NeuralScale: Efficient Scaling of Neurons for Resource-Constrained Deep Neural Networks

Eugene Lee Chen-Yi Lee
Institute of Electronics, National Chiao Tung University
eugenelet.ee06g@nctu.edu.tw cylee@si2lab.org

Abstract

Deciding the amount of neurons during the design of a deep neural network to maximize performance is not intuitive. In this work, we attempt to search for the neuron (filter) configuration of a fixed network architecture that maximizes accuracy. Using iterative pruning methods as a proxy, we parameterize the change of the neuron (filter) number of each layer with respect to the change in parameters, allowing us to efficiently scale an architecture across arbitrary sizes. We also introduce architecture descent which iteratively refines the parameterized function used for model scaling. The combination of both proposed methods is coined as NeuralScale. To prove the efficiency of NeuralScale in terms of parameters, we show empirical simulations on VGG11, MobileNetV2 and ResNet18 using CIFAR10, CIFAR100 and TinyImageNet as benchmark datasets. Our results show an increase in accuracy of 3.04%, 8.56% and 3.41% for VGG11, MobileNetV2 and ResNet18 on CIFAR10, CIFAR100 and TinyImageNet respectively under a parameter-constrained setting (output neurons (filters) of default configuration with scaling factor of 0.25).

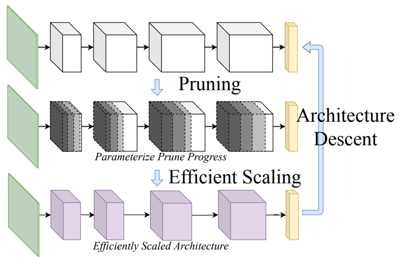
1. Introduction

The human brain contains around 100 billion of neurons [21] that are structured in such a way that they are utilized in an efficient manner. As the design of deep neural network (DNN) is inspired by the human brain, there's one key ingredient that is missing from the current design of DNNs: the efficient utilization of resources (parameters).

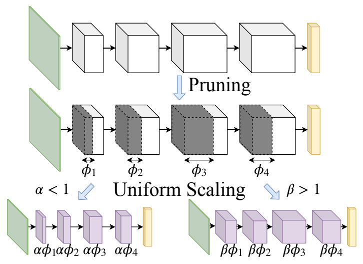
The success of DNN is a composition of many factors. On an architectural level, various architectures have been proposed to increase the accuracy of DNNs targeting efficiency in computational cost (FLOPs) and size (parameters). In a modern DNN architectures, hyperparameters like width (neurons/filters), depth, skip-connections and activation functions [38, 42] are the building blocks. Notable architectures that are constructed using those building blocks are: VGGNet [44], ResNet [18], DenseNet [24], GoogLeNet [45] and MobileNets [23, 43, 22]. Apart from the advances in architecture design, initialization of weights also helps in improving the accuracy of a DNN [11, 17].

We focus on optimizing the configuration of convolutional neural networks (CNNs) and shed light on the selection of the number of filters for each layer given a fixed architecture and depth. Our approach is complementary to the

Work supported by MOST of Taiwan: 107-2221-E-009-125-MY3.



(a) NeuralScale.


Pruning
ϕ_1
ϕ_2
ϕ_3
ϕ_4
$\alpha < 1$
Uniform Scaling
$\beta > 1$
$\alpha\phi_1$ $\alpha\phi_2$ $\alpha\phi_3$ $\alpha\phi_4$
$\beta\phi_1$ $\beta\phi_2$ $\beta\phi_3$ $\beta\phi_4$

(b) Uniform Scaling.

Figure 1: The input is represented as a green slab on the left, the output layer is the yellow bar on the right and intermediate layers are represented by 3D blocks with the width as its neuron (filter) number. The purple blocks are the final configuration of the neurons (filters). (a) shows our proposed method that non-linearly scales neurons (filters) across different layers to maximize performance. (b) is a uniform scaling method that is shown to be effective in [12].

modern variants of CNNs (VGGNet, ResNet, MobileNet, etc.) through the introduction of a guided approach in tuning its width instead of just blindly stacking additional layers to boost accuracy. Our approach also investigates the conventional wisdom on filter selection stating that as we go deeper into the network, more filters are required to capture high-level information embedded in the features and to compensate with the gradual reduction in the spatial dimension which has efficiency in FLOPs as a byproduct.

Intuitively speaking, the design of traditional CNNs is ad-hoc and introduces redundancy [29, 4]. This redundancy gives the opportunity for filter pruning techniques [15, 13, 29, 32] to strive, by conserving or improving accuracy using a lower parameter count. Current advances in pruning has led to a recent work by Liu *et al.* [33] that studies pruning in a new perspective. They show that pruning can be viewed as an architecture search method instead of just for removing redundancy. We incorporated this perspective along with the recent findings of EfficientNet [47] stating that through the search of an optimal ratio between the width, depth and resolution of a given architecture and dataset, the accuracy of a network can improve if scaled accordingly. Both these works led us to think that instead of finding the scale or ratio among the width, depth and resolution, we can scale the width of a CNN across several layers independently using global iterative pruning as a proxy. We hypothesize that if we are given a DNN with minimal redundant parameters, through the modeling of the change of neurons (filters) in each layer of the DNN with respect to the change in the total parameters of the DNN, we are able to scale the DNN across various sizes efficiently. Our approach can also be viewed as a variant of neural architecture search (NAS) where the search is on finding the optimal configuration of neurons (filters) across layers instead of searching for the optimal structure involving skip-connections or filter types [54, 30, 41, 50, 1, 31]. Our approach is comparatively light-weight as the only resource intensive task lies on the pruning of network. The gist of our proposed method is shown in Figure 1a.

The rest of the paper is structured as follows. We first show related work on available pruning techniques and the role of pruning for neural architecture search in Section 2. We then show the details of our approach in Section 3. Extensive experiments on our proposed method is shown in Section 4. We finally conclude our paper in Section 5.

2. Related Work

Pruning of Deep Neural Network. Pruning of neural networks has been studied for decades with the goal of parameter reduction [27, 16]. Parameter reduction is important if a DNN needs to be deployed to targeted devices with limited storage capacity and computational resources. There are two ways to prune a network: structured pruning [32, 20, 39, 28, 34, 52] and unstructured pruning [13, 14, 15, 9]. For structured pruning, entire filters or neurons are removed from a layer of a network. Such pruning method can be deployed directly to a general computing architecture, e.g. GPU, to see improvement in speed. For unstructured pruning, individual weights are pruned, resulting in a sparse architecture. A dedicated hardware is required to exploit the speed-up contributed by the sparsity of weights.

To prune a network, there are various criteria that have been studied. The most intuitive approach is to prune the weights based on its magnitude [15, 13, 29]. It was believed that the importance of weights is related to its magnitude. Although this approach is widely used in other works [9, 29, 19, 51], it is also shown on several works [35, 20, 27, 16] that magnitude pruning does not result in an architecture with the best performance. Pruning based on magnitude is adopted because of its simplicity when compared to more complicated pruning techniques, e.g. [16, 27] requires the computation of Hessian for pruning. A study proposed the use of geometric median [20] as a replacement of magnitude pruning for the criteria of network pruning. [35, 37] has also challenged the reliability of magnitude pruning and proposed the use of Taylor expansions to approximate the contribution of a filter to the final loss. Another intuitive way of pruning is through the addition of a regularizer to the loss function to force redundant weights to zero during training [15]. It has also been discovered that the scaling parameter used in Batch Normalization (BN) [25] can be used for structured pruning and yields performance better than pruning using magnitude [32]. A follow-up work that takes the shift term of BN into consideration for pruning is proposed in [52].

Neural Architecture Search via Pruning. There's a tremendous surge of efforts placed into the research of neural architecture search (NAS) techniques in the recent years on coming up with the most efficient architecture possible for a given task [54, 30, 41, 46, 3, 31]. NAS techniques are usually computationally expensive, limiting its applicability to research or corporate environment with limited computing resources. The search space of NAS is very broad and is defined distinctively across different works. Most of the search space involves the search of a suitable set of operations to be placed in a cell. The connections between different operations is also considered in the search space [2]. These cells are then stacked to increase network depth.

In our work, we focus on the decision of the number of neurons (filters) required for each layer in a DNN. We use existing pruning techniques as a proxy to tackle this problem. The idea of using pruning as an architecture search method is not novel and has been discussed in [33, 9, 10, 7] where its applicability can be seen in MorphNet [12]. Liu *et al.* [33] show that through pruning, we are removing redundancy from a network and the resulting network is efficient in terms of parameters. They also show that training the pruned architecture from scratch has comparable, if not better, accuracy than fine-tuned networks, indicating that the accuracy gain is from the resulting efficient architecture obtained via pruning. For the case of unstructured pruning, it is studied in [9, 10, 53] that a Lottery Ticket (LT) can be found via iterative unstructured pruning. A LT is a sparse architecture that is the result of unstructured pruning and has accuracy better than the original network (usually found at a parameter count of an order less than the parameter count of the original network). This finding indicates that pruning does introduce inductive bias [5] and adds evidence on the suitability of using pruning as an architecture search technique. This idea is proven in MorphNet [12] where a pruned architecture is scaled uniformly to meet the targeted parameter count and is repeated for several iterations. A single iteration of [12] is illustrated in Figure 1b.

3. Method

3.1. Parameter Tracking via Iterative Pruning

To efficiently allocate neurons (filters) across different layers of a DNN that results in optimal accuracy given a parameter constraint, we model the change of neurons (filters) across layers with respect to the change of parameters. First, we need to begin with a network with minimal redundant parameters. To do so, a structured pruning method proposed by Molchanov *et al.* [35] that prunes iteratively is adopted. They proposed a pruning method that prunes neurons based on its importance. The importance of a parameter can be measured as the loss induced when it's removed from the network. They proposed the use of Taylor approximation as an efficient way to find parameters that are of less importance. A comprehensive comparison between their approach and an oracle (full combinatorial search over all parameters that results in minimum increase in loss) is done, proving its reliability. Here, we will give a brief introduction of their parameter pruning technique borrowed from their paper. The importance of a parameter is quantified by the error induced when it is removed from the network:

$$\mathcal{I}_m = (E(\mathcal{D}, \boldsymbol{W}) - E(\mathcal{D}, \boldsymbol{W} | w_m = 0))^2. \tag{1}$$

Here, $\boldsymbol{W} = \{w_0, w_1, ..., w_M\}$ are the set of parameters of a neural network supported by a dataset $\mathcal{D} = \{(x_0, y_0), (x_1, y_1), ..., (x_K, y_K)\}$ of K independent samples composed of inputs x_k and outputs y_k. (1) can be approximated by the second-order Taylor expansion as:

$$\mathcal{I}_m^{(2)}(\boldsymbol{W}) = (g_m w_m - \frac{1}{2} w_m \boldsymbol{H}_m \boldsymbol{W})^2. \tag{2}$$

$\boldsymbol{H}$ is the Hessian matrix where $\boldsymbol{H}_m$ is the m-th row of it and $g_m = \frac{\partial E}{\partial w_m}$. (2) can be further approximated using the first-order expansion:

$$\mathcal{I}_m^{(1)}(\boldsymbol{W}) = (g_m w_m)^2. \tag{3}$$

To minimize computational cost, (3) will be used since it is shown in [35] that the performance is on par with the second-order expansion and the first-order Taylor expansion is often used to estimate the importance of DNN components (weights, kernels or filters) [48, 36, 8]. Consistent with their work, a gate $\boldsymbol{z}_m$ is placed after batch normalization layers [25] where the importance approximation is defined as:

$$\mathcal{I}_m^{(1)}(\boldsymbol{z}) = (\frac{\partial E}{\partial \boldsymbol{z}_m})^2 = (\sum_{s \in \mathcal{S}_m} g_m w_m)^2, \tag{4}$$

where $\mathcal{S}_m$ corresponds to the set of weights $\boldsymbol{W}_{s \in \mathcal{S}_m}$ before the gate.

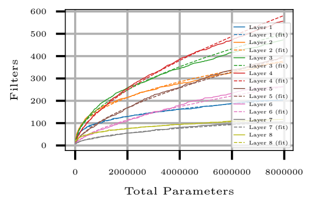
For a network composed of L layers, we define the the set of neurons (filters) for the entire network as $\{\phi_l\}_{l=1}^L$. ϕ_l is the number of neurons (filters) of layer l. We then define the total number of parameters in a network as τ. As we are using an iterative pruning method, on every pruning iteration, we will obtain a set of τ's and ϕ_l's for the l-th layer which can be represented as $\xi_l = \{\tau, \phi_l\}$. After pruning for N iterations, we obtain $\boldsymbol{\xi}_l = \{\{\tau^{(n)}, \phi_l^{(n)}\}_{n=1}^N\}$. In our implementation, we start feeding ξ_l's into $\boldsymbol{\xi}_l$ when all layers in a network is pruned by at least a single parameter. We conjecture that when all layers are pruned by at least a single parameter, most redundancy is removed and the residual parameters compose an efficient configuration. We stop pruning once the number of neurons (filters) is less than ϵ (we pick ϵ as 5% of the total neurons (filters) of the network in our implementation). Upon the completion of pruning, we have $\boldsymbol{\xi} \in \mathbb{R}^{2 \times N \times L}$. The entire pruning process begins once we have trained our network for P epochs (commonly known as network pre-training; we use the term *pre-training epochs* in our context instead) using a learning rate μ. The choice of P is studied and the conventional wisdom on the requirement of pre-training a network to convergence before pruning is investigated in the Supp. Section 6.3. The pruning algorithm is summarized in Algorithm 1.

3.2. Efficient Scaling of Parameters

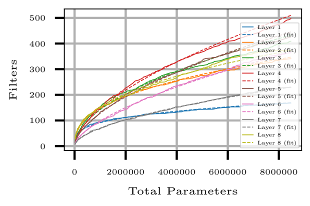
The goal of this work is to scale the neurons (filters) of a network across different layers to satisfy the targeted total parameter size denoted as $\hat{\tau}$. For parameter scaling to match the targeted size, uniform scaling is used in MorphNet [12] and MobileNets [23] where all layers are scaled with a constant width multiplier. It is intuitive that the scale applied to neurons (filters) of different layers should be layer-dependent to maximize performance. In this work, we propose an efficient method to scale the number of neurons (filters) across different layers to maximize performance.

Algorithm 1 Iterative Prune

```
1: procedure ITERATIVEPRUNE($f, \mathcal{D}$)
2:     for $P$ epochs do
3:         Update $f$ using learning rate $\mu$
4:     while $\sum_{l=1}^{L} |\boldsymbol{z}_l|_1 > \epsilon$ do
5:         Train $f$ for $Q$ iterations
6:         $\boldsymbol{\xi}' \leftarrow$ Prune $f$ using criteria (4)
7:         if all layers pruned at least once then
8:             $\boldsymbol{\xi} \leftarrow \{\boldsymbol{\xi}, \boldsymbol{\xi}'\}$        ▷ Record parameters
9:     return $\boldsymbol{\xi}$
```



(a) CIFAR10.



(b) CIFAR100.

Figure 2: Growth in number of filters of different layers across various network sizes. Each color represents independent layers of the convolutional filters of VGG11. Solid line is the residual filters obtained using an iterative pruning method and dashed line represents our approach on curve fitting. It can be observed that there's a pattern in the change of filters w.r.t. parameters which is dataset dependent.

By using iterative pruning as a proxy, we parameterize the change of neurons (filters) across different layers with respect to the total parameters or simply put as $\frac{\partial \phi_l}{\partial \tau}$.

Modeling Parameter Growth. As discussed in the previous subsection, $\boldsymbol{\xi}_l$ collected for each layer resembles the efficient set of neurons (filters) for each layer at a given size constraint. We can use this as a proxy to model $\frac{\partial \phi_l}{\partial \tau}$. We first observe how the distribution of the residual neurons (filters) obtained using pruning scale across different τ's, e.g. we use VGG11 as our network and CIFAR10/100 as our dataset to show the parameter growth across various sizes in Figure 2. We can observe that parameters across different layers don't scale linearly across different sizes, implying that uniform scaling is not an efficient scaling method. Figure 2 also shows us that the growth of the parameters resembles a power function that is monotonic. To fit the curves, we use:

$$\phi_l(\tau|\alpha_l, \beta_l) = \alpha_l \tau^{\beta_l}, \tag{5}$$

where every layer is paramterized independently by α_l and β_l. To obtain these parameters, we can linearize the problem by taking $\ln$ on both sides of (5), giving us:

$$\ln \phi_l(\tau|\alpha_l, \beta_l) = \ln \alpha_l + \beta_l \ln \tau. \tag{6}$$

Since we pruned the network iteratively for N iterations, we obtain a set of linear equations which can be formulated into a matrix of the form:

$$\underbrace{\begin{bmatrix} 1 & \ln \tau^{(1)} \\ 1 & \ln \tau^{(2)} \\ \vdots & \vdots \\ 1 & \ln \tau^{(N)} \end{bmatrix}}_{\boldsymbol{\mathcal{T}}} \underbrace{\begin{bmatrix} \ln \alpha_1 & \ln \alpha_2 & \dots & \ln \alpha_L \\ \beta_1 & \beta_2 & \dots & \beta_L \end{bmatrix}}_{\boldsymbol{\Theta}} = \tag{7}$$

$$\underbrace{\begin{bmatrix} \ln \phi_1^{(1)} & \ln \phi_2^{(1)} & \dots & \ln \phi_L^{(1)} \\ \ln \phi_1^{(2)} & \ddots & & \\ \vdots & & & \\ \ln \phi_1^{(N)} & & & \ln \phi_L^{(N)} \end{bmatrix}}_{\boldsymbol{\Phi}}.$$

We can solve for $\boldsymbol{\Theta}$ in (7) using the least-squares approach or by taking the pseudoinverse of $\boldsymbol{\mathcal{T}}$ and multiply it with $\boldsymbol{\Phi}$:

$$\boldsymbol{\Theta} = (\boldsymbol{\mathcal{T}}^T \boldsymbol{\mathcal{T}})^{-1} \boldsymbol{\mathcal{T}}^T \boldsymbol{\Phi}. \tag{8}$$

By using this method, we are able to fit the curves or $\boldsymbol{\xi}_l$'s obtained using iterative pruning of a network where the fitted results are shown in Figure 2. Our approach is a cost effective way of neural architecture parametrization and only takes two parameters ($\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$) per layer to parameterize the non-linear growth of the neuron (filter) count across various parameter sizes or $\frac{\partial \phi_l}{\partial \tau}$. The simplicity of our approach also prevents the overfitting of noise embedded in the samples obtained via pruning. The search of parameters for efficient scaling is summarized in Algorithm 2.

Algorithm 2 Search Parameters

```
1: procedure SEARCHPARAMS($\boldsymbol{\xi}$)
2:     $\boldsymbol{\mathcal{T}}, \boldsymbol{\Phi} \leftarrow \boldsymbol{\xi}$        ▷ Convert to matrix form
3:     $\boldsymbol{\Theta} \leftarrow (\boldsymbol{\mathcal{T}}^T \boldsymbol{\mathcal{T}})^{-1} \boldsymbol{\mathcal{T}}^T \boldsymbol{\Phi}$
4:     return $\boldsymbol{\Theta}$
```

Meeting Parameter Constraints. Since our approach fully parameterizes the independent scaling of network width across various sizes, we can meet tight parameter constraints during deployment of a DNN to devices with limited resource budget. For approaches like uniform scaling [12, 33, 23], only the number of output neurons (filters) can be scaled while the network size is a function of the input and output neurons (filters), hence meeting parameter constraints can only be done by performing an iterative grid search on the number of output neurons (filters) required.

For our approach, parameter scaling is intuitive as we can apply gradient descent on $\boldsymbol{\Phi}$ w.r.t. τ. To do so, we define a DNN as $f(\boldsymbol{x}|\boldsymbol{W}, \boldsymbol{\Phi}(\boldsymbol{\tau}|\boldsymbol{\Theta}))$ where f is a DNN architecture, $\boldsymbol{x}$ is its input, $\boldsymbol{W}$ are the weights of the DNN, and the additional condition $\boldsymbol{\Phi}(\boldsymbol{\tau}|\boldsymbol{\Theta})$ is introduced to parameterize the number of neurons (filters) required for each layer of a DNN. We then define a function h that computes the number of parameters of a DNN. h is architecture dependent.

Given a parameter constraint $\hat{\tau}$ that needs to be met, we can generate an architecture having total parameters close to $\hat{\tau}$ by performing stochastic gradient descent (SGD) on $\boldsymbol{\Phi}(\tau, \boldsymbol{\Theta})$ w.r.t. τ. Like other gradient descent problems, parameter initialization is important and the best way to do so is to fit $\hat{\tau}$ into (5) giving us:

$$\phi_l(\hat{\tau}|\alpha_l, \beta_l) = \alpha_l \hat{\tau}^{\beta_l}. \tag{9}$$

This gives us a good initial point, however there will still be a discrepancy between $h(f(\boldsymbol{x}|\boldsymbol{W}, \boldsymbol{\Phi}(\hat{\boldsymbol{\tau}}|\boldsymbol{\Theta})))$ and $\hat{\tau}$ which can be fixed by applying SGD on $\frac{1}{2}(\phi_l(\tau|\alpha_l, \beta_l) - \hat{\tau})^2$ w.r.t. to τ where the update of τ is given as:

$$\tau_i = \tau_{i-1} - \Delta\tau_{i-1} \tag{10}$$

$$= \tau_{i-1} - \eta(h(f(\boldsymbol{x}|\boldsymbol{W}, \boldsymbol{\Phi}(\tau_{i-1}|\boldsymbol{\Theta}))) - \hat{\tau}) \sum_{l=1}^{L} \beta_l \alpha_l \tau_{i-1}^{\beta_l - 1}. \tag{11}$$

Here, the subscript of τ corresponds to the SGD iteration and the full proof of (11) is given in Supp. Section 6.1. Since the number of parameters of an architecture is a monotonic function of Φ, this problem is convex and converges easily. If the learning rate η is set carefully, we are able to obtain an architecture with total parameters close to $\hat{\tau}$. We summarize this procedure in Algorithm 3.

Algorithm 3 Generate Network Using Searched Parameters

```
1: procedure GENERATENET($\boldsymbol{\Theta}, f, \hat{\tau}$)
2:     $\tau \leftarrow \hat{\tau}$                              ▷ Initialize parameter
3:     while not converged do
4:         $\tau \leftarrow \tau - \Delta\tau$     ▷ Update using SGD as in (11)
5:     for $l \leftarrow 1$ to $L$ do ▷ Layer-wise architecture update
6:         $f_l \leftarrow$ set output neurons (filters) as $\phi_l(\tau|\boldsymbol{\Theta}_l)$ (5)
7:     return $f$
```

3.3. Architecture Descent for Model Refinement

Like any gradient descent algorithm, initialization plays an important role and affects the convergence of an algorithm. Our approach is similar in way where we attempt to search for the configuration of an architecture given an initial configuration, e.g. VGGNet [44] and ResNet [18] consist of a set of predefined filter numbers for different configurations. As our approach behaves similarly to gradient descent, we coin it *architecture descent* as there is no gradient involved and it is descending in the loss surface by making iterative changes to the architecture's configuration.

We define an iteration of architecture descent as a single run of Algorithm 1, 2 and 3 that corresponds to iterative pruning, parameter searching and network generation. Upon the completion of iterative pruning and parameter searching, we obtain a set of parameters that scales our network in a more efficient manner. We can then use this set of parameters to scale-up our network as shown in Algorithm 3 for further pruning. We then proceed with several iterations of architecture descent until the changes in the architecture configuration is minuscule, indicating convergence. By applying architecture descent, we are descending on the loss surface that is parameterized by $\boldsymbol{\Theta}$ instead of a loss surface parameterized by its weights $\boldsymbol{W}$ performed in gradient descent. Architecture descent is summarized in Algorithm 4. NeuralScale is a composition of all algorithms we proposed as illustrated in Figure 1a.

Algorithm 4 Architecture Descent

```
1: procedure ARCHITECTUREDESCENT($f, \mathcal{D}, \tau$)
2:     while not converged do
3:         $\boldsymbol{\xi} \leftarrow$ ITERATIVEPRUNE($f, \mathcal{D}$)▷ Taylor FO [35]
4:         $\boldsymbol{\Theta} \leftarrow$ SEARCHPARAMS($\boldsymbol{\xi}$)
5:         $f \leftarrow$ GENERATENET($\boldsymbol{\Theta}, f, \tau$)
6:     return $f$              ▷ Network with scaled parameters
```

4. Experiments

In this section, we show experiments illustrating the importance of architecture descent. We then proceed with the benchmarking of our approach using public datasets, e.g. CIFAR10/100 [26] and TinyImageNet (subset of ImageNet [6] with images downsampled to 64×64 and consists of 200 classes having 500 training and 50 validation samples for each class) on commonly used architectures, e.g. VGG11 [44], MobileNetV2 [43] and ResNet18 [18]. All experiments are run on a single GTX1080Ti GPU using PyTorch [40] as our deep learning framework. We use SGD as our optimizer with an initial learning rate of 0.1, momentum set to 0.5 and a weight decay factor of 5^{-4}. Training of network that uses CIFAR10 and CIFAR100 are run for 300 epochs using a step decay of learning rate by at factor of 10 at epochs 100, 200 and 250 whereas network trained using TinyImageNet are run for 150 epochs with a decay in learning rate by a factor of 10 at epochs 50 and 100. For iterative pruning, we first train our network for $P = 10$ epochs using a learning rate of 0.1 and is decayed by a factor of 10 every 10 epochs. Source code is made available at `https://github.com/eugenelet/NeuralScale`.

4.1. Importance of Architecture Descent

For all experiments, we run architecture descent for 15 iterations. We show configurations with total parameters matching total parameters of network with its default set of filters uniformly scaled to a ratio, r. $r = 0.25, 2$ for CIFAR10 and CIFAR100. $r = 0.25, 1$ for TinyImageNet.

VGG11. Using a relatively shallow network, we demonstrate the application of architecture descent using CIFAR10 and CIFAR100 as shown in Figure 3. By observing the resulting architecture configuration, we can make two conjectures. First, we show that conventional wisdom on network design that gradually increases the number of filters as we go deeper in a convolutional network does not guarantee optimal performance. It can be observed that the conventional wisdom on network design holds up to some level (layer 4) and bottlenecking of parameters can be observed up to the penultimate layer which is followed by a final layer which is comparatively larger. Second, the scaling of network should not be done linearly as was done in [23] and should follow a non-linear rule that we attempt to approximate using a power function. If we look closely, by applying architecture descent on datasets of higher complexity generates network with configuration that has more filters allocated toward the end. Our conjecture is that more resources are needed to capture the higher level features when the task is more difficult whereas for simple classification problem like CIFAR10, more resource is allocated to earlier layers to generate more useful features. These observations give us a better understanding on how resource should be allocated in DNNs and can be used as a guideline for deep learning practitioners in designing DNNs. A single iteration of architecture descent for VGG11 on CIFAR10/100 is approximately 20 minutes.

MobileNetV2. We show the application of architecture descent on a more sophisticated architecture known as MobileNetV2 using CIFAR100 and TinyImageNet in Figure 3. Here, we only apply our search algorithm on deciding the size of the bottleneck layers while the size of the expansion layer follows the same expansion rule found in [43] where an expansion factor of $6\times$ is used. The resulting configuration closely resembles the one found using a feedforward network like VGG. It can also be observed that resources are allocated toward the output for a more sophisticated dataset. A single iteration of architecture descent for MobileNetV2 on CIFAR100 and TinyImageNet is approximately 50 minutes and 1.2 hour respectively.

ResNet18. The application of architecture descent on ResNet18 using CIFAR100 and TinyImageNet is shown in Figure 3. We observe a different pattern of architecture configuration when compared to a simple feed forward network like VGG. This is an interesting observation as it agrees with the interpretation of residual networks as an ensemble of many paths of different lengths shown in [49]. Another observation is that if we look only at a single layer of every residual block (each block consists of two layers), the searched configuration for ResNet follows the pattern found in VGG where there's a smooth gradient of filter progression across layers. A single iteration of architecture descent for ResNet18 on CIFAR100 and TinyImageNet is approximately 50 minutes and 45 minutes respectively.

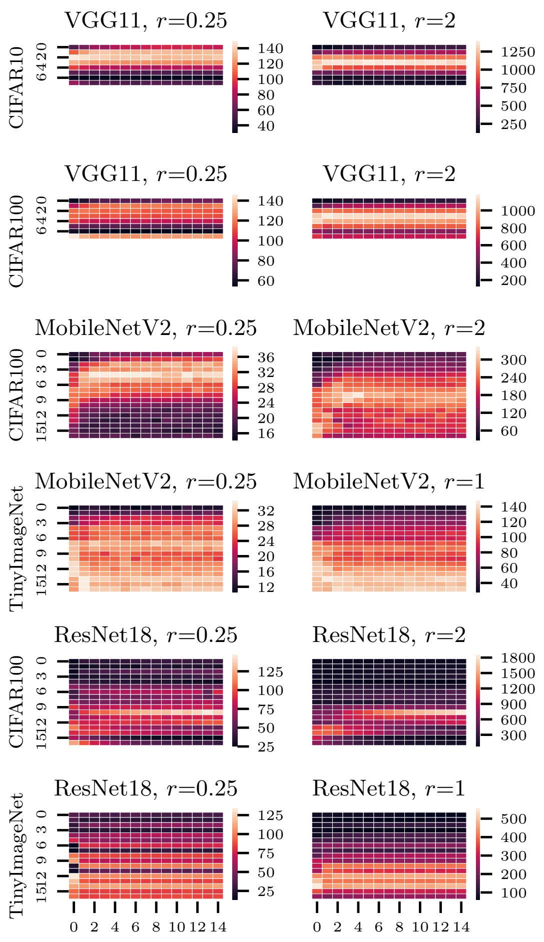
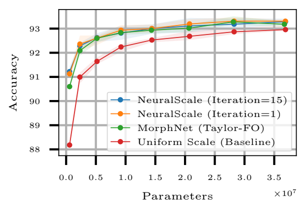


Figure 3: Shows the number of filters for each layer by running architecture descent for 15 iterations on various architecture-dataset pair. Vertical and horizontal axis of each plot corresponds to the filter number and architecture descent iteration respectively. r is the uniform scaling ratio applied to the default configuration. Best viewed in color.

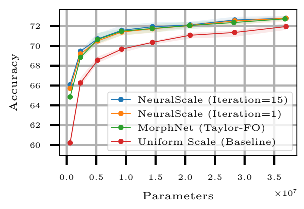
4.2. Benchmarking of NeuralScale

Here, we compare the accuracy of NeuralScale with the first (Iteration=1, to compare with MorphNet [12]) and last (Iteration=15) iteration of architecture descent with a uniform scaling (baseline) method and a method where a network is first pruned using Taylor-FO [35] until it has 50%

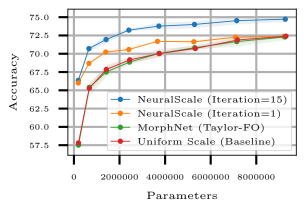
of filters left and then scaled uniformly (resembling the first iteration of MorphNet [12] and the use of [35] is to match our pruning method for a fair comparison), named as *MorphNet (Taylor-FO)* in our comparison tables and plots. The accuracy is obtained by averaging across the maximum test accuracy of 5 independent simulations. For the accuracy plots in Figure 4 and 5, the output filters of the original network are scaled to the ratios from 0.25 to 2 with an increment of 0.25 for CIFAR10/100 along with 0.25, 0.5, 0.75 and 1 for TinyImageNet. For the comparison tables, only the ratios 0.25, 0.75, 2 for CIFAR10/100 and 0.25, 0.75 for TinyImageNet are reported. Comparisons are also made with a structured magnitude pruning method [29] where we first pre-train our network using the same prescription for other methods and proceed with 40 and 20 epochs of fine-tuning for CIFAR10/100 and TinyImageNet respectively, using a learning rate of 0.001. We only show results for VGG11-CIFAR10, MobileNetV2-CIFAR100 and ResNet18-TinyImageNet in Table 1 and the rest are deferred to Supp. Section 6.2. Note that all methods are trained from scratch and only [29] is trained using the pretrain-prune-finetuning pipeline. Results show that the hypothesis in [33] holds (training from scratch performs better). As our approach is designed for platforms with structured parallel computing capability like GPUs, we report the latency of different methods instead of FLOPs. Note that our approach isn't optimized for latency. Here, latency is defined as the time required for an input to propagate to the output. All latencies reported in Table 1 are based on a batch size of 100 where 20 batches are first fed for warm-up of cache and is proceeded with 80 batches which are averaged to give the final latencies. As a comparison of latency based solely on the scale of parameter in Table 1 is unintuitive, we show a plot comparing accuracy of different methods against latency in Figure 5.



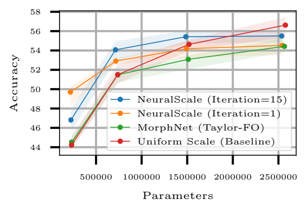
(a) VGG11 on CIFAR10.



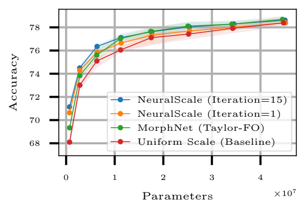
(b) VGG11 on CIFAR100.



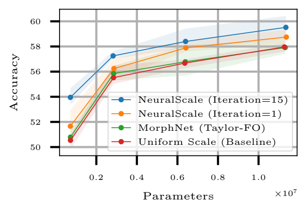
(c) MobileNetV2 on CIFAR100.



(d) MobileNetV2 on TinyImageNet.



(e) ResNet18 on CIFAR100.

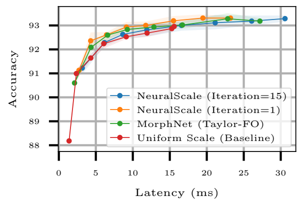


(f) ResNet18 on TinyImageNet.

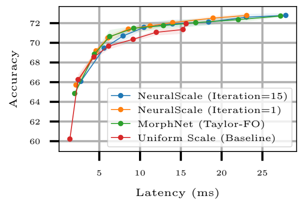
Figure 4: All plots are organized such that each row corresponds to a single architecture, e.g. (a),(b) corresponds to VGG11, (c),(d) corresponds to MobileNetV2 and (e),(f) corresponds to ResNet18. Each plot consists of the accuracy comparison of different scaling methods (applied on the width), plotted against different parameter scales. The shaded region of each line corresponds to the maximum and minimum accuracy across 5 independent simulations.

VGG11. By observing the comparison plot shown in Figure 4a and 4b, our approach compares favourably in terms of parameter efficiency for CIFAR10 and CIFAR100. As shown in Table 1, at the lowest parameter count, an accuracy gain of 3.04% is obtained for CIFAR10. Efficiency in latency of our approach is also comparable with the baseline approach as shown in Figure 5a and 5b. Diminishing returns are observed when the network increases in size. We conjecture that as the network grows larger, more subspaces are covered, hence the network can still adapt to the sub-optimal configuration by suppressing redundant parameters. Another observation is that the performance gain is more substantial on a more complicated dataset which is intuitive as inductive bias is introduced in an architectural level.

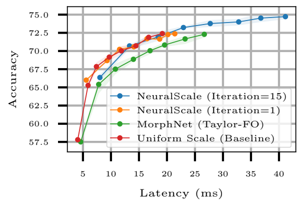
MobileNetV2. The application to MobileNetV2 is to show the extensibility of our approach to a delicately handcrafted architecture. Our approach is superior in parameter efficiency (most cases) when compared to other methods as shown in Figure 4c and 4d. As shown in Table 1, an accuracy gain of 8.56% for CIFAR100 relative to baseline is observed at a scaling ratio of 0.25. Our approach is also efficient in latency as shown in Figure 5c. An unintuitive observation can be seen on the experiment for TinyImageNet where the accuracy at iteration 1 outperforms iteration 15 for NeuralScale at ratio 0.25. The accuracy is below the baseline when more parameters are used. As the default configuration of MobileNetV2 has consistent filters, the default shortcut connections are identity mappings. We hypothesize that the switch from identity mapping to a convolutional mapping for the shortcut connection is the culprit. Empirical study is done in Supp. Section 6.4 with results supporting our hypothesis and explaining the observations.



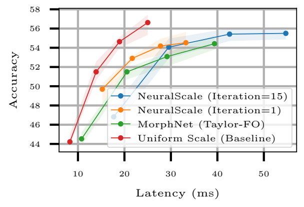
(a) VGG11 on CIFAR10.



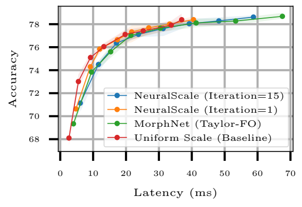
(b) VGG11 on CIFAR100.



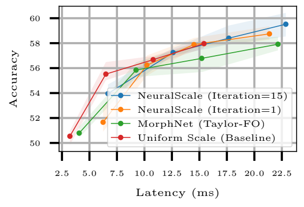
(c) MobileNetV2 on CIFAR100.



(d) MobileNetV2 on TinyImageNet.



(e) ResNet18 on CIFAR100.



(f) ResNet18 on TinyImageNet.

Figure 5: The layout of these plots are structured as shown in Figure 4. The only difference is that the accuracies of different methods are plotted against latencies.

Table 1: Comparison of various network-dataset pairs.

Method	Params	Latency	Accuracy (%)
VGG11 CIFAR10			
Uniform Scale (Baseline)	0.58M	1.29ms	88.18 ± 0.16
	5.20M	4.31ms	91.64 ± 0.10
	36.89M	18.86ms	92.96 ± 0.09
Li *et al.* [29]†	5.20M	4.75ms	91.12 ± 0.02
MorphNet [12] (Taylor-FO [35])	0.58M	2.07ms	90.60 ± 0.11
	5.22M	8.43ms	92.60 ± 0.09
	36.72M	48.52ms	93.18 ± 0.11
NeuralScale (Iteration = 1)	0.58M	2.56ms	91.13 ± 0.07
	5.20M	8.89ms	92.61 ± 0.15
	36.88M	37.39ms	$\mathbf{93.31 \pm 0.05}$
NeuralScale (Iteration = 15)	0.58M	2.94ms	$\mathbf{91.22 \pm 0.15}$
	5.20M	12.52ms	$\mathbf{92.63 \pm 0.12}$
	36.90M	53.35ms	93.29 ± 0.09
MobileNetV2 CIFAR100			
Uniform Scale (Baseline)	0.20M	5.37ms	57.80 ± 0.31
	1.42M	7.46ms	67.85 ± 0.38
	9.30M	19.69ms	72.40 ± 0.22
Li *et al.* [29]†	1.42M	7.71ms	67.12 ± 0.08
MorphNet [12] (Taylor-FO [35])	0.20M	6.14ms	57.51 ± 0.36
	1.42M	10.95ms	67.51 ± 0.48
	9.30M	26.53ms	72.29 ± 0.28
NeuralScale (Iteration = 1)	0.19M	5.69ms	66.00 ± 0.12
	1.40M	11.73ms	70.23 ± 0.25
	9.21M	21.32ms	72.37 ± 0.12
NeuralScale (Iteration = 15)	0.19M	7.84ms	$\mathbf{66.36 \pm 0.28}$
	1.41M	17.89ms	$\mathbf{71.94 \pm 0.45}$
	9.27M	40.48ms	$\mathbf{74.73 \pm 0.26}$
ResNet18 TinyImageNet			
Uniform Scale (Baseline)	0.73M	3.02ms	50.54 ± 0.37
	6.36M	11.56ms	56.68 ± 0.28
Li *et al.* [29]†	6.36M	11.93ms	54.72 ± 0.24
MorphNet [12] (Taylor-FO [35])	0.72M	3.80ms	50.79 ± 0.38
	6.39M	14.83ms	56.78 ± 0.85
NeuralScale (Iteration = 1)	0.72M	5.96ms	51.66 ± 0.80
	6.42M	14.58ms	57.89 ± 0.28
NeuralScale (Iteration = 15)	0.72M	6.42ms	$\mathbf{53.95 \pm 0.53}$
	6.40M	17.52ms	$\mathbf{58.40 \pm 0.54}$

† Fine-tuned using pre-trained network (not trained from scratch).

ResNet18. As shown from the accuracy comparison plot in Figure 4e and 4f, substantial accuracy gain under different parameter scales is observed. From Table 1, an accuracy gain of 3.41% is observed for TinyImageNet at a scale of 0.25. Accuracy gain using architecture descent is also more pronounced here. The accuracy gain here is in contradictory with the results in [33] (no gain in accuracy observed) probably due to the better pruning technique we use [35]. In Figure 5e and 5f our approach is comparable and in most cases better than the baseline configuration in latency.

5. Conclusion

In this work, we propose a method to efficiently scale the neuron (filter) number of an architecture. We hypothesize that the scaling of network should follow a non-linear rule and is shown empirically that through our approach, networks efficient in parameters across different scales can be generated using this rule. Our empirical results on architecture descent also shed light on the efficient allocation of parameters in a deep neural network. As our approach is computationally-efficient and is complementary to most techniques and architectures, the inclusion to existing deep learning framework is cost-effective and results in a model of higher accuracy under the same parameter constraint.

References

[1] Irwan Bello, Barret Zoph, Vijay Vasudevan, and Quoc V Le. Neural optimizer search with reinforcement learning. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 459–468. JMLR. org, 2017.

[2] Gabriel Bender, Pieter-Jan Kindermans, Barret Zoph, Vijay Vasudevan, and Quoc Le. Understanding and simplifying one-shot architecture search. In *International Conference on Machine Learning*, pages 549–558, 2018.

[3] Han Cai, Ligeng Zhu, and Song Han. Proxylessnas: Direct neural architecture search on target task and hardware. *arXiv preprint arXiv:1812.00332*, 2018.

[4] Michael Cogswell, Faruk Ahmed, Ross Girshick, Larry Zitnick, and Dhruv Batra. Reducing overfitting in deep networks by decorrelating representations. *arXiv preprint arXiv:1511.06068*, 2015.

[5] Nadav Cohen and Amnon Shashua. Inductive bias of deep convolutional networks through pooling geometry. *arXiv preprint arXiv:1605.06743*, 2016.

[6] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[7] Xiaohan Ding, Guiguang Ding, Yuchen Guo, Jungong Han, and Chenggang Yan. Approximated oracle filter pruning for destructive cnn width optimization. *arXiv preprint arXiv:1905.04748*, 2019.

[8] Xiaohan Ding, Xiangxin Zhou, Yuchen Guo, Jungong Han, Ji Liu, et al. Global sparse momentum sgd for pruning very deep neural networks. In *Advances in Neural Information Processing Systems*, pages 6379–6391, 2019.

[9] Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. *arXiv preprint arXiv:1803.03635*, 2018.

[10] Jonathan Frankle, Gintare Karolina Dziugaite, Daniel M Roy, and Michael Carbin. The lottery ticket hypothesis at scale. *arXiv preprint arXiv:1903.01611*, 2019.

[11] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256, 2010.

[12] Ariel Gordon, Elad Eban, Ofir Nachum, Bo Chen, Hao Wu, Tien-Ju Yang, and Edward Choi. Morphnet: Fast & simple resource-constrained structure learning of deep networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1586–1595, 2018.

[13] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[14] Song Han, Jeff Pool, Sharan Narang, Huizi Mao, Shijian Tang, Erich Elsen, Bryan Catanzaro, John Tran, and William J Dally. Dsd: regularizing deep neural networks with dense-sparse-dense training flow. *arXiv preprint arXiv:1607.04381*, 3(6), 2016.

[15] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. In *Advances in neural information processing systems*, pages 1135–1143, 2015.

[16] Babak Hassibi and David G Stork. Second order derivatives for network pruning: Optimal brain surgeon. In *Advances in neural information processing systems*, pages 164–171, 1993.

[17] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

[18] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[19] Yihui He, Ji Lin, Zhijian Liu, Hanrui Wang, Li-Jia Li, and Song Han. Amc: Automl for model compression and acceleration on mobile devices. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 784–800, 2018.

[20] Yang He, Ping Liu, Ziwei Wang, Zhilan Hu, and Yi Yang. Filter pruning via geometric median for deep convolutional neural networks acceleration. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4340–4349, 2019.

[21] Suzana Herculano-Houzel. The human brain in numbers: a linearly scaled-up primate brain. *Frontiers in human neuroscience*, 3:31, 2009.

[22] Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, et al. Searching for mobilenetv3. *arXiv preprint arXiv:1905.02244*, 2019.

[23] 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 preprint arXiv:1704.04861*, 2017.

[24] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4700–4708, 2017.

[25] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[26] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. Technical report, Citeseer, 2009.

[27] Yann LeCun, John S Denker, and Sara A Solla. Optimal brain damage. In *Advances in neural information processing systems*, pages 598–605, 1990.

[28] Namhoon Lee, Thalaiyasingam Ajanthan, and Philip HS Torr. Snip: Single-shot network pruning based on connection sensitivity. *arXiv preprint arXiv:1810.02340*, 2018.

[29] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning filters for efficient convnets. *arXiv preprint arXiv:1608.08710*, 2016.

[30] 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 *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 19–34, 2018.

[31] Hanxiao Liu, Karen Simonyan, and Yiming Yang. Darts: Differentiable architecture search. *arXiv preprint arXiv:1806.09055*, 2018.

[32] Zhuang Liu, Jianguo Li, Zhiqiang Shen, Gao Huang, Shoumeng Yan, and Changshui Zhang. Learning efficient convolutional networks through network slimming. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2736–2744, 2017.

[33] Zhuang Liu, Mingjie Sun, Tinghui Zhou, Gao Huang, and Trevor Darrell. Rethinking the value of network pruning. *arXiv preprint arXiv:1810.05270*, 2018.

[34] Jian-Hao Luo, Jianxin Wu, and Weiyao Lin. Thinet: A filter level pruning method for deep neural network compression. In *Proceedings of the IEEE international conference on computer vision*, pages 5058–5066, 2017.

[35] Pavlo Molchanov, Arun Mallya, Stephen Tyree, Iuri Frosio, and Jan Kautz. Importance estimation for neural network pruning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 11264–11272, 2019.

[36] Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, and Jan Kautz. Pruning convolutional neural networks for resource efficient inference. *arXiv preprint arXiv:1611.06440*, 2016.

[37] Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, and Jan Kautz. Pruning convolutional neural networks for resource efficient transfer learning. *arXiv preprint arXiv:1611.06440*, 3, 2016.

[38] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010.

[39] Kirill Neklyudov, Dmitry Molchanov, Arsenii Ashukha, and Dmitry P Vetrov. Structured bayesian pruning via log-normal multiplicative noise. In *Advances in Neural Information Processing Systems*, pages 6775–6784, 2017.

[40] 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 Autodiff Workshop*, 2017.

[41] Hieu Pham, Melody Y Guan, Barret Zoph, Quoc V Le, and Jeff Dean. Efficient neural architecture search via parameter sharing. *arXiv preprint arXiv:1802.03268*, 2018.

[42] Prajit Ramachandran, Barret Zoph, and Quoc V Le. Searching for activation functions. *arXiv preprint arXiv:1710.05941*, 2017.

[43] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4510–4520, 2018.

[44] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[45] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[46] 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 *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2820–2828, 2019.

[47] Mingxing Tan and Quoc V Le. Efficientnet: Rethinking model scaling for convolutional neural networks. *arXiv preprint arXiv:1905.11946*, 2019.

[48] Lucas Theis, Iryna Korshunova, Alykhan Tejani, and Ferenc Huszár. Faster gaze prediction with dense networks and fisher pruning. *arXiv preprint arXiv:1801.05787*, 2018.

[49] Andreas Veit, Michael J Wilber, and Serge Belongie. Residual networks behave like ensembles of relatively shallow networks. In *Advances in neural information processing systems*, pages 550–558, 2016.

[50] Sirui Xie, Hehui Zheng, Chunxiao Liu, and Liang Lin. Snas: stochastic neural architecture search. *arXiv preprint arXiv:1812.09926*, 2018.

[51] Tien-Ju Yang, Yu-Hsin Chen, and Vivienne Sze. Designing energy-efficient convolutional neural networks using energy-aware pruning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 5687–5695, 2017.

[52] Chenglong Zhao, Bingbing Ni, Jian Zhang, Qiwei Zhao, Wenjun Zhang, and Qi Tian. Variational convolutional neural network pruning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2780–2789, 2019.

[53] Hattie Zhou, Janice Lan, Rosanne Liu, and Jason Yosinski. Deconstructing lottery tickets: Zeros, signs, and the supermask. *arXiv preprint arXiv:1905.01067*, 2019.

[54] Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.

6. Supplementary Material

6.1. Update of τ

To meet the memory limitation (parameter constraint) of any platform, we only need to update τ in our approach. We show the proof of the update of τ as follows:

$$\tau_i = \tau_{i-1} - \Delta\tau_{i-1} \tag{12}$$

$$= \tau_{i-1} - \eta \frac{\partial \frac{1}{2}(h(f(\boldsymbol{x}|\boldsymbol{W}, \boldsymbol{\Phi}(\tau_{i-1}|\boldsymbol{\Theta}))) - \hat{\tau})^2}{\partial \tau_{i-1}} \tag{13}$$

$$= \tau_{i-1} - \eta(h(f(\boldsymbol{x}|\boldsymbol{W}, \boldsymbol{\Phi}(\tau_{i-1}|\boldsymbol{\Theta}))) - \hat{\tau}) \tag{14}$$

$$\cdot \frac{\partial h(f(\boldsymbol{x}|\boldsymbol{W}, \boldsymbol{\Phi}(\tau_{i-1}|\boldsymbol{\Theta})))}{\partial \boldsymbol{\Phi}(\tau_{i-1}|\boldsymbol{\Theta})} \frac{\partial \boldsymbol{\Phi}(\tau_{i-1}|\boldsymbol{\Theta})}{\partial \tau_{i-1}} \tag{15}$$

$$= \tau_{i-1} - \eta(h(f(\boldsymbol{x}|\boldsymbol{W}, \boldsymbol{\Phi}(\tau_{i-1}|\boldsymbol{\Theta}))) - \hat{\tau}) \sum_{l=1}^{L} \beta_l \alpha_l \tau_{i-1}^{\beta_l - 1}. \tag{16}$$

Note that (15) comes from the chain rule of derivative.

6.2. Accuracy Comparisons

In this section we show the tabularized comparison of VGG11-CIFAR100, MobileNetV2-CIFAR100 and ResNet18-TinyImageNet which was not shown in the main paper. As shown in Table 2, an accuracy gain of 5.85%, 2.40% and 3.04% is observed for VGG11, MobileNetV2 and ResNet18 on CIFAR100, TinyImageNet and CIFAR100 respectively.

6.3. Pre-Training Epochs P

The pruning of neural network is usually done on a pre-trained network. As we want our algorithm to be efficient in terms of search cost, we explore the possibility of reduction in time or epochs for network pre-training by tuning the pre-training epochs P. To our surprise, having a large P does not result in an architecture with the best performance. Here, we investigate how P affects the accuracy of the final configuration, proving that conventional wisdom on when to apply pruning might be flawed. Experiments will be shown on VGG11 and MobileNetV2 on CIFAR10 and CIFAR100 respectively. All results shown are based on the final (iteration=15) iteration of architecture descent.

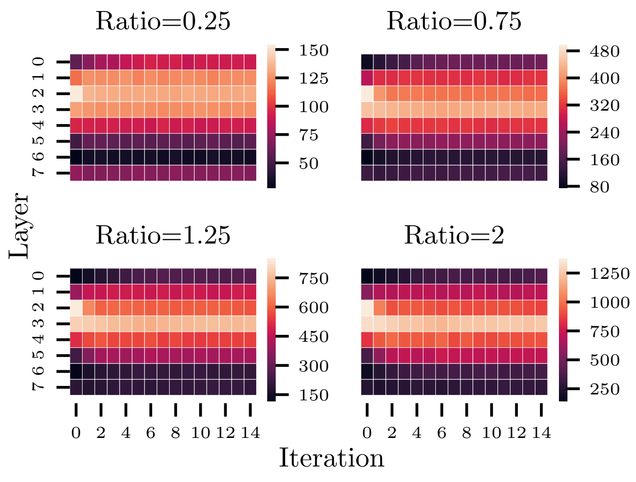
VGG11. CIFAR10 will be used for the experimentation on P for VGG11. We show architecture and results obtained by setting P to be 0, 2, 5, 10, 30 and 60. The searched architecture is shown in Figure 6. We next show the comparison plot using different pre-training epochs in Figure 7 accompanied by Table 3. For a simple network like VGG11, the number of pre-training epochs doesn't have too much of an impact in performance which can be clearly observed in the resulting filter configuration in Figure 6.

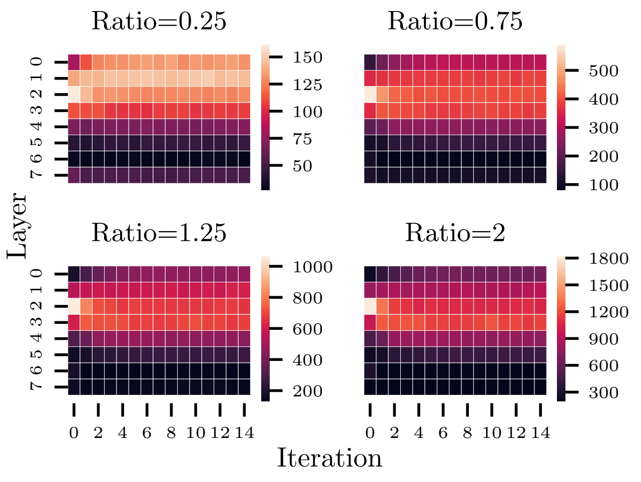
Table 2: Comparison of various network-dataset pairs.

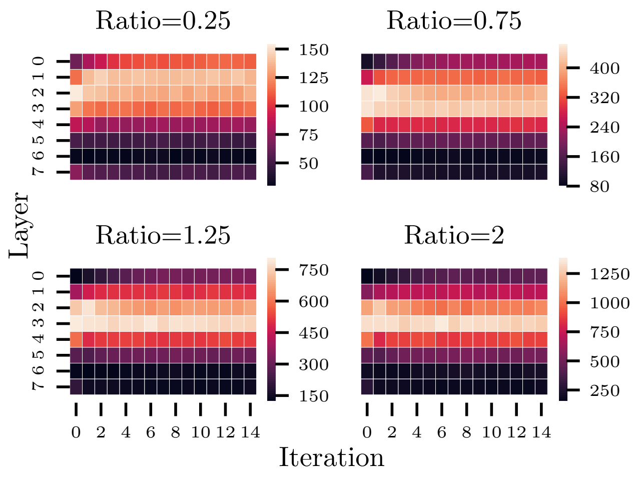
Method	Params	Latency	Accuracy (%)
VGG11 CIFAR100			
Uniform Scale (Baseline)	0.59M	1.30ms	60.22 ± 0.45
	5.23M	4.28ms	68.56 ± 0.21
	36.99M	18.83ms	71.94 ± 0.25
Li *et al.* [29]†	5.23M	4.77ms	68.41 ± 0.09
MorphNet [12] (Taylor-FO [35])	0.59M	1.78ms	64.85 ± 0.17
	5.21M	7.18ms	70.64 ± 0.38
	36.80M	41.52ms	72.72 ± 0.09
NeuralScale (Iteration = 1)	0.59M	1.95ms	65.71 ± 0.28
	5.23M	7.36ms	70.50 ± 0.16
	36.98M	33.24ms	$\mathbf{72.78 \pm 0.19}$
NeuralScale (Iteration = 15)	0.59M	2.52ms	$\mathbf{66.07 \pm 0.21}$
	5.23M	10.19ms	$\mathbf{70.70 \pm 0.45}$
	36.98M	43.95ms	$\mathbf{72.78 \pm 0.13}$
MobileNetV2 TinyImageNet			
Uniform Scale (Baseline)	0.23	8.53ms	44.22 ± 0.40
	1.52M	18.87ms	54.63 ± 0.46
Li *et al.* [29]†	1.52M	18.76ms	52.71 ± 0.28
MorphNet [12] (Taylor-FO [35])	0.23M	10.47ms	44.53 ± 0.50
	1.51M	28.88ms	53.08 ± 0.52
NeuralScale (Iteration = 1)	0.22M	14.96ms	$\mathbf{49.70 \pm 0.73}$
	1.49M	26.98ms	54.18 ± 0.57
NeuralScale (Iteration = 15)	0.22M	17.16ms	46.82 ± 0.89
	1.49M	41.20ms	$\mathbf{55.42 \pm 0.44}$
ResNet18 CIFAR100			
Uniform Scale (Baseline)	0.71M	2.53ms	68.10 ± 0.40
	6.32M	9.98ms	75.10 ± 0.34
	44.75M	47.04ms	78.39 ± 0.29
Li *et al.* [29]†	6.32M	10.18ms	73.91 ± 0.12
MorphNet [12] (Taylor-FO [35])	0.72M	3.73ms	69.34 ± 0.31
	6.29M	15.03ms	75.60 ± 0.40
	44.53M	98.54ms	$\mathbf{78.68 \pm 0.17}$
NeuralScale (Iteration = 1)	0.71M	4.51ms	70.63 ± 0.13
	6.38M	11.95ms	75.83 ± 0.15
	45.15M	50.24ms	78.39 ± 0.22
NeuralScale (Iteration = 15)	0.71M	5.71ms	$\mathbf{71.14 \pm 0.45}$
	6.36M	19.54ms	$\mathbf{76.35 \pm 0.20}$
	45.05M	90.18ms	78.62 ± 0.13

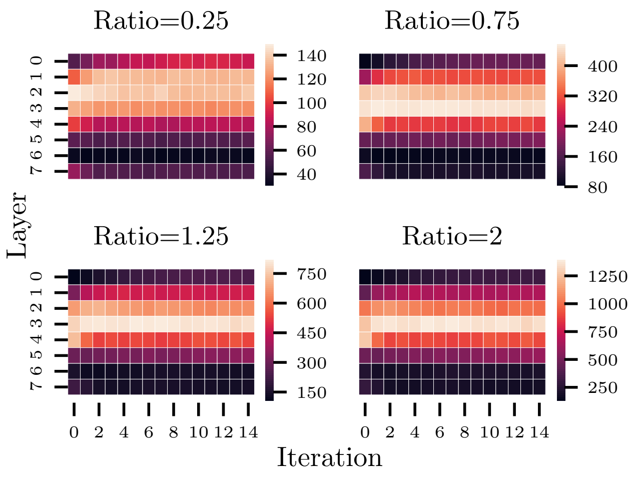
† Fine-tuned using pre-trained network (not trained from scratch).

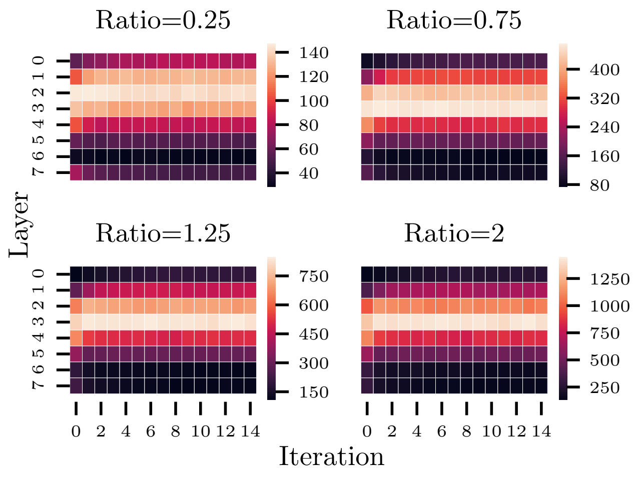
MobileNetV2. CIFAR100 will be used for the experimentation on P for MobileNetV2. We show architecture and results obtained by setting P to be 0, 2, 5, 10, 30 and


(a) $P = 0$


(b) $P = 2$


(c) $P = 5$


(d) $P = 10$


(e) $P = 30$

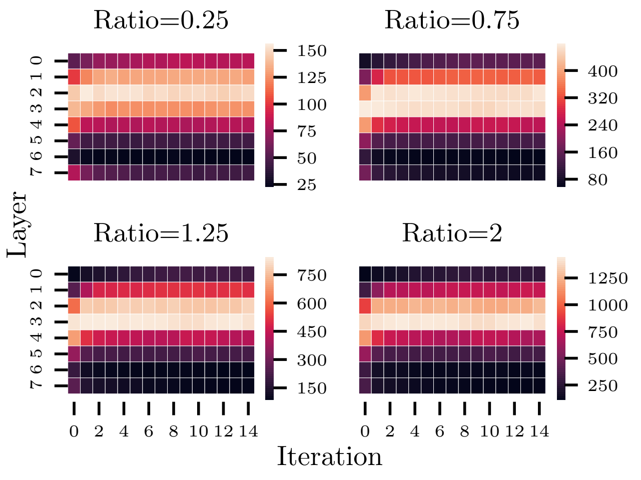
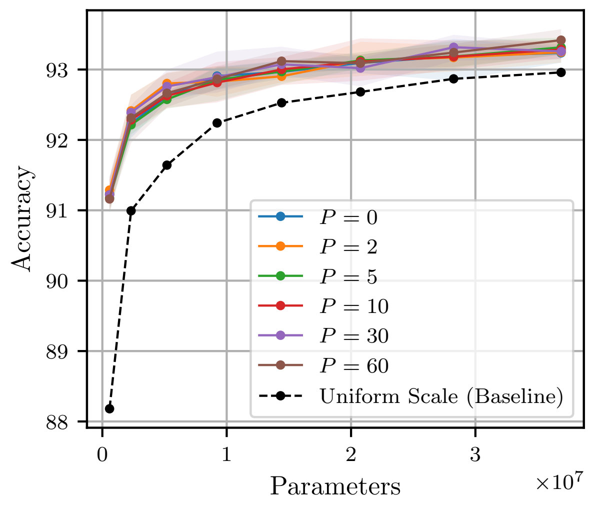
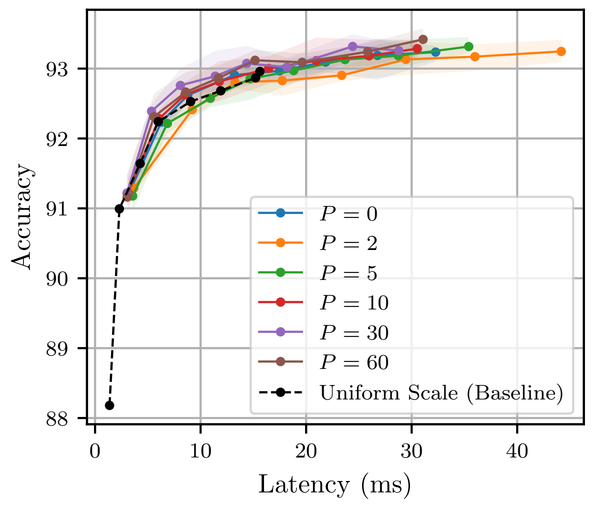

(f) $P = 60$

Figure 6: Showing the difference in searched architecture by running architecture descent on VGG11 for CIFAR10 using various value of pre-training epochs P.



(a) Accuracy vs Parameter.



(b) Accuracy vs Latency.

Figure 7: Accuracy comparison plot for VGG11 on CIFAR10 that uses different pre-training epochs P before pruning. (a) shows the accuracy comparison under different parameters using different value of P. (b) shows the comparison of accuracy under different latencies using different value of P.

60. The searched architecture is shown in Figure 8. We next show the comparison plot using different pre-training epochs in Figure 9 accompanied by Table 4. It is interesting to see that for a deeper and more complicated network like MobileNetV2, there's a notable variation in the distribution of filters with respect to the number of pre-training epochs. The accuracy comparison in Figure 9 shows that

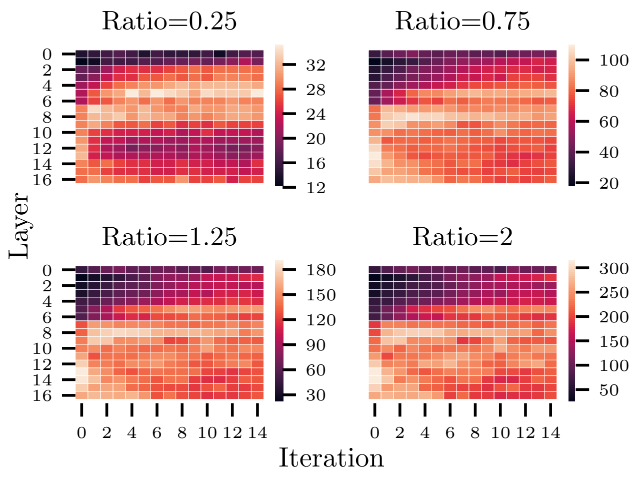
Table 3: Accuracy comparison on VGG11 for CIFAR10 using different pre-training epochs P.

Method	Params	Latency	Accuracy (%)
Uniform Scale (Baseline)	0.58M	1.30ms	88.18 ± 0.16
	5.20M	4.31ms	91.64 ± 0.10
	36.89M	19.50ms	92.96 ± 0.09
NeuralScale ($P = 0$)	0.58M	3.01ms	91.23 ± 0.05
	5.20M	12.35ms	92.62 ± 0.06
	36.89M	53.26ms	93.24 ± 0.09
NeuralScale ($P = 2$)	0.58M	3.49ms	$\mathbf{91.29 \pm 0.09}$
	5.20M	20.24ms	$\mathbf{92.80 \pm 0.09}$
	36.90M	81.27ms	93.25 ± 0.10
NeuralScale ($P = 5$)	0.58M	3.34ms	91.18 ± 0.13
	5.19M	17.30ms	92.58 ± 0.08
	36.90M	63.85ms	93.31 ± 0.08
NeuralScale ($P = 10$)	0.58M	2.93ms	91.22 ± 0.15
	5.20M	12.53ms	92.63 ± 0.12
	36.90M	55.44ms	93.29 ± 0.09
NeuralScale ($P = 30$)	0.58M	2.82ms	91.22 ± 0.15
	5.20M	11.85ms	92.76 ± 0.13
	36.90M	51.02ms	93.26 ± 0.08
NeuralScale ($P = 60$)	0.58M	2.85ms	91.16 ± 0.17
	5.20M	12.58ms	92.66 ± 0.18
	36.89M	61.14ms	$\mathbf{93.42 \pm 0.13}$

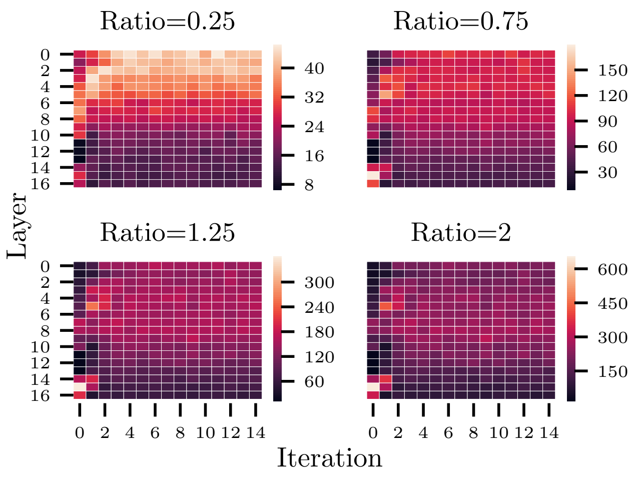
having large number of pre-training epochs doesn't help the efficiency in parameters and instead impedes it. It is shown that $P = 2$ or $P = 5$ gives us a configuration of filters that is the most efficient in terms of parameters for MobileNetV2 on CIFAR100. This is an interesting observation which sheds light on the number of pre-training iterations required prior to network pruning for optimal performance.

6.4. Using Convolutional Layers as Shortcut Connection

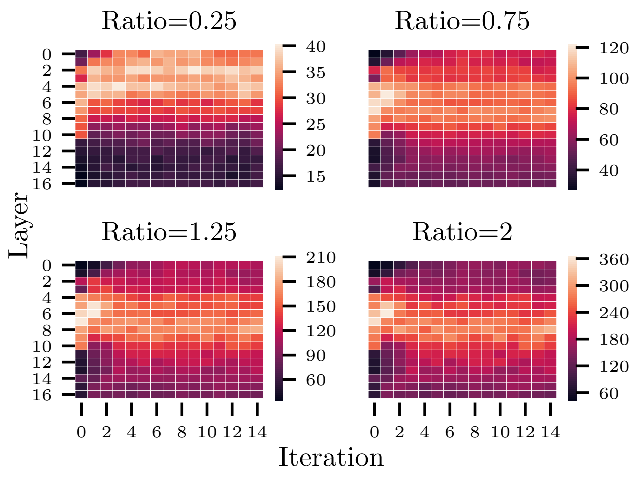
By default, MobileNetV2 has shortcut connections composed of identity mappings. By modifying the filter sizes of MobileNetV2, the shortcut connection has to be changed to a convolutional one instead to compensate the difference in filter sizes on both ends of the shortcut connection. A surprising finding is that the change from identity mapping to convolutional mapping affects the original performance significantly, despite the increase in parameter. We show experiments comparing two kinds of shortcut connection (identity and convolutional) on the original configuration which is uniformly scaled to different ratios. We name the method that uses convolutional shortcuts as *ConvCut*. A comparison plot comparing ConvCut with other scaling



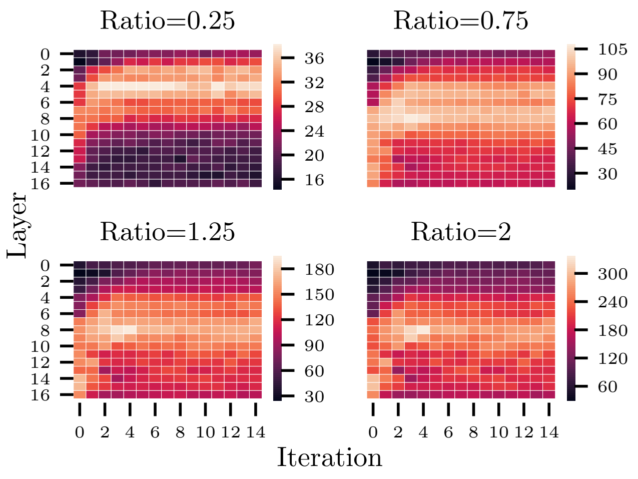
(a) $P = 0$



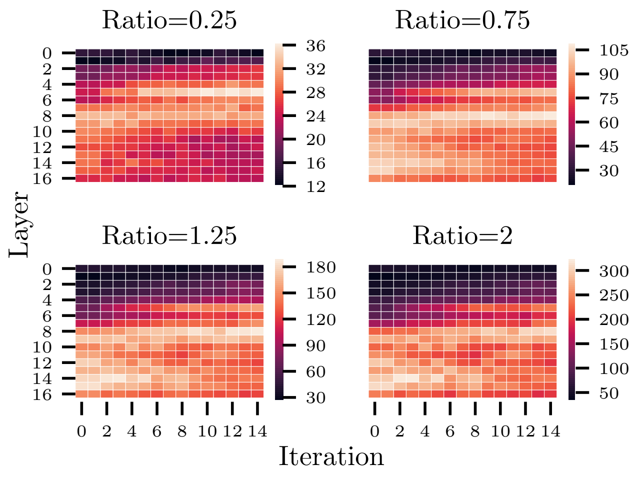
(b) $P = 2$



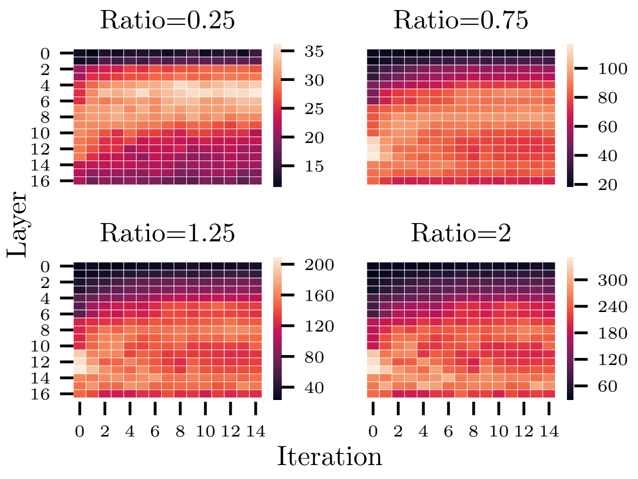
(c) $P = 5$



(d) $P = 10$

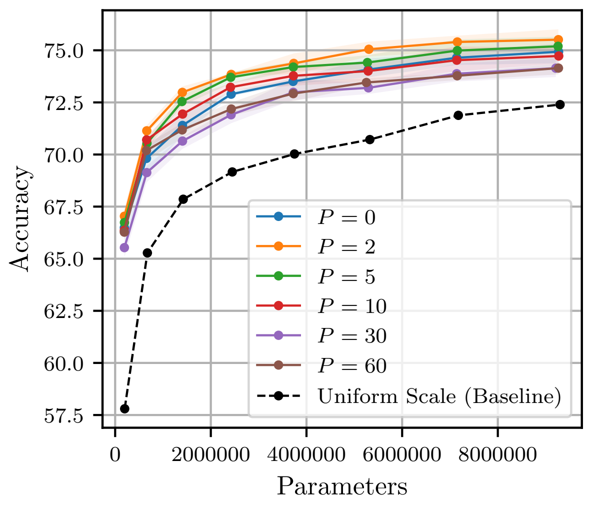


(e) $P = 30$

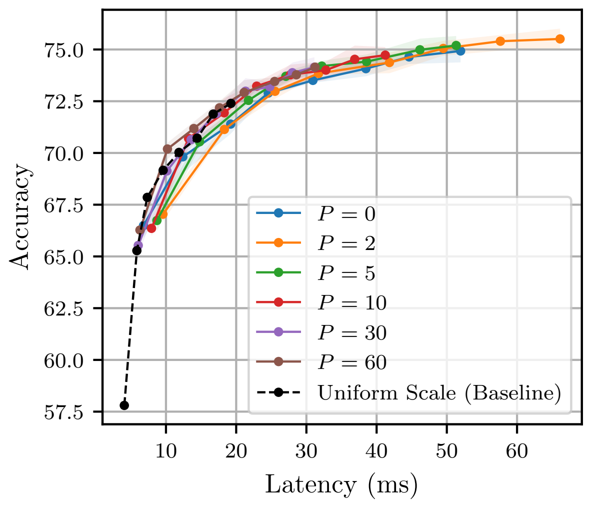


(f) $P = 60$

Figure 8: Showing the difference in searched architecture by running architecture descent on MobileNetV2 for CIFAR100 using various value of pre-training epochs P.



(a) Accuracy vs Parameters.



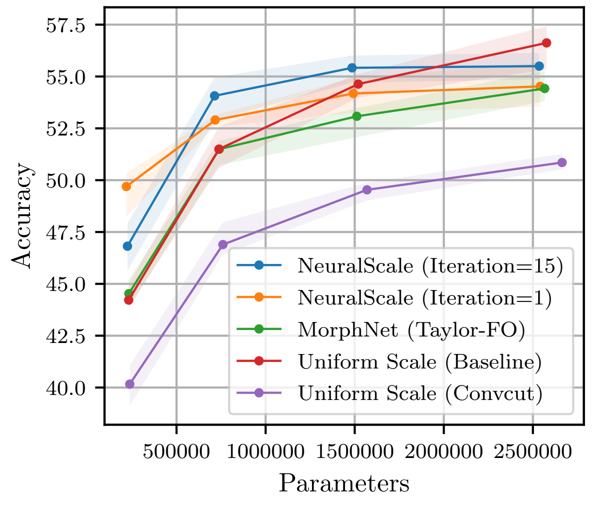
(b) Accuracy vs Latency.

Figure 9: Accuracy comparison plot for MobileNetV2 on CIFAR100 that uses different pre-training epochs P before pruning. (a) shows the accuracy comparison under different parameters using different value of P. (b) shows the comparison of accuracy under different latencies using different value of P.

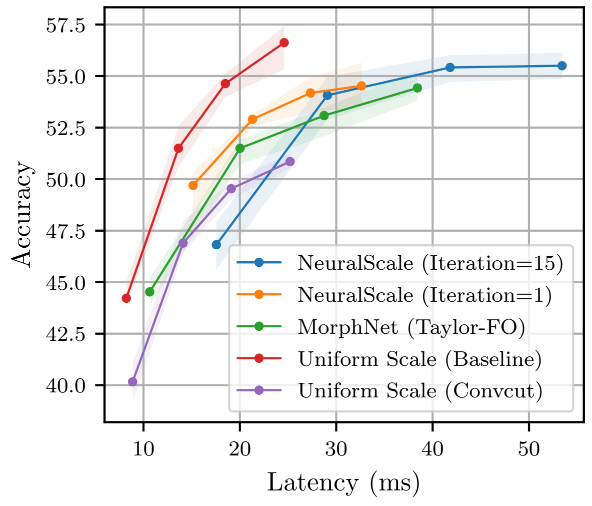
methods using ResNet18 and MobileNetV2 on TinyImageNet is shown in Figure 10 and 11 respectively. Results are summarized in Table 6 and Table 5 for ResNet18 and MobileNetV2 respectively. It can be observed that the switch from identity to convolutional mapping doesn't have drastic impact on the accuracy of ResNet18 but a significant drop in accuracy can be observed for MobileNetV2. Our conjecture is that the design of linear bottleneck layers in MobileNetV2 is to embed a low-dimensional manifold where switching from identity to convolutional mapping for shortcut layer that connects linear bottleneck layers introduces noise to this manifold which is harmful for information propagation and network training. Despite from the setback of accuracy drop through the introduction of convolutional shortcut layers, our approach is still able to induce accuracy gain in a low parameter count setting when compared to the baseline configuration setting, showing the importance of searching for the optimal configuration of filters. An unbiased comparison is to compare our approach with the convolutional shortcut (ConvCut) version of MobileNetV2 using the default set of filter configuration as shown in Figure 10 where both (ours and ConvCut) use convolutional layer as shortcut connection. On an apple-to-apple comparison, our approach shows superiority in parameter efficiency. This empirical study also explains the superiority in accuracy of iteration 1 when compared to iteration 15 of our approach as can be observed in Figure 10a. From our observation, iteration 1 of our approach generates a configuration composed repeated filters on some blocks, resulting in an architecture consisting of both identity and convolutional shortcut con-

Table 4: Accuracy comparison on MobileNetV2 for CIFAR100 using different pre-training epochs P.

Method	Params	Latency	Accuracy (%)
Uniform Scale (Baseline)	0.20M	5.53ms	57.80 ± 0.31
	1.42M	7.56ms	67.85 ± 0.38
	9.30M	20.43ms	72.40 ± 0.22
NeuralScale ($P = 0$)	0.19M	6.64ms	66.49 ± 0.43
	1.40M	18.77ms	71.39 ± 0.45
	9.27M	52.49ms	74.93 ± 0.34
NeuralScale ($P = 2$)	0.19M	9.42ms	$\mathbf{67.04 \pm 0.28}$
	1.40M	24.95ms	$\mathbf{72.98 \pm 0.26}$
	9.27M	67.55ms	$\mathbf{75.51 \pm 0.41}$
NeuralScale ($P = 5$)	0.19M	8.61ms	66.74 ± 0.39
	1.40M	21.37ms	72.54 ± 0.18
	9.26M	51.63ms	75.19 ± 0.26
NeuralScale ($P = 10$)	0.19M	7.82ms	66.36 ± 0.28
	1.41M	18.02ms	71.94 ± 0.45
	9.27M	43.00ms	74.73 ± 0.26
NeuralScale ($P = 30$)	0.19M	6.20ms	65.53 ± 0.31
	1.41M	13.35ms	70.64 ± 0.23
	9.21M	31.77ms	74.14 ± 0.35
NeuralScale ($P = 60$)	0.19M	6.15ms	66.28 ± 0.13
	1.40M	13.74ms	71.18 ± 0.24
	9.27M	32.15ms	74.15 ± 0.18



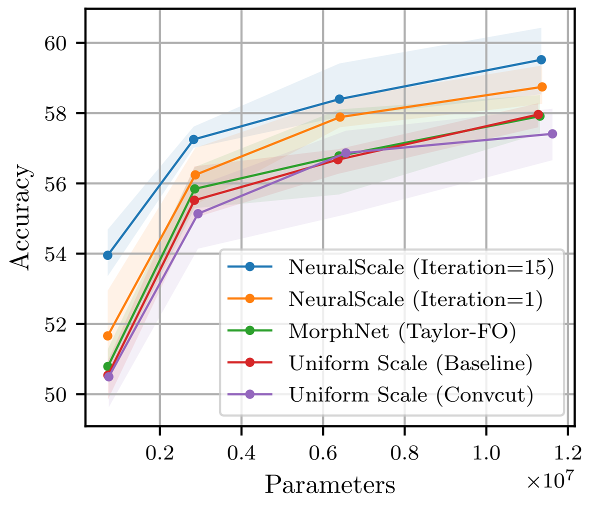
(a) Accuracy vs Parameter.



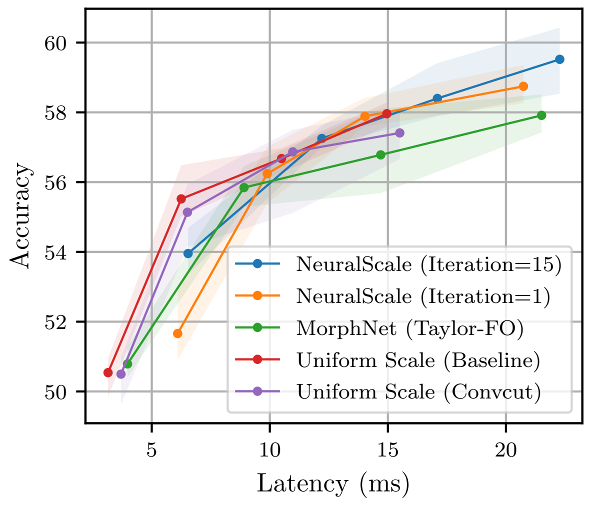
(b) Accuracy vs Latency.

Figure 10: Accuracy comparison plot for MobileNetV2 on TinyImageNet with inclusion of ConvCut.

nection. Hence, it is not surprising that iteration 1 outperforms iteration 15 of our approach as it has both traits: identity shortcut and optimized filter configuration.



(a) Accuracy vs Parameter.



(b) Accuracy vs Latency.

Figure 11: Accuracy comparison plot for ResNet18 on TinyImageNet with inclusion of ConvCut.

Table 5: Accuracy comparison on MobileNetV2 on Tiny-ImageNet (includes ConvCut).

Method	Params	Latency	Accuracy (%)
Uniform Scale (Baseline)	0.23M	8.53ms	44.22 ± 0.40
	1.52M	18.87ms	54.63 ± 0.46
	2.58M	24.70ms	**56.62 ± 0.70**
MorphNet [12] (Taylor-FO [35])	0.23M	10.47ms	44.53 ± 0.50
	1.51M	28.88ms	53.08 ± 0.52
	2.57M	38.45ms	54.42 ± 0.53
Uniform Scale (ConvCut)	0.24M	9.23ms	40.16 ± 0.63
	1.57M	19.23ms	49.54 ± 0.30
	2.66M	25.39ms	50.85 ± 0.27
NeuralScale (Iteration = 1)	0.22M	14.96ms	**49.70 ± 0.73**
	1.49M	26.98ms	54.18 ± 0.57
	2.54M	32.09ms	54.52 ± 0.72
NeuralScale (Iteration = 15)	0.22M	17.16ms	46.82 ± 0.89
	1.49M	41.20ms	**55.42 ± 0.44**
	2.54M	52.76ms	55.50 ± 0.51

Table 6: Accuracy comparison on ResNet18 on TinyImageNet (includes ConvCut).

Method	Params	Latency	Accuracy (%)
Uniform Scale (Baseline)	0.73M	3.02ms	50.54 ± 0.37
	6.36M	11.56ms	56.68 ± 0.28
	11.27M	15.46ms	57.96 ± 0.23
MorphNet [12] (Taylor-FO [35])	0.72M	3.80ms	50.79 ± 0.38
	6.39M	14.83ms	56.78 ± 0.85
	11.31M	22.07ms	57.91 ± 0.38
Uniform Scale (ConvCut)	0.75M	3.64ms	50.50 ± 0.46
	6.56M	11.99ms	56.87 ± 0.88
	11.62M	15.98ms	57.41 ± 0.58
NeuralScale (Iteration = 1)	0.72M	5.96ms	51.66 ± 0.80
	6.42M	14.58ms	57.89 ± 0.28
	11.37M	22.11ms	58.75 ± 0.37
NeuralScale (Iteration = 15)	0.72M	6.42ms	**53.95 ± 0.53**
	6.40M	17.52ms	**58.40 ± 0.54**
	11.35M	25.94ms	**59.52 ± 0.63**