The Thirty-Second AAAI Conference
on Artificial Intelligence (AAAI-18)

DeepRebirth: Accelerating Deep Neural Network Execution on Mobile Devices

Dawei Li,[1] **Xiaolong Wang,**[1] **Deguang Kong**
[1]Samsung Research America, Mountain View, CA
{xiaolong.w, dawei.l}@samsung.com, {doogkong}@gmail.com

Abstract

Deploying deep neural networks on mobile devices is a challenging task. Current model compression methods such as matrix decomposition effectively reduce the deployed model size, but still cannot satisfy real-time processing requirement. This paper first discovers that the major obstacle is the excessive execution time of non-tensor layers such as pooling and normalization without tensor-like trainable parameters. 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. The proposed optimization operations significantly accelerate the model execution and also greatly reduce the run-time memory cost since the slimmed model architecture contains less hidden layers. 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 more than 3x speed-up and 2.5x run-time memory saving on GoogLeNet with only 0.4% drop on top-5 accuracy in ImageNet. Furthermore, by combining with other model compression techniques, DeepRebirth offers an average of $106.3ms$ inference time on the CPU of Samsung Galaxy S5 with 86.5% top-5 accuracy, 14% faster than SqueezeNet which only has a top-5 accuracy of 80.5%.

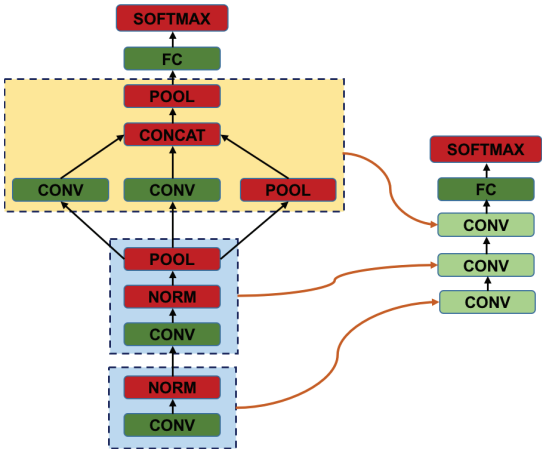


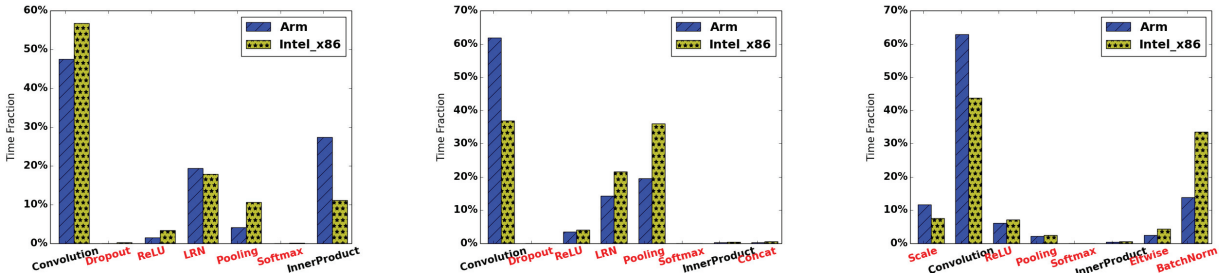
Figure 1: An illustration of proposed DeepRebirth model acceleration pipeline. DeepRebirth optimizes a trained deep learning model (left) to an accelerated "slim" model (right). Such optimization is achieved with two operations: *Streamline Slimming* which absorbs non-tensor layers (i.e., pooling and 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.

Introduction

Recent years have witnessed the breakthrough of deep learning techniques for many computer vision tasks, e.g., image classification (Krizhevsky, Sutskever, and Hinton 2012; Szegedy et al. 2014), object detection and tracking (Ren et al. 2015; Yu et al. 2016; Du et al. 2017), video understanding (Donahue et al. 2015; Li et al. 2017), content generation (Goodfellow et al. 2014; Zhang, Song, and Qi 2017), disease diagnosis (Shen, Wu, and Suk ; Zhang et al. 2017) and privacy image analytics (Tran, Kong, and Liu 2016). 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 devices, which is a highly intriguing feature. Here we define the execution efficiency as the model inference speed, the energy cost and the run-time memory consumption. In reality, it takes more than 651ms to recognize an image using GoogleNet on Samsung S5 (Table 4) with 33.2 MB run-time memory and 984mJ energy costs (Table 5). The effective solution is expected to provide minimum accuracy loss by leveraging widely used deep neural network architectures (such as GoogLeNet and ResNet) with support of deep model acceleration on different types of layers.

Excessive execution time in Non-tensor layers



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

Figure 2: 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.

Table 1: Compare DeepRebirth with Existing Acceleration Methods on CPU of Samsung Galaxy S5 Mobile Device.

	Parameter Compression[1] (Kim et al. 2015)	SqueezeNet (Iandola et al. 2016)	MobileNet[2] (Howard et al. 2017)	DeepRebirth (ours)
Accuracy	85.7%	80.3%	83.7%	**86.5%**
Execution Time	558.3 ms	122.7 ms	109.5 ms	**106.3 ms**
Energy Cost	902 mJ	288 mJ	243 mJ	**226 mJ**
Memory Cost	35.8 MB	36.5 MB	22.6 MB	**14.8 MB**

In this work, 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, fully connected layers and convolutional layers are tensor layers since they contain 2-d and 4-d tensor-type weight parameters, respectively. Whereas pooling layer and LRN layer are both non-tensor layers because they do not contain any high-order tensor-type weight parameters. 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) *streamline slimming*; (b) *branch slimming*. In streamline slimming, the new tensor layers are re-generated by substituting the original non-tensor layers and their neighborhood 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 their parallel tensor branches horizontally (shown in Figure 4, such as the inception module in GoogLeNet (Szegedy et al. 2014)). Overall, reducing the execution time on non-tensor layers can greatly reduce the model inference time given the fact that tensor-layer has been able to get optimized to the minimum as suggested by (Han, Mao, and Dally 2016; Kim et al. 2015). Finally, we can combine both non-tensor and tensor layer optimization and further reduce the latency as well as the model size.

[1]The accuracy reported here is based on an compression rate of roughly 50%. In the original paper, the authors reported a small 0.24% accuracy loss with compressed rate 31.9%. For our model at the same 31.9% compression rate, we also only have a small 0.31% accuracy loss.

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 (§).* Compared to existing works, (Han, Mao, and Dally 2016; Kim et al. 2015; Yu et al. 2017) are designed to reduce the model size by approximating the tensor-type layers using methods like low rank approximation and quantization. For non-tensor layers (e.g., normalization and pooling layers) which are generally designed and used for speeding up the network training and obtaining better generalization performance, optimization for faster execution has *not* been discussed so far. In this paper, we emphasize and validate experimentally that the proposed method is orthogonal to compression techniques on tensor-type layers. Consequently, our method can be combined with these techniques for further acceleration.

To summarize, we make 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.

• DeepRebirth performs both streamline slimming and branch slimming by merging non-tensor layers with its neighboring 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 for processing a single image with only 0.4% drop on Top-5 accuracy on ImageNet without any weights compression method. DeepRebirth achieves around 106.3 ms for processing a single image with Top-5 accuracy up to 86.5%.

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%
Average	49.89%	33.17%	35.22%

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, (Krizhevsky, Sutskever, and Hinton 2012)), GoogLeNet(Figure 2b, (Szegedy et al. 2014)) and ResNet(Figure 2c, (He et al. 2015)).

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 (Jia et al. 2014), 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 have as high affect on 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.*

DeepRebirth

To reduce the inference time on non-tensor layers, we propose DeepRebirth to accelerate the model execution at both streamline 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 redundant tensor layers. This process is similar to viewing the Inception model as a logical culmination as suggested by (Arora et al. 2013). DeepRebirth covers two major components: (a) streamline slimming; (b) branch slimming; which will be illustrated in the following.

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 black box 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 $poo2/3\times3\_s2$ (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 new 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^i_{CNN}. More mathematically,

$$\begin{aligned} X^i \xrightarrow{f_{\text{conv}}} Y^i_{\text{cv}} \xrightarrow{f_{\text{bn}}} Y^i_{\text{cv+bn}} \xrightarrow{f_{\text{sl}}} Y^i_{\text{cv+bn+sl}} \\ \xrightarrow{f_{\text{pooling}}} Y^i_{\text{cv+bn+sl+pl}} \xrightarrow{\cdot} ..... := Y^i_{\text{CNN}} \end{aligned} \quad (2)$$

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

[2]We use the Caffe implementation of 0.5 MobileNet-224 which has similar speed with our model.

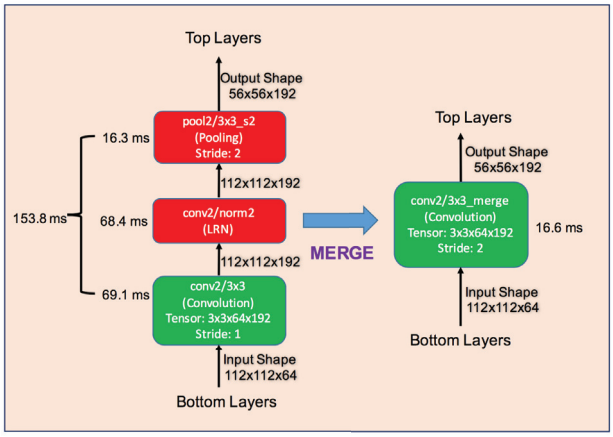


Figure 3: Streamline Slimming: The GoogLeNet example and the running time is measured using bvlc_googlenet model in Caffe on a Samsung Galaxy S5. Left panel: convolution (in green), LRN (in red), pooling (in red). Right Panel: single convolution layer. The three layers in the left panel are merged and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

pipeline such as LRN layer f_{LRN}. The layer parameters are represented by:

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

where $\mathbf{W}_{\text{conv}}$, $\mathbf{B}_{\text{conv}}$ represent convolution layer weight and bias matrix respectively, μ, $\boldsymbol{\sigma^2}$ and m are mean, variance, and sample number in mini-batch of normalization layer f_{bn}, γ and β are scaling weight and bias in scaling layer f_{sl} respectively, p represents the nearby p regions in pooling layer f_{pooling}, and κ, ρ and α 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

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

such that it can get the same feature map value Y^i_{CNN} 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 (Ioffe and Szegedy 2015) 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.,*

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

Clearly, the optimal solution is given by:

$$(\tilde{\mathbf{W}}^*, \tilde{\mathbf{B}}^*) = argmin_{\mathbf{W},\mathbf{B}} \sum_i \|Y^i_{\text{CNN}} - \tilde{f}(\mathbf{W}, \mathbf{B}; X^i)\|^2_F. \tag{5}$$

More formally, we have lemma 1.

Lemma 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 2.

Lemma 2. *Let W_j, B_j, μ_j, σ^2_j, γ_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^i_{CNN} is obtained after the three layers of f_{conv}, f_{bn}, f_{sl} in the sequence order, i.e., $Y^i_{CNN} := Y^i_{cv+bn+sl}$, we have closed form solution for the parameters in the new convolution layer:*

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

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

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

On the other hand, according to the definition of batch normalization (Ioffe and Szegedy 2015) and scaling, we have

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

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

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

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. □

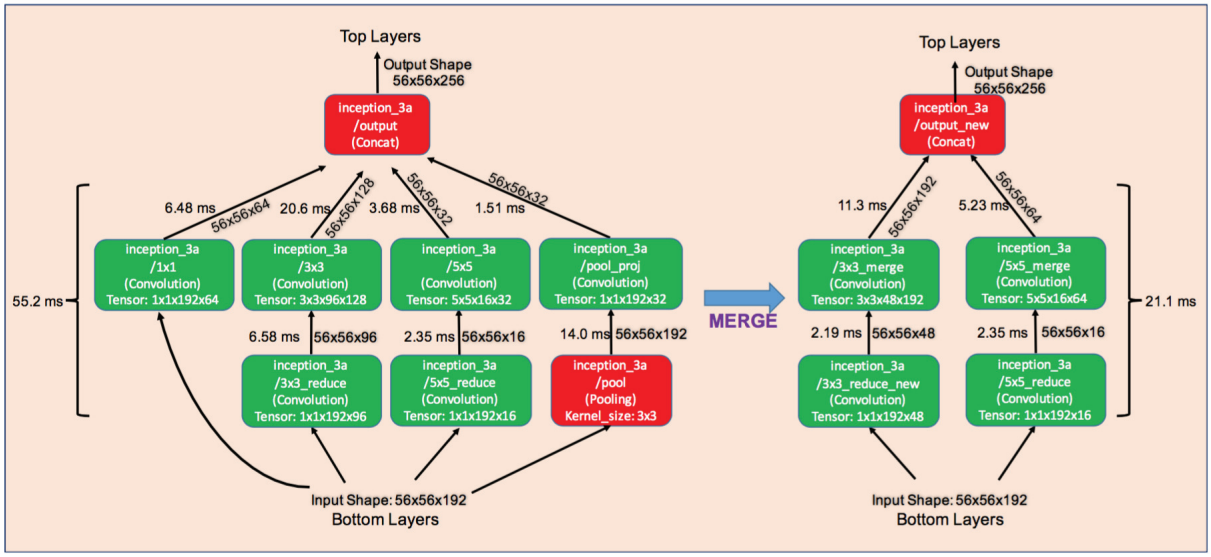


Figure 4: Branch Slimming: The GoogLeNet example and the running time is measured using bvlc_googlenet model in Caffe on a Samsung Galaxy S5. Left panel: four branches in parallel, convolution layer, convolution + convolution, convolution + convolution, convolution + pooling. Right panel: two branches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

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 (1×1, 3×3 and 5×5) and one 3×3 pooling branch (Szegedy et al. 2014). The output feature maps of each branch are concatenated as input of the following 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 1×1 (e.g., the 3×3 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 Figure 4, 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.

• *Branch Reducing*: Current deep neural networks usually include convolution branches with 1×1 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 1×1 "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 1×1 convolution branch and increase the number of feature maps generated by the 3×3 convolution from 128 to 196.

Slimming over tensor-branches should be careful. In our work, we demonstrate that in GoogLeNet architecture, the tensor-branch with smaller convolutional kernel can be slimmed without affecting the performance, and thus we are able to reduce 4 branches (3 tensor branches and 1 non-tensor branch) into 2 tensor branches. However, when the original architecture only has 2 tensor branches (e.g., in ResNet), slimming any branch will affect the performance.

Branch convolutional layer slimming analysis Let Y_{L} and Y_{R} be the feature map learned using convolution layers respectively given model parameter weight and bias, *i.e.,*

$$\begin{cases} Y_{\mathrm{L}}^i = \mathbf{W}_{\mathrm{L}} * X^i + \mathbf{B}_{\mathrm{L}}; & \triangleright \text{left branch} \\ Y_{\mathrm{R}}^i = \mathbf{W}_{\mathrm{R}} * X^i + \mathbf{B}_{\mathrm{R}}; & \triangleright \text{right branch} \end{cases} \tag{10}$$

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

$$Y_{LR}^i = [Y_{\text{left}}^i; Y_{\text{right}}^i], \quad Y_{LR}^i = \mathbf{W}_{\text{LR}} * X^i + \mathbf{B}_{\text{LR}}, \qquad (11)$$

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

If $\mathbf{W}_{\text{L}}$ and $\mathbf{W}_{\text{R}}$ have the same kernel size, we can get

$$\mathbf{W}_{\text{LR}} = [\mathbf{W}_{\text{left}}; \mathbf{W}_{\text{right}}], \quad \mathbf{B}_{\text{LR}} = [\mathbf{B}_{\text{left}}; \mathbf{B}_{\text{right}}]. \qquad (12)$$

by substituting Eq.(10) into Eq.(11). Otherwise, we need to adjust Y_{L} and Y_{R} to the same size and learn the model parameters by minimizing:

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

Adapting DeepRebirth to Overall Pipeline

DeepRebirth can be easily applied to a pre-trained deep learning model as modern deep model architectures are well-structured with repeating substructures such as the inception module in GoogLeNet and the residual module in ResNet. Generally, there are three golden rules we need to follow: (1) identify the repeating substructures, (2) determine the input dimension and output dimension for each substructure, and (3) apply either streamline slimming or branch slimming based on the substructure type.

To reconcile the new learned layer with other parts of model, one further step is to fine-tuning the model parameters[3], as suggested in (Yosinski et al. 2014; Razavian et al. 2014). In DeepRebirth, we leverage Xavier (Glorot and Bengio 2010) 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. Generally, 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. We will explore this discussion in our future work.

Evaluation

To evaluate the performance of DeepRebirth, we performed the comprehensive evaluation on top of GoogLeNet, AlexNet and ResNet. Our implementation is based on Caffe (Jia et al. 2014) deep learning framework, and we compile it using Android NDK for mobile evaluation. OpenBLAS (Xianyi, Qian, and Chothia 2014) is used for efficient linear algebra calculations.

[3]One exception is the BatchNorm layer which can be directly merged to a preceding convolutional layer using Eq.(9)

Table 3: GoogLeNet Accuracy on Slimming Each Layer

Step	Slim Layer(s)	Top-1 Accuracy	Top-5 Accuracy
0	N/A	68.72%	88.89%
1	conv1	68.65%	88.73%
2	conv2	68.66%	88.82%
3	inception_3a	68.35%	88.50%
4	inception_3b	68.21%	88.27%
5	inception_4a	68.34%	88.60%
6	inception_4b-4d	68.31%	88.61%
7	inception_4e	68.26%	88.43%
8	inception_5a	68.22%	88.41%
9	inception_5b	68.03%	**88.43%**
Tucker Decomposition	**ALL**	66.71%	**86.54%**

GoogLeNet

We use Caffe's GoogLeNet implementation (i.e., bvlc_googlenet) with its pre-trained 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 (Kim et al. 2015) 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 (Iandola et al. 2016), 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 regenerated 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 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 functionality of previous non-tensor layers before optimization. By applying 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 differ-

Table 4: Layer breakdown of GoogLeNet forwarding time cost

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

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

Device	GoogLeNet	GoogLeNet -Tucker	GoogLeNet -Slim	GoogLeNet -Slim-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**

ent optimization approaches, we evaluate the layer-wise running speed on a Samsung Galaxy S5 smartphone with Caffe. Each test run includes 50 subtests with a random input and we 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 (Kim et al. 2015), 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 S6: a high-end mobile ARM CPU, (3) Macbook Pro: an Intel x86 CPU, and (4) Titan X: a powerful server GPU. We demonstrate the experimental results in Table 5 and observe 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 SqueezeNet (Iandola et al. 2016), 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 on Samsung Galaxy S5 (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.

Table 6: Storage, Energy and Runtime-Memory Comparison

Model	Energy	Storage	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-Slim	**447 mJ (2.2x)**	23.77 MB	13.2 MB	**882 (2.52x)**
GoogLeNet-Slim-Tucker	**226 mJ (4.4x)**	11.99 MB	14.8 MB	**785 (2.24x)**
SqueezeNet	288 mJ	4.72 MB	36.5 MB	321

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 memory impact due to the increased number of layers.

AlexNet and ResNet

We apply the proposed framework to other popular deep neural structures: AlexNet (Krizhevsky, Sutskever, and Hinton 2012) 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 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 more advanced 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

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 $\rightarrow$ 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)

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)

to 104 ms. Moreover, the run-time memory cost is reduced by 2.21x. The accuracy is only slightly reduced. Meanwhile, since batch normalization layers can be directly merged to their preceding convolutional layers using Eq.(9), additional 30%-45% speed-up can be achieved without accuracy loss as indicated by Figure 2c.

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 reducing 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. (Vanhoucke, Senior, and Mao 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, Denten *et al.* (Denton et al. 2014) applied the low-rank approximation approach to compress the neural networks with linear structures. Afterwards, hashing functions, which have been widely adopted to improve efficiency of traditional computer vision tasks (Wang, Kumar, and Chang 2010; Du, Abd-Almageed, and Doermann 2013), were utilized to reduce model sizes by randomly grouping connection weights (Chen et al. 2015). More recently, Han et al.(Han, Mao, and Dally 2016) 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. 2016).

Recently, SqueezeNet (Iandola et al. 2016) has became 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. (Springenberg et al. 2014) 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 re-learning the merged layers and does not require to train from scratch. Essentially, DeepRebirth can be considered as a special form of distillation (Hinton, Vinyals, and Dean 2015) that transfers the knowledge from the cumbersome substructure of multiple layers to the new accelerated substructure.

Conclusion and Future Work

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 on different deep learning architectures, we obtain significant speed-up on different processors (including mobile processors), which will readily facilitate the deployment of deep learning models on mobile devices in the new AI tide.

In future work, we plan to integrate DeepRebirth with other state-of-the-art tensor layer compression methods and also extend our evaluation to heterogeneous mobile processors such as mobile GPUs, DSPs. We envision that understanding the characteristics of these different chips can help us design better algorithms and further improve the model execution efficiency.

Acknowledgements

We thank all the anonymous reviewers for their insightful comments and valuable suggestions.

References

Arora, S.; Bhaskara, A.; Ge, R.; and Ma, T. 2013. Provable bounds for learning some deep representations. *CoRR* abs/1310.6343.

Bottou, L. 2012. *Stochastic Gradient Tricks*, volume 7700. Springer. 430445.

Chen, W.; Wilson, J. T.; Tyree, S.; Weinberger, K. Q.; and Chen, Y. 2015. Compressing neural networks with the hashing trick. *CoRR, abs/1504.04788*.

Denton, E. L.; Zaremba, W.; Bruna, J.; LeCun, Y.; and Fergus, R. 2014. Exploiting linear structure within convolutional networks for efficient evaluation. In *Advances in Neural Information Processing Systems*, 1269–1277.

Donahue, J.; Anne Hendricks, L.; Guadarrama, S.; Rohrbach, M.; Venugopalan, S.; Saenko, K.; and Darrell, T. 2015. Long-term recurrent convolutional networks for visual recognition and description. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2625–2634.

Du, X.; Abd-Almageed, W.; and Doermann, D. S. 2013. Large-scale signature matching using multi-stage hashing. In *2013 12th International Conference on Document Analysis and Recognition, Washington, DC, USA, August 25-28, 2013*, 976–980.

Du, X.; El-Khamy, M.; Lee, J.; and Davis, L. S. 2017. Fused DNN: A deep neural network fusion approach to fast and robust pedestrian detection. In *2017 IEEE Winter Conference on Applications of Computer Vision, WACV 2017, Santa Rosa, CA, USA, March 24-31, 2017*, 953–961.

Glorot, X., and Bengio, Y. 2010. 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*.

Gong, Y.; Liu, L.; Yang, M.; and Bourdev, L. 2014. Compressing deep convolutional networks using vector quantization. *arXiv preprint arXiv:1412.6115*.

Goodfellow, I.; Pouget-Abadie, J.; Mirza, M.; Xu, B.; Warde-Farley, D.; Ozair, S.; Courville, A.; and Bengio, Y. 2014. Generative adversarial nets. In *Advances in neural information processing systems*, 2672–2680.

Han, S.; Liu, X.; Mao, H.; Pu, J.; Pedram, A.; Horowitz, M. A.; and Dally, W. J. 2016. Eie: Efficient inference engine on compressed deep neural network. *International Conference on Computer Architecture (ISCA)*.

Han, S.; Mao, H.; and Dally, W. J. 2016. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *International Conference on Learning Representations (ICLR)*.

He, K.; Zhang, X.; Ren, S.; and Sun, J. 2015. Deep residual learning for image recognition. *arXiv preprint arXiv:1512.03385*.

Hinton, G.; Vinyals, O.; and Dean, J. 2015. Distilling the Knowledge in a Neural Network. *ArXiv e-prints*.

Howard, A. G.; Zhu, M.; Chen, B.; Kalenichenko, D.; Wang, W.; Weyand, T.; Andreetto, M.; and Adam, H. 2017. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *CoRR* abs/1704.04861.

Iandola, F. N.; Moskewicz, M. W.; Ashraf, K.; Han, S.; Dally, W. J.; and Keutzer, K. 2016. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <1mb model size. *arXiv:1602.07360*.

Ioffe, S., and Szegedy, C. 2015. 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*, 448–456.

Jia, Y.; Shelhamer, E.; Donahue, J.; Karayev, S.; Long, J.; Girshick, R.; Guadarrama, S.; and Darrell, T. 2014. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*.

Kim, Y.; Park, E.; Yoo, S.; Choi, T.; Yang, L.; and Shin, D. 2015. Compression of deep convolutional neural networks for fast and low power mobile applications. *CoRR* abs/1511.06530.

Krizhevsky, A.; Sutskever, I.; and Hinton, G. E. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, 1097–1105.

Li, W.; Wen, L.; Chang, M.-C.; Nam Lim, S.; and Lyu, S. 2017. Adaptive rnn tree for large-scale human action recognition. In *The IEEE International Conference on Computer Vision (ICCV)*.

Razavian, A. S.; Azizpour, H.; Sullivan, J.; and Carlsson, S. 2014. CNN features off-the-shelf: an astounding baseline for recognition. *CoRR* abs/1403.6382.

Ren, S.; He, K.; Girshick, R. B.; and Sun, J. 2015. Faster R-CNN: towards real-time object detection with region proposal networks. *CoRR* abs/1506.01497.

Shen, D.; Wu, G.; and Suk, H.-I. Deep learning in medical image analysis. *Annual Review of Biomedical Engineering* (0).

Springenberg, J. T.; Dosovitskiy, A.; Brox, T.; and Riedmiller, M. A. 2014. Striving for simplicity: The all convolutional net. *CoRR* abs/1412.6806.

Szegedy, C.; Liu, W.; Jia, Y.; Sermanet, P.; Reed, S. E.; Anguelov, D.; Erhan, D.; Vanhoucke, V.; and Rabinovich, A. 2014. Going deeper with convolutions. *CoRR* abs/1409.4842.

Tran, L.; Kong, D.; and Liu, J. 2016. Privacy-cnh: A framework to detect photo privacy with convolutional neural network using hierarchical features. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA.*, 1317–1323.

Vanhoucke, V.; Senior, A.; and Mao, M. Z. 2011. Improving the speed of neural networks on cpus.

Wang, J.; Kumar, S.; and Chang, S.-F. 2010. Semi-supervised hashing for scalable image retrieval. In *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on*, 3424–3431. IEEE.

Xianyi, Z.; Qian, W.; and Chothia, Z. 2014. Openblas. *URL: http://xianyi. github. io/OpenBLAS*.

Yosinski, J.; Clune, J.; Bengio, Y.; and Lipson, H. 2014. How transferable are features in deep neural networks? *CoRR* abs/1411.1792.

Yu, F.; Li, W.; Li, Q.; Liu, Y.; Shi, X.; and Yan, J. 2016. POI: multiple object tracking with high performance detection and appearance feature. In *ECCV Workshops*.

Yu, X.; Liu, T.; Wang, X.; and Tao, D. 2017. On compressing deep models by low rank and sparse decomposition. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Zhang, J.; Li, Q.; Caselli, R. J.; Ye, J.; and Wang, Y. 2017. Multi-task dictionary learning based convolutional neural network for computer aided diagnosis with longitudinal images. *CoRR* abs/1709.00042.

Zhang, Z.; Song, Y.; and Qi, H. 2017. Age progression/regression by conditional adversarial autoencoder. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.