

# DeepRebirth: Accelerating Deep Neural Network Execution on Mobile Devices

Dawei Li  
Samsung Research America  
dawei.l@samsung.com

Xiaolong Wang  
Samsung Research America  
xiaolong.w@samsung.com

Deguang Kong  
doogkong@gmail.com

## Abstract

Deploying deep neural networks on mobile devices is a challenging task due to computation complexity and memory intensity. Current model reduction methods (e.g., matrix approximation using SVD) cannot satisfy real-time processing requirement. This paper first discovers that the major obstacle is the excessive execution time of non-tensor layers (with tensor-like parameters) such as pooling and normalization. This motivates us to design a novel acceleration framework: DeepRebirth through “slimming” existing consecutive and parallel non-tensor and tensor layers. The layer slimming is executed at different substructures: (a) streamline slimming by merging the consecutive non-tensor and tensor layer vertically; (b) branch slimming by merging non-tensor and tensor branches horizontally. These different optimization operations accelerate the model execution and reduce the run-time memory cost significantly. To maximally avoid accuracy loss, the parameters in new generated layers are learned with layer-wise fine-tuning based on both theoretical analysis and empirical verification. As observed in the experiment, DeepRebirth achieves 3x-5x speed-up and energy saving on GoogLeNet with only 0.4% accuracy drop on top-5 categorization in ImageNet. Further, by combining with other model compression techniques, DeepRebirth offers an average of 65ms model forwarding time on a single image using Samsung Galaxy S6 with 86.54% top-5 accuracy with 2.5x run-time memory saving.

## 1. Introduction

Recent years have witnessed the breakthrough of deep learning techniques for many computer vision tasks such as image classification and object detection. More and more mobile applications adopt deep learning techniques to provide accurate, intelligent and effective services. However, the execution speed of deep learning models on mobile devices becomes a bottleneck for deployment of many applications due to limited computing resources.

In this paper, we focus on improving the execution efficiency of deep learning models on mobile CPUs, which

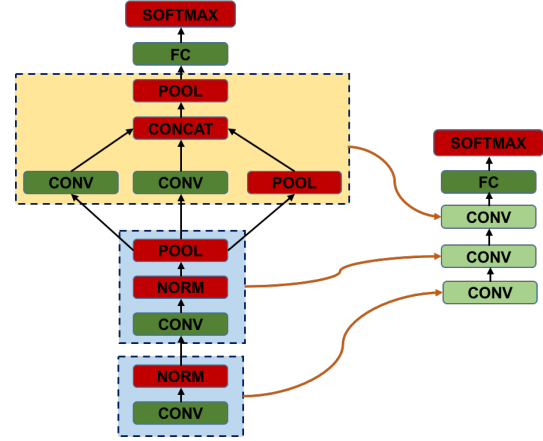


Figure 1: An illustration of proposed DeepRebirth model acceleration pipeline. DeepRebirth optimizes a trained deep learning model (left) to an accelerated “slim” model (right). Such optimization is achieved with two operations: Streamline Slimming which absorbs non-tensor layers (i.e., pooling and normalization) to their bottom convolutional layer (in light blue background) and Branch Slimming which absorbs non-tensor branches and convolutional branches with small convolution filters (e.g., 1x1) to a convolutional branch with large convolution filter (e.g., 5x5) (in light yellow background). We name new generated layers as slim layers.

is a highly intriguing feature. On one hand, a large majority of mobile devices are equipped with mobile GPUs, however, the speed-up achieved is quite limited when compared to CPU [18], not to mention the complexity caused by different mobile GPU architectures. On the other hand, major deep learning frameworks such as Caffe [14] and Tensorflow [1] only support CPU implementation on mobile devices currently, and therefore an efficient CPU-friendly model is highly desirable. In reality, it takes more than 651ms to recognize an image using GoogleNet on Samsung S5 (Table 4) with 984mJ energy costs (Table 5). The effective solution is expected to provide minimum accuracy loss by leveraging widely used deep learning framework (such as GoogLeNet and ResNet) with support of deep model acceleration on different types of layers.

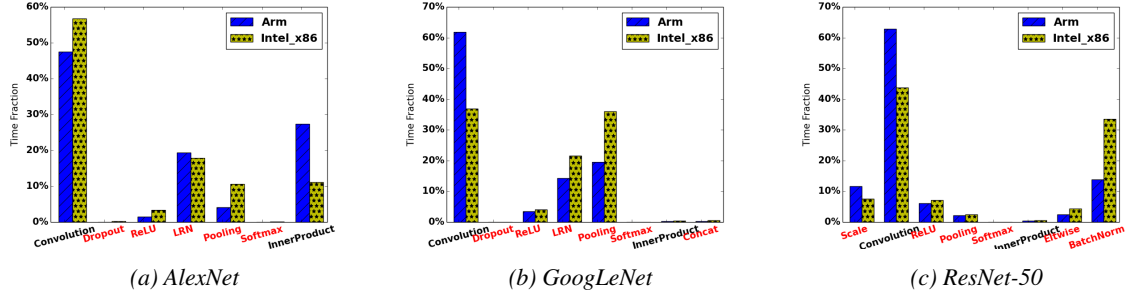


Figure 2: Time Decomposition for each layer. Non-tensor layers (e.g., dropout, ReLU, LRN, softmax, pooling, etc) shown in red color while non-tensor layers (e.g., convolution, inner-product) shown in black color.

Table 1: Compare DeepRebirth with Existing Acceleration Methods on CPU of Samsung Galaxy s6 Mobile Device.

	Parameter Compression[15]	SqueezeNet [12]	DeepRebirth (ours)
Accuracy	85.7%	80.3%	<b>86.5%</b>
Execution Time	342.5 ms	75.3 ms	<b>65.3 ms</b>
Energy Cost	902 mJ	288 mJ	<b>226 mJ</b>
Memory Cost	35.8 MB	365. MB	<b>14.8 MB</b>

### Excessive execution time in Non-tensor layers

In this paper, we find that non-tensor layers consume too much time in model execution (shown in Fig. 2) where *tensor layer* and *non-tensor layer* are defined based on whether the layer contains tensor-type parameters. For example<sup>1</sup>, fully connected layer and the convolution layer are tensor-layers since they contain 2-d and 4-d tensor-type weight parameters, respectively. Motivated by this, this paper proposes DeepRebirth, a new deep learning model acceleration framework that significantly reduces the execution time on non-tensor layers. In particular, we paid our efforts in two directions: (a) *streaming slimming*; (b) *branch slimming*. In streamline slimming, the new tensor layers are re-generated by substituting the original non-tensor layers and their neighborhood sparse tensor layers in the feed-forward model (shown in Figure 3), while in branch slimming, the newly generated tensor layers are created by fusing non-tensor branches with sparse tensor branches at different branches horizontally (shown in Figure 4, such as the inception module in GoogLeNet ([20])). Overall, reducing the execution time on non-tensor layers can greatly reduce the model forwarding time given the fact that tensor-layer has been optimized to the minimum as suggested by ([9], [15]). Therefore, we can combine both non-tensor and tensor layer optimization together and further reduce the latency as well as the model size.

<sup>1</sup>Pooling layer and LRN layer are both non-tensor layers because they do not contain any high-order tensor-type weight parameters. Other examples of non-tensor layers include dropout layer, normalization layer, softmax layer, etc.

**Difference with existing works** The central idea of DeepRebirth is based on the acceleration of non-tensor layers because *non-tensor layers are major obstacles for real-time mobile CPU execution (§2)*. Compared to existing works, ([9], [15]) are designed to reduce the model size by approximating the tensor-type layers using low rank approximation and vector quantization techniques. While they can provide obvious acceleration for large fully-connected layers (used in AlexNet, VGGNet), the application scenarios of these methods are quite limited and ineffective since recent advanced deep learning architectures (e.g., Inception and ResNet) have removed large fully-connected layers. Moreover, for non-tensor layers (e.g., normalization and pooling layers) that are generally used for speeding up the network training and obtaining better generalization performance, optimization for faster execution has *not* been discussed so far.

To summarize, this paper makes the following contributions.

- DeepRebirth is the first work that identifies the excessive execution time of non-tensor layers is the major obstacle for real-time deep model processing on mobile devices.
- DeepRebirth is also the first work that focuses on optimizing non-tensor layers and significantly accelerates a deep learning model on mobile devices while reducing the required runtime-memory with less layers<sup>2</sup>.
- DeepRebirth performs both streaming rebirth and branch rebirth by merging non-tensor layers with its neighboring sparse tensor layers vertically and horizontally, where the new generated tensor layer parameters are re-trained in a principled way that achieves the same functionality as the original layers.
- DeepRebirth obtained the state-of-the-art speeding up on popular deep learning models with negligible accuracy loss, which enables GoogLeNet to achieve 3x-5x speed-up

<sup>2</sup>Weights approximation method such as Tucker Decomposition effectively reduces the model size (i.e., the number of learned weights) and thus reduce the storage cost on hard drive. However, since the decomposition methods increase the number of layers of the model, the actual runtime-memory (RAM) cost (which is much more scarce resource than hard drive storage) can be even larger than the model before decomposition.

Table 2: Percentage of Forwarding Time on Non-tensor Layers

Network	Intel x86	Arm	Titan X
AlexNet	32.08%	25.08%	22.37%
GoogLeNet	62.03%	37.81%	26.14%
ResNet-50	55.66%	36.61%	47.87%
ResNet-152	49.77%	N/A	44.49%
<b>Average</b>	<b>49.89%</b>	<b>33.17%</b>	<b>35.22%</b>

for processing a single image with only 0.4% drop on Top-5 accuracy on ImageNet without any weights compression method. DeepRebirth achieves around 65 ms for processing a single image with Top-5 accuracy up to 86.5%.

## 2. Non-tensor layer execution latency

To give a better understanding of the deep learning model execution latency, we evaluate the execution time cost of different types of layers within a given network structure on several major processors (Intel x86 CPU, Arm CPU and Titan X GPU) using state-of-the-art network structures including AlexNet (Figure 2a, [16]), GoogLeNet(Figure 2b, [20]) and ResNet(Figure 2c, [10]).

We define “percentage non-tensor layer latency” (denoted as % Latency) as the time ratio spent on non-tensor layers across the whole network, *i.e.*,

$$\% \text{ Latency} = \frac{\text{Time spent on Non-tensor layer}}{\text{Time spent over the entire network}}, \quad (1)$$

where larger value indicates the larger execution time cost.

**Observations and Insights** The results are shown in Figure 2 and Table 2. We can see, for classical deep models (e.g., AlexNet), among these non-tensor layers, “LRN” and “Pooling” layers are major obstacles that slow-down the model execution. ResNet-50 has abandoned the “LRN” layers by introducing the *batch normalization* layer, but the findings remain valid as it takes up more than 25% of the time on ARM CPU and more than 40% on Intel x86 CPU (in Caffe ([14])), it was decomposed into a “Batch-Norm” layer followed by a “Scale” layer as shown in Figure 2c). The time fraction spent over such layers ranges from 22.37% to 62.03%. Among different types of processors, non-tensor layers have the largest impact on Intel x86 CPUs, and more specifically 62.03% of the computing time. On the other hand, although non-tensor layers do not affect the mainstream ARM CPUs, on average they still cost about 1/3 of the computing time. Therefore, *there is a great potential to accelerate models by optimizing non-tensor layers.*

## 3. DeepRebirth

To accelerate the model execution in non-tensor layers, we propose DeepRebirth to accelerate the model execution

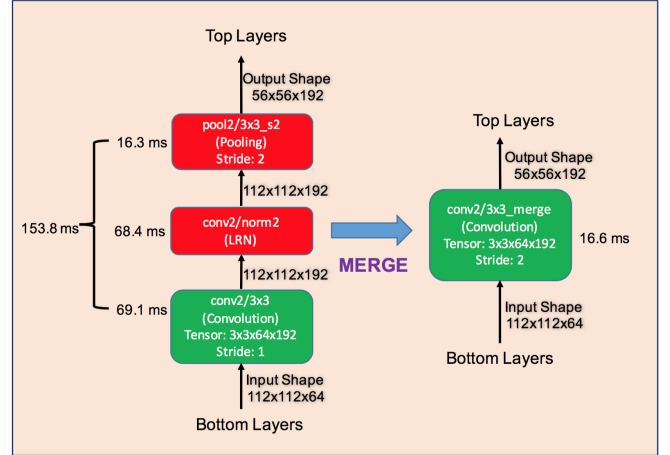


Figure 3: Streamline Slimming: The GoogLeNet example and the running time is measured using `bvlc_googlenet` model in Caffe on a Samsung Galaxy S5. Left panel: convolution (in green), LRN (in red), pooling (in red). Right Panel: single convolution layer. The three layers in the left panel are merged and regenerated as a convolution layer (*i.e.*, slim layer) in the right panel.

at both streaming substructure and branching substructure. The idea of our method is to merge these highly correlated layers and substitute them as a new “slim” layer from the analysis and modeling of the correlations of the current layer and preceding layers (or parallel layers). As in general deep learning models, the probability distribution of the dataset can be represented by these large, sparse layers. This process is similar to viewing the Inception model as a logical culmination as suggested by [2]. DeepRebirth covers several major components: (a) streamline slimming; (b) branch slimming; which will be illustrated in the following.

### 3.1. Streamline Slimming

For deep network architecture with streamline layer connections, in order to accelerate the execution, we first identify the layers which have large latency and redundancy. The slimming design is motivated by the key observations:

- Non-tensor layers usually follow a tensor layer such as convolution layer as shown in Figure 3.

- Several consecutive layers can be viewed as a blackbox for non-linear transformations, and therefore this can be replaced by a new tensor-layer by parameter learning to simulate the functionality of original several layers (Figure 3).

**Method** The streamline slimming regenerates a new tensor layer (*i.e.*, slim layer) by merging non-tensor layers with its bottom tensor units in the feed-forward structure. After layer-wise regeneration, we retrain the deep neural network model by fine-tuning the parameters of the new generated layers. There are two types of streamline slimming in the proposed scheme. The choice of operation depends on the type of non-tensor layers.

- *Pooling Layer*: The pooling layer down-samples feature maps learned from previous layers. Therefore, to absorb a pooling layer to a convolution layer, we remove the pooling layer and set the stride value of the new convolution layer as the product of the stride values for both the original pooling layer and the convolution layer. With a larger stride value for the new slim layer, it further reduces the computation required for executing the new model.

- *Non-Pooling Layer*: For non-pooling layers such as LRN and batch normalization, we directly prune those layers from the original deep neural network.

**Example** Figure 3 illustrates how the streamline slimming works. This is one representative part in GoogLeNet where the convolution layer  $conv2/3 \times 3$  is followed by a LRN layer  $conv2/norm2$  and a pooling layer  $poo2/3 \times 3_{s2}$  (The ReLU layer with negligible latency is retained to keep accuracy). Before processing, the 2 non-tensor layers without a single learned parameter weight take even more time than running the convolution layer. After slimming, we generate a new slim convolution layer  $conv2/3 \times 3_{merge}$ , the time spent on the rebirth layer is greatly reduced compare to original layers.

**Theoretical analysis** Given the input image  $X^i$ , after several tensor and non-tensor layers, we can get the output feature map  $Y_{CNN}^i$ . More mathematically,

$$X^i \xrightarrow{f_{conv}} Y_{cv}^i \xrightarrow{f_{bn}} Y_{cv+bn}^i \xrightarrow{f_{sl}} Y_{cv+bn+sl}^i \xrightarrow{f_{pooling}} Y_{cv+bn+sl+pl}^i \rightarrow \dots := Y_{CNN}^i \quad (2)$$

where  $f_{conv}$ ,  $f_{bn}$ ,  $f_{sl}$ , and  $f_{pooling}$  denote convolution layer, batch normalization layer, scaling layer and pooling layer respectively. There could be other types of layers in the pipeline such as LRN layer  $f_{LRN}$ . The layer parameters are represented by:

$$\begin{cases} f_{conv} : \mathbf{W}_{conv}, \mathbf{B}_{conv}; \\ f_{bn} : m, \mu, \sigma^2; \\ f_{sl} : \gamma, \beta; \\ f_{pooling} : p; \\ f_{LRN} : \kappa, \rho, \alpha. \\ \dots \end{cases} \quad (3)$$

where  $\mathbf{W}_{conv}$ ,  $\mathbf{B}_{conv}$  represent convolution layer weight and bias matrix respectively,  $\mu$ ,  $\sigma^2$  and  $m$  are mean, variance, and sample number in mini-batch of normalization layer  $f_{bn}$ ,  $\gamma$  and  $\beta$  are scaling weight and bias in scaling layer  $f_{sl}$  respectively,  $p$  represents the nearby  $p$  regions in pooling layer  $f_{pooling}$ , and  $\kappa$ ,  $\rho$  and  $\alpha$  are consecutive feature channel parameters and normalization parameters in LRN layer  $f_{LRN}$ .

To achieve the desired functionality with acceleration, the idea is to find a new mapping function

$$\tilde{f}(\tilde{\mathbf{W}}, \tilde{\mathbf{B}}) : X^i \rightarrow Y_{CNN}^i,$$

such that it can get the same feature map value  $Y_{CNN}^i$  given the same input feature map  $X^i$  for any image  $i$ . Note that operations in Eq.(2) transform the feature maps using convolution operations before changing the distributions of activations to avoid ‘‘Internal covariate shift’’ in batch normalization [13] at min-batch level, which can be viewed as a new ‘‘scaling convolution’’ which transforms the input features in the fully connected layers, and therefore we build a single unique convolution operation that replaces several non-tensor layers by setting the new optimization goal, i.e.,

$$\tilde{f}(\tilde{\mathbf{W}}, \tilde{\mathbf{B}}) =: f_{conv}(\tilde{\mathbf{W}}_{conv}, \tilde{\mathbf{B}}_{conv}); \quad (4)$$

Clearly, the optimal solution is given by:

$$(\tilde{\mathbf{W}}^*, \tilde{\mathbf{B}}^*) = \underset{\mathbf{W}, \mathbf{B}}{\operatorname{argmin}} \sum_i \|Y_{CNN}^i - \tilde{f}(\mathbf{W}, \mathbf{B}; X^i)\|_F^2. \quad (5)$$

More formally, we have lemma 3.1.

**Lemma 3.1.** *Given the input/output feature map pairs  $(X^i, Y^i) \forall i$ , operations on the convolution layers followed by non-tensor layers (e.g., normalization layer in Eq. 3) can be re-trained by learning the new convolution layer  $\tilde{f}(\tilde{\mathbf{W}}, \tilde{\mathbf{B}})$  via Eq.(5) using SGD.*

The proof is obvious and therefore we skip it here. In particular, we have lemma 3.2.

**Lemma 3.2.** *Let  $W_j$ ,  $B_j$ ,  $\mu_j$ ,  $\sigma_j^2$ ,  $\gamma_j$  and  $b_j$  be the corresponding  $j$ -th dimension in the reshaped weight vector or bias vector in Eq.(3), and  $\tilde{W}_j$ ,  $\tilde{B}_j$  be the learned new convolution layer parameter in Eq.(5). Then, if  $Y_{CNN}^i$  is obtained after the three layers of  $f_{conv}$ ,  $f_{bn}$ ,  $f_{sl}$  in the sequence order, i.e.,  $Y_{CNN}^i := Y_{cv+bn+sl}^i$ , we have closed form solution for the parameters in the new convolution layer:*

$$\begin{aligned} \tilde{W}_j &= \eta_j W_j, \\ \tilde{B}_j &= \eta_j B_j + \beta_j - \eta_j \frac{\mu_j}{m}, \\ \eta_j &= \frac{\gamma_j}{\sqrt{\frac{\sigma_j^2}{m}}}. \end{aligned} \quad (6)$$

*Proof.* Let  $Y_j$  be the  $j$ -th dimension in feature map after convolution operation in Eqs.(4, 5), i.e.,  $Y_j = (Y_{CNN}^i)_j$ . On one hand, based on the definition of convolution operations (denoted as  $*$ ), we have

$$Y_j = (\tilde{W} * X)_j + \tilde{B}_j. \quad (7)$$

On the other hand, according to the definition of batch nor-



malization [13] and scaling, we have

$$\begin{aligned}
Y_j &= \gamma_j \left( f_{bn} \cdot f_{conv}(X) \right)_j + \beta_j, \quad \triangleright \text{Scaling} \\
&= \gamma_j \left( \frac{f_{conv}(X)_j - \frac{\mu_j}{m}}{\sqrt{\frac{\sigma_j^2}{m}}} \right) + \beta_j, \quad \triangleright \text{BN} \\
&= \gamma_j \left( \frac{(W * X)_j + B_j - \frac{\mu_j}{m}}{\sqrt{\frac{\sigma_j^2}{m}}} \right) + \beta_j. \quad \triangleright \text{Convolution}
\end{aligned} \tag{8}$$

Let  $\eta_j = \frac{\gamma_j}{\sqrt{\frac{\sigma_j^2}{m}}}$ , then Eq.(8) is equivalent to:

$$Y_j = \underbrace{\eta_j (W * X)_j}_{\text{weight}} + \underbrace{\left( \eta_j B_j - \frac{\eta_j \mu_j}{m} + \beta_j \right)}_{\text{bias}}. \tag{9}$$

Compared to Eq.(7), we have  $\tilde{W}_j = \eta_j W_j$  and  $\tilde{B}_j = \eta_j B_j + \beta_j - \eta_j \frac{\mu_j}{m}$ . This completes the proof.  $\square$

### 3.2. Branch Slimming

Given the fact that non-tensor layers require more time on computation, if we can learn new tensor layers by fusing non-tensor layers with the tensor units at the same level, then the the execution time will be decreased. Then we have the deign of *branch slimming*.

**Example** One representative unit is the inception module in GoogLeNet. For example, in Figure 4, layer “inception\_3a” of GoogLeNet has 4 branches: 3 convolution branches take feature maps from the bottom layer at various scales ( $1 \times 1$ ,  $3 \times 3$  and  $5 \times 5$ ) and 1 additional  $3 \times 3$  pooling branch [20]. The output feature maps of each branch are concatenated as input for the following top layer.

**Method** For deep network architecture with parallel branches, the output of each branch constitutes part of the feature maps as the input for the next layer. We identify non-tensor branches that have large latency (e.g., the pooling branch in Figure 4). Similar to streamline slimming, if we can use a faster tensor branch to simulate the function of the non-tensor branch by relearning its parameters, we can achieve clear speed-up.

To absorb a non-tensor branch into a tensor branch, we re-create a new tensor layer (i.e., slim layer) by fusing the non-tensor branch and a tensor unit with relatively small latency to output the feature maps that were originally generated by the non-tensor branch. If the non-tensor branch has a kernel size larger than  $1 \times 1$  (e.g., the  $3 \times 3$  pooling branch in Figure 4), the picked tensor branch’s kernel size should be at least the size of the non-tensor branch. As shown in this figure, we re-learn a new tensor layer “inception\_3a” by merging the  $3 \times 3$  pooling branch with the  $5 \times 5$  convolution branch at the same level, and the number of feature maps obtained by the  $5 \times 5$  convolution is increased from 32 to 64.

- **Branch Reducing:** Current deep neural networks usually include convolution branches with  $1 \times 1$  convolution layers (e.g., inception\_3a/3x3\_reduce in Figure 4) aiming to reduce feature maps channels. This unit will be processed by a following convolution layer with larger kernel size. For greater speed-up, we further reduce the number of feature maps generated by the  $1 \times 1$  “reducer”. For layer inception\_3a/3x3\_reduce, we reduce the number of output feature maps from 96 to 48.

- **Tensor-Branch Slimming:** A convolution branch with a smaller kernel size can be absorbed to a convolution branch with a larger kernel size. The method is similar to the slimming of non-tensor branches. To keep other layers’ structures in network unchanged, we remove the small-kernel convolution branch and increase the number of feature maps generated by the large-kernel convolution layers. For examples, for layer inception\_3a/3x3\_reduce, we remove the  $1 \times 1$  convolution branch and increase the number of feature maps generated by the  $3 \times 3$  convolution from 128 to 196.

#### Consecutive convolutional layer slimming analysis

Let  $f^k$  be the  $k$ -th convolution layer<sup>3</sup> with network weight parameters  $\mathbf{W}^k$  and  $\mathbf{B}^k$ , the output feature map after  $k$ -th convolution layers is:

$$\begin{cases} Y^i = f^k \cdot f^{k-1} \cdot f^{k-2} \dots f^1(X^i), \\ f^k : \mathbf{W}^k, \mathbf{B}^k. \quad \forall k \end{cases} \tag{10}$$

The idea is to use one single convolution layer to represent the feature transformations using multiple layers, i.e., we wish to find a function  $\hat{f}(\hat{\mathbf{W}}, \hat{\mathbf{B}})$  such that:

$$Y^i = \hat{\mathbf{W}} * X^i + \hat{\mathbf{B}}. \tag{11}$$

Fortunately, we have lemma 3.3.

**Lemma 3.3.** *Operations on several consecutive convolution layers can be learned by re-training a new convolution structure  $\hat{W}^*$  via Eqs.(12), i.e.,*

$$\begin{aligned}
\hat{\mathbf{W}}^* &= \mathbf{W}^{k-1} * \mathbf{W}^{k-2} * \dots * \mathbf{W}^1, \\
\hat{\mathbf{B}}^* &= \left( \mathbf{W}^{k-1} * \mathbf{W}^{k-2} * \dots * \mathbf{W}^2 \right) * \mathbf{B}_1 \\
&\quad + \left( \mathbf{W}^{k-1} * \mathbf{W}^{k-2} * \dots * \mathbf{W}^3 \right) * \mathbf{B}_2 \\
&\quad + \dots + \mathbf{B}_k.
\end{aligned} \tag{12}$$

**Branch convolutional layer rebirth analysis** Let  $Y_L$  and  $Y_R$  be the feature map learned using convolution layers respectively given model parameter weight and bias, i.e.,

$$\begin{cases} Y_L^i = \mathbf{W}_L * X^i + \mathbf{B}_L; \quad \triangleright \text{left branch} \\ Y_R^i = \mathbf{W}_R * X^i + \mathbf{B}_R; \quad \triangleright \text{right branch} \end{cases} \tag{13}$$

<sup>3</sup>  $f^k = f_{conv}^k$ , for notation simplicity purpose, we ignore all subscript.

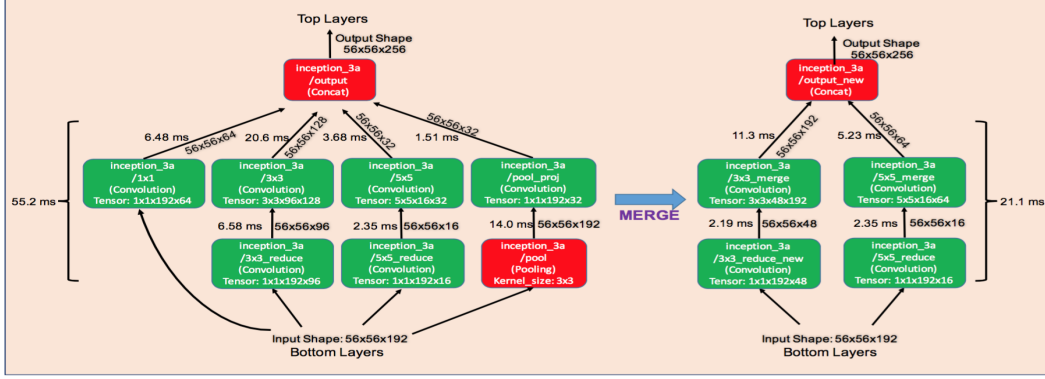


Figure 4: Branch Slimming: The GoogLeNet example and the running time is measured using `bvlc_googlenet` model in Caffe on a Samsung Galaxy S5. Left panel: four branches in parallel, convolution layer, convolution + convolution, convolution + convolution, convolution + pooling. Right panel: two branches in parallel, convolution + convolution, convolution + convolution. The four branches are merged into two branches.

Let  $Y_L^i$  be the concatenation of feature maps in left and right branches. We wish to learn a new convolution function  $\hat{f}(\mathbf{W}_{LR}, \mathbf{B}_{LR})$ , such that

$$Y_{LR}^i = [Y_{\text{left}}^i; Y_{\text{right}}^i], \quad Y_{LR}^i = \mathbf{W}_{LR} * X^i + \mathbf{B}_{LR}, \quad (14)$$

with  $Y_L^L \in \mathbf{R}^{M' \times N' \times K'_L}$  and  $Y_L^R \in \mathbf{R}^{M' \times N' \times K'_R}$  having the same kernel size.

If  $\mathbf{W}_L$  and  $\mathbf{W}_R$  have the same kernel size, we can get

$$\mathbf{W}_{LR} = [\mathbf{W}_{\text{left}}; \mathbf{W}_{\text{right}}], \quad \mathbf{B}_{LR} = [\mathbf{B}_{\text{left}}; \mathbf{B}_{\text{right}}]. \quad (15)$$

by substituting Eq.(13) into Eq.(14). Otherwise, we need to adjust  $Y_L$  and  $Y_R$  to the same size and learn the model parameters by minimizing:

$$(\hat{\mathbf{W}}_{LR}^*, \hat{\mathbf{B}}_{LR}^*) = \underset{\mathbf{W}, \mathbf{B}}{\operatorname{argmin}} \sum_i \|Y_{LR}^i - (\hat{\mathbf{W}} * X^i + \hat{\mathbf{B}})\|_F^2.$$

### 3.3. Adapting DeepRebirth to overall pipeline

To reconcile the new learned layer with other parts of model, one further step is to fine-tuning the model parameters, as suggested in [22, 17]. In DeepRebirth, we leverage Xavier [6] initialization to initialize the parameters in the new layer while keeping the weights of other layers unchanged. In the optimization procedure, we set the learning rate of new layers 10 times over those in other layers empirically. The proposed optimization scheme is applied from the bottom layer to the top layer. Another alternative is to learn multiple slim layers at the same time (we merge and fine-tune 3 sequential inception layers 4b-4d together for GoogLeNet) or merge layers in sequential orders other than from bottom to top.

## 4. Evaluation

To evaluate the performance of DeepRebirth, we performed the comprehensive evaluation on top of GoogLeNet, AlexNet and ResNet.

### 4.1. GoogLeNet

We use Caffe’s GoogLeNet implementation (i.e., `bvlc_googlenet`) with its pre-trained model weights. Then we apply the proposed DeepRebirth optimization scheme to accelerate the running speed of GoogLeNet, which is denoted as “GoogLeNet-Slim”. After non-tensor layer optimization (streamline and branch slimming), we further apply tucker decomposition approach ([15]) to reduce the model size (i.e., the number of learned weights) by 50%, represented as “GoogLeNet-Slim-Tucker”. In addition, we directly employ tucker decomposition method to compress original GoogLeNet. This is indicated as “GoogLeNet-Tucker”. Thus, we have 4 variations of GoogLeNet to compare, namely GoogLeNet, GoogLeNet-Slim, GoogLeNet-Tucker and GoogLeNet-Slim-Tucker. We also compare with SqueezeNet [12], a state-of-the-art compact neural network which includes only 1.2M learnable parameters (vs. 5M for GoogLeNet).

**Accuracy** We evaluate the accuracy loss in contrast to original ones after performing the accelerated models. The accuracy changing along with the optimization steps conducted on ImageNet ILSVRC-2012 validation dataset are listed in Table 3. During the whole optimization procedure of model training, we set the base learning rate for the re-generated layer as 0.01 (the rest layers are 0.001). We apply stochastic gradient descent training method ([3]) to learn the parameters with a batch size of 32. During our training phase, we set 40,000 as the step size together with 0.1 for gamma value and 0.9 for momentum parameter. At each step, the model generally converges at around 90,000 iterations (2 epochs).

The result indicates that DeepRebirth has almost negligible impact on the model accuracy, and the accuracy even increases at certain step (e.g., step 5). This indicates that “the new-born” layers perfectly simulate the functionalities of previous non-tensor layers before optimization. By ap-

Table 3: GoogLeNet Accuracy on Slimming Each Layer

Step	Slim Layer(s)	Top-5 Accuracy
0	N/A	88.89%
1	conv1	88.73%
2	conv2	88.82%
3	inception_3a	88.50%
4	inception_3b	88.27%
5	inception_4a	88.60%
6	inception_4b-4d	88.61%
7	inception_4e	88.43%
8	inception_5a	88.41%
9	inception_5b	<b>88.43%</b>
Tucker Decomposition		<b>86.54%</b>

plying tucker decomposition method on the slim model to reduce the weights by half (GoogLeNet-Slim-Tucker), we observe that there is a larger drop on accuracy (around 2%). However, directly applying tucker decomposition method (GoogLeNet-Tucker) to reduce the GoogLeNet weights to a half drops the top-5 accuracy to 85.7%. These results imply that our method performs reasonable well even after streamline and branch slimming.

**Speed-Up** To evaluate and compare the latency of different optimization approaches, we evaluate the layer-wise running speed on a Samsung Galaxy S5 smart phone which has an ARMv7 quad-core CPU @ 2.5 GHz and 2 GB RAM. We use Caffe’s integrated benchmark module to test the model forwarding time. Each test run includes 50 subtests with a random input. We try 10 test runs on each compared model and report the best test run in terms of forwarding time. During the whole experiment, we turn on the airplane mode and close all other apps.

As demonstrated in Table 4, we observe that GoogLeNet-Slim is 3x faster than GoogLeNet. In addition, as pointed [15], the original GoogLeNet model has too many small layers and this results in performance fluctuation. In the worst scenario, GoogLeNet takes around 950 ms for a single forwarding while with reduced number of layers, GoogLeNet-Slim takes only up to 250 ms, which is almost 4x speed-up. The Tucker Decomposition method further reduces the computation for around 50% at the cost of around 2% accuracy loss. On the other hand, directly applying tucker decomposition on tensor layers doesn’t show any significant acceleration.

We evaluate the speed-up on other popular processors besides Galaxy S5, including (1) Moto E: a low-end mobile ARM CPU, (2) Samsung Galaxy S5: a middle-end mobile ARM CPU, (3) Samsung Galaxy S6: a high-end mobile ARM CPU, (4) Macbook Pro: an Intel x86 CPU, and (5) Titan X: a powerful server GPU. We demonstrate the experimental results in Table 5. We see the significant speed-up on various types of CPUs. Even on the low-end mobile CPU (i.e., Moto E), around 200 ms model forwarding time is achieved by combining tensor weights compression method. Finally, comparing the proposed approach with

Table 4: Breakdown of GoogLeNet forwarding time cost using different methods on each layer.

Layer	GoogLeNet[20]	GoogLeNet-Tucker[15]	GoogLeNet-Slim (ours)	GoogLeNet-Slim-Tucker (ours)
conv1	94.92 ms	87.85 ms	8.424 ms	6.038 ms
conv2	153.8 ms	179.4 ms	16.62 ms	9.259 ms
inception_3a	55.23 ms	85.62 ms	21.17 ms	9.459 ms
inception_3b	98.41 ms	66.51 ms	25.94 ms	11.74 ms
inception_4a	30.53 ms	36.91 ms	16.80 ms	8.966 ms
inception_4b	32.60 ms	41.82 ms	20.29 ms	11.65 ms
inception_4c	46.96 ms	30.46 ms	18.71 ms	9.102 ms
inception_4d	36.88 ms	21.05 ms	24.67 ms	10.05 ms
inception_4e	48.24 ms	32.19 ms	28.08 ms	14.08 ms
inception_5a	24.64 ms	14.43 ms	10.69 ms	5.36 ms
inception_5b	24.92 ms	15.87 ms	14.58 ms	6.65 ms
loss3	3.014 ms	2.81 ms	2.97 ms	2.902 ms
<b>Total</b>	<b>651.4 ms</b>	<b>614.9 ms (1.06x)</b>	<b>210.6 ms (3.09x)</b>	<b>106.3 ms (6.13x)</b>

Table 5: Execution time using different methods (including SqueezeNet) on different mobile devices

Device	GoogLeNet[20]	GoogLeNet-Tucker[15]	GoogLeNet-Slim (ours)	GoogLeNet-Slim-Tucker (ours)	SqueezeNet[12]
Moto E	1168.8 ms	897.9 ms	406.7 ms	<b>213.3 ms</b>	291.4 ms
Samsung Galaxy S5	651.4 ms	614.9 ms	210.6 ms	<b>106.3 ms</b>	136.3 ms
Samsung Galaxy S6	424.7 ms	342.5 ms	107.7 ms	<b>65.34 ms</b>	75.34 ms
Macbook Pro (CPU)	91.77 ms	78.22 ms	23.69 ms	<b>15.18 ms</b>	17.63 ms
Titan X	10.17 ms	10.74 ms	6.57 ms	7.68 ms	<b>3.29 ms</b>

SqueezeNet ([12]), we are very excited to see that our optimization approach can obtain faster speed on all mobile devices with much higher accuracy (the Top-5 accuracy for SqueezeNet is 80%) as listed in Table 5.

**Energy, Storage and Runtime-Memory Cost** We measure the energy cost of each compared model using PowerTutor Android app ([23]) on Samsung Galaxy S6 (similar results are obtained on other mobile devices). The original GoogLeNet consumes almost 1 Joule per image while GoogLeNet-Slim consumes only 447 mJ. Applying tucker decomposition further reduces the energy cost to only 1/4 at 226 mJ. When deploying to the mobile devices, we remove the loss1 and loss2 branches from the trained models so that the storage cost of each model is reduced by 24.33 MB. GoogLeNet-Slim which achieves significant speed-up does not save much storage cost compared to the original GoogLeNet model. However, for modern mobile devices, storage is not a scarce resource (e.g., Samsung Galaxy S5 has 16 GB or 32 GB storage), so a 20 MB deep learning model is “affordable” on mobile devices. Meanwhile, we can always perform the tensor weights compression method to further reduce the storage cost.

Another benefit of layer slimming is run-time memory saving. The generated GoogLeNet-Slim model reduces the number of layers and consumes only 13.2 MB to process one image. This feature is also very useful for the cloud based deep learning service which can process a much larger batch at one run. As shown in table 6, one Titan X GPU can run a batch size of 882 with the GoogLeNet-Slim model while the original GoogLeNet can only allow a batch size of 350. On the other hand, SqueezeNet though has much less trained parameters, it has much larger run-time

Table 6: GoogLeNet Execution Storage vs. Energy vs. Runtime-Memory Cost

Model	Energy	Storage	Runtime Memory	Max Batch Size on Titan X
GoogLeNet[20]	984 mJ	26.72 MB	33.2 MB	350
GoogLeNet-Tucker[15]	902 mJ	14.38 MB	35.8 MB	323
GoogLeNet-Slim (ours)	<b>447 mJ (2.2x)</b>	23.77 MB	13.2 MB	<b>882 (2.52x)</b>
GoogLeNet-Slim-Tucker (ours)	<b>226 mJ (4.4x)</b>	11.99 MB	14.8 MB	<b>785 (2.24x)</b>
SqueezeNet[12]	288 mJ	4.72 MB	36.5 MB	321

Table 7: AlexNet Result (Accuracy vs. Speed vs. Energy cost)

Step	Slim Layer(s)	Top-5 Accuracy	Speed-up	Energy Cost
0	N/A	80.03%	445 ms	688 mJ
1	conv1+norm1 → conv1	79.99%	343 ms (1.29x)	555 mJ (1.24x)
2	conv2+norm2 → conv2	79.57%	274 ms (1.63x)	458 mJ (1.51x)

Table 8: ResNet (conv1-res2a) Result (Accuracy vs. Speed up). For each step, we absorb the “BatchNorm” and “Scale” layers to the bottom convolution layer.

Step	Slim Layer(s)	Top-5 Accuracy	Speed-up	Runtime-Mem Batch32
0	N/A	92.36%	189 ms	2505 MB
1	conv1	92.13%	162 ms (1.17x)	2113 MB (1.19x)
2	res2a_branch1	92.01%	140 ms (1.35x)	1721 MB (1.46x)
3	res2a_branch2a-2c	91.88%	104 ms (1.82x)	1133 MB (2.21x)

memory impact due to the increased number of layers.

## 4.2. AlexNet and ResNet

We apply the proposed framework to other popular deep neural structures: AlexNet ([16]) and ResNet ([10]). Note that we did not apply tensor weights compression to those two models which can further reduce the model forwarding latency. First, we study the classical AlexNet model. We apply streamline slimming approach to re-generate new slim layers by merging the first two convolution layers followed by LRN layers. We illustrate the result in Table 7. This indicates that by applying slimming to the first two layers, the model forwarding time of AlexNet is reduced from 445 ms to 274 ms on Samsung Galaxy S5, and the Top-5 accuracy is slightly dropped from 80.03% to 79.57%.

We apply the acceleration scheme to the state-of-the-art ResNet model. In the experiment, we use the popular 50-layer ResNet-50 model as baseline. We mainly apply the acceleration framework to conv1 and res2a layers (res2a has 2 branches; one branch has 1 convolution layer and another branch has 3 convolution layers). We present the result in Table 8. The time latency on Samsung Galaxy S5 for the processed layers (i.e., conv1 and res2a) is reduced from 189 ms to 104 ms. Moreover, the run-time memory cost is reduced by 2.21x. The accuracy is only slightly reduced.

## 5. Related Work

Reducing the model size and accelerating the running speed are two general ways to facilitate the deployment of deep learning models on mobile devices. Many efforts have been spent on improving the model size. In particular,

most works focus on optimizing tensor-layers to reduce the model size due to the high redundancy in the learned parameters in tensor layers of a given deep model. Vanhoucke et al. [21] proposed a fixed-point implementation with 8-bit integer activation to reduce the number of parameter used in the deep neural network while [7] applied vector quantization to compressed deep convnets. These approaches, however, mainly focus on compressing the fully connected layer without considering the convolutional layers. To reduce the parameter size, Denten et al. [5] applied the low-rank approximation approach to compress the neural networks with linear structures. Afterwards, hashing function was utilized to reduce model sizes by randomly grouping connection weights [4]. More recently, Han et al. [9] proposed to effectively reduce model size and achieve speed-up by the combination of pruning, huffman coding and quantization. However, the benefits can only be achieved by running the compressed model on a specialized processor [8].

Recently, SqueezeNet [12] has become widely used for its much smaller memory cost and increased speed. However, the near-AlexNet accuracy is far below the state-of-the-art performance. Compared with these two newly networks, our approach has much better accuracy with more significant acceleration. Springenberg et al. [19] showed that the conv-relu-pool substructure may not be necessary for a neural network architecture. The authors find that max-pooling can simply be replaced by another convolution layer with increased stride without loss in accuracy. Different from this work, DeepRebirth replaces a complete substructure (e.g., conv-relu-pool, conv-relu-LRN-pool) with a single convolution layer, and aims to speed-up the model execution on the mobile device. In addition, our work slims a well-trained network by relearning the merged rebirth layers and does not require to train from scratch. Essentially, DeepRebirth can be considered as a special form of distillation [11] that transfers the knowledge from the cumbersome substructure of multiple layers to the rebirthed accelerated substructure.

## 6. Conclusion

An acceleration framework – DeepRebirth is proposed to speed up the neural networks with satisfactory accuracy, which operates by re-generating new tensor layers from optimizing non-tensor layers and their neighborhood units. DeepRebirth is also compatible with state-of-the-art deep models like GoogleNet and ResNet, where most parameter weight compression methods failed. By applying DeepRebirth at different deep learning architectures, we obtain significant speed-up on different processors (including mobile CPUs), which will greatly facilitate the deployment of deep learning models on mobile devices in the new AI tide.



## References

- [1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org. **1**
- [2] S. Arora, A. Bhaskara, R. Ge, and T. Ma. Provable bounds for learning some deep representations. *CoRR*, abs/1310.6343, 2013. **3**
- [3] L. Bottou. *Stochastic Gradient Tricks*, volume 7700, page 430445. Springer, January 2012. **6**
- [4] W. Chen, J. T. Wilson, S. Tyree, K. Q. Weinberger, and Y. Chen. Compressing neural networks with the hashing trick. *CoRR*, abs/1504.04788, 2015. **8**
- [5] E. L. Denton, W. Zaremba, J. Bruna, Y. LeCun, and R. Fergus. Exploiting linear structure within convolutional networks for efficient evaluation. In *Advances in Neural Information Processing Systems*, pages 1269–1277, 2014. **8**
- [6] X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. In *In Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS10). Society for Artificial Intelligence and Statistics*, 2010. **6**
- [7] Y. Gong, L. Liu, M. Yang, and L. Bourdev. Compressing deep convolutional networks using vector quantization. *arXiv preprint arXiv:1412.6115*, 2014. **8**
- [8] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally. Eie: Efficient inference engine on compressed deep neural network. *International Conference on Computer Architecture (ISCA)*, 2016. **8**
- [9] S. Han, H. Mao, and W. J. Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *International Conference on Learning Representations (ICLR)*, 2016. **2, 8**
- [10] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. *arXiv preprint arXiv:1512.03385*, 2015. **3, 8**
- [11] G. Hinton, O. Vinyals, and J. Dean. Distilling the Knowledge in a Neural Network. *ArXiv e-prints*, Mar. 2015. **8**
- [12] F. N. Iandola, M. W. Moskewicz, K. Ashraf, S. Han, W. J. Dally, and K. Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <1mb model size. *arXiv:1602.07360*, 2016. **2, 6, 7, 8**
- [13] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*, pages 448–456, 2015. **4, 5**
- [14] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014. **1, 3**
- [15] Y. Kim, E. Park, S. Yoo, T. Choi, L. Yang, and D. Shin. Compression of deep convolutional neural networks for fast and low power mobile applications. *CoRR*, abs/1511.06530, 2015. **2, 6, 7, 8**
- [16] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, page 2012. **3, 8**
- [17] A. S. Razavian, H. Azizpour, J. Sullivan, and S. Carlsson. CNN features off-the-shelf: an astounding baseline for recognition. *CoRR*, abs/1403.6382, 2014. **6**
- [18] sh1r0, zif520, and strin. BWorld Robot Control Software. <https://github.com/sh1r0/caffe-android-lib/issues/23>, 2015. [Online; accessed 19-July-2016]. **1**
- [19] J. T. Springenberg, A. Dosovitskiy, T. Brox, and M. A. Riedmiller. Striving for simplicity: The all convolutional net. *CoRR*, abs/1412.6806, 2014. **8**
- [20] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. E. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. *CoRR*, abs/1409.4842, 2014. **2, 3, 5, 7, 8**
- [21] V. Vanhoucke, A. Senior, and M. Z. Mao. Improving the speed of neural networks on cpus. 2011. **8**
- [22] J. Yosinski, J. Clune, Y. Bengio, and H. Lipson. How transferable are features in deep neural networks? *CoRR*, abs/1411.1792, 2014. **6**
- [23] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. M. Mao, and L. Yang. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, CODES/ISSS '10*, pages 105–114, New York, NY, USA, 2010. ACM. **7**