Augmenting Legacy Networks for Flexible Inference

Jason Clemons[1][0000−0001−5533−417X], Iuri Frosio[1], Maying Shen[1], Jose M. Alvarez[1], and Stephen Keckler[1]

NVIDIA Corporation, Santa Clara, CA, USA

Abstract. Once deployed in the field, Deep Neural Networks (DNNs) run on devices with widely different compute capabilities and whose computational load varies over time. Dynamic network architectures are one of the existing techniques developed to handle the varying computational load in real-time deployments. Here we introduce LeAF (Legacy Augmentation for Flexible inference), a novel paradigm to augment the key-phases of a pre-trained DNN with alternative, trainable, shallow phases that can be executed in place of the original ones. At run time, LeAF allows changing the network architecture without any computational overhead, to effectively handle different loads. LeAF-ResNet50 has a storage overhead of less than 14% with respect to the legacy DNN; its accuracy varies from the original accuracy of 76.1% to 64.8% while requiring 4 to 0.68 GFLOPs, in line with state-of-the-art results obtained with non-legacy and less flexible methods. We examine how LeAF's dynamic routing strategy impacts the accuracy and the use of the available computational resources as a function of the compute capability and load of the device, with particular attention to the case of an unpredictable batch size. We show that the optimal configurations for a given network can indeed vary based on the system metrics (such as latency or FLOPs), batch size and compute capability of the machine.

Keywords: Dynamic architecture, real-time systems, legacy, fast inference

1 Introduction

Deep Neural Networks (DNNs) deliver state-of-the-art results in a wide range of applications. Very often, however, they are characterized by a high inference cost [2,28]; when pushed to their limit, DNNs use a large amount of computational resources for small gains in model accuracy. These high resource costs pose a barrier to the adoption of state-of-the-art DNNs in the field.

The reasons for these inefficiencies are varied. Large or deep DNNs require high compute at inference time. Efficient, static architectures like residual connections [6,9,13,14], pixel shuffle [23] or pruning [1,10,17,26] partially alleviate this issue, but low utilization of computational resources is still possible when a DNN *overthinks* about easy cases [25,32]. The complexity of dealing with inefficient DNNs is further increased when taking into account deploying models onto

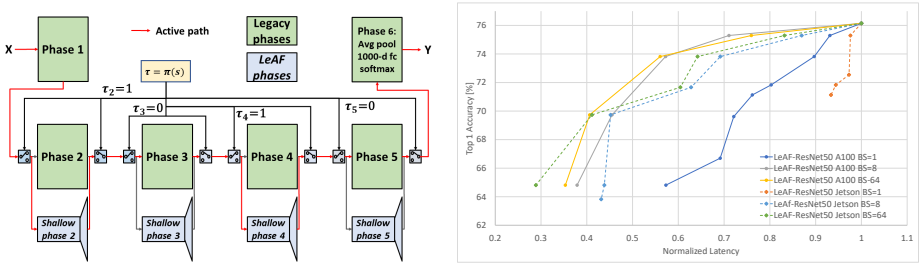

Fig. 1. The left panels shows LeAF-ResNet50, where shallow phases augment the legacy DNN. The external routing policy, $\boldsymbol{\tau} = \pi(\boldsymbol{s})$, is disentangled from LeAF-ResNet50: it can be any user-defined function of the state of the compute device, $\boldsymbol{s}$; here, the shallow phases 2 and 4 are executed together with the legacy phases 3 and 5. The right panel compares LeAF-ResNet50 performance on an A100 GPU(solid lines) and Jetson Xavier (dashed lines) as the batch size BS is varied between 1,8 and 64. The latency is normalized to the latency for the given batch size when using the base network. Notice that the curves are different for different batch sizes.

a wide range of devices, as different computing systems require diverse optimal implementations of the same DNN [7,25]. Even the computational load on the same device changes over time [11]. These issues can be alleviated by affording a DNN the ability to vary its architecture *on the fly* based on current system state. Thus, providing this capability is highly desirable for practical implementation and field deployment.

A variety of solutions to create DNNs with *variable* architecture exists. *Dynamic* DNNs are models that incorporate a gating policy function to route the input towards complementary paths with varying compute complexity [21,25,27], each specialized during training to handle different classes of data, thus making dynamic DNNs highly accurate and efficient. *Anytime prediction* DNNs [12,24] use early exits that allow selecting among a limited set of compute/accuracy compromises. All-for-one [2] DNNs are initially trained to be *flexible*, *i.e.*, such that sub-networks can be run while keeping a high accuracy; a single DNN configuration is then selected for deployment, based on the target device.

The analysis of networks with dynamic routing reveals interesting aspects in terms of real performance and efficiency on deployed systems. Different paths are characterized by different compute requirements and thus performances, that also change with the device and its current load status. In other words, when optimizing performance for deployment, all factors including not only the target model performance, but also the batch size and the device compute capability must be taken into account.

We introduce a DNN augmentation paradigm that provides full real-time control of the DNN configuration, and preserve the legacy network. Our technique produces a *flexible* DNN whose architecture can be changed on the fly accordingly to the constraints imposed by the state of the compute device, and achieving accuracy/compute cost compromises in the same ballpark of that of

existing state-of-the-art methods. We explore using this flexible inference and characterize the performance.

In this work we provide several contributions:

- We introduce LeAF(Legacy Augmentation for Flexible inference), a paradigm to transform pre-trained, static DNNs into *flexible* ones, while also allowing execution of the legacy DNN. LeAF networks disentangle the gating policy problem from the computational aspects, allowing selection of the optimal accuracy/compute cost compromise on the fly.
- We perform a thorough analysis of the computational aspects of LeAF-DNNs for different tasks, network architectures and compute devices. We analyze the relation between FLOPs and latency and highlight the importance of the batch size for the deployment of effective DNNs. The validity of these insights extends beyond LeAF, showing for instance that results reported in literature for batch size 1 do not generalize naively to larger batch sizes.
- We demonstrate that DNNs with variable architectures allow many possible configurations, but few of them belong to the Pareto set in the accuracy / compute cost plane; we leverage this by fine-tuning LeAF-DNNs only for those configurations, but the insight applies again to a larger class of variable architecture networks.

2 Related Work

There are a few key differences in the attempts to speed up and reduce the energy consumption of DNNs. The first major difference is the one between *static* and *variable* network architectures. These can be further subdivided into *anytime prediction networks* with early exits and architectures with variable data-flow paths (generally referred to as *dynamic networks* in literature). An orthogonal classification can be made between methods that disentangle the (or "use an *external*") routing policy from the task at hand (like LeAF) and those that do not (or have an *internal* routing policy).

2.1 Static architectures

DNN training can be made faster or less energy hungry by the adoption of skip/residual connections that help gradient propagation [6,9,11,14], but this option does little to save energy at inference time - on the contrary, it may favour the adoption of deeper networks characterized by costly inference. To reduce latency or energy consumption at inference time, software implementations making an effective use of the underlying hardware (like pixel shuffle [23]) or reducing the computational cost of the DNN operations (as for reduced precision [3,33]) exist. Another widely used possibility when deploying DNNs is offered by pruning. Weight pruning methods such as SNIP [15] or N:M pruning [26] remove individual parameters to induce sparsity, but need hardware support to get the actual acceleration, whereas channel pruning methods are more practical on modern hardware (e.g., GPUs). Typically, pruning is formulated as the

resource (memory, computes, latency, etc.) constrained problem of selecting an optimal sub-network. Some channel pruning methods remove the least salient channels until reaching the desired cost in terms of FLOPs [16,17,30], but these same method hardly achieves optimal latency because of the documented discrepancy between the FLOPs and latency on complex devices. Platform aware methods achieves better latency reduction performances. HALP [22] estimates the channel latency cost using prior knowledge of the target platform, and consequently minimizes the loss drop under the user-defined latency constraints. NetAdapt [29] iteratively removes channels until reaching the latency goal with empirical latency measurements. However, these result in a single sub-networks that are device-specific by definition, and are also constraint-specific. Thus, storing all of the sets of parameters associated with different network training, each aimed at satisfying a different constraint, is possible but impractical because of the large memory footprint and the overhead associated with moving such a large amount of data in case of an architecture switch.

2.2 Variable architectures

DNNs with a variable execution path are implemented in different flavors. One is to have the input automatically routed towards different blocks of the DNN, each specialized during training to handle different classes (*e.g.,* to classify pets or vehicles). These DNNs are referred to as *dynamic* networks. One example is the outrageously large neural network [21] that can be seen as a large mixture of experts each with small inference computational cost, but overall characterized by an huge memory footprint. Other dynamically routed networks, like SkipNet [27] or ConvNet-AIG [25], are designed such that, for a given input, different set of filters or layers are skipped. Slimmable networks [31] also allow skipping set of filters, but do not include any automatically routing mechanism, and therefore cannot be referred to as a dynamic network. Dynamic DNNs can surpass the original network in terms of accuracy at a lower computational cost, but training is often non trivial as the gating function is non-differentiable, and it also introduces a small computational overhead. Furthermore, the network architecture and training does not support the legacy preservation of the original DNN, contrary to LeAF.

A different type of DNNs with variable architecture is represented by *anytime prediction* networks, where early exit blocks can be added on top of an existing architecture as in the case of Branchynet [24] or Patience-based Early Exit [32]. Measuring the confidence of the network output at an early exit allows preventing the network overthinking and thus getting a high confidence result in a short amount of time. Some anytime prediction DNNs adopt an ad-hoc architecture that leverages information at multiple scales (see the Multi-Scale DenseNet (MSDNet) [12]), while other authors propose changing the cost function for anytime prediction networks to handle the different levels of prediction noise generated at different depths [11]. Dual dynamic Inference DDI [28] mixes anytime prediction networks and dynamic bypasses, using an LSTM network for gating functions.

Overall, variable architecture DNNs achieves better performance / compute cost compromises than static ones, but they have their own drawback. Legacy is often not considered, as we do in LeAF. Anytime prediction networks have limited flexibility, as the number of exits scales less than linearly with the network depth, while in LeAF it is proportional to the number of combinations of the shallow phases although only a fraction of these is useful in practice as we show in Section 3.2. Furthermore, early exits are not easily integrated into non-purely sequential architectures, like a U-nets. Lastly, the advantage claimed for variable architecture DNNs often refer to edge devices working with batch size 1. This is however only one of the interesting cases for latency reduction and/or energy saving: in some applications, like stereo vision, the same DNN may process image pairs, while servers are often required to handle large and possibly unpredictable batch sizes.

2.3 Internal vs. external routing policy

Dynamic networks include the computation of the data routing path into their architecture [7] - we say that they have an *internal routing policy*. Anytime prediction DNNs offer the possibility to automatically stop the execution based on the estimated confidence at a given exit (internal routing policy), but the user also has the possibility to decide a-priori the desired exit and the resulting performance / compute cost compromise, thus adopting an *external routing policy*. In LeAF, we disentangle the problem of creating a variable network architecture from that of selecting the best compute path for a given batch of any size. This is something that we have in common with slimmable networks [31], probably the closest work to ours, and one of the few aimed at creating a *flexbile* network architecture, where the routing policy can be explicitly controlled by the user and changed on the fly to meet the time-varying constraints of the compute device. The One-For-All approach [2] partially achieves the same level of flexibility by adopting a network with an ad-hoc architecture that is trained to maximize the accuracy of any of the possible sub-networks (10^{19}) that can be extracted from it. This offers the flexibility of easily picking the sub-networks with the best performance / compute cost compromise for any target device. However, since the sub-network has a static architecture with no routing option, any form of flexibility is eventually lost once the DNN is fine-tuned and deployed in the field. None of the aforementioned methods consider the legacy aspect in any way, as LeAF-networks do.

3 Legacy Augmentation for Flexible Inference (LeAF)

3.1 LeAF overview

Many DNNs include sets of layers with similar characteristics, that we call *phases*. For example, Resnet [8] has 6 phases (Fig. 1), Mobilenetv2 [20] has 12. LeAF augments the legacy DNN with alternative, user-selectable *shallow*

Table 1. LeAF training parameters for ResNet-50 and Mobilenetv2. We use SGD with step scheduler for the learning rate and batch size 32

Network	Training step	Learning Rate	Parameters	Epochs	Shallow phases
LeAF-ResNet-50	base-training	.1	lr_step=10, lr_gamma=.1	60	$2^4 = 16$
LeAF-ResNet-50	fine-tuning	.1	lr_step=10, lr_gamma=.1	60	6
LeAF-Mobilenetv2	base-training	4.5e-3	lr_step=1,lr_gamma=.96,weight_decay=4e-6	80	$2^5 = 32$
LeAF-Mobilenetv2	fine-tuning	4.5e-4	lr_step=1,lr_gamma=.97, weight_decay=4e-6	100	10

phases that can be executed in place of the original ones. Each shallow phase is designed to respect the original size of the input and output tensors, whereas internally the number of channels is a fraction of the original one. Thus, LeAF can be applied to a variety of network shapes such U-nets, ResNets and GANs. For a flexbile, LeAF-DNN with n phases, the number of possible execution paths is then 2^n, including the original one. During execution, any policy function $\tau = \pi(s)$ can be used to control the set of active shallow phases (Fig. 1), where s is the current state of the system (and could, for instance, include the current load or target accuracy). As τ is a binary vector, reconfiguration overhead is minimal allowing configuration updates in real time for any processed batch.

3.2 Training LeAF

To create a LeAF-DNN, we start from a pre-trained DNN with parameters θ that are frozen while we train the parameters ω of the shallow phases. This significantly decreases the overhead of training a DNN with multiple paths, while also allowing to continue to execute the legacy network. To preserve the accuracy of the legacy DNN, we also freeze any statistics in the batch normalization layers.

We train LeAF-DNNs using the adaptive loss in Eq. 1 based on [11] [31] where $J(\omega, \tau_i; \theta, X, y)$ is the traditional loss (*e.g.,* cross entropy) computed for the execution profile τ_i, whereas α_i is a multiplicative factor corresponding to the fraction of time a path is expected to run. $E[J(\omega; \theta, \tau_i; X, y)]$ is the sample average computed over the last 100 training iterations and is treated as a constant re-weighting factor during training. In our experiments we set $\alpha_i = 1/|\tau|$ (therefore assigning to any path has the same probability of being executed).

$$J_{tot\_adaptive}(\omega; \theta, X, y) = \sum_{i=0}^{|\tau|} \frac{J(\omega, \tau_i; \theta, X, y)}{E[J(\omega, \tau_i; \theta, X, y)]} \cdot \alpha_i, \tag{1}$$

Before training, we initialize the weights ω of the shallow phases by sampling from the original weights θ, as we found this to be more stable. Training is then performed in two steps, using the parameters in Table 1. In the base-training step, we use all the 2^n configurations of the LeAF-DNN, and set $\alpha_i = 1/(2^n)\ \forall i$. In the forward pass, for each batch we loop over all the configurations and accumulate the loss in Eq. 1 before doing the optimization step[1].

[1] We note that if the number of configurations becomes large, it may be more efficient to randomly sample the configuration for each mini-batch.

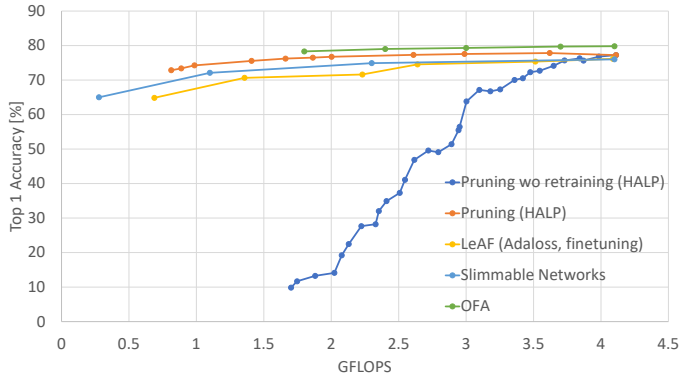

Fig. 2. Accuracy vs. FLOPs for various dynamic and flexible network methods including pruning and LeAF.

Once the base-training step is complete, we measure the accuracy and compute cost of each configuration, including a target batch size[2,3]. Fig. 3 shows the accuracy vs. FLOPs for LeAF-Resnet-50 on Imagenet for batch size 256. Some of the network configurations achieve a sub-optimal accuracy/FLOPs ratio: including these in the cost function in Eq. 1 is detrimental of the final accuracy. Therefore, we identify the configurations that reside in the Pareto set (Fig. 3), set $\alpha_i = 0$ in Eq. 1 for all those configurations that do not, and proceed with the fine-tuning training phase. To identify that a configuration is in the Pareto set we iterate over all the configurations and determine if there another configuration with a lower system cost and higher model performance/accuracy. If no such other configuration exists then the this configuration is considered in the Pareto set. We eventually store the Pareto set configurations with their final accuracy and costs in a look-up-table that can be easily used to implemented the policy gating function $\tau = \pi(s)$.

Fig. 2 shows the results of various techniques for producing flexible networks. It can be seen that LeAF results have similar performance to other techniques. Thus we choose to use LeAF as a proxy to understand the impact of various system characteristics on the execution of models. We believe the similarities of the techniques makes the insights applicable to other dynamic network executions.

[2] The compute cost can be derived analytically in case of FLOPs, or experimentally in case of latency, energy or power consumption.

[3] When the compute cost is measured in FLOPS, the batch size will be normalized away. Other systems cost metrics (*e.g.,* latency) may be a function of the batch size, as detailed in the Results Section.

4 Evaluation

LeAF provides a novel solution to augmenting pre-trained models to generate flexible networks. The LeAF-DNNs with variable architecture can be used to adapt the network working point based on system performance and achieve performance / compute cost compromises similar to those of similar techniques. For LeAF-ResNet-50 we show a top-1 accuarcy of 64.8% using approximately .69 GFLOPs with the same model that contains the weights of original ResNet-50, and 76.146% at 4 GFLOPs (legacy network). This comes at a cost of 13.8% more memory for the parameters in the model. We show that we can apply LeAF to efficient networks such as MobileNetV2 for as little as 7.7% memory overhead to store the weights.

4.1 Methodology

We apply LeAF to different network architectures to measure its effectiveness and to study in detail the computational behavior of dynamic neural networks. In particular, we consider as representative case studies the widely used ResNet-50 and MobileNetV2 for image classification on Imagenet [5], each augmented with LeAF.

To train, we use NVIDIA DGX systems with either 8 Tesla V100-DGXS-16GB GPUs or 8 A100-SXM-80GB GPUs, with PyTorch 1.10 [19], CUDA 11.3 and cuDNN 8.2.2. We use pretrained DNNs from Torchvision [19] as legacy networks for ResNet-50 and MobileNetV2.

For each network we grouped the layers into the phases as shown in Fig. 1. For ResNet we create shallow phases by taking the ResNet residual layers in *conv*2_*x*, *conv*3_*x*, *conv*4_*x* and *conv*5_*x* and using a fraction of the original channel count. We add a 1×1 convolutional layer to the end of each shallow phase to return the channel count back to its size in the original phase, and allow switching between the shallow and original phase.We adopt a similar grouping strategy for MobileNetV2 where we augment each of the 7 bottleneck groups (but the first one) with shallow phases. In our evaluation we use shallow phases containing 25% of the channels in the original phases. The shallow phases require an additional 13.8% parameters in ResNet-50 and only 7.7% in MobileNetV2. The lower amount in MobileNetV2 is due to the impact of the scaling factor on the expansion factors in the inverted residuals. We then train each LeAF-network with the procedure in the former Section and the meta parameters in Table 1.

To measure the latency on a high-end GPU, we use a system with an AMD EPYC 7742, 512 GB RAM and Ampere based A100-SXM-80GB GPU. We measure the average GPU execution time through the pyTorch profiler and report it as latency. We ensure the clocks are locked to stock values.

Beyond evaluation on the A100 GPU, we also measure the performances of LeAF-ResNet-50 on a Jetson Xavier NX system with 384 CUDA cores, maintaining the FP32 precision of the network. We lock the clocks of the GPU to 1.1GHz for this evaluation. We vary the batch size from 1 to 256 by powers of 2.

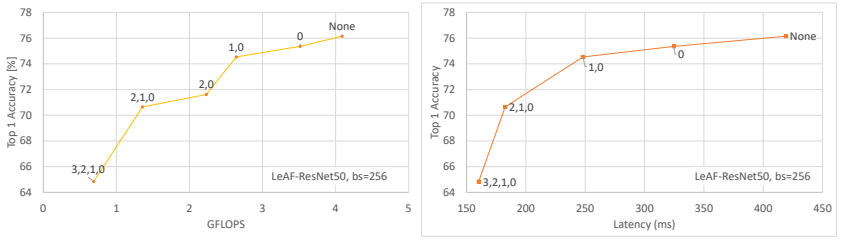

Fig. 3. Left: Accuracy vs. GFLOPs on a A100 GPU for LeAF-ResNet-50 for a batch size of 256. Points are labeled with the active set of shallow phases. Right: Accuracy Vs Latency on the same GPU for LeAF-ResNet-50 for a batch size of 256.

In the following, we perform a detailed analysis on the relation between FLOPs, latency and batch size for LeAF-ResNet-50 and then move to the evaluation of LeAFon MobileNetV2 for image classification.

4.2 LeAF-ResNet-50: On High End GPU

Fig. 3 shows the Top-1 accuracy vs. FLOPs for LeAF-Resnet-50 in the left plot and Top-1 accuarcy vs. latency in the right plot for a batch size of 256 on an A100 GPU. It can be seen that the plots have a similar but distinctly different shape. This demonstrates that while GFLOPs can be used as a proxy for latency it is better to use latency directly. This can be further illustrated by examining the x-axis distance of the configuration with $\{0\}$ to the neighboring configurations of $\{None\}$ and $\{1, 0\}$. In the GFLOPs plot (left) configuration $\{0\}$ is closer to $\{None\}$ while in the latency plot (right) it is closer to configuration $\{1, 0\}$. Furthermore the GFLOPs plot includes configuration $\{2, 0\}$ on the Pareto frontier while that point is not included when measuring latency directly.

This reinforces the idea, already emerged in previous works [22][4][29], that latency and FLOPs are not completely interchangeable on complex, parallel system like GPUs. For example, an underutilized GPU can actually provide more parallel FLOPs without increasing the execution time by utilizing more of the hardware. Furthermore, the scheduling of work on a GPU is complex and can have unexpected interactions. This has an impact on the development of truly flexible architectures for DNN: depending on the system constraints to satisfy, one has to build the correct Pareto set for the problem in hand.

We bring to the attention of the reader another phenomenon that, to the best of our knowledge, has not been highlighted before. Fig. 4 shows that, for varying batch size, the Pareto set in the accuracy vs. latency space varies as well, that may be (at least at first thought) unexpected. We believe the reason is again the complexity of a GPU system, whose behavior in terms of latency is nonlinear with respect to the occupancy of the computational cores. As for a given batch size a shallow phase may saturate the compute capability of a GPU while another may not, increasing the batch size may sometime (but not always) lead to zero latency overhead when switching from the shallow to the legacy phase. Considering that

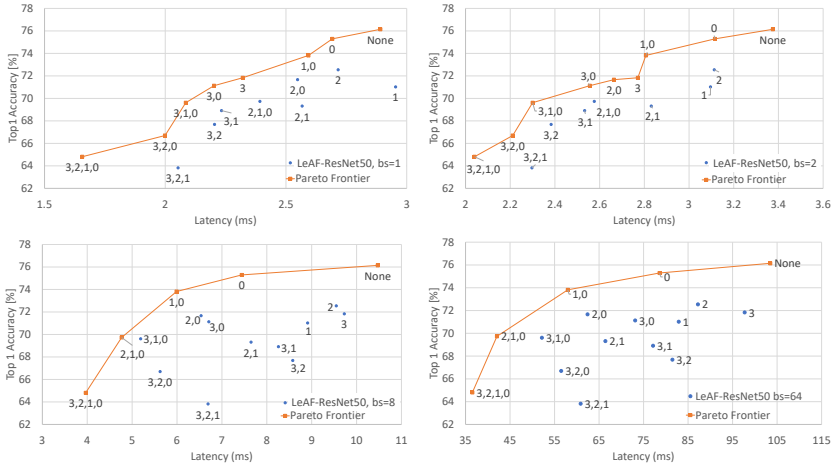

Fig. 4. Accuracy vs. latency for LeAF-ResNet-50 for batch size (bs) of 1, 2, 8, and 64, after base-training. Each point is labeled with the set of active shallow phases.

the GPU task scheduler is not under direct control, predicting the latency based on the configuration τ and batch size becomes a hard exercise (that is one of the reasons why in OFA [2] latency is predicted through an additional DNN). Therefore, when deploying a LeAF-DNN in the field and optimizing for latency (or power, energy, and so on), one should select the working batch size *a priori* and estimate the Pareto set accordingly to it. Alternatively, (although we did not explore this possibility here), one could also create a LeAF-DNN that is trained to deliver the highest accuracy for the configurations and batch sizes with the highest system performance.

4.3 LeAF-ResNet-50 Latency On Mobile

Fig. 5 shows the Pareto sets in the accuracy vs. latency space at batch sizes of 1, 8, 16 and 64 when running on the Jetson Xavier NX. Similar to what was already reported for the high-end A100 GPU, we notice that in this case the Pareto set also varies as a function of the batch size for smaller batch sizes. Fig. 5 shows that the relative positions of the configurations in the accuracy vs. latency space are less stable for lower batch sizes as demonstrated by the difference in the Pareto sets and relative configuration locations for the batch sizes 1 and 8 when compared to batch sizes 16 and 64. However, it can be seen that the relative position of the configurations stays consistent between batch sizes 16 and 64. Though not shown here, our experiments show that this relative positioning continues for higher batch sizes as well. Thus, for low batch size on this platform, it would be best to use different configurations for different batch sizes when optimizing for the latency. For larger batch size that fully utilize the GPU, the relative system performance stabilizes.

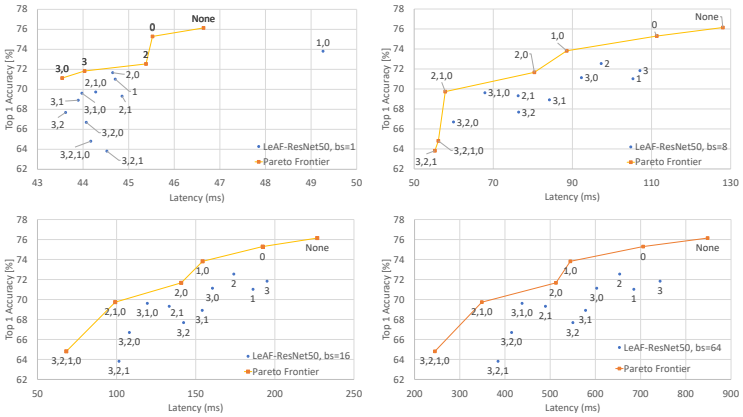

Fig. 5. Accuracy vs. latency for LeAF-ResNet-50 on an Jetson Xavier NX for batch size (bs) of 1, 8, 16, and 64 after base-training. Each point is labeled with the set of active shallow phases.

The upper right panel of Fig. 5 shows an example of the complex interactions of scheduling work on these systems. The configuration $\{1, 0\}$ repeatably took longer to execute than the base configuration. We believe this to be caused by an interaction between the kernel sizes, run-time framework and work scheduling.

To investigate the computational aspects of LeAF-ResNet-50 in more in detail on this device, we used the nvprof profiling tool [18] and analyzed the load on the GPU on Jetson Xavier NX system when running LeAF-ResNet-50. We found that increasing the batch size of the GPU leads to higher utilization metrics. The achieved *occupancy* is a measure of the utilization of the GPU's full compute capability, whereas the simultaneous multiprocessor (SM) *efficiency* is a measure of how many of the SMs are running a given kernel. For batch size 8, these two metrics are 46% and 89% respectively, meaning that almost 90% of the SMs are active, while they are lower for smaller batch sizes ($< 40\%$ and $< 85\%$, respectively). For batch sizes larger than 8 the processing becomes more serialized leading to a latency increase that is more predictable and the Pareto set no longer changes. We find that at batch size 64 and above, 95% of the SMs are active further showing that more compute will be serialized. This is consistent with our hypothesis that the Pareto set stops changing as the GPU compute capability is saturated.

Overall, we found that the Pareto set is a function of the batch size both on high-end GPUs (like the A100 tested in the main paper) and on GPUs with smaller compute capabilities (like the Jetson Xavier NX tested here). Thus, when trying to deploy a network without having to retrain, it is valuable to be able adjust the network's architecture dynamically based not only on the specific device and its load conditions, but also considering the specific application and the expected distribution of the batch size so that we can optimize the system performance. For example, mobile platforms may use a single camera or stereo camera as inputs. Depending on which is used, the optimal flexible network configuration can change. This can be extended easily to 8 cameras in an au-

tonomous vehicle application where there may be different LeAF configurations used to ensure optimal system performance.

4.4 Analysis As Function of Device

Our analysis of the LeAF ResNet-50 network shows different Pareto sets based on the device that is being used to run the network as noted by comparing Fig. 4 and Fig. 5. For instance, when running on the Jetson Xavier NX and increasing the batch size from 1 to 256 by powers of 2, we found that above a batch size of 8 the shallow phase set $\{2, 0\}$ is present in the Pareto set. However, when running the same experiment on the A100 GPU, we found that $\{2, 0\}$ is only present in the Pareto set for a batch size of 2. Furthermore, we can see that in Fig. 5 the Pareto set reaches a steady state with 6 shallow phase configurations for the mobile system as the batch size increases. However, for the A100 GPU in the main paper, the Pareto set reaches a steady state with 5 configurations as the batch size increases. The difference is the shallow phase configuration $\{2, 0\}$. It can also be seen that the points that do not belong to the Pareto set form a different relative shape between the two devices. This shows that optimal configuration for the same network and batch size can vary based on the device. This is due to the fact that different phases saturate the capabilities of different devices at different thresholds.

4.5 LeAF on MobileNetV2

We apply LeAF to MobileNetV2 to investigate its adoption on NAS-based DNNs that are already targeted at efficient inference. As shown in Fig. 6, before fine-tuning the top-1 accuracy of LeAF-MobileNetV2 goes from 71.79% for the legacy configuration at 9.6 GFLOPs to 50.05% at 4.75 GFLOPs when all the five shallow phases are active. The Pareto set contains 10 configurations (out of the 32 possible execution paths) confirming that, in a DNN with variable architecture, not all the paths are equally important. After fine-tuning the Pareto set configurations, we observe an average increase of 2.4% in terms of accuracy (green triangles in Fig. 6), that is more pronounced for low-compute configurations.

Fig. 7 shows the normalized latency and FLOPs for the different configurations of LeAF-MobileNetV2 for batch size 1 and 256 on an A100 GPU. The figures shows a stronger latency / FLOPs correlation (64% vs. 56%) for a larger batch size. The fact that, for batch size 1, a significant decrease in FLOPs does not correspond to a significant decrease in latency suggests that the system is likely underutilized, while for larger batch sizes the occupancy is higher and, since operations begin to be serialized even on a GPU, the scaling between latency and FLOPs tend to become more linear.

The results on LeAF-Mobilenetv2 show that applying LeAF to DNNs designed to be already fairly efficient led to a larger degradation in performances (when compared to the results obtained for ResNet-50) as more shallow phases are activated. This is expected, given the design space exploration performed to

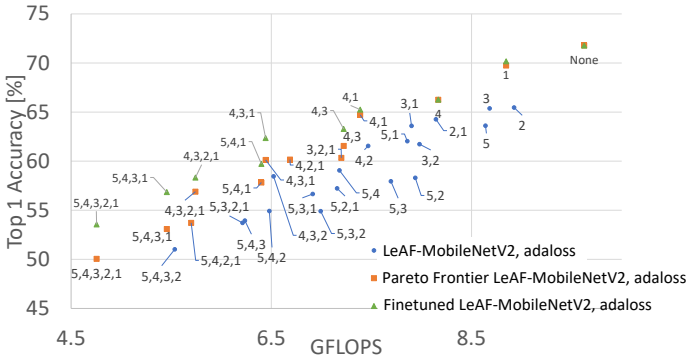

Fig. 6. Accuracy vs. FLOPs for all the compute configurations of LeAF-MobileNetV2, after base-training and after fine-tuning, for batch size 32.

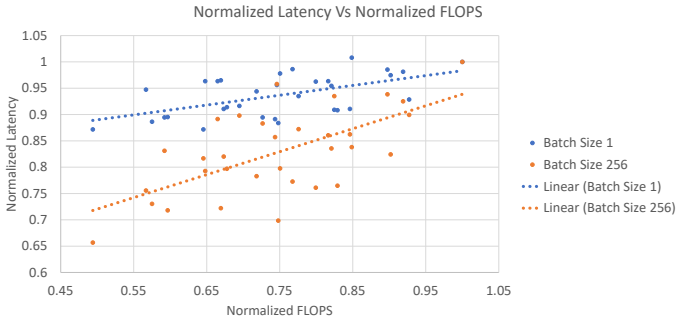

Fig. 7. MobileNetV2 Latency vs. FLOPs for batch sizes 1 and 256. The relationship between FLOPs and latency is batch size dependent.

generate such networks. Nonetheless, LeAF continues to generate a single model that can be tuned in real time to satisfy the system constraints.

5 Discussion and conclusion

In LeAF, we disentangle the selection and the execution of the compute path in flexible DNNs, to increase their versatility. This turns out to be a critical feature for the deployment of many real-time systems, where the currently available system resources determine the operating point to use. For example, a mostly idle system could use a highly accurate but costly DNN configuration, and switch to a lower performing, less costly one in case of high load scenario. LeAF provides flexible networks by augmenting a legacy network with lower system resource paths. This technique allows us to convert any network into a flexible one while

preserving the ability to run the original network at full accuracy. Our results show that this technique produces models that are competitive with other state of the art techniques for dynamic networks that, however, do not have this same capability. We have shown the performance and capability of our technique while varying the system compute capability, batch size and network models.

While our experiments were performed using LeAF networks, we believe the results are applicable to other dynamic and flexible models. We have demonstrated the need to directly use the target system metric in our comparison of FLOP count and latency performance on the actual systems. We have shown that as the compute capability of the system is varied, the target configurations for running with a constrained resources will probably vary as well. Our analysis shows the importance of the batch size on the performance of DNNs with variable architecture - something that so far has not been highlighted enough. The fact that beyond a target device one should also specify a target batch size adds an additional axis to the problem of creating a flexible DNN. Optimizing performance in real-time systems with time varying constraints is a complex problem. LeAF helps to provide a way to maximize model performance as system resource constraints are varied during run-time.

References

1. Alvarez, J.M., Salzmann, M.: Learning the number of neurons in deep networks. NeurIPS (2016)
2. Cai, H., Gan, C., Wang, T., Zhang, Z., Han, S.: Once for all: Train one network and specialize it for efficient deployment. In: ICLR (2020)
3. Cai, Y., Yao, Z., Dong, Z., Gholami, A., Mahoney, M.W., Keutzer, K.: Zeroq: A novel zero shot quantization framework. In: CVPR (2020)
4. Dai, X., Zhang, P., Wu, B., Yin, H., Sun, F., Wang, Y., Dukhan, M., Hu, Y., Wu, Y., Jia, Y., et al.: Chamnet: Towards efficient network design through platform-aware model adaptation. In: CVPR (2019)
5. Deng, J., Dong, W., Socher, R., Li, L.J., Li, K., Fei-Fei, L.: Imagenet: A large-scale hierarchical image database. In: 2009 IEEE conference on computer vision and pattern recognition. pp. 248–255. Ieee (2009)
6. Ding, X., Zhang, X., Ma, N., Han, J., Ding, G., Sun, J.: Repvgg: Making vgg-style convnets great again. In: CVPR (2021)
7. Han, Y., Huang, G., Song, S., Yang, L., Wang, H., Wang, Y.: Dynamic neural networks: A survey. IEEE Trans. PAMI (2021)
8. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: CVPR (2016)
9. He, K., Zhang, X., Ren, S., Sun, J.: Identity mappings in deep residual networks. In: Leibe, B., Matas, J., Sebe, N., Welling, M. (eds.) ECCV (2016)
10. He, Y., Zhang, X., Sun, J.: Channel pruning for accelerating very deep neural networks. In: ICCV (2017)
11. Hu, H., Dey, D., Hebert, M., Bagnell, J.: Learning anytime predictions in neural networks via adaptive loss balancing. AAAI (2019)
12. Huang, G., Chen, D., Li, T., Wu, F., van der Maaten, L., Weinberger, K.: Multi-scale dense networks for resource efficient image classification. In: ICLR (2018)

13. Huang, G., Sun, Y., Liu, Z., Sedra, D., Weinberger, K.Q.: Deep networks with stochastic depth. In: Leibe, B., Matas, J., Sebe, N., Welling, M. (eds.) ECCV (2016)
14. Jastrzebski, S., Arpit, D., Ballas, N., Verma, V., Che, T., Bengio, Y.: Residual connections encourage iterative inference. In: ICLR (2018)
15. Lee, N., Ajanthan, T., Torr, P.H.: Snip: Single-shot network pruning based on connection sensitivity. CoRR **abs/1810.02340** (2018)
16. Li, B., Wu, B., Su, J., Wang, G.: Eagleeye: Fast sub-net evaluation for efficient neural network pruning. In: ECCV (2020)
17. Molchanov, P., Mallya, A., Tyree, S., Frosio, I., Kautz, J.: Importance estimation for neural network pruning. In: CVPR (2019)
18. NVIDIA: CUDA Toolkit Documentation. `https://docs.nvidia.com/cuda/profiler-users-guide/index.html`, accessed: 2021-10-30
19. Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., Chintala, S.: PyTorch: An Imperative Style, High-Performance Deep Learning Library. In: NEURIPS (2019)
20. Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., Chen, L.C.: Mobilenetv2: Inverted residuals and linear bottlenecks. In: CVPR (2018)
21. Shazeer, N., Mirhoseini, A., Maziarz, K., Davis, A., Le, Q.V., Hinton, G.E., Dean, J.: Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. In: ICLR (2017)
22. Shen, M., Yin, H., Molchanov, P., Mao, L., Liu, J., Alvarez, J.M.: Halp: Hardware-aware latency pruning. CoRR **abs/2110.10811** (2021)
23. Shi, W., Caballero, J., Huszar, F., Totz, J., Aitken, A.P., Bishop, R., Rueckert, D., Wang, Z.: Real-time single image and video super-resolution using an efficient sub-pixel convolutional neural network. In: CVPR (2016)
24. Teerapittayanon, S., McDanel, B., Kung, H.: Branchynet: Fast inference via early exiting from deep neural networks. In: ICPR (2016)
25. Veit, A., Belongie, S.: Convolutional networks with adaptive inference graphs. International Journal of Computer Vision (2020)
26. Wang, W., Chen, M., Zhao, S., Chen, L., Hu, J., Liu, H., Cai, D., He, X., Liu, W.: Accelerate cnns from three dimensions: A comprehensive pruning framework. In: ICML (2021)
27. Wang, X., Yu, F., Dou, Z.Y., Darrell, T., Gonzalez, J.E.: Skipnet: Learning dynamic routing in convolutional networks. In: ECCV (2018)
28. Wang, Y., Shen, J., Hu, T.K., Xu, P., Nguyen, T., Baraniuk, R., Wang, Z., Lin, Y.: Dual dynamic inference: Enabling more efficient, adaptive, and controllable deep inference. IEEE Journal of Selected Topics in Signal Processing (2020)
29. Yang, T.J., Howard, A., Chen, B., Zhang, X., Go, A., Sandler, M., Sze, V., Adam, H.: Netadapt: Platform-aware neural network adaptation for mobile applications. In: ECCV (2018)
30. Yu, J., Huang, T.S.: Network slimming by slimmable networks: Towards one-shot architecture search for channel numbers. CoRR **abs/1903.11728** (2019), `http://arxiv.org/abs/1903.11728`
31. Yu, J., Yang, L., Xu, N., Yang, J., Huang, T.: Slimmable neural networks. In: ICLR (2019)
32. Zhou, W., Xu, C., Ge, T., McAuley, J.J., Xu, K., Wei, F.: Bert loses patience: Fast and robust inference with early exit. In: NeurIPS (2020)

33. Zhu, C., Han, S., Mao, H., Dally, W.J.: Trained ternary quantization. In: ICLR (2017)