Master's Thesis
Optimizing Neural Networks for Mobile Devices

Radboud University, Nijmegen

Erdi Çallı

July 11, 2017

Abstract

Recently, convolutional neural networks became the state of the art method in image processing. However, there exists a gap between their potential and real life applications. This gap is caused by the computational requirements of convolutional neural networks. State of the art convolutional neural networks require heavy computations. We investigate methods to reduce the computational requirements and preserve the accuracy. Then we combine these methods to create a new model that is comparable with state of the art. We benchmark our model on mobile devices and compare its performance with others.

Contents

Chapter 1

Introduction

The state of the art in image processing has changed when graphics processing units (GPU) were used to train neural networks. GPUs contain many cores, they have very large data bandwidth and they are optimized for efficient matrix operations. In 2012, [KSH12] used two GPUs to train an 8 layer convolutional neural network (CNN). With this model, they won the ImageNet Large Scale Visual Recognition Competition (ILSVRC) classification task ([DBS+12]). Their model has improved the previous (top-5) classification accuracy record from $\sim 74\%$ to $\sim 84\%$. This caused a big trend shift in computer vision.

As the years' pass, GPUs got more and more powerful. In 2012, [KSH12] used GPUs that had 3 GB memory each. Today there are GPUs with up to 16 GB memory. The number of floating point operations per second (FLOPs) has also increased from 2.5 tera FLOPs (TFLOPs) to 12 TFLOPs. This gradual but steep change has allowed the use of more layers and more parameters. For example, [SZ14] introduced a model called VGGNet. Their model used up to 19 layers and shown that adding more layers increases the accuracy. [HZRS15] introduced a new method called residual connections, that allowed the use of up to 200 layers. Building up on such models, in 2016 ILSVRC winning (top-5) classification accuracy has increased to $\sim 97\%$.

In contrast, [SLJ+14] have shown that incorporating layers to compose blocks (i.e. inception blocks) works better than stacking layers. Their proposal has also been supported by [CPC16]. [CPC16] has shown the relation between number of parameters of a model and its top-1 classification accuracy in ILSVRC dataset. According to their report, 48 layer Inception-v3 ([SVI+16]) provides better top-1 classification accuracy than 152 layer

ResNet ([HZRS15]). They also show that Inception-v3 requires fewer number of floating point operations to compute results. Their results reveal that, providing more layers and parameters does not necessarily yield better results.

ILSVRC is one of the most famous competitions in image processing. Every year, the winners of this competition are driving the research on the field. However, this competition is not considering the computational cost of solutions. The computational cost is an important factor to express the cost of real life applications of a model. For example, the 2016 winner of ILSVRC, used an ensemble of large models[1]. Such an ensemble is very expensive to use in real life because of its high computational cost. But because the cost is hidden, these results are creating an unreal expectation in public. It seems as if these methods are applicable without a cost. In this thesis, we want to come up with a state of the art solution that could easily be applicable in real life. To define *applicable in real life*, we benchmark our solution on mobile devices. These devices are affordable and they have great availability, we believe that it is a proper platform for bridging the gap between expectations of the public and reality. In this thesis, we will answer,

> How can we reduce the computational cost of inference in convolutional neural networks?

First, we will briefly describe neural networks and some underlying concepts. We will mention the computational cost of necessary operations. Then, we will provide known solutions to reduce these complexities. In chapter two we will explain the experiments we ran and describe a convolutional neural network designed to work on mobile devices. In chapter three, we will present the results of our experiments.

1.1 Notations

We will be dealing tensors of various shapes. Therefore we will be defining a notation that will help us through the process. We will define sets of semantically similar tensors using capital Latin letters. For example, we will use W to describe the indexed set or an array of weights in a network. To represent an element of this set we will use superscript variables, such

[1]`http://image-net.org/challenges/LSVRC/2016/results#team`

as $w^{(k)}$. Since these sets represent a semantic group of variables that may have different properties, such as shape, or dimensions, or type, it would be misleading to represent them using a tensor or a matrix. Having such a definition, we will not be separating scalars, vectors, matrices or tensors using capitals or bolds. However, we will be defining these variables whenever necessary. We will use the $w^{(k)} \in \mathbb{R}^{5\times5}$ notation to describe a matrix with 5 rows and 5 columns with real numbers as values. To describe the coordinates of a variable, we will use subscript variables. We use $w^{(k)}_{i,j}$ to represent the ith column and jth row of this matrix. We use commas or parentheses to group these variables or dimensions semantically. If we need to operate (such as addition or multiplication) on the subscript variables, we will explicitly use $*$ as $w^{(k)}_{i*2,j*2}$ to represent multiplications so that we will not cause any confusion.

1.2 Neural Networks

In this section, we will describe neural networks briefly, provide some terminology and give some examples.

Neural networks are *weighted graphs.* They consist of an ordered set of *layers*, where every layer is a set of *nodes.* The first layer of the neural network is called the *input layer*, and the last one is called the *output layer.* The layers in between are called *hidden layers.* Layers are a semantic group of nodes. Nodes belonging to one layer are connected to the nodes in the following and/or the previous layers. These connections are weighted edges, and they are referred to as *weights.*

Given an input, neural network nodes have *outputs*, which are real numbers. The output of a node is calculated by applying a function (ψ) to the outputs of the nodes belonging to previous layers. Preceding that, the output of the input layer ($o^{(0)}$) is equal to the input data (see Eq. 1.1). By calculating the layer outputs consecutively we calculate the output of the output layer. This process is called *inference.* We use the following notations to denote the concepts that we just explained.

L:	the number of layers in a neural network
$l^{(k)}$:	layer k
$m^{(k)}$:	the number of nodes in $l^{(k)}$
$l_i^{(k)}$:	node i in $l^{(k)}$
$o^{(k)}$:	the output vector representing the outputs of nodes in $l^{(k)}$
$o_i^{(k)}$:	the output of $l_i^{(k)}$
$w^{(k)}$:	weight matrix connecting nodes in $l^{(k-1)}$ to nodes in $l^{(k)}$
$w_{i,j}^{(k)}$:	the weight connecting nodes $l_i^{(k-1)}$ and $l_j^{(k)}$
$b^{(k)}$:	the bias vector for $l^{(k)}$
$\psi_{(k)}$:	function to determine $o^{(k)}$ given $o^{(k-1)}$
σ:	activation function
X:	all inputs of the dataset as
Y:	all provided outputs of the dataset
$\hat{Y}$:	approximations of all outputs given all inputs
x_n:	nth input data
y_n:	nth output data
$\hat{y}_n$:	approximation of y_n given x_n

Therefore, the structure of a neural network is determined by the number of layers and the functions that determine the outputs of layers.

$$o^{(k)} = \begin{cases} \psi_{(k)}(o^{(k-1)}), & \text{if } k \geq 1 \\ x_n, & k = 0 \end{cases} \tag{1.1}$$

1.2.1 Fully Connected Layers

As the name suggests, for two consecutive layers to be *fully connected*, all nodes in the previous layer must be connected to all nodes in the following layer.

Let us assume two consecutive layers, $l^{(k-1)} \in \mathbb{R}^{m^{(k-1)} \times 1}$ and $l^{(k)} \in \mathbb{R}^{m^{(k)} \times 1}$. For these layers to be fully connected, the weight matrix connecting them would be defined as $w^{(k)} \in \mathbb{R}^{m^{(k-1)} \times m^{(k)}}$. Most fully connected layers also include a bias term ($b^{(k)} \in \mathbb{R}^{m^{(k)}}$). The output of a fully connected layer, $o^{(k)}$, would simply be calculated using layer function $\psi^{(FC)}$ as

$$o^{(k)} = \psi^{(FC)}_{(k)}(o^{(k-1)}) = (o^{(k-1)})^T w^{(k)} + b^{(k)}$$

The computational complexity of $\psi^{(FC)}_{(k)}$ is

$$\mathcal{O}(\psi^{(FC)}_{(k)}) = \mathcal{O}(m^{(k-1)} m^{(k)})$$

1.2.2 Activation Function and Nonlinearity

By stacking fully connected layers, we can increase the depth of a neural network. By doing so we may be able to increase approximation quality of the neural network. However, the $\psi^{(FC)}$ we have defined is a linear function. Therefore if we stack multiple fully connected layers using the current $\psi^{(FC)}$, we would end up with a linear model.

To achieve non-linearity, we apply *activation functions* to the results of ψ. There are many activation functions (such as *tanh* or *sigmoid*) but one very commonly used activation function is ReLU [NH10]. ReLU is defined as

$$\text{ReLU}(x) = \begin{cases} x, & \text{if } x \geq 0 \\ 0 & \text{otherwise} \end{cases} \tag{1.2}$$

As [GBB11] explained, ReLU leads to sparsity. As a result, given an input, only a subset of nodes are non-zero (active) and every possible subset results with a linear function. This linearity allows a better flow of gradients, leading to faster training. Also, the ReLU is not relying on any computation, so it is easier to compute compared to hyperbolic or exponential alternatives.

We will redefine the fully connected $\psi^{(FC)}$ with activation function (σ) as

$$\psi^{(FC)}_{(k)}(o^{(k)}) = \sigma((o^{(k)})^T w^{(k)} + b^{(k)})$$

The activation function does not strictly belong to the definition of fully connected layers. But for simplicity, we are going to include them in the layer functions (ψ).

$\psi^{(FC)}$ is one of the most basic building blocks of neural networks. By stacking building blocks in different types and configurations, we come up with different neural network structures. The outputs of every layer, starting from the input are calculated as

$$O = \{\psi_{(k)}(o^{(k-1)}) \mid k \in [1, \ldots, L]\}$$

1.2.3 Loss

To represent the quality of an approximation, we are going to use a loss (or cost) function. A good example to understanding loss would be the loss of a salesman. Assuming a customer who would pay at most \$10 for a given product, if the salesman sells this product for \$4, the salesman would face a loss of \$6 from his potential profit. Or if the salesman tries to sell this product for \$14, the customer will not purchase it and he will face a loss of \$10. In this example, the salesman would want to minimize the loss to earn as much as possible. There are two common properties of loss functions. First, the loss is never negative. Second, if we compare two approximations, the one with smaller loss is better at approximating the data.

Root Mean Square Error

A commonly used loss function is root mean square error (RMSE). Given an approximation ($\hat{y}_n \in \mathbb{R}^N$) and the expected output ($y_n \in \mathbb{R}^N$), RMSE can be calculated as

$$\mathcal{L} = \mathrm{RMSE}(\hat{y}_n, y_n) = \sqrt{\frac{\sum_{i=1}^{N} (\hat{y}_{n,i} - y_{n,i})^2}{N}}$$

Softmax Cross Entropy

Another commonly used loss function is softmax cross entropy (SCE). Softmax cross entropy is used for classification tasks where we are trying to find the class that our input belongs to. Softmax cross entropy first calculates the class probabilities given the input using the softmax function. It is defined as

$$p(i|\hat{y}_n) = \frac{e^{\hat{y}_{n,i}}}{\sum_{j=1}^{N} e^{\hat{y}_{n,j}}}$$

Then comparing it with the the expected output ($y_n \in \mathbb{R}^N$), SCE loss can be calculated as

$$\mathcal{L} = \mathrm{CE}(\hat{y}_n, y_n) = -\sum_{i=1}^{N} y_{n,i} log(p(i|\hat{y}_n))$$

SCE depends on the softmax to turn the node outputs into probabilities. Therefore, it makes sense to use it for classification tasks where the output data is representing a probability distribution. However, RMSE punishes the exact difference in outputs. Therefore, we can say that it is better for tasks like regression which represent exact values in output nodes. [GDN13] provides a comprehensive comparison of both methods.

1.2.4 Minimizing Loss

To provide better approximations, we will try to optimize the neural network parameters. One common way to optimize these parameters is to use stochastic gradient descent (SGD). SGD is an iterative learning method that starts with some initial (random) parameters. Given $\theta \in (W \cup B)$ to be a parameter that we want to optimize. The learning rule updating θ for a simple example would be

$$\theta = \theta - \eta \nabla_\theta \mathcal{L}(f(x), y)$$

where η is the learning rate, and $\nabla_\theta \mathcal{L}(f(x), y)$ is the partial derivative of the loss in terms of given parameter, θ. One iteration is completed when we update every parameter for given example(s). By performing many iterations, SGD aims to find a global minimum for the loss function, given data and initial parameters.

There are several other optimizers that work in different ways. We will be using Adam Optimizer ([KB14]), Momentum Optimizer ([Qia99]) and SGD.

1.2.5 Convolutional Layer

So far we have seen the key elements we can use to create and train fully connected neural networks. To be able to apply neural networks to image inputs, we can define convolutional layers using convolution operation. Please note that, in this section, we are assuming one or two dimensional convolutions with same padding.

Let us assume a 3 dimensional layer output $o^{(k-1)} \in \mathbb{R}^{\mathcal{H}_{k-1} \times \mathcal{W}_{k-1} \times m^{(k-1)}}$ where the dimensions $\mathcal{H}_{k-1}$ representing the length of the height dimension, $\mathcal{W}_{k-1}$ representing the length of the width dimension and $m^{(k-1)}$ representing number of nodes in that layer. Convolution operation first creates a sliding window of size $K \times K \times m^{(k-1)}$ that goes through height and width dimensions.

The contents of this sliding window would be patches ($p^{(k-1)}_{(I,J)} \in \mathbb{R}^{K \times K \times m^{(k-1)}}$) where $0 < I \leq \mathcal{W}_k$ and $0 < J \leq \mathcal{W}_k$. By multiplying the weight matrix $w^{(k)} \in \mathbb{R}^{K \times K \times m^{(k-1)} \times m^{(k)}}$ to the patch $p^{(k-1)}_{(I,J)}$ centered at (I, J), we create the set of output nodes for that point $o^{(k)}_{(I,J)} \in \mathbb{R}^{1 \times m^{(k)}}$. While calculating the patches, we also make use of a parameter called stride, $s_k \in \mathbb{N}^+$. s_k defines the number of vertical and horizontal steps to take between each patch.

In other words, strides (s_k) are used to define the width $(\mathcal{W}_k)$ and height $(\mathcal{H}_k)$ of the output in layer k as

$$\mathcal{W}_k = \left\lfloor \frac{\mathcal{W}_{k-1}}{s_k} \right\rfloor, \mathcal{H}_k = \left\lfloor \frac{\mathcal{H}_{k-1}}{s_k} \right\rfloor$$

Using this relationship between dimensions of outputs, we can define a convolutional layer as

$$\psi^{(Conv)}_{(k)} : \mathbb{R}^{\mathcal{H}_{k-1} \times \mathcal{W}_{k-1} \times m^{(k-1)}} \rightarrow \mathbb{R}^{\mathcal{H}_k \times \mathcal{W}_k \times m^{(k)}}$$

To perform this operation, we need to define and create the patch at location (I, J) as

$$p^{(k-1)}_{(I,J)} \in \mathbb{R}^{K \times K \times m^{(k-1)}}$$

$$p^{(k-1)}_{(I,J)} \subseteq o^{(k-1)}$$

The subindices (i, j) of patch $(p^{(k-1)}_{(I,J)})$ are a direct reference to the features at subindex (a, b) of the output. Using these indices, elements of this patch are defined as

$$p^{(k-1)}_{(I,J),(i,j)} \in \mathbb{R}^{m^{(k-1)}}, 0 < i \leq K, 0 < j \leq K$$

$$o^{(k-1)}_{a,b} \in \mathbb{R}^{m^{(k-1)}}, 0 < a \leq \mathcal{H}_{k-1}, 0 < b \leq \mathcal{W}_{k-1}$$

This direct reference is

$$p^{(k-1)}_{(I,J),(i,j)} = o^{(k-1)}_{a,b}$$

where the relationship between subindices of the output layer (a, b) and the patch $((I, J), (i, j))$ are defined dependent on the strides and the kernel size as

$$a = Is_k + (i - \lfloor K/2 \rfloor)$$

$$b = Js_k + (j - \lfloor K/2 \rfloor)$$

Having the definition for a patch $p^{(k-1)}_{(I,J)}$ and the indices related to it, we can define the output of next layer as

$$\psi^{(Conv)}_{(k)}(o^{(k-1)}) = o^{(k)} = \{o^{(k)}_{(I,J)} \mid \forall(I,J)(\exists p^{(k-1)}_{(I,J)})[o^{(k)}_{(I,J)} = \sigma(p^{(k-1)}_{(I,J)} w^{(k)} + b^{(k)}) \}$$

where the weight and the bias are defined as

$$w^{(k)} \in \mathbb{R}^{K \times K \times m^{(k-1)} \times m^{(k)}}$$

$$b^{(k)} \in \mathbb{R}^{m^{(k)}}$$

In other words, the output of layer is a set of vectors ($o^{(k)} = \{o^{(k)}_{(I,J)}\}$). For every pair of indices (I,J), there exists a patch $p^{(k-1)}_{(I,J)}$ defined by the outputs of the previous layer. We apply the weight, the bias and the activation function to these patches to calculate the set $o^{(k)}$. Given this description, we can define the complexity of this operation as

$$\mathcal{O}(\psi^{(Conv)}_{(k)}) = \mathcal{O}(\mathcal{W}_k \mathcal{H}_k K^2 m^{(k-1)} m^{(k)})$$

1.2.6 Pooling

Pooling is a way of reducing the dimensionality of a layer. Depending on the task, one may choose from different pooling methods. Similar to convolution operation, pooling methods also work with patches $p^{(k-1)}_{(I,J)} \in \mathbb{R}^{K \times K \times m^{(k-1)}}$ and strides s_{k-1}. But this time, instead of applying a weight, bias and activation function, they apply simpler functions. Here we will see two types of pooling layers.

Max Pooling

Max pooling takes the maximum value in a channel within the patch. Let's define the first subindex of a patch as if it is referring to a node as

$$p^{(k-1)}_{(I,J),i} \in \mathbb{R}^{K \times K}, 0 < i \leq m^{(k-1)}$$

Using this definition, max pooling can be defined as

$$\psi^{(maxpool)}_{(k)}(o^{(k-1)}) = o^{(k)} = \{o^{(k)}_{(I,J),i} \mid \forall((I,J),i)(\exists p^{(k-1)}_{(I,J),i})[o^{(k)}_{(I,J),i} = \max(p^{(k-1)}_{(I,J),i})]\}$$

In other words, for every index $(I,J), i$, there exists a $K \times K$ matrix. The value of the output at index $(I,J), i$ is defined as the maximum value of that matrix. Max pooling is mostly used after the first or second convolutional layer to reduce the dimensionality of the input in classification tasks.

Average Pooling

Average pooling averages the values within the patch per channel. The subindices of patch $p_{(I,J)}^{(k-1)}$ are defined as

$$p_{(I,J),i,a,b}^{(k-1)} \in \mathbb{R}$$

Using this definition, average pooling can be defined as

$$\psi_{(k)}^{(avgpool)}(o^{(k-1)}) = o^{(k)} = \{o_{(I,J),i}^{(k)} \mid \forall((I,J),i)(\exists p_{(I,J),i}^{(k-1)})[o_{(I,J),i}^{(k)} = \sum_{a=1}^{K}\sum_{b=1}^{K}\frac{p_{(I,J),i,a,b}^{(k-1)}}{K^2}]\}$$

In other words, for every index $(I,J),i$, there exists a $K \times K$ matrix. The value of the output at index $(I,J),i$ is defined as the average value of that matrix.

Global Pooling Methods

Global pooling methods take the output layer as one patch and reduce height and width dimensions to a single channel by applying the target function (max or average). Global average pooling is mostly used before the last fully connected layers in classification tasks.

1.2.7 Deconvolution

Introduced by [ZKTF10], deconvolution operation aims to increase the dimensionality of an input. To do that, it basically transposes the convolution operation. Deconvolution operation creates patches of $p_{(I,J)}^{(k-1)} \in \mathbb{R}^{1\times 1\times m^{(k-1)}}$ from the input, and applies a weight matrix of $w^{(k)} \in \mathbb{R}^{m^{(k-1)}\times K\times K\times m^{(k)}}$. In other words, it creates a $K \times K \times m^{(k)}$ output from every $1 \times 1 \times m^{(k-1)}$ patch and expands the height and width of the input.

So far we have seen some building blocks of convolutional neural networks. Now we move on to subjects that are related to reducing the complexity or increasing efficiency.

1.3 Efficient Operations

In this section, we are going to look at some ways to reduce the computational complexities of fully connected layers and convolutional layers.

1.3.1 Factorization

Factorization is approximating a weight matrix using smaller matrices. As explained by [ZZHS16], [DZB+14], [CS16], factorization has interesting uses with neural networks. Let us assume that we have a fully connected layer k. Using factorization, we can approximate $w^{(k)} \in \mathbb{R}^{m^{(k-1)} \times m^{(k)}}$ using two smaller matrices, $U_{w^{(k)}} \in \mathbb{R}^{m^{(k-1)} \times n}$ and $V_{w^{(k)}} \in \mathbb{R}^{n \times m^{(k)}}$. If we can find matrices such that $U_{w^{(k)}} V_{w^{(k)}} \approx w^{(k)}$, we can rewrite $\psi^{(FC)}_{(k)}$ as

$$\psi^{(FC)}_{(k)}(o) \approx \psi'^{(FC)}_{(k)}(o) = \sigma(o^T U_{w^{(k)}} V_{w^{(k)}} + b^{(k)})$$

Therefore, we can reduce the complexity of layer k by setting a sufficiently small n. As we have mentioned before, $\mathcal{O}(\psi^{(FC)}_{(k)}) = \mathcal{O}(m^{(k-1)} m^{(k)})$. When we approximate this operation, the complexity becomes

$$\mathcal{O}(\psi'^{(FC)}_{(k)}) = \mathcal{O}(n(m^{(k-1)} + m^{(k)}))$$

One thing that is similar between a convolutional layer and a fully connected layer is that both are performing matrix multiplications to calculate results. The only difference is, a convolutional layer is performing this matrix multiplication for every width and height dimension of the output layer. Therefore the same technique can be used with convolutional layers. If we apply factorization, the complexity of a convolutional layer would become

$$\mathcal{O}(\psi'^{(Conv)}_{(k)}) = \mathcal{O}(\mathcal{W}_k \mathcal{H}_k K^2 n(m^{(k-1)} + m^{(k)}))$$

When factorizing fully connected and convolutional layers, if there is a good enough approximation satisfying the following equation, we can reduce the complexity without affecting the results.

$$n < \frac{m^{(k-1)} m^{(k)}}{m^{(k-1)} + m^{(k)}} \tag{1.3}$$

The quality of the approximation will influence how this operation affects the accuracy.

SVD

Singular Value Decomposition (SVD) ([GR70]), is a factorization method that we can use to calculate the elements this approximation. SVD decomposes the weight matrix $w^{(k)} \in \mathbb{R}^{m^{(k-1)} \times m^{(k)}}$ into 3 parts as

$$w^{(k)} = USV^T$$

Where, $U \in \mathbb{R}^{m^{(k-1)} \times m^{(k-1)}}$ and $V \in \mathbb{R}^{m^{(k)} \times m^{(k)}}$. And $S \in \mathbb{R}^{m^{(k-1)} \times m^{(k)}}$ is a rectangular diagonal matrix. The diagonal values of S are called as the singular values of $w^{(k)}$. Selecting the n highest values from S and corresponding columns and rows from U and V, respectively, lets us create a *low-rank decomposition* of $w^{(k)}$ as

$$w^{(k)} \approx U' S' V'^T$$

where $U' \in \mathbb{R}^{m^{(k-1)} \times n}$, $V' \in \mathbb{R}^{n \times n}$, and $S' \in \mathbb{R}^{n \times m^{(k)}}$. By choosing a sufficiently small rank (n) satisfying Equation 1.3 and setting $U_{w^{(k)}} = U'S'$ and $V_{w^{(k)}} = V'^T$, we can approximate the weights, and reduce the complexity of a layer. [ZZHS16] applies this method to reduce the execution time of a network by 4 times and increase accuracy by 0.5%.

1.4 Convolution Operation Alternatives

As we have described above, the computational complexity of convolution operation is quite high. Here we will look at some alternative methods to define models with lower complexity.

1.4.1 Kernel Composing Convolutions

As [AP16] explains, a convolution operation with a weight matrix $w^{(k)} \in \mathbb{R}^{K \times K \times m^{(k-1)} \times m^{(k)}}$, could be composed using two convolution operations with kernels $w^{(k,1)} \in \mathbb{R}^{1 \times K \times m^{(k-1)} \times n}$ and $w^{(k,2)} \in \mathbb{R}^{K \times 1 \times n \times m^{(k)}}$ as

$$w^{(k)} \approx w^{(k,1)} w^{(k,2)}$$

Their technique, instead of factorizing learned weight matrices, aims to define kernels as if they were factorized and learn these values. They also aim to increase non-linearity by adding bias and activation function in between. Therefore this operation can be defined as

$$\psi'^{(KCConv)}_{(k)}(o) = \psi^{(Conv)}_{(k,2)}(\psi^{(Conv)}_{(k,1)}(o))$$

As [AP16] explained, their method forces the separability of the weight matrix as a hard constraint. By performing such an operation, they convert the computational complexity of a convolution operation to $\mathcal{O}(Kn(m^{(k-1)} + m^{(k)}))$. Suggesting that, similar to factorization methods, choosing a low-rank satisfying Equation 1.3, we can define a convolution operation with less computational complexity.

1.4.2 Separable Convolutions

Suggested by [Sif14], separable convolutions separate the standard convolution operation into two parts. These parts are called depthwise convolutions and pointwise convolutions. Separable convolutions are used by [Cho16], [HZC$^+$17] and [HZC$^+$17] to reduce complexity of neural networks.

Depthwise Convolution

Depthwise convolutions apply a separate convolution operation (referred to as inner convolution) on every input channel. Therefore, number of output channels of a depthwise convolution is the number of input channels times the number of output channels of these inner convolutions. In other words, it results with a number of output channels that is equal to (or folds of) the number of input channels. Unless defined otherwise, we use inner convolution operations with one output channel. Therefore the number of input channels are equal to the number of output channels for our depthwise convolutions.

Given a patch $p_{(I,J)}^{(k-1)} \in \mathbb{R}^{K \times K \times m^{(k-1)}}$, depthwise convolution has a single weight matrix $w^{(k,dw)} \in \mathbb{R}^{K \times K \times m^{(k-1)}}$. Let us assume that the subscripts of patch $p_{(I,J)}^{(k-1)}$ and weight matrix $w^{(k,dw)}$ are described as $p_{(I,J),i}^{(k-1)} \in \mathbb{R}^{K \times K \times 1}$ and $w_i^{(k,dw)} \in \mathbb{R}^{K \times K \times 1}$.

The depthwise convolution operation is defined as

$$\psi_{(k)}^{(dw)} : \mathbb{R}^{\mathcal{H}_{k-1} \times \mathcal{W}_{k-1} \times m^{(k-1)}} \to \mathbb{R}^{\mathcal{H}_k \times \mathcal{W}_k \times m^{(k-1)}}$$

$$\psi_{(k)}^{(dw)}(o^{(k-1)}) = o^{(k,dw)}$$

where the output $o^{(k,dw)}$ is given as

$$o^{(k,dw)} = \{o_{I,J,i}^{(k,dw)} \mid \forall(I,J,i)(\exists p_{(I,J),i}^{(k)})[o_{I,J,i}^{(k,dw)} = (w_i^{(k,dw)})^T p_{(I,J),i}^{(k-1)}]\}$$

In other words, depthwise convolution applies a $1 \times K \times K$ kernel to every $K \times K \times 1$ output channel to calculate every output channel of $o_{I,J}^{(k,dw)}$. The complexity of this operation is

$$\mathcal{O}(\psi_k^{(dw)}) = \mathcal{O}(H_k W_k K^2 m^{(k-1)})$$

Pointwise Convolution

Pointwise convolution ($\psi_{(k)}^{(pw)} : \mathbb{R}^{\mathcal{H}_k \times \mathcal{W}_k \times m^{(k-1)}} \rightarrow \mathbb{R}^{\mathcal{H}_k \times \mathcal{W}_k \times m^{(k)}}$) is a regular convolution operation with kernel size 1 ($K = 1$). The weight matrix that we will use for this operation is $w^{(k,pw)} \in \mathbb{R}^{1 \times 1 \times m^{(k-1)} \times m^{(k)}}$. The complexity of this operation is

$$\mathcal{O}(\psi_{(k)}^{(pw)}) = \mathcal{O}(\mathcal{H}_k \mathcal{W}_k m^{(k-1)} m^{(k)})$$

Since we have defined depthwise and pointwise convolutions, we can combine them to describe separable convolution function as

$$\psi_{(k)}^{(SConv)} : \mathbb{R}^{\mathcal{H}_{k-1} \times \mathcal{W}_{k-1} \times m^{(k-1)}} \rightarrow \mathbb{R}^{\mathcal{H}_k \times \mathcal{W}_k \times m^{(k)}}$$

$$\psi_{(k)}^{(SConv)}(o^{(k-1)}) = \psi_{(k)}^{(pw)}(\psi_{(k)}^{(dw)}(o^{(k-1)}))$$

The complexity of this operation is

$$\begin{aligned}
\mathcal{O}(\psi_{(k)}^{(SConv)}) =& \mathcal{O}(\psi_{(k)}^{(pw)} + \mathcal{O}(\psi_{(k)}^{(dw)})\\
=& \mathcal{O}(\mathcal{H}_k \mathcal{W}_k m^{(k-1)} m^{(k)} + \mathcal{H}_k \mathcal{W}_k K^2 m^{(k-1)})\\
=& \mathcal{O}(\mathcal{H}_k \mathcal{W}_k m^{(k-1)} (m^{(k)} + K^2))
\end{aligned}$$

1.5 Pruning

Pruning aims to reduce the model complexity by *deleting* the parameters that has low or no impact on the result. [LDS$^+$89] has shown that using the second order derivative of a parameter, we can estimate the effect it will have on the training loss. By removing the parameters that have low effect in the outcome, they have reduced the network complexity and increased accuracy. [HPTT16] has shown that there may be some neurons that are not being activated by the activation function (i.e. ReLU in their case). Therefore, they count the activations in neurons and remove the ones that have are not getting activated. Following pruning, they retrain their network and achieve better accuracy than non-pruned network. [HPTD15] shows that we can prune the weights that are very close to 0. By doing that they reduce the number of parameters in some networks about 10 times with no loss in accuracy. To do that, they train the network, prune the unnecessary weights, and train the remaining network again. [TBCS16] shows that using Fisher

Information Metric we can determine the importance of a weight. Using this information they prune the unimportant weights. They also use Fisher Information Metric to determine the number of bits to represent individual weights. Also, [Ree93] compiled many pruning algorithms.

1.5.1 Pruning Weights

This subcategory of pruning algorithms try to optimize the number of floating point operations by removing some individual values. In theory, it should benefit the computational complexity to remove individual scalars from $w^{(k)}$. However, in practice, we are defining our layers using dense matrix multiplications. To our knowledge, multiplying two dense matrices is faster than multiplying a dense matrix with a sparse matrix, unless we prune about 90% of the weight matrix.

1.5.2 Pruning Nodes

Since it is not possible to remove individual weights and reduce the computational complexity, we are going to look at another case of pruning. This case focuses on pruning a node and all the weights connected to it. Let us assume two fully connected layers, k and k+1. The computational complexity of computing the outputs of these two layers would be $\mathcal{O}(\psi^{(FC)}_{(k+1)}(\psi^{(FC)}_{(k)}(o^{(k-1)})) = \mathcal{O}(m^{(k)}(m^{(k-1)}+m^{(k+1)})$. Assuming that we have removed a single node from layer k, this complexity would drop by $m^{(k-1)} + m^{(k+1)}$.

Similar to the fully connected layer, a convolutional layer k also contains $m^{(k)}$ nodes. The only difference is, in a convolutional layer, these nodes are repeated in dimensions H_k and W_k. Therefore, it is possible to apply this technique to convolutional layers.

We will explain and use two simpler pruning criteria.

Activation Counts

We can count the activations per node to determine which nodes are not used. We can set a range using the mean and variance of activation counts and prune the nodes outside this range. By doing so, we can determine the nodes that are not frequently used or the nodes that are too frequently used.

Activation Variance

We can also collect statistics about output values per node. Using this information it is possible to determine which nodes are more important for the results by calculating the variance per node and removing the low variance nodes regardless of the mean value.

1.5.3 Pruning Convolutional Layers

Introduced by [HS06], autoencoders consist of encoder and decoder blocks. Encoder blocks use convolution operations to reduce the dimensionality of input. Decoder blocks use deconvolution operation to increase the dimensionality back to it's original form. The output of encoders are approximations of the input.

We use autoencoders in some of our pruning experiments because autoencoders have a clearly comparable baseline. In the baseline of an autoencoder, the dimensionality of the input would be equal to the output dimensions of every layer. Assuming an input $x_n \in \mathbb{R}^{\mathcal{H}_0 \times \mathcal{W}_0 \times m^{(0)}}$, we can assume the baseline autoencoder for this input by satisfying the following equality for every layer

$$\mathcal{H}_k \mathcal{W}_k m^{(k)} = \mathcal{H}_0 \mathcal{W}_0 m^{(0)}$$

Normally, an autoencoder aims to reduce the dimensionality using encoder blocks. This baseline definition is not good as an encoder, but it is a tool to compare our results.

1.6 Quantization

A floating point variable can not represent all decimal numbers perfectly. An n-bit floating point variable can only represent 2^n decimals. The decimals that can not be represented perfectly using 32-bits are going to be represented with some error. Quantization is the process used to represent values using less bits and some error.

At a higher level, the computational complexity does not depend on number of bits. But if we dive deeper in the computer architecture, using less bits to represent variables provide some major advantages. As the amount of bits get lesser, amount of cpu-cycles to perform an operation reduces and the cost of transferring data from memory to cpu cache reduces. Also it increases the amount of data that can fit into cache. One major disadvantage is, most architectures implement optimizations that speed up 16/32/64-bit floating point operations. By using less bits, we are giving up on these optimizations.

1.6.1 Weight Clustering

Introduced by [NH92], weight clustering starts with regular weight matrices. Once model is trained, they use clustering to find a set of weights($W' \in \mathbb{R}^a$) that approximate the model. Then they store the cluster index per weight in $d^{(k)} \in \mathbb{N}^{m^{(k-1)} \times m^{(k)}}$. By redefining $w^{(k)}_{i,j} = W'_{d^{(k)}_{i,j}}$, they perform weight clustering. Please note that this method does not necessarily reduce model complexity by itself. It reduces the model size by storing indices using less bits. In theory, such a method when applied before factorization should provide a lower rank in low-rank decomposition.

1.7 Improving Network Efficiency

1.7.1 Residual Connections

[HZRS15] introduced a method called residual connections. Let us assume blocks to be groups of consecutive layers in our network. We create a residual connection when we add the input of a block to the output of last layer in the block to calculate the output of the block. Let us assume a block b with input $o^{(b-1)} \in \mathbb{R}^{m^{(b-1)}}$ and output $o^{(b)} \in \mathbb{R}^{m^{(b)}}$. We call these two blocks residually connected if we perform $o^{(b)'} = o^{(b)} + o^{(b-1)}$ and set $o^{(b)'}$ as the input of the next block.

Residual connections allow us to train deeper networks by preventing the vanishing gradient problem. As we increase the number of layers in a neural network, the gradient values for weights in the former layers start getting smaller and smaller. They get so small that they become irrelevant and do not change anything. Residual connections increase the effect of deeper layers on the output. Therefore, their gradients do not vanish because they have significant contribution to the result.

Assume that we have residually connected layers from layer a to layer b.

The output of layer b would be equal to

$$o^{(b)} = \psi_{(b)}(o^{(b-1)}) + \sum_{k=a}^{b-1} o^{(k)}$$

Then the gradient of the former layers would have terms that are independent of the following layers, solving the vanishing gradient problem.

[HZRS15] trained networks with and without residual connections on ImageNet dataset. Their results show that introduction of the residual connections reduce the top-1 error rate of their 34 layer network from 28.54% to 25.3%.

[HZRS15] also introduced two types of residually connected blocks. The first is called a residual block, consisting of two convolution operations and a residual connection between the input and the output of the block. The second is called a residual bottleneck block, consisting of three convolution operations. First reducing number of channels with a one by one kernel, second applying a three by three kernel, third applying another one by one kernel to increase the number of dimensions. They have used residual blocks to train networks up to 34 layers. For networks having 50 or more layers, they have used the residual bottleneck block. Their 50-layer network using residual bottleneck blocks achieves 22.85% top-1 error rate on ImageNet dataset while their 34-layer network is achieving 25.3% top-1 error rate.

1.7.2 Batch Normalization

[IS15] introduced a method called batch normalization. Batch normalization aims to normalize the output distribution of a every node in a layer. By doing so it allows the network to be more stable.

Assume the layer k with $o^{(k)} \in \mathbb{R}^{m^{(k)}}$ where $m^{(k)}$ is the number of nodes. Batch normalization has four parameters. Mean is $\mu^{(k)} \in \mathbb{R}^{m^{(k)}}$, variance is $\sigma^{(k)} \in \mathbb{R}^{m^{(k)}}$, scale is $\gamma^{(k)} \in \mathbb{R}^{m^{(k)}}$ and offset is $\beta^{(k)} \in \mathbb{R}^{m^{(k)}}$.

Since we are interested in normalizing the nodes, even if k was a convolutional layer, the shapes of these parameters would not change. Therefore, batch normalization function BN can be defined as

$$BN(o^{(k)}) = \frac{\gamma^{(k)}(o^{(k)} - \mu^{(k)})}{\sigma^{(k)}} + \beta^{(k)}$$

1.7.3 Regularization

Regularization methods aim to prevent overfitting in neural networks. Overfitting is the case where the weights of a a neural network converge for the training dataset. Meaning that the network performs very good for the training dataset, while it is not generalized to work with any other data. Regularization methods try to prevent this.

One common regularization method is to add a new term to the loss, which influence the weight in certain ways. We also add a term λ which determines the effect of this regularization. Setting λ too high will influence the gradient descent steps more than the data itself. In such a case, we may end up with a non-optimal solution. Setting λ too low will reduce the effects of regularization. We look at two types of regularizers, L1 and L2.

L1 Regularization

L1 regularization pushes regularized values towards zero. Therefore, it is good to force the weights to become small or very close to zero. L1 regularization is defined as

$$L1 = \lambda \sum_{w \in \mathbf{W}} |w|$$

L2 Regularization

L2 regularization punishes values with a square term. Therefore, L2 regularization pushes the weights towards zero. However, it pushes the values that are greater than one or minus one more than the values in between. L2 regularization is defined as

$$L2 = \lambda \sum_{w \in \mathbf{W}} w^2$$

1.8 Datasets

In this section we will see the datasets that we have experimented with. Since we are mostly focusing on convolutional neural networks, we will look at 3 image classification datasets.

1.8.1 MNIST

MNIST dataset [LCB98] consists of 60.000 training and 10.000 test samples. Each sample is a 28×28 black and white image of a handwritten digit (0 to 9). To our knowledge, the best model trained on MNIST achieve almost zero (0.23%, [CMS12]) error rate.

1.8.2 CIFAR10

CIFAR10 dataset [KH09] consists of 50.000 training and 10.000 test samples. Each sample is a 32×32 colored image belonging to one of 10 classes. The classes are airplane, automobile, bird, cat, deer, dog, frog, horse, ship and truck. To our knowledge, the best model trained on CIFAR10 achieve 3.47% ([Gra14]) error rate.

1.8.3 ImageNet

The dataset used in ILSVRC is called ImageNet. ImageNet [DBS+12] comes with 1.281.167 training images and 50.000 validation images consisting of 1000 classes containing multiple dog species and daily objects. ImageNet comes with bounding boxes showing where the object is in the image. We are interested in the object detection task. So we crop these bounding boxes and feed them to our neural network for training. The best submission from 2016 challenge has achieved 0.02991 error rate. This is equal to 97.009% top-5 accuracy.

Chapter 2

Methods

So far we have explained some neural network building blocks and some techniques for reduced complexity or increased efficiency. In this chapter, first we are going to explain the experiments we ran to get a better understanding of these methods. Then, we are going to talk about the model we came up with combining these methods.

2.1 Pruning

Since pruning individual weights do not affect complexity directly, we are going to focus on pruning nodes. To do that, as [HPTT16] explained, we are going to use *training cycles*. We will define two neural networks for two easy problems. Then we will train each neural network using its training dataset. After the training is done, we will collect ReLU output statistics on that training dataset without updating weights. Using these statistics, we will try to understand which nodes have minimal effect on the outcome. We will prune these nodes by removing the relevant dimensions from the weight matrices. Then we will go back to the training step. We will keep iterating over these steps until we can not find any nodes to be pruned.

First we will explain these two easy problems. Then, we will define the methods that we have used to optimize the number of pruned nodes.

2.1.1 Fully Connected Networks

We have defined a neural network consisting of 2 input dimensions ($x_n \in \mathbb{R}^2$), one fully connected layer with 1000 nodes and one fully connected output with a single node ($y_n \in \mathbb{R}$). We have defined the expected output as the summation of two inputs, ($y_= x_{n,1} + x_{n,2}$). As a result, we precisely know the optimum neural network structure that would be able to perform this calculation. Which is a neural network with one fully connected layer with one node and an output layer with one node fully connected to that. All weights equal to 1 and all biases equal to 0.

We have calculated the loss using RMSE, and used Momentum Optimizer (learning rate 0.01 and momentum 0.9) to learn the weights. We have generated 1.000.000 samples, we trained the network with batch size 1000.

2.1.2 Convolutional Neural Networks

We have extended our pruning experiments to convolutional neural networks. To do so, we have trained an autoencoder on MNIST dataset. We have defined our autoencoder with two encoder and two decoder layers. Each encoder layer ($\psi_{(1)}^{(Conv)}$ and $\psi_{(2)}^{(Conv)}$) is running a convolution with kernel size 3 and stride of 2. After each encoding layer, we add bias, apply batch normalization and ReLU activation. Each decoding layer ($\psi_{(3)}^{(Deconv)}$ and $\psi_{(4)}^{(Deconv)}$) is running deconvolutions with kernel size of 3 and strides of two. Followed by adding bias, batch normalization and ReLU activations. We defined the loss as the root mean square of the input and the output of the network.

Overall, the functions and the variables we have used to define this neural network are

$$\begin{aligned}
x_n &\in \mathbb{R}^{28\times 28\times 1} \\
\psi_{(1)}^{(Conv)} &: \mathbb{R}^{28\times 28\times 1} \rightarrow \mathbb{R}^{14\times 14\times 32} \\
\psi_{(2)}^{(Conv)} &: \mathbb{R}^{14\times 14\times 32} \rightarrow \mathbb{R}^{7\times 7\times 64} \\
\psi_{(3)}^{(Deconv)} &: \mathbb{R}^{7\times 7\times 64} \rightarrow \mathbb{R}^{14\times 14\times 32} \\
\psi_{(4)}^{(Deconv)} &: \mathbb{R}^{14\times 14\times 32} \rightarrow \mathbb{R}^{28\times 28\times 1} \\
\mathcal{L} &= \text{RMSE}(x_n, \hat{y}_n)
\end{aligned}$$

2.1.3 Pruning Strategies

In this subsection we will look at the strategies we have followed to optimize the number of pruned nodes.

Activations

We experiment with 2 pruning configurations. There are different alternatives on choosing the neurons or weighs to prune. But we stick to the two simplest methods that were explained.

Regularization

In both experiments, we test no regularization, L1 regularization and L2 regularization to see how they effect pruning.

Distortion

In cases where some nodes are mostly activated together, we can assume that they are representing similar features. To prevent such cases, we tried to distort the weights between training cycles and force some difference in feature representations.

2.2 Convolution Operation Alternatives

In general, convolution operations are expensive. In this section, we experiment with *kernel composing convolutions* and *separable convolutions* to see which one is the better alternative. To see the differences between these operations, we ran two experiments for each. One classifying MNIST dataset, the other classifying CIFAR-10 dataset.

2.2.1 Baseline Model

We have defined a convolutional neural network with three convolutional layers followed by one fully connected layer. Each convolutional layer has kernel size 5. Convolutional layers are followed bias addition, batch normalization, and finally ReLU activations. First two convolution layers are followed by a max pooling layer with kernel size 2 and strides of 2. The third convolutional layer is followed by a global average pooling layer, where we reduce the width and height dimensions to the average of all values in those dimensions. Using SCE loss and momentum optimizer (learning rate 10^{-4}, and momentum 0.9), we train these networks for 20000 steps with batch size 32.

Overall, the functions and the variables we have used to define this neural network are

$$\begin{aligned}
x_n &\in \mathbb{R}^{32\times32\times1} \\
\psi_{(1)}^{(Conv)} &: \mathbb{R}^{32\times32\times3} \rightarrow \mathbb{R}^{32\times32\times32} \\
\psi_{(2)}^{(avgpool)} &: \mathbb{R}^{32\times32\times32} \rightarrow \mathbb{R}^{16\times16\times32} \\
\psi_{(3)}^{(Conv)} &: \mathbb{R}^{16\times16\times32} \rightarrow \mathbb{R}^{16\times16\times64} \\
\psi_{(4)}^{(avgpool)} &: \mathbb{R}^{16\times16\times64} \rightarrow \mathbb{R}^{8\times8\times64} \\
\psi_{(5)}^{(Conv)} &: \mathbb{R}^{8\times8\times64} \rightarrow \mathbb{R}^{8\times8\times128} \\
\psi_{(6)}^{(avgpool)} &: \mathbb{R}^{8\times8\times128} \rightarrow \mathbb{R}^{128} \\
\psi_{(7)}^{(FC)} &: \mathbb{R}^{128} \rightarrow \mathbb{R}^{10} \\
\mathcal{L} &= \mathrm{SCE}(x_n, \hat{y}_n)
\end{aligned}$$

$\psi_{(k)}^{(KCConv)}$ and $\psi_{(k)}^{(SConv)}$ we obtain 3 versions of this model.

2.2.2 Kernel Composing Convolutions

By changing second and third $\psi_{(k)}^{(Conv)}$ with $\psi_{(k)}^{(KCConv)}$ we compare the effects of using kernel composing convolution operation. Remember that kernel composing convolution operation requires an additional parameter for number of intermediate output channels. We use the number of output channels for that.

2.2.3 Separable Convolutions

By changing second and third $\psi_{(k)}^{(Conv)}$ with $\psi_{(k)}^{(SConv)}$ we compare the effects of using separable convolution operation.

2.2.4 Non-Linear Separable Convolutions

[HZC+17] proposed that, adding batch normalization and ReLU activations between depthwise and pointwise convolutions in separable convolutions would increase the model accuracy. To test that we compare non-linear separable convolutions and separable convolutions.

2.3 Separable Resnet

Inspired by ResNet ([HZRS15], [HZRS16]), we have created a convolutional neural network with residual connections. In Figure 2.1 we show the architecture of our neural network.

2.3.1 Model Choices

These are the choices that we made while designing our neural network. We try to keep the model complexity low while trying to achieve maximum accuracy.

- In [HZRS15], ResNet-34 starts with a 7×7 convolution with strides of two and it is followed by max pooling layer with strides of two and kernel size 2. If we think about this design choice, we see that the kernel size choice (7×7) is to minimize the loss of information caused by two layers with strides of two. From another point of view, those two layers decrease the image size from 224×224 to 56×56. This decreases the complexity of future layers. When we apply such a convolution, the first convolutional layer becomes very complex compared to the rest of the network. To prevent that we propose a different first layer. We propose to change the first convolutional kernels from 7×7 to 3×3 and halve the output channels. Then, before applying the max pooling layer, we apply a 3×3 depthwise convolution that multiplies the number of channels with two. Then we apply a 1×1 convolution (pointwise) to that. By doing so we reduce the complexity of these layers about four times. We ran experiments comparing this proposed first layer and the original one.

- Except for the first convolutional operation, we have replaced every convolution layer with a separable convolution layer. The first layer

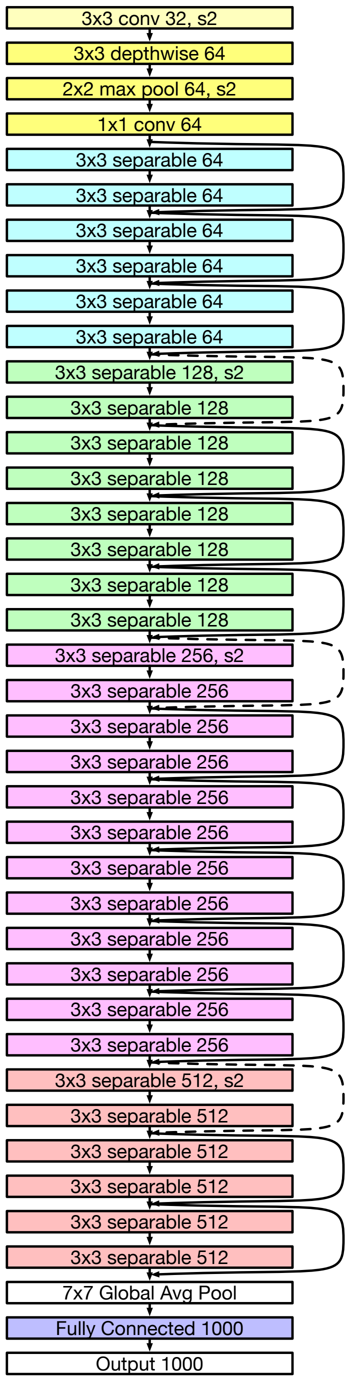


Figure 2.1: Separable Resnet-34. Branching lines represent residual connections, dashed ones are padded with zeros to match the number of channels. If s=2, depthwise convolution is ran with strides of two and residual branch is average pooled with strides 2 and kernel size 2. ReLU and Batch Normalization operations are hidden.

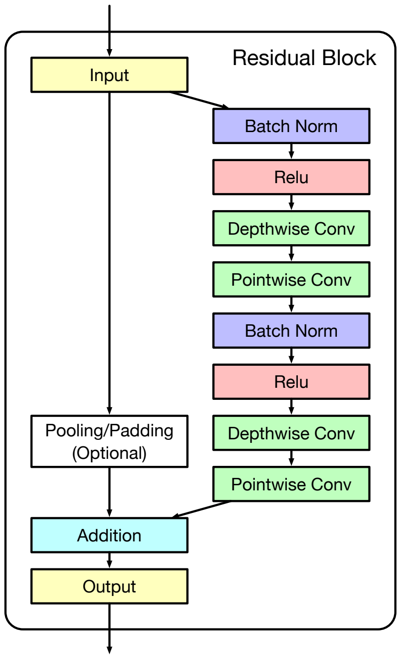


Figure 2.2: Full pre-activation residual connections.

has 3 input and 32 output channels. Therefore, for this layer the number of FLOPs for a convolution operation ($K * K * m^{(k-1)} * m^{(k)} = 3 * 3 * 3 * 16 = 864$) is sufficiently higher than a separable convolution ($K * K * m^{(k-1)} + m^{(k-1)} m^{(k)} = 3 * 3 * 3 + 3 * 32 = 123$). But for this layer, separable convolution comes with a disadvantage. By definition, it applies a depthwise convolution for color channels separately. We believe that the feature representations of the input are dependent on the information from different color channels. Therefore we argue that applying a depthwise convolution without mixing the colors is inefficient, and not change the first convolution operation to a separable convolution.

- As [HZRS16] proposed, we are using full pre-activation residual connections. See Figure 2.2.

2.3.2 Training

We train our network using CIFAR-10 and ImageNet datasets.

CIFAR-10

We divided the dataset for 50.000 training images and 10.000 validation images. We used momentum optimizer with momentum 0.9 and learning rates 0.1, 0.01, 0.001 for steps 0 to 40.000, 40.000 to 60.000 and 60.000 to 80.000 respectively. We have defined the loss with SCE of the truth and prediction, with an addition of L2 norm of weights multiplied by 0.001. We trained our model using the training images for 80.000 steps with batch size 128.

We preprocess the images using the routines defined in Tensorflow tutorials[1][2]. We start by taking 24×24 random crops and then we randomly flip the image to left or right. Then we randomly change the brightness and contrast. Then we normalize this image by subtracting the mean and dividing by variance by using a method called per_image_standardize.

For CIFAR-10 training, we make some changes in our model. Since we have defined our input as a 24×24 image, we can apply a total of three convolutions with stride of two. After these, the image dimensions become 3×3. After this point for our convolutions to make sense, we can not apply convolutions with strides of two until we apply global average pooling. Therefore, we remove the strides from green and pink blocks in Figure 2.1 and we do not multiply the number of channels by two.

We train our model with CIFAR-10 to be able to verify our configuration before we train on ImageNet.

ImageNet

We trained our model in ImageNet training dataset. We used momentum optimizer with momentum 0.9 and learning rates 0.01, 0.001, 0.0001, 0.00001 for steps 0 to 150.000, 150.000 to 300.000, 300.000 to 500.000 and 500.000 to 600.000 respectively. We have defined the loss with SCE of the truth and prediction, with an addition of L2 norm of weights multiplied by 0.001. We train our model using the training images for 200.000 steps with batch size 128.

We preprocess images using the routines defined for open sourced Ten-

[1]`https://github.com/tensorflow/models/tree/master/tutorials/image/cifar10`

[2]`https://www.tensorflow.org/tutorials/deep_cnn#cifar-10_model`

sorflow implementation of inception network[3]. We start by creating a new random bounding box overlapping with the original bounding box and make sure that 0.1 of the bounding box is inside our new bounding box. Then we crop this new bounding box and resize it using bilinear resizing algorithm. Then we randomly flip it to left or right. Then we distort the colors using random brightness and saturation. Then we normalize this input to the range of $[-1, 1]$ by subtracting 0.5 and multiplying by 2.

2.3.3 Pruning Nodes

After training the model for the given number of steps, we prune the nodes and retrain the model until it converges. We repeat this process until we can not find any nodes to prune.

Pruning Residual Connections

The addition operation in the residual block creates a one-to-one relationship between the outputs of different blocks. With the existence of such a relationship, it is not possible to prune a residual block's output nodes without pruning the former or latter blocks. Therefore, we first group the residual blocks that are directly connected to each other. We consider the residual blocks that do not apply pooling/padding operation to be directly connected to the previous residual block (in Figure 2.1, consecutive straight arrows between dashed arrows are directly connected). Then we calculate the indexes of nodes to keep from the output of every residual block. We union these indexes and prune the remaining nodes from the outputs of every directly connected residual block. We also prune the operations within residual blocks separately.

2.3.4 Factorization

We apply pruning weights, weight clustering before factorization to be able to reduce the computational complexity as much as possible.

[3]`https://github.com/tensorflow/models/blob/master/slim/preprocessing/inception_preprocessing.py`

Pruning Weights

We set weight values that are very close to 0 to 0. Despite the fact that this operation does not change computational complexity, when combined with factorization, it helps finding a lower rank.

Weight Clustering

We round the weights after the second decimal. Again, doing so helps factorization to have a lower rank.

Factorization

We factorize the trained, pruned and rounded weights using SVD. We can not apply this method to depthwise convolution operation. So we only factorize the convolution, pointwise convolution and fully connected weights. To do that, we calculate U, S and V for each of the weight matrices. We lower the rank of the decomposition by one. Then, we calculate the approximation error. If the approximation error is above a threshold, we check if this low rank decomposition would reduce the complexity. If it does, we use the decomposition. Otherwise we use the non-factorized weights.

2.3.5 Quantization

We quantize the weights from 32-bits to 8-bit and run performance and accuracy benchmarks.

2.4 Benchmarking

We benchmark our models on a *One Plus X*[4] mobile device, equipped with a *Snapdragon 801*[5] chipset. Our benchmarks consist of running consecutive inferences on a model, for a period of time. While those inferences are running, we collect hardware statistics using simpleperf[6]. Simpleperf lets us collect the hardware level performance statistics for a given process.

[4]`https://oneplus.net/x/specs`
[5]`https://www.qualcomm.com/products/snapdragon/processors/801`
[6]`https://android.googlesource.com/platform/prebuilts/simpleperf/`

Our benchmarking app uses a static input image (with dimensions depending on the model). So that we can ignore the overhead of pre-processing. Also, we perform no post-processing. Doing so, we try to avoid the effects of any other computation that could change the benchmark results.

We ran this benchmarking tool for various models. Since running the benchmark does not require a trained network, we could easily generate multiple models and benchmark them. These models include; Inception-Resnet v2 ([SIV16]), Inception v1 ([SLJ+14]), Inception v2 ([SLJ+14]), Inception v3 ([SVI+16]), Inception v4 ([SIV16]), VGG-19 ([SZ14]), ResNet-50, ResNet-101, ResNet-152 and ResNet-200 ([HZRS15], [HZRS16]) and Mobilenet ([HZC+17]).

Chapter 3

Results

3.1 Pruning

3.1.1 Fully Connected Layers

By applying distortions to remaining weights between training cycles, we have achieved the optimum result, where we have only one hidden node remaining. When we did not apply the distortions, after the first training cycle, we could not find any nodes to prune. Using both activation count and activation value statistics we found an ideal solution in 7 ± 2 training cycles. L1 or L2 regularization did not make an obvious difference.

3.1.2 Convolutional Neural Networks

In this setting we did not see any change when we applied the distortions. Compared to L1 regularization, with L2 regularization, we have seen some improvement in the number of nodes pruned and the final result. The most ideal case was, with no distortion, l2 regularization, pruning nodes based on activation value statistics under a threshold. We tried various thresholds, but average activation values minus two times the standard deviation of activation values worked the best. Using this setting, we have pruned the autoencoder from $1 - 32 - 64 - 32 - 1$ to $1 - 2 - 4 - 4 - 1$ nodes per layer. It took us 10 ± 4 training cycles to achieve these results. The loss we have achieved is very close to 0. Results can be seen in Figure 3.1.

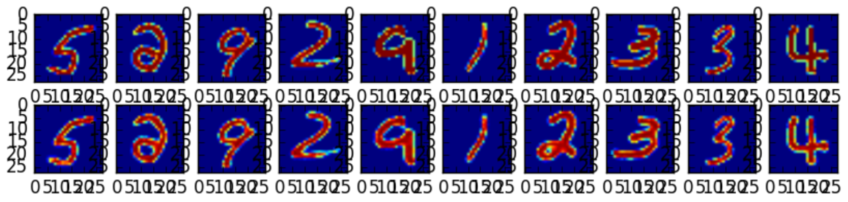


Figure 3.1: Sample results from the validation dataset after pruning

3.2 Convolution Operation Alternatives

3.2.1 MNIST

In our experiments with MNIST dataset, we have not seen a comparable difference between experiments with different operations. All models have achieved $99 \pm 0.3\%$ top-1 accuracy.

3.2.2 CIFAR-10

In our experiments with CIFAR-10 dataset, we have seen that kernel composing convolutions and convolutions are almost similar in terms of accuracy performance. We have seen that separable convolutions are slightly better than both operations. We have seen that separable convolutions achieve 82.3% mean top-1 validation accuracy while regular convolutions achieve 81.6% and kernel composing convolutions achieve 81.8% in the whole validation dataset. Also in Figure 3.2 we see the validation performance comparison for these operations. In our experiments we could not see any difference between adding non-linear separable convolutions and separable convolutions.

3.3 Separable Resnet

This section is left blank intentionally. We are waiting for the results of ImageNet trained network.

3.4 Benchmarks and Comparisons

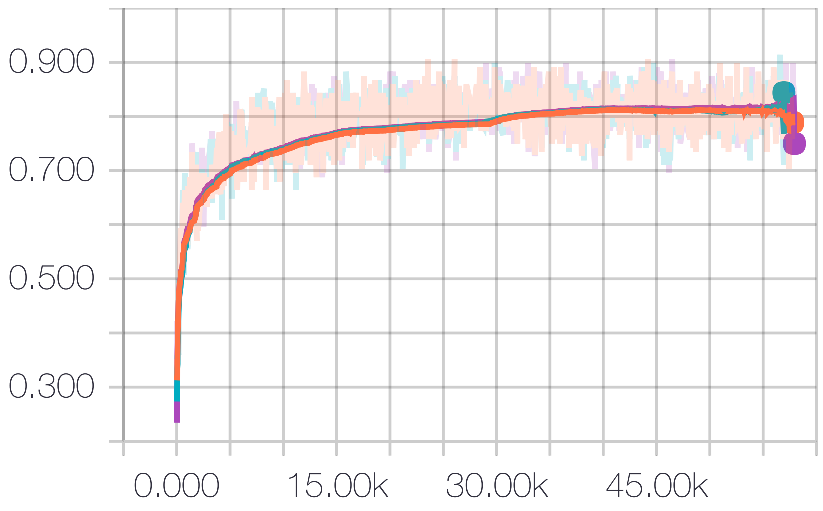
We have benchmarked 20 models in different sizes and shapes with our benchmarking device. Here we will report the key models that made a difference.

Inception-Resnet-v2 ([SIV16]) has a model size of 224 MB and our benchmarking device could execute 0.3-0.4 inferences per second. The cpu-utilization with this model was 0.006568. On a 4 core device, this number is extremely low, so we can conclude this model is cursed by the memory bottleneck bandwidth.

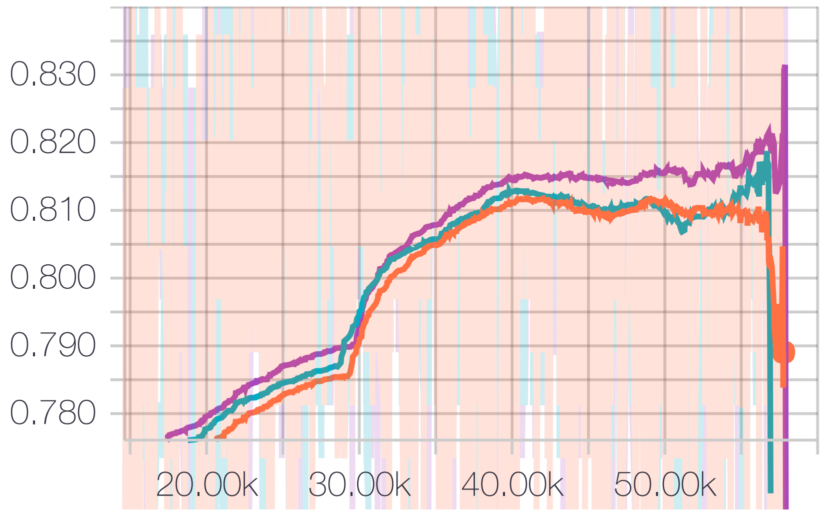
We have seen that ResNet-50 ([HZRS15], [HZRS16]) with a model size of 100 MB got executed for 1.3-1.7 inferences per second with a cpu-utilization of 2.7. We found this amount to be great compared to the previous model.

Among many models we have inspected, only 1.0 MobileNet-224 ([HZC+17]) performed faster than our separable resnet. 1.0 Mobilenet-224 had a model size of 17.1 MB and our benchmarking device could perform 5 to 6 inferences per second on with a cpu-utilization of 2.7. On paper, this model has We did not test smaller versions of MobileNet because they are far from having a state of the art accuracy.

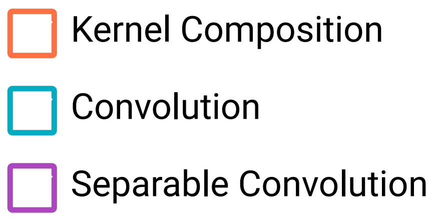
Our model, separable resnet had a model size of 9.9 MB. We could perform 4 to 5 inferences per second with a cpu-utilization of 3.1 on our benchmarking device.



(a) Smoothed top-1 accuracies in every step. For validation dataset.



(b) Zoomed in version of Figure 3.2a.



(c) Color codes

Figure 3.2: Top-1 accuracy comparison of kernel composing convolution, convolution and separable convolution operations. Accuracies for sub-samples of the validation dataset, compared after every training step. We see the thick lines representing smoothed values.

Chapter 4

Discussion

4.1 Pruning and Factorization

In our experiments with pruning, we started with very large models for the given problem. We believe that this might have left a false impression. As it did in our experiments, pruning a model would not reduce the model complexity by 1000 times for every model. Depending on the model size and the problem definition, pruning will help us reduce the model complexity, but if our model is sufficiently small for the problem definition, it may not help us at all. In our experiments we have seen that pruning separable resnet did not reduce the model complexity significantly.

Similarly, if we apply factorization on a very large model, we gain huge speed ups with minor effort. But if our model is compact enough, the complexity gains from these models reduce significantly.

4.2 Using Tensorflow

We have been using latest versions of Tensorflow. It comes with some advantages, such as:

- We do not implement lower level operations (such as convolutions). It gives us the opportunity to focus on higher level implementations, such as pruning, or factorization.

- Most of the operations are highly optimized for many platforms and devices. If we were to implement a model in C++, we'd have to implement it twice, one for training in GPU and another for running in the mobile device.

- Tensorflow provides the necessary tools to deploy models on mobile devices.

And it comes with some disadvantages, such as:

- When we started our work, Tensorflow was in version 0.10. By the date we write this, it is on 1.2. There have been 4 major releases that we had to modify our codebase for.
- Not all operations are properly implemented. For example, before version 1.2, Tensorflow implementation of separable convolutions were not very well optimized. They were as fast as convolution operations. Before that we could only hope that they would optimize their implementation.
- It is difficult to implement operations (e.g. ef operator [AYN+17]) or play around with existing ones. The documentation describing C++ internals and build procedures (as of Tensorflow 1.2) are not good enough.
- Tensorflow does not provide tools to implement low-bit variables (e.g. a 2-bit integer). So it is not possible to implement some methods that make use of variable width decimals. This limitation makes some methods impossible to use or useless. For example it is not possible to use methods that represent weights using variable width decimals. Also, storing low bit weight indices in combination with a small global weight array to reduce the model size is useless. Since we can not use low bit integers to represent these indices, our model size does not shrink at all.

4.3 Operation Comparison

In Section 2.2, we have defined a neural network to compare convolution, separable convolution and kernel composing convolution operations. Before our experiments, we have tried to find the best settings for some parameters, such as learning rate, regularization constant and optimizer. We think that using the same settings may have influenced our results. Especially because the number of parameters change considerably when we use separable convolutions, instead of convolutions.

4.4 Model Comparison

Comparing models that aim for mobile devices is difficult. First, there are device/chip specific properties (i.e. l1-cache size and bandwidth speed) that in theory effect the speed of the model greatly. In theory, mobile device performance of a model depends on the amount of floating point multiplications and the model size. The amount of floating point multiplications would modify the model speed almost linearly. The model size is important, because in theory, it determines the amount of data transferred from memory to l1-cache. If our model and the required space to perform the operations in it are sufficiently small to fit in the l1-cache, it would speed up our model greatly by getting rid of all cache-misses. If these were bigger than the l1-cache, it would create a lack of storing and loading from memory, which would lead to waiting for data coming from memory (memory bandwidth bottleneck). So in theory, model size would effect the performance non-linearly. In practice, we could not find a good way of predicting the number of inferences based on number of floating point operations and model size.

One thing that greatly affects this process is the implementation of the model executor. We think that the Tensorflow implementation of the model executor is not meant to load the whole model on the cpu.

Chapter 5

Conclusion

In this research, we have investigated some methods to reduce the computational cost of convolutional neural networks. To do that, we experimented with some methods that could be used to define models with lower computational cost. We also experimented with some methods to reduce the computational complexity of a given model.

In our experiments we have observed that the models using separable convolutions result with a slightly better accuracy compared to models using convolution or kernel compositing convolution operations. We also saw that kernel composing convolution operation is a good alternative to convolution operation, only if the kernel size is larger than 3. However, most state of the art convolutional neural networks use convolutions with kernel size 3. Therefore, this method is not really an alternative to convolution operation.

To be able to experiment with pruning using larger models, we have implemented a tool to describe pruning routines. We implemented pruners for various layer types and functions, such as convolutions, separable convolutions, batch normalization, fully connected layers and residual connections. We also implemented a module to collect activation statistics for given ReLU activations. Using this tool other researchers can also experiment with pruning neural networks using Tensorflow.

We have also implemented a tool that applies quantization, pruning and factorization to a given trained model. Just as we have seen with pruning, we were unable to achieve the speed ups and accuracy gains reported previously.

When developing models aimed for mobile environments, we think that training a compact model and applying compression techniques is a better alternative to compressing a large network.

Bibliography

[AP16] Jose Alvarez and Lars Petersson. Decomposeme: Simplifying convnets for end-to-end learning. *arXiv preprint arXiv:1606.05426*, 2016.

[AYN+17] Arman Afrasiyabi, Ozan Yildiz, Baris Nasir, Fatos T Yarman Vural, and A Enis Cetin. Energy saving additive neural network. *arXiv preprint arXiv:1702.02676*, 2017.

[Cho16] François Chollet. Xception: Deep learning with depthwise separable convolutions. *arXiv preprint arXiv:1610.02357*, 2016.

[CMS12] Dan C. Ciresan, Ueli Meier, and Jürgen Schmidhuber. Multi-column deep neural networks for image classification. *CoRR*, abs/1202.2745, 2012.

[CPC16] Alfredo Canziani, Adam Paszke, and Eugenio Culurciello. An analysis of deep neural network models for practical applications. 05 2016.

[CS16] Jaeyong Chung and Taehwan Shin. Simplifying deep neural networks for neuromorphic architectures. In *Design Automation Conference (DAC), 2016 53nd ACM/EDAC/IEEE*, pages 1–6. IEEE, 2016.

[DBS+12] J Deng, A Berg, S Satheesh, H Su, and A Khosla. Image net large scale visual recognition competition. *(ILSVRC2012)*, 2012.

[DZB+14] 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*, pages 1269–1277, 2014.

[GBB11] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, pages 315–323, 2011.

[GDN13] Pavel Golik, Patrick Doetsch, and Hermann Ney. Cross-entropy vs. squared error training: a theoretical and experimental comparison. In *Interspeech*, volume 13, pages 1756–1760, 2013.

[GR70] Gene H Golub and Christian Reinsch. Singular value decomposition and least squares solutions. *Numerische mathematik*, 14(5):403–420, 1970.

[Gra14] Benjamin Graham. Fractional max-pooling. *CoRR*, abs/1412.6071, 2014.

[HPTD15] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. In *Advances in Neural Information Processing Systems*, pages 1135–1143, 2015.

[HPTT16] Hengyuan Hu, Rui Peng, Yu-Wing Tai, and Chi-Keung Tang. Network trimming: A data-driven neuron pruning approach towards efficient deep architectures. 07 2016.

[HS06] Geoffrey E Hinton and Ruslan R Salakhutdinov. Reducing the dimensionality of data with neural networks. *science*, 313(5786):504–507, 2006.

[HZC$^+$17] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[HZRS15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. 12 2015.

[HZRS16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. In *European Conference on Computer Vision*, pages 630–645. Springer, 2016.

[IS15] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[KB14] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[KH09] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. 2009.

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[LCB98] Yann LeCun, Corinna Cortes, and Christopher JC Burges. The mnist database of handwritten digits, 1998.

[LDS$^+$89] Yann LeCun, John S Denker, Sara A Solla, Richard E Howard, and Lawrence D Jackel. Optimal brain damage. In *NIPs*, volume 2, pages 598–605, 1989.

[NH92] Steven J Nowlan and Geoffrey E Hinton. Simplifying neural networks by soft weight-sharing. *Neural computation*, 4(4):473–493, 1992.

[NH10] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010.

[Qia99] Ning Qian. On the momentum term in gradient descent learning algorithms. *Neural networks*, 12(1):145–151, 1999.

[Ree93] Russell Reed. Pruning algorithms-a survey. *IEEE transactions on Neural Networks*, 4(5):740–747, 1993.

[Sif14] L Sifre. *Rigid-motion scattering for image classification.* PhD thesis, Ph. D. thesis, 2014.

[SIV16] Christian Szegedy, Sergey Ioffe, and Vincent Vanhoucke. Inception-v4, inception-resnet and the impact of residual connections on learning. *CoRR*, abs/1602.07261, 2016.

[SLJ$^+$14] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. 09 2014.

[SVI$^+$16] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[SZ14] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. 09 2014.

[TBCS16] Ming Tu, Visar Berisha, Yu Cao, and Jae-sun Seo. Reducing the model order of deep neural networks using information theory. In *VLSI (ISVLSI), 2016 IEEE Computer Society Annual Symposium on*, pages 93–98. IEEE, 2016.

[ZKTF10] Matthew D Zeiler, Dilip Krishnan, Graham W Taylor, and Rob Fergus. Deconvolutional networks. In *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on*, pages 2528–2535. IEEE, 2010.

[ZZHS16] Xiangyu Zhang, Jianhua Zou, Kaiming He, and Jian Sun. Accelerating very deep convolutional networks for classification and detection. *IEEE transactions on pattern analysis and machine intelligence*, 38(10):1943–1955, 2016.