Deep Reorganization: Retaining Residuals in TinyML

Hashan Roshantha Mendis[1], Chih-Kai Kang[1,2], Chun-Han Lin[3], Ming-Syan Chen[2], Pi-Cheng Hsiu[1]
[1]Research Center for Information Technology Innovation, Academia Sinica, Taiwan
[2]Graduate Institute of Electrical Engineering, National Taiwan University, Taiwan
[3]Dept. of Computer Sci. & Information Engineering, National Taiwan Normal University, Taiwan
{rosh.mendis, pchsiu}@citi.sinica.edu.tw, ckkang@arbor.ee.ntu.edu.tw, chlin@ntnu.edu.tw, mschen@ntu.edu.tw

ABSTRACT

Designing intelligent, tiny devices with limited memory is immensely challenging, exacerbated by the additional memory requirement of residual connections in deep neural networks. In contrast to existing approaches that eliminate residuals to reduce peak memory usage at the cost of significant accuracy degradation, this paper presents DERO, which reorganizes residual connections by leveraging insights into the types and interdependencies of operations across residual connections. Evaluations were conducted across diverse model architectures designed for common computer vision applications. DERO consistently achieves peak memory usage comparable to plain-style models without residuals, while closely matching the accuracy of the original models with residuals.

CCS CONCEPTS

• **Computer systems organization** → **Embedded software**; **Neural networks**.

KEYWORDS

TinyML, deep neural networks, peak memory, residual connections

1 INTRODUCTION

Advancements in deep neural networks (DNNs) now enable edge devices to perform inference [13, 14], reducing reliance on cloud servers. Edge inference offers faster response times, lower latency, reduced bandwidth usage, and enhanced privacy. Techniques such as model quantization [19], network pruning [22], and neural architecture search [14] have been developed to enhance inference efficiency on edge devices by decreasing the *storage* and *computation* requirements of DNN models. However, despite increased deployment on resource-constrained edge devices, reducing neural network memory usage remains a challenge.

DNN inference on MCU-based tiny devices is particularly difficult due to their extremely limited memory capacity (e.g., on-chip SRAM is typically <320 KB on the STM32 family of MCUs). Resource-intensive tasks, such as image classification or object detection, often exceed the available SRAM capacity with their feature maps and dense prediction results, and accommodating residual connections further exacerbates the high peak memory issue, as illustrated in Figure 1. While writable

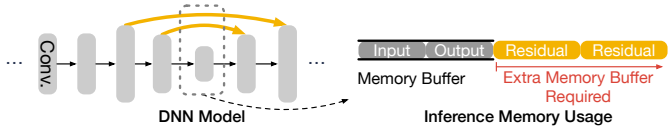


Figure 1: Extra memory requirement for residual connections

Flash memory can be used to handle large DNNs, its performance is limited by the inefficient link between the Flash and on-chip SRAM, as well as the SRAM capacity. Thus, the SRAM must be used to store the feature maps during inference, severely restricting the size and capability of the deployable DNN model.

Reducing the model size alone may not suffice for tiny device deployment due to certain computation blocks still consuming excessive memory. Therefore, various methods, such as operator execution reordering [15], temporary data swapping to secondary storage [9], convolution loop optimization [11], and depth-first inference [1, 13], have been proposed to reduce peak memory usage during DNN inference. Although these methods reduce the memory requirement of *feature maps*, they cannot eliminate the additional memory requirement of *residual connections*, which are prevalent in modern DNN architectures. Essentially, residual connections require an additional SRAM buffer for downstream computations, as shown in Figure 1, so they can still cause memory bottlenecks during inference for severely resource-constrained tiny devices.

Intuitively, reducing the number of residual connections [21] or shortening the connections [20] can lower the peak memory usage. Recent efforts have strived to remove all residual connections in the network, thereby creating *plain-style* models with minimized peak memory usage. However, residual connections are crucial as they facilitate efficient gradient propagation, enhance feature map reuse, and shorten the information path by transmitting information from earlier to later layers in the network [7, 8]. Although methods, such as parameterizing weight kernels [3], distilling residual connections with a teacher model [12], and specializing the activation functions [21], have been proposed to derive accurate plain-style models, their accuracy is still inferior to that of original models with residual connections, particularly for those with concatenation type residuals (e.g., DenseNet [10]). Therefore, simply removing residual connections may severely impact the training efficiency of DNNs and, consequently, degrade the model accuracy.

This work is motivated by our observation that the memory usage of residual connections varies depending on the types and interdependency of operations across the residuals. Certain operations can be performed in-place, allowing them to overwrite their inputs without affecting their outputs. This suggests that residual connections can be aggregated at specific positions in the network without the need for an additional memory buffer. However, improper placement of residual connections can be detrimental to model accuracy, as they may not efficiently propagate the gradient during training. Therefore, the start

DAC '24, June 23–27, 2024, San Francisco, CA, USA

ACM ISBN 979-8-4007-0601-1/24/06…$15.00
https://doi.org/10.1145/3649329.3656219

and end points of residual connections need to be appropriately selected to reduce memory usage without compromising accuracy.

We propose the concept of *residual reorganization*, aiming to achieve inference peak memory requirements comparable to plain-style models, while maintaining the training efficiency and accuracy of the original model with residuals. Instead of simply removing residuals, we realize our concept via *DERO*, a simple yet systematic approach that exploits the characteristics and processing behavior of operations to determine the appropriate placement of residual connections for peak memory reduction. DERO addresses two key design challenges in residual reorganization. The first challenge is to reduce memory usage without degrading training efficiency, which is addressed through *dependency-aware residual reconstruction* to determine the specific start and end positions of residuals. The second challenge is to maintain valid tensor computations after residual reconstruction without degrading model accuracy, achieved through *residual-aware operation refinement*, which modifies, augments, and combines specific operations.

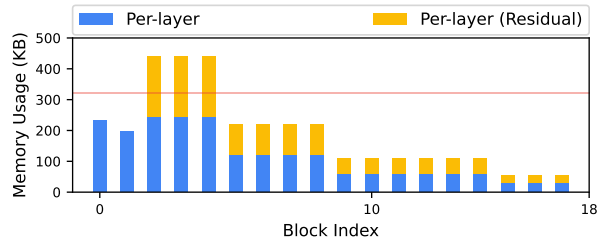
We evaluated DERO on various modern DNNs, validated its performance on large-scale datasets, including *ImageNet* for image classification and *MSCOCO* for object detection, and deployed the reorganized models on an STMicroelectronics microcontroller. DERO was compared to recent plain-style approaches, *RepCONV* [3] and *JointRD* [12], which remove residuals. When techniques like depth-first inference [13] are employed, the peak memory usage of the original models can be reduced by between 26% and 75%, and when combined with DERO, this reduction can be further extended to between 33% and 83%, reaching the level of plain-style approaches. Moreover, unlike plain-style approaches, which may significantly degrade model accuracy, DERO retains the training efficiency and closely matches the accuracy of the original models. Our experiments demonstrate that plain-style approaches are ineffective for advanced models like DenseNet121, but DERO enables their deployment on tiny MCUs with only 320 KB SRAM capacity.

The remainder of this paper is organized as follows. Section 2 provides background information, and Section 3 explains the motivation. Section 4 presents the design of DERO, including residual reconstruction and operation refinement. Experimental results are reported in Section 5, with limitations and potential improvements discussed in Section 6. Section 7 contains some concluding remarks.

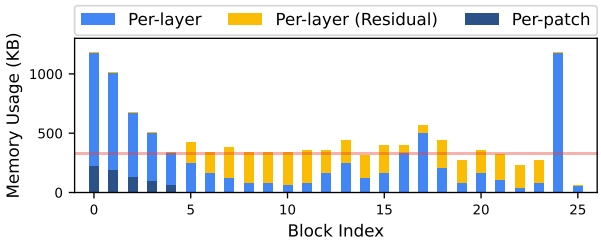
2 UNDERSTANDING THE MEMORY USAGE OF RESIDUAL NETWORKS

Higher image resolutions and increased channel numbers in DNNs can result in larger intermediate feature maps, and certain tasks like object detection may also incur high memory usage due to dense predictions [14]. This issue is compounded by the additional memory buffer used to store the intermediate feature maps associated with one or multiple residuals in SRAM for downstream operations (Figure 1). Residual connections can be categorized as *sum-based* or *concatenation-based*, depending on how they are aggregated, and different types vary in their memory usage. Inputs to a residual aggregation with different dimensions are typically aligned using a 1×1 convolution [7].

We examined the memory usage of ResNet50 and YOLOv5n, two popular DNNs, to understand the impact of residual connections. As shown in Figure 2(a), the initial blocks of ResNet50 have high memory usage, with three blocks exceeding the SRAM capacity (320 KB) due to the buffering of residual connections. This indicates that addressing the memory bottleneck in residual connections can reduce the peak memory



(a) ResNet50 (Input resolution: 112)



(b) YOLOv5n

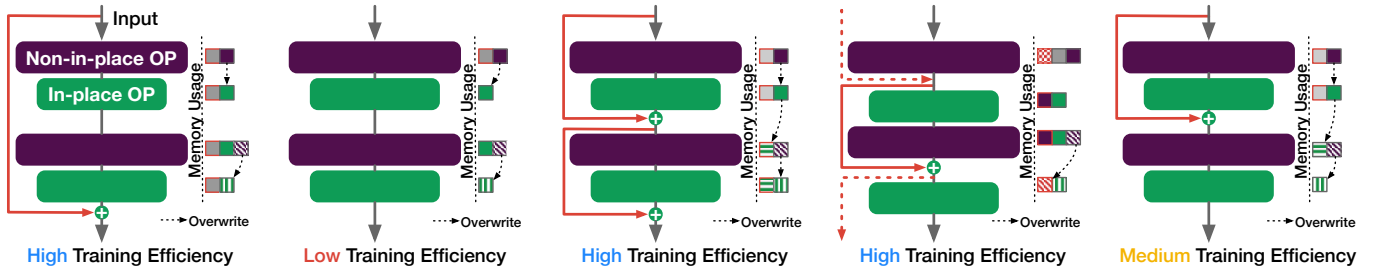
Figure 2: Block-level network memory usage breakdown

usage of the whole network. For YOLOv5n, as shown in Figure 2(b), peak memory usage can be reduced by employing depth-first inference [1, 13], where the output feature map of a layer is divided into smaller patches and convolution is performed on the receptive field of each patch, rather than computing layer-by-layer. This requires only one patch activation to be stored in SRAM, not the entire feature map. Note that depth-first inference is typically applied to multiple consecutive blocks when their feature map memory usage exceeds the SRAM constraint. To implement depth-first inference, sufficient buffer space in SRAM is needed to store the final output activations at the endpoint of a receptive field [13]. Therefore, we observe that the layer selected as the endpoint of each receptive field should still fit within the device's SRAM capacity. However, this is an issue for networks like YOLOv5n because the memory usage of most layers in the network exceeds the SRAM constraint. In such cases, reducing the high memory usage of residual connections enables the selection of a suitable endpoint for each receptive field, thus making techniques such as depth-first inference feasible.

3 RESIDUAL REMOVAL IS UNNECESSARY

Figure 3 illustrates the relationship between residual connections, memory usage, and training efficiency across various residual placement configurations. The typical configuration in Figure 3(a) improves training efficiency by preserving the gradient magnitude [7], but it requires an extra memory buffer to store intermediate feature maps. In contrast, removing residual connections (Figure 3(b)), eliminates the need for the extra buffer but degrades training efficiency [3, 12]. We observe that strategically placing residual connections within the network, as shown in Figure 3(c), can avoid the extra buffer without compromising training efficiency, as retaining the residuals still ensures efficient gradient propagation. However, improper configurations can lead to either no reduction in memory usage or reduced training efficiency. For instance, as depicted in Figure 3(d), shortened residual connections may still require the extra memory buffer, and partially connected residuals, as shown in Figure 3(e), may decrease training efficiency.

We make a key observation regarding how the behavior of operations affects memory buffer allocations. Specifically, an operation can



(a) Typical (b) Plain-style (c) Configuration-1 (d) Configuration-2 (e) Configuration-3

Figure 3: Comparing the memory buffer usage and model training efficiency of different residual connection placement configurations.

be performed in-place or non-in-place, depending on how memory is allocated for its input and output tensors. In an in-place operation, the output of the operation is written to the same memory location as the input, effectively overwriting the input data, and thus a separate memory buffer for the output is not required during computation. Common examples of such operations include pooling and batch normalization. In contrast, a non-in-place operation (e.g., convolution) writes the output to a new memory location, leaving the input tensor unchanged, thus requiring two separate memory buffers to store the input and output. This observation motivates us to investigate a new method that reorganizes residual connections in DNN models, taking into account the interdependency and types of operations spanning across the residuals, such that peak memory usage can be reduced without compromising model accuracy.

4 DERO: DEEP REORGANIZATION

We propose DEep ReOrganization (DERO), a systematic approach that uses simple heuristics to reorganize residual connections. The processing flow of DERO, as illustrated in Figure 4, involves reconstructing the residual connections of a given DNN model from scratch, in order to reduce its memory buffer usage, while limiting the impact on accuracy, subsequently followed by refining those affected operations to ensure valid computation. The reorganized model is then retrained before deployment. This process is fully automated, requiring no manual adjustments to generate the reorganized model. Note that DERO is orthogonal to existing techniques like model compression [19, 22], which reduce the model storage size, as well as inference reordering [1, 15] and receptive field redistribution [13], which reduce the peak memory consumption.

DERO addresses two design challenges related to residual reorganization. The first challenge is to ensure that the reorganization reduces memory requirements without compromising the model's training efficiency. This is non-trivial because inappropriate residual placement may not reduce memory usage but instead impact training efficiency only. The second challenge is to ensure that the reorganized model can still be validly computed without degrading model accuracy. This is difficult because using straightforward approaches to avoid tensor dimension mismatches may affect accuracy while also increasing the computation and memory overhead. We tackle the first challenge with dependency-aware residual reconstruction (Section 4.1), which strategically places residual connections to reduce memory usage while ensuring efficient gradient propagation. The second challenge is resolved with residual-aware operation refinement (Section 4.2), which makes low-cost changes to the affected operations to avoid invalid computations and accuracy loss.

4.1 Dependency-aware Residual Reconstruction

Residual reconstruction generates a new DNN model from scratch. As shown in Figure 4, the original model is first replicated without any residual connections, resulting in an equivalent new DNN model with the same operations in the original topological order. Next, each operation in the new DNN is iterated to identify the appropriate *start* and *end* points of new residual connections. An end point, specifically the position where a residual is aggregated, is determined based on the memory allocation behavior of the operation in order to avoid additional buffer usage. Starting each residual connection from the previous residual's end point allows efficient gradient propagation during training.

Residual End Point Selection. We leverage the fact that an operation can be classified as either in-place or non-in-place (Section 3), by exploiting its behavior in terms of whether it overwrites the input tensor's memory buffer or not. DERO inserts an end point immediately before each non-in-place operation, to eliminate the need for an extra memory buffer to aggregate the residual connection, thus ensuring the entire model uses only two memory buffers during inference. No action is taken for in-place operations as they overwrite their input without requiring extra buffer space. Whether an operation is considered in-place or non-in-place depends on its definition and the manner in which it is implemented by the inference engine, specifically with regard to whether the operation's input can be overwritten.

Residual Start Point Selection. To maintain training efficiency, DERO follows the identity mapping property [8] to select the start point for each residual connection. Here, identity mapping refers to the direct connection of the layer input to the layer output. DERO treats each residual's end point as a new start point for the next residual. Essentially, it constructs a new residual connection between two adjacent end points to ensure signals can propagate continuously from one layer to another during both the forward and backward passes. This approach enables gradients to flow through the network without degradation from operations, making it easier to train the reorganized model while preserving the training efficiency of the original model.

4.2 Residual-aware Operation Refinement

As a result of residual reconstruction, some residuals in the original model may have been removed and new residuals may have been added, potentially resulting in invalid tensor computations. Consequently, specific operations may encounter compatibility issues because of tensors with incompatible dimensions (i.e., either too large or too small), leading to invalid tensor computations. To ensure valid tensor computation while maintaining a parameter count similar to the original model, we employ *hyper-parameter redistribution*, where hyper-parameters are adjusted uniformly to scale down oversized tensors and scale up undersized tensors, followed by augmenting the aggregation operations.

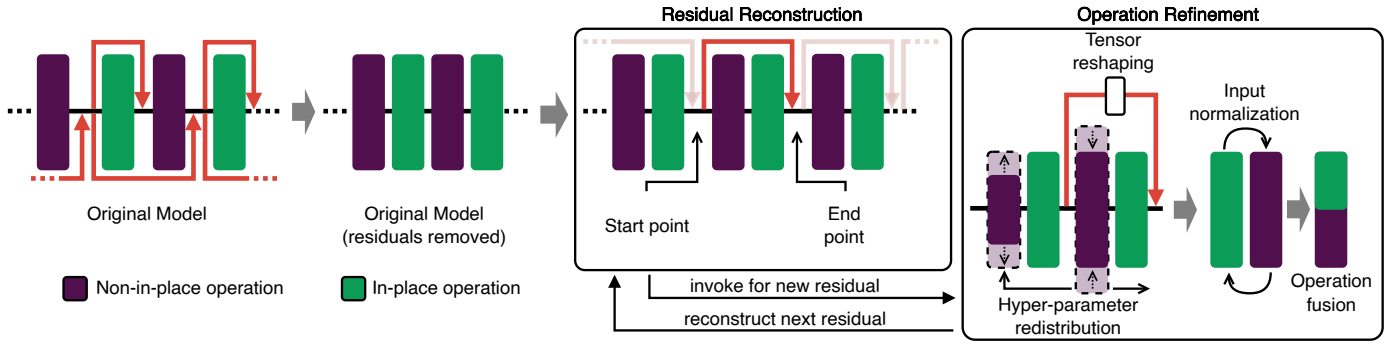


Figure 4: DERO processing workflow

Additionally, to further avoid model accuracy degradation, we swap specific operations to allow operation inputs to be normalized and fuse operations to prevent an increase in memory usage. As shown in Figure 4, these operation refinement steps are performed in the aforementioned order for each new residual added to the new DNN during residual reconstruction.

Ensuring Validity via Hyper-parameter Redistribution and Tensor Reshaping. A straightforward way to avoid tensor dimension mismatches is to drop invalid parameters of the affected operations, which may, however, impact model accuracy. Instead, we determine the number of invalidated operation parameters (e.g., in convolution filters) resulting from a removed residual connection and uniformly distribute them across all operations associated with that residual. This approach is influenced by findings in [13, 16], suggesting that models within a narrow search space exhibit similar performance. Thus, hyper-parameter redistribution allows the reorganized model to maintain a comparable structure and size to the original model, ensuring similar performance between them. On the other hand, aggregation-related dimension mismatches due to new residual connections are typically resolved with point-wise convolutions and batch normalization [7], but excessive use of them can hinder gradient propagation and increase computational costs. Instead, to ensure valid tensor dimensions during residual aggregation, we reshape those associated tensors using sum pooling and padding, which respectively decrease and increase the dimensions of the residual as needed, and combine the tensors with batch normalization. These low-cost operations allow the magnitude of gradients to be preserved during training.

Avoiding Accuracy Loss using Input Normalization with Operation Fusion. When new residual connections are added while retaining the same operation order as the original model, the input features of certain critical operations, such as convolution, will become non-normalized. Non-normalized input features may typically lead to slow convergence, unstable training, and poor generalization performance [6]. Input normalization can be ensured by performing batch normalization and ReLu activation prior to convolution [8], but this requires an additional memory buffer because the input of batch normalization needs to be buffered for the downstream residual aggregation. Hence, to avoid such an issue, we swap the order of batch normalization and convolution operations within the model and subsequently combine these two operations together. Specific utility functions available in deep learning frameworks (e.g., PyTorch) can be used to combine these two operations into a single one, where convolution is first performed to produce an output tensor, then its mean and variance computed along the channel axis are used to normalize the output tensor, and finally, a scaling factor and bias are applied to obtain the desired range of activations. This process is performed after hyper-parameter redistribution and tensor reshaping.

5 EVALUATION

5.1 Experimental Setup

Datasets and Networks. We evaluated DERO on two prominent large-scale benchmark datasets representing TinyML workloads, namely ImageNet for image classification and MSCOCO for object detection. For ImageNet, we used ResNet34 [7], ResNet50 [7], MCUNet [13], and DenseNet121 [10] as our benchmark models, while for MSCOCO we used YOLOv5n [5]. These models are widely recognized for their respective vision applications, each of which features a varying number of skip connections with different types and lengths, allowing for a comprehensive study of the impact of residual reorganization.

Baselines and Metrics. We compared DERO with four baselines: the original model with residuals, a plain-style version of the original model trained similarly to the original, and two recent plain-style approaches, namely *RepCONV* [3] and *JointRD* [12], which remove all residual connections and perform custom training strategies. Specifically, RepCONV features a unique model architecture that is decoupled during both training and inference, while JointRD utilizes a teacher model to train the plain-style model. All evaluated plain-style models do not require an additional memory buffer for residual connections. The performance metrics investigated encompass model accuracy, peak memory usage, and inference latency.

Training and Deployment. Models were trained on 8 NVIDIA GTX 2080Ti GPUs, with ResNet34/50, MCUNet, and DenseNet trained for 90 epochs and YOLOv5n for 300 epochs. Subsequently, they were quantized to INT8[1] and deployed on the STM32F746 (Arm Cortex-M7, 320 KB SRAM) MCU. As these models exceeded the storage capacity of the on-chip Flash memory, a 64 MB external Flash memory module (MT25QL512) was used. We developed a custom lightweight inference runtime based on CMSIS-NN, in a manner similar to TinyEngine [14]. Note that DERO is inference engine agnostic, but if optimized for more in-place operations, it may require less memory space for feature maps, making the memory requirement of residuals more dominant and thus enhancing DERO's benefits. Further details on the training settings, the DERO algorithm and visualizations of the reorganized evaluation models can be found on the DERO open source link [2].

[1]DERO increases residuals by 2 to 5 times, resulting in slight accuracy drops in overparameterized models (e.g., ResNet34/50) and moderate drops in compact models (e.g., MCUNet) after INT8 quantization. As per standard practice, the models were fine-tuned for 10 epochs before deployment to recover the lost accuracy.

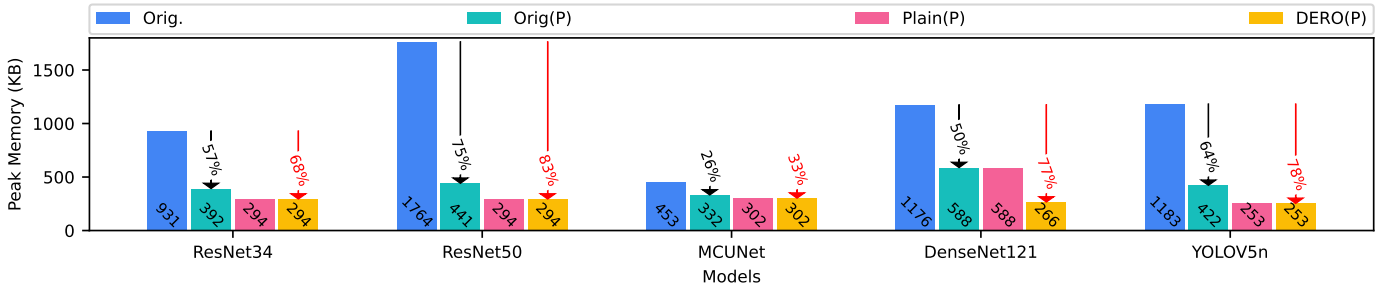


Figure 5: Peak memory consumption (P: inference is executed patch-by-patch)

5.2 Model Accuracy

Model	Orig.	Plain	RepCONV	JointRD	DERO
ResNet34	72.29	66.7	71.79	71.01	72.32
ResNet50	75.37	63.84	68.90	74.14	75.56
MCUNet	55.58	52.12	52.06	53.23	55.59
DenseNet121	73.62	-	-	-	71.55
YOLOv5n	27.40	22.40	-	-	25.90

Table 1: Summary of model accuracy (%)

The top-1 accuracy for ResNet34/50, MCUNet, and DenseNet, as well as the mean average precision (mAP) for YOLOv5n, is reported in Table 1. All the plain-style models exhibit poor training efficiency, resulting in significant accuracy degradation when compared to the corresponding original models (i.e., 0.5 to 11% accuracy loss). This discrepancy is particularly pronounced in deeper models like ResNet50 and DenseNet121, where residuals are crucial to avoid the vanishing gradient problem [7]. Figure 6 depicts the training process of ResNet50, highlighting the notable training inefficiency observed in plain-style models. Similar trends are observed in the training of the other evaluated models. Note that RepCONV and JointRD were originally designed for architectures like ResNet and MCUNet, and therefore both approaches can not be feasibly applied to train DenseNet121, as indicated by a dash in Table 1.

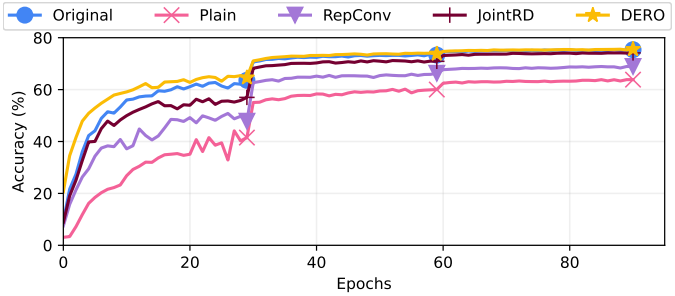


Figure 6: ResNet50 training efficiency

In contrast, DERO achieves similar accuracy to the original models, by retaining residuals to maintain training efficiency. In particular, DERO excels at matching accuracy when the original model features numerous short residuals, such as ResNet34/50 and MCUNet. However, for models with long residuals, such as DenseNet121, DERO experiences a 2% drop in accuracy. This is because DERO reconstructs DenseNet121's long, concatenation-based residual connections into shorter, sum-based connections, thus trading feature reusability across different layers for peak memory reduction. Unlike DERO, neither RepCONV nor JointRD is designed for object detection, so they cannot be feasibly applied to YOLOv5n. When compared to the original YOLOv5n, the plain-style model experiences an mAP loss of 5%. In contrast, DERO has an mAP drop of only 1.5%, primarily because YOLOv5n shares similar characteristics with DenseNet121, specifically long residuals and a mixture of concatenation and sum-based residual types.

5.3 Peak Memory Usage

Figure 5 shows the peak memory usage of the evaluated models when executed using layer-wise inference, as well as depth-first inference, which executes the network patch-by-patch [13]. Here, a memory reduction technique such as depth-first inference is necessary for deploying large models on tiny devices with limited SRAM capacity (Section 2). In the figure, the bars labeled Plain(P) represent the peak memory usage minimized by all evaluated plain-style approaches, including RepVGG and JointRD, when employing depth-first inference, while DERO(P) indicates the peak memory usage of DERO when using depth-first inference. In all cases, depth-first inference reduces the peak memory usage of original models by between 26% and 75%, and in most cases, plain-style approaches can further reduce the peak memory usage down to between 33% and 83% by entirely removing the residual connections. ResNet50 has the highest peak memory usage among the evaluated original models, and hence a larger memory reduction is achieved with depth-first inference. By contrast, MCUNet experiences a comparatively smaller reduction because it is already optimized for depth-first inference and tailored to the tight memory constraints of tiny devices.

DERO achieves the same peak memory usage as plain-style approaches for ResNet34/50, MCUNet, and YOLOv5n when executed using depth-first inference, as DERO reorganizes the residuals to avoid an additional memory buffer. Interestingly, DERO(P) significantly reduces the peak memory usage of DenseNet121, by 27% compared to Plain(P). This reduction is primarily because plain-style approaches rely on padding to maintain valid tensor computations after removing the residual connections in DenseNet121, which fails to reduce the peak memory consumption as the size of feature maps remains unchanged. By contrast, DERO performs hyper-parameter redistribution to ensure valid tensor computations, resulting in a decrease in the size of the original feature maps.

5.4 Inference Latency

Table 2 shows the inference latency of the evaluated models on the STM32F746 MCU. Note that the high inference latency observed across all deployments is primarily attributed to the costly data transfers from the external Flash storage. For execution using depth-first inference, the patch size was optimized for each individual approach,

given the device's SRAM capacity constraint. A smaller patch size leads to increased computation overhead and inference latency, and vice versa [13].

Model	Orig.	Orig(P)	Plain(P)	DERO(P)
ResNet34	-	201.4	166.8	167.0
ResNet50	-	-	169.7	169.9
MCUNet	-	7.9	6.4	6.4
DenseNet121	-	-	-	73.3
YOLOv5n	-	-	51.6	52.1

Table 2: Inference latency (seconds)

None of the original models are deployable when executed using layer-wise inference due to the device's SRAM capacity (as indicated by a dash in Table 2). However, the original ResNet34 and MCUNet become deployable when executed using depth-first inference (i.e., Orig(P)), although they can be executed faster using DERO(P), as DERO reduces the peak memory usage, allowing the use of larger patch sizes. In contrast, four models (ResNet34/50, MCUNet, and YOLOv5n) are deployable when plain-style approaches are employed with depth-first inference (i.e., Plain(P)). These models are also deployable under DERO(P) and achieve comparable latency to Plain(P), even though DERO does not compromise model accuracy (Section 5.2). However, DenseNet121 remains undeployable when using the plain-style approaches, as they cannot effectively reduce its peak memory usage (Section 5.3). By contrast, DERO considerably reduces the peak memory consumption of DenseNet121, consequently making it deployable. Note that DERO maintains a similar number of parameters and FLOPS as the original model due to hyper-parameter redistribution, except for DenseNet, where the FLOPs are nearly halved by replacing concatenation-based residuals with sum-based residuals.

6 DISCUSSION

DERO retains residual connections but may alter their number, structure (e.g., single/multiple-path), and aggregation types compared to the original model. As a result, while DERO can reduce peak memory usage to match that of plain-style models, networks like DenseNet and YOLO, which heavily rely on unique residual architectures, may experience a slight drop in accuracy (Section 5.2). Future work could explore strategies to preserve essential characteristics of the original model, for example, by selectively reorganizing only specific portions that affect peak memory usage.

Our evaluation demonstrates DERO's effectiveness on common convolutional networks like ResNet, DenseNet, and MCUNet. We anticipate that these results will apply to a broader family of networks that share similar architectural foundations. For example, models like MobileNet [17] and EfficientNet [18], which belong to the same model family as MCUNet due to their use of the same block structure (MBConv), are likely to benefit from DERO as well. In cases where networks lack residual connections that cause the memory bottleneck, we expect the peak memory usage of the reorganized model to remain unchanged compared to the original model. For non-convolutional architectures like Transformer networks [4], their memory bottleneck may lie in attention maps, so eliminating the additional memory buffer for residuals may not minimize the peak memory usage. However, DERO can still offer a viable solution for Transformers if other techniques can reduce the memory usage of attention maps, making residuals the new memory bottleneck.

7 CONCLUSION

This paper advocates that removing residual connections is unnecessary for minimizing peak memory usage. We propose the concept of deep residual reorganization, which reconstructs the residuals of a network by considering the memory allocation characteristics and processing behavior of operations to minimize peak memory usage, without significantly compromising training efficiency and model accuracy. Targeting tiny microcontrollers, we demonstrate that even a simple heuristic-based approach like DERO can facilitate the deployment of complex models like DenseNet, which were previously considered undeployable due to high peak memory consumption.

Our work paves the way for future research that leverages residual reorganization, potentially in combination with neural architecture search frameworks like TinyNAS [14], to find models that do not require additional memory space for residuals, thereby deriving higher-accuracy models within a given memory constraint and improving the accuracy-latency trade-off for MCU-class models.

ACKNOWLEDGEMENT

This work was supported in part by National Science and Technology Council, Taiwan, under Grant NSTC 110-2222-E-001-003-MY3 and by Academia Sinica under Grant AS-IA-113-M04-ASSA.

REFERENCES

[1] Manoj Alwani, Han Chen, Michael Ferdman, and Peter Milder. 2016. Fused-layer CNN Accelerators. In *Proc. of IEEE/ACM MICRO*. 1–12.
[2] DERO Open Source Project. 2024. https://github.com/EMCLab-Sinica/DERO.
[3] Xiaohan Ding, Xiangyu Zhang, Ningning Ma, Jungong Han, Guiguang Ding, and Jian Sun. 2021. RepVGG: Making VGG-style ConvNets Great Again. In *Proc. of IEEE CVPR*. 13733–13742.
[4] Alexey Dosovitskiy et al. 2021. An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale. In *Proc. of ICLR*.
[5] Glenn Jocher et al. 2022. ultralytics/yolov5:v7. 0-YOLOv5 SOTA Realtime Instance Segmentation.
[6] Jonathan Frankle, David J Schwab, and Ari S Morcos. 2021. Training BatchNorm and Only BatchNorm: On the Expressive Power of Random Features in CNNs. In *Proc. of ICLR*. 1–28.
[7] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *Proc. of IEEE CVPR*. 770–778.
[8] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Identity Mappings in Deep Residual Networks. In *Proc. of ECCV*. 630–645.
[9] Chien-Chin Huang, Gu Jin, and Jinyang Li. 2020. Swapadvisor: Pushing Deep Learning Beyond the GPU Memory Limit via Smart Swapping. In *Proc. of ACM ASPLOS*. 1341–1355.
[10] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q Weinberger. 2017. Densely Connected Convolutional Networks. In *Proc. of IEEE CVPR*. 4700–4708.
[11] Weiwen Jiang et al. 2019. Accuracy vs. Efficiency: Achieving Both Through FPGA-implementation Aware Neural Architecture Search. In *Proc. of IEEE/ACM DAC*.
[12] Guilin Li et al. 2020. Residual Distillation: Towards Portable Deep Neural Networks without Shortcuts. In *Proc. of NeurIPS*. 8935–8946.
[13] Ji Lin, Wei-Ming Chen, Han Cai, Chuang Gan, and Song Han. 2021. MCUNetV2: Memory-Efficient Patch-based Inference for Tiny Deep Learning. In *Proc. of NeurIPS*.
[14] Ji Lin, Wei-Ming Chen, John Cohn, Chuang Gan, and Song Han. 2020. MCUNet: Tiny Deep Learning on IoT Devices. In *Proc. of NeurIPS*. 11711–11722.
[15] Xuan Peng et al. 2020. Capuchin: Tensor-based GPU Memory Management for Deep Learning. In *Proc. of ACM ASPLOS*. 891–905.
[16] Ilija Radosavovic, Raj Prateek Kosaraju, Ross Girshick, Kaiming He, and Piotr Dollar. 2020. Designing Network Design Spaces. In *Proc. of IEEE CVPR*. 10428–10436.
[17] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. MobileNetV2: Inverted Residuals and Linear Bottlenecks. In *Proc. of IEEE CVPR*. 4510–4520.
[18] Mingxing Tan and Quoc Le. 2019. EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks. In *Proc. of ICML*.
[19] Kuan Wang, Zhijian Liu, Yujun Lin, Ji Lin, and Song Han. 2019. HAQ: Hardware-Aware Automated Quantization With Mixed Precision. In *Proc. of IEEE CVPR*. 8612–8620.
[20] Olivia Weng et al. 2023. Tailor: Altering Skip Connections for Resource-Efficient Inference. *ACM TRETS* (2023).
[21] Guodong Zhang, Aleksandar Botev, and James Martens. 2022. Deep Learning without Shortcuts: Shaping the Kernel with Tailored Rectifiers. In *Proc. of ICLR*. 1–25.
[22] Tianyun Zhang et al. 2021. A Unified DNN Weight Pruning Framework using Reweighted Optimization Methods. In *Proc. of IEEE/ACM DAC*. 493–498.