EPNAS: Efficient Progressive Neural Architecture Search - Supplementary Materials

Yanqi Zhou*[1]
yanqiz@google.com
Peng Wang[3]
wangpeng54@baidu.com
Sercan Arik*[2]
soarik@google.com
Haonan Yu*[4]
haonanu@gmail.com
Syed Zawad*[5]
szawad@nevada.unr.edu
Feng Yan[5]
fyan@unr.edu
Greg Diamos*[6]
greg@landing.ai

[1] Google Brain
[2] Google Cloud AI
[3] Baidu Research
[4] Horizon Robotics
[5] University of Nevada
[6] Landing AI

Content.

In this supplementary materials, we provide:

1. Discussion of our work related to other efficient gradient-based methods, e.g. *DARTS[6] and YOSO [9].*

2. Setting details of EPNAS for "scale LSTM" and "insert LSTM" (Sec.4.2). We will release our code for reproducing all our results.

3. More details on model resource constraints.

4. Additional experimental details for image recognition and speech recognition, and visualization of the searched architectures from module search, and layer-by-layer search without constraints and with constraints.

1 Related with gradient-based NAS.

Most recently, gradient-based methods, such as DARTS [6] One-Shot NAS [3] or YOSO [9], are relying on a large pre-defined graph, and the gradients can be performed directly over soft-relaxed ops [6] or connections [3, 9]. They can be more efficient than ENAS or EAS. However, we argue that multiple constraints from hardware platform (Eq.(1) in the paper) are commonly non-differentiable, therefore, those NAS algorithms can not directly apply in our cases. In addition, RL based methods contains more randomness, more flexibility of network

* This work is done when the author is at Baidu Research USA.

architectures, which helps avoid over-fitting problems and generalizes better to a new task domain.

In addition our searched models with EPNAS, as shown in Fig. 1 and discussed in Sec. 4, are meaningful and can inspire architecture design, especially when taking platform constraints in the design process, which provides a good starting point and motivates novel architecture design. We think in the future, rather than fully relying on either automatic search or manually empirical design, a better way might be combining the two.

2 Details of scale & insert LSTM

As stated in Sec.3.2, for "scale LSTM", the decoded scale action $\mathbf{a}_s \in \mathbb{R}^{6\times F}$ scales the network. Here F is the number of groups partitioning all filters of the network, and in each group, the first 3 logits (after softmax) determine whether to `Scale up`, `Keep the same` or `Scale down` the filter channel following a predefined order. For example, if the set of available channel sizes is $[12,24,36,48]$, and one filter in the group has channel size of 24, the `Scale up` action will change its filter channel from 24 to 36 without moving out of bounds, and vice versa with `Scale down` action. The last 3 logits are used in the same way for `Scale up`, `Keep the same` or `Scale down` the filter size following a predefined order in each group. This mixture of actions can provide much more flexible morphing of networks, helping us to discover networks satisfying our requirements.

For "insert LSTM", the decoded insert action $\mathbf{a}_i \in \mathbb{R}^{3+2N_l+N_{op}+N_{cz}+N_{ac}+N_{fw}+2}$, where the first 3 logits determine the action of `insert`, `keep` or `remove`, and the following N_l logits determine the place to perform the operation when it is `insert` or `remove`, *i.e.* "Src1" as indicated in the paper. Then, the following N_l logits determine the place l of skip connection, *i.e.* "Src2" as indicated in the paper. Here N_l is the maximum number of layers. In order to generate a layer number dependent discrete distribution, we adopt `tf.nn.embeddinglookup` operation to select $L+5$ logits out of N_l to perform softmax, where L is the number of layers. After obtaining inserting place l, we then select l logits from N_l to find the feature from skip connection. Note setting "Src2" to l means no skip connection is needed. The next N_{op} logits determine the type of operation to have, and the coming N_{cz}, N_{ac} and N_{fw} determine the selected channel size, type of activation and filter width of the operation. The final 2 logits determine whether to `add` or `concatenate` the features from "Src1" and "Src2".

3 Model resource constraints

In this paper, we select three resource constraints: model size, compute complexity, and compute intensity as they can capture important hardware performance of an algorithm. We introduce the resource metrics in more details below:

(i) **Model size** is quantified by the total amount of memory used by the model parameters. For a given neural network, model size depends on the dimensions of the weight tensors, and the precision of each tensor. In this paper, we fix the precision of weights (to 4 bytes) and focus only on the tensor sizes. Reducing the model size may involve reducing the input resolution (e.g., decreasing the number of frequency channels in spectral representation), removing layers, reducing the number of hidden units (e.g., for recurrent cells), or reducing the number of filters (e.g., for convolutions). Small model size can also be obtained with

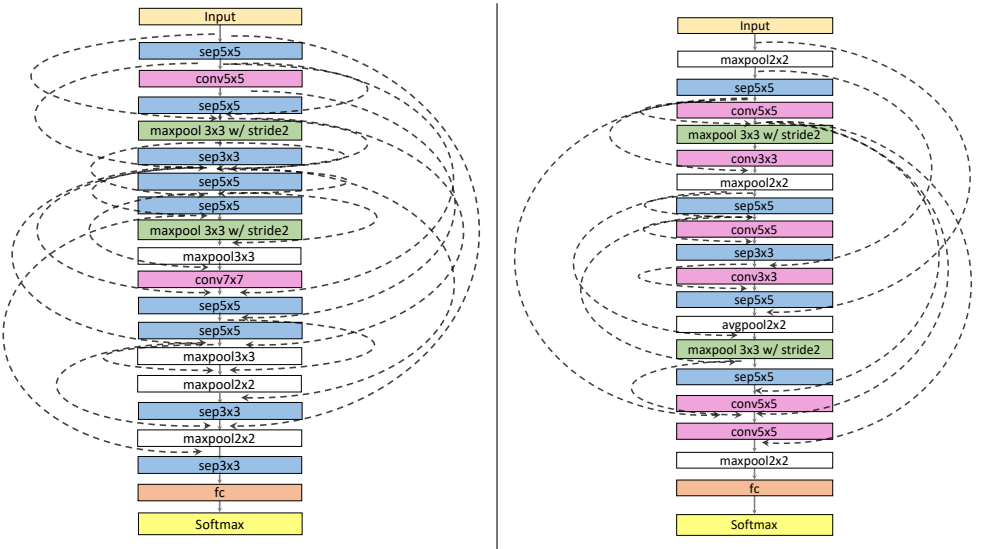

Figure 1: Models searched by EPNAS. (a) Searched without constraints. (b) Searched with constraints, *i.e. Model size* $\leq 3M$, *Compute complexity* ≤ 80, *Compute intensity* ≥ 80.

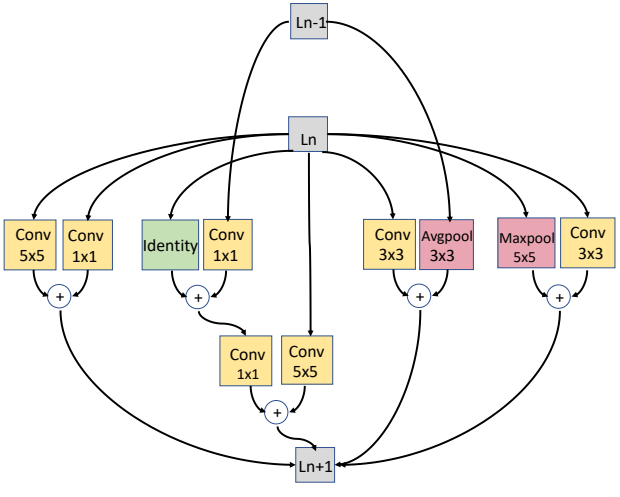

Figure 2: One of the best cells searched with module search.

more parameter sharing (e.g., depthwise-separable convolutions with short filter sizes) and repetitive computations (e.g., recurrent layers with long sequence lengths and small number of hidden units).

(ii) **Compute complexity** is quantified by the total number of floating-point operations (FLOPs). [1] For most operations used in this paper, Tensorflow profiling tool [2] includes

[1] All point-wise operations such as nonlinearities are counted as 1 FLOP, which is motivated by the trend of implementing them as a single instruction. Complexities of register memory-move operations are ignored. A matrix-matrix multiplication, between W, an $m \times n$ matrix and X, an $n \times p$ matrix is assumed to have a complexity of $2mnp$ FLOPs. Similar expression can be generalized for multi-dimensional tensors, that are used in convolutional layers.

Table 1: Search space of scale and insert actions in layer-by-layer search for keyword spotting.

Feature	Search space
Layer type	[conv2d, dep-sep-conv2d, dilated-conv2d, GRU, AvgPool2d, FC]
Number of Layers	[1, 2, 3, 4, 5]
Kernel size in time	[1, 4, 8, 16, 20]
Kernel size in frequency	[1, 2, 4, 8, 10]
Channel size (or hidden units)	[4, 12, 16, 32, 64, 128, 192, 256]
Stride in time	[1, 2, 4, 8, 10]
Stride in frequency (or dilation rate)	[1, 2, 3, 4, 5]
Number of GRU directions	[1, 2]
Dropout rate	[0.8, 0.9, 1.0]
Src1 Layer	[i for i in range(MAX_LAYERS)]
Src2 Layer	[i for i in range(MAX_LAYERS)]

FLOP counts, which we directly used. Straightforward approaches that reduce the inference complexity are mostly similar to the approaches that reduce the model size, such as reducing the number of hidden units or the number of filters. In general, reduction of complexity encourages models with minimal redundancy (e.g., by joining concatenated linear operations). (iii) **Compute intensity** is defined as the average number of FLOPs per data access [2]. Compute intensity is a measure of how efficiently an algorithm can reuse data. For modern multicore architectures like GPUs and TPUs [4], it reflects the runtime of an algorithm. In general, if an algorithm has higher data reuse during computation, it requires less memory bandwidth and achieves higher compute intensity. High compute intensity encourages neural networks with more locality and more parallelism. As a simple example, consider matrix-matrix multiplication of an $m \times n$ matrix and an $n \times p$ matrix. The compute intensity would be proportional to $\frac{mnp}{mn+np} = \frac{1}{1/p+1/m}$. Increasing it would favor for increases in p and m. If there is a constraint on their sum, due to the total model size or overfitting considerations, higher compute intensity would favor for p and m values close to each other. One example of a high compute intensity operation is multi-dimensional convolution with appropriately large channel sizes. On the other hand, recurrent layers used in typical language or speech processing applications, or some recently-popular techniques like multi-branch networks [8], yield low compute intensity.

In order to impose these constraints and let EPNAS learn various actions such as reducing the model size or increasing the compute intensity, we need to increase model search space by adding more flexibility in choosing kernel size, channel size, etc. (unlike AMC, EAS or ENAS). For example, EPNAS supports $9.8 \cdot 10^{36}$ possible networks while ENAS has a search space of $1.6 \cdot 10^{29}$ possible networks for CIFAR10 with the same number of layers. In the following sections, we elaborate our architecture generation policy for handling larger search space.

For real-valued fast Fourier transform (FFT), the complexity is assumed as $2.5Nlog_2(N)$ FLOPs for a vector of length N [1].

[2] Note that our definition differs from [4] as they model the compute intensity per weight access, ignoring input and output data.

4 Details of experiments

First, the plot of Fig.4 in the paper is based on **5 time experiments**, and the mean accuracy of the experiments are plotted.

Second, we set F for the "Scale LSTM" to be 4 in all our experiments. To determine F, we use the validation set of CIFAR10, and choose a minimum partition that can generate better performance than SOTA results, *i.e.*ENAS.

Third, each *conv2d* layer is composed in the order of activation, convolution and batch-normalization.

Table 2: Comparison of EPNAS with multiple resource constraints vs. the state-of-the-art models on KWS. Here, sz means Model size (M). $c.i.$ means compute intensity (FLOPs/byte) and $c.c.$ means compute complexity (MFLOPs). Conv2d (2-D convolution) and DS-Conv2d (2-D depth-separable convolution) are parametrized by the number of layers, channel size, kernel size in time and frequency, and stride in time and frequency, respectively. GRU is parametrized by the number of layers, number of hidden units, and the number of directions. FC (fully connected) is parametrized by number of layers and number of hidden units. AvgPool2d (2-D average pooling) is parametrized by pooling in time and frequency.

Model	Resource constraints	Architecture	Test accuracy (%)	Model size (sz) (Million)	Comp. intensity ($c.i.$) (FLOPs/byte)	Comp. complexity ($c.c.$) (MFLOPs)
DS-CNN []	-	Conv2d(1,64,10,4,2,2) DS-Conv2d (4,64,3,3,1,1) AvgPool2d	93.39	0.023	6.07	1.76
CRNN []	-	Conv2d (1,100,10,4,2,1) GRU (2,136,1) FC (1,188)	94.40	2.447	46.21	15.76
EPNAS: Layer-by-Layer Search	-	DS-Conv2d (1,4,4,1,1,1) GRU (1,64,1) GRU (1,128,1) Conv2d (1,12,16,2,4,4) Conv2d (1,4,16,4,4,4) Conv2d (1,64,16,4,4,4) FC (1,32)	95.81	0.143	3.39	3.58
EPNAS: Layer-by-Layer Search	$sz \leq 0.05$ M	GRU (2,64,1)	0.047	94.04	1.40	3.69
EPNAS: Layer-by-Layer Search	$sz \leq 0.1$ M	Conv2d (3,32,4,8,1,3) AvgPool2d	94.82	0.067M	6.53	8.11
EPNAS: Layer-by-Layer Search	$c.c. \leq 1$	GRU (3,32,1) FC (1,256)	0.425M	93.16	0.89	2.45
EPNAS: Layer-by-Layer Search	$c.c. \leq 5$	GRU (5,64,1) FC (2,16)	95.02	0.171M	3.30	6.38
EPNAS: Layer-by-Layer Search	$c.i. \geq 10$	GRU (3,128,2)	0.733M	95.64	13.59	21.83
EPNAS: Layer-by-Layer Search	$c.i. \geq 50$	Conv2d (3,192,8,4,1,3) AvgPool2d (8,1) FC (2,16)	95.18	2.626M	210.13	58.70
EPNAS: Layer-by-Layer Search	$sz \leq 0.1$ M & $c.i. \geq 10$	Conv2d (2,32,20,2,1,2) GRU (3,16,1) GRU (2,12,1) Conv2d (2,4,20,8,1,2)	93.65	0.074M	12.57	10.29
EPNAS: Layer-by-Layer Search	$sz \leq 0.1$ M & $c.c. \leq 1$	GRU (2,32,2)	93.07	0.035M	1.00	2.77

CIFAR10. In Fig. 1(a), we visualize searched models searched without constraints (last row (EPNAS Layer-by-Layer Search w prediction) in Tab.3), which achieves 3.02% test error. This is better than 3.54% of ENAS [7] and 3.63% of PNAS [5]. Compare to the model searched by ENAS, our network is deeper, has more variation of operations, and could embrace larger filter size.

In Fig. 1(b), we visualize a model searched with multiple constraints, *i.e.* $Model\ size \leq 3M, Compute\ complexity \leq 80, Compute\ intensity \geq 80$. It achieves 3.7% test error, which is slightly worse than our best performing model but meets the required conditions. We can see the model has early down sampling of input image, which is a common operation for reducing model size. In addition, it has sparse while long range connections, indicating most of the short range connection, such as that in DenseNet, might be less useful than long range connection in achieving good performance.

Key word spotting (KWS). In Tab. 1, we list the search space used for KWS dataset, and in Tab. 2, we extend the Tab. 5 in our paper with more experiments and with the architecture searched with various constrains. Here we didn't show the skip connection inside, but majorly show the discovered operations in the network.

Due to the relative small search space we need to explore for this dataset, it is easier for us to clearly compare the generated model with and without constraints.

We can see *GRU* provides good performance with small model size, while has low compute intensity. *Conv2d* has high compute intensity while requires relatively large model size. EPNAS automatically adjusts the configurations based on given constraints or the importance of different constraints without laborious manually tuning.

4.1 Keyword Spotting

We consider layer-by-layer search for KWS. The initial model is chosen as a small and simple architecture, i.e., a single fully connected layer with 12 hidden units (which yields a test accuracy of 65%). The policy network is trained with Adam optimizer with a learning rate of 0.0006. An episode size of 5 and a batch size of 10 is used for all experiments, i.e. 10 models are trained in different branches. Every episode the best 10 models are selected. The weights of the controller are initialized uniformly in [-0.1, 0.1]. The size of LSTMs for network embedding and the controllers are similar to those of the image classification task.

EPNAS can efficiently generate models that meet both constraints after about 120 searched models whereas Random Search is not able to generate any models meeting both constraints within 400 searched models. EPNAS attempts to maximize the model performance in this domain and finally finds an architecture with 93.65% test accuracy that meets both resource constraints. Random Search can barely find a model that violate the constraints by a small margin (model size = 0.13 M and compute intensity = 10.69 FLOPs/byte).

Table 3 presents the results of EPNAS for KWS along with the optimal architectures. Without any resource constraints, a SOTA performance of 95.81%, can be obtained using an architecture composed of depth-separable convolutions (that apply significant downsampling), followed by GRUs and multiple 2-D convolutions. When aggressive resource constraints are imposed, we observe that EPNAS finds architectures that outperform hand-optimized SOTA architectures. A tight model size constraint results in an architecture composed of GRUs with small hidden units. Similarly, tight constraints on computational complexity also yields GRUs with small hidden units. Both cases disfavor convolutions with

Table 3: Comparison of EPNAS with multiple resource constraints vs. the state-of-the-art models on KWS. Here, *sz* means Model size (M). *c.i.* means compute intensity (FLOPs/byte) and *c.c.* means compute complexity (MFLOPs).

Model	Resource constraints	Test accuracy (%)	Model size (*sz*)	Comp. intensity (*c.i.*) (FLOPs/byte)	Comp. complexity (*c.c.*) (MFLOPs)
DS-CNN [10]	-	93.39	0.023M	6.07	1.76
CRNN [10]	-	94.40	2.447M	46.21	15.76
EPNAS: Layer-by-Layer Search	-	95.81	0.143M	3.39	3.58
EPNAS: Layer-by-Layer Search	$sz \leq 0.1$ M	94.82	0.067M	6.53	8.11
EPNAS: Layer-by-Layer Search	$c.c. \leq 5$	95.02	0.171M	3.30	6.38
EPNAS: Layer-by-Layer Search	$c.i. \geq 50$	95.18	2.626M	210.13	58.70
EPNAS: Layer-by-Layer Search	$sz \leq 0.1$ M & $c.i. \geq 10$	93.65	0.074M	12.57	10.29
EPNAS: Layer-by-Layer Search	$sz \leq 0.1$ M & $c.c. \leq 1$	93.07	0.035M	1.00	2.77

large filter sizes and high number of filters, which are commonly observed for nonlinear downsampling. When compute intensity is considered, an efficient architecture is achieved by enabling most of the computation on 2-D convolutions with large channel sizes. Lastly, we consider joint constraints, and we observe that very competitive accuracy results can be obtained even in the regime of a small feasible architecture space. For example, EPNAS finds models under 0.1M parameters with high compute intensity (>10 FLOPs/Byte) with 93.65% test accuracy.

References

[1] Fft benchmark methodology. http://www.fftw.org/speed/method.html.

[2] Tensorflow profiler and advisor. https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/profiler/README.md.

[3] 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.

[4] Norman P. Jouppi et al. In-datacenter performance analysis of a tensor processing unit. *SIGARCH Comput. Archit. News*, 45(2):1–12, June 2017.

[5] Chenxi Liu et al. Progressive neural architecture search. In *ECCV*, pages 19–34, 2018.

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

[7] Hieu Pham et al. Efficient neural architecture search via parameter sharing. *arXiv preprint arXiv:1802.03268*, 2018.

[8] Christian Szegedy et al. Rethinking the inception architecture for computer vision. *CoRR*, abs/1512.00567, 2015. URL http://arxiv.org/abs/1512.00567.

[9] Xinbang Zhang, Zehao Huang, and Naiyan Wang. You only search once: Single shot neural architecture search via direct sparse optimization. *arXiv preprint arXiv:1811.01567*, 2018.

[10] Yundong Zhang, Naveen Suda, Liangzhen Lai, and Vikas Chandra. Hello edge: Keyword spotting on microcontrollers. *arXiv preprint arXiv:1711.07128*, 2017.