Large Scale Neural Architecture Search with Polyharmonic Splines

Ulrich Finkler[1], **Michele Merler**[1], **Rameswar Panda**[1], **Mayoore S. Jaiswal**[2], **Hui Wu**[1], **Kandan Ramakrishnan**[3], **Chun-Fu Chen**[1], **Minsik Cho**[1*], **David Kung**[1], **Rogerio Feris**[1], **Bishwaranjan Bhattacharjee**[1]
[1]IBM Research [2]Nvidia [3]Baylor College [*]currently at Apple
{ufinkler, mimerler, wuhu, chenrich, minsikcho, kung, rsferis, bhatta}@us.ibm.com,
rpanda@ibm.com, mjaiswal@nvidia.com, kandan.ramakrishnan@bcm.edu

Abstract

Neural Architecture Search (NAS) is a powerful tool to automatically design deep neural networks for many tasks, including image classification. Due to the significant computational burden of the search phase, most NAS methods have focused so far on small datasets. All attempts at conducting NAS at large scale have employed small proxy sets, and then transferred the learned architectures to larger datasets by replicating or stacking the searched cells. We propose a NAS method based on polyharmonic splines that can perform search directly on large scale target datasets. We demonstrate the effectiveness of our method on the ImageNet22K benchmark[1] (Deng et al. 2009), which contains 14 million images distributed over $21,841$ categories. By exploring the search space of the ResNet (He et al. 2016) and Big-Little Net ResNext (Chen et al. 2019a) architectures directly on ImageNet22K, our polyharmonic splines NAS method designed a model which achieved a top-1 accuracy of 40.03% on ImageNet22K, an absolute improvement of 3.13% over the state of the art with similar global batch size (Codreanu, Podareanu, and Saletore 2017).

Introduction

Designing Deep Neural Networks is a challenging task and requires a Subject Matter Expert (SME). One way of reducing the design burden and still be able to obtain a custom designed architecture for a given training problem is to use Neural Architecture Search (NAS) (Elsken, Metzen, and Hutter 2019; Zoph and Le 2017). Most NAS methods have been tried on small scale datasets like Cifar10-100 (Krizhevsky 2009) and then subsequently architectures have been transferred to larger datasets like ImageNet1K (Borsos, Khorlin, and Gesmundo 2019; Lu et al. 2020; Wistuba 2019). This is mainly due to the heavy computational burden of the search phase, which despite recent progress (Liu, Simonyan, and Yang 2018; Zela, Siems, and Hutter 2020), remains mostly intractable at large scale. No previous published work has explored NAS on large scale datasets like ImageNet22K directly, and very few have even attempted it on ImageNet1K itself.

In this paper we describe a novel method for NAS that can conduct search directly on large-scale datasets using Polyharmonic Splines. First, we describe in detail how specific operations within a deep neural network (convolution, ReLu, average pooling, batchnorm) are well suited to be approximated by a spline. Given a search space consisting of a set of d operations to optimize over, the search phase of our proposed NAS method based on polyharmonic splines requires only an initial set of $2d + 3$ points, followed by few additional points to complete its optimization. This means that the number of evaluations required in the search phase depends only on the number of operations under search, not on the number of possible values for each of those operations. This drastically reduces the computational cost of the search phase of our NAS approach, allowing it to be performed directly on large scale target datasets.

We demonstrate the effectiveness of our method on the ImageNet22K benchmark of 14 million images over $21,841$ labels. By exploring the search space of ResNet (He et al. 2016), Big-Little Net (Chen et al. 2019a) and ResNext (Xie et al. 2017b) architectures, the method designed a model which achieved a top-1 accuracy of 40.03% on ImageNet22K. This significantly improves the best published performance (Codreanu, Podareanu, and Saletore 2017) of 36.7% on such large-scale dataset.

Related Work

NAS. Neural Architecture Search (NAS), which aims at automatic design of deep learning networks for various applications spanning from image classification (Liu et al. 2018a; Tan et al. 2019; Wu et al. 2019; Xie et al. 2019a) to NLP (Liu, Simonyan, and Yang 2018; Pham et al. 2018; Zoph and Le 2017), from object detection (Chen et al. 2019b; Ghiasi, Lin, and Le 2019; Wang et al. 2019) to semantic segmentation (Liu et al. 2019) and control tasks (Gaier and Ha 2019), has attracted intense attention in recent years. A number of NAS strategies have been proposed, including evolutionary methods (Baker et al. 2018; Liu et al. 2018b; Real et al. 2019; Zhou et al. 2020), reinforcement learning (Liu et al. 2018a; Pham et al. 2018;

[1]ImageNet22K was used for research purposes only to allow benchmarking against prior results from others on this data set. The trained models in this work are not used for commercial purposes

Zhong et al. 2018; Zoph and Le 2017), and gradient-based methods (Chang et al. 2019; Liu, Simonyan, and Yang 2018; Nayman et al. 2019; Xie et al. 2019b; Zela et al. 2020). Efficiency on specific platforms has also been a very active area of research within the NAS umbrella, with the development of search strategies that optimize not only accuracy but also latency (Cai, Zhu, and Han 2019; Howard et al. 2019; Tan et al. 2019; Tan and Le 2019; Wu et al. 2019). Methods based on micro-search of primary building cells (Zhong et al. 2018; Zoph et al. 2018), and parameter sharing between child models (Bender et al. 2018; Brock et al. 2018; Liu, Simonyan, and Yang 2018; Pham et al. 2018; Zela, Siems, and Hutter 2020) have also been recently proposed. Another popular approach to address efficiency in NAS is to search for an architectural building block on a small dataset (e.g., CIFAR10) and then transfer the block to the larger target dataset by replicating and stacking it multiple times in order to increase network capacity (Panda et al. 2021).

Bayesian Methods. Bayesian optimization which uses an acquisition function to obtain suitable candidates has been widely used in NAS (Domhan, Springenberg, and Hutter 2015; Mendoza et al. 2016; Kandasamy et al. 2018; Cao, Wang, and Kitani 2019). The acquisition function measures the utility by accounting for both the predicted response and the uncertainty in the prediction. While Bayesian optimization methods usually consume more computation to determine future points than other methods, this pays dividends when the evaluations are very expensive. Many different strategies have been studied including classic Gaussian process-based optimization (Swersky et al. 2014; Kandasamy et al. 2018), tree-based models (Bergstra, Yamins, and Cox 2013), random forests (Falkner, Klein, and Hutter 2018) to effectively optimize both neural network architectures and their hyperparameters. Several works also predict the validation accuracy (Zhang, Ren, and Urtasun 2018), or the curve of validation accuracy with respect to training time (Baker et al. 2018).

Optimization Basis

Neural Architecture Search in its essence optimizes a function $y = f(\vec{x})$, $\vec{x}$ residing in a multidimensional space of network and training parameters and y being a metric for the quality of the network, for example the top-1 accuracy on a validation data set. The parameters captured in $\vec{x}$ may be discrete. f might not be differentiable in terms of the parameters x_i. Importantly, f is generally very expensive to evaluate. For a given parameter set $\vec{x}$, the network has to be trained and evaluated on the full training and validation sets.

The interpolation quality of methods to approximate scalar functions of multidimensional arguments based on a limited set of support points depend on the properties of the interpolated function. Hence it is important to understand the properties of f and if it is likely that there exists a well behaved interpolating function $F : R \rightarrow R^N$ through the potentially discrete support points.

This section discusses the algebraic structure of popular residual neural networks and establishes that the resulting $f(\vec{x})$ is with high probability well suited for interpolation with a polyharmonic spline for parameters that have a sufficiently large number of possible values. For parameters with only a few choices, using separate splines for the discrete cases can be more effective. Furthermore, the algebraic analysis provides insights into the sensitivity of global network parameters and hence search spaces.

Neural Network Forward Pass as Piecewise Linear Function

The forward pass through a typical neural network consisting of convolutions, fully connected layers, ReLUs, batchnorms and average-pooling can be interpreted as a piecewise linear function that effectively transforms an input, for example an image as a set of 224×224 values across three channels, into d_f values in a *feature space*, which forms the input to a final linear classifier.

Fully Connected Layers A single neuron with N inputs performs the operation $y = \phi\left(\sum_{i=1}^{N} w_i x_i + b\right)$ with an activation function ϕ. In the following we discuss the case $\phi(x) = max(0, x)$, i.e. ReLU.

The condition $0 = \sum_{i=1}^{N} w_i x_i + b$ defines a hyperplane in the N dimensional input space of $\vec{x}$ such that $y > 0$ for locations above the hyperplane and $y = 0$ for locations on or below the hyperplane. The output of the neuron without activation is in fact the distance of $\vec{x}$ from the hyperplane. Multiple neurons with inputs $\vec{x}$ from the same input space define a set of hyperplanes that partition the input space into a set different regions. In each region, $\vec{y}$ is determined by a set of linear equations.

For example in a two-dimensional input space, 2 neurons $y^{[1]}$ and $y^{[2]}$ with linearly independent weight vectors and ReLU activation create four regions

$$\vec{y} = (\sum_{i=1}^{N} w_i^{[1]} x_i + b^{[1]}, \sum_{i=1}^{N} w_i^{[2]} x_i + b^{[2]}) \quad ; \quad \vec{y} = (0, 0)$$

$$\vec{y} = (\sum_{i=1}^{N} w_i^{[1]} x_i + b^{[1]}, 0) \quad ; \quad \vec{y} = (0, \sum_{i=1}^{N} w_i^{[2]} x_i + b^{[2]})$$

A second layer of similar structure, a linear operation followed by ReLU activation, potentially partitions each of the segments of the input space again. In the following we discuss how other layers, namely *convolution*, *batchnorm* and *average pooling*, are operations of analogous structure and hence the entire set of operations leading to the final linear classifier is a piecewise linear function that maps hypervolumes in the input space into the feature space. For a trained neural network, this piecewise linear function is optimized to create linearly separable clusters of mapped training points in the feature space.

Convolution Convolution in a neural network is

$$y_{ij}^{(k)} = \sum_{a,b,c} W_{abc}^{(k)} x_{(i+a)(j+b)c}$$

wherein c is the input channel, e.g. color in an rgb image or filter from a prior convolution. The tuple (i, j) is the location within the input at which a *patch* of the size of a filter is scanned. k identifies the filter, i.e. the output channel of the convolution. a and b iterate through the positions within the input patch and filter. Each tensor $\bar{W}$ is of size $N_A{*}N_B{*}N_C$, the filter width, height and number of input channels.

We can apply an index transformation $\gamma = a * N_B * N_c + b * N_C + c$ which transforms $W^{(k)}_{abc}$ into $W_{k\gamma}$. For a fixed position (i, j) of the convolution we can drop the indices i and j to obtain $y_k = \sum W_{k\gamma} x_\gamma$. The vectors $\vec{x}$ are points in a vector space of dimension $N_A * N_B * N_C$. I.e., on one input patch a convolution has the same algebraic structure as a neuron. Average-pooling is a convolution with a filter whose elements are all identical. Note that these operations can be expressed in the form of a fully connected layer, with a set of neurons that correspond to the filter channels that is present multiple times with different subsets of the input weights set to zero to express the selection of input patches.

Batch Norm On first glance, a batch norm layer does not look like a linear function. But a closer look at the details reveals that the running means and averages are treated as constants in backpropagation, they do not have gradients. The batch norm (mean and variance across the batch, width and height for a channel) is an estimate for the constant that is used during inference on the trained model, which is in itself an estimate for the mean and variance across the entire training set. As the weights change during training, the estimates for running means and variances follow.

Pytorch uses Bessel's correction $s = \sigma \frac{n}{n-1}$, an unbiased estimate of the population variance $\sigma = \frac{1}{n}\sum_i (x_i - \mu)^2$ from a finite sample for *BatchNorm2d*.

With x_{ijkl} as an input tensor of form NCWH, $n = \sum_{ikl} 1$ being the number of elements in a 'channel slice' and t as the iteration number, mean and unbiased variance for a 'channel slice' are

$$\begin{aligned}\mu_j(t) &= \frac{1}{n} h_j(t) \quad with \quad h_j(t) = \sum_{ikl} x_{ijkl}(t) \\ \psi_j(t) &= \frac{1}{n-1} g_j(t) \quad with \quad g_j(t) = \sum_{ikl} (x_{ijkl}(t) - \mu_j(t))^2\end{aligned}$$

With running mean and variance

$$\begin{aligned}M_j(t) &= (1-m)M_j(t-1) + m\mu_j(t) \quad with \quad M_j(0) = 0 \\ S_j(t) &= (1-m)S_j(t-1) + m\psi_j(t) \quad with \quad S_j(0) = 1\end{aligned}$$

batch norm in training (B) and eval (B') mode result in $B(x_{ijkl}(t)) = \frac{x_{ijkl}(t) - \mu_j(t)}{\sqrt{\sigma_j(t) + \epsilon}}$ $\quad B'(x_{ijkl}(t)) = \frac{x_{ijkl}(t) - M_j(T)}{\sqrt{S_j(T) + \epsilon}}$
Hence, with M and S as constants, batch norm maps a linear transformation.

Base Transformations, Projections and Convolutional Neural Networks

The rank-factorization $\bar{A} = \bar{C}\bar{B}$ of a $m \times n$ matrix $\bar{A}$ of rank r expresses $\bar{A}$ as a product of a $m \times r$ full column rank matrix $\bar{C}$ and a $r \times n$ full row rank matrix $\bar{B}$. For a square matrix of full rank the equation $y_k = \sum_{i=1}^{N} w_{ki} x_i$ is a transformation of vector x_i from one base of R^N to another.

Hence, conceptually a $m \times n$ matrix of rank r maps a vector from a space with m dimensions into a subspace of r dimensions and from there into a space with n dimensions. If $r < n$, then we can express all of the n dimensional vectors in terms of an r dimensional basis. Thus, a trained network whose matrices are not close to full rank is inefficient. Furthermore, we can interpret a convolution and subsequent activation on a single 'patch' as a transformation from a $m = w * h * c_{in}$ dimensional space to a $n = c_{out}$ dimensional space, w and h being the width and height of the filters and c_{in} and c_{out} being the numbers of the input channels and output channels, respectively.

Assuming that the combined convolution matrix $\bar{W}_{k\gamma}$ has close to full rank in an efficient network, the activations resulting from a convolution can be interpreted as the coefficients in an expression that approximates the input tensor in terms of a set of functions defined by the filter tensors, a form of compression. The "approximation" has to be of limited loss, since the correlation between classification results and inputs has to be preserved. In some sense training optimizes the approximation such that it aides in transforming inputs into linearly separable points in the *feature space* and at the same time it allows to approximate with limited loss of information across all its inputs.

Based on above observations, the essential part of the functionality of the *convolutional cone* leading to a linear classifier in a neural network is provided by mapping points from one space to another to create a piecewise linear function whose output in the feature space is highly linearly separable with respect to the number of hyperplanes in the final classification layer. What primarily matters is how many linear pieces the network provides relative to the number of hyperplanes, i.e. classes, in the final classifier.

A second aspect is probability of network convergence, which suggests to focus on residual networks with batchnorm. While the specific structure of a network should have relatively little influence on final accuracy, it may have a significant impact on the network size that is required to generate a competitive piecewise linear approximation. Which lead us to focus architecture search on different families of residual networks, specifically ResNet (He et al. 2016), ResNeXt (Xie et al. 2017a) and BLResNeXt (Chen et al. 2019a). Typical residual networks have 4 *groups* of residual blocks with different numbers of layers. For example ResNet50 has groups of input width 64, 128, 256, 512 channels with an expansion factor of 4 and depths of 3, 4, 6, 3. ResNet18 on the other hand has the same widths, but an expansion factor of 1 and depths of 2, 2, 2, 2.

The number of classes, i.e. the number of hyperplanes in the feature space, provides guidance on the optimal number of dimensions for the feature space. In an N dimensional space, we can place N linearly independent hyperplanes such that all volumes bounded by hyperplanes are infinite. If we place more hyperplanes, some volumes have to be limited to a constant that depends on the placement of the hyperplanes. Hence, a dimesion of the feature space that is larger than the number of classes should be beneficial. Note that for ResNet50 the dimension of the feature space is 2048 and for ResNet18 it is 512.

If the leading layers "choke" the dimensions of the vector spaces leading to the feature space too much, the analysis suggests that this will negatively influence the accuracy of the network. For example the initial convolution in ResNet18 has an input space of $7 \times 7 \times 3 = 147$ dimensions and maps this to 64 dimensions. A 3×3 convolution with 64 channels to 128 channels maps 576 dimensions to 128, i.e. creates a significant reduction. Furthermore, the number of activations drops by a factor of 4 through every group.

As an experimental test we designed a simple neural network r18U based on ResNet18 whose behavior should be closer to ResNet50 by adjusting only the block widths. We eliminated the *max-pool* layer and compensated for its reduction by increasing the stride of the initial convolution to 2 based on the hypothesis that the non-linear layer is not essential for the functionality of the neural network. Indeed, this did not negatively impact final accuracy but appeared to improve conversion. We increased the numbers of channels of the initial convolution and the subsequent layer-groups from 64,64,128,256,512 to 128,256,512,1024,2048. Note that the dimension of the feature space matches that of ResNet50 and the output dimension of the output of the initial convolution almost matches the input dimension. The network r18U achieved a validation accuracy of over 76.5% on ImageNet1K, compared to ResNet50's 75.1%, in our pytorch setup. Network r18U is less efficient than ResNet50, it has significantly more parameters. Larger depth increases the potential number of pieces in the piecewise linear function for a similar number of neurons.

Our algebraic analysis, experimental results and theory (two layer theorem) suggest that roughly the same final accuracy can be achieved by many network families, the distinguishing factor is the required model size. Imposing structure via convolutions, higher depth, residual blocks and their substructures increases the granularity of the piecewise linear function for a network with a given number of neurons.

Polyharmonic Spline NAS

Given the insights above, we developed a method to efficiently investigate a high dimensional parameter search space. We analyzed the search space of a single network family like *ResNet* or *ResNeXt*. Widths and depths of blocks already provide 8 dimensions within any deep network family search space. On top of those, parameters internal to the blocks, such as cardinality in *ResNeXt*, would further increase the number of dimensions, but with less impact on final accuracy, based on the algebraic interpretation.

Three key issues make NAS challenging for larger classification problems such as ImageNet1K or ImageNet22K: search (and evaluation) time, memory consumption, and probability of getting stuck in local minima.

First, in order to obtain a measure for the final accuracy of a model, it has to be trained nearly to saturation, that is, over a sufficiently large number of epochs to ensure that all the variants tested reach close to their full potential. In our experiments, early stopping proved to be misleading, since smaller networks tend to initially converge faster and the crossover point was near the end of a complete learning rate schedule. Training ImageNet1K for 90 epochs and ImageNet22K for 60 epochs (values experimentally shown to provide meaningful accuracies) requires large amounts of computation. Hence, as in most NAS approaches, *minimizing the number of evaluations is critical*.

Second, the algebraic observations suggested that the better performing networks are large, such that GPU *memory limitations become a factor*. "Supernet" approaches such as FBNet(Wu et al. 2019) would not fit even two variants of the larger and more performing networks into a 16 GB GPU. In fact, the most accurate networks on ImageNet22K tend to use most of the memory of even a 32 GB GPU.

Third, the topologies of the hypersurfaces defined by the parameter dimensions and the achieved final accuracy as the evaluation axis may have local minima, i.e. *gradient descend based methods are suboptimal to find minima in the accuracy-hypersurface*.

Importantly, the algebraic investigation suggests that small changes to parameters such as the number of filters or layers in a "convolution group" or cardinalities block elements within a "network family" like ResNeXt or BLResNeXt will result in small changes in the final accuracy. Because they cause only small changes in the degrees of freedom in the number of pieces in the piecewise linear function and the equations that govern them. Our experimental results support this hypothesis. Hence, it is appropriate to assume that there exists a set of continuous functions in $f : R^d \to R$ that pass through a set of reasonably spaced support points over d discrete parameters and the resulting validation accuracy after training the network (close) to saturation.

Polyharmonic Splines

Polyharmonic splines are a valid option to define a function that passes through such a set of support points. Given a set of N *support points* $(\vec{x}_j, f(\vec{x}_j))$ for a function $f : R^d \to R$, a polyharmonic spline interpolation has the form

$$s(\vec{x}) = \sum_{j=1}^{N} c_j \Phi(||\vec{x} - \vec{x}_j||) + p(\vec{x})$$

where $||\vec{x}||$ is the Euclidian norm in R^d and $p(\vec{x})$ is a real valued polynomial in d variables (Iske and Arnold 2004). Φ is a radial basis function. Polyharmonic radial basis functions are solutions to a polyharmonic equation, a partial differential equation of the form $\Delta^m f = 0$. Polyharmonic splines minimize the "curvature" of the hyperplane passing through all the support points, hence they minimize oscillations while providing a smooth, continuous surface. This interpolation expression is differentiable. If we assume there exists a continuous and differentiable function $f : \mathcal{R}^d \to \mathcal{R}$ that correctly models the behavior of the system that generates the support points, then there exists a polyharmonic spline such that the integral of the difference between spline and f in an interpolation *d-box* vanishes as the number of support points increases. We chose a radial basis function of $\Phi(r) = r^3$. Solving the equation system to determine the coefficients for the polyharmonic spline proved to be sensitive to numerical instability. Hence we chose a pivoting Householder QR decomposition, trading performance for numerical stability. For the linear equation system that determines the spline coefficients to have a solution, the matrix formed by the support points must have full rank. Since interpolation accuracy is in general higher than extrapolation accuracy, a minimal set of support points it needed that spans a *d-box* in the d-dimensional space in which the spline interpolates the approximated function which leads to a support point matrix of full rank. We chose the following set for d parameters:

- $\{max_d | d = 1..N\}$, vector of the largest usable or legal parameter values in the *d-box* for all dimensions.
- $\{min_d | d = 1..N\}$, smallest usable or legal parameter values.
- $\{(max_d + min_d)/2 | d = 1..N\}$, point near the *d-box* center.
- $\{max_d | d \neq k, d \in \{1..N\}, min_d | d == k\}$ for all $k = 1..N$.
- $\{min_d | d \neq k, d \in \{1..N\}, max_k | d == k\}$ for all $k = 1..N$.

For two dimensions, these are the corners of a square and its center. For three dimensions, the corners of a cube and its center. A total of $2d + 3$ support points is sufficient to span an initial cubic spline that interpolates within a *d-box*. Additional support points can be added to improve the quality of the interpolation. To maintain numerical stability, new support points have to maintain a certain distance from previous support points. We eliminated splines due to numerical instability by checking $Ax - B = 0$ on the solution with an error margin for floating point computation errors.

Minima Search

Since gradient descent is susceptible to local minima, we used a hierarchical Monte Carlo Sampling (MCS) approach to search for a minimum in the interpolation d-box, with a sequence of low discrepancy, specifically a Halton sequence. For a given d-box, reducing the average distance between sample points by a factor of two requires an exponential increase in additional sample points, i.e. progress in terms of finding better minima slows dramatically as more sample points are added. As the average distance between sampling points decreases relative to the distance between support points, the curvature minimizing property of the polyharmonic spline reduces the risk of missing minima. Hence we adopted the hierarchical approach described in Algorithm 1, which iteratively shrinks the d-box around the minimum found so far, greatly reducing the time to determine a good approximation for the minimum within the d-box. Search was stopped if no further improvement within floating point accuracy was achieved within a certain compute budget.

Note that in our architecture search a potentially bad estimate for a minimum does not limit progress of the search. It merely leads to a potentially suboptimal measurement point and hence to potentially requiring more measurement points to reach convergence between prediction and measurement.

Algorithm 1: Polyharmonic Splines NAS

```
Input:
(a) Initial set of $2d + 3$ support points $\vec{x} = \{\boldsymbol{x}_1, \boldsymbol{x}_2, ...\}$;
(b) Measured values $\vec{y}_m = f_m(\vec{x})$ for initial points;
(c) Min. Diff. $\epsilon$ between prediction ($f_s$) and measure ($f_m$)
begin
    Solve eq. system for spline $f_s(\vec{x})$ over initial set;
    Compute $\boldsymbol{x}_{max} = \arg\max_{\boldsymbol{x}} f_s(\vec{x})$ via nested MCS;
    Compute measurement $f_m(\boldsymbol{x}_{max})$ by training network;
    $\boldsymbol{x}_{top} = \boldsymbol{x}_{max}$;
    while $|f_m(\boldsymbol{x}_{max}) - f_s(\boldsymbol{x}_{max})| > \epsilon$ do
        Add $\boldsymbol{x}_{max}$ to set of support points $\vec{x}$;
        Solve eq. system for spline $f_s(\vec{x})$ over all points;
        Compute $\boldsymbol{x}_{max} = \arg\max_{\boldsymbol{x}} f_s(\vec{x})$ via nested MCS;
        Compute $f_m(\boldsymbol{x}_{max})$ by training network;
        if $f_m(\boldsymbol{x}_{max}) > f_m(\boldsymbol{x}_{top})$ then
            $\boldsymbol{x}_{top} = \boldsymbol{x}_{max}$;
        end
    end
end
Result: Optimal Parameters Configuration $\boldsymbol{x}_{top}$
```

Experiments

In our experiments we investigated a couple of micro (within basic block structure) and multiple macro (overall network dimensions) parameters for two network families, namely ResNet (He et al. 2016) and Big-Little ResNeXt (Chen et al. 2019a). Search was performed directly on the target dataset ImageNet22K. Training experiments were conducted on multiple GPU clusters[2], using 34 nodes each equipped with 6 Nvidia Volta V100 GPUs with 16GB GPU cache each. for a total of 204 GPUs. All GPUs in a node have NVLink connection, and the nodes are connected by Mellanox EDR 100G Infiniband and have access to shared GPFS storage. Software used included PowerAI Vision 1.6, NCCL and Pytorch, using its distributed data parallel package. Batch size was set at 32 per GPU, for a total batch size of 6,528. The initial learning rate was set at 0.1 and then followed polynomial decay, optimizer SGD with momentum 0.9 and weight decay 0.0001.

Experimental Dataset: ImageNet22K

ImageNet22K contains 14 million images representing 21,841 categories organized in a hierarchy derived from WordNet and including top level concepts such as sport, animal, food, person, tools, music, etc. On average there are 654 images per class, ranging from 1 to 3,047. The scale and imbalance of ImageNet22K make it particularly challenging even for human designed architectures, with a limited set of published results (Chilimbi et al. 2014; Cho et al. 2017; Codreanu, Podareanu, and Saletore 2017; Zhang et al. 2015), as opposed to the smaller ImageNet1K version. Recently, some works have used ImageNet22K as pre-training for Imagene1K evaluation (Dosovitskiy et al. 2021). To the best of our knowledge, this is the first work to perform NAS directly at the scale of ImageNet22K, not by transfer from smaller proxy sets (Panda et al. 2021).

Following standard practice (Bhattacharjee et al. 2017; Chilimbi et al. 2014; Deng et al. 2010) we randomly partitioned the ImageNet22K dataset into 50% training and 50%

[2]This research used resources of the Oak Ridge Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725, as well as the IBM T.J. Watson Research Center Scaling Cluster (WSC).

Table 1: Simplified ResNet18 with 15 points over 5 dimensions of search including 13 initial points plus 2 incremental

Point Type	Dimensions					Top-1 Accuracy %	
	c1	g1	g2	g3	g4	Measured	Predicted
Initial	150	300	600	1200	2400	38.03	-
	32	32	32	32	32	10.33	-
	150	32	32	32	32	10.54	-
	32	300	32	32	32	15.46	-
	32	32	600	32	32	19.45	-
	32	32	32	1200	32	23.15	-
	32	32	32	32	2400	31.25	-
	75	150	300	600	1200	35.46	-
	32	300	600	1200	2400	38.03	-
	150	32	600	1200	2400	37.15	-
	150	300	32	1200	2400	36.76	-
	150	300	600	32	2400	34.40	-
	150	300	600	1200	32	29.04	-
Incremental	50	116	330	1200	2400	37.31	41.02
	80	208	475	736	2400	-	38.92

validation, consisting of approximately 7 million images each, with the number of images per label approximately equal in both sets.

Results

We applied our polyharmonic spline NAS method to the the ResNet18 and BLResNext50 architectures search spaces. For each point in the spline, training and evaluation was conducted on half of the ImageNet22k dataset. Once the optimal configuration was determined by our search, that network was trained and evaluated on the full ImageNet22k dataset.

ResNet18 Search Space. For the ResNet18 architecture we removed maxpool layer and increased the stride for the first convolution to 2. We performed search over $d = 5$ dimensions $c1, g1, g2, g3, g4$: the number of filters in the first convolution ($c1$) and in the four groups of layers ($g1, g2, g3, g4$). The ranges for each dimension are as follows: $c1 \in [32, 150]$, $g1 \in [32, 300]$, $g2 \in [32, 600]$, $g3 \in [32, 1200]$ and $g4 \in [32, 2400]$. Therefore the possible combinations spanning the entire search space are $\sim$3 trillion ($120 \times 135 \times 570 \times 1170 \times 2368$). Our spline NAS approach starts from *only* an initial set of 13 support points ($2d+3$, as explained before) and then iterate from there with few additional points. This results in a tremendous computational gain for our NAS method which allows to perform search directly on large scale datasets, as opposed to traditional NAS approaches needing to resort to small proxy sets.

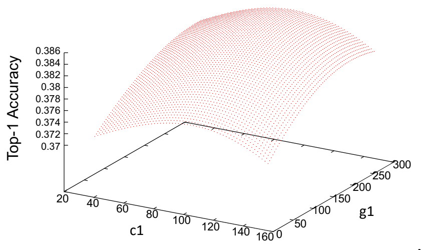
Table 1 shows the coordinates for the initial 13 support points to span the first spline and two incremental points. An incremental point is a measurement on the optimum estimated by the prior set of points. The first prediction, based on the minimal point set, predicted a reasonable point, but the estimate of 41% is clearly very optimistic. Adding a measurement at this point produces a new point with a prediction close to the best support point. Figure 1 shows projections of the polyharmonic spline derived from the 14 measured points as show in Table 1. The interpolation suggests that the parameter $d5$ is the dominant limiting factor, since it shows the steepest slope at the edge of the "box". This matches the algebraic interpretation, $\sim$22k classes could benefit from a higher dimensional feature space. The earlier layers/group show maxima within the box for maximum values for the later layers, indicating that once the degrees of freedom of a later part of the network are saturated, adding more capacity to earlier layers becomes counterproductive. Hence, we measured configuration of $[300, 600, 1200, 2400, 5400]$, projecting the ratios of the optimum from the spline with adjustments to fit into available GPU memory. This achieved, all other hyperparameters remaining equal, a top-1 accuracy of 39.76%. Since this network was larger and hence may benefit from a different learning rate schedule, we performed 2 epochs of fine tuning, which increased the accuracy to 40.37%. Being limited by GPU memory and compute resource, we performed one more experiment to increase the number of pieces in the piecewise linear function without increasing the model size significantly by replacing the 7×7 convolution with stride 2 at the beginning with a basic block. A basic block consists of two 3×3 convolutions, one of which has stride 2, which has a similar aperture and the same reduction of the activations. Indeed, this yielded the expected improvement, 40.68% top-1 accuracy with fine-tuning.

Table 2: BLResNext50, 3 dimensions (α, β, and ϕ), 9 points for initial spline, 1 incremental

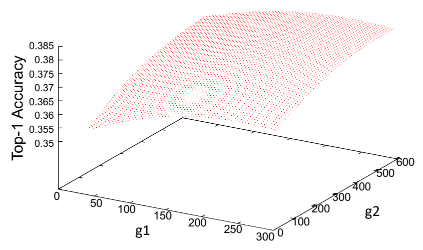
Point type	Initial									Incremental
α	8	2	2	8	4	2	8	2	8	2
β	8	2	8	2	4	2	2	8	8	8
ϕ	1	1	1	1	1.5	2	2	2	2	3
Top-1 %	38.17	38.83	38.75	38.18	39.90	40.96	40.53	40.99	40.48	41.64

BLResNext50 Search Space Big-Little Net(Chen et al. 2019a) is a mechanism that splits each block within a deep network into multiple paths through which different resolutions of an input are passed. In our search space we considered two paths. The first, called big branch, through which a downsized (by half) version of the input is passed, containing C kernels and L layers. The second, called little branch, processing the input in its original resolution, but containing C/α kernels and L/β layers. The big-little version of ResNext (Chen et al. 2019a) with a depth of 50 is deeper and offers more alternate paths through groups than the basic ResNet18, and hence theoretically allows for more pieces in the piecewise linear function relative to the number of network parameters. Thus, this family of networks promises to achieve higher accuracy within a given GPU memory capacity, which was the limiting factor for the ResNet18 case.

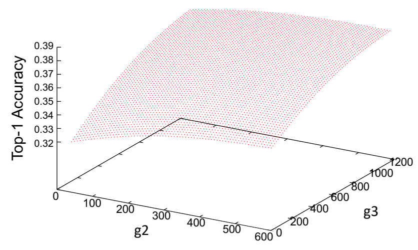
We defined our search space as spanning only three variables, hence reducing the number of needed measurements for optimization. We chose as parameters the $\alpha \in [2, 8]$ and $\beta \in [2, 8]$ parameters of the big-little structure and a multiplier $\phi \in [1.0, 2.0]$ to the group width, that gets uniformly applied across the network. The original group widths for BLResNext50 were $64, 128, 256, 512$. With a bottleneck expansion factor of 4, this results in a 2048-dimensional feature space. The choice $\phi = 2$ for example results in group width $128, 256, 512, 1024$ and a 4096-dimensional feature space. The total combinations of the search space are at least 539 ($7 \times 7 \times 11$, considering ϕ only at discrete increments of 0.1), but the spline optimization needs only $2 \times 3 + 3 = 9$



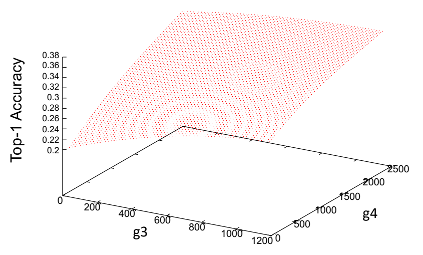
(a) Projection $[c1, g1, 600, 1200, 2400]$



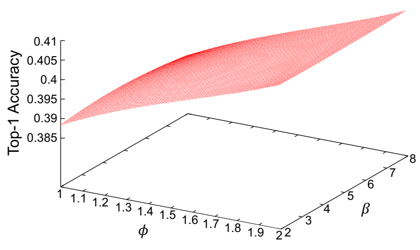
(b) Projection $[150, g1, g2, 1200, 2400]$



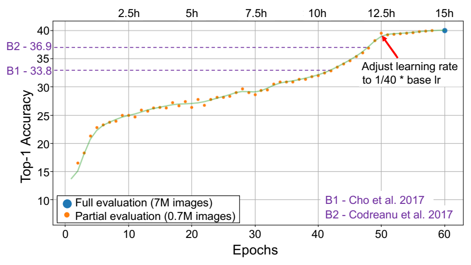
(c) Projection $[150, 300, g2, g3, 2400]$



(d) Projection $[150, 300, 600, g3, g4]$



(e) Projection $[\alpha = 2, \beta, \phi]$



(f) BLResNext50 18x8d test accuracy

Figure 1: (a-d)Top-1 accuracy on ImageNet22k (half) for the ResNet18 architecture by projecting the search space to two dimensions. For each projection, three parameters are fixed and performance is inspected over the search space spanning the range of the remaining pair of parameters: (a) varying $c1$ and $g1$, (b) varying $g1$ and $g2$, (c) varying $g2$ and $g3$, (d) varying $g3$ and $g4$. (e) Top-1 accuracy (half ImageNet22K) over parameters β and ϕ for BLResNext50. (f) BLResNext50 18x8d test accuracy on the full ImageNet22K (34 nodes, 204 GPUs, 6,528 batch size on Summit Supercomputer).

Table 3: Comparison of ImageNet22K results. The last two row denotes the Spline NAS recommended architecture. FLOPs are estimated with a network forward pass using input image resolution of 256x256

Model	Batch Size	GPUs	Top-1(Top-5) Accuracy %	FLOPs (G)	Training Time(Hours)	Number of Epochs
ResNet-101 (Cho et al. 2017)	5,120	256	33.8 (-)	-	7	-
WRN-50-4-2 (Codreanu, Podareanu, and Saletore 2017)	6,400	200	36.9 (65.1)	-	-	24
BLResNext101 32x8d	6,528	204	39.7 (68.3)	11.25	16	60
BLResNext50 18x8d (ours)	**6,528**	**204**	**40.03 (69.04)**	**17.88**	**15**	**60**

supporting measurements followed two additional ones. Table 2 shows that α and β have only a small influence on accuracy compared to ϕ. Figure 1(e) shows the projection $acc(\alpha = 2, \beta, \phi)$. The dependencies within the "box" are nearly linear and the minimum is located in the $\alpha = 2, \beta = 8, \phi = 2$ corner, clearly indicating that an increase in width has the best probability to increase accuracy significantly. Hence, we measured $\alpha = 2, \beta = 8, \phi = 3$, which yielded to top-1 accuracy of 41.64%. This was interesting also because the shape of the spline suggested an increment in the variable ϕ beyond the initially designed search space range. The total amount of Nvidia V100 GPU hours needed for the NAS Spline search was approximately $30,000$. This is the accumulation of conducting training and validation on half of the ImageNet22K dataset for all the configurations corresponding to the initial $2 * 3 + 3 = 9$ points and additional 3 data points. Each point measurement took about $2,500$ GPU hours. For reference, the original reinforcement learning based NAS (Zoph and Le 2017) method would require a minimum of $539 \times 2,500 = 13.47M$ GPU hours.

The final recommended BL-ResNext50 architecture was trained and evaluated on the full ImageNet22K dataset using the Summit supercomputer over 34 nodes with 204 Volta GPUs with a global batch size of $6,528$ using Pytorch distributed data parallel. Figure 1(f) shows how the top-1 accuracy climbed as the learning progressed. In Table 3 the comparison of our result against previously published results on ImageNet22K as well as a baseline SME designed architecture based on the BL-ResNext are summarized. The SME designed architecture was a BL-ResNext 101 based model in comparison to the BL-ResNext 50 based Spline recommended model. As can be seen, the Spline recommended architecture resulted in a jump of 0.33% increase in top-1 accuracy. This is the first published result which has crossed 40% in overall top-1 accuracy on ImageNet22K.

Conclusions

We described a novel NAS method based on polyharmonic splines that can perform search directly on large scale datasets. The number of evaluations required during the search phase of our NAS approach is proportional to the number of operations in the search space, not in the number of possible values they each operation could have, making the approach tractable at large scale. We demonstrate the effectiveness of our method on the ImageNet22K benchmark, achieving a state of the art top-1 accuracy of 40.03%. This result paves the way to apply polyharmonic spline based NAS to other architectures and operations in networks, potentially also including hyperparameters in the search space.

References

Baker, B.; Gupta, O.; Raskar, R.; and Naik, N. 2018. Accelerating neural architecture search using performance prediction. In *ICLR Workshops*.

Bender, G.; Kindermans, P.-J.; Zoph, B.; Vasudevan, V.; and Le, Q. 2018. Understanding and simplifying one-shot architecture search. In *ICML*.

Bergstra, J.; Yamins, D.; and Cox, D. 2013. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. In *International conference on machine learning*, 115–123.

Bhattacharjee, B.; Hill, M.; Wu, H.; Chandakkar, P.; Smith, J.; and Wegman, M. 2017. Distributed learning of deep feature embeddings for visual recognition tasks. In *IBM Journal of R and D*.

Borsos, Z.; Khorlin, A.; and Gesmundo, A. 2019. Transfer nas: Knowledge transfer between search spaces with transformer agents. *arXiv preprint 1906.08102*.

Brock, A.; Lim, T.; Ritchie, J.; and Weston, N. 2018. SMASH: One-shot model architecture search through hypernetworks. In *ICLR*.

Cai, H.; Zhu, L.; and Han, S. 2019. ProxylessNAS: Direct neural architecture search on target task and hardware. In *ICLR*.

Cao, S.; Wang, X.; and Kitani, K. M. 2019. Learnable embedding space for efficient neural architecture compression. *arXiv preprint arXiv:1902.00383*.

Chang, J.; zhang, x.; Guo, Y.; MENG, G.; XIANG, S.; and Pan, C. 2019. Data: Differentiable architecture approximation. In *Advances in Neural Information Processing Systems*.

Chen, C.-F. R.; Fan, Q.; Mallinar, N.; Sercu, T.; and Feris, R. 2019a. Big-little net: An efficient multi-scale feature representation for visual and speech recognition. In *ICLR*.

Chen, Y.; Yang, T.; Zhang, X.; MENG, G.; Xiao, X.; and Sun, J. 2019b. Detnas: Backbone search for object detection. In *Advances in Neural Information Processing Systems*.

Chilimbi, T.; Suzue, Y.; Apacible, J.; and Kalyanaraman, K. 2014. Project adam: Building an efficient and scalable deep learning training system. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 571–582.

Cho, M.; Finkler, U.; Kumar, S.; Kung, D.; Saxena, V.; and Sreedhar, D. 2017. Powerai ddl. In *arXiv preprint 1708.02188*.

Codreanu, V.; Podareanu, D.; and Saletore, V. 2017. Achieving deep learning training in less than 40 minutes on imagenet-1k and best accuracy and training time on imagenet-22k and places-365. https://bit.ly/2VdG5B7.

Deng, J.; Dong, W.; Socher, R.; Li, L.-J.; Li, K.; and Fei-Fei, L. 2009. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR*.

Deng, J.; Berg, A.; Li, k.; and Li, F.-F. 2010. What does classifying more than 10,000 image categories tell us? In *ECCV*.

Domhan, T.; Springenberg, J. T.; and Hutter, F. 2015. Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*.

Dosovitskiy, A.; Beyer, L.; Kolesnikov, A.; Weissenborn, D.; Zhai, X.; Unterthiner, T.; Dehghani, M.; Minderer, M.; Heigold, G.; Gelly, S.; Uszkoreit, J.; and Houlsby, N. 2021. An image is worth 16x16 words: Transformers for image recognition at scale. In *International Conference on Learning Representations*.

Elsken, T.; Metzen, J. H.; and Hutter, F. 2019. Neural architecture search: A survey. *JMLR* 20(55):1–21.

Falkner, S.; Klein, A.; and Hutter, F. 2018. Bohb: Robust and efficient hyperparameter optimization at scale. *arXiv preprint arXiv:1807.01774*.

Gaier, A., and Ha, D. 2019. Weight agnostic neural networks. In *Advances in Neural Information Processing Systems*.

Ghiasi, G.; Lin, T.; and Le, Q. V. 2019. Nas-fpn: Learning scalable feature pyramid architecture for object detection. In *CVPR*.

He, K.; Zhang, X.; Ren, S.; and Sun, J. 2016. Deep residual learning for image recognition. In *CVPR*.

Howard, A.; Sandler, M.; Chu, G.; Chen, L.-C.; Chen, B.; Tan, M.; Wang, W.; Zhu, Y.; Pang, R.; Vasudevan, V.; Le, Q. V.; and Adam, H. 2019. Searching for mobilenetv3. In *ICCV*.

Iske, A., and Arnold, V. I. 2004. *Multiresolution Methods in Scattered Data Modelling*, volume 37. SpringerVerlag.

Kandasamy, K.; Neiswanger, W.; Schneider, J.; Poczos, B.; and Xing, E. P. 2018. Neural architecture search with bayesian optimisation and optimal transport. In *Advances in neural information processing systems*, 2016–2025.

Krizhevsky, A. 2009. Learning multiple layers of features from tiny images. Technical report.

Liu, C.; Zoph, B.; Neumann, M.; Shlens, J.; Hua, W.; Li, L.-J.; Fei-Fei, L.; Yuille, A.; Huang, J.; and Murphy, K. 2018a. Progressive neural architecture search. In *ECCV*.

Liu, H.; Simonyan, K.; Vinyals, O.; Fernando, C.; and Kavukcuoglu, K. 2018b. Hierarchical representations for efficient architecture search. In *ICLR*.

Liu, C.; Chen, L.-C.; Schroff, F.; Adam, H.; Hua, W.; Yuille, A.; and Fei-Fei, L. 2019. Auto-deeplab: Hierarchical neural architecture search for semantic image segmentation. In *CVPR*.

Liu, H.; Simonyan, K.; and Yang, Y. 2018. Darts: Differentiable architecture search. *arXiv preprint 1806.09055*.

Lu, Z.; Sreekumar, G.; Goodman, E.; Banzhaf, W.; Deb, K.; and Boddeti, V. N. 2020. Neural architecture transfer. *arXiv preprint 2005.05859*.

Mendoza, H.; Klein, A.; Feurer, M.; Springenberg, J. T.; and Hutter, F. 2016. Towards automatically-tuned neural networks. In *Workshop on Automatic Machine Learning*, 58–65.

Nayman, N.; Noy, A.; Ridnik, T.; Friedman, I.; Jin, R.; and Zelnik-Manor, L. 2019. Xnas: Neural architecture search with expert advice. In *Advances in Neural Information Processing Systems*.

Panda, R.; Merler, M.; Jaiswal, M.; Wu, H.; Ramakrishnan, K.; Finkler, U.; Chen, C.-F.; Cho, M.; Kung, D.; Feris, R. S.; and Bhattacharjee, B. 2021. Nastransfer: Analyzing architecture transferability in large scale neural architecture search. In *AAAI*.

Pham, H.; Guan, M.; Zoph, B.; Le, Q.; and Dean, J. 2018. Efficient neural architecture search via parameters sharing. In *ICML*.

Real, E.; Aggarwal, A.; Huang, Y.; and Le, Q. V. 2019. Regularized evolution for image classifier architecture search. In *AAAI*.

Swersky, K.; Duvenaud, D.; Snoek, J.; Hutter, F.; and Osborne, M. A. 2014. Raiders of the lost architecture: Kernels for bayesian optimization in conditional parameter spaces. *arXiv preprint arXiv:1409.4011*.

Tan, M., and Le, Q. V. 2019. Efficientnet: Rethinking model scaling for convolutional neural networks. In *ICML*.

Tan, M.; Chen, B.; Pang, R.; Vasudevan, V.; Sandler, M.; Howard, A.; and Le, Q. V. 2019. Mnasnet: Platform-aware neural architecture search for mobile. In *CVPR*.

Wang, N.; Gao, Y.; Chen, H.; Wang, P.; Tian, Z.; and Shen, C. 2019. NAS-FCOS: Fast neural architecture search for object detection. *arXiv preprint 1906.04423*.

Wistuba, M. 2019. Xfernas: Transfer neural architecture search. *arXiv preprint 1907.08307*.

Wu, B.; Dai, X.; Zhang, P.; Wang, Y.; Sun, F.; Wu, Y.; Tian, Y.; Vajda, P.; Jia, Y.; and Keutzer, K. 2019. Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search. In *CVPR*.

Xie, S.; Girshick, R.; Dollár, P.; Tu, Z.; and He, K. 2017a. Aggregated residual transformations for deep neural networks. In *CVPR*.

Xie, S.; Girshick, R.; Dollár, P.; Tu, Z.; and He, K. 2017b. Aggregated residual transformations for deep neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 1492–1500.

Xie, S.; Kirillov, A.; Girshick, R.; and He, K. 2019a. Exploring randomly wired neural networks for image recognition. In *ICCV*.

Xie, S.; Zheng, H.; Liu, C.; and Lin, L. 2019b. SNAS: stochastic neural architecture search. In *ICLR*.

Zela, A.; Elsken, T.; Saikia, T.; Marrakchi, Y.; Brox, T.; and Hutter, F. 2020. Understanding and robustifying differentiable architecture search. In *ICLR*.

Zela, A.; Siems, J.; and Hutter, F. 2020. Nas-bench-1shot1: Benchmarking and dissecting one-shot neural architecture search. In *ICLR*.

Zhang, H.; Hu, Z.; Wei, J.; Xie, P.; Kim, G.; Ho, Q.; and Xing, E. 2015. Poseidon: A system architecture for efficient gpu-based deep learning on multiple machines. In *arXiv preprint 1512.06216*.

Zhang, C.; Ren, M.; and Urtasun, R. 2018. Graph hypernetworks for neural architecture search. *arXiv preprint arXiv:1810.05749*.

Zhong, Z.; Yan, J.; Wu, W.; Shao, J.; and Liu, C.-L. 2018. Practical block-wise neural network architecture generation. In *CVPR*.

Zhou, D.; Zhou, X.; Zhang, W.; Loy, C. C.; Yi, S.; Zhang, X.; and Ouyang, W. 2020. Econas: Finding proxies for economical neural architecture search. In *CVPR*.

Zoph, B., and Le, Q. V. 2017. Neural architecture search with reinforcement learning. In *ICLR*.

Zoph, B.; Vasudevan, V.; Shlens, J.; and Le, Q. V. 2018. Learning transferable architectures for scalable image recognition. In *CVPR*.