MaskConnect: Connectivity Learning by Gradient Descent

Karim Ahmed and Lorenzo Torresani

Department of Computer Science, Dartmouth College
karim@cs.dartmouth.edu, LT@dartmouth.edu

Abstract. Although deep networks have recently emerged as the model of choice for many computer vision problems, in order to yield good results they often require time-consuming architecture search. To combat the complexity of design choices, prior work has adopted the principle of modularized design which consists in defining the network in terms of a composition of topologically identical or similar building blocks (a.k.a. modules). This reduces architecture search to the problem of determining the number of modules to compose and how to connect such modules. Again, for reasons of design complexity and training cost, previous approaches have relied on simple rules of connectivity, e.g., connecting each module to only the immediately preceding module or perhaps to all of the previous ones. Such simple connectivity rules are unlikely to yield the optimal architecture for the given problem.
In this work we remove these predefined choices and propose an algorithm to learn the connections between modules in the network. Instead of being chosen a priori by the human designer, the connectivity is learned simultaneously with the weights of the network by optimizing the loss function of the end task using a modified version of gradient descent. We demonstrate our connectivity learning method on the problem of multi-class image classification using two popular architectures: ResNet and ResNeXt. Experiments on four different datasets show that connectivity learning using our approach yields consistently higher accuracy compared to relying on traditional predefined rules of connectivity. Furthermore, in certain settings it leads to significant savings in number of parameters.

Keywords: Connectivity Learning, Image Categorization

1 Introduction

Deep neural networks have emerged as one of the most prominent models for problems that require the learning of complex functions and that involve large amounts of training data. While deep learning has recently enabled dramatic performance improvements in many application domains, the design of deep architectures is still a challenging and time-consuming endeavor. The difficulty lies in the many architecture choices that impact—often significantly—the performance of the system. In the specific domain of image categorization, which is the focus of this paper, significant research effort has been invested in the empirical study

of how depth, filter sizes, number of feature maps, and choice of nonlinearities affect performance [17,27,36,30,48,42]. Recently, several authors have proposed to simplify the architecture design by defining convolutional neural networks (CNNs) in terms of composition of topologically identical or similar building blocks or *modules*. This strategy was arguably first popularized by the VGG nets [38] which were built by stacking a series of convolutional layers having identical filter size (3×3). Other examples are ResNets [23] which are constructed by stacking residual blocks of fixed topology, ResNeXt models [47] which use multi-branch residual block modules, DenseNets [25] which use dense blocks as building blocks, or Multi-Fiber networks [7] which use parallel branches ("fibers") connected by routers ("transistors").

While the principle of modularized design has greatly simplified the challenge of building effective architectures for image analysis, the choice of how to combine and aggregate the computations of these building blocks still rests on the shoulders of the human designer. To avoid a combinatorial explosion of options, prior work has relied on simple, uniform rules of aggregation and composition. For example, in ResNets and DenseNets each building block is connected only to the preceding one, via identity mapping, convolution or pooling. ResNeXt models [47] use a set of simplifying assumptions: the branching factor C (also referred to as *cardinality*) is fixed to the same constant in all layers of the network, all branches of a module are fed the same input, and the outputs of parallel branches are aggregated by a simple additive operation that provides the input to the next module. While these simple rules of connectivity render network design more manageable, they are unlikely to yield the optimal connectivity for the given problem.

In this paper we remove these predefined choices and propose an algorithm that learns to combine and aggregate building blocks of a neural network by directly optimizing connectivity of modules with respect to the given task. In this new regime, the network connectivity naturally arises as a result of training rather than being hand-defined by the human designer. While in principle this involves a search over an exponential number of connectivity configurations, our method can efficiently optimize the training loss with respect to connectivity using a variant of backpropagation. This is achieved by means of *connectivity masks*, i.e., learned binary parameters that act as "switches" determining the final connectivity in our network. The masks are learned together with the convolutional weights of the network, as part of a joint optimization with respect to the given loss function for the problem.

We evaluate our method on the problem of multi-class image classification using two popular modular architectures: ResNet and ResNeXt. We demonstrate that models with our learned connectivity consistently outperform the networks based on predefined rules of connectivity for the same budget of residual blocks (and parameters). An interesting byproduct of our approach is that, in certain settings, it can automatically identify modules that are superfluous, i.e., unnecessary or detrimental for the end objective. At the end of the optimization, these unused modules can be pruned away without impacting the learned hypothesis while reducing substantially the runtime and the number of parameters to store.

By recasting the training procedure as an optimization over learning weights *and* connectivity, our method effectively searches over a larger space of solutions. This yields networks achieving higher accuracy than those constrained to use predefined connectivities. The average training time overhead is moderate, ranging between 13% (for ResNet models) and 39% (for ResNeXt models) compared to learning using fixed connectivity which, however, yields lower accuracy. Finally we point out that, although our experiments are carried out using ResNet and RexNeXt models, our approach is general and applicable without major modifications to other forms of network architectures and other tasks beyond image categorization. In principle our method can also be used to learn connectivity among layers of a traditional (i.e., non-modular) neural network or a CNN. However, modern networks typically include a very large number of layers (hundreds or even thousands [24]), which would make our approach very costly. Learning connectivity among modules is more manageable as each module encapsulates many layers and thus the total number of modules is typically small even for deep networks.

2 Related Work

Despite their wide adoption, deep networks often require laborious model search in order to yield good results. As a result, significant research effort has been devoted to the design of algorithms for automatic model selection. However, most of this prior work falls within the genre of hyper-parameter optimization [6,39,40] rather than architecture or connectivity learning. Evolutionary search has been proposed as an interesting framework to learn both the structure as well as the connections in a neural network [31,41,34,29,46,44,15,33,14]. Architecture search has also been recently formulated as a reinforcement learning problem with impressive results [49]. Several authors have proposed learning connectivity by pruning unimportant weights from the network [28,19,21,18,20]. However, these prior methods operate in stages where initially a network with full connectivity is learned and then connections are greedily removed according to an importance criterion. Compare to all these prior approaches, our work provides the advantage of learning the connectivity by direct global optimization of the loss function of the problem at hand rather than by greedy optimization of an auxiliary proxy criterion or by costly evolutionary search. Our technical approach shares similarities with the "Shake-Shake" regularization [16]. This procedure was demonstrated on two-branch ResNeXt models and consists in randomly scaling tensors produced by parallel branches during training while at test time the network uses uniform weighting of tensors. Conversely, our algorithm *learns* an optimal binary scaling of the parallel tensors with respect to the training objective and uses the resulting network with sparse connectivity at test time. While our algorithm is limited to optimizing the connectivity structure within a predefined architecture, Adams et al. [1] proposed a nonparametric Bayesian approach that searches over an infinite network using MCMC. Our approach can be viewed as a middle ground between two extremes: using hand-defined networks versus learning/searching the full

architecture from scratch. The advantage is that our connectivity learning can be done without adding a significant training time overhead (only 13-39% depending on the architecture) compared to using fixed connectivity. The disadvantage is that the space of models considered by our approach is a lot more constrained than in the case of general architecture search. Saxena and Verbeek [35] introduced convolutional neural fabric which are learnable 3D trellises that locally connect response maps at different layers of a CNN. Similarly to our work, they enable optimization over an exponentially large family of connectivities, albeit different from those considered here. Finally, our approach is also related to conditional computation methods [45,5,3,4,37,10,13,12,8,2], which learn to drop out blocks of units. However, unlike these techniques, our algorithm learns a fixed, sparse connectivity that does *not* change with the input and thus it keeps the runtime cost and the number of used parameters constant.

3 Technical Approach

3.1 Modular architecture

We begin by defining the modular architecture that will be used by our framework. In order to present our method in its full generality, we will describe it in the context of a *general* modular architecture, which we will then instantiate in the form of the two models used in our experiments (ResNet and ResNeXt).

We assume that the general modular architecture consists of a stack of L modules. (When using ResNet the modules will be residual blocks, while for ResNeXt each module will consist of multiple parallel branches.) We denote with $\mathbf{x}_j$ the input to the j-th module for $j = 1, \ldots, L$. The input of each module is an activation tensor computed from one the previous modules. We assume that the module implements a function $\mathcal{G}(.)$ parameterized by learnable weights θ_j. The weights may for example represent the coefficients of convolutional filters. Thus, the output $\mathbf{y}_j$ computed by the j-th module is given by $\mathbf{y}_j = \mathcal{G}(\mathbf{x}_j; \theta_j)$. In prior modular architectures, such as ResNet, ResNeXt and DenseNet, the connectivity between modules is hand-defined a priori according to a very simple rule: the input of a module is the output of the preceding module. In other words, $\mathbf{x}_j \leftarrow \mathbf{y}_{j-1}$. While this makes network design straightforward, it greatly limits the topology of architectures considered for the given task. In the next subsection we describe how to parameterize the architecture to remove these constraints and to enable connectivity learning in modular networks.

3.2 Masked architecture

We now introduce learnable *masks* defining the connectivity in the network. Specifically, we want to allow each module j to take input from one or more of the preceding modules $k = 1, \ldots, j-1$. To achieve this we define for each module a binary mask vector that controls the input pathway of that module. The binary mask vectors are learned jointly with the weights of the network. Let

$\mathbf{m}_j = [m_{j,1}, m_{j,2}, \ldots, m_{j,j-1}]^\top \in \{0,1\}^{j-1}$ be the binary mask vector defining the *active* input connections feeding the j-th module. If $m_{j,k} = 1$, then the activation volume produced by the k-th module is fed as input to the j-th module. If $m_{j,k} = 0$, then the output from the k-th module is ignored by the j-th module. The tensors from the *active* input connections are all added together (in an element-wise fashion) to form the input to the module. Thus, if we denote again with $\mathbf{y}_k$ the output activation tensor computed by the k-th module, the input $\mathbf{x}_j$ to the j-th module will be given by the following equation:

$$\mathbf{x}_j = \sum_{k=1}^{j-1} m_{j,k} \cdot \mathbf{y}_k \tag{1}$$

Then, the output of this module will be obtained through the usual computation, i.e., $\mathbf{y}_j = \mathcal{G}(\mathbf{x}_j; \theta_j)$. We note that under this model we no longer have predefined connectivity among modules. Instead, the mask $\mathbf{m}_j$ now determines *selectively* for each module which outputs from the previous modules will be aggregated and form the input to the block. In this paper we constrain the aggregations of outputs from the active connections to be in the form of simple additions as this does not require new parameters. When different modules yield feature maps of different sizes, we use zero-padding shortcuts to increase the dimensions of feature tensors to the largest size (as in [23]). These shortcuts are parameter free. We leave to future work the investigation of more sophisticated, parameterized aggregation schemes.

We point out that depending on the constraints defined over $\mathbf{m}_j$, different interesting models can be realized. For example, by introducing the constraint that $\sum_k m_{j,k} = 1$ for each block j, then each module will receive input from only one of the preceding modules (since each $m_{j,k}$ must be either 0 or 1). At the other end of the spectrum, if we set $m_{j,k} = 1$ for all modules j, k, then all connections would be active. In our experiments we will demonstrate that the best results are typically achieved for values in between these two extremes, i.e., by connecting each module to K previous modules where K is an integer-valued hyperparameter such that $1 < K < (j-1)$. We refer to this hyperparameter as the *fan-in* of a module. As discussed in the next section, the mask vector $\mathbf{m}_j$ for each block is learned simultaneously with all the other weights in the network via backpropagation. Finally, we note that it may be possible for a module in the network to become unused. This happens when, as a result of the optimization, module k is such that $m_{j,k} = 0$ for all j. In this case, at the end of the optimization, we prune the module in order to reduce the number of parameters to store and to speed up inference (note that this does not affect the function computed by the network). In the next subsection we discuss our method for jointly learning the weights and the masks in the network.

3.3 MaskConnect: learning to connect

We refer to our learning algorithm as MaskConnect. It performs joint optimization of a given learning objective ℓ with respect to both the weights of the network

(θ) as well as the masks ($\mathbf{m}$). Since in this paper we apply our method to the problem of image categorization, we use the traditional multi-class cross-entropy objective for the loss ℓ. However, our approach can be applied without change to other loss functions and other tasks benefitting from connectivity learning.

In MaskConnect the weights have real values, as in traditional networks, while the masks have binary values. This renders the optimization challenging. To learn these binary parameters, we adopt a modified version of backpropagation, inspired by the algorithm proposed by Courbariaux et al. [9] to train neural networks with binary weights. During training we store and update a real-valued version $\tilde{\mathbf{m}}_j \in [0,1]^{j-1}$ of the masks, with entries clipped to lie between 0 and 1.

In general, the training via backpropagation consists of three steps: 1) forward propagation, 2) backward propagation, and 3) parameters update. At each iteration, we stochastically binarize the real-valued masks into binary-valued vectors $\mathbf{m}_j \in \{0,1\}^{j-1}$ which are then used for the forward propagation and backward propagation (steps 1 and 2). Instead, during the parameters update (step 3), the method updates the real-valued masks $\tilde{\mathbf{m}}_j$. The weights θ of the convolutional and fully connected layers are optimized using standard backpropagation. We discuss below the details of our mask training procedure, under the constraint that at any time there can be only K active entries in the binary mask $\mathbf{m}_j$, where K is a predefined integer hyperparameter with $1 \leq K \leq j-1$. In other words, we impose the following constraints:

$$m_{j,k} \in \{0,1\}\ \forall j,k, \quad \text{and} \quad \sum_{k=1}^{j-1} m_{j,k} = K\ \forall j.$$

These constraints imply that each module receives input from exactly K previous modules.

Forward Propagation. During the forward propagation, our algorithm first normalizes the real-valued entries in the mask of each block j to sum up to 1, such that $\sum_{k=1}^{j-1} \tilde{m}_{j,k} = 1$. This is done so that $\text{Mult}(\tilde{m}_{j,1}, \tilde{m}_{j,2}, \ldots, \tilde{m}_{j,j-1})$ defines a proper multinomial distribution over the $j-1$ possible input connections into module j. Then, the binary mask $\mathbf{m}_j$ is stochastically generated by drawing K *distinct* samples $a_1, a_2, \ldots, a_K \in \{1, \ldots, (j-1)\}$ from the multinomial distribution over the connections. Finally, the entries corresponding to the K samples are activated in the binary mask vector, i.e., $m_{j,a_k} \leftarrow 1$, for $k = 1, ..., K$. The input activation volume to the module j is then computed according to Eq. 1 from the sampled binary masks. We note that the sampling from the Multinomial distribution ensures that the connections with largest $\tilde{m}_{j,k}$ values will be more likely to be chosen, while at the same time the stochasticity of this process allows different connectivities to be explored, particularly during early stages of the learning when the real-valued masks still have fairly uniform distributions.

Backward Propagation. In the backward propagation step, the gradient $\partial\ell/\partial y_k$ with respect to each output is obtained via back-propagation from $\partial\ell/\partial x_j$ and the binary masks $m_{j,k}$.

Mask Update. In the parameter update step our algorithm computes the gradient with respect to the binary masks for each module. Then, using these computed gradients and the given learning rate, it updates the real-valued masks via gradient descent. At this time we clip the updated real-valued masks to constrain them to remain within the valid interval $[0, 1]$ (as in [9]).

Pseudocode for our training procedure is given in Appendix A. After joint training over θ and $\mathbf{m}$, we have found beneficial to (1) freeze the binary masks to the top-K values for each mask (i.e., by setting as active connections in $\mathbf{m}_j$ those corresponding to the K largest values in $\tilde{\mathbf{m}}_j$) and then (2) fine-tune the weights θ of the network with respect to these fixed binary masks.

In the next subsections we discuss how we instantiated our general approach for the two architectures considered in our experiments: ResNet and ResNeXt.

3.4 MaskConnect applied to ResNet

The application of our algorithm to ResNets is quite straightforward. ResNets are modular networks obtained by stacking residual blocks. A residual block implements a residual function $\mathcal{F}(.)$ with reference to the layer input. Figure 1(a)(left) illustrates an example of these modular components where the 3 layers in the block implement the residual function $\mathcal{F}(\mathbf{x}; \theta)$. A shortcut connections adds the residual block output $\mathcal{F}(\mathbf{x})$ to its input $\mathbf{x}$. Thus the complete function $\mathcal{G}(.)$ implemented by a residual block computes $\mathcal{G}(\mathbf{x}; \theta) = \mathcal{F}(\mathbf{x}; \theta) + \mathbf{x}$. The ResNets originally introduced in [23] use a hand-defined connectivity that passes the output of a block to the immediately subsequent block, i.e., $\mathbf{x}_{j+1} \leftarrow \mathcal{F}(\mathbf{x}_j; \theta_j) + \mathbf{x}_j$. Here we propose to use MaskConnect to learn the input connections for each individual residual block in the network. This changes the input provided to block $j + 1$ in the network to be $\mathbf{x}_{j+1} \leftarrow \sum_{k=1}^{j} m_{j+1,k} \left[\mathcal{F}(\mathbf{x}_k; \theta_k) + \mathbf{x}_k\right]$, where binary parameters $m_{j+1,k}$ are learned automatically by our approach simultaneously with the weights θ subject to the constraint that $\sum_{k=1}^{j} m_{j+1,k} = K$. This implies that under our model each residual block now receives input from exactly K out of the preceding blocks. The output tensors from the K selected blocks are aggregated using element-wise addition and passed as input to the module. Our experiments present results for varying values of fan-in hyperparameter K, which controls the density of connectivity.

3.5 MaskConnect applied to multi-branch ResNeXt

The adaptation of MaskConnect to ResNeXt architectures is slightly more complex, as ResNeXt is based on a multi-branch topology. ResNeXt was motivated by the observation that it is beneficial to arrange residual blocks not only along the depth dimension but also to implement parallel multiple threads of computation feeding from the same input layer. The outputs of the parallel residual blocks are then summed up together with the original input and passed on to the next module. The resulting multi-branch module is illustrated in Figure 1(b)(left). More formally, let $\mathcal{F}(\mathbf{x}; \theta_j^{(i)})$ be the transformation implemented by the j-th

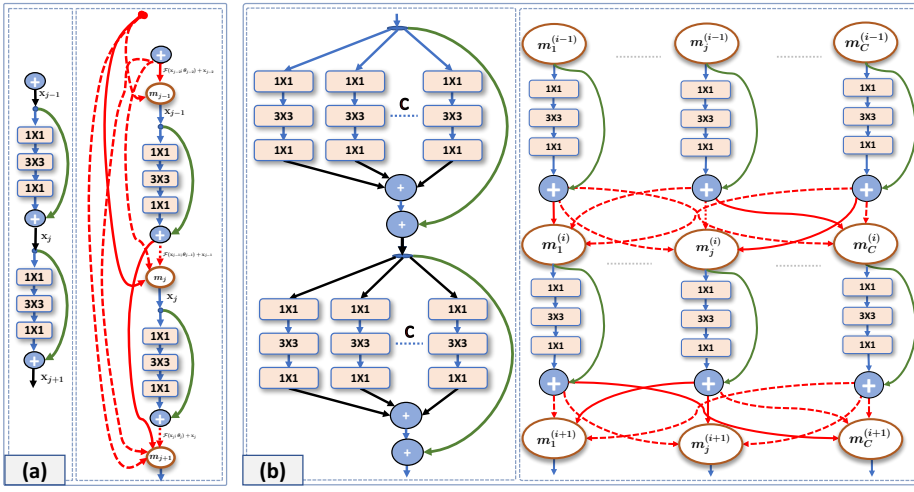

Fig. 1: Application of MaskConnect to two forms of modular network: **(a)** ResNet [22] and **(b)** multi-branch ResNeXt [47]. In traditional ResNet (a)(left) the connections between blocks are fixed (black links) so that each block receives input from only the preceding block. Our approach (a)(right) learns the optimal input connections (solid red links) for each individual block from a collection of potential connections (solid and dotted red links). Similarly, in traditional ResNeXt (b)(left) each module consists of C parallel residual blocks which are all aggregated and fed to the next module (black links). MaskConnect (b)(right) replaces the fixed aggregation points of RexNeXt with learnable masks $\mathbf{m}$ defining the active input connections (solid red links) for each individual residual block.

residual block in module i-th of the network, where $j = 1, \ldots, C$ and $i = 1, \ldots, L$, with L denoting the total number of modules stacked on top of each other to form the complete network. The hyperparameter C is called the cardinality of the module and defines the number of parallel branches within each module. The hyperparameter L controls the total depth of the network. Then, in traditional ResNeXt, the output of the i-th module is computed as:

$$\mathbf{y}_i = \mathbf{x}_i + \sum_{j=1}^{C} \mathcal{F}(\mathbf{x}_i; \theta_j^{(i)}) \tag{2}$$

In [47] it was experimentally shown that increasing the cardinality C is a more effective way of improving accuracy compared to increasing depth or the number of filters. In other words, given a fixed budget of parameters, ResNeXt nets were shown to consistently outperform single-branch ResNets.

However, in an attempt to ease network design, a couple of restrictive limitations were embedded in the architecture of ResNeXt modules: (1) the C parallel

feature extractors in each module operate on the same input; (2) the number of active branches is constant at all depth levels of the network.

MaskConnect allows us to remove these restrictions without adding any significant burden on the process of manual network design, with the exception of a single additional integer hyperparameter (K) for the entire network. As in ResNeXt, our proposed architecture consists of a stack of L multi-branch modules, each containing C parallel feature extractors. However, differently from ResNeXt, each branch in a module can take a different input. The input pathway of each branch is controlled by a binary mask vector. Let $\mathbf{m}_j^{(i)} = [m_{j,1}^{(i)}, m_{j,2}^{(i)}, \ldots, m_{j,C}^{(i)}]^\top \in \{0,1\}^C$ be the binary mask vector defining the *active* input connections feeding the j-th residual block in module i. We note that under this model we no longer have fixed aggregation nodes summing up *all* outputs computed from a module. Instead, the mask $\mathbf{m}_j^{(i)}$ now determines *selectively* for each block which branches from the previous module will be aggregated to form the input to the next block. Under this new scheme, the parallel branches in a module receive different inputs and as such are likely to yield more diverse features.

As before, different constraints over $\mathbf{m}_j^{(i)}$ will give rise to different forms of architecture. By introducing the constraint that $\sum_k m_{j,k}^{(i)} = 1$ for all blocks j, then each residual block will receive input from only one branch (since each $m_{j,k}^{(i)}$ must be either 0 or 1). If instead we set $m_{j,k}^{(i)} = 1$ for all blocks j, k in each module i, then all connections would be active and we would obtain again the fixed ResNeXt architecture. In our experiments we present results obtained by varying the fan-in hyperparameter K such that $1 < K < C$. We also note that it may be possible for a residual block in the network to become unused, as a result of the optimization over the mask values. Thus, at any point in the network the total number of active parallel threads can be any number smaller than or equal to C. This implies that a variable branching factor is learned adaptively for the different depths in the network.

4 Experiments

We tested our approach on the task of image categorization using two different examples of modularized architecture: ResNet [23] and ResNeXt [47]. We used the following datasets for our evaluation: CIFAR-10 [26], CIFAR-100 [26], Mini-ImageNet [43], as well as the full ImageNet [11]. In this paper we include the results achieved on CIFAR-100 and ImageNet [11], while the results for CIFAR-10 [26] and Mini-ImageNet [43] (showing consistent improvements up to nearly 4% over fixed connectivity) can be found in Appendix A.

4.1 CIFAR-100

CIFAR-100 contains images of size 32x32. It consists of 50,000 training images and 10,000 test images. Each image is labeled as belonging to one of 100 possible classes.

Table 1: CIFAR-100 accuracies achieved by models trained using the connectivity of **ResNet** [22] (Fixed-Prev), a fixed random connectivity (Fixed-Random), and the connectivity learned by our approach (Learned)

Model	Connectivity	Accuracy (%)
ResNet-38	Fixed-Prev, K=1 [22]	68.54
	Fixed-Random, K=10	62.67
	Learned, K=10	**70.40**
ResNet-74	Fixed-Prev, K=1 [22]	70.64
	Fixed-Random, K=15	66.93
	Learned, K=15	**72.81**
ResNet-110	Fixed-Prev, K=1 [22]	71.21
	Fixed-Random, K=20	67.22
	Learned, K=20	**73.15**

CIFAR-100 results based on the *ResNet* architecture.

Effect of fan-in (K). The fan-in hyperparameter (K) defines the number of *active* input connections feeding each residual block. We study the effect of the fan-in on the performance of models built and trained using our proposed approach. We use residual blocks consisting of two 3x3 convolutional layers. We use a model obtained by stacking $L = 18$ residual blocks with total depth of $D = 2 + 2L = 38$ layers. We trained and tested this architecture using different fan-in values: $K = 1, .., 17$. All these models have the same learning capacity as varying K does not affect the number of parameters. The results are shown in Figure 2.

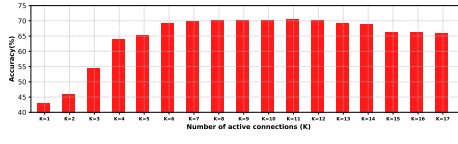

Fig. 2: Varying the fan-in (K), i.e., the number of learned active connections to each residual block. The plot reports accuracy achieved by MaskConnect on CIFAR-100 using a **ResNet**-38 architecture ($L = 18$ blocks). All models have the same number of parameters (0.57M). The best accuracy is achieved at $K = 10$.

We notice that the best accuracy is achieved using $K = 10$. Using a very low or very high fan-in yields lower accuracy. However, the algorithm does not appear to be overly sensitive to the fan-in hyperparameter, as a wide range of values for K (from $K = 7$ to $K = 13$) produce accuracy close to the best.

Varying the model. We trained several ResNet models differing in depth, using both MaskConnect as as well as the traditional predefined connectivity. For these experiments we use a stack of L residual blocks with two 3x3 convolutional layers for each block. We choose $L \in \{18, 36, 54\}$ to build networks with depths $D = 2+2L$ equal to 38, 74, and 110 layers, respectively. We show the classification

accuracy achieved by different models in Table 1. We report the results achieved using MaskConnect with fan-in $K = 10$, $K = 15$, $K = 20$ for models of depth $D = 38$, $D = 74$, $D = 110$, respectively. Fixed-Prev denotes the performance of ResNet, where each block is connected to only the previous block ($K = 1$). We also include the accuracy achieved by choosing a random connectivity (Fixed-Random) using the same fan-in values K as our approach and training the parameters while keeping the random connectivity fixed. This baseline is useful to show that our model achieves higher accuracy over traditional ResNet not because of the higher number of connections (i.e., $K > 1$), but rather because it learns the connectivity. Indeed, the results in Table 1 show that learning the connectivity using MaskConnect yields consistently higher accuracy than using multiple random connections or a single connection to the previous block.

CIFAR-100 results based on multi-branch *ResNeXt*.

Effect of fan-in (K). Even for ResNeXt, we start by studying the effect of the fan-in hyperparameter (K). For this experiment we use a model obtained by stacking $L = 6$ multi-branch residual modules, each having cardinality $C = 8$ (number of branches in each module). We use residual blocks consisting of 3 convolutional layers with a bottleneck implementing dimensionality reduction on the number of feature channels, as shown in Figure 1(b). The bottleneck for this experiment was set to $w = 4$. Since each residual block consists of 3 layers, the total depth of the network in terms of learnable layers is $D = 2 + 3L = 20$.

We trained and tested this architecture using different fan-in values: $K = 1, .., 8$. Again, varying K does not alter the number of parameters. The results are shown in Figure 3. We can see that the best accuracy is achieved by connecting each residual block to $K = 4$ branches out of the total $C = 8$ in each module. Note that when setting $K = C$, there is no need to learn the masks. In this case each mask is simply replaced by an element-wise addition of the outputs from all the branches. This renders the model equivalent to ResNeXt [47], which has fixed connectivity. Based on the results of Figure 3, in all our experiments below we use $K = 4$ (since it gives the best accuracy) but also $K = 1$ since it gives high sparsity which, as we will see shortly, implies savings in number of parameters.

Varying the models. In Table 2 we show the classification accuracy achieved with ResNeXt models of different depth and cardinality (the details of each model are listed in Appendix A). For each architecture we also include the accuracy achieved with full (as opposed to learned) connectivity, which corresponds to ResNeXt. These results show that learning the connectivity produces consistently higher accuracy than using fixed connectivity, with accuracy gains of up to 2.2% compared to the state-of-the-art ResNeXt model. Furthermore, we can notice that the accuracy of models based on random connectivity (Fixed-Random) is considerably lower compared to our approach, despite having the same connectivity density ($K = 4$). This shows that the improvements of our approach over ResNeXt are not due to sparser connectivity but they are rather due to *learned*

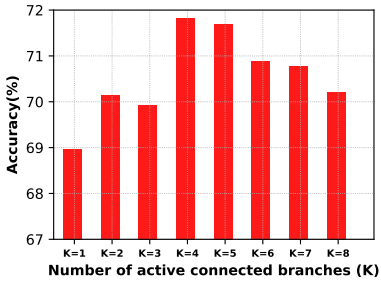

Fig. 3: Varying the fan-in (K) of our model, i.e., the number of active input branches to each residual block. The plot reports accuracy achieved on CIFAR-100 using a network stack of L=6 **ResNeXt** modules having cardinality C=8 and bottleneck width w=4. All models have the same number of parameters (0.28M).

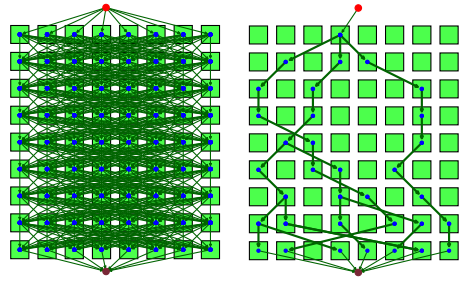

Fig. 4: A visualization of the fixed connectivity of **ResNext** (left) vs the connectivity learned by our method (right) using K=1. Each green square is a residual block, each row of C=8 square is a multi-branch module. Arrows indicate pathways connecting residual blocks of adjacent modules. It can be noticed that MaskConnect learns sparse connections. The squares without in/out edges are those pruned at the end of learning. This gives rise to a branching factor that varies along the depth of the net.

connectivity. We note that these improvements in accuracy come at little computational training cost: the average training time overhead for learning masks and weights is about 39% using our unoptimized implementation compared to learning only the weights given a fixed connectivity.

Parameter savings. Our proposed approach provides the benefit of automatically identifying residual blocks that are unnecessary. At the end of the training, the unused residual blocks can be pruned away. This yields savings in the number of parameters to store and in test-time computation. In Table 2, columns *Train* and *Test* under *Params* show the original number of parameters (used during training) and the number of parameters after pruning (used at test-time). Note that for the biggest architecture, our approach using $K = 1$ yields a parameter saving of 40% compared to ResNeXt with full connectivity (20.5M vs 34.4M), while achieving the same accuracy. Thus, in summary, using fan-in $K = 4$ gives models that have the same number of parameters as ResNeXt but they yield higher accuracy; using fan-in $K = 1$ gives a significant saving in number of parameters and accuracy on par with ResNeXt.

Visualization of the learned connectivity. Figure 4 provides an illustration of the connectivity learned by MaskConnect for $K = 1$ versus the fixed connectivity of ResNeXt for model $\{D = 29, w = 8, C = 8\}$. While ResNeXt feeds the same

Table 2: CIFAR-100 accuracies achieved by two **ResNeXt** architectures trained using predefined full connectivity (Fixed-Full) [47], random connectivity (Fixed-Random, K=4), and the connectivity learned by our algorithm (Learned, K=1, K=4). Each model was trained 4 times, using different random initializations. We report the best test performance as well as the mean test performance computed from the 4 runs. We list the number of parameters used during training (Params-Train) and the number of parameters obtained after pruning the unused blocks (Params-Test). Our learned connectivity using K=4 produces accuracy gains of up to 2.2% compared to the strong ResNeXt model, while using K=1 yields results equivalent to ResNeXt but it induces a significant reduction in number of parameters at test time (e.g., a saving of 40% for model {29,64,8})

Architecture	**Connectivity**	**Params**		**Accuracy (%)**
{Depth (D), Bottleneck width (w), Cardinality (C)}		*Train*	*Test*	*best (mean±std)*
{29,8,8}	Fixed-Full, K=8 [47]	0.86M	0.86M	73.52 (73.37±0.13)
	Learned, K=1	0.86M	0.65M	**73.91** (73.76±0.14)
	Learned, K=4	0.86M	0.81M	**75.89** (75.77±0.12)
	Fixed-Random, K=4	0.86M	0.85M	72.85 (72.66±0.24)
{29,64,8}	Fixed-Full, K=8 [47]	34.4M	34.4M	82.23 (82.12±0.12)
	Learned, K=1	34.4M	20.5M	**82.31** (82.15±0.15)
	Learned, K=4	34.4M	32.1M	**84.05** (83.94±0.11)
	Fixed-Random, K=4	34.4M	34.3M	81.96 (81.73±0.20)

input to all blocks of a module, our algorithm learns different input pathways for each block and yields a branching factor that varies along depth.

4.2 ImageNet

Finally, we evaluate our approach on the large-scale ImageNet 2012 dataset [11], which includes images of 1000 classes. We train our approach on the training set (1.28M images) and evaluate it on the validation set (50K images).

ImageNet results based on the *ResNet* architecture. For this experiment we use a stack of L=16 residual blocks with 3 convolutional layers with a bottleneck architecture. Thus, the total number of layers is D=2 + 3L=50. Compared to the traditional ResNet using fixed connectivity, the same network trained using MaskConnect with fan-in K=10 yields a top-1 accuracy gain of 1.94% (78.09% vs 76.15%).

ImageNet results based on multi-branch *ResNeXt*. In Table 3, we report the top accuracies for three different ResNeXt architectures. For these experiments we set K=C/2. We can observe that for all three architectures, our learned connectivity yields an improvement in accuracy over fixed full connectivity [47].

Table 3: ImageNet accuracies (single crop) achieved by different architectures using the predefined connectivity of **ResNeXt** (Fixed-Full) versus the connectivity learned by our algorithm (Learned)

Architecture	**Connectivity**	**Accuracy**	
{Depth (D), Bottleneck width (w), Cardinality (C)}		*Top-1*	*Top-5*
{50,4,32}	Fixed-Full, K=32 [47]	77.8	93.3
	Learned, K=16	**79.1**	**94.1**
{101,4,32}	Fixed-Full, K=32 [47]	78.8	94.1
	Learned, K=16	**79.5**	**94.5**
{101,4,64}	Fixed-Full, K=64 [47]	79.6	94.7
	Learned, K=32	**79.8**	**94.8**

5 Conclusions

In this paper we introduced an algorithm to learn the connectivity of deep modular networks. The problem is formulated as a single joint optimization over the weights and connections between modules in the model. We tested our approach on challenging image categorization benchmarks where it led to significant accuracy improvements over the state-of-the-art ResNet and ResNeXt models using fixed connectivity. An added benefit of our approach is that it can automatically identify superfluous blocks, which can be pruned after training without impact on accuracy for more efficient testing and for reducing the number of parameters to store.

While our experiments were carried out on two particular architectures (ResNet and ResNeXt) and a specific form of building block (residual block), we expect the benefits of our approach to extend to other modules and network structures. For example, it could be applied to learn the connectivity of skip-connections in DenseNets [25], which are currently based on predefined connectivity rules. In this paper, our masks perform non-parametric additive aggregation of the branch outputs. It would be interesting to experiment with learnable (parametric) aggregations of the outputs from the individual branches. Our approach is limited to learning connectivity within a given, fixed architecture. Future work will explore the use of learnable masks for full architecture discovery.

Acknowledgements. This work was funded in part by NSF award CNS-120552. We gratefully acknowledge NVIDIA and Facebook for the donation of GPUs used for portions of this work.

References

1. Adams, R.P., Wallach, H.M., Ghahramani, Z.: Learning the structure of deep sparse graphical models. In: Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics, AISTATS 2010, Chia Laguna Resort, Sardinia, Italy, May 13-15, 2010. pp. 1–8 (2010)
2. Almahairi, A., Ballas, N., Cooijmans, T., Zheng, Y., Larochelle, H., Courville, A.: Dynamic capacity networks. In: International Conference on Machine Learning. pp. 2549–2558 (2016)
3. Bengio, E., Bacon, P.L., Pineau, J., Precup, D.: Conditional computation in neural networks for faster models. arXiv preprint arXiv:1511.06297 (2015)
4. Bengio, Y.: Deep learning of representations: Looking forward. In: International Conference on Statistical Language and Speech Processing. pp. 1–37. Springer (2013)
5. Bengio, Y., Léonard, N., Courville, A.: Estimating or propagating gradients through stochastic neurons for conditional computation. arXiv preprint arXiv:1308.3432 (2013)
6. Bergstra, J., Bengio, Y.: Random search for hyper-parameter optimization. Journal of Machine Learning Research **13**, 281–305 (2012)
7. Chen, Y., Kalantidis, Y., Li, J., Yan, S., Feng, J.: Multi-fiber networks for video recognition. In: European Conference on Computer Vision (ECCV) (2018)
8. Cho, K., Bengio, Y.: Exponentially increasing the capacity-to-computation ratio for conditional computation in deep learning. arXiv preprint arXiv:1406.7362 (2014)
9. Courbariaux, M., Bengio, Y., David, J.: Binaryconnect: Training deep neural networks with binary weights during propagations. In: Advances in Neural Information Processing Systems 28, Montreal, Quebec, Canada. pp. 3123–3131 (2015)
10. Davis, A., Arel, I.: Low-rank approximations for conditional feedforward computation in deep neural networks. arXiv preprint arXiv:1312.4461 (2013)
11. Deng, J., Dong, W., Socher, R., Li, L., Li, K., Li, F.: Imagenet: A large-scale hierarchical image database. In: 2009 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2009), 20-25 June 2009, Miami, Florida, USA. pp. 248–255 (2009)
12. Denoyer, L., Gallinari, P.: Deep sequential neural network. arXiv preprint arXiv:1410.0510 (2014)
13. Eigen, D., Ranzato, M., Sutskever, I.: Learning factored representations in a deep mixture of experts. arXiv preprint arXiv:1312.4314 (2013)
14. Fernando, C., Banarse, D., Blundell, C., Zwols, Y., Ha, D., Rusu, A.A., Pritzel, A., Wierstra, D.: Pathnet: Evolution channels gradient descent in super neural networks. CoRR **abs/1701.08734** (2017), `http://arxiv.org/abs/1701.08734`
15. Floreano, D., Dürr, P., Mattiussi, C.: Neuroevolution: from architectures to learning. Evolutionary Intelligence **1**(1), 47–62 (2008)
16. Gastaldi, X.: Shake-shake regularization. CoRR **abs/1705.07485** (2017), `http://arxiv.org/abs/1705.07485`
17. Glorot, X., Bordes, A., Bengio, Y.: Deep sparse rectifier neural networks. In: Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics, AISTATS 2011, Fort Lauderdale, USA, April 11-13, 2011. pp. 315–323 (2011)
18. Guo, Y., Yao, A., Chen, Y.: Dynamic network surgery for efficient dnns. In: Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain. pp. 1379–1387 (2016)

19. Han, S., Mao, H., Dally, W.J.: Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. In: International Conference on Learning Representations (ICLR) (2015)
20. Han, S., Pool, J., Narang, S., Mao, H., Tang, S., Elsen, E., Catanzaro, B., Tran, J., Dally, W.J.: DSD: regularizing deep neural networks with dense-sparse-dense training flow. In: International Conference on Learning Representations (ICLR) (2016)
21. Han, S., Pool, J., Tran, J., Dally, W.J.: Learning both weights and connections for efficient neural network. In: Advances in Neural Information Processing Systems 28, Montreal, Quebec, Canada. pp. 1135–1143 (2015)
22. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. CoRR **abs/1512.03385** (2015)
23. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: Computer Vision and Pattern Recognition (CVPR), 2016 IEEE Conference on (2016)
24. He, K., Zhang, X., Ren, S., Sun, J.: Identity mappings in deep residual networks. In: Computer Vision - ECCV 2016 - 14th European Conference, Amsterdam, The Netherlands, October 11-14, 2016, Proceedings, Part IV. pp. 630–645 (2016)
25. Huang, G., Liu, Z., Weinberger, K.Q.: Densely connected convolutional networks. In: IEEE Conference on Computer Vision and Pattern Recognition, CVPR (2017)
26. Krizhesvsky, A.: Learning multiple layers of features from tiny images (2009), technical Report `https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf`
27. Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. In: Advances in Neural Information Processing Systems 25, Lake Tahoe, Nevada, United States. pp. 1106–1114 (2012)
28. LeCun, Y., Denker, J.S., Solla, S.A.: Optimal brain damage. In: Advances in Neural Information Processing Systems 2, [NIPS Conference, Denver, Colorado, USA, November 27-30, 1989]. pp. 598–605 (1989)
29. Liu, H., Simonyan, K., Vinyals, O., Fernando, C., Kavukcuoglu, K.: Hierarchical representations for efficient architecture search. arXiv preprint arXiv:1711.00436 (2017)
30. Maas, A.L., Hannun, A.Y., Ng, A.Y.: Rectifier nonlinearities improve neural network acoustic models. Proc. ICML **30**, 1 (2013)
31. Pham, H., Guan, M.Y., Zoph, B., Le, Q.V., Dean, J.: Efficient neural architecture search via parameter sharing. arXiv preprint arXiv:1802.03268 (2018)
32. Ravi, S., Larochelle, H.: Optimization as a model for few-shot learning. In: International Conference on Learning Representations (ICLR) (2017)
33. Real, E., Moore, S., Selle, A., Saxena, S., Suematsu, Y.L., Le, Q.V., Kurakin, A.: Large-scale evolution of image classifiers. CoRR **abs/1703.01041** (2017)
34. Salimans, T., Ho, J., Chen, X., Sidor, S., Sutskever, I.: Evolution strategies as a scalable alternative to reinforcement learning. arXiv preprint arXiv:1703.03864 (2017)
35. Saxena, S., Verbeek, J.: Convolutional neural fabrics. In: Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain. pp. 4053–4061 (2016)
36. Sermanet, P., Eigen, D., Zhang, X., Mathieu, M., Fergus, R., LeCun, Y.: Overfeat: Integrated recognition, localization and detection using convolutional networks. In: International Conference on Learning Representations (ICLR) (2013)

37. Shazeer, N., Mirhoseini, A., Maziarz, K., Davis, A., Le, Q., Hinton, G., Dean, J.: Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. arXiv preprint arXiv:1701.06538 (2017)

38. Simonyan, K., Zisserman, A.: Very deep convolutional networks for large-scale image recognition. In: International Conference on Learning Representations (ICLR) (2015)

39. Snoek, J., Larochelle, H., Adams, R.P.: Practical bayesian optimization of machine learning algorithms. In: Advances in Neural Information Processing Systems 25, Lake Tahoe, Nevada, United States. pp. 2960–2968 (2012)

40. Snoek, J., Rippel, O., Swersky, K., Kiros, R., Satish, N., Sundaram, N., Patwary, M.M.A., Prabhat, Adams, R.P.: Scalable bayesian optimization using deep neural networks. In: Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015. pp. 2171–2180 (2015)

41. Such, F.P., Madhavan, V., Conti, E., Lehman, J., Stanley, K.O., Clune, J.: Deep neuroevolution: genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning. arXiv preprint arXiv:1712.06567 (2017)

42. Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S.E., Anguelov, D., Erhan, D., Vanhoucke, V., Rabinovich, A.: Going deeper with convolutions. In: IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015, Boston, MA, USA, June 7-12, 2015. pp. 1–9 (2015)

43. Vinyals, O., Blundell, C., Lillicrap, T., Kavukcuoglu, K., Wierstra, D.: Matching networks for one shot learning. In: Advances in Neural Information Processing Systems 29, Barcelona, Spain. pp. 3630–3638 (2016)

44. Wierstra, D., Gomez, F.J., Schmidhuber, J.: Modeling systems with internal state using evolino. In: Genetic and Evolutionary Computation Conference, GECCO 2005, Proceedings, Washington DC, USA, June 25-29, 2005. pp. 1795–1802 (2005)

45. Wu, Z., Nagarajan, T., Kumar, A., Rennie, S., Davis, L.S., Grauman, K., Feris, R.: Blockdrop: Dynamic inference paths in residual networks. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. pp. 8817–8826 (2018)

46. Xie, L., Yuille, A.L.: Genetic cnn. In: ICCV. pp. 1388–1397 (2017)

47. Xie, S., Girshick, R.B., Dollár, P., Tu, Z., He, K.: Aggregated residual transformations for deep neural networks. In: IEEE Conference on Computer Vision and Pattern Recognition, CVPR (2017)

48. Zeiler, M.D., Fergus, R.: Visualizing and understanding convolutional networks. In: Computer Vision - ECCV 2014 - 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part I. pp. 818–833 (2014)

49. Zoph, B., Le, Q.V.: Neural architecture search with reinforcement learning. In: International Conference on Learning Representations (ICLR) (2017)

A Appendix

A.1 Pseudocode of MaskConnect applied to ResNet

Algorithm 1 MaskConnect training algorithm For ResNet.

Input: a labeled example $(\mathbf{x}_1, \mathbf{z}_1)$, K: fan-in (number of active inputs out of the preceding blocks), η: learning rate, ℓ: the loss over the minibatch, $\mathcal{F}(\mathbf{x};\theta)$ is the residual function, $\tilde{\mathbf{m}}_j \in [0,1]^j$: real-valued branch masks for block j.
Output: updated $\tilde{\mathbf{m}}_j$
1. Forward Propagation:
Normalize the real-valued mask to sum up to 1: $\tilde{m}_{j,k} \leftarrow \frac{\tilde{m}_{j,k}}{\sum_{k'=1}^{j} \tilde{m}_{j,k'}}$
Reset binary mask: $\mathbf{m}_j \leftarrow \mathbf{0}$
Draw K *distinct* samples from multinomial mask distribution:
$a_1, a_2, \ldots, a_K \leftarrow \text{Mult}(\tilde{m}_{j,1}, \tilde{m}_{j,2}, \ldots, \tilde{m}_{j,j-1})$
Set active binary mask based on drawn samples:
$m_{j,a_k} \leftarrow 1$ for $k = 1, ..., K$
Compute input $\mathbf{x}_j$ to j-th block ($j \geq 2$): $\mathbf{x}_j \leftarrow \sum_{k=1}^{j-1} m_{j,k} \left[\mathcal{F}(\mathbf{x}_k;\theta_k) + \mathbf{x}_k\right]$
2. Backward Propagation:
Compute $\frac{\partial \ell}{\partial \mathbf{x}_j}$
Compute $\frac{\partial \ell}{\partial \mathbf{x}_k}$ from $\frac{\partial \ell}{\partial \mathbf{x}_j}$ and $m_{j,k}$ for $j > k$
3. Parameter Update:
Compute $\frac{\partial \ell}{\partial m_{j,k}}$ given $\frac{\partial \ell}{\partial \mathbf{x}_j}$ and $[\mathcal{F}(\mathbf{x}_k;\theta_k) + \mathbf{x}_k]$
$\tilde{m}_{j,k} \leftarrow \text{clip}(\tilde{m}_{j,k} - \eta \cdot \frac{\partial \ell}{\partial m_{j,k}})$

A.2 Pseudocode of MaskConnect applied to multi-branch ResNeXt

Algorithm 2 MaskConnect training algorithm For ResNeXt.

Input: a labeled examples $(\mathbf{x}, \mathbf{z})$, C: cardinality (number of branches), K: fan-in (number of active branch connections), η: learning rate, ℓ: the loss over the minibatch, $\tilde{\mathbf{m}}_j^{(i)} \in [0,1]^C$: real-valued branch masks for block j in module i from previous training iteration.
Output: updated $\tilde{\mathbf{m}}_j^{(i)}$
1. Forward Propagation:
Normalize the real-valued mask to sum up to 1: $\tilde{m}_{j,k}^{(i)} \leftarrow \frac{\tilde{m}_{j,k}^{(i)}}{\sum_{k'=1}^{C} \tilde{m}_{j,k'}^{(i)}}$, for $j = 1, \ldots, C$
Reset binary mask: $\mathbf{m}_j^{(i)} \leftarrow \mathbf{0}$
Draw K *distinct* samples from multinomial mask distribution:
$a_1, a_2, \ldots, a_K \leftarrow \text{Mult}(\tilde{m}_{j,1}^{(i)}, \tilde{m}_{j,2}^{(i)}, \ldots, \tilde{m}_{j,C}^{(i)})$
Set active binary mask based on drawn samples:
$m_{j,a_k}^{(i)} \leftarrow 1$ for $k = 1, ..., K$
Compute input $\mathbf{x}_j^{(i)}$ of block j in module i given branch activations $\mathbf{y}_k^{(i-1)}$ and mask $\mathbf{m}_j^{(i)}$: $\mathbf{x}_j^{(i)} \leftarrow \sum_{k=1}^{C} m_{j,k}^{(i)} \cdot \mathbf{y}_k^{(i-1)}$
2. Backward Propagation:
Compute $\frac{\partial \ell}{\partial \mathbf{x}_j^{(i)}}$ from $\frac{\partial \ell}{\partial \mathbf{y}_j^{(i)}}$
Compute $\frac{\partial \ell}{\partial \mathbf{y}_k^{(i-1)}}$ from $\frac{\partial \ell}{\partial \mathbf{x}_j^{(i)}}$ and $m_{j,k}^{(i)}$
3. Parameter Update:
Compute $\frac{\partial \ell}{\partial m_{j,k}^{(i)}}$ given $\frac{\partial \ell}{\partial \mathbf{x}_j^{(i)}}$ and $\mathbf{y}_k^{(i-1)}$
$\tilde{m}_{j,k}^{(i)} \leftarrow \text{clip}(\tilde{m}_{j,k}^{(i)} - \eta \cdot \frac{\partial \ell}{\partial m_{j,k}^{(i)}})$

A.3 Experiments on CIFAR-10 based on ResNeXt Architecture

The CIFAR-10 dataset consists of color images of size 32x32. The training set contains 50,000 images, the testing set 10,000 images. Each image in CIFAR-10 is categorized into one of 10 possible classes. In Table 4, we report the performance of different models trained on CIFAR-10. From these results we can observe that our models using learned connectivity achieve consistently better performance over the equivalent models trained with the fixed connectivity [47].

Table 4: CIFAR-10 accuracies (single crop) achieved by different multi-branch architectures trained using the predefined connectivity of ResNeXt (Fixed-Full) versus the connectivity learned by our algorithm (Learned). Each model was trained 4 times, using different random initializations. For each model we report the best test performance as well as the mean test performance computed from the 4 runs

Architecture	**Connectivity**	**Accuracy (%)**
{Depth (D), Bottleneck width (w), Cardinality (C)}		*Top-1* best (mean±std)
{20,4,8}	Fixed-Full K=8 [47]	91.39 (91.13±0.11)
	Learned K=4	**92.85** (92.76±0.10)
{29,4,8}	Fixed-Full K=8 [47]	92.77 (92.65±0.09)
	Learned K=4	**93.88** (93.76±0.12)
{29,8,8}	Fixed-Full K=8 [47]	93.26 (93.14±0.11)
	Learned K=4	**95.11** (94.96±0.12)
{29,64,8}	Fixed-Full K=8 [47]	96.35 (96.23±0.12)
	Learned K=4	**96.83** (96.73±0.11)

A.4 Experiments on Mini-ImageNet based on ResNeXt Architecture

Mini-ImageNet is a subset of the full ImageNet [11] dataset. It was used in [43,32]. It is created by randomly selecting 100 classes from the full ImageNet [11]. For each class, 600 images are randomly selected. We use 500 examples per class for training, and the other 100 examples per class for testing. The selected images are resized to size 84x84 pixels as in [43,32]. The advantage of this dataset is that it poses the recognition challenges typical of the ImageNet photos but at the same time it does not need require the powerful resources needed to train on the full ImageNet dataset. This allows to include the additional baselines involving random fixed connectivity (Fixed-Random).

We report the performance of different models trained on Mini-ImageNet in Table 5. From these results, we see that our models using learned connectivity with

Table 5: Mini-ImageNet accuracies achieved by different multi-branch networks trained using the predefined full connectivity of ResNeXt (Fixed-Full) versus the connectivity learned by our algorithm (Learned). Additionally, we include models trained using random fixed connectivity (Fixed-Random) for $K = 4$. For each model we report the best and the mean test performance computed from 4 different training runs. Our method for joint learning of weights and connectivity yields a gain of over 3% in Top-1 accuracy over ResNeXt, which uses the same architectures but a fixed branch connectivity

Architecture {Depth (D), Bottleneck width (w), Cardinality (C)}	**Connectivity**	**Accuracy** *Top-1* best (mean±std)
{20,4,8}	Fixed-Full K=8 [47]	62.12 (61.86±0.15)
	Learned K=4	**66.09** (65.94±0.16)
	Fixed-Random K=4	62.42 (61.81±0.32)
{29,8,8}	Fixed-Full K=8 [47]	68.11 (67.89±0.19)
	Learned K=4	**71.36** (71.18±0.19)
	Fixed-Random K=4	67.97 (67.53±0.20)

fan-in K=4 yield a nice accuracy gain over the same models trained with the fixed full connectivity of ResNeXt [47]. The absolute improvement (in Top-1 accuracy) is 3.87% for the 20-layer network and 3.17% for the 29-layer network. We can notice that the accuracy of the models with fixed random connectivity (Fixed-Random) is considerably lower compared to our nets with learned connectivity, despite having the same connectivity density ($K = 4$). This shows that the improvement of our approach over ResNeXt is not due to sparser connectivity but it is rather due to *learned* connectivity.

A.5 Visualizations of learned connectivity based on ResNeXt Architecture

TheSupp plot in Figure 5 shows how the number of active branches varies as a function of the module depth for model $\{D = 29, w = 4, C = 8\}$ trained on CIFAR-100. For $K = 1$, we can observe that the number of active branches tends to be larger for deep modules (closer to the output layer) compared to early modules (closer to the input). We observed this phenomenon consistently for all architectures. This suggests that having many parallel threads of computation is particularly important in deep layers of the network. Conversely, the setting $K = 4$ tends to produce a fairly uniform number of active branches across the modules and the number is quite close to the maximum value C. For this reason, there is little saving in terms of number of parameters when using $K = 4$, as there are rarely unused blocks.

Table 6: Specifications of the architectures used in our experiments on the CIFAR-10 and CIFAR-100 datasets based on ResNeXt architecture. The architectures differ in terms of depth (D), bottleneck width (w), and cardinality (C). Inside the brackets we specify the residual block used in each multi-branch module by listing the number of input channels, the size of the convolutional filters, as well as the number of filters (number of output channels). To the right of each bracket we list the cardinality (i.e., the number of parallel branches in the module). $\times 2$ means that the same multi-branch module is stacked twice. The first layer for all models is a convolutional layer with 16 filters of size 3×3. The last layer performs global average pooling followed by a softmax

{D=20, w=4, C=8}	{D=29, w=4, C=8}	{D=29, w=8, C=8}	{D=29, w=64, C=8}
3, 3×3, 16	3, 3×3, 16	3, 3×3, 16	3, 3×3, 64
[16, 1×1, 4 4, 3×3, 4 4, 1×1, 64] (C=8)	[16, 1×1, 4 4, 3×3, 4 4, 1×1, 64] (C=8)	[16, 1×1, 8 8, 3×3, 8 8, 1×1, 64] (C=8)	[64, 1×1, 64 64, 3×3, 64 64, 1×1, 256] (C=8)
[64, 1×1, 4 4, 3×3, 4 4, 1×1, 64] (C=8)	[64, 1×1, 4 4, 3×3, 4 4, 1×1, 64] (C=8),×2	[64, 1×1, 8 8, 3×3, 8 8, 1×1, 64] (C=8),×2	[256, 1×1, 64 64, 3×3, 64 64, 1×1, 256] (C=8),×2
[64, 1×1, 8 8, 3×3, 8 8, 1×1, 128] (C=8)	[64, 1×1, 8 8, 3×3, 8 8, 1×1, 128] (C=8)	[64, 1×1, 16 16, 3×3, 16 16, 1×1, 128] (C=8)	[256, 1×1, 128 128, 3×3, 128 128, 1×1, 512] (C=8)
[128, 1×1, 8 8, 3×3, 8 8, 1×1, 128] (C=8)	[128, 1×1, 8 8, 3×3, 8 8, 1×1, 128] (C=8),×2	[128, 1×1, 16 16, 3×3, 16 16, 1×1, 128] (C=8),×2	[512, 1×1, 128 128, 3×3, 128 128, 1×1, 512] (C=8),×2
[128, 1×1, 16 16, 3×3, 16 16, 1×1, 256] (C=8)	[128, 1×1, 16 16, 3×3, 16 16, 1×1, 256] (C=8)	[128, 1×1, 32 32, 3×3, 32 32, 1×1, 256] (C=8)	[512, 1×1, 256 256, 3×3, 256 256, 1×1, 1024] (C=8)
[256, 1×1, 16 16, 3×3, 16 16, 1×1, 256] (C=8)	[256, 1×1, 16 16, 3×3, 16 16, 1×1, 256] (C=8),×2	[256, 1×1, 32 32, 3×3, 32 32, 1×1, 256] (C=8),×2	[1024, 1×1, 256 256, 3×3, 256 256, 1×1, 1024] (C=8),×2
Average Pool 100 fc, softmax	Average Pool 100 fc, softmax	Average Pool 100 fc, softmax	Average Pool 100 fc, softmax

The plot in Figure 6 shows the number of active branches as a function of module depth for model $\{D = 50, w = 4, C = 32\}$ trained on ImageNet, using $K = 16$.

A.6 Implementation details

Architectures and settings for experiments on CIFAR-100 based on ResNet

The architectures used in theses experiments are the same used in the original ResNet paper [23] for CIFAR-10 experiments except for the last fully-connected layer and softmax which have output size of 100 instead of 10. We used the data augmentation strategy where four pixels are padded on each side of the input image, and a 32x32 crop is randomly sampled from the padded image or its horizontal flip, with per-pixel mean subtracted [27]. For testing, we use the original 32x32 image. The stacks have output feature map of size 32, 16, and 8 respectively. The models are trained on 2 GPUs with a mini-batch size of 128, with a weight decay of 0.0001 and momentum of 0.9. We adopt *four* incremental training phases with a total of 80 epochs. In *phase 1* we train the model for 30

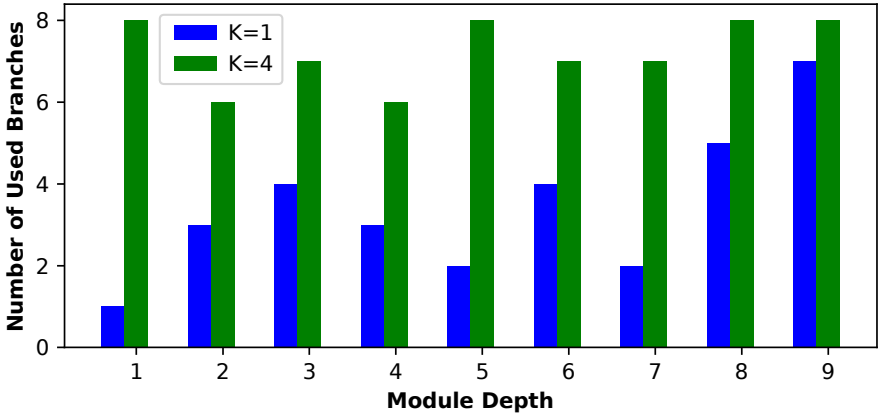

Fig. 5: Number of active branches as a function of module depth for model $\{D = 29, w = 4, C = 8\}$ trained on CIFAR-100. We report how the number of active branches varies for model trained with fan-in $K = 1$ as well as for the net trained with $K = 4$. The setting $K = 1$ tends to leave many blocks unused, especially in the early modules of the network.

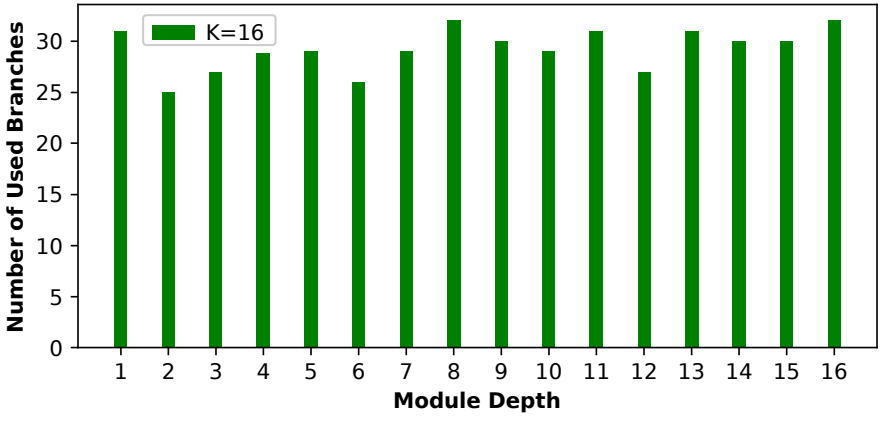

Fig. 6: Number of active branches as a function of module depth for model $\{D = 50, w = 4, C = 32\}$ trained on ImageNet, using fan-in $K = 16$.

epochs with a learning rate of 0.1 for the convolutional and fully-connected layers, and a learning rate of 0.3 for the masks. In *phase 2* we freeze the connectivity by setting as active connections for each block those corresponding to its top-K values in the masks. With these fixed learned connectivity, we finetune the model from *phase 1* for 30 epochs with a learning rate of 0.1 for the weights. Then, in *phase 3* we finetune the weights of the model from *phase 2* for 10 epochs with a learning rate of 0.01 using again the fixed learned connectivity from phase 1. Finally, in *phase 4* we finetune the weights of the model from *phase 3* for 10 epochs with a learning rate of 0.001.

Architectures and settings for experiments on ImageNet based on ResNet

The architectures for our ImageNet experiments are those specified in the original ResNet paper [23]. For these experiments, we follow the data augmentation strategy described in [23]. The size of the input image is 224x224 and it is randomly cropped from the resized original image. We use a mini-batch size of 256, with a weight decay of 0.0001 and a momentum of 0.9. We use *four* incremental training phases with a total of 120 epochs. In *phase 1* we train the model for 30 epochs with a learning rate of 0.1 for the convolutional and fully-connected layers, and a learning rate of 0.3 for the masks. In *phase 2* we finetune the model from *phase 1* for another 30 epochs with a learning rate of 0.1 and a learning rate of 0.0 for the masks (i.e., we use the fixed connectivity learned in phase 1). In *phase 3* we finetune the weights from *phase 2* for 30 epochs with a learning rate of 0.01 and the learning rate of the masks is 0.0. Finally, in *phase 4* we finetune the weights from *phase 3* for 30 epochs with a learning rate of 0.001 while the learning rate of the masks is still set to 0.0.

Architectures and settings for experiments on CIFAR-100 and CIFAR-10 based on ResNeXt

The specifications of the architectures used in all our experiments on CIFAR-10 and CIFAR-100 are given in Table 6.

Several of these architectures are those presented in the original ResNeXt paper [47] and are trained using the same setup, including the data augmentation strategy. Four pixels are padded on each side of the input image, and a 32x32 crop is randomly sampled from the padded image or its horizontal flip, with per-pixel mean subtracted [27]. For testing, we use the original 32x32 image. The stacks have output feature map of size 32, 16, and 8 respectively. The models are trained on 8 GPUs with a mini-batch size of 128 (16 per GPU), with a weight decay of 0.0005 and momentum of 0.9. We adopt *four* incremental training phases with a total of 320 epochs. In *phase 1* we train the model for 120 epochs with a learning rate of 0.1 for the convolutional and fully-connected layers, and a learning rate of 0.2 for the masks. In *phase 2* we freeze the connectivity by setting as active connections for each block those corresponding to its top-K values in the masks. With these fixed learned connectivity, we finetune the model from *phase 1* for 100 epochs with a learning rate of 0.1 for the weights. Then, in *phase*

3 we finetune the weights of the model from *phase 2* for 50 epochs with a learning rate of 0.01 using again the fixed learned connectivity from phase 1. Finally, in *phase 4* we finetune the weights of the model from *phase 3* for 50 epochs with a learning rate of 0.001.

Architectures and settings for experiments on ImageNet based on ResNeXt

The architectures for our ImageNet experiments are those specified in the original ResNeXt paper [47].

Also for these experiments, we follow the data augmentation strategy described in [47]. The input image has size 224x224 and it is randomly cropped from the resized original image. We use a mini-batch size of 256 on 8 GPUs (32 per GPU), with a weight decay of 0.0001 and a momentum of 0.9. We use *four* incremental training phases with a total of 120 epochs. In *phase 1* we train the model for 30 epochs with a learning rate of 0.1 for the convolutional and fully-connected layers, and a learning rate of 0.2 for the masks. In *phase 2* we finetune the model from *phase 1* for another 30 epochs with a learning rate of 0.1 and a learning rate of 0.0 for the masks (i.e., we use the fixed connectivity learned in phase 1). In *phase 3* we finetune the weights from *phase 2* for 30 epochs with a learning rate of 0.01 and the learning rate of the masks is 0.0. Finally, in *phase 4* we finetune the weights from *phase 3* for 30 epochs with a learning rate of 0.001 while the learning rate of the masks is still set to 0.0.

Architectures and settings for experiments on Mini-ImageNet based on ResNeXt

For the experiments on the Mini-ImageNet dataset, a 64x64 crop is randomly sampled from the scaled 84x84 image or its horizontal flip, with per-pixel mean subtracted AlexNet. For testing, we use the center 64x64 crop. The specifications of the models are identical to the CIFAR-100 models used in the previous subsection, except that the first input convolutional layer in the network is followed by a max pooling layer. The models are trained on 8 GPUs with a mini-batch size of 256 (32 per GPU), with a weight decay of 0.0005 and momentum of 0.9. Similar to training CIFAR-100 dataset, we also adopt *four* incremental training phases with a total of 320 epochs.

Table 7: Mini-ImageNet architectures with varying depth (D), and bottleneck width (w). Inside the brackets we specify the residual block used in each multi-branch module by listing the number of input channels, the size of the convolutional filters, as well as the number of filters (number of output channels). To the right of each bracket we list the cardinality (C) (i.e., the number of parallel branches in the module). $\times 2$ means that the same multi-branch module is stacked twice

{D=20, w=4, C=8}	**{D=29, w=8, C=8}**
3, 3×3, 16	3, 3×3, 16
(Max Pool,3×3, stride=2)	(Max Pool, 3×3, stride=2)
[16, 1×1, 4 4, 3×3, 4 4, 1×1, 64] (C=8)	[16, 1×1, 8 8, 3×3, 8 8, 1×1, 64] (C=8)
[64, 1×1, 4 4, 3×3, 4 4, 1×1, 64] (C=8)	[64, 1×1, 8 8, 3×3, 8 8, 1×1, 64] (C=8), ×2
[64, 1×1, 8 8, 3×3, 8 8, 1×1, 128] (C=8)	[64, 1×1, 16 16, 3×3, 16 16, 1×1, 128] (C=8)
[128, 1×1, 8 8, 3×3, 8 8, 1×1, 128] (C=8)	[128, 1×1, 16 16, 3×3, 16 16, 1×1, 128] (C=8), ×2
[128, 1×1, 16 16, 3×3, 16 16, 1×1, 256] (C=8)	[128, 1×1, 32 32, 3×3, 32 32, 1×1, 256] (C=8)
[256, 1×1, 16 16, 3×3, 16 16, 1×1, 256] (C=8)	[256, 1×1, 32 32, 3×3, 32 32, 1×1, 256] (C=8), ×2
Average Pool 100 fc, softmax	Average Pool 100 fc, softmax