNet

Table 3. Comparison of ResNet-18 with STResNet MCU variants, accuracy on ImageNet [1], performance data on STM32N6 board [*- too large to fit on internal RAM].

Model	Params	Top-1	Drop	Size Red.	Latency	RAM
ResNet-18* [1]	11.68	70.5	0.0	1.0×	-	-
STResNetTiny	3.99	71.6	+1.1	3×	21.29	1.39
STResNetMilli	3.00	70.0	−0.5	3.89×	18.29	1.39
STResNetMicro	1.50	66.7	−3.8	7.8×	14.36	0.882
STResNetNano	0.95	58.8	−11.7	12.3×	10.91	0.833
STResNetPico	0.62	48.8	−21.7	18.8×	8.24	0.833

Table 3 shows performance of STResNet models on imagenet dataset along with Flash, RAM and latency requirments measured on STM32N6 [27] Neural Art NPU. The STResNet family demonstrates a strong trade-off between accuracy and efficiency on MCU hardware. Compared to ResNet-18, STResNet variants achieve up to 18.8× model size reduction and 2.6× faster latency on the STM32N6 board. While accuracy gradually decreases with smaller models, STResNetTiny even surpasses ResNet-18 by +1.1%, showing that the architecture effectively balances compactness and performance for edge deployment.

All the models are trained on ImageNet [28] dataset using default timm [29] training pipeline for 300 epochs. Each model is benchmarked on STM32N6 using ST Edge AI Developer Cloud [27] to get Flash, RAM and latency values.

Table 4 presents a comparative analysis of several lightweight classification models with fewer than 4 M parameters on the ImageNet-1K dataset [28]. The proposed *STResNet* family demonstrates competitive or superior accuracy compared to state-of-the-art handcrafted models such as MobileNet [2, 14, 15], ShuffleNet [3, 30], and SqueezeNet [5]. Specifically, *STResNetTiny* achieves 71.6% Top-1 accuracy with 3.99 M parameters, closely matching MobileNetV2-1.0 while exhibiting lower RAM usage (1.39 vs 2.01 MB and latency (21.3 ms vs. 22.4 ms). The mid-sized *STResNetMilli* attains 70.0% accuracy with 3.0 M parameters—outperforming MobileNetV1-0.75 and ShuffleNetV2-1.0× by 1.6–2.6% absolute Top-1 accuracy at comparable or lower latency. The compact *STResNetMicro* reaches 66.7% accuracy with only 1.5 M parameters, exceeding ShuffleNetV2-0.5× by 5.7% and demonstrating a favorable trade-off between accuracy and model size. Even the smallest variant, *STResNetNano*, maintains 58.8% accuracy at under 1 M parameters—comparable to SqueezeNet 1.1 but with significantly lower latency (10.9 ms vs. 119.9 ms). *STResNetMicro* has the best accuracy/size/latency tradeoff compared of all other comparable state of the art models.

Another key aspect is the quantization-friendliness of compact models such as MobileNet [2, 14, 15], SqueezeNet [5], EfficientNet [4], and ShuffleNet [3, 30]. Many of these networks suffer notable accuracy degradation under ultra-low-bit quantization (<8-bit), as shown in prior studies [31, 32]. SqueezeNet and EfficientNet are often excluded from such benchmarks due to their reliance on specialized layers that complicate quantization, typically requiring quantization-aware training (QAT) to recover accuracy. In contrast, STResNet, derived from the regular ResNet [1] architecture, is inherently quantization-friendly. Its uniform convolutional structure and absence of exotic operators make it well-suited for MCU and NPU deployment, where low-bit quantization is essential. This design ensures strong performance in scenarios demanding both compactness and quantization efficiency.

Table 4. Comparison of lightweight models on ImageNet-1K [28] with < 4M parameters. Accuracy is reported on FP32 models; RAM, latency, and INT8 performance are measured on STM32N6 [27]

Model	Params (M)	Top-1 Acc. (%)	RAM (MB)	Latency (ms)
MobileNetV1-1.00 [2]	4.20	70.6	1.53	20.76
MobileNetV3-large-0.75 [15]	4.00	73.3	1.38	36.46
STResNetTiny	3.99	71.6	1.39	21.29
MobileNetV2-1.00 [2]	3.50	71.8	2.01	22.44
STResNetMilli	3.00	70.0	1.39	18.29
MobileNetV1-0.75 [2]	2.60	68.4	1.29	11.50
MobileNetV3-small-1.0 [15]	2.53	67.4	1.579	54.35
ShuffleNetV2 1.0× [30]	2.30	69.4	0.738	34.15
MobileNetV2-0.5 [14]	2.00	65.4	1.24	11.51
MobileNetV3-small-0.75 [15]	1.99	65.4	1.579	33.12
ShuffleNetV1 1.0× [3]	1.87	68.13	0.628	15.54
MobileNetV2-0.35 [14]	1.70	60.3	0.90	10.39
STResNetMicro	1.50	66.7	0.882	14.36
ShuffleNetV2 0.5× [30]	1.40	61.0	0.735	8.54
MobileNetV1-0.5 [2]	1.30	63.7	0.574	8.09
SqueezeNet 1.0 [5]	1.25	57.5	8.75	119.97
SqueezeNet 1.1 [5]	1.24	58.2	0.785	9.46
STResNetNano	0.95	58.8	0.833	10.91
MobileNetV1-0.25 [2]	0.50	50.6	0.383	3.88
STResNetPico	0.60	48.8	0.833	8.24

6.2. **STYOLO**

Object detection models are trained on MS COCO [33] dataset for 300 epochs using YOLOX [16] training pipeline with the customization as explained in section 5.

Table 5 summarizes the performance comparison of various lightweight object detection models on the STM32N6 (Benchmarked at 320 px input resolution). The results demonstrate that the proposed *STYOLO* family consistently achieves a favorable balance between accuracy, latency, and memory efficiency compared to existing compact detectors such as YOLOv5n [26], YOLOv8n [18], and YOLOX-nano [16]. *STYOLOMicro* achieves a mAP of 30.54% with only 1.69 M parameters outperforming YOLOv5n (+2.54 mAP) while maintaining comparable RAM usage (2.46 MB vs. 2.11 MB). Although YOLOv8n reaches a higher accuracy (35.60 mAP), it does so using custom Ultralytics training pipeline which features a lot of mAP improvement techniques. *STYOLOMilli* is 2 mAP lower compared to YOLO8n using ReLU activation. We believe that using our backbone with YOLOv8n training pipeline, we can improve the accuracy of Ultralytics models as well and it is future work of this paper. Similarly, *STYOLOMilli* and *STYOLOTiny* models surpass their YOLOX-Tiny counterparts in accuracy (by +0.85 mAP and +2.65 mAP respectively) with lower parameter counts and improved scalability across hardware tiers.

At smaller configurations, *STYOLONano* achieves competitive accuracy (23.6 mAP), slightly below YOLOX-nano (23.8 mAP). Interestingly, YOLOX-nano is 3× slower than STYOLOTiny, highlighting that certain specialized operators in compact models are not MCU/NPU-friendly. The *STYOLOPico* variant defines the lower bound of the design space with only 0.74 M parameters, offering a deployable option for sub-1 MB memory targets. Using ReLU activations instead of SiLU further simplifies fixed-point deployment while maintaining accuracy, reinforcing the design's suitability for embedded applications. As shown in Table 6, the proposed RAM-efficient projection layer reduces runtime memory in STYOLOMicro from 4.26 MB to 2.46 MB (a 42% decrease) with comparable accuracy (30.54 mAP → 30.12 mAP) and slightly better latency (47.32 ms → 42.99 ms), validating its effectiveness in minimizing feature map size without loss of representation quality.

Table 5. Comparison of Small Object Detection Models and Benchmarking on STM32N6 at 320px, [*- too large to fit in internal RAM]

Model	Res.	Act.	mAP	Params	STM32N6 Benchmark (320px)		
					Lat.	Flash	RAM
YOLO8n	640	SiLU	37.30	3.20	40.98	3.28	2.17
YOLO8n	640	ReLU	35.60	3.20	40.98	3.28	2.17
YOLOX-Tiny*	416	ReLU	32.80	5.06	-	-	-
STYOLOMilli	640	ReLU	33.65	3.25	76.90	3.49	3.49
YOLO5n	640	SiLU	28.00	1.90	35.98	2.83	2.11
STYOLOMicro	640	ReLU	30.54	1.69	47.32	2.00	2.46
STYOLONano	640	ReLU	26.25	1.13	41.37	1.47	2.46
STYOLOPico	640	ReLU	20.54	0.74	28.53	1.07	0.75
STYOLOPico	416	ReLU	18.14	0.74	28.53	1.07	0.75
YOLOX-Nano	416	ReLU	23.80	1.08	139.30	1.31	2.41
STYOLONano	416	ReLU	23.60	1.13	41.37	1.47	2.46

Table 6. STYOLO-RAM improvement using projection layer

Model	Res.	Act.	mAP	Params (M)	STM32N6 Benchmark (320px)		
					Lat. (ms)	Flash (MB)	RAM (MB)
STYOLOMicro	640	ReLU	30.54	1.69	47.32	2.00	4.26
STYOLOMicro (Proj)	640	ReLU	**30.12**	1.69	42.99	2.00	**2.46**

6.3. **Ultralytics Experiments**

To verify that the proposed methodology—integrating optimized pre-trained backbones with existing neck and head architectures—generalizes across different training environments, we extended our experiments to the Ultralytics YOLOv11 framework. Specifically, the STResNetMilli and STResNetMicro models, pre-trained on the ImageNet-1K dataset, were attached to channel-adjusted neck and head modules from YOLOv11 and subsequently fine-tuned on the full MS-COCO dataset. The resulting STResNetMicro-YOLOv11 model achieved a performance level comparable to the original YOLOv11n. When trained under the same training environment as Ultralytics and learning rate optimization settings described earlier, the proposed model closely matched the performance of YOLOv11n as shown in Table 7, validating the adaptability and robustness of our approach across diverse training pipelines.It is also evident from the table that STResNet backbone performed almost similar to YOLO11n at lower resolution training which is very cruicial for MCU and NPU devices given the memory constraints.

7. **Future Work**

In future, we plan to extend this framework in three directions: (1) integrate mixed-precision quantization into the CompressNAS search to jointly optimize bit-width and rank for better accuracy–efficiency balance; (2) adapt the STResNet backbone for multi-task edge vision tasks such as segmentation and keypoint detection; and (3) develop cross-hardware adaptive compression methods to automatically tune decomposition and channel scaling for diverse MCU, NPU, and DSP architectures.

Table 7. STResNet backbone performance with Ultralytics pipeline.

Backbone /Head	**mAP(50:95) /Resolution**	**Params (M)**
YOLO11n/YOLOv11n	22.60/224	2.60
STResNetMicro/YOLOv11n	22.60/224	2.90
YOLO11n/YOLOv11n	39.50/640	2.60
STResNetMicro/YOLOv11n	38.60/640	2.90
STResNetMilli/YOLOv11n	40.50/640	4.46
STResNetMicro/YOLOv11s	42.01/640	5.50
YOLO11s	47.00/640	9.4

8. Conclusion

In this work, we presented a family of lightweight, hardware-efficient neural networks tailored for MCU and NPU deployment. The proposed *STResNet* combines low-rank layer decomposition with NAS-guided channel optimization (CompressNAS), forming a simplified yet effective ResNet variant that achieves 3–12× compression while maintaining competitive accuracy. Unlike handcrafted models such as MobileNet or EfficientNet, STResNet avoids specialized operators, ensuring quantization stability and efficient fixed-point execution.

Building on this compact backbone, we introduced the *STYOLO* series of object detectors that integrate STResNet within a YOLOX-style framework. Extensive experiments and hardware benchmarks on the STM32N6 Neural Art NPU show that STYOLO models deliver superior accuracy–efficiency trade-offs compared to YOLOv5n, YOLOv8n, and YOLOX-Nano. The proposed RAM-efficient projection layer further reduced memory usage by 42% and improved latency by 9%, demonstrating its effectiveness for edge deployment.

References

[1] K. He, X. Zhang, S. Ren, and J. Sun. "Deep residual learning for image recognition". In: *CVPR*. 2016.

[2] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam. "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications". In: *arXiv preprint arXiv:1704.04861*. 2017.

[3] X. Zhang, X. Zhou, M. Lin, and J. Sun. "ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2018, pp. 6848–6856.

[4] M. Tan and Q. V. Le. "EfficientNet: Rethinking model scaling for convolutional neural networks". In: *ICML*. 2019.

[5] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer. "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size". In: *arXiv preprint arXiv:1602.07360*. 2016.

[6] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le. "Learning Transferable Architectures for Scalable Image Recognition". In: *CVPR*. 2018.

[7] B. Jacob et al. "Quantization and training of neural networks for efficient integer-arithmetic-only inference". In: *CVPR*. 2018.

[8] S. Han, H. Mao, and W. J. Dally. "Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding". In: *ICLR* (2016).

[9] E. Denton et al. "Exploiting linear structure within convolutional networks for efficient evaluation". In: *NeurIPS*. 2014.

[10] T. Elsken, J. H. Metzen, and F. Hutter. "Neural architecture search: A survey". In: *Journal of Machine Learning Research*. 2019.

[11] B. Wu et al. "FBNet: Hardware-aware efficient convnet design via differentiable neural architecture search". In: *CVPR*. 2019.

[12] Y. Kim, E. Park, and S. Yoo. "Compression of deep convolutional neural networks for fast and low power mobile applications". In: *ICLR*. 2016.

[13] V. Lebedev and V. Lempitsky. "Speeding-up convolutional neural networks using fine-tuned CP-decomposition". In: *ICLR*. 2015.

[14] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen. "MobileNetV2: Inverted Residuals and Linear Bottlenecks". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2018, pp. 4510–4520.

[15] A. Howard, M. Sandler, G. Chu, L.-C. Chen, B. Chen, M. Tan, W. Wang, Y. Zhu, R. Pang, V. Vasudevan, Q. V. Le, and H. Adam. "Searching for MobileNetV3". In: *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 2019, pp. 1314–1324.

[16] Z. Ge, S. Liu, F. Wang, Z. Li, and J. Sun. "YOLOX: Exceeding YOLO Series in 2021". In: *arXiv preprint arXiv:2107.08430*. 2021. URL: https://arxiv.org/abs/2107.08430.

[17] G. Jocher, A. Chaurasia, and J. Qiu. *YOLOv5 by Ultralytics*. https://github.com/ultralytics/yolov5. Accessed: 2025-10-03. 2020.

[18] G. Jocher, A. Chaurasia, and J. Qiu. *Ultralytics YOLOv8*. https://github.com/ultralytics/ultralytics. Version 8.0, Accessed: 2025-10-03. 2023.

[19] G. Jocher and J. Qiu. *Ultralytics YOLO11*. Version 11.0.0. 2024. URL: https://github.com/ultralytics/ultralytics.

[20] T.-Y. Lin, M. Maire, S. Belongie, L. Bourdev, R. Girshick, J. Hays, P. Perona, D. Ramanan, C. L. Zitnick, and P. Dollár. *Microsoft COCO: Common Objects in Context*. cite arxiv:1405.0312Comment: 1) updated annotation pipeline description and figures; 2) added new section describing datasets splits; 3) updated author list. 2014. URL: http://arxiv.org/abs/1405.0312.

[21] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le. "Learning transferable architectures for scalable image recognition". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2018, pp. 8697–8710.

[22] M. Tan, B. Chen, R. Pang, V. Vasudevan, M. Sandler, A. Howard, and Q. V. Le. "Mnasnet: Platform-aware neural architecture search for mobile". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 2019, pp. 2820–2828.

[23] H. Cai, L. Zhu, and S. Han. "ProxylessNAS: Direct neural architecture search on target task and hardware". In: *International Conference on Learning Representations (ICLR)*. 2019.

[24] K. He, X. Zhang, S. Ren, and J. Sun. "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[25] G. Jocher et al. *YOLOv5 by Ultralytics*. https://github.com/ultralytics/yolov5. 2021.

[26] G. Jocher. *YOLOv5 by Ultralytics*. Version 7.0. May 2020. DOI: 10.5281/zenodo.3908559. URL: https://github.com/ultralytics/yolov5.

[27] STMicroelectronics. *STM32N6 AI NPU platform for next-generation embedded vision*. https://www.st.com/en/microcontrollers-microprocessors/stm32n6.html. 2024.

[28] J. Deng, R. Socher, L. Fei-Fei, W. Dong, K. Li, and L.-J. Li. "ImageNet: A large-scale hierarchical image database". In: *2009 IEEE Conference on Computer Vision and Pattern Recognition(CVPR)*. Vol. 00. June 2009, pp. 248–255. DOI: 10.1109/CVPR.2009.5206848. URL: https://ieeexplore.ieee.org/abstract/document/5206848/.

[29] R. Wightman. *PyTorch Image Models (timm)*. https://github.com/rwightman/pytorch-image-models. 2019. DOI: 10.5281/zenodo.4414861.

[30] N. Ma, X. Zhang, H.-T. Zheng, and J. Sun. "ShuffleNet V2: Practical Guidelines for Efficient CNN Architecture Design". In: *Proceedings of the European Conference on Computer Vision (ECCV)*. 2018, pp. 116–131.

[31] Y. Li, Z. Dong, H. Yang, S. Liu, Z. Hu, and Y. Wang. "BRECQ: Pushing the Limit of Post-Training Quantization by Block Reconstruction". In: *International Conference on Learning Representations (ICLR)*. 2021.

[32] J. Park, S. Li, W. Kim, S. Lin, and S. W. Keckler. "ProFit: Progressive Filter Pruning Globally at Fine-grained Level". In: *International Conference on Machine Learning (ICML)*. 2020, pp. 7590–7600.

[33] *COCO Detection Challenge*. https://codalab.lisn.upsaclay.fr/competitions/7384.

Appendix A. STResNet Architecture

Table 8. STResNet-Pico Architecture

Block / Layer	In → Out	Conv Layers (Inner Channels)
Stem	3 → 64	1x1/3→3, 7x7/3→8, 1x1/8→32; **proj.1x1/32→64**
Layer1	64 → 64	Block1: 1x1/64→24, 3x3/24, 1x1/24→64; 1x1/64→16, 3x3/16, 1x1/16→64 Block2: 1x1/64→24, 3x3/24, 1x1/24→64; 1x1/64→8, 3x3/8, 1x1/8→64
Layer2	64 → 128	Block1: 1x1/64→24, 3x3/24 s2, 1x1/24→128; 1x1/128→8, 3x3/8, 1x1/8→128 Block2: 1x1/128→8, 3x3/8, 1x1/8→128; 1x1/128→8, 3x3/8, 1x1/8→128
Layer3	128 → 256	Block1: 1x1/128→8, 3x3/8 s2, 1x1/8→256; 1x1/256→8, 3x3/8, 1x1/8→256 Block2: 1x1/256→8, 3x3/8, 1x1/8→256; 1x1/256→8, 3x3/8, 1x1/8→256
Layer4	256 → 512	Block1: 1x1/256→8, 3x3/8 s2, 1x1/8→512; 1x1/512→8, 3x3/8, 1x1/8→512 Block2: 1x1/512→8, 3x3/8, 1x1/8→512; 1x1/512→8, 3x3/8, 1x1/8→512

Table 9. STResNet-Tiny Architecture

Block / Layer	In → Out	Conv Layers (Inner Channels)
Stem	3 → 64	1x1/3→3, 7x7 s2/3→16, 1x1/16→32; **proj.1x1/32→64**
Layer1	64 → 64	Block1: 3x3/64→64, 3x3/64→64 Block2: 3x3/64→64, 3x3/64→64
Layer2	64 → 128	Block1: 3x3 s2/64→128; 1x1/128→96, 3x3/96, 1x1/96→128 Block2: 3x3/128→128; 1x1/128→80, 3x3/80, 1x1/80→128
Layer3	128 → 256	Block1: 3x3 s2/128→256; 1x1/256→192, 3x3/192, 1x1/192→256 Block2: 3x3/256→256; 1x1/256→96, 3x3/96, 1x1/96→256
Layer4	256 → 512	Block1: 1x1/256→208, 3x3 s2/208→208, 1x1/208→512; 1x1/512→88, 3x3/88, 1x1/88→512 Block2: 1x1/512→192, 3x3/192, 1x1/192→512; 1x1/512→112, 3x3/112, 1x1/112→512

Table 10. STResNet-Micro Architecture

Block / Layer	**In → Out**	**Conv Layers (Inner Channels)**
Stem	3 → 64	1x1/3→3, 7x7/3→8, 1x1/8→32; **proj.1x1/32→64**
Layer1	64 → 64	Block1: 1x1/64→64, 3x3/64, 1x1/64→64; 1x1/64→64, 3x3/64, 1x1/64→64 Block2: 1x1/64→64, 3x3/64, 1x1/64→64; 1x1/64→64, 3x3/64, 1x1/64→64
Layer2	64 → 128	Block1: 1x1/64→40, 3x3/40 s2, 1x1/40→128; 1x1/128→32, 3x3/32, 1x1/32→128 Block2: 1x1/128→88, 3x3/88, 1x1/88→128; 1x1/128→32, 3x3/32, 1x1/32→128
Layer3	128 → 256	Block1: 1x1/128→88, 3x3/88 s2, 1x1/88→256; 1x1/256→72, 3x3/72, 1x1/72→256 Block2: 1x1/256→80, 3x3/80, 1x1/80→256; 1x1/256→32, 3x3/32, 1x1/32→256
Layer4	256 → 512	Block1: 1x1/256→80, 3x3/80 s2, 1x1/80→512; 1x1/512→8, 3x3/8, 1x1/8→512 Block2: 1x1/512→72, 3x3/72, 1x1/72→512; 1x1/512→64, 3x3/64, 1x1/64→512