Connectivity Learning in Multi-Branch Networks

Karim Ahmed
Department of Computer Science
Dartmouth College
karim@cs.dartmouth.edu

Lorenzo Torresani
Department of Computer Science
Dartmouth College
LT@dartmouth.edu

Abstract

Recent studies in the design of convolutional networks have shown that *branching*, i.e., splitting the computation along parallel but distinct threads and then aggregating their outputs, represents a new promising dimension for significant improvements in performance. To combat the complexity of design choices in multi-branch architectures, prior work has adopted simple strategies, such as a fixed branching factor, the same input being fed to all parallel branches, and an additive combination of the outputs produced by all branches at aggregation points. In this work we remove these predefined choices and propose an algorithm to learn the connections between branches in the network. Instead of being chosen a priori by the human designer, the multi-branch connectivity is learned simultaneously with the weights of the network by optimizing a single loss function defined with respect to the end task. We demonstrate our approach on the problem of multi-class image classification where it yields consistently higher accuracy compared to the state-of-the-art "ResNeXt" multi-branch network given the same learning capacity.

1 Introduction

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, several authors have proposed to simplify the architecture design by defining convolutional neural networks (CNNs) in terms of combinations of basic building blocks. This idea of modularized design was adopted in residual networks (ResNets) [3]. While in ResNets residual blocks are stacked one on top of each other to form very deep networks, the recently introduced ResNeXt models [5] have shown that it is also beneficial to arrange these building blocks in parallel to build multi-branch convolutional networks. The modular component of ResNeXt then consists of C parallel branches, corresponding to residual blocks with identical topology but distinct parameters. Network built by stacking these multi-branch components have been shown to lead to better results than single-thread ResNets of the same capacity.

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. In order to avoid a combinatorial explosion of options, prior work has relied on simple, uniform rules of aggregation and composition. For example, ResNeXt models [5] are based on the following 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.

In this paper we remove these predefined choices and propose an algorithm that learns to combine and aggregate building blocks of a neural network. In this new regime, the network connectivity naturally arises as a result of the training optimization rather than being hand-defined by the human designer.

31st Conference on Neural Information Processing Systems (NIPS 2017), Long Beach, CA, USA.

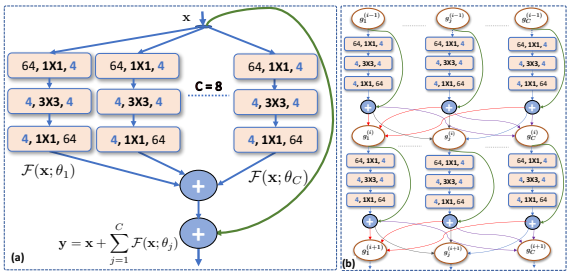

Figure 1: (a) The multi-branch RexNeXt module consisting of C parallel residual blocks [5]. (b) Our approach replaces the fixed aggregation points of RexNeXt with learnable gates $\mathbf{g}_j^{(i)}$ defining the input connections for each individual residual block j in each module i.

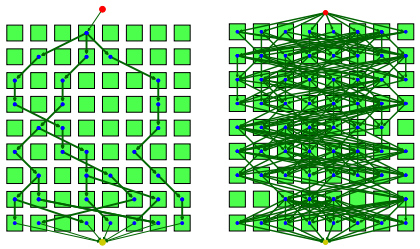

Figure 2: Connectivity learned by our method on CIFAR-100 for fan-in $K = 1$ (left) and $K = 4$ (right). Each green square is a residual block, each row of $C = 8$ square is a multi-branch module. The net consists of a stack of $M = 9$ modules. Arrows indicate pathways connecting residual blocks of adjacent modules. The squares without in/out edges are deemed superfluous and can be pruned at the end of learning.

This is achieved by means of *gates*, i.e., learned binary parameters that act as "switches" determining the final connectivity in our network. The gates are learned together with the convolutional weights of the network, as part of a joint optimization via backpropagation with respect to a traditional multi-class classification objective. We demonstrate that, given the same budget of residual blocks (and parameters), our learned architecture consistently outperforms the predefined ResNeXt network in all our experiments. An interesting byproduct of our approach is that it can automatically identify residual blocks that are superfluous, i.e., unnecessary or detrimental for the end objective. At the end of the optimization, these unused residual blocks can be pruned away without any impact on the learned hypothesis while yielding substantial savings in number of parameters to store and in test-time computation.

2 Technical Approach

2.1 Modular multi-branch architecture

The multi-branch architecture of ResNeXt. We begin by providing a brief review of the ResNeXt [5] architecture, which consists of a stack of multi-branch modules. Each module contains C residual blocks [3] that implement parallel multiple threads of computation feeding from the same input. 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(a). More formally, let $\mathcal{F}(\mathbf{x};\theta_j^{(i)})$ be the transformation implemented by the j-th residual block in module i-th of the network, where $j = 1,\ldots,C$ and $i = 1,\ldots,M$, with M 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 M controls the total depth of the network: under the assumption of 3 layers per residual block (as shown in the figure), the total depth of the network is given by $D = 2 + 3M$ (an initial convolutional layer and an output fully-connected layers add 2 layers). Note that in ResNeXt all residual blocks in a module have the same topology ($\mathcal{F}$) but each block has its own parameters ($\theta_j^{(i)}$ denotes the parameters of residual block j in module i). Then, the output of the i-th module is computed as $\mathbf{y} = \mathbf{x} + \sum_{j=1}^{C} \mathcal{F}(\mathbf{x};\theta_j^{(i)})$. Tensor $\mathbf{y}$ represents the input to the $(i+1)$-th module. In [5] it was experimentally shown that given a fixed budget of parameters, ResNeXt multi-branch networks consistently outperform single-branch ResNets of the same learning capacity. We note, however, that in an attempt to ease network design, several restrictive limitations were embedded in the architecture of ResNeXt modules: each ResNeXt module implements C parallel feature extractors that operate on the same input; furthermore, the number of active branches is constant at all depth levels of the network. Our approach removes these restrictions without adding any significant burden on the process of manual network design.

Our gated multi-branch architecture. As in ResNeXt, our proposed architecture consists of a stack of M 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 gate vector that is learned jointly with the weights of the network. Let $\mathbf{g}_j^{(i)} = [g_{j,1}^{(i)}, g_{j,2}^{(i)}, \ldots, g_{j,C}^{(i)}]^\top \in \{0,1\}^C$ be the binary gate vector defining the *active* input connections feeding the j-th residual block in module i. If $g_{j,k}^{(i)} = 1$, then the activation volume produced by the k-th branch in module $(i-1)$ is fed as input to the j-th residual block of module i. If $g_{j,k}^{(i)} = 0$, then the output from the k-th branch in the previous module is ignored by the j-th residual block of the current module. Thus, if we denote with $\mathbf{y}_k^{(i-1)}$ the output tensor computed by the k-th branch in module $(i-1)$, the input $\mathbf{x}_j^{(i)}$ to the j-th residual block in module i will be given by:

$$\mathbf{x}_j^{(i)} = \sum_{k=1}^{C} g_{j,k}^{(i)} \cdot \mathbf{y}_k^{(i-1)} \tag{1}$$

Then, the output of this block will be obtained through the usual residual computation, i.e., $\mathbf{y}_j^{(i)} = \mathbf{x}_j^{(i)} + \mathcal{F}(\mathbf{x}_j^{(i)}; \theta_j^{(i)})$. We note that under this model we no longer have fixed aggregation nodes summing up *all* outputs from a module. Instead, the gate $\mathbf{g}_j^{(i)}$ now determines *selectively* for each block which branches from the previous module will be aggregated and provided as input to the block. Under this scheme, the parallel branches in a module receive different inputs.

Depending on the constraints posed over $\mathbf{g}_j^{(i)}$, different interesting models can be realized. By imposing that $\sum_k g_{j,k}^{(i)} = 1$ for all blocks j, then each residual block will receive input from only one branch (since each $g_{j,k}^{(i)}$ must be either 0 or 1). At the other end of the spectrum, if we set $g_{j,k}^{(i)} = 1$ 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 found that the best results are achieved for a middle ground between these two extremes, i.e., by connecting each block to K branches where K is an integer-valued hyperparameter such that $1 < K < C$. We refer to this hyperparameter as the *fan-in* of a block. Finally, we note that it may be possible for a residual block in the network to become unused. This happens when block k in module $(i-1)$ is such that $g_{jk}^{(i)} = 0$ for all $j = 1, \ldots, C$. In this case, at the end of the optimization we prune the block without affecting the function computed by the network, so as to reduce the number of parameters to store and to speed up inference. This implies that a variable branching factor is learned adaptively for the different depths in the network.

2.2 GATECONNECT: learning to connect branches

We refer to our learning algorithm as GATECONNECT. It jointly optimizes a given loss ℓ with respect to both the weights of the network (θ) and the gates ($\mathbf{g}$).

To learn the binary parameters $\mathbf{g}$, we adopt a modified version of backpropagation, inspired by the algorithm proposed by Courbariaux et al. [1] to train neural networks with binary weights. During training we store and update a real-valued version $\tilde{\mathbf{g}}_j^{(i)} \in [0,1]^C$ of the branch gates, with entries clipped to lie in the continuous interval from 0 to 1. At each iteration, we stochastically binarize the real-valued branch gates into binary-valued vectors $\mathbf{g}_j^{(i)} \in \{0,1\}^C$ subject to the constraint that it contains only K active entries, where K is a predefined integer hyperparameter with $1 \leq K \leq C$. In other words: $g_{j,k}^{(i)} \in \{0,1\}, \quad \sum_{k=1}^{C} g_{j,k}^{(i)} = K \quad \forall j \in \{1, \ldots, C\}$ and $\forall i \in \{1, \ldots, M\}$.

Forward Propagation. In the forward propagation, our algorithm first normalizes the C real-valued gates for each block j to sum up to 1 (i.e., $\sum_{k=1}^{C} \tilde{g}_{j,k}^{(i)} = 1$) so that $\text{Mult}(\tilde{g}_{j,1}^{(i)}, \tilde{g}_{j,2}^{(i)}, \ldots, \tilde{g}_{j,C}^{(i)})$ defines a proper multinomial distribution. Then, the binary gate $\mathbf{g}_j^{(i)}$ is stochastically generated by drawing K *distinct* samples $a_1, a_2, \ldots, a_K \in \{1, \ldots, C\}$ from the multinomial distribution over the branch connections. Finally, the entries corresponding to the K samples are activated in the binary gate vector. The input activation volume to the residual block j is then computed according to Eq. 1.

Backward Propagation. In the backward propagation step, the gradient $\partial \ell / \partial y_k^{(i-1)}$ with respect to each branch output is obtained via back-propagation from $\partial \ell / \partial x_j^{(i)}$ and the binary gates $g_{j,k}^{(i)}$.

Gate Update. In the update step, our algorithm computes the gradient with respect to the binary branch gates. Then, it updates the real-valued branch gates via gradient descent. At this time we clip the updated real-valued branch gates to constrain them to remain within the valid interval $[0, 1]$.

Algorithm 1 GATECONNECT training algorithm.

Input: a minibatch of labeled examples (x^i, y^i), C: cardinality (number of branches), K: fan-in (number of active branch connections), η: learning rate, ℓ: the loss over the minibatch, $\tilde{\mathbf{g}}_j^{(i)} \in [0,1]^C$: real-valued branch gates for block j in module i from previous training iteration.

Output: updated $\tilde{\mathbf{g}}_j^{(i)}$

1. Forward Propagation:

Normalize the real-valued gate to sum up to 1:

$\tilde{g}_{j,k}^{(i)} \leftarrow \frac{\tilde{g}_{j,k}^{(i)}}{\sum_{k'=1}^{C} \tilde{g}_{j,k'}^{(i)}}$, for $j = 1, \ldots, C$

Reset binary gate: $\mathbf{g}_j^{(i)} \leftarrow \mathbf{0}$

Draw K *distinct* samples from multinomial gate distribution: $a_1, a_2, \ldots, a_K \leftarrow \text{Mult}(\tilde{g}_{j,1}^{(i)}, \tilde{g}_{j,2}^{(i)}, \ldots, \tilde{g}_{j,C}^{(i)})$

Set active binary gate based on drawn samples:

$g_{j,a_k}^{(i)} \leftarrow 1$ for $k = 1, ..., K$

Compute output $\mathbf{x}_j^{(i)}$ of the gate, given branch activations $\mathbf{y}_k^{(i-1)}$: $\mathbf{x}_j^{(i)} \leftarrow \sum_{k=1}^{C} g_{j,k}^{(i)} \cdot \mathbf{y}_k^{(i-1)}$

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 $g_{j,k}^{(i)}$

3. Parameter Update:

Compute $\frac{\partial \ell}{\partial g_{j,k}^{(i)}}$ given $\frac{\partial \ell}{\partial \mathbf{x}_j^{(i)}}$ and $\mathbf{y}_k^{(i-1)}$

$\tilde{g}_{j,k}^{(i)} \leftarrow \text{clip}(\tilde{g}_{j,k}^{(i)} - \eta \cdot \frac{\partial \ell}{\partial g_{j,k}^{(i)}})$

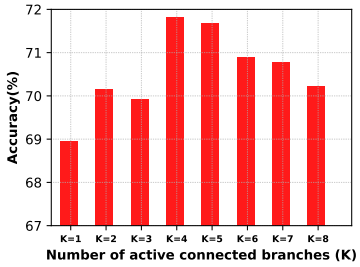

Figure 3: Varying the fan-in (K) of our model, i.e., the number of active branches provided as input to each residual block. The plot reports accuracy achieved on CIFAR-100 using a network stack of $M = 6$ ResNeXt modules having cardinality $C = 8$ and bottleneck width $w = 4$. All models have the same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

Architecture	**Connectivity**	**Params**		**Accuracy (%)**
D,w,C		*Train*	*Test*	*Top-1*
{29,8,8}	Fixed-Full, K=8 [5]	0.86M	0.86M	73.52
	Learned, K=1	0.86M	0.65M	**73.94**
	Learned, K=4	0.86M	0.81M	**75.72**
	Fixed-Random, K=4	0.86M	0.85M	72.85
{29,64,8}	Fixed-Full, K=8 [5]	34.4M	34.4M	82.23
	Learned, K=1	34.4M	20.5M	**82.31**
	Learned, K=4	34.4M	32.1M	**84.05**
	Fixed-Random, K=4	34.4M	34.3M	81.96

Table 1: CIFAR-100 accuracies achieved by different architectures using: (1) the predefined full connectivity of ResNeXt (Fixed-Full), (2) the connectivity learned by our algorithm (Learned) and (3) fixed connectivity (Fixed-Random) defined by setting $K = 4$ random active connections per branch.

After joint training over θ and $\mathbf{g}$, we fine-tune the weights θ with fixed binary gates, by choosing as active connections for each block j in module i those corresponding to the top K values in $\tilde{\mathbf{g}}_j^{(i)}$. Pseudocode for our training procedure is given in Algorithm 1.

3 Experiments

We tested our approach on two image categorization datasets: CIFAR-100 [4] and ImageNet [2].

Effect of fan-in (K). Figure 3 shows the effect of fan-in (K) on performance. The network in this study has $M = 6$ multi-branch residual modules, each having cardinality $C = 8$ (number of branches in each module). We trained and tested this architecture on CIFAR-100 using different fan-in values: $K = 1, .., 8$. Note that varying K does not affect the number of parameters. 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. Using a very low or very high fan-in yields lower accuracy. Note that when setting $K = C$, each gate is simply replaced by an element-wise addition of the outputs from all the branches. This renders the model equivalent to ResNeXt [5], which has fixed connectivity.

Varying the architectures. In Table 1 we show the classification accuracy achieved on CIFAR-100 with different architectures. For each architecture we report results obtained using GATECONNECT with fan-in $K = 1$ and $K = 4$. 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 2.2% compared to the state-of-the-art ResNeXt model. For each architecture we also report results using sparse random connectivity (Fixed-Random). For these models, each gate is set to have $K = 4$ randomly-chosen active connections, and the connectivity is kept fixed during learning of the parameters. We can see that the accuracy of these nets is a lot lower compared to our models, 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* connectivity.

Parameter savings. Our proposed approach provides the benefit of automatically identifying during training 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 1, 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. Figure 2 provides an illustration of the connectivity learned by GATECONNECT for model $\{D = 29, w = 8, C = 8\}$ using $K = 1$ (left) and $K = 4$ (right). While ResNeXt feeds the same 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.

Large-scale evaluation on ImageNet. Finally, we evaluate our approach on the large-scale ImageNet 2012 dataset [2]. We train our models on the training set (1.28M images) and evaluate them on the validation set (50K images). For these experiments we set $K = C/2$. We tried two architectures on this dataset. The first architecture ($\{D = 50, w = 4, C = 32\}$) achieves a top-1 accuracy of 77.8% when using fixed connectivity (ResNeXt), and 79.1% when using our learned connectivity. Similarly, for the second architecture ($\{D = 101, w = 4, C = 32\}$) the accuracy is 78.8% with fixed connectivity, while learning the connectivity with our algorithm yields an accuracy of 79.3%.

References

[1] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Advances in Neural Information Processing Systems 28, Montreal, Quebec, Canada*, pages 3123–3131, 2015.

[2] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Fei-Fei Li. 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*, pages 248–255, 2009.

[3] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Computer Vision and Pattern Recognition (CVPR), 2016 IEEE Conference on*, 2016.

[4] Alex Krizhesvsky. Learning multiple layers of features from tiny images, 2009. Technical Report `https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf`.

[5] Saining Xie, Ross B. Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. Aggregated residual transformations for deep neural networks. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR*, 2017.