NetAdapt: Platform-Aware Neural Network Adaptation for Mobile Applications

Tien-Ju Yang[1][0000−0003−4728−0321], Andrew Howard[2], Bo Chen[2],
Xiao Zhang[2], Alec Go[2], Mark Sandler[2], Vivienne Sze[1], and Hartwig Adam[2]

[1] Massachusetts Institute of Technology
[2] Google Inc.
{tjy,sze}@mit.edu, {howarda,bochen,andypassion,ago,sandler,hadam}@google.com

Abstract. This work proposes an algorithm, called NetAdapt, that *automatically adapts* a pre-trained deep neural network to a mobile platform given a resource budget. While many existing algorithms simplify networks based on the number of MACs or weights, optimizing those indirect metrics may not necessarily reduce the direct metrics, such as latency and energy consumption. To solve this problem, NetAdapt incorporates direct metrics into its adaptation algorithm. These direct metrics are evaluated using *empirical measurements*, so that detailed knowledge of the platform and toolchain is not required. NetAdapt automatically and progressively simplifies a pre-trained network until the resource budget is met while maximizing the accuracy. Experiment results show that NetAdapt achieves better accuracy versus latency trade-offs on both mobile CPU and mobile GPU, compared with the state-of-the-art automated network simplification algorithms. For image classification on the ImageNet dataset, NetAdapt achieves up to a 1.7× speedup in *measured inference latency* with equal or higher accuracy on MobileNets (V1&V2).

1 Introduction

Deep neural networks (DNNs or networks) have become an indispensable component of artificial intelligence, delivering near or super-human accuracy on common vision tasks such as image classification and object detection. However, DNN-based AI applications are typically too computationally intensive to be deployed on resource-constrained platforms, such as mobile phones. This hinders the enrichment of a large set of user experiences.

A significant amount of recent work on DNN design has focused on improving the efficiency of networks. However, the majority of works are based on optimizing the "indirect metrics", such as the number of multiply-accumulate operations (MACs) or the number of weights, as proxies for the resource consumption of a network. Although these indirect metrics are convenient to compute and integrate into the optimization framework, they may not be good approximations to the "direct metrics" that matter for the real applications such as latency

---

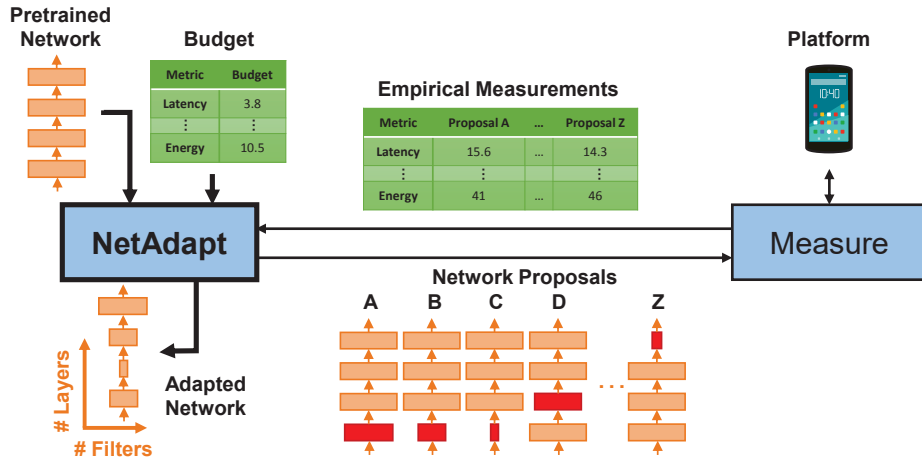⋆ This work was done while Tien-Ju Yang was an intern at Google.

Fig. 1. NetAdapt automatically adapts a pretrained network to a mobile platform given a resource budget. This algorithm is guided by the direct metrics for resource consumption. NetAdapt eliminates the requirement of platform-specific knowledge by using empirical measurements to evaluate the direct metrics. At each iteration, NetAdapt generates many network proposals and measures the proposals on the target platform. The measurements are used to guide NetAdapt to generate the next set of network proposals at the next iteration.

and energy consumption. The relationship between an indirect metric and the corresponding direct metric can be highly non-linear and platform-dependent as observed by [15, 25, 26]. In this work, we will also demonstrate empirically that a network with a fewer number of MACs can be slower when actually running on mobile devices; specifically, we will show that a network of 19% less MACs incurs 29% longer latency in practice (see Table 1).

There are two common approaches to designing efficient network architectures. The first is designing a single architecture with no regard to the underlying platform. It is hard for a single architecture to run optimally on all the platforms due to the different platform characteristics. For example, the fastest architecture on a desktop GPU may not be the fastest one on a mobile CPU with the same accuracy. Moreover, there is little guarantee that the architecture could meet the resource budget (e.g., latency) on all platforms of interest. The second approach is manually crafting architectures for a given target platform based on the platform's characteristics. However, this approach requires deep knowledge about the implementation details of the platform, including the toolchains, the configuration and the hardware architecture, which are generally unavailable given the proprietary nature of hardware and the high complexity of modern systems. Furthermore, manually designing a different architecture for each platform can be taxing for researchers and engineers.

In this work, we propose a platform-aware algorithm, called *NetAdapt*, to address the aforementioned issues and facilitate platform-specific DNN deploy-

ment. NetAdapt (Fig. 1) incorporates *direct metrics* in the optimization loop, so it does not suffer from the discrepancy between the indirect and direct metrics. The direct metrics are evaluated by the empirical measurements taken from the target platform. This enables the algorithm to support any platform without detailed knowledge of the platform itself, although such knowledge could still be incorporated into the algorithm to further improve results. In this paper, we use latency as the running example of a direct metric and resource to target even though our algorithm is generalizable to other metrics or a combination of them (Sec. 4.3).

The network optimization of NetAdapt is carried out in an automatic way to gradually reduce the resource consumption of a pretrained network while maximizing the accuracy. The optimization runs iteratively until the resource budget is met. Through this design, NetAdapt can generate not only a network that meets the budget, but also a family of simplified networks with different trade-offs, which allows dynamic network selection and further study. Finally, instead of being a black box, NetAdapt is designed to be easy to interpret. For example, through studying the proposed network architectures and the corresponding empirical measurements, we can understand why a proposal is chosen and this sheds light on how to improve the platform and network design.

The main contributions of this paper are:

- A framework that uses direct metrics when optimizing a pretrained network to meet a given resource budget. Empirical measurements are used to evaluate the direct metrics such that no platform-specific knowledge is required.
- An automated constrained network optimization algorithm that maximizes accuracy while satisfying the constraints (i.e., the predefined resource budget). The algorithm outperforms the state-of-the-art automatic network simplification algorithms by up to $1.7\times$ in terms of reduction in *measured inference latency* while delivering equal or higher accuracy. Moreover, a family of simplified networks with different trade-offs will be generated to allow dynamic network selection and further study.
- Experiments that demonstrate the effectiveness of NetAdapt on different platforms and on real-time-class networks, such as the small MobileNetV1, which is more difficult to simplify than larger networks.

2   Related Work

There is a large body of work that aims to simplify DNNs. We refer the readers to [21] for a comprehensive survey, and summarize the main approaches below.

The most related works are pruning-based methods. [6, 14, 16] aim to remove individual redundant weights from DNNs. However, most platforms cannot fully take advantage of unstructured sparse filters [26]. Hu et al. [10] and Srinivas et al. [20] focus on removing entire filters instead of individual weights. The drawback of these methods is the requirement of *manually* choosing the compression rate for each layer. MorphNet [5] leverages the sparsifying regularizers to automatically determine the layerwise compression rate. ADC [8] uses reinforcement

learning to learn a policy for choosing the compression rates. The crucial difference between all the aforementioned methods and ours is that they are not guided by the direct metrics, and thus may lead to sub-optimal performance, as we see in Sec. 4.3.

Energy-aware pruning [25] uses an energy model [24] and incorporates the estimated energy numbers into the pruning algorithm. However, this requires designing models to estimate the direct metrics of each target platform, which requires detailed knowledge of the platform including its hardware architecture [3], and the network-to-array mapping used in the toolchain [2]. NetAdapt does not have this requirement since it can directly use empirical measurements.

DNNs can also be simplified by approaches that involve directly designing efficient network architectures, decomposition or quantization. MobileNets [9, 18] and ShuffleNets [27] provide efficient layer operations and reference architecture design. Layer-decomposition-based algorithms [13, 23] exploit matrix decomposition to reduce the number of operations. Quantization [11, 12, 17] reduces the complexity by decreasing the computation accuracy. The proposed algorithm, NetAdapt, is complementary to these methods. For example, NetAdapt can adapt MobileNets to further push the frontier of efficient networks as shown in Sec. 4 even though MobileNets are more compact and much harder to simplify than the other larger networks, such as VGG [19].

3   Methodology: NetAdapt

We propose an algorithm, called NetAdapt, that will allow a user to automatically simplify a pretrained network to meet the resource budget of a platform while maximizing the accuracy. NetAdapt is guided by direct metrics for resource consumption, and the direct metrics are evaluated by using empirical measurements, thus eliminating the requirement of detailed platform-specific knowledge.

3.1   Problem Formulation

NetAdapt aims to solve the following non-convex constrained problem:

$$\begin{aligned} &\underset{Net}{\text{maximize}} \quad Acc(Net) \\ &\text{subject to} \quad Res_j(Net) \leq Bud_j, \ j = 1, \ldots, m, \end{aligned} \tag{1}$$

where Net is a simplified network from the initial pretrained network, $Acc(\cdot)$ computes the accuracy, $Res_j(\cdot)$ evaluates the direct metric for resource consumption of the jth resource, and Bud_j is the budget of the jth resource and the constraint on the optimization. The resource can be latency, energy, memory footprint, etc., or a combination of these metrics.

Based on an idea similar to progressive barrier methods [1], NetAdapt breaks this problem into the following series of easier problems and solves it iteratively:

$$\begin{aligned} &\underset{Net_i}{\text{maximize}} \quad Acc(Net_i) \\ &\text{subject to} \quad Res_j(Net_i) \leq Res_j(Net_{i-1}) - \Delta R_{i,j}, \ j = 1, \ldots, m, \end{aligned} \tag{2}$$

Algorithm 1: NetAdapt

Input: Pretrained Network: Net_0 (with K CONV and FC layers), Resource
Budget: Bud, Resource Reduction Schedule: ΔR_i
Output: Adapted Network Meeting the Resource Budget: $\hat{Net}$

1 i = 0;
2 Res_i = TakeEmpiricalMeasurement(Net_i);
3 **while** $Res_i > Bud$ **do**
4   Con = Res_i - ΔR_i;
5   **for** k *from 1 to K* **do**
          /* TakeEmpiricalMeasurement is also called inside
             ChooseNumFilters for choosing the correct number of filters
             that satisfies the constraint (i.e., current budget). */
6 N_Filt_k, Res_Simp_k = ChooseNumFilters(Net_i, k, Con);
7 Net_Simp_k = ChooseWhichFilters(Net_i, k, N_Filt_k);
8 Net_Simp_k = ShortTermFineTune(Net_Simp_k);
9 Net_{i+1}, Res_{i+1} = PickHighestAccuracy($Net\_Simp_.$, $Res\_Simp_.$);
10   i = i + 1;
11 $\hat{Net}$ = LongTermFineTune(Net_i);
12 return $\hat{Net}$;

---

where Net_i is the network generated by the ith iteration, and Net_0 is the initial
pretrained network. As the number of iterations increases, the constraints (i.e.,
current resource budget $Res_j(Net_{i-1}) - \Delta R_{i,j}$) gradually become tighter. $\Delta R_{i,j}$,
which is larger than zero, indicates how much the constraint tightens for the jth
resource in the ith iteration and can vary from iteration to iteration. This is
referred to as "resource reduction schedule", which is similar to the concept of
learning rate schedule. The algorithm terminates when $Res_j(Net_{i-1}) - \Delta R_{i,j}$
is equal to or smaller than Bud_j for every resource type. It outputs the final
adapted network and can also generate a sequence of simplified networks (i.e.,
the highest accuracy network from each iteration $Net_1, ..., Net_i$) to provide the
efficient frontier of accuracy and resource consumption trade-offs.

3.2   Algorithm Overview

For simplicity, we assume that we only need to meet the budget of one resource,
specifically latency. One method to reduce the latency is to remove filters from
the convolutional (CONV) or fully-connected (FC) layers. While there are other
ways to reduce latency, we will use this approach to demonstrate NetAdapt.

The NetAdapt algorithm is detailed in pseudo code in Algorithm 1 and in
Fig. 2. Each iteration solves Eq. 2 by reducing the number of filters in a *single*
CONV or FC layer (the **Choose # of Filters** and **Choose Which Filters**
blocks in Fig. 2). The number of filters to remove from a layer is guided by
empirical measurements. NetAdapt removes entire filters instead of individual
weights because most platforms can take advantage of removing entire filters,
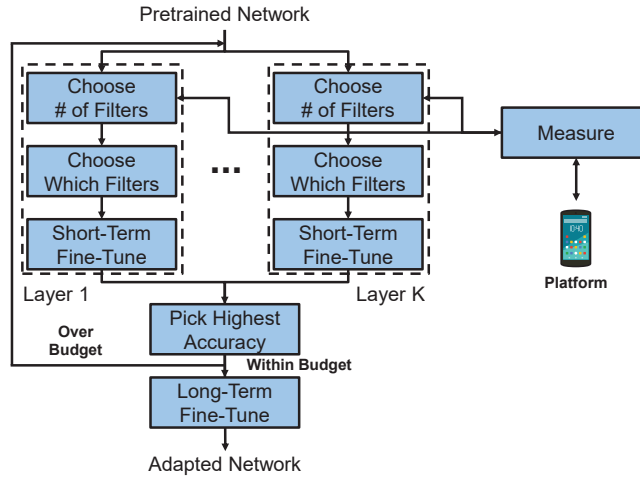
Fig. 2. This figure visualizes the algorithm flow of NetAdapt. At each iteration, NetAdapt decreases the resource consumption by simplifying (i.e., removing filters from) one layer. In order to maximize accuracy, it tries to simplify each layer individually and picks the simplified network that has the highest accuracy. Once the target budget is met, the chosen network is then fine-tuned again until convergence.

and this strategy allows reducing both filters and feature maps, which play an important role in resource consumption [25]. The simplified network is then fine-tuned for a short length of time in order to restore some accuracy (the **Short-Term Fine-Tune** block).

In each iteration, the previous three steps (highlighted in bold) are applied on each of the CONV or FC layers individually[3]. As a result, NetAdapt generates K (i.e., the number of CONV and FC layers) network proposals in one iteration, each of which has a single layer modified from the previous iteration. The network proposal with the highest accuracy is carried over to the next iteration (the **Pick Highest Accuracy** block). Finally, once the target budget is met, the chosen network is fine-tuned again until convergence (the **Long-Term Fine-Tune** block).

3.3   Algorithm Details

This section describes the key blocks in the *NetAdapt* algorithm (Fig. 2).

Choose Number of Filters This step focuses on determining *how many* filters to preserve in a specific layer based on empirical measurements. NetAdapt gradually reduces the number of filters in the target layer and measures the resource consumption of each of the simplified networks. The maximum number

---

[3] The algorithm can also be applied to a group of multiple layers as a single unit (instead of a single layer). For example, in ResNet [7], we can treat a residual block as a single unit to speed up the adaptation process.
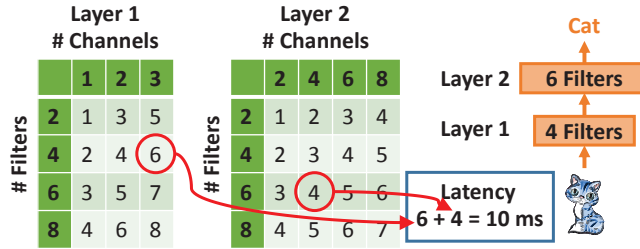
Fig. 3. This figure illustrates how layer-wise look-up tables are used for fast resource consumption estimation.

of filters that can satisfy the current resource constraint will be chosen. Note that when some filters are removed from a layer, the associated channels in the following layers should also be removed. Therefore, the change in the resource consumption of other layers needs to be factored in.

Choose Which Filters This step chooses *which* filters to preserve based on the architecture from the previous step. There are many methods proposed in the literature, and we choose the magnitude-based method to keep the algorithm simple. In this work, the N filters that have the largest $\ell 2$-norm magnitude will be kept, where N is the number of filters determined by the previous step. More complex methods can be adopted to increase the accuracy, such as removing the filters based on their joint influence on the feature maps [25].

Short-/Long-Term Fine-Tune Both the short-term fine-tune and long-term fine-tune steps in NetAdapt involve network-wise end-to-end fine-tuning. Short-term fine-tune has fewer iterations than long-term fine-tune.

At each iteration of the algorithm, we fine-tune the simplified networks with a relatively smaller number of iterations (i.e., short-term) to regain accuracy, in parallel or in sequence. This step is especially important while adapting small networks with a large resource reduction because otherwise the accuracy will drop to zero, which can cause the algorithm to choose the wrong network proposal.

As the algorithm proceeds, the network is continuously trained but does not converge. Once the final adapted network is obtained, we fine-tune the network with more iterations until convergence (i.e., long-term) as the final step.

3.4   Fast Resource Consumption Estimation

As mentioned in Sec. 3.3, NetAdapt uses empirical measurements to determine the number of filters to keep in a layer given the resource constraint. In theory, we can measure the resource consumption of each of the simplified networks on the fly during adaptation. However, taking measurements can be slow and difficult to parallelize due to the limited number of available devices. Therefore, it may be prohibitively expensive and become the computation bottleneck.
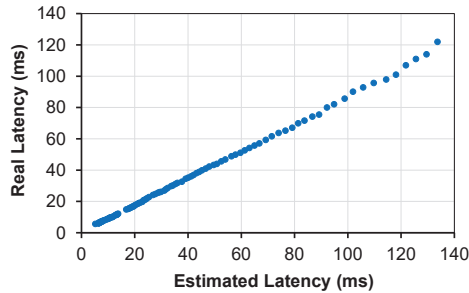
Fig. 4. The comparison between the estimated latency (using layer-wise look-up tables) and the real latency on a single large core of Google Pixel 1 CPU while adapting the 100% MobileNetV1 with the input resolution of 224 [9].

We solve this problem by building layer-wise look-up tables with pre-measured resource consumption of each layer. When executing the algorithm, we look up the table of each layer, and sum up the layer-wise measurements to estimate the network-wise resource consumption, which is illustrated in Fig. 3. The reason for not using a network-wise table is that the size of the table will grow exponentially with the number of layers, which makes it intractable for deep networks. Moreover, layers with the same shape and feature map size only need to be measured once, which is common for modern deep networks.

Fig. 4 compares the estimated latency (the sum of layer-wise latency from the layer-wise look-up tables) and the real latency on a single large core of Google Pixel 1 CPU while adapting the 100% MobileNetV1 with the input resolution of 224 [9]. The real and estimated latency numbers are highly correlated, and the difference between them is sufficiently small to be used by NetAdapt.

4   Experiment Results

In this section, we apply the proposed NetAdapt algorithm to MobileNets [9, 18], which are designed for mobile applications, and experiment on the ImageNet dataset [4]. We did not apply NetAdapt on larger networks like ResNet [7] and VGG [19] because networks become more difficult to simplify as they become smaller; these networks are also seldom deployed on mobile platforms. We benchmark NetAdapt against three state-of-the-art network simplification methods:

– **Multipliers** [9] are simple but effective methods for simplifying networks. Two commonly used multipliers are the width multiplier and the resolution multiplier; they can also be used together. Width multiplier scales the number of filters by a percentage across all convolutional (CONV) and fully-connected (FC) layers, and resolution multiplier scales the resolution of the input image. We use the notation "50% MobileNetV1 (128)" to denote applying a width multiplier of 50% on MobileNetV1 with the input image resolution of 128.

- **MorphNet** [5] is an automatic network simplification algorithm based on sparsifying regularization.
- **ADC** [8] is an automatic network simplification algorithm based on reinforcement learning.

We will show the performance of NetAdapt on the small MobileNetV1 (50% MobileNetV1 (128)) to demonstrate the effectiveness of NetAdapt on real-time-class networks, which are much more difficult to simplify than larger networks. To show the generality of NetAdapt, we will also measure its performance on the large MobileNetV1 (100% MobileNetV1 (224)) across different platforms. Lastly, we adapt the large MobileNetV2 (100% MobileNetV2 (224)) to push the frontier of efficient networks.

4.1   Detailed Settings for MobileNetV1 Experiments

We perform most of the experiments and study on MobileNetV1 and detail the settings in this section.

NetAdapt Configuration MobileNetV1 [9] is based on depthwise separable convolutions, which factorize a $m \times m$ standard convolution layer into a $m \times m$ depthwise layer and a 1×1 standard convolution layer called a pointwise layer. In the experiments, we adapt each depthwise layer with the corresponding pointwise layer and choose the filters to keep based on the pointwise layer. When adapting the small MobileNetV1 (50% MobileNetV1 (128)), the latency reduction ($\Delta R_{i,j}$ in Eq. 2) starts at 0.5 and decays at the rate of 0.96 per iteration. When adapting other networks, we use the same decay rate but scale the initial latency reduction proportional to the latency of the initial pretrained network.

Network Training We preserve ten thousand images from the training set, ten images per class, as the holdout set. The new training set without the holdout images is used to perform short-term fine-tuning, and the holdout set is used to pick the highest accuracy network out of the simplified networks at each iteration. The whole training set is used for the long-term fine-tuning, which is performed once in the last step of NetAdapt.

Because the training configuration can have a large impact on the accuracy, we apply the same training configuration to all the networks unless otherwise stated to have a fairer comparison. We adopt the same training configuration as MorphNet [5] (except that the batch size is 128 instead of 96). The learning rate for the long-term fine-tuning is 0.045 and that for the short-term fine-tuning is 0.0045. This configuration improves ADC network's top-1 accuracy by 0.3% and almost all multiplier networks' top-1 accuracy by up to 3.8%, except for one data point, whose accuracy is reduced by 0.2%. We use these numbers in the following analysis. Moreover, all accuracy numbers are reported on the validation set to show the true performance.

Mobile Inference and Latency Measurement We use Google's Tensor-Flow Lite engine [22] for inference on a mobile CPU and Qualcomm's Snapdragon Neural Processing Engine (SNPE) for inference on a mobile GPU. For experiments on mobile CPUs, the latency is measured on a single large core of
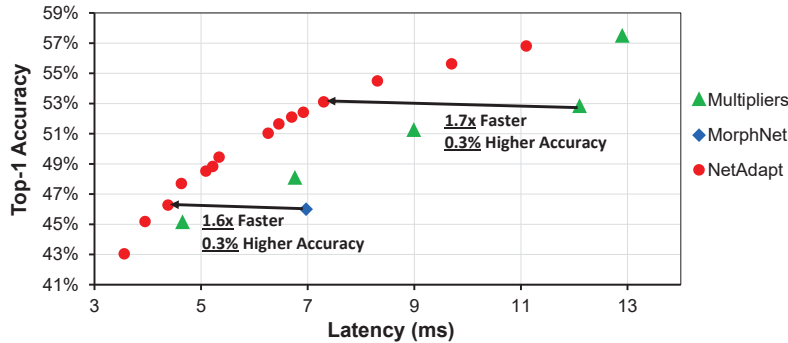
Fig. 5. The figure compares NetAdapt (adapting the small MobileNetV1) with the multipliers [9] and MorphNet [5] on a mobile CPU of Google Pixel 1.

Google Pixel 1 phone. For experiments on mobile GPUs, the latency is measured on the mobile GPU of Samsung Galaxy S8 with SNPE's benchmarking tool. For each latency number, we report the median of 11 latency measurements.

4.2   Comparison with Benchmark Algorithms

Adapting Small MobileNetV1 on a Mobile CPU In this experiment, we apply NetAdapt to adapt the small MobileNetV1 (50% MobileNetV1 (128)) to a mobile CPU. It is one of the most compact networks and achieves real-time performance. It is more challenging to simplify than other larger networks (include the large MobileNetV1). The results are summarized and compared with the multipliers [9] and MorphNet [5] in Fig. 5. We observe that NetAdapt outperforms the multipliers by up to 1.7× faster with the same or higher accuracy. For MorphNet, NetAdapt's result is 1.6× faster with 0.3% higher accuracy.

Adapting Large MobileNetV1 on a Mobile CPU In this experiment, we apply NetAdapt to adapt the large MobileNetV1 (100% MobileNetV1 (224)) on a mobile CPU. It is the largest MobileNetV1 and achieves the highest accuracy. Because its latency is approximately 8× higher than that of the small MobileNetV1, we scale the initial latency reduction by 8×. The results are shown and compared with the multipliers [9] and ADC [8] in Fig. 6. NetAdapt achieves higher accuracy than the multipliers and ADC while increasing the speed by 1.4× and 1.2×, respectively.

While the training configuration is kept the same when comparing to the benchmark algorithms discussed above, we also show in Fig. 6 that the accuracy of the networks adapted using NetAdapt can be further improved with a better training configuration. After simply adding dropout and label smoothing, the accuracy can be increased by 1.3%. Further tuning the training configuration for each adapted network can give higher accuracy numbers, but it is not the focus of this paper.
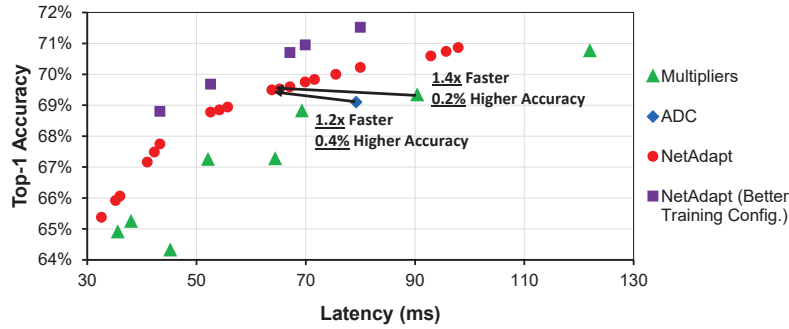
Fig. 6. The figure compares NetAdapt (adapting the large MobileNetV1) with the multipliers [9] and ADC [8] on a mobile CPU of Google Pixel 1. Moreover, the accuracy of the adapted networks can be further increased by up to 1.3% through using a better training configuration (simply adding dropout and label smoothing).
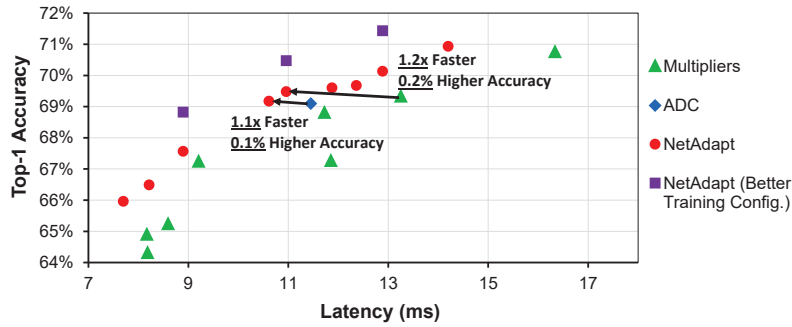

Fig. 7. This figure compares NetAdapt (adapting the large MobileNetV1) with the multipliers [9] and ADC [8] on a mobile GPU of Samsung Galaxy S8. Moreover, the accuracy of the adapted networks can be further increased by up to 1.3% through using a better training configuration (simply adding dropout and label smoothing).

Adapting Large MobileNetV1 on a Mobile GPU In this experiment, we apply NetAdapt to adapt the large MobileNetV1 on a mobile GPU to show the generality of NetAdapt. Fig. 7 shows that NetAdapt outperforms other benchmark algorithms by up to 1.2× speed-up with higher accuracy. Due to the limitation of the SNPE tool, the layerwise latency breakdown only considers the computation time and does not include the latency of other operations, such as feature map movement, which can be expensive [25]. This affects the precision of the look-up tables used for this experiment. Moreover, we observe that there is an approximate 6.2ms (38% of the latency of the network before applying NetAdapt) non-reducible latency. These factors cause a smaller improvement on the mobile GPU compared with the experiments on the mobile CPU. Moreover, when the better training configuration is applied as previously described, the accuracy can be further increased by 1.3%.

Network	Top-1 Accuracy (%)		# of MACs ($\times 10^6$)		Latency (ms)	
25% MobileNetV1 (128) [9]	45.1	(+0)	13.6	(100%)	4.65	(100%)
MorphNet [5]	46.0	(+0.9)	15.0	(110%)	6.52	(140%)
NetAdapt	46.3	(+1.2)	11.0	(81%)	6.01	(129%)
75% MobileNetV1 (224) [9]	68.8	(+0)	325.4	(100%)	69.3	(100%)
ADC [8]	69.1	(+0.3)	304.2	(93%)	79.2	(114%)
NetAdapt	69.1	(+0.3)	284.3	(87%)	74.9	(108%)

Table 1. The comparison between NetAdapt (adapting the small or large Mo-bileNetV1) and the three benchmark algorithms on image classification when targeting the number of MACs. The latency numbers are measured on a mobile CPU of Google Pixel 1. We roughly match their accuracy and compare their latency.
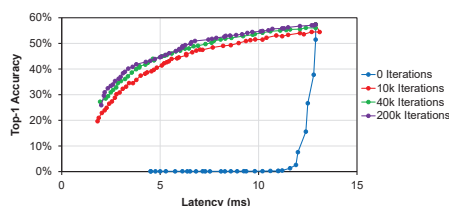

Fig. 8. The accuracy of different short-term fine-tuning iterations when adapting the small MobileNetV1 (without long-term fine-tuning) on a mobile CPU of Google Pixel 1. Zero iterations means no short-term fine-tuning.
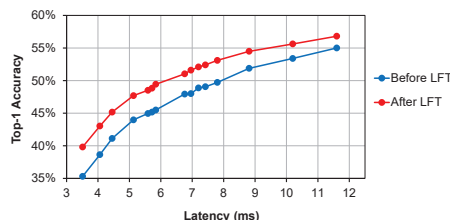
Fig. 9. The comparison between before and after long-term fine-tuning when adapting the small MobileNetV1 on a mobile CPU of Google Pixel 1. Although the short-term fine-tuning preserves the accuracy well, the long-term fine-tuning gives the extra 3.4% on average (from 1.8% to 4.5%).

4.3   Ablation Studies

Impact of Direct Metrics In this experiment, we use the indirect metric (i.e., the number of MACs) instead of the direct metric (i.e., the latency) to guide NetAdapt to investigate the importance of using direct metrics. When computing the number of MACs, we only consider the CONV and FC layers because batch normalization layers can be folded into the corresponding CONV layers, and the other layers are negligibly small. Table 1 shows that NetAdapt outperforms the benchmark algorithms with lower numbers of MACs and higher accuracy. This demonstrates the effectiveness of NetAdapt. However, we also observe that the network with lower numbers of MACs may not necessarily be faster. This shows the necessity of incorporating direct measurements into the optimization flow.

Impact of Short-Term Fine-Tuning Fig. 8 shows the accuracy of adapting the small MobileNetV1 with different short-term fine-tuning iterations (without long-term fine-tuning). The accuracy rapidly drops to nearly zero if no short-term fine-tuning is performed (i.e., zero iterations). In this low accuracy region, the algorithm picks the best network proposal solely based on noise and hence

Initialization (ms)	Decay Rate	# of Total Iterations	Top-1 Accuracy (%)	Latency (ms)
0.5	0.96	28	47.7	4.63
0.5	1.0	20	47.4	4.71
0.8	0.95	20	46.7	4.65

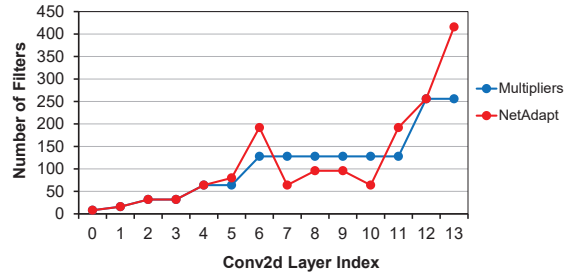**Table 2.** The influence of resource reduction schedules.

Fig. 10. NetAdapt and the multipliers generate different simplified networks when adapting the small MobileNetV1 to match the latency of 25% MobileNetV1 (128).

gives poor performance. After fine-tuning a network for a short amount of time (ten thousand iterations), the accuracy is always kept above 20%, which allows the algorithm to make a better decision. Although further increasing the number of iterations improves the accuracy, we find that using forty thousand iterations leads to a good accuracy versus speed trade-off for the small MobileNetV1.

Impact of Long-Term Fine-Tuning Fig. 9 illustrates the importance of performing the long-term fine-tuning. Although the short-term fine-tuning preserves the accuracy well, the long-term fine-tuning can still increase the accuracy by up to another 4.5% or 3.4% on average. Since the short-term fine-tuning has a short training time, the training is terminated far before convergence. Therefore, it is not surprising that the final long-term fine-tuning can further increase the accuracy.

Impact of Resource Reduction Schedules Table 2 shows the impact of using three different resource reduction schedules, which are defined in Sec. 3.1. Empirically, using a larger resource reduction at each iteration increases the adaptation speed (i.e., reducing the total number of adaptation iterations) at the cost of accuracy. With the same number of total iterations, the result suggests that a smaller initial resource reduction with a slower decay is preferable.

4.4   Analysis of Adapted Network Architecture

The network architectures of the adapted small MobileNetV1 by using NetAdapt and the multipliers are shown and compared in Fig. 10. Both of them have similar latency as 25% MobileNetV1 (128). There are two interesting observations.

Network	Top-1 Accuracy (%)		Latency (ms)	
75% MobileNetV2 (224) [18]	69.8	(+0)	61.4	(100%)
NetAdapt (Similar Latency)	70.9	(+1.1)	61.6	(100%)
NetAdapt (Similar Accuracy)	70.0	(+0.2)	53.5	(87%)

Table 3. The comparison between NetAdapt (adapting the large MobileNetV2 (100% MobileNetV2 (224))) and the multipliers [18] on a mobile CPU of Google Pixel 1. We compare the latency at similar accuracy and the accuracy at similar latency.

First, NetAdapt removes more filters in layers 7 to 10, but fewer in layer 6. Since the feature map resolution is reduced in layer 6 but not in layers 7 to 10, we hypothesize that when the feature map resolution is reduced, more filters are needed to avoid creating an information bottleneck.

The second observation is that NetAdapt keeps more filters in layer 13 (i.e. the last CONV layer). One possible explanation is that the ImageNet dataset contains one thousand classes, so more feature maps are needed by the last FC layer to do the correct classification.

4.5   Adapting Large MobileNetV2 on a Mobile CPU

In this section, we show encouraging early results of applying NetAdapt to MobileNetV2 [18]. MobileNetV2 introduces the inverted residual with linear bottleneck into MobileNetV1 and becomes more efficient. Because MobileNetV2 utilizes residual connections, we only adapt individual inner (expansion) layers or reduce all bottleneck layers of the same resolution in lockstep. The main differences between the MobileNetV1 and MobileNetV2 experiment settings are that each network proposal is short-term fine-tuned with ten thousand iterations, the initial latency reduction is 1ms, the latency reduction decay is 0.995, the batch size is 96, and dropout and label smoothing are used. NetAdapt achieves 1.1% higher accuracy or 1.2× faster speed than the multipliers as shown in Table 3.

5   Conclusion

In summary, we proposed an automated algorithm, called NetAdapt, to adapt a pretrained network to a mobile platform given a real resource budget. NetAdapt can incorporate direct metrics, such as latency and energy, into the optimization to maximize the adaptation performance based on the characteristics of the platform. By using empirical measurements, NetAdapt can be applied to any platform as long as we can measure the desired metrics, without any knowledge of the underlying implementation of the platform. We demonstrated empirically that the proposed algorithm can achieve better accuracy versus latency trade-off (by up to 1.7× faster with equal or higher accuracy) compared with other state-of-the-art network simplification algorithms. In this work, we aimed to highlight the importance of using direct metrics in the optimization of efficient networks; we hope that future research efforts will take direct metrics into account in order to further improve the performance of efficient networks.

Bibliography

[1] Audet, C., J. E. Dennis, J.: A progressive barrier for derivative-free nonlinear programming. SIAM Journal on Optimization **20**(1), 445–472 (2009)

[2] Chen, Y.H., Emer, J., Sze, V.: Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks. In: Proceedings of the 43rd Annual International Symposium on Computer Architecture (ISCA) (2016)

[3] Chen, Y.H., Krishna, T., Emer, J., Sze, V.: Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. IEEE Journal of Solid-State Circuits **52**, 127–138 (2016)

[4] Deng, J., Dong, W., Socher, R., Li, L.J., Li, K., Fei-Fei, L.: Imagenet: A large-scale hierarchical image database. In: IEEE Conference on Computer Vision and Pattern Recognition (CVPR). pp. 248–255. IEEE (2009)

[5] Gordon, A., Eban, E., Nachum, O., Chen, B., Yang, T.J., Choi, E.: Morphnet: Fast & simple resource-constrained structure learning of deep networks. In: IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (2018)

[6] Han, S., Pool, J., Tran, J., Dally, W.: Learning both weights and connections for efficient neural network. In: Advances in Neural Information Processing Systems. pp. 1135–1143 (2015)

[7] He, K., Zhang, X., Ren, S., Sun, J.: Deep Residual Learning for Image Recognition. In: IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (2016)

[8] He, Y., Han, S.: Adc: Automated deep compression and acceleration with reinforcement learning. arXiv preprint arXiv:1802.03494 (2018)

[9] Howard, A.G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., Adam, H.: Mobilenets: Efficient convolutional neural networks for mobile vision applications. arXiv preprint arXiv:1704.04861 (2017)

[10] Hu, H., Peng, R., Tai, Y.W., Tang, C.K.: Network Trimming: A Data-Driven Neuron Pruning Approach towards Efficient Deep Architectures. arXiv preprint arXiv:1607.03250 (2016)

[11] Hubara, I., Courbariaux, M., Soudry, D., El-Yaniv, R., Bengio, Y.: Binarized neural networks. In: Advances in Neural Information Processing Systems. pp. 4107–4115 (2016)

[12] Jacob, B., Kligys, S., Chen, B., Zhu, M., Tang, M., Howard, A., Adam, H., Kalenichenko, D.: Quantization and training of neural networks for efficient integer-arithmetic-only inference. arXiv preprint arXiv:1712.05877 (2017)

[13] Kim, Y.D., Park, E., Yoo, S., Choi, T., Yang, L., Shin, D.: Compression of deep convolutional neural networks for fast and low power mobile applications. arXiv preprint arXiv:1511.06530 (2015)

[14] Le Cun, Y., Denker, J.S., Solla, S.A.: Optimal brain damage. In: Advances in Neural Information Processing Systems (1990)

[15] Liangzhen Lai, Naveen Suda, V.C.: Not all ops are created equal! In: SysML (2018)

[16] Molchanov, P., Tyree, S., Karras, T., Aila, T., Kautz, J.: Pruning convolutional neural networks for resource efficient transfer learning. arXiv preprint arXiv:1611.06440 (2016)

[17] Rastegari, M., Ordonez, V., Redmon, J., Farhadi, A.: Xnor-net: Imagenet classification using binary convolutional neural networks. In: European Conference on Computer Vision (ECCV) (2016)

[18] Sandler, M., Howard, A.G., Zhu, M., Zhmoginov, A., Chen, L.C.: Inverted residuals and linear bottlenecks: Mobile networks for classification, detection and segmentation. In: IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (2018)

[19] Simonyan, K., Zisserman, A.: Very Deep Convolutional Networks for Large-Scale Image Recognition. In: International Conference on Learning Representations (ICLR) (2014)

[20] Srinivas, S., Babu, R.V.: Data-free parameter pruning for deep neural networks. arXiv preprint arXiv:1507.06149 (2015)

[21] Sze, V., Chen, Y.H., Yang, T.J., Emer, J.S.: Efficient processing of deep neural networks: A tutorial and survey. Proceedings of the IEEE **105**(12), 2295–2329 (Dec 2017). https://doi.org/10.1109/JPROC.2017.2761740

[22] TensorFlow Lite: https://www.tensorflow.org/mobile/tflite/

[23] Yang, Z., Moczulski, M., Denil, M., de Freitas, N., Smola, A., Song, L., Wang, Z.: Deep fried convnets. In: Proceedings of the IEEE International Conference on Computer Vision. pp. 1476–1483 (2015)

[24] Yang, Tien-Ju and Chen, Yu-Hsin and Emer, Joel and Sze, Vivienne: A Method to Estimate the Energy Consumption of Deep Neural Networks. In: Asilomar Conference on Signals, Systems and Computers (2017)

[25] Yang, Tien-Ju and Chen, Yu-Hsin and Sze, Vivienne: Designing energy-efficient convolutional neural networks using energy-aware pruning. In: IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (2017)

[26] Yu, J., Lukefahr, A., Palframan, D., Dasika, G., Das, R., Mahlke, S.: Scalpel: Customizing dnn pruning to the underlying hardware parallelism. In: Proceedings of the 44th Annual International Symposium on Computer Architecture (2017)

[27] Zhang, X., Zhou, X., Lin, M., Sun, J.: Shufflenet: An extremely efficient convolutional neural network for mobile devices. arXiv preprint arXiv:1707.01083 (2017)