AOWS: Adaptive and optimal network width search with latency constraints

Maxim Berman[*1]   Leonid Pishchulin[2]   Ning Xu[2]   Matthew B. Blaschko[1]   Gérard Medioni[2]

[1]Center for Processing Speech and Images, Department of Electrical Engineering, KU Leuven
[2]Amazon Go

Abstract

Neural architecture search (NAS) approaches aim at automatically finding novel CNN architectures that fit computational constraints while maintaining a good performance on the target platform. We introduce a novel efficient one-shot NAS approach to optimally search for channel numbers, given latency constraints on a specific hardware. We first show that we can use a black-box approach to estimate a realistic latency model for a specific inference platform, without the need for low-level access to the inference computation. Then, we design a pairwise MRF to score any channel configuration and use dynamic programming to efficiently decode the best performing configuration, yielding an optimal solution for the network width search. Finally, we propose an adaptive channel configuration sampling scheme to gradually specialize the training phase to the target computational constraints. Experiments on ImageNet classification show that our approach can find networks fitting the resource constraints on different target platforms while improving accuracy over the state-of-the-art efficient networks.

1. Introduction

Neural networks define the state of the art in computer vision for a wide variety of tasks. Increasingly sophisticated deep learning-based vision algorithms are being deployed on various target platforms, but they must be adapted to the platform-dependent latency/memory requirements and different hardware profiles. This motivates the need for task-aware neural architecture search (NAS) methods [1, 35, 25].

Multiple NAS approaches have been proposed in the literature and successfully applied to image recognition [3, 23, 22, 12, 28, 36] and language modeling tasks [35]. Despite their impressive performance, many of these approaches are prohibitively expensive, requiring the training of thousands of architectures in order to find a best performing model [35, 23, 36, 12, 19]. Some methods therefore try to
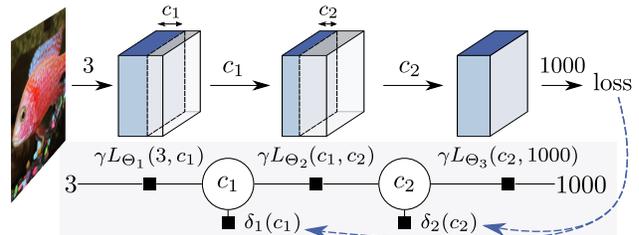

Figure 1: Overview of OWS applied to a 3-layer neural network. *Top:* slimmable network; *bottom:* MRF for optimal selection of channel numbers c_1 and c_2.

dramatically reduce compute overhead by summarizing the entire search space using a single over-parametrized neural network [22, 28]. AutoSlim [31] nests the entire search space (varying channel numbers) in a single *slimmable network* architecture [33, 32], trained to operate at different channel number configurations at test time.

In this work, we build on the concept of slimmable networks and propose a novel adaptive optimal width search (AOWS) for efficiently searching neural network channel configurations. We make several key contributions. First, we introduce a simple black-box latency modeling method that allows to estimate a realistic latency model for a specific hardware and inference modality, without the need for low-level access to the inference computation. Second, we design an optimal width search (OWS) strategy, using dynamic programming to efficiently decode the best performing channel configuration in a pairwise Markov random field (MRF). We empirically show that considering the entire channel configuration search space results into better NAS solutions compared to a greedy iterative trimming procedure [31]. Third, we propose an adaptive channel configuration sampling scheme. This approach gradually specializes the NAS proxy to our specific target at training-time, leading to an improved accuracy-latency trade-off in practice. Finally, we extensively evaluate AOWS on the ImageNet classification task for 3 target platforms and show significant accuracy improvements over state-of-the-art efficient networks.

Related work. The last years have seen a growing interest for automatic neural architecture search (NAS) methods [1, 35, 25]. Multiple NAS approaches have been proposed and successfully applied to image recognition [3, 23, 22, 12, 28, 36, 19] and language modeling tasks [35]. Pioneer approaches [35, 36] use reinforcement learning to search for novel architectures with lower FLOPs and improved accuracy. MNasNet [23] directly searches network architecture for mobile devices. They sample a few thousand models during architecture search, train each model for a few epochs only and evaluate on a large validation set to quickly estimate potential model accuracy. Many of these approaches require very heavy computations, and therefore resort to proxy tasks (e.g. small number of epochs, smaller datasets, reduced search space) before selecting the top-performing building blocks for further learning on large-scale target task [23, 19, 36]. To overcome these limitations, one group of methods directly learns the architectures for large-scale target tasks and target hardware platforms. For instance, [3] assumes a network structure composed of blocks (*e.g.* MNasNet [23]) and relies on a gradient-based approach, similar to DARTS [13], to search inside each block. Another group of methods intends to dramatically reduce compute overhead by summarizing the entire search space using a single over-parametrized neural network [22, 28, 33, 32]. Single-path NAS [22] uses this principle of nested models and combines search for channel numbers with a search over kernel sizes. However, single-path NAS restricts the search over channel numbers to 2 choices per layer and only optimizes over a subset of the channel numbers of the network, fixing the backbone channel numbers and optimizing only the expansion ratios of the residual branches in architectures such as Mobilenet-v2 [20].

AutoSlim [31] uses a *slimmable network* architecture [33, 32], which is trained to operate at different channel number configurations, as a model for the performance of a network trained to operate at a single channel configuration. Thus, the entire search space (varying channel numbers) is nested into one unique network. Once the slimmable network is trained, AutoSlim selects the final channel numbers with a greedy iterative trimming procedure, starting from the maximum-channel number configuration, until the resource constraints are met. Our approach is closely related to AutoSlim, as we also build on slimmable networks. In section 3, we further detail these prior works [32, 33, 31], highlighting their similarities and differences with our approach, which we introduce in sections 4 to 6.

2. Neural architecture search

We now briefly outline the NAS problem statement. A general NAS problem can be expressed as:

Problem 2.1 (NAS problem). Given a search space $\mathcal{S}$, a set of resource constraints $\mathcal{C}$, minimize $\Delta(N)$ for $N \in \mathcal{S} \cap \mathcal{C}$.

In a supervised learning setting, the error $\Delta(N)$ is typically defined as the error on the validation set after training network N on the training set. In the following, we discuss the choices of the search space $\mathcal{S}$ and of the constraint set $\mathcal{C}$.

Search space. The hardness of the NAS problem depends on the search space. A neural network in $\mathcal{S}$ can be represented by its computational graph, types of each node in the graph, and the parameters of each node. More specialized NAS approaches fix the neural network connectivity graph and operations but aim at finding the right parameters for these operations, e.g. kernel sizes, or number of input/output channels (*width*) per layer in the network. Single-path NAS [22] searches for kernel sizes and channel numbers, while AutoSlim [31] searches for channel numbers only. The restriction of the NAS problem to the search of channel numbers allows for a much more fine-grained search than in more general NAS methods. Furthermore, channel number calibration is essential to the performance of the network and is likely to directly affect the inference time.

Even when searching for channel numbers only, the size of the search space is a challenge: if a network N with n layers is parametrized by its channel numbers $(c_0, \ldots, c_n)$, where c_i can take values among a set of choices $C_i \subseteq \mathbb{N}$, the size of the design space

$$\mathcal{S}_{\text{channels}} = \{N(c_0, c_1, \ldots, c_n), c_i \in C_i\} \qquad (1)$$

is exponential in the number of layers[1]. Therefore, efficient methods are needed to explore the search space, *e.g.* by relying on approximations, proxies, or by representing many elements of the search space using a single network.

Resource constraints. The resource constraints $\mathcal{C}$ in the NAS problem (problem 2.1) are hardware- and inference engine-specific constraints used in the target application. $\mathcal{C}$ considered by many NAS approaches is a bound on the number of FLOPs or performance during a single inference. While FLOPs can be seen as a metric broadly encompassing the desired physical limitations the inference is subjected to (*e.g.* latency and power consumption), it has been shown that FLOPs correlate poorly with these end metrics [29]. Therefore, specializing the NAS to a particular inference engine and expressing the resource constraints as a bound on the target platform limitations is of particular interest. This has given a rise to more resource-specific NAS approaches, using resource constraints of the form

$$\mathcal{C} = \{N | M(N) < M_T\} \qquad (2)$$

where $M(N)$ is the resource metric and M_T its target. $M(N)$ can represent latency, power consumption constraints, or combinations of these objectives [29, 30]. Given

---

[1]For ease of notation, we adopt $C_0 = \{I\}$ and $C_n = \{O\}$ where I is the number of input channels of the network and O its output channels, set by the application (*e.g.* 3 and 1000 resp. in an ImageNet classification task)

the size of the search space, NAS often requires evaluating the resource metric on a large number of networks during the course of the optimization. This makes it often impracticable to rely on performance measurements on-hardware during the search. Multiple methods therefore rely on a model, such as a latency model [22], which is learned beforehand and maps a given network N to an expected value of the resource metric $M(N)$ during the on-hardware inference.

3. Slimmable networks and AutoSlim

We now briefly review slimmable networks [33] and the AutoSlim [31] approach.

Slimmable networks. Slimmable neural network training [33] is designed to produce models that can be evaluated at various network widths at test time to account for different accuracy-latency trade-offs. At each training iteration t a random channel configuration $\boldsymbol{c}^t = (c_0^t, \ldots, c_n^t)$ is selected, where each channel number c_i is picked among a set of choices C_i representing the desired operating channels for layer i. This allows the optimization to account for the fact that number of channels will be selected dynamically at test time. The so-called *sandwich rule* (where each iteration minimizes the error of the maximum and minimum size networks in addition to a random configuration) and *in-place distillation* (application of knowledge-distillation [9] between the maximum network and smaller networks) have been further introduced by [32] to improve slimmable network training and increase accuracy of the resulting networks. Dynamically selecting channel numbers at test time requires re-computing of batch normalization statistics. [32] showed that for large batch sizes, these statistics can be estimated using the inference of a single batch, which is equivalent to using the batch normalization layer in *training mode* at test time.

Channel number search. A slimmable network is used for the determination of the optimized channel number configurations under specified resources constraints. This determination relies on the following assumption:

Assumption 3.1 (Slimmable NAS assumption). *The performance of a slimmable network evaluated for a given channel configuration $\boldsymbol{c} \in C_0 \times \ldots \times C_n$ is a good proxy for the performance of a neural network trained in a standard fashion with only this channel configuration.*

Given this assumption, AutoSlim proposes a greedy iterative trimming scheme in order to select the end channel configuration from a trained slimmable network. The procedure starts from the maximum channel configuration $\boldsymbol{c} = \boldsymbol{M}$. At each iteration:

- The performance of channel configuration $\boldsymbol{c}'^k = (c_0, \ldots, c_{k-1}, d, c_{k+1}, \ldots, c_n)$ with $d = \max_{c < c_k} C_k$ is measured on a validation set for all $k \in [1, n-1]$;

- The configuration among $(\boldsymbol{c}'^k)_{k=1 \ldots n-1}$ that least increases the validation error is selected for next iteration.

This trimming is repeated until the resource constraint $M(N(\boldsymbol{c})) < M_T$ is met. The output of AutoSlim is a channel configuration $\boldsymbol{c}$ that satisfies the resource constraint, which is then trained from scratch on the training set.

Discussion. Reliance on the one-shot slimmable network training makes AutoSlim training very efficient, while channel configuration inference via greedy iterative slimming is also performed efficiently by using only one large batch per tested configuration [31]. The greedy optimization strategy employed by AutoSlim is known to yield approximation guarantees with respect to an optimal solution for resource constrained performance maximization under certain assumptions on the underlying objective, notably submodularity [5]. However, in practice, optimization of a slimmable network configuration does not satisfy submodularity or related conditions, and the employment of an iterative greedy algorithm is heuristic.

In this work we also build on the ideas of slimmable network training. However, in contrast to AutoSlim, we show that better NAS solutions can be found by employing a non-greedy optimization scheme that considers the entire channel configuration search space and efficiently selects a single channel configuration meeting the resource requirements. This is achieved through the use of a Lagrangian relaxation of the NAS problem, statistic aggregation during training, and Viterbi decoding (section 5). Selecting optimal channel configuration under available compute constraints requires precise hardware-specific latency model. Thus in section 4 we propose an accurate and simple black-box latency estimation approach that allows to obtain a realistic hardware-specific latency model without the need for low-level access to the inference computation. Finally, we propose a biased path sampling to progressively reduce the search space at training time, allowing a gradual specialization of the training phase to fit the target computational constraints. Our dynamic approach (section 6) specializes the NAS proxy to our specific target and leads to improved accuracy-latency trade-offs in practice.

4. Black-box latency model for network width search

We propose a latency model suited to the quick evaluation of the latency of a network $L(N)$ with varying channel numbers, which we use in our method. While other works have designed latency models [6, 22, 26], creating an accurate model for the fine-grained channel number choices allowed by our method is challenging. In theory, the FLOPs of a convolutional layer scale as

$$c_{\text{in}} c_{\text{out}} W H k^2 / s^2, \tag{3}$$

where c_{in}, c_{out} are input and output channel numbers, (W, H) are the input spatial dimensions, k is the kernel size and s the stride. However, the dependency of the latency measured in practice to the number of FLOPs is highly non-linear. This can be explained by various factors: (i) parallelization of the operations make the latency dependent on external factors, such as the number of threads fitting on a device for given parameters; (ii) caching and memory allocation mechanisms are function of the input and output shapes; (iii) implementation of the operators in various inference libraries such as CuDNN or TensorRT are tuned towards a particular choice of channel numbers.

Rather than attempting to model the low-level phenomena that govern the dependency between the channel numbers and the inference time, we use a look-up table modelling the latency of each layer in the network as a function of the channel numbers. For each layer $i = 0 \ldots n - 1$, we encode as Θ_i the layer parameters that are likely to have an impact on the layer latency. In the case of the mobilenet-v1 network used in our experiments, we used $\Theta_i = (H, W, s, k, dw)$, where $H \times W$ the layer input size, s its stride, k its kernel size and $dw \in \{0, 1\}$ an indicator of the layer type: fully convolutional, or pointwise + depthwise convolutional. We assume that the latency can be written as a sum over layers

$$L(N(c_0, \ldots, c_n)) = \sum_{i=0}^{n-1} L_{\Theta_i}(c_i, c_{i+1}), \qquad (4)$$

where each layer's latency depends on the input and output channel numbers c_i, c_{i+1} as well as the fixed parameters Θ_i.

Populating each element $L_{\Theta_i}(c_i, c_j)$ in the lookup table is non-trivial. The goal is to measure the contribution of each individual layer to the global latency of the network. However, the measure of the inference latency of one layer in isolation includes a memory allocation and CPU communication overhead that is not necessarily present once the layer is inserted in the network. Indeed, memory buffers allocated on the device are often reused across different layers.

We therefore profile entire networks, rather than profiling individual layers in isolation. We measure the latency of a set of p channel configurations $(c^1 \ldots c^p)$ such that each individual layer configuration in our search space

$$\{L_{\Theta_i}(c_i, c_{i+1}), i \in [0, n-1], c_i \in C_i, c_{i+1} \in C_j\} \quad (5)$$

is sampled at least once. This sampling can be done uniformly among channel configurations, or biased towards unseen layer configurations using dynamic programming, as detailed in supplementary A. As a result, we obtain a set of measured latencies $(L(N(c^j)) = l_j)_{j=1 \ldots P}$, which by eq. (4) yield a linear system in the variables of our latency model $L_{\Theta_i}(c_i, c_{i+1})$

$$\sum_{i=0}^{n-1} L_{\Theta_i}(c_i^j, c_{i+1}^j) = l_j \quad \forall j = 1 \ldots P. \qquad (6)$$

This system can be summarized as $\boldsymbol{Ax} = \boldsymbol{l}$ where $\boldsymbol{A}$ is a sparse matrix encoding the profiled configurations, $\boldsymbol{l}$ is the corresponding vector of measured latencies and $\boldsymbol{x}$ contains all the variables in our latency model (i.e. the individual layer latencies in eq. (5)). We solve the linear system using least-squares to obtain the desired individual layer latencies.

We have found that this "black-box" approach results in a very accurate latency model for the search of channel numbers. The method is framework-agnostic and does not depend on the availability of low-level profilers on the inference platform. Moreover, access to a low-level profiler would still require solving the problem of assigning the memory allocation and transfers to the correct layer in the network. Our approach deals with this question automatically, and optimally assigns these overheads in order to best satisfy the assumed latency model of eq. (4).

The solution to linear system in eq. (6) can be slightly improved by adding monotonicity priors, enforcing inequalities of the form $L_{\Theta_i}(c_i, c_k) \leq L_{\Theta_i}(c_j, c_k)$ if $c_i < c_j$ and $L_{\Theta_i}(c_i, c_k) \leq L_{\Theta_i}(c_i, c_l)$ if $c_k < c_l$, as one expects the latency to be increasing in the number of input/output channels of the layer. Similar inequalities can be written between configurations with differing input sizes. It is straightforward to write all these inequalities as $\boldsymbol{Vx} \leq \boldsymbol{0}$ where $\boldsymbol{V}$ is a sparse matrix, and added to the least-squares problem. Rather than enforcing these inequalities in a hard way, we found it best to use a soft prior, which translates into

$$\min_{\boldsymbol{x}} \|\boldsymbol{Ax} - \boldsymbol{l}\|^2 + \lambda \|\max(\boldsymbol{Vx}, \boldsymbol{0})\|_1, \qquad (7)$$

where the weighting parameter λ is set using a validation set; this minimization can be solved efficiently using a second-order cone program solver [4, 17].

5. Optimal width search (OWS) via Viterbi inference

For the special case of optimizing the number of channels under a latency constraint, the NAS problem 2.1 writes as

$$\min_{\boldsymbol{c} \in C_0 \times \ldots \times C_n} \Delta(N(\boldsymbol{c})) \quad \text{s.t. } L(N(\boldsymbol{c})) < L_T \qquad (8)$$

with L_T our latency target. We consider the following Lagrangian relaxation of the problem:

$$\max_{\gamma} \min_{\boldsymbol{c}} \Delta(N(\boldsymbol{c})) + \gamma(L(N(\boldsymbol{c})) - L_T) \qquad (9)$$

with γ a Lagrange multiplier, similar to the formulation proposed by [21] for network compression. If the subproblems

$$\min_{\boldsymbol{c}} \Delta(N(\boldsymbol{c})) + \gamma L(N(\boldsymbol{c})) \qquad (10)$$

can be solved efficiently, the maximization in eq. (9) can be solved by binary search over γ by using the fact that the objective is concave in γ [2, prop. 5.1.2]. This corresponds

to setting the runtime penalty in eq. (10) high enough that the constraint is satisfied but no higher.

Our key idea to ensure that eq. (10) can be solved efficiently is to find an estimate of the error of a network that decomposes over the individual channel choices as $\Delta(N(\boldsymbol{c})) \approx \sum_{i=1}^{n-1} \delta_i(c_i)$; indeed, given that our latency model decomposes over pairs of successive layers (eq. (4)), this form allows to write eq. (10) as

$$\min_{\boldsymbol{c}} \sum_{i=1}^{n-1} \delta_i(c_i) + \gamma \sum_{i=0}^{n-1} L_{\Theta_i}(c_i, c_{i+1}), \qquad (11)$$

which is solved efficiently by the Viterbi algorithm [27] applied to the pairwise MRF illustrated in fig. 1

We leverage this efficient selection algorithm in a procedure that we detail in the remainder of this section. As in section 3, we train a slimmable network. In order to ensure faster exploration of the search space, rather than sampling one unique channel configuration per training batch, we sample a different channel configuration separately for each element in the batch. This can be implemented efficiently at each layer i by first computing the "max-channel" output for all elements in the batch, before zeroing-out the channels above the sampled channel numbers for each individual element. This batched computation is in aggregate faster than a separate computation for each element.

For each training example $\boldsymbol{x}^{(t)}$, a random configuration $\boldsymbol{c}^{(t)}$ is sampled, yielding a loss $\ell(\boldsymbol{x}^{(t)}, \boldsymbol{c}^{(t)})$; we also retain the value of the loss corresponding to the maximum channel configuration $\ell(\boldsymbol{x}^{(t)}, \boldsymbol{M})$ – available due to *sandwich rule* training (section 3). For each $i = 1 \ldots n - 1$, we consider all training iterations $T_i(c_i) = \{t \mid c_i^{(t)} = c_i\} \subseteq \mathbb{N}$ where a particular channel number $c_i \in C_i$ was used. We then define

$$\delta_i(c_i) = \frac{1}{|T_i(c_i)|} \sum_{t \in T_i(c_i)} \ell(\boldsymbol{x}^{(t)}, \boldsymbol{c}^{(t)}) - \ell(\boldsymbol{x}^{(t)}, \boldsymbol{M}) \quad (12)$$

as the per-channel error rates in eq. (11). Measuring the loss relative to the maximum configuration loss follows the intuition that good channel numbers lead to lower losses on average. Empirically, we found that computing the average in eq. (12) over the last training epoch yields good results.

Equation (11) is designed for efficient inference by neglecting the interaction between the channel numbers of different layers. We show in our experiments (section 7) that this trade-off between inference speed and modeling accuracy compares favorably to the greedy optimization strategy described in section 3. On the one hand, the number of training iterations considered in eq. (12) is sufficient to ensure that the per-channel error rates are well estimated. Approaches that would consider higher-order interactions between channel numbers would require an exponentially higher number of iterations to achieve estimates with the same level of statistical accuracy. On the other hand, this decomposition
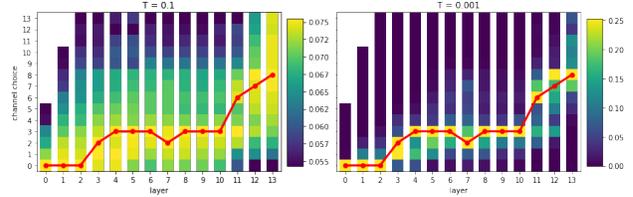

Figure 2: Min-sum relaxation at temperatures $T = 0.1$ (left) and $T = 0.001$ (right). Red: min-sum path. Colored: marginal sampling probabilities. The sampled configurations approach the min-sum path as $T \to 0$.

allows the performance of an exhaustive search over channel configurations using the Viterbi algorithm (eq. (11)). We have observed that this selection step takes a fraction of a second and does not get stuck in local optima as occurs when using a greedy approach. The greedy approach, by contrast, took hours to complete.

6. Adaptive refinement of Optimal Width Search (AOWS)

We have seen in section 5 how layerwise modeling and Viterbi inference allows for an efficient global search over configurations. In this section, we describe how this efficient selection procedure can be leveraged in order to refine the training of the slimmable network thereby making assumption 3.1 more likely to hold.

Our strategy for adaptive refinement of the training procedure stems from the following observation: during the training of the slimmable model, by sampling uniformly over the channel configurations we visit many of configurations that have a latency greater than our objective L_T, or that have a poor performance according to our current channel estimates $\delta_i(c_i)$. As the training progresses, the sampling of the channel configurations should be concentrated around the region of interest in the NAS search space.

In order to refine the sampling around the solutions close to the minimum of eq. (11), we relax the Viterbi algorithm (min-sum) using a differentiable dynamic programming procedure described in [16]. This strategy relaxes the minimization in eq. (11) into a smoothed minimization, which we compute by replacing the min operation by a log-sum-exp operation in the Viterbi forward pass. The messages sent from variable c_i to variable c_{i+1} become

$$m(c_{i+1}) = \log \sum_{c_i} \exp -\frac{1}{T}\big(m(c_i) + \delta_i(c_{i+1}) \\ + \gamma L_{\Theta_i}(c_i, c_{i+1})\big), \qquad (13)$$

where T is a temperature parameter that controls the smoothness of the relaxation. The forward-backward pass of the relaxed min-sum algorithm yields log-marginal probabilities $\log p_i(c_i)$ for each layer whose mass is concentrated close

to configurations minimizing eq. (11). For $T = 1$, these correspond to the marginal probabilities of the pairwise CRF defined by the energy of eq. (11). In the limit $T \to 0$, the probabilities become Dirac distributions corresponding to the MAP inference of the CRF as computed by the Viterbi algorithm (fig. 2).

We introduce the following dynamic training procedure. First, we train a slimmable network for some warmup epochs, using uniform sampling of the configurations as in section 3. We then turn to a biased sampling scheme. We initially set $T = 1$. At each iteration, we

1. sample batch configurations according to the marginal probabilities $p_i(c_i)$,
2. do a training step of the network,
3. update the unary statistics (eq. (12)),
4. decrease T according to an annealing schedule.

This scheme progressively favours configurations that are close to minimizing eq. (11). This reduction in diversity of the channel configurations ensures that:

- training of the slimmable model comes closer to the training of a single model, thereby making assumption 3.1 more likely to hold;
- per-channel error rates (eq. (12)) are averaged only over relevant configurations, thereby enforcing an implicit coupling between the channel numbers of different layers in the network.

Our experiments highlight how this joint effect leads to channel configurations with a better accuracy/latency trade-off.

7. Experiments

Experimental setting. We focus on the optimization of the channel numbers of MobileNet-v1 [11]. The network has 14 different layers with adjustable width. We consider up to 14 channel choices for each layer i, equally distributed between 20% and 150% of the channels of the original network. These numbers are rounded to the nearest multiple of 8, with a minimum of 8 channels. We train AOWS and OWS models for 20 epochs with batches of size 512 and a constant learning rate 0.05. For the AOWS versions, after 5 warmup epochs (with uniform sampling), we decrease the temperature following a piece-wise exponential schedule detailed in supplementary B. We train the selected configurations with a training schedule of 200 epochs, batch size 2048, and the training tricks described in [8], including cosine annealing [14].

7.1. TensorRT latency target

We first study the optimization of MobileNet-v1 under TensorRT (TRT)[2] inference on a NVIDIA V100 GPU. Table 1 motivates this choice by underlining the speedup al-

---

[2] https://developer.nvidia.com/tensorrt

Table 1: Run-time vs. accuracy comparison for timings obtained with batch size 64. The GPU+TRT column lists the latencies and speedups allowed by TensorRT. AOWS is obtained with Mobilenet-v1/TensorRT optimization with $L_T = 0.04$ms, and a longer training schedule of 480 epochs.

Method	GPU ms/fr	GPU+TRT ms/fr $_{speedup}$	Top-1 Error (%)
AOWS	0.18	**0.04** $_{4.5x}$	27.5
AutoSlim [31]	0.15	**0.04** $_{3.75x}$	28.5
Mobilenet-v1 [11]	0.25	0.05 $_{5x}$	29.1
Shufflenet-v2 [15]	0.13	0.07 $_{1.9x}$	30.6
MNasNet [23]	0.26	0.07 $_{3.7x}$	26.0
SinglePath-NAS [22]	0.28	0.07 $_{4.0x}$	25.0
ResNet-18 [7]	0.25	0.08 $_{3.1x}$	30.4
FBNet-C [28]	0.32	0.09 $_{3.6x}$	25.1
Mobilenet-v2 [20]	0.28	0.10 $_{2.8x}$	28.2
Shufflenet-v1 [34]	0.21	0.10 $_{2.1x}$	32.6
ProxylessNAS-G [3]	0.31	0.12 $_{2.7x}$	24.9
DARTS [13]	0.36	0.16 $_{2.3x}$	26.7
ResNet-50 [7]	0.83	0.19 $_{4.3x}$	23.9
Mobilenet-v3-large [10]	0.30	0.20 $_{1.5x}$	24.8
NASNet-A* [35]	0.60	-	26.0
EfficientNet-b0 [24]	0.59	0.47 $_{1.3x}$	23.7

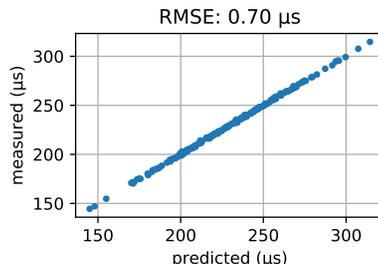* TRT inference failed due to loops in the underlying graph

Figure 3: Measured vs. predicted latency of 200 randomly sampled networks in our search space for the TensorRT latency model, trained using 9500 inference samples.

lowed by TRT inference, compared to vanilla GPU inference under the MXNet framework. While the acceleration makes TRT attractive for production environments, we see that it does not apply uniformly across architectures, varying between 1.3x for EfficientNet-b0 and 5x for mobilenet-v1.

Latency model. Figure 3 visualizes the precision of our latency model as described in section 4 for 200 randomly sampled configurations in our search space, and show that our pairwise decomposable model (section 4) adequately predicts the inference time on the target platform.

Proxy comparison. Figure 4 shows the correlation between the error predictor and the observed errors for several
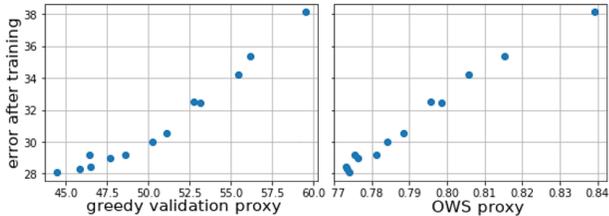
Figure 4: Comparison of the slimmable proxy used by greedy validation (left) and our error estimates used in OWS (right). The proxy errors of 13 networks in our search space are compared to the final error after full training of these configurations.

networks in our search space. The slimmable proxy used in AutoSlim uses the validation errors of specific configurations in the slimmable model. OWS uses the simple layerwise error model of eq. (12). We see that both models have good correlation with the final error. However, the slimmable proxy requires a greedy selection procedure, while the layerwise error model leads to an efficient and global selection.

Optimization results. We set the TRT runtime target $L_T = 0.04$ms, chosen as the reference runtime of AutoSlim mobilenet-v1. Table 2 gives the final top-1 errors obtained by the configurations selected by the different algorithms. *greedy* reproduces AutoSlim greedy selection procedure with this TRT latency target on the slimmable proxy (section 3). OWS substitutes the global selection algorithm based on channel estimates (eq. (11)). Finally, AOWS uses the adaptive path sampling procedure (section 6). Figure 5 illustrates the differences between the found configurations (which are detailed in supplementary D). As in [31], we observe that the configurations generally have more weights at the end of the networks, and less at the beginning, compared to the original mobilenet-v1 architecture [11].

Despite the simplicity of the per-channel error rates, we see that OWS leads to a superior configuration over greedy, on the same slimmable model. This indicates that greedy selection can fall into local optimas and miss more advantageous global channel configurations. The AOWS approach uses the Viterbi selection but adds an adaptive refinement of the slimmable model during training, which leads to superior final accuracy.

Table 1 compares the network found by AOWS with architectures found by other NAS approaches. The proposed AOWS reaches the lowest latency on-par with AutoSlim [31], while reducing the Top-1 image classification error by 1%. This underlines the importance of the proposed platform-specific latency model, and the merits of our algorithm.

AOWS training epochs. One important training hyperparameter is the number of training epochs of AOWS. Table 3 shows that training for 10 epochs leads to a suboptimal

Table 2: Accuracies and latencies of channel configurations found for TRT optimization with $L_T = 0.04$ms.

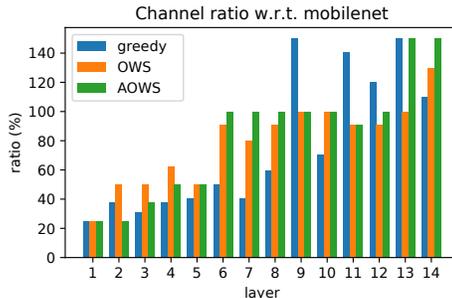| Method	ms/fr	Top-1 error (%)
greedy	0.04	29.3
OWS	0.04	28.2
AOWS	0.04	27.8

Figure 5: Channel configurations found, as a ratio with respect to the original channels of MobileNet-v1.

model; however, the results at epoch 30 are on-par with the results at epoch 20, which motivates our choice of picking our results at epoch 20.

Table 3: Effect of the number of epochs when training AOWS, for TRT optimization under $L_T = 0.04$ms.

Epochs	10	20	30
Top-1 error (%)	28.1	27.8	27.9

7.2. FLOPS, CPU and GPU targets

We experiment further with the application of AOWS to three different target constraints. First, we experiment with a FLOPs objective. The expression of the FLOPs decomposes over pairs of successive channel numbers, and can therefore be written analytically as a special case of our latency model (section 4). Table 4 gives the FLOPs and top-1 errors obtained after end-to-end training of the found configurations. We note that final accuracies obtained by AOWS are on-par or better than the reproduced AutoSlim variant (greedy). AutoSlim [31] lists better accuracies in the 150 and 325 MFLOPs regimes; we attribute this to different choice of search space (channel choices) and training hyperparameters, which were not made public; one other factor is the use of a 480 epochs training schedule, while we limit to 200 here.

We turn to realistic latency constraints, considering CPU inference on an Intel Xeon CPU with batches of size 1, and GPU inference on an NVIDIA V100 GPU with batches of

Table 4: Optimizing for FLOPs

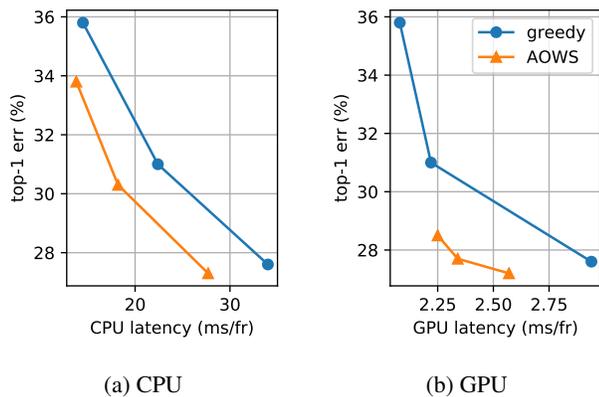| Variant	MFLOPs	Top-1 error (%)
AutoSlim [31]	150	32.1
greedy$_{150}$	150	35.8
AOWS	150	35.9
AutoSlim [31]	325	28.5
greedy$_{325}$	325	31.0
AOWS	325	29.7
AutoSlim [31]	572	27.0
greedy$_{572}$	572	27.6
AOWS	572	26.7

(a) CPU (b) GPU

Figure 6: Pareto front of greedy, vs. Pareto front of AOWS optimized for CPU and GPU latency models.

size 16, under PyTorch [18].[3] Tables 5 and 6 show the results for 3 latency targets, and the resulting measured latency. We also report the latencies of the channel numbers on the greedy solution space corresponding to the three configurations in table 4. By comparison of the accuracy/latency tradeoff curves in fig. 6, it is clear that using AOWS leads to more optimal solutions than greedy; in general, we consistently find models that are faster and more accurate.

We observe that the gains of AOWS over greedy are more consistent than in the case of the FLOPs optimization (section 4). We note that the analytical FLOPs objective varies more regularly in the channel configurations, and therefore presents less local optima, than empirical latencies measured on-device. This might explain why the greedy approach succeeds at finding appropriate configurations in the case of the FLOPs model better than in the case of realistic latency models.

8. Conclusion

Efficiently searching for novel network architectures while optimizing accuracy under latency constraints on a

---

[3]See supplementary C for details on framework version and harware.

Table 5: Optimizing for CPU latency (@ indicates the latency targets)

Variant	ms/fr	Top-1 error (%)
AOWS @ 15ms	13.8	33.8
AOWS @ 20ms	18.2	30.3
AOWS @ 30ms	27.7	27.3
greedy$_{150}$	14.5	35.8
greedy$_{325}$	22.4	31.0
greedy$_{572}$	34.0	27.6

Table 6: Optimizing for GPU latency (@ indicates the latency targets)

Variant	ms/fr	Top-1 error (%)
AOWS @ 2.2ms	2.25	28.5
AOWS @ 2.4ms	2.34	27.7
AOWS @ 2.6ms	2.57	27.2
greedy$_{150}$	2.08	35.8
greedy$_{325}$	2.22	31.0
greedy$_{572}$	2.94	27.6

target platform and task is of high interest for the computer vision community. In this paper we propose a novel efficient one-shot NAS approach to optimally search CNN channel numbers, given latency constraints on a specific hardware. To this end, we first design a simple but effective black-box latency estimation approach to obtain precise latency model for a specific hardware and inference modality, without the need for low-level access to the inference computation. Then, we introduce a pairwise MRF framework to score any network channel configuration and use the Viterbi algorithm to efficiently search for the most optimal solution in the exponential space of possible channel configurations. Finally, we propose an adaptive channel configuration sampling strategy to progressively steer the training towards finding novel configurations that fit the target computational constraints. Experiments on ImageNet classification task demonstrate that our approach can find networks fitting the resource constraints on different target platforms while improving accuracy over the state-of-the-art efficient networks. Code and trained models have been released at http://github.com/bermanmaxim/AOWS.

References

[1] Peter J. Angeline, Gregory M. Saunders, and Jordan B. Pollack. An evolutionary algorithm that constructs recurrent neural networks. *IEEE transactions on neural networks*, 5(1):54–65, 1994. 1, 2

[2] Dimitri P Bertsekas. *Nonlinear programming*. Athena Scientific, 2nd edition, 1995. 4

[3] Han Cai, Ligeng Zhu, and Song Han. ProxylessNAS: Direct neural architecture search on target task and hardware. In *ICLR*, 2019. 1, 2, 6

[4] Steven Diamond and Stephen Boyd. CVXPY: A Python-embedded modeling language for convex optimization. *Journal of Machine Learning Research*, 17(83):1–5, 2016. 4

[5] Satoru Fujishige. *Submodular Functions and Optimization*. Elsevier, 2005. 3

[6] Jussi Hanhirova, Teemu Kämäräinen, Sipi Seppälä, Matti Siekkinen, Vesa Hirvisalo, and Antti Ylä-Jääski. Latency and throughput characterization of convolutional neural networks for mobile computer vision. In *Proceedings of the 9th ACM Multimedia Systems Conference*, pages 204–215, 2018. 3

[7] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, pages 770–778, 06 2016. 6

[8] Tong He, Zhi Zhang, Hang Zhang, Zhongyue Zhang, Junyuan Xie, and Mu Li. Bag of tricks for image classification with convolutional neural networks. *ArXiv*, abs/1812.01187, 2018. 6

[9] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015. 3

[10] Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, Quoc V. Le, and Hartwig Adam. Searching for mobilenetv3. In *The IEEE International Conference on Computer Vision (ICCV)*, October 2019. 6

[11] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *ArXiv*, abs/1704.04861, 2017. 6, 7, II

[12] Chenxi Liu, Barret Zoph, Maxim Neumann, Jonathon Shlens, Wei Hua, Li-Jia Li, Li Fei-Fei, Alan Yuille, Jonathan Huang, and Kevin Murphy. Progressive neural architecture search. In *The European Conference on Computer Vision (ECCV)*, September 2018. 1, 2

[13] Hanxiao Liu, Karen Simonyan, and Yiming Yang. DARTS: Differentiable architecture search. In *International Conference on Learning Representations*, 2019. 2, 6

[14] Ilya Loshchilov and Frank Hutter. Sgdr: Stochastic gradient descent with restarts. *ArXiv*, abs/1608.03983, 2016. 6

[15] Ningning Ma, Xiangyu Zhang, Hai-Tao Zheng, and Jian Sun. Shufflenet v2: Practical guidelines for efficient cnn architecture design. In *The European Conference on Computer Vision (ECCV)*, September 2018. 6

[16] Arthur Mensch and Mathieu Blondel. Differentiable dynamic programming for structured prediction and attention. In *ICML*, 2018. 5

[17] B. O'Donoghue, E. Chu, N. Parikh, and S. Boyd. Conic optimization via operator splitting and homogeneous self-dual embedding. *Journal of Optimization Theory and Applications*, 169(3):1042–1068, June 2016. 4

[18] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. In *NIPS-W*, 2017. 8

[19] Hieu Pham, Melody Y. Guan, Barret Zoph, Quoc V. Le, and Jeff Dean. Efficient neural architecture search via parameter sharing. *International Conference on Machine Learning*, 2018. 1, 2

[20] Mark Sandler, Andrew G. Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4510–4520, 2018. 2, 6

[21] Shivangi Srivastava, Maxim Berman, Matthew B. Blaschko, and Devis Tuia. Adaptive compression-based lifelong learning. In *Proceedings of the British Machine Vision Conference (BMVC)*, 2019. 4

[22] Dimitrios Stamoulis, Ruizhou Ding, Di Wang, Dimitrios Lymberopoulos, Bodhi Priyantha, Jie Liu, and Diana Marculescu. Single-path nas: Device-aware efficient convnet design. *ArXiv*, abs/1905.04159, 2019. 1, 2, 3, 6

[23] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V. Le. MnasNet: Platform-aware neural architecture search for mobile. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019. 1, 2, 6

[24] Mingxing Tan and Quoc V. Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In *ICML*, 2019. 6

[25] Frank Hutter Thomas Elsken, Jan Hendrik Metzen. Neural architecture search: A survey. *ArXiv*, abs/1808.05377, 2018. 1, 2

[26] Stylianos I. Venieris and Christos-Savvas Bouganis. Latency-driven design for FPGA-based convolutional neural networks. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, 2017. 3

[27] Andrew Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE transactions on Information Theory*, 13(2):260–269, 1967. 5

[28] Bichen Wu, Xiaoliang Dai, Peizhao Zhang, Yanghan Wang, Fei Sun, Yiming Wu, Yuandong Tian, Peter Vajda, Yangqing Jia, and Kurt Keutzer. Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019. 1, 2, 6

[29] Tien-Ju Yang, Yu-Hsin Chen, and Vivienne Sze. Designing energy-efficient convolutional neural networks using energy-aware pruning. *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 6071–6079, 2016. 2

[30] Tien-Ju Yang, Andrew Howard, Bo Chen, Xiao Zhang, Alec Go, Mark Sandler, Vivienne Sze, and Hartwig Adam. Netadapt: Platform-aware neural network adaptation for mobile applications. In *ECCV*, 2018. 2

[31] Jiahui Yu and Thomas Huang. AutoSlim: Towards One-Shot Architecture Search for Channel Numbers. *arXiv e-prints*, page arXiv:1903.11728, Mar 2019. 1, 2, 3, 6, 7, 8

[32] Jiahui Yu and Thomas S. Huang. Universally slimmable networks and improved training techniques. In *The IEEE International Conference on Computer Vision (ICCV)*, October 2019. 1, 2, 3

[33] Jiahui Yu, Linjie Yang, Ning Xu, Jianchao Yang, and Thomas Huang. Slimmable neural networks. In *International Conference on Learning Representations*, 2019. 1, 2, 3

[34] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. Shufflenet: An extremely efficient convolutional neural network for mobile devices. *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 6848–6856, 2018. 6

[35] Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *ICLR*, 2016. 1, 2, 6

[36] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V. Le. Learning transferable architectures for scalable image recognition. *CoRR*, abs/1707.07012, 2017. 1, 2

AOWS: adaptive and optimal network width search with latency constraints

Supplementary

A. Latency model: biased sampling

We describe the biased sampling strategy for the latency model, as described in section 4. Using the notations of section 4, the latency model is the least-square solution of a linear system $Ax = l$. The variables of the system are the individual latency of every layer configuration $L_{\Theta_i}(c_i, c_{i+1})$. To ensure that the system is complete, each of these layer configurations must be present at least once among the model configurations benchmarked in order to establish the latency model. Instead of relying on uniform sampling of the channel configurations, we can bias the sampling in order to ensure that the variable of the latency model $L_{\Theta_i}(c_i, c_{i+1})$ that has been sampled the least amount of time is present.

As in AOWS, we rely on a Viterbi algorithm in order to determine the next channel configuration to be benchmarked. Let $N(c_i, c_{i+1})$ be the number of times variable $L_{\Theta_i}(c_i, c_{i+1})$ has already been seen in the benchmarked configurations, and

$$M = \min_{\substack{i \in [0, n-1] \\ c_i \in C_i \\ c_{i+1} \in C_{i+1}}} N(c_i, c_{i+1}) \qquad (A.1)$$

the minimum value taken by N. The channel configuration we choose for the next benchmarking is the solution minimizing the pairwise decomposable energy

$$\min_{c_0, \ldots, c_n} \sum_{i=0}^{n-1} -[N(c_i, c_{i+1}) = M]. \qquad (A.2)$$

using the Iverson bracket notation. This energy ensures that at least one of the least sampled layer configurations is present in the sampled configuration.

This procedure allows to set a lower bound on the count of all variables among the benchmarked configurations. The sampling can be stopped when the latency model has reached an adequate validation accuracy.

B. Optimization hyperparameters

For the temperature parameter, we used a piece-wise exponential decay schedule, with values 1 at epoch 5, to 10^{-2} at epoch 6, 10^{-3} at epoch 10, and $5 \cdot 10^{-4}$ at epoch 20.

C. Framework versions and CPU/GPU models

We detail the frameworks and hardware used in the experiments of sections 7.1 and 7.2. Although we report latencies in terms of ms/frame, the latency models are estimated with batches of size bigger than 1. In general, we want to stick to realistic operating settings: GPUs are more efficient for bigger batches, and the batch choice impacts the latency/throughput tradeoff.

The *TRT experiments* are done on an NVIDIA V100 GPU with TensorRT 5.1.5 driven by MXNet v1.5, CUDA 10.1, CUDNN 7.6, with batches of size 64. The *CPU inference experiments* are done on an Intel Xeon® Platinum 8175 with batches of size 1, under PyTorch 1.3.0. The *GPU inference experiments* are done on an NVIDIA V100 GPU with batches of size 16, under PyTorch 1.3.0 and CUDA 10.1.

D. Layer channel numbers and final configuration numbers found

In table D.1, we detail the search space in the channel numbers described in section 7.

Table D.1: Search space: channel configurations for all 14 layers in MobileNet-v1. The first layer always has an input with 3 channels; the last layer always outputs 1000 channels for ImageNet classification. The bold values indicate the initial MobileNet-v1 configuration numbers.

i	C_i
1	8, 16, 24, **32**, 40, 48
2	16, 24, 32, 40, 48, 56, **64**, 72, 80, 88, 96
3	24, 40, 48, 64, 80, 88, 104, 112, **128**, 144, 152, 168, 176, 192
4	24, 40, 48, 64, 80, 88, 104, 112, **128**, 144, 152, 168, 176, 192
5	48, 80, 104, 128, 152, 176, 208, 232, **256**, 280, 304, 336, 360, 384
6	48, 80, 104, 128, 152, 176, 208, 232, **256**, 280, 304, 336, 360, 384
7	104, 152, 208, 256, 304, 360, 408, 464, **512**, 560, 616, 664, 720, 768
8	104, 152, 208, 256, 304, 360, 408, 464, **512**, 560, 616, 664, 720, 768
9	104, 152, 208, 256, 304, 360, 408, 464, **512**, 560, 616, 664, 720, 768
10	104, 152, 208, 256, 304, 360, 408, 464, **512**, 560, 616, 664, 720, 768
11	104, 152, 208, 256, 304, 360, 408, 464, **512**, 560, 616, 664, 720, 768
12	104, 152, 208, 256, 304, 360, 408, 464, **512**, 560, 616, 664, 720, 768
13	208, 304, 408, 512, 616, 720, 816, 920, **1024**, 1128, 1232, 1328, 1432, 1536
14	208, 304, 408, 512, 616, 720, 816, 920, **1024**, 1128, 1232, 1328, 1432, 1536

I

Table D.2: Channel configurations found in the TRT optimization (section 7.1), visualized in fig. 5, and with top-1 errors given in table 2 in the paper. Results are compared to the original Mobilenet-v1 [11] channels.

method	configuration
greedy	8, 24, 40, 48, 104, 128, 208, 304, 768, 360, 720, 616, 1536, 1128
OWS	8, 32, 64, 80, 128, 232, 408, 464, 512, 512, 464, 464, 1024, 1328
AOWS	8, 16, 48, 64, 128, 256, 512, 512, 512, 512, 464, 512, 1536, 1536
Mobilenet-v1 [11]	32, 64, 128, 128, 256, 256, 512, 512, 512, 512, 512, 512, 1024, 1024