Master's Thesis

Optimizing Neural Networks for Mobile Devices

Radboud University, Nijmegen

Erdi Çallı

June 24, 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] won the ImageNet Large Scale Visual Recognition Competition (ILSVRC) classification task ([DBS+12]). They used two GPUs to train an 8 layer convolutional neural network (CNN). 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. Today there are GPUs with up to 12 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 affects 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 is increased to $\sim 97\%$.

In contrast, [SLJ+14] have shown that having a good harmony within the network worked better than having more parameters. It has been supported by [CPC16]. They have 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. Therefore, revealing that of providing more layers and parameters would not yield better results.

ILSVRC is one of the most famous competitions in image processing. Every year, the winners of this competition are a driving the research on the field. But this competition is not considering the competitive value of limiting number of operations. If we look at the models of 2016 competitors, we see that they use ensembles of models[1]. These ensembles are far from being usable in real life because they require a great amount of operations per inference. Not mentioning the number of operations from the result is misleading for the AI community and the public. It creates an unreal expectation that these models are applicable in real life. In this thesis, we want to come up with a state of the art solution that requires a low number of floating point operations per inference. Therefore, bridging the gap between expectations and reality. We will answer,

> How can we reduce the inference complexity of convolutional neural networks?

First, we will briefly describe neural networks and some underlying concepts. We will mention the complexities 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 our results.

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 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 as $w^{(k)}$. We use the parentheses to separate the index from the power. We use those indexed sets to represent a semantic group of variables that may have different properties, such as shape, or dimensions, or type. Therefore, it wouldn't be right to represent them using a tensor or a matrix. Having such a definition, we will not

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

be separating scalars, vectors, matrices or tensors using capitals or bolds. However, we will be defining the variables we use 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. $w^{(k)}_{i,j}$ would represent the ith column and jth row of this matrix. We may choose to write the same variable as $w^{(k)}_{ij}$, without the comma between indices. Or we may choose to use the commas or parentheses to group 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 chapter, we will try to describe neural networks briefly. We will 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*. In our case, 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 mostly called 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) the outputs of the nodes belonging to previous layers . Preceding that, the output of the input layer is calculated using the input data (see Eq. 1.2). 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.

$$
\begin{aligned}
L&: \text{the number of layers in a neural network}\\
l^{(k)}&: \text{layer } k\\
m^{(k)}&: \text{the number of nodes in } l^{(k)}\\
l_i^{(k)}&: \text{node } i \text{ in } l^{(k)}\\
o^{(k)}&: \text{the output vector representing the outputs of nodes in } l^{(k)}\\
o_i^{(k)}&: \text{the output of } l_i^{(k)}\\
w^{(k)}&: \text{weight matrix connecting nodes in } l^{(k-1)} \text{ to nodes in } l^{(k)}\\
w_{i,j}^{(k)}&: \text{the weight connecting nodes } l_i^{(k-1)} \text{ and } l_j^{(k)}\\
b^{(k)}&: \text{the bias vector for } l^{(k)}\\
\psi_{(k)}&: \text{function to determine } o^{(k)} \text{ given } o^{(k-1)}\\
\sigma&: \text{activation function}\\
X&: \text{all inputs of the dataset as}\\
Y&: \text{all provided outputs of the dataset}\\
\hat{Y}&: \text{approximations of all outputs given all inputs}\\
x_n&: n\text{th input data}\\
y_n&: n\text{th output data}\\
\hat{y}_n&: \text{approximation of } y_n \text{ given } x_n
\end{aligned}
\tag{1.1}
$$

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

$$
o^{(k)} = \begin{cases} \psi_{(k)}(o^{(k-1)}), & \text{if } k \geq 1 \\ x_n, & k = 0 \end{cases} \tag{1.2}
$$

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's 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 ($b^{(k)} \in \mathbb{R}^{m^{(k)}}$). In fully connected layers, $o^{(k)}$ would simply be calculated using layer function $\psi^{(FC)}$.

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

The computational complexity of $\psi^{(FC)}$ would become

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

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].

$$ReLU(x) = \begin{cases} x, & \text{if } x \geq 0 \\ 0 & \text{otherwise} \end{cases} \tag{1.3}$$

Therefore we will redefine the fully connected $\psi^{(FC)}$ as;

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

The activation functions 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.

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

And we will describe the whole operation with a function f, such that,

$$f : x \to \hat{y}$$

1.2.3 Loss

To represent the quality of an approximation, we are going to use a loss (or cost) function. A good example 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, loss is never negative. Second, if we compare two approximations, the one with a smaller loss is better.

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,

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

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.

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

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

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

1.2.4 Stochastic Gradient Descent

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 theta 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 use convolutional layers and convolution operation. Please note that, in this chapter, we are assuming one or two dimensional convolutions with same padding.

Let's 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 $\mathcal{W}_{k-1}$ representing the length of the width dimension, $\mathcal{H}_{k-1}$ representing the length of the height dimension and $m^{(k-1)}$ representing number of nodes. Convolution operation first creates a sliding window that goes through width and height 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 a weight matrix $w^{(k)} \in \mathbb{R}^{K \times K \times m^{(k-1)} \times m^{(k)}}$ with every patch, we create a set of output nodes $o^{(k)}_{(I,J)} \in \mathbb{R}^{1 \times m^{(k)}}$. 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.

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

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

$$p_{(I,J)}^{(k-1)} \subset o^{(k-1)}$$

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

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

$$o_{a,b}^{(k-1)} \in \mathbb{R}^{m^{(k)}-1}$$

$$p_{(I,J),(i,j)}^{(k-1)} = o_{a,b}^{(k-1)}$$

where

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

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

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

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

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

The complexity of convolution operation can be described as,

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

1.2.6 Pooling

Pooling is a way of reducing the dimensionality of an output. Depending on the task, one may choose from different pooling methods. Similar to convolution operation, pooling methods also work with patches $p_{(I,J)}^{(k-1)} \in \mathbb{R}^{K \times K \times m^{(k-1)}}$ and strides s_{k-1}. But this time, instead of applying a weight, bias and activation function, they apply functions. The function that we will make use of is $M : \mathbb{R}^{K \times K \times m} \rightarrow \mathbb{R}^m$. By defining different variations of M, we will define *max pooling* and *average pooling*.

Max Pooling

Max pooling takes the maximum value in a channel within the patch.

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

$$\psi_{(k)}^{(maxpool)}(o^{(k-1)}) = o^{(k)} = \{o_{(I,J),i}^{(k)} \mid \forall((I,J),i)(\exists p_{(I,J),i}^{(k-1)})[o_{(I,J),i}^{(k)} = \max(p_{(I,J),i}^{(k-1)})]\}$$

Average Pooling

Average pooling averages the values within the patch per channel.

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

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

So far we have seen the fundamental building blocks of neural networks. Now we move on to subjects that are related to reducing the complexity.

1.2.7 Deconvolution

Introduced by [ZKTF10], deconvolution operation 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.

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.

1.4 Factorization

Factorization is approximating a weight matrix using smaller matrices. Factorization has interesting uses with neural networks. 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,

$$\psi_{(k)}^{(FC)}(o) \approx \psi_{(k)}^{\prime(FC)}(o) = \sigma(o^T U_{w^{(k)}} V_{w^{(k)}} + b^{(k)})$$

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

$$\mathcal{O}(\psi_{(k)}^{\prime(FC)}) = \mathcal{O}(n(m^{(k-1)} + m^{(k)}))$$

One thing that's 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 possibly performing this matrix multiplication many times. Therefore the same technique can be used with convolutional layers. If we apply factorization, the complexity of a convolutional layer would become,

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

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

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

The quality of the approximation will determine the affect of this operation on the accuracy.

1.4.1 SVD

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

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

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 M. 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 $w^{(k)}$.

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

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 n value 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%.

1.4.2 Weight Clustering

Introduced by [NH92], weight sharing starts with regular weight matrices, $w^{(k)} \in \mathbb{R}^{m^{(k-1)} \times m^{(k)}}$ where $k \in [1, \ldots, L]$. Once the weights are learned, they use clustering to find a set of weights, $W' \in \mathbb{R}^{a}$. 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.

1.5 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 reduce that complexity.

1.5.1 Kernel Composition

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 $w^{(k,2)} \in \mathbb{R}^{K \times 1 \times n \times m^{(k)}}$. Their technique, instead of factorizing learned weight matrices, aims to learn the factorized kernels. They also aim to increase non-linearity by adding bias and activation function in between. Therefore defining;

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

This 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 from $\mathcal{O}(KKm^{(k-1)}m^{(k)})$ to $\mathcal{O}(Kn(m^{(k-1)}+m^{(k)}))$. Suggesting that, if we can find an n satisfying $\frac{\mathcal{O}(KKm^{(k-1)}m^{(k)})}{\mathcal{O}(Kn(m^{(k-1)}+m^{(k)})} > 1$, we can reduce the complexity of this layer. This equation can be rewritten as;

$$\frac{Km^{(k-1)}m^{(k)}}{m^{(k-1)}+m^{(k)}} > n$$

1.5.2 Separable Convolutions

Suggested by [Sif14], separable convolutions separate the standard convolution operation into two parts. These parts are called depthwise convolutions and pointwise convolutions. Depthwise convolution applies a given number of filters on every input channel, one by one therefore results with output channels equal to input channel times number of filters. Separable convolutions are used by [Cho16], [HZC+17] and [HZC+17] to reduce complexity of neural networks.

Depthwise Convolution

Given a patch $p^{(k-1)}_{(I,J)} \in \mathbb{R}^{K\times K\times m^{(k-1)}}$, depthwise convolution has a weight matrix $w^{(k,Depthwise)} \in \mathbb{R}^{K\times K\times m^{(k-1)}}$. Let's assume that the subscripts of $p^{(k-1)}_{(I,J)}$ and $w^{(k,Depthwise)}$ are described as $p^{(k-1)}_{(I,J),i} \in \mathbb{R}^{K\times K}$ and $w^{(k,Depthwise)}_i \in \mathbb{R}^{K\times K}$,

The depthwise convolution operation is defined as

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

$$\psi^{(Depthwise)}_{(k)}(o^{(k-1)}) = o^{(k,depthwise)}$$

$$o^{(k,depthwise)} = \{o^{(k,depthwise)}_{I,J,i} \mid \forall(I,J,i)(\exists p^{(k)}_{(I,J),i})[o^{(k,depthwise)}_{I,J,i} = \sum_{a=1}^{K}\sum_{b=1}^{K} p^{(k-1)}_{(I,J),i,a,b} w^{(k,Depthwise)}_{i,a,b}]\}$$

Its complexity can be defined as,

$$\mathcal{O}(\psi^{(Depthwise)}_{k}) = \mathcal{O}(H_kW_kK^2m^{(k-1)})$$

Pointwise Convolution

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

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

Therefore we can describe $\psi_{(k)}^{(Separable)} : \mathbb{R}^{\mathcal{H}_{k-1} \times \mathcal{W}_{k-1} \times m^{(k-1)}} \rightarrow \mathbb{R}^{\mathcal{H}_k \times \mathcal{W}_k \times m^{(k)}}$ as,

$$\psi_{(k)}^{(Separable)}(o^{(k-1)}) = \psi_{(k)}^{(Pointwise)}(\psi_{(k)}^{(Depthwise)}(o^{(k-1)}))$$

The complexity of this operation is,

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

1.6 Pruning

Pruning aims to reduce the model complexity by *deleting* the parameters that has low or no impact in the result. [LDS+89] has shown that using 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, 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 every single weight. Also, [Ree93] compiled many pruning algorithms.

1.6.1 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 layer operations 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% in the weight matrix.

1.6.2 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's 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.

1.7 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.

In a higher level, the computational complexity doesn't depend on number of bits. But if we dive deeper in the computer architecture, using less bits to represent variables provide some major advantages. It takes less cpu-cycles to perform an operation, reduces the cost of transferring data from memory to cpu and finally increases the amount of data that can fit into cache. One major disadvantage is, most architectures implement optimiza-

tions that speed up 16/32/64-bit floating point operations. By using less bits, we are giving up on these optimizations.

1.8 Improving Network Efficiency

1.8.1 Residual Connections

[HZRS15] introduced a method called residual connections. Assuming groups of consecutive layers in out network, we create a residual connection when we add the input of a block to the output it to calculate the input of the next block. Let's 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 don't 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%.

1.8.2 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.

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

1.8.3 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 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 punishes high weight values. We also add a term λ which determines the effect of this regularization. This parameter, if too high would prevent the network from learning, if too low would have no effect.

L1 Regularization

L1 regularization punishes everything other than zero. Therefore it is good to force lots of weights to become very close to zero.

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

L2 Regularization

L2 regularization punishes values with a square term. Therefore larger values are punished more. L2 regularization punishes values greater than one or minus one more than values between.

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

1.9 Efficient Structures

Some structures help neural networks represent more information using less parameters. We are going to look at some structures that are known to work well with convolutional neural networks.

1.9.1 Residual Blocks

[HZRS15] introduced two types of residually connected blocks. First is called a residual block, consisting of two convolution operations and a residual connection between the input and the output of the block. Second one is 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. [HZRS15] 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.

1.10 Datasets

1.10.1 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, best model trained on MNIST achieve almost zero (0.23%, [CMS12]) error rate.

1.10.2 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, best models trained on CIFAR10 achieve 3.47% ([Gra14]) error rate.

1.10.3 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. Best submission from 2016 challenge has achieved 0.02991 error rate. Which is equal to 97.009% top-1 accuracy.

Chapter 2

Methods

So far we have explained some neural network building blocks and some techniques for reduced complexity. We have run some experiments to better understand some of these techniques. In this chapter, we are going to talk about these experiments and tools we had to built to run these experiments. First we identified some methods and to understand them better, we ran experiments on them. Then we have tried to combine these methods to come up with a model.

2.1 Pruning

Since pruning individual weights do not effect complexity directly, we are going to focus on pruning nodes. To do that, as [HPTT16] explained, we are going to use *training cycles*. First, we will define a neural network and a problem. Then we will train this network using the training dataset. After the training is done, we will run inference on the training dataset and collect ReLU output statistics. 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't find any nodes to be pruned.

2.1.1 Fully Connected Summation

To increase our understanding on this method, we started with a very easy problem. We have implemented a neural network consisting of 2 input dimensions ($x_n \in \mathbb{R}^2$), one fully connected layer with 1000 nodes and a 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, ($y_= x_{n,1} + x_{n,2}$). So that 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 mean squared error, 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 MNIST Autoencoder

To expand our pruning experiments to convolutional neural networks, we have implemented an autoencoder for MNIST Dataset. An autoencoder consists of two parts. First is the encoder, which aims to reduce the dimensionality of input. The other is decoder, which aims to convert the encoded data back to it's original form. Therefore an autoencoder reduces the dimensions of an input data and then tries to recreate that data from those reduced dimensions.

We have defined the auto encoder with two encoder layers followed by two decoder layers. Each encoding layer running a convolution with kernel size 3 and stride of 2. Following each of these, we are applying batch normalization and ReLU activation. Each decoding layer is running deconvolutions with kernel size 3 and stride of 2. 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.

The information contained in one input image $x_n \in \mathbb{R}^{28\times28\times1}$ is represented with 784 floating points. Therefore, a good auto-encoder should be reducing this number with each encoding layer. While doing so, it should be able to convert the encoded image back to it's original form with minimal loss while decoding.

Since we will try to find and prune unused nodes, we will start with a

configuration that is larger than necessary.

$$
\begin{aligned}
x_n &\in \mathbb{R}^{28\times28\times1} \\
\psi_{(1)}^{(Conv)} &: \mathbb{R}^{28\times28\times1} \rightarrow \mathbb{R}^{14\times14\times32} \\
\psi_{(2)}^{(Conv)} &: \mathbb{R}^{14\times14\times32} \rightarrow \mathbb{R}^{7\times7\times64} \\
\psi_{(3)}^{(Deconv)} &: \mathbb{R}^{7\times7\times64} \rightarrow \mathbb{R}^{14\times14\times32} \\
\psi_{(4)}^{(Deconv)} &: \mathbb{R}^{14\times14\times32} \rightarrow \mathbb{R}^{28\times28\times1} \\
\mathcal{L} &= RMSE(x_n, \hat{y}_n)
\end{aligned}
$$

2.1.3 Regularization

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

2.1.4 Distortion

In case of two nodes in one layer, if the weights connecting to them are proportional, it may not be possible to prune one of them. To prevent that, we introduce a little bit distortion to weights, forcing nodes representing similar features to have a difference.

2.1.5 Activations

We test 2 pruning configurations for activation counts, activation values. There are many alternatives on choosing the neurons or weighs to prune. But we stick to the two simplest methods.

Activation Counts

We count the activations per node, regardless the activation value. Using this information we determine which nodes are not used. We set a range using the mean and variance of activation counts. We prune the nodes outside this range.

Activation Values

We collect statistics about output values per node. Using this information we determine which nodes are more important for the results by calculating the variance per node and removing the low variance ones regardless of the mean value.

2.2 Convolution Operation Alternatives

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

We have defined a network with three convolutional layers followed by one fully connected layer. Each convolutional layer has kernel size 5. Convolutional layers are followed by batch normalization, then ReLU activations. First two convolution layers are followed by an average 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. We using SCE loss and momentum optimizer (learning rate 10^{-4}, and momentum 0.9), we train this network for 20000 steps with batch size 32.

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

By changing $\psi_{(k)}^{(Conv)}$ with $\psi_{(k)}^{(ConvCompose)}$ and $\psi_{(k)}^{(Separable)}$ we obtain 3 versions of this model. Please remember that $\psi_{(k)}^{(ConvCompose)}$ requires an additional parameter for number of intermediate output channels. We use the number of output channels for that. We train each configuration for both MNIST and CIFAR-10 for a fair comparison.

2.2.1 Non-Linearity in Separable Convolutions

[HZC+17] proposed that, adding batch normalization and ReLU activations between depthwise and pointwise convolutions would increase the model accuracy. To test that we ran the previous experiment comparing 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 dephtwise convolution that multiplies the number

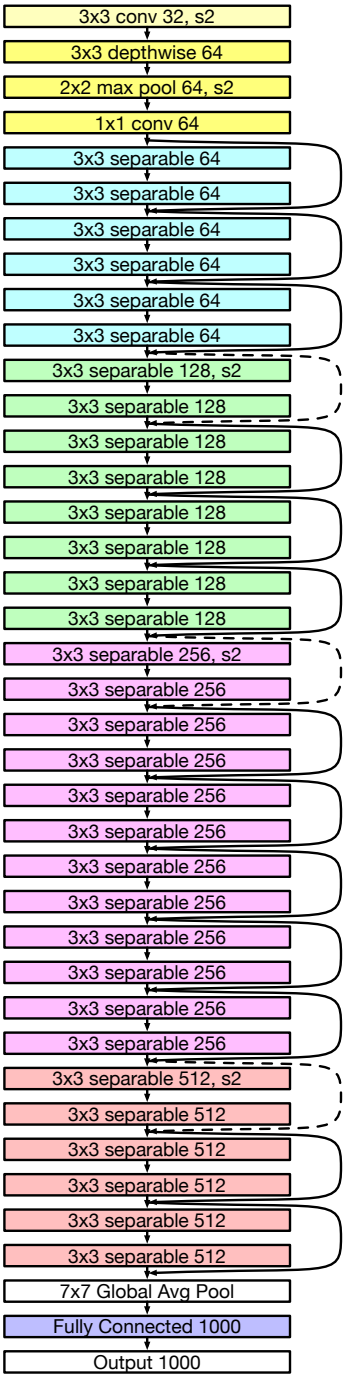


Figure 2.1: Separable Resnet-34. Branching lines represent residual connections, dashed ones are padded with zeros to match the number of channels. If s=2, depthwise convolution is ran with strides of two and residual branch is average pooled with strides 2 and kernel size 2. ReLU and Batch Normalization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

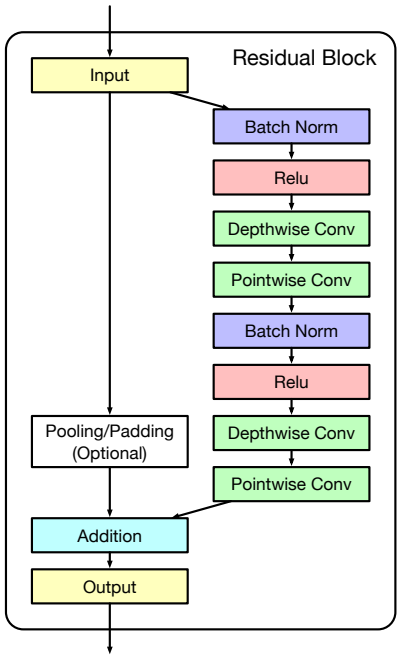


Figure 2.2: Full pre-activation residual connections.

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 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't 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 don't 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

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

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

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 tensorflow 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

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 don't 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.

Pruning with Adam Optimizer

We have trained our network using Adam Optimizer ([KB14]) as well. Adam optimizer keeps two moment matrices for each weight matrix. Therefore, if we are pruning a node, we prune relevant moment matrices accordingly.

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

2.3.4 Factorization

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 achieving a lower rank decomposition.

Rounding Weights

We round the weights after the second decimal. 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 doesn't 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 Summation

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 didn't apply the distortions, after the first training cycle, we couldn't 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 didn't make an obvious difference.

3.1.2 MNIST Autoencoder

In this setting we didn't 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.

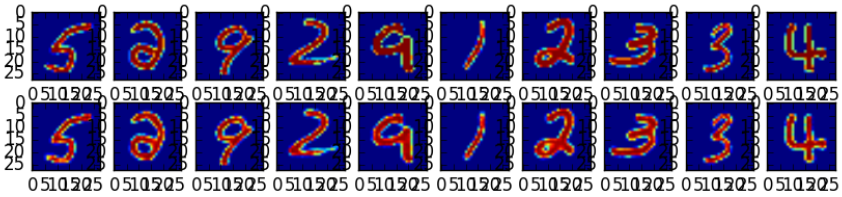


Figure 3.1: Sample results from the validation dataset after pruning

3.2 Convolution Operation Alternatives

By comparing the convolution operation alternatives we try to figure out if separable convolution or kernel composition methods can achieve similar performance to the convolution operation. In our experiments 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 a validation performance comparison for these operations.

3.2.1 Non-Linearity in Separable Convolutions

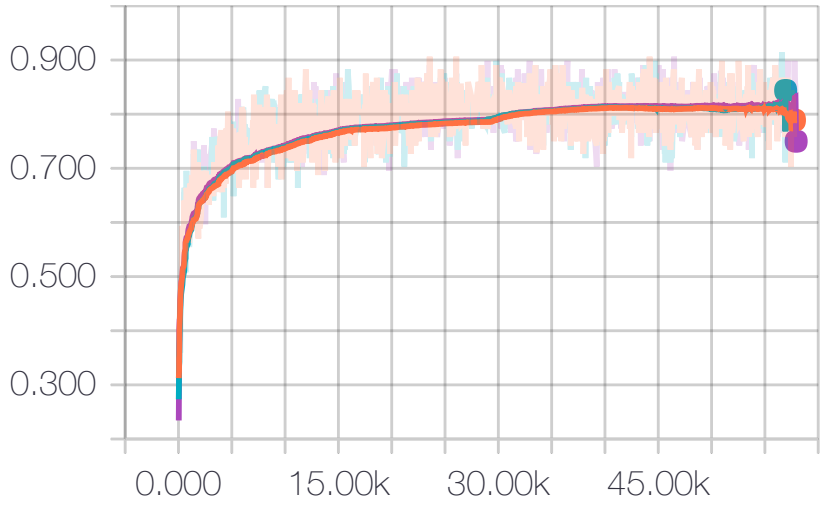
In our experiments we couldn't see any difference between adding non-linear separable convolutions and separable convolutions.

3.2.2 Proposed First Convolutional Layer

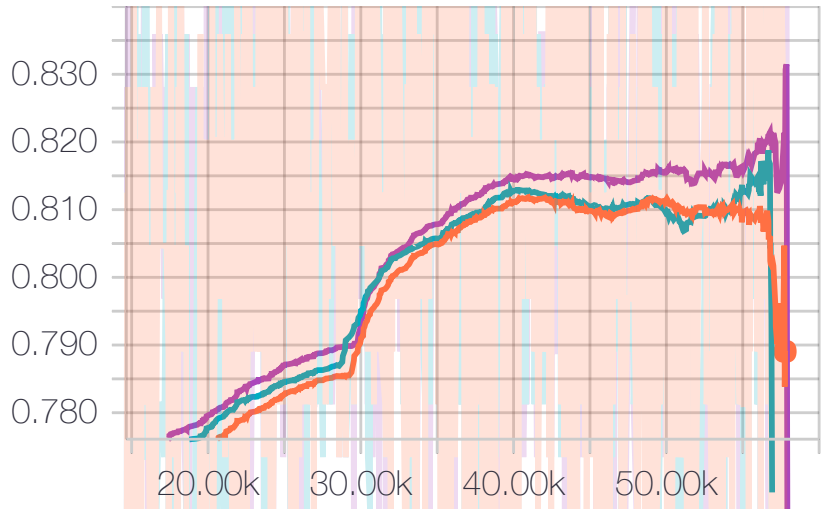
In our experiments we didn't see many differences in terms of accuracy. Even though, our proposed method starts a bit better, both models converge on 85% top-1 accuracy.

3.3 Benchmarks and Comparisons

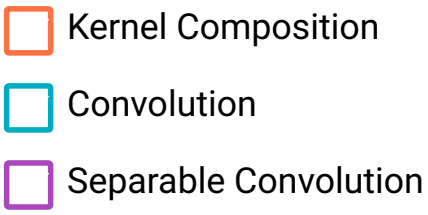
We have benchmarked 20 models in different sizes and shapes with our benchmarking device. Here we will report the key models that made a difference.



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

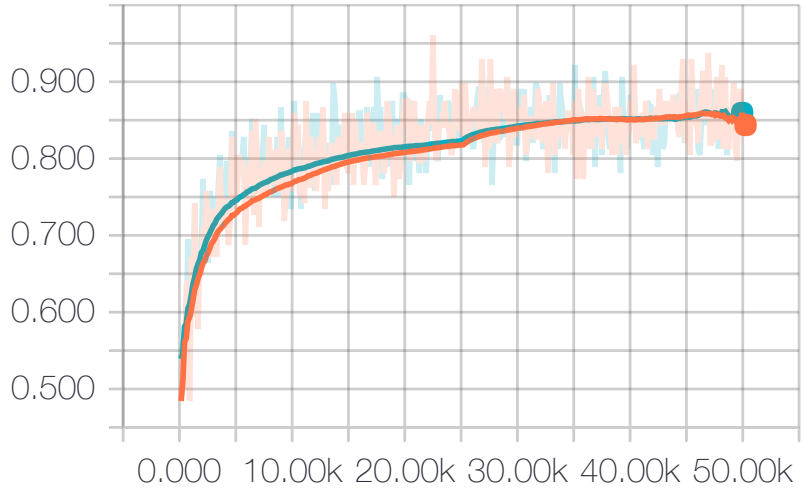


(b) Zoomed in version of Figure 3.2a.

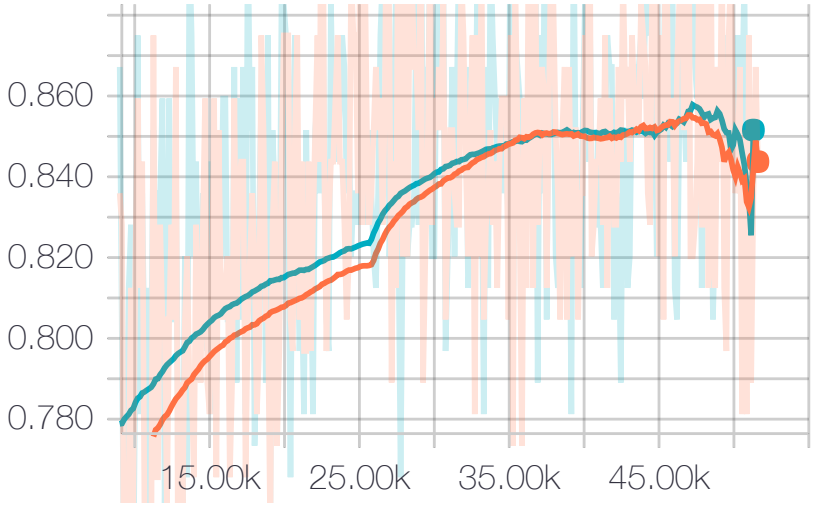


(c) Color codes

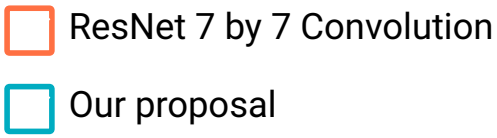
Figure 3.2: Top-1 accuracy comparison of kernel composition, convolution and separable convolution operations. Accuracies for sub-samples of the validation dataset, compared after every training step. We see the thick lines representing smoothed values.



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



(b) Zoomed in version of Figure 3.3a.



(c) Color codes

Figure 3.3: Top-1 accuracy comparison of 7 by 7 convolution and our proposed convolution as the first layer(s) of the model. Accuracies for subsamples of the validation dataset, compared after every training step. We see the thick lines representing smoothed values.

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 comparably as fast as 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 didn't 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.

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 don't worry about implementing lower level operations.
- Most of the operations are highly optimized form any 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. Now 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 extremely difficult to implement operations (e.g. ef operator [AYN+17]) or play around with existing ones because (as of Tensorflow 1.2) there exists no documentation describing C++ internals and build procedures. Since we chose not to go into i

- Tensorflow doesn't 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't use low bit integers to represent these indices, our model size doesn't shrink at all.

4.3 Model Comparison

Comparing models that aim for mobile devices is hard. 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 couldn't 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

Bibliography

[AAB+16] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

[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.

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

[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.

[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.