DEEPREBIRTH: A GENERAL APPROACH FOR ACCELERATING DEEP NEURAL NETWORK EXECUTION ON MOBILE DEVICES

Dawei Li
Computer Science and Engineering Department
Lehigh University
Bethlehem, PA 18015, USA
dal312@lehigh.edu

Xiaolong Wang
Samsung Research America (SRA)
visionxiaolong@gmail.com

Deguang Kong
Samsung Research America (SRA)
doogkong@gmail.com

Mooi Choo Chuah
Computer Science and Engineering Department
Lehigh University
Bethlehem, PA 18015, USA
chuah@cse.lehigh.edu

ABSTRACT

Deploying deep neural networks on mobile devices is a challenging task due to computation complexity and memory intensity. Existing works solve this problem by reducing model size using weight compression methods based on dimension reduction (i.e., SVD, Tucker decomposition and Quantization). However, the execution speed of these compressed models are still far below the real-time processing requirement of mobile services. To address this limitation, we propose a novel acceleration framework: DeepRebirth by exploring the deep learning model parameter sparsity through merging the parameter-free layers with their neighbor convolution layers to a single dense layer. The design of DeepRebirth is motivated by the key observation: *some layers (i.e., normalization and pooling) in deep learning models actually consume a large portion of computational time even few learned parameters are involved, and acceleration of these layers has the potential to improve the processing speed significantly.* Essentially, the functionality of several merged layers is replaced by the new dense layer – rebirth layer in DeepRebirth. In order to preserve the same functionality, the rebirth layer model parameters are re-trained to be functionality equivalent to the original several merged layers. The extensive experiments performed on ImageNet using several popular mobile devices demonstrate that DeepRebirth is not only providing huge speed-up in model deployment and significant memory saving but also maintaining the model accuracy, i.e., 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 single image using Samsung Galaxy S6 with only 2.4% accuracy drop. In addition, 2.5x run-time memory saving is achieved with rebirth layers.

1 INTRODUCTION

Recent years have witnessed the breakthrough of deep learning techniques for image classification and object recognition. Mobile device becomes more and more popular due to its convenient mobiles services provided for end users. More and more mobile applications require deep learning techniques to provide accurate, intelligent and effective services. However, the execution speed of the deep learning model on mobile devices becomes a bottleneck for many applications due to the large model size, deep network structure and complicated model parameters, which hinders the real-time deployment. However, if deep learning service is only provided at cloud side, transmis-

sion of images though internet may compromise image owners' privacy and is also limited by the availability of the Internet.

Running deep learning models efficiently on mobile CPUs is a highly intriguing feature due to many reasons: (1) CPU is available for all mobile devices, even phones released many years ago; (2) powerful CUDA-enabled GPUs are generally not available on (compact) mobile devices; (3) though a large majority of mobile devices are equipped with mobile GPUs, the speed-up achieved on the mobile GPUs is quite limited when compared to CPU sh1r0 et al. (2015), not to mention the complexity caused by different mobile GPU architectures; (4) major deep learning frameworks such as Caffe Jia et al. (2014) and Tensorflow Abadi et al. (2015) only support CPU implementation on mobile devices currently, and therefore an efficient CPU-friendly model is highly desirable.

However, most of current mobile CPUs cannot meet the needs of deep learning model deployment because it takes much longer time and higher energy cost to process an image using pre-trained deep learning models. For example, it takes more than 651ms to recognize an image using GoogleNet on Samsung S5 (Table 4) with 984mJ energy costs (Table 5). Therefore a question that naturally follows is: *can we develop an efficient deep learning acceleration framework to facilitate deployment of deep learning service on mobile device?*

This problem is challenging due to the fact that the practical solution is highly desirable to support different practical scenarios by addressing the following challenges (C1–C3).

C1: Minimum accuracy loss. The solution is expected to provide minimum accuracy loss while provide great speed-up.

C2: Leveraging existing trained deep framework. In order to provide the best deep learning service, the mechanism is designed to taking advantage of existing state-of-the-art deep learning architectures (e.g., GoogLeNet and ResNet) instead of training from scratch.

C3: Supporting different deep learning architecture components. The proposed technique should provide generic framework that can be applied to these popular deep learning models that may consist of different types of layers. In general, all neural network layers can be grouped into two categories: *tensor layer* and *non-tensor layer* based on whether the layer contains tensor-type parameters. For example, fully connected layer and the convolution layer are both tensor-layers since they contain 2-d and 4-d tensor-type weight parameters, respectively. Pooling layer and LRN layer are both non-tensor layers because they do not contain any high-order tensor-type weight parameters[1]. Therefore, the framework is expected to support both tensor and non-tensor layers optimization.

However, the current solutions for deep learning model acceleration are still quite limited in addressing these challenges. The main goal of works (Han et al. (2016b), Li (2013); Kim et al. (2015); Jiaxiang Wu & Cheng (2016)) is to reduce the model size by approximating the tensor-type layers using low rank approximation and vector quantization techniques. While they can provide some acceleration for *only* fully-connected layers (used in AlexNet, VGGNet), the application scenarios of these methods are very limited and ineffective because modern 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, none works, to the best of our knowledge, have discussed how to accelerate the execution process.

To bridge these gaps, this paper proposes DeepRebirth, a new deep learning model acceleration framework by exploring the sparsity of deep neural network layers to accelerate both non-tensor layers and tensor layers from two types of rebirth: *streaming merging* and *branch merging*. In streaming merging, the new tensor layers are generated by merging non-tensor layers with its neighboring sparse tensor layers in the feed-forward structure as illustrated in Figure 2, while in branch merging, the new tensor layers are created by fusing non-tensor banches with the sparse tensor branches (at the same level) as shown in Figure 3, i.e., the inception module in GoogLeNet (Szegedy et al. (2014)). The design of DeepRebirth is guided by the key observation:

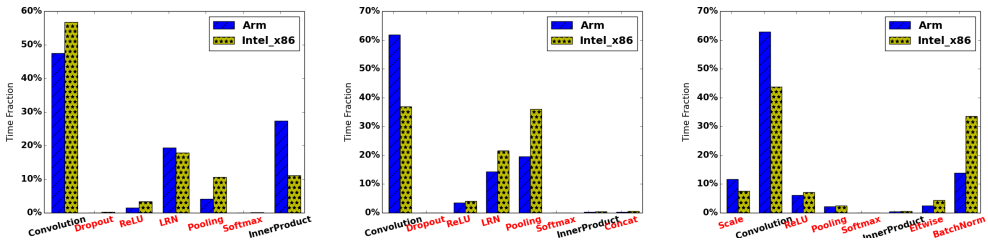
Non-tensor layers are the major obstacles for real-time mobile CPU execution (Section 2).

Then reducing the execution time on non-tensor layers can greatly reduce the overall model forwarding time. In order to reduce the execution time, both *streaming merging* and *branch merging*

[1]Other examples of non-tensor layers include dropout layer, normalization layer, softmax layer, etc.

Table 1: 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%
Average	49.89%	33.17%	35.22%



(a) AlexNet (b) GoogLeNet (c) ResNet-50

Figure 1: Time Decomposition for each layer. Non-tensor layers (e.g., dropout, ReLU, LRN, softmax, pooling, etc) shown in red color while tensor layers (e.g., convolution, inner-product) shown in black color.

are applied to merge non-tensor layers into tensor layers. 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 (Han et al. (2016b), Kim et al. (2015)). Ideally, we can combine both non-tensor and tensor layer optimization together and further reduce latency as well as the model size. To summarize, this paper makes the following contributions.

· Our approach is the first work that optimizes non-tensor layers and significantly accelerates a deep learning model on CPUs while reducing the required runtime-memory since there are less layers in the reconstructed deep learning model[2].

· To address the challenges of (C1–C3), we perform both streaming merging and branch merging based on the original structure of old layers while the new tensor layers are generated by merging non-tensor layers with its neighboring sparse tensor layers vertically and horizontally.

· As demonstrated in the experiment, our approach has obtained the state-of-the-art speeding up on popular deep learning models with negligible accuracy loss. Our proposed method enables GoogLeNet to achieve 3x-5x speed-up for processing a single image with only 0.4% drop on Top-5 accuracy on ImageNet without any weights compression method. By further applying model compression techniques, we achieve around 65 ms for processing a single image with Top-5 accuracy of 86.5%. Furthermore, we show that our methods work for state-of-the-art non-tensor layers, e.g., batch normalization, in very deep neural network models such as ResNet He et al. (2015).

2 What is the impact of non-tensor layers on speed?

Experimental Settings To give a better understanding of the neural network latency, we evaluate the time cost of different types of layers within a given network. We measure their latency by using the time percentage measurement where larger value indicates longer time [3]. Our experiment is carried on different processors including Intel x86 CPU, Arm CPU and Titan X GPU. Along with different processors, we also use different state-of-the-art networks to evaluate. These networks

[2]Tensor weights decomposition 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.

[3]The accumulated percentage for a given network is 100%.

include AlexNet (Figure 1a, Krizhevsky et al.), GoogLeNet(Figure 1b, Szegedy et al. (2014)) and ResNet(Figure 1c, He et al. (2015)). We list the results in Figure 1 and Table 1.

Observations and Insights As demonstrated in the results, for classical deep models (e.g., AlexNet), among the non-tensor layers, "LRN" and "Pooling" layers are the 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 (Jia et al. (2014)), it was decomposed into a "BatchNorm" layer followed by a "Scale" layer as shown in Figure 1c). 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, though non-tensor layers do not affect the mainstream ARM CPUs, on average they still cost about 1/3 of the computing time. All these numbers confirm our intuition: *there is a great potential to accelerate the model by optimizing those non-tensor layers.*

3 DEEPREBIRTH

This section covers the design of DeepRebirth in three aspects: streaming merging, branching merging and adapting DeepRebirth to the whole model.

In general deep learning models, the probability distribution of the dataset can be represented by a large, very sparse deep neural network that is constructed layer after layer. From analyzing the correlations of the current layer and preceding layers (or parallel layers), we can merge the highly correlated layers and substitute it as a new "rebirth" layer. This process is similar to viewing the Inception model as a logical culmination as suggested by Arora et al. (2013).

3.1 STREAMLINE MERGING

For deep network architecture with streamline layer connections, in order to accelerate the execution, we first identify the layers that have large latency but also have potentials to be merged or processed. The merging design is motivated by the following two key observations.

- Non-tensor layers are usually following a tensor layer such as convolution layer as shown in Figure 2.
- 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 learning the parameters to approximate the functionality of original several layers. An example is shown in Figure 2.

Method The streamline merging regenerates a new tensor layer (i.e., rebirth 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 streamline merging operations in the proposed scheme. The choice of merging operation is depending on the type of non-tensor layers.

- *Merging Pooling Layer*: The pooling layer down-samples feature maps learned from previous layers. Therefore, to merge a pooling layer to a convolution layer, we remove the pooling layer and set the stride value of the "merged" 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 "merged" convolution layer, it further reduces the computation required for executing the new model.
- *Merging 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 2 illustrates how the optimization works using streamline merging. 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 which has negligible latency is retained to keep accuracy). Before merging, the 2 non-tensor layers without a single learned parameter weight take even more time than running the convolution layer. After merging

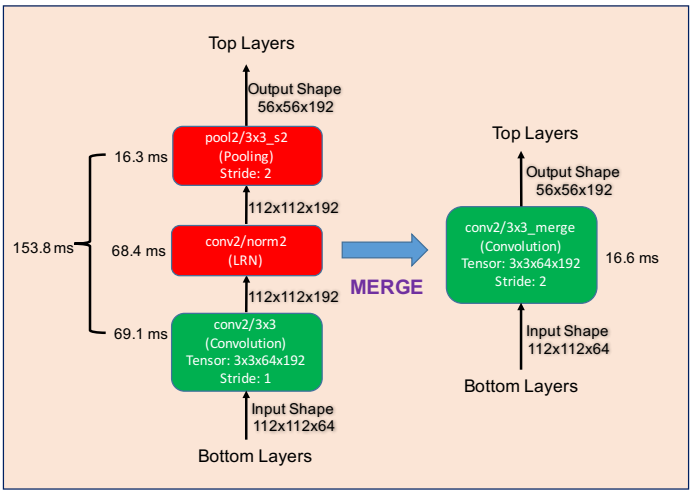


Figure 2: Streamline Merging: The GoogLeNet example and the running time is measured using bvlc_googlenet model in Caffe on a Samsung Galaxy S5. Left panel: convolution (in green), LRN (in red), pooling (in red). Right Panel: single convolution layer. The three layers in the left panel are merged and regenerated as a convolution layer (i.e., rebirth layer) in the right panel.

process them to generate a new rebirth convolution layer $conv2/3 \times 3\_merge$, the time spent on the rebirth layer is greatly reduced compare to the original layers.

3.2 Branch Merging

The design of branch merging is motivated by the following key observation. Given the fact that non-tensor layer requires more time on computation, if we can learn new tensor layers by fusing non-tensor layers with the tensor units at the same layer level, then the the execution time will be decreased.

Example One representative unit is the inception module in GoogLeNet. For example as illustrated in Figure 3, layer "inception_3a" of GoogLeNet has 4 branches: 3 convolution branches take feature maps from the bottom layer at various scales (1×1, 3×3 and 5×5) and 1 additional 3×3 pooling branch Szegedy et al. (2014). 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 3). Similar to streamline merging, 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 merge a non-tensor branch into a tensor branch, we re-create a new tensor layer (i.e., rebirth 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 1×1 (e.g., the 3×3 pooling branch in Figure 3), 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 3×3 pooling branch with the 5×5 convolution branch at the same level, and the number of feature maps obtained by the 5×5 convolution is increased from 32 to 64.

Explore Sparsity for Tensor-layer Branch Reducing and Merging

- *Reducing*: Current deep neural networks usually include convolution branches with 1×1 convolution layers (e.g., inception_3a/3x3_reduce in Figure 3) 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

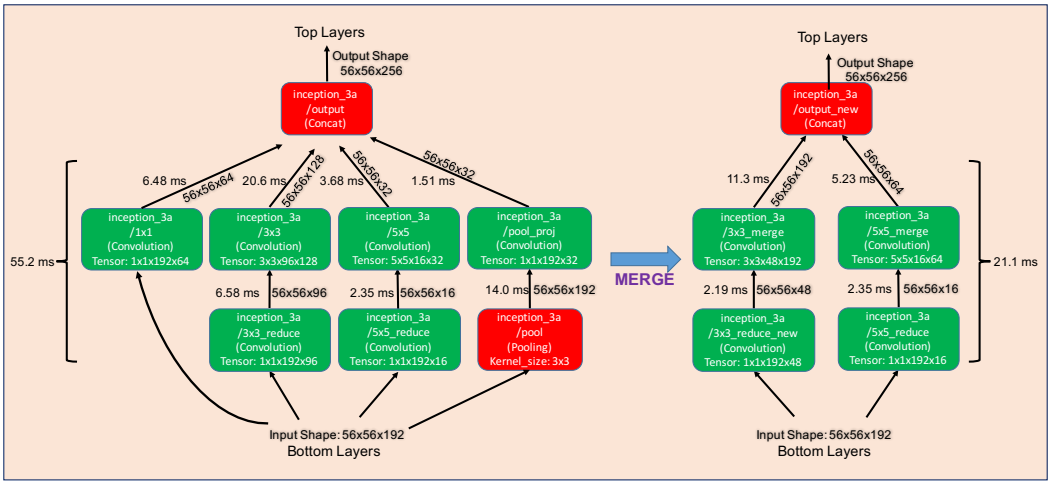


Figure 3: Branch Merging: The GoogLeNet example and the running time is measured using bvlc_googlenet model in Caffe on a Samsung Galaxy S5. Left panel: four branches in parallel, convolution layer, convolution + convolution, convolution + convolution, convolution + pooling. Right panel: two branches in parallel, convolution + convolution, convolution + convolution. The four branches are merged into two branches.

by the 1×1 "reducer". For layer inception_3a/3x3_reduce, we reduce the number of output feature maps from 96 to 48.

- *Merging*: A convolution branch with a smaller kernel size can be merged to a convolution branch with a larger kernel size. The method is similar to the merging of non-tensor layers. 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 1×1 convolution branch and increase the number of feature maps generated by the 3×3 convolution from 128 to 196.

3.3 Adapting DeepRebirth to overall model

The new generated layer (i.e., rebirth layer) is required to learn the new parameters using fine-tuning as discussed in Yosinski et al. (2014); Razavian et al. (2014). We use standard initialization methods to (e.g., Xavier Glorot & Bengio (2010) initialization) to initialize the parameters in the new layer while keeping the weights of other layers unchanged. In our optimization procedure, we set the learning rate of the new learning layers 10 times over those in other layers. The proposed optimization scheme is applied from the bottom layer to the top layer. It is also possible to learn multiple rebirth layers at the same time (we merge and fine-tune 3 sequential inception layers 4b-4d together for GoogLeNet) or merge layers in orders other than bottom-to-top.

4 Evaluation

4.1 GoogLeNet

To evaluate the performance of DeepRebirth, we performed a comprehensive evaluation using ifferent optimization approaches on top of 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-Merge" (see structure in appendix). After non-tensor layer optimization (streamline and branch merging), we further apply tucker decomposition approach (Kim et al. (2015)) to reduce the model size (i.e., the number of learned weights) by 50%, represented as "GoogLeNet-Merge-Tucker". In addition, we directly employ tucker decomposition method to compress original GoogLeNet. This

Table 2: GoogLeNet Accuracy on each layer after merging

Step	Merged 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	**88.43%**
Tucker Decomposition	N/A	**86.54%**

is indicated as "GoogLeNet-Tucker". Thus, we have 4 models to compare, namely GoogLeNet, GoogLeNet-Merge, GoogLeNet-Tucker and GoogLeNet-Merge-Tucker.

4.1.1 ACCURACY

Since one of our major goals is to propose a new acceleration approach which can speed up the model running time with satisfied accuracy (in constrast to the original model), we list the accuracy changes along with the optimization steps conducted on ImageNet ILSVRC-2012 validation dataset as indicated in Table 2. 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 (Bottou (2012)) 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 set 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 the proposed method has almost negligible impact on the model accuracy, and the accuracy even increases at certain step (e.g., step 5). This indicates that "the newborn" layers perfectly simulate the functionalities of previous non-tensor layers before optimization. By applying tucker decomposition method on the merged model to reduce the weights by half (GoogLeNet-Merge-Tucker), we observer 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 layer mergings.

4.1.2 SPEED-UP

To evaluate and compare the latency of different optimization approaches, we evaluate the 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 phone to the airplane mode and close all other apps.

As is demonstrated in Table 3, we observe that for the best case scenario, GoogLeNet-Merge is 3x faster than GoogLeNet and for the worst case scenario, GoogLeNet takes around 950 ms for a single forwarding while GoogLeNet-Merge takes only around 250 ms, which is almost 4x speed-up. This is because the original GoogLeNet model has too many small layers and this results in performance fluctuation. The same finding is also sharply observed in Kim et al. (2015) . 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.

Table 3: Breakdown of GoogLeNet forwarding time cost using different methods on each layer.

Device	GoogLeNet	GoogLeNet -Tucker	GoogLeNet -Merge	GoogLeNet -Merge-Tucker
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
Total	**651.4 ms**	**614.9 ms (1.06x)**	**210.6 ms (3.09x)**	**106.3 ms (6.13x)**

Table 4: Execution time using different methods (including SqueezeNet) on different mobile devices

Device	GoogLeNet	GoogLeNet -Tucker	GoogLeNet -Merge	GoogLeNet -Merge-Tucker	SqueezeNet
Moto E	1168.8 ms	897.9 ms	406.7 ms	213.3 ms	291.4 ms
Samsung Galaxy S5	651.4 ms	614.9 ms	210.6 ms	106.3 ms	136.3 ms
Samsung Galaxy S6	424.7 ms	342.5 ms	107.7 ms	65.34 ms	75.34 ms
Macbook Pro (CPU)	91.77 ms	78.22 ms	23.69 ms	15.18 ms	17.63 ms
Titan X	10.17 ms	10.74 ms	6.57 ms	7.68 ms	3.29 ms

Not limited to mobile platform of Samsung Galaxy S5, we also apply the speed-up schemes on other popular processors. These mobile devices include (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 4. The promising result indicates that the proposed method achieves 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 further applying tensor weights compression method. Finally, we compare the proposed approach with SqueezeNet (Iandola et al. (2016)) which is a state-of-the-art compressed CNN model. We are very excited to see that our optimization approach can obtain faster speed with higher accuracy compared to SqueezeNet(80% for Top-5)'s performance on all CPU platforms as listed in Table 4.

4.1.3 Energy, Storage and Runtime-Memory Cost

We measure the energy cost of each compared model using PowerTutor Android app (Zhang et al. (2010)) on Samsung Galaxy S5. The original GoogLeNet consumes almost 1 Joule per image while GoogLeNet-Merge 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-Merge 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.

Table 5: GoogLeNet Execution Storage vs. Engery vs. Runtime-Memory Cost

Model	Energy	Storage	Runtime Memory	Max Batch Size on Titan X
GoogLeNet	984 mJ	26.72 MB	33.2 MB	350
GoogLeNet-Tucker	902 mJ	14.38 MB	35.8 MB	323
GoogLeNet-Merge	**447 mJ (2.2x)**	23.77 MB	13.2 MB	**882 (2.52x)**
GoogLeNet-Merge-Tucker	**226 mJ (4.4x)**	11.99 MB	14.8 MB	**785 (2.24x)**
SqueezeNet	288 mJ	4.72 MB	36.5 MB	321

Table 6: AlexNet Result (Accuracy vs. Speed vs. Energy cost)

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

Another benefit of layer merging is run-time memory saving. The generated GoogLeNet-Merge 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 5, one Titan X GPU can run a batch size of 882 with the GoogLeNet-Merge 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 memory impact due to the increased number of layers.

4.2 AlexNet and ResNet

To further analyze the generality of proposed DeepRebirth acceleration framework, besides GoogLeNet, we also apply the proposed framework to other popular deep neural structures: AlexNet (Krizhevsky et al.) and ResNet (He et al. (2015)). 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 merging approach to re-generate new layers by merging the first two convolution layers followed by LRN layers. We illustrate the result in Table 6. This indicates that by applying merging 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 also 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 7. 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.

Table 7: ResNet (conv1-res2a) Result (Accuracy vs. Speed up).

Step	Merged 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)

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. (2011) proposed a fixed-point implementation with 8-bit integer activation to reduce the number of parameter used in the deep neural network while Gong et al. (2014) 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, Denton et al. (2014) applied the low-rank approximation approach to compress the neural networks with linear structures. Afterwards, hashing function was utilized by Chen et al. (2015) to reduce model sizes by randomly grouping connection weights. More recently, Han et al. (2016b) 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 Han et al. (2016a). In general, reducing the model size can help deployment of deep learning models, this, however, does not necessarily bring significant speed up for running deep learning models. Compared to these works, instead of reducing model size, DeepRebirth provides a generic framework to accelerate the running speed that can be applied for different deep learning architectures on different low-level hardware (CPU, GPU, etc).

In order to improve the network running efficiency, some scalable networks have been proposed by balancing the running speed and the accuracy. Rastegari et al. (2016) designed a binary deep learning network (called XNOR-Net) where both the network weight and the input can be binarized for memory and computational saving. However, this network design depressed the accuracy greatly. The top-5 accuracy obtained by this framework is reduced by more than 10% for ResNet-18 model along with 2x speed-up. Another popular newly designed small model -SqueezeNet Iandola et al. (2016) becomes 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. (2014) shows 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 a mobile device. In addition, our work fine-tunes a trained network by relearning the merged "rebirth" layers and does not require to train from scratch.

6 CONCLUSION

We have proposed DeepRebirth acceleration framework which can speed up the neural networks with satisfactory accuracy. Our method operates by re-generating new tensor layers from optimizing the non-tensor layers and their neighboring units. Moreover, as a generic method, DeepRebirth is 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 the significant speed-up on different processors, especially on mobile CPUs. This will greatly facilitate the deployment of deep learning models on mobile phones and make it possible to provide more smart and intelligent services in the new AI tide.

REFERENCES

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning

on heterogeneous systems, 2015. URL http://tensorflow.org/. Software available from tensorflow.org.

Sanjeev Arora, Aditya Bhaskara, Rong Ge, and Tengyu Ma. Provable bounds for learning some deep representations. *CoRR*, abs/1310.6343, 2013. URL http://arxiv.org/abs/1310.6343.

Lon Bottou. *Stochastic Gradient Tricks*, volume 7700, pp. 430445. Springer, January 2012. URL https://www.microsoft.com/en-us/research/publication/stochastic-gradient-tricks/.

Wenlin Chen, James T Wilson, Stephen Tyree, Kilian Q Weinberger, and Yixin Chen. Compressing neural networks with the hashing trick. *CoRR, abs/1504.04788*, 2015.

Emily L Denton, Wojciech Zaremba, Joan Bruna, Yann LeCun, and Rob Fergus. Exploiting linear structure within convolutional networks for efficient evaluation. In *Advances in Neural Information Processing Systems*, pp. 1269–1277, 2014.

Xavier Glorot and Yoshua 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.

Yunchao Gong, Liu Liu, Ming Yang, and Lubomir Bourdev. Compressing deep convolutional networks using vector quantization. *arXiv preprint arXiv:1412.6115*, 2014.

Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. Eie: Efficient inference engine on compressed deep neural network. *International Conference on Computer Architecture (ISCA)*, 2016a.

Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *International Conference on Learning Representations (ICLR)*, 2016b.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *arXiv preprint arXiv:1512.03385*, 2015.

Forrest N. Iandola, Matthew W. Moskewicz, Khalid Ashraf, Song Han, William J. Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <1mb model size. *arXiv:1602.07360*, 2016.

Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

Yuhang Wang Qinghao Hu Jiaxiang Wu, Cong Leng and Jian Cheng. Quantized convolutional neural networks for mobile devices. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

Yong-Deok Kim, Eunhyeok Park, Sungjoo Yoo, Taelim Choi, Lu Yang, and Dongjun Shin. Compression of deep convolutional neural networks for fast and low power mobile applications. *CoRR*, abs/1511.06530, 2015. URL http://arxiv.org/abs/1511.06530.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, pp. 2012.

Jinyu Li. Restructuring of deep neural network acoustic models with singular value decomposition. In *Interspeech*, January 2013. URL https://www.microsoft.com/en-us/research/publication/restructuring-of-deep-neural-network-acoustic-models-with-singular-value-decomposi

Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *ECCV*, 2016.

Ali Sharif Razavian, Hossein Azizpour, Josephine Sullivan, and Stefan Carlsson. CNN features off-the-shelf: an astounding baseline for recognition. *CoRR*, abs/1403.6382, 2014. URL http://arxiv.org/abs/1403.6382.

sh1r0, zif520, and strin. BWorld Robot Control Software. https://github.com/sh1r0/caffe-android-lib/issues/23, 2015. [Online; accessed 19-July-2016].

Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, and Martin A. Riedmiller. Striving for simplicity: The all convolutional net. *CoRR*, abs/1412.6806, 2014. URL http://arxiv.org/abs/1412.6806.

Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. *CoRR*, abs/1409.4842, 2014. URL http://arxiv.org/abs/1409.4842.

Vincent Vanhoucke, Andrew Senior, and Mark Z Mao. Improving the speed of neural networks on cpus. 2011.

Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. How transferable are features in deep neural networks? *CoRR*, abs/1411.1792, 2014. URL http://arxiv.org/abs/1411.1792.

Lide Zhang, Birjodh Tiwana, Zhiyun Qian, Zhaoguang Wang, Robert P. Dick, Zhuoqing Morley Mao, and Lei 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, pp. 105–114, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-905-3. doi: 10.1145/1878961.1878982. URL http://doi.acm.org/10.1145/1878961.1878982.

Appendices

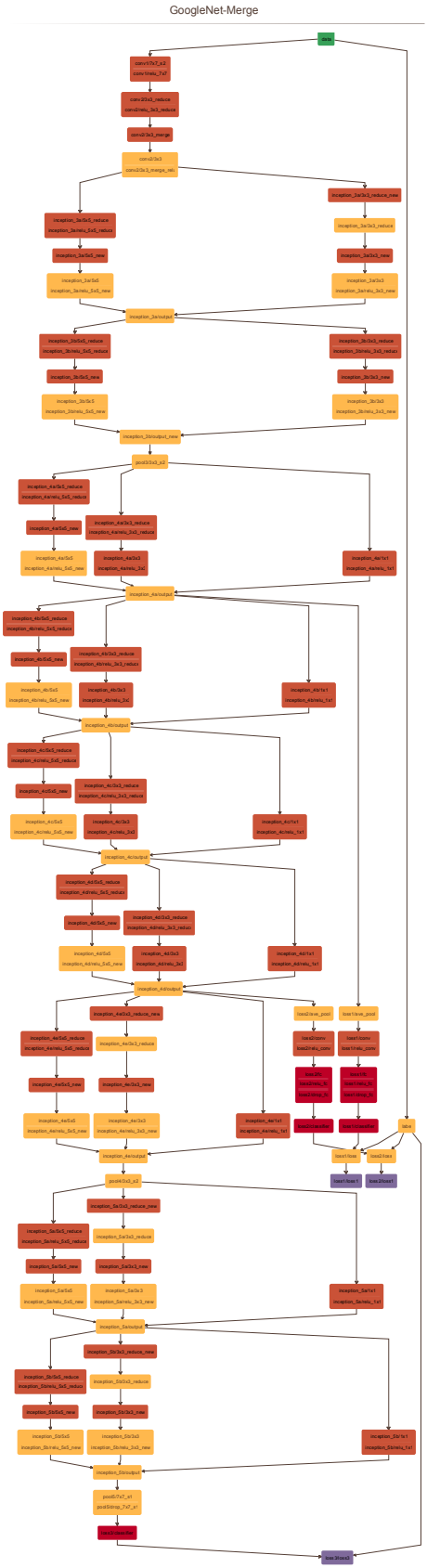


Figure 4: An illustration of GoogleNet-Merge's structure in details.