Master's Thesis

Optimizing Neural Networks for Mobile Devices

Radboud University, Nijmegen

Erdi Çallı

June 10, 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 Neural Networks?
> How do these modifications effect accuracy?

First, we will briefly describe neural networks and some of their underlying building blocks. We will mention the complexities of necessary operations. Then, we will provide solutions to reduce these complexities.

1.1 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. 1.8.2).

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

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.

$$
\begin{aligned}
l_k &: \text{ a column vector of nodes for layer } k \\
m_k &: \text{ the number of nodes in } l_k \\
l_{k,i} &: \text{ node } i \text{ in } l_k \\
o_k &: \text{ the output vector representing the outputs of nodes in } l_k \\
o_{k,i} &: \text{ the output of } l_{k,i} \\
\mathbf{w}^{(k)} &: \text{ weight matrix connecting nodes in } l_{k-1} \text{ to nodes in } l_k \\
w_{i,j}^{(k)} &: \text{ the weight connecting nodes } l_{(k-1),i} \text{ and } l_{k,j} \\
\mathbf{b}^{(k)} &: \text{ the bias term for } l_k \\
\psi_k &: \text{ function to determine } o_k \text{ given } o_{k-1} \\
\sigma &: \text{ activation functions} \\
\mathbf{x} &: \text{ all inputs of the dataset, consisting of } N \text{ data points} \\
\mathbf{y} &: \text{ all outputs of the dataset} \\
\hat{\mathbf{y}} &: \text{ approximation of the output} \\
x_n &: n\text{th input data } (0 < n \leq N) \\
y_n &: n\text{th output data } (0 < n \leq N) \\
\hat{y}_n &: \text{ approximation of } y_n \text{ given } x_n \ (0 < n \leq N) \\
\text{FC} &: \text{ stands for Fully Connected (e.g. } \psi^{(FC)})
\end{aligned}
\tag{1.1}
$$

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

$$
o_k = \begin{cases} \psi(o_{k-1}), & \text{if } k \geq 1 \\ \mathbf{x}, & k = 0 \end{cases} \tag{1.2}
$$

1.1.1 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 $\mathbf{w}^{(k)} \in \mathbb{R}^{m_{k-1} \times m_k}$. Most fully connected layers also include a bias term for every node in l_k ($\mathbf{b}^{(k)} \in \mathbb{R}^{m_k}$). In fully connected layers, o_k would simply be calculated using layer function $\psi^{(FC)}$.

$$\psi_k^{(FC)}(o_{k-1}) = o_{k-1}^T \mathbf{w}^{(k)} + \mathbf{b}^{(k)}$$

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

$$\mathcal{O}(\psi_k^{(FC)}) = \mathcal{O}(m_{k-1} m_k)$$

1.1.2 Activation Function and Nonlinearity

By stacking fully connected layers, we can increase the depth of a neural network. By doing so we want 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 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 $ReLU$.

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

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

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

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 any Neural Network. Stacking $\mathcal{K}$ of them after the input, we can try to approximate an output given an input. To do that we will calculate the outputs of every layer, starting from the input.

$$\mathbf{o} = \{\psi_k(o_{k-1}) | k \in [1, \ldots, \mathcal{K}|]\}$$

1.1.3 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.

A commonly used loss function is Root Mean Square Error (RMSE). Given an approximation ($\hat{y} = o_{\mathcal{K}}$) and the expected output (y), RMSE can be calculated as,

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

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.

If approximation is exactly the same as the output ($\hat{y} = y$), the total loss would be 0.

1.1.4 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 $\theta \in (\mathbf{w} \cup \mathbf{b})$ to be a parameter that we want to optimize. The learning rule updating theta would be;

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

where η is the learning rate, and $\nabla_\theta L(f(x), y)$ 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.

1.1.5 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 (or an at least 2 dimensional data), we can use convolutional layers and convolution operation. Please note that we are assuming one or two dimensional convolutions with same padding.

Let's assume a 3 dimensional layer output $o_{k-1} \in \mathbb{R}^{H_{k-1} \times W_{k-1} \times m_{k-1}}$ where the dimensions W_{k-1} representing the width, H_{k-1} representing height and m_{k-1} representing number of nodes. Convolution operation first creates a sliding window that goes through width and height. The contents of this sliding window would be patches ($p_{k-1,(I,J)} \in \mathbb{R}^{K \times K \times m_{k-1}}$). By multiplying a weight matrix $\mathbf{w}^{(k)} \in \mathbb{R}^{K \times K \times m_{k-1} \times m_k}$ with every patch, we create a set of output nodes $o_{(k+1),(I,J)} \in \mathbb{R}^{1 \times m_k}$. Those output nodes represent the features at location (I, J). By performing this operation for every patch, we calculate the outputs of a convolutional layer. While calculating the patches, we also make use of a parameter called stride, $s_k \in \mathbb{N}^+$. s defines the number of vertical and horizontal indexes between each patch.

$$W_k = \left\lfloor \frac{W_{k-1}}{s_k} \right\rfloor$$

$$H_k = \left\lfloor \frac{H_{k-1}}{s_k} \right\rfloor$$

$$\psi_k^{(Conv)} : \mathbb{R}^{H_{k-1} \times W_{k-1} \times m_{k-1}} \rightarrow \mathbb{R}^{H_k \times W_k \times m_k}$$

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

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

$$p_{k-1,(I,J)} = (p_{k-1,(I,J),(i,j)})$$

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

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

where

$$o_{k-1} = (o_{k-1,(a,b)})$$

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

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

and

$$0 < I \leq H_k$$
$$0 < J \leq W_k$$

$$\psi_k^{(Conv)}(o_{k-1}) = (\sigma(p_{k-1,(I,J)}\mathbf{w}^{(k)} + \mathbf{b}^{(k)}))$$

this definition can be corrected/better.

1.1.6 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 $p_{k-1,(I,J)} \in \mathbb{R}^{K\times K\times m_{k-1}}$ and strides s. But this time, instead of applying a weight, bias and activation function, they apply functions. The function that we will make use of is $M : \mathbb{R}^{K\times K\times m} \to \mathbb{R}^m$. 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.

$$M^{(max)}(p) = \{max(\{p_{i,j,l} \mid i \in [1,\ldots,K], j \in [1,\ldots,K]\}) \mid l \in [1,\ldots,m]\}$$

Average Pooling

Average pooling averages the values within the patch per channel.

$$M^{(avg)}(p) = \{\sum_{i=1}^{K}\sum_{j=1}^{K}\frac{p_{i,j,m}}{K^2} \mid l \in [1,\ldots,m]\}$$

1.2 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 Factorization

Factorization is approximating a weight matrix using smaller matrices. This has interesting uses with Neural Networks. Assume that we have a fully connected layer k. Using factorization, we can approximate $\mathbf{w}^{(k)} \in \mathbb{R}^{m_{k-1} \times m_k}$ using two smaller matrices, $U_{\mathbf{w}^{(k)}} \in \mathbb{R}^{m_{k-1} \times n}$ and $V_{\mathbf{w}^{(k)}} \in \mathbb{R}^{n \times m_k}$. If we can find matrices such that $U_{\mathbf{w}^{(k)}} V_{\mathbf{w}^{(k)}} \approx \mathbf{w}^{(k)}$, we can rewrite,

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

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,

$$\mathcal{O}(\psi_k'^{(FC)}) = \mathcal{O}(n(m_{k-1} + m_k))$$

Therefore, if there is a good enough approximation, satisfying $n < \frac{m_{k-1} m_k}{m_{k-1} + m_k}$, we can reduce the complexity of a fully connected layer without effecting the results. One thing that's similar between a convolutional layer and a fully connected layer is that both are performing matrix multiplication 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.

Luc says: If XW is not exactly equal XOP, how will you deal with this?

1.3.1 SVD

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

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

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 $\mathbf{w}^{(k)}$.

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

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 $U_{\mathbf{w}^{(k)}} = U'S'$ and $V_{\mathbf{w}^{(k)}} = V'^T$, 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%.

1.3.2 Weight Sharing

Introduced by [NH92], weight sharing starts with regular weight matrices, $\mathbf{w}^{(k)} \in \mathbb{R}^{m_{k-1} \times m_k}$ where $k \in [1, \ldots, \mathcal{K}]$. Once the weights are learned, they use clustering to find a set of weights, $\mathbf{W} \in \mathbb{R}^a$. Then they store the cluster index per weight in $d^{(k)} \in \mathbb{N}^{m_{k-1} \times m_k}$. By redefining $\mathbf{w}^{(k)}_{i,j} = \mathbf{W}_{d^{(k)}_{i,j}}$, they perform weight sharing. 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 decomposition.

1.4 Convolution Operation Alternatives

1.4.1 Kernel Composition

As [AP16] explains, a convolution operation with a weight matrix $\mathbf{w}^{(k)} \in \mathbb{R}^{K \times K \times m_{k-1} \times m_k}$, could be composed using two convolution operations with kernels $\mathbf{w}^{(k_1)} \in \mathbb{R}^{1 \times K \times m_{k-1} \times n}$ and $\mathbf{w}^{(k_2)} \in \mathbb{R}^{K \times 1 \times n \times m_k}$. 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;

$$\frac{Km_{k-1}m_k}{m_{k-1}+m_k} > n$$

1.4.2 Separable Convolutions

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.

Depthwise Convolution

Given a patch $p \in \mathbb{R}^{K \times K \times m}$, depthwise convolution has a weight matrix $\mathbf{w}^{(k,Depthwise)} \in \mathbb{R}^{K \times K \times m}$. For easiness, let's assume variants of p and $\mathbf{w}^{(k,Depthwise)}$, described as $p' \in \mathbb{R}^{K \times K}$ and $\mathbf{w}'^{(k,Depthwise)} \in \mathbb{R}^{K \times K}$,

$$\mathbf{w}'^{(k,Depthwise)}_m = \{\mathbf{w}^{(k,Depthwise)}_{i,j,m} \mid i \in [1,\dots,K], j \in [1,\dots,K]\}$$

$$p'_m = \{p_{i,j,m} \mid i \in [1,\dots,K], j \in [1,\dots,K]\}$$

Therefore, depthwise convolution operation $\psi^{(Depthwise)}_k : \mathbb{R}^{W_{k-1} \times H_{k-1} \times m_{k-1}} \to \mathbb{R}^{W_k \times H_k \times m_{k-1}}$ and it's complexity can be defined as,

$$\psi^{(Depthwise)}_k(p) = \{p'_i \mathbf{w}'^{(k,Depthwise)}_i \mid i \in [1,\dots,m]\}$$

$$\mathcal{O}(\psi^{(Depthwise)}_k) = \mathcal{O}(H_k W_k K^2 m_{k-1})$$

Pointwise Convolution

Pointwise convolution ($\psi^{(Pointwise)}_k : \mathbb{R}^{H_k \times W_k \times m_{k-1}} \to \mathbb{R}^{H_k \times W_k \times m_k}$) is a regular convolution operation with kernel size 1 ($K = 1$). The weight matrix that we'll use for this operation is $\mathbf{w}^{(k,Pointwise)} \in \mathbb{R}^{m_{k-1} \times m_k}$. As you can see, $\mathbf{w}^{(k,Pointwise)}$ is the same shape as a fully connected layer weight matrix. The pointwise convolution operation can be defined as,

$$o_{k-1,(I,J)} \in \mathbb{R}^{1 \times m_{k-1}} \text{ where } 0 < I \leq H_k \text{ and } 0 < J \leq W_k$$

$$\psi^{(Pointwise)}_k(o_{k-1}) = \{o_{k-1,(I,J)} \mathbf{w}^{(k,Pointwise)} \mid I \in [1,\dots,H_k], J \in [1,\dots,W_k]\}$$

$$\mathcal{O}(\psi^{(Pointwise)}_k) = \mathcal{O}(H_k W_k m_{k-1} m_k)$$

Therefore we can describe $\psi^{(Separable)}_k : \mathbb{R}^{W_{k-1} \times H_{k-1} \times m_{k-1}} \to \mathbb{R}^{W_k \times H_k \times m_k}$ as,

$$\psi^{(Separable)}_k(o_{k-1}) = \psi^{(Pointwise)}_k(\psi^{(Depthwise)}_k(o_{k-1}))$$

The complexity of this operation is,

$$\begin{aligned}\mathcal{O}(\psi_k^{(Separable)}) =& \mathcal{O}(\psi_k^{(Pointwise)}) + \mathcal{O}(\psi_k^{(Depthwise)}) \\ =& \mathcal{O}(H_k W_k m_{k-1} m_k + H_k W_k K^2 m_{k-1}) \\ =& \mathcal{O}(H_k W_k m_{k-1}(m_k + K^2))\end{aligned}$$

1.5 Pruning

Pruning aims to reduce the model complexity by *deleting* the parameters that has low or no impact 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. After that 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 a weight. 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 $\mathbf{w}^{(k)}$. However, we are defining our layer operations using matrix multiplications. To our knowledge, most of the matrix multiplication implementations require dense matrices. Another option would be to use sparse matrix multiplication. But to be able to gain *any* speed up using sparse matrices, we need to prune about 90% of the weights. Which doesn't seem feasible.

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's assume two fully connected layers, k and $k+1$. The computational complexity of computing the outputs of these two layers would be $\mathcal{O}(\psi_{k+1}^{(FC)}(\psi_{k}^{(FC)}(o_{k-1})) = \mathcal{O}(m_k(m_{k-1} + m_{k+1})$. Assuming that we have removed a single node from layer k, this complexity would drop by $\mathcal{O}(m_{k-1} + m_{k+1})$.

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.

1.6 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.

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

1.7 Improving Network Efficiency

1.7.1 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 of the block 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 for calculating the output. Therefore, their gradients stay do not vanish because they have significant contribution to the result.

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 shape of these parameters would not change.

$$BN(o_k) = \frac{\gamma_k(o_k - \mu_k)}{\sigma_k} + \beta_k$$

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 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

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

L2 Regularization

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

1.8 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.

1.8.1 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. They have 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 the 34 layer network from %28.54 to %25.3.

explain this as you explained residual bottleneck blocks.

1.8.2 Residual Bottleneck Blocks

[HZRS15] have used residual blocks to train networks up to 34 layers. For networks greater than 34 layers, they have introduced a second block called as residual bottleneck block. Residual bottleneck blocks consist of three convolution operations with different kernel sizes and number of output nodes.

Before explaining how residual bottleneck blocks are configured, we need more notations to define the layers inside the same block. Let's assume the block b with input $o_{b-1} \in \mathbb{R}^{H_{b-1} \times W_{b-1} \times m_{b-1}}$. We will index the layers inside the block b with a pair (b, k) where k stands for the index of convolutional layer inside the block. For example if we're talking about the second convolutional layer in block b, the input of this convolutional layer would be

$o_{(b,1)} \in \mathbb{R}^{H_{(b,1)} \times W_{(b,1)} \times m_{(b,1)}}$ and the output would be $o_{(b,2)} \in \mathbb{R}^{H_{(b,2)} \times W_{(b,2)} \times m_{(b,2)}}$. also since we are talking about layers with different kernel sizes, we need to define them with indexes as well. Therefore the kernel size of convolutional layer (b, k) would be $K_{(b,k)} \in \mathbb{R}$. Therefore, we can define the residual bottleneck block as;

$$\psi_b^{(ResidualBottleneck)} : \mathbb{R}^{H_{b-1} \times W_{b-1} \times m_{b-1}} \rightarrow \mathbb{R}^{H_b \times W_b \times m_b}$$

with the helper function S

$$\psi_b^{(ResidualBottleneck)}(o) = S(o) + \psi_{(b,3)}^{(Conv)}(\psi_{(b,2)}^{(Conv)}(\psi_{(b,1)}^{(Conv)}(o)))$$

$$S_b(o) = \begin{cases} o, & \text{if } (W_b, H_b, m_b) = (W_{b-1}, H_{b-1}, m_{b-1}) \\ P_b(M_b^{(avg)}(o)), & \text{otherwise} \end{cases}$$

where P is a padding function that equalizes the number of nodes in the outputs and $M_b^{(avg)}$ is the average pooling function that equalizes the width and height dimensions of the input and output of the block. To their definition the first convolution operation in the block reduces the number of nodes with kernel size $K_{(b,1)} = 1$. The number of nodes is defined with a dependency to the stride of block ($s_b \in \{1, 2\}$) as $m_{(b,1)} = m_b/(4/s_b)$. By that logic, if a residual block is reducing the width and height by half, it is doubling the number of nodes to represent more information. Second convolution operation kernel size $K_{(b,2)} = 3$ and has the same number of nodes as the previous layer $m_{(b,2)} = m_{(b,1)}$. The third convolutional layer quadruples the number of nodes ($m_{(b,3)} = 4m_{(b,1)}$) with kernel size $K_{(b,3)} = 1$.

Their 50-layer network using residual bottleneck blocks achieves %22.85 top-1 error rate on ImageNet dataset.

1.9 Datasets

1.9.1 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.

1.9.2 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 Tools

1.10.1 Tensorflow

To develop and train our neural networks, we are going to be using tensorflow [AAB+16]. Tensorflow provides us the necessary tools to deploy our trained models on mobile devices.

Chapter 2

Methods

2.1 Baseline Models

We used 4 baseline models to test some of the methods we have described. These models helped us to assess the efficiency of some methods that we have covered.

2.1.1 Fully Connected Summation

We have implemented a neural network consisting of 2 inputs, $\mathbf{i} = (i_1, i_2)$, 1 fully connected layer with $n = 1000$ hidden nodes and 1 output, o. We have used ReLU [NH10] activations on our hidden layer. For the sake of simplicity, we have defined the expected output y as $y = i_1 + i_2$. We chose a simple problem so that we precisely know the most optimum neural network structure that would be able to perform this calculation. Which is the same network where the fully connected layer has one hidden node, all weights equal to 1 and all biases equal to 0.

We have calculated the loss using mean squared error, and optimized it using Momentum Optimizer (learning rate 0.01 and momentum 0.3). Using 1.000.000 samples, we trained the network with batch size 1000. With these parameters, we ran a training session with 10 epochs and we have observed that the loss didn't converge to 0. Therefore, the model was unable to find the correct solution with this optimizer.

Block Name	Output Dimensions ($h \times w \times c$)
Input Image	$28 \times 28 \times 1$
Encoder 1	$14 \times 14 \times 4$
Encoder 2	$7 \times 7 \times 16$
Decoder 1	$14 \times 14 \times 4$
Decoder 2	$28 \times 28 \times 1$

Table 2.1: The baseline network that could perform lossless encoding by only shifting values.

2.1.2 MNIST auto-encoder

we have implemented an auto encoder for MNIST Dataset [LCB98]. MNIST contains 28×28 grayscale images of handwritten digits. The autoencoder consists of two parts. First part is the encoder. The encoder aims to reduce the dimensionality of input. The decoder aims to convert the encoded data back to it's original form.

We have defined the auto encoder with two encoder blocks followed by two decoder blocks. Each encoding block is running convolutions with kernel size 3, strides of 2 and $SAME$ padding. Then we are adding bias to this result, following this we are applying batch normalization [IS15] and then ReLU activation [NH10]. Each decoding block is running deconvolutions with kernel size 3 and strides of 2. Followed by adding bias, batch normalization and ReLU activations.

The information contained in one $28 \times 28 \times 1$ matrix is represented with 784 units (floating points in this case). Therefore, a good auto-encoder should be capable of reducing this number when encoding. Similarly, converting the reduced matrix back to it's original form with minimal loss while decoding. The baseline auto-encoder we will compare our results is the non-encoding one given in table 2.1.

In our case, our encoder blocks are reducing the matrix width and height to half. Therefore, if they output 4 times the number of input channels, they should represent the same information losslessly. Similarly our decoder blocks are doubling the matrix width and height. Therefore if they output a quarter of the number of input channels, they should be able to decode the encoded information perfectly. In Table 2.1 we have defined the layer output dimensions for that baseline auto-encoder.

To define the network to experiment on, we chose $[32, 64, 32]$ as the output channels of Encoder 1, Encoder 2 and Decoder 1 respectively.

This definition may not be good. check it.

2.1.3 MNIST Classifier

Our MNIST Classifier is consisting of three Convolutional Layers and one Fully Connected layer. This configuration is defined in Table 2.2. Please note that the third convolution is a 7×7 convolution applied to a 7×7 input, with $VALID$ padding. Therefore, this layer is working as a fully connected layer. We keep it as a convolution operation to be able to experiment with convolution operations.

While initializing the weights, we use a truncated normal distribution with 0 mean and 0.1 standard deviation. We calculate the loss by the cross entropy between labels and logits. To train this network, we use RMSProp Optimizer with learning rate 10^{-4}. We train this network for 20000 steps with batch size 50. To express the performance of this model, we use the accuracy score on the test dataset.

what is the accuracy of baseline model
change the standard deviation of weight distribution to 0.1, remove biases and redo the experiments by increasing the learning rate and momentum.

2.1.4 CIFAR-10 Classifier

Our CIFAR-10 baseline model is defined in Table 2.3. The input is distorted as defined in the table.

While initializing the weights, we use a truncated normal distribution with 0 mean and 0.1 standard deviation. We calculate the loss by the cross entropy between labels and logits. To train this network we use Adam Optimizer with default parameters. We train this network for N steps with batch size M. To express the performance of this model, we use the accuracy score on the test dataset.

In this model we are using Adam Optimizer to do the training with default parameters. The loss is the cross entropy between logits and labels.

Layer	Configuration	Output
Input Image		$28 \times 28 \times 1$
Convolution	$N = 5, \text{strides} = 1, \text{padding} = SAME$	$28 \times 28 \times 32$
Add Bias		$28 \times 28 \times 32$
ReLU		$28 \times 28 \times 32$
Max Pool	size=2×2	$14 \times 14 \times 32$
Convolution	$N = 5, \text{strides} = 1, \text{padding} = SAME$	$14 \times 14 \times 64$
Add Bias		$14 \times 14 \times 64$
ReLU		$14 \times 14 \times 64$
Max Pool	size=2×2	$7 \times 7 \times 64$
Convolution	$N = 7, \text{strides} = 1, \text{padding} = VALID$	1×128
Add Bias		1×128
ReLU		1×128
FC Layer		1×10
Add Bias		1×10

Table 2.2: Network configuration for MNIST Classifier, output of every row is applied as the input of next.

remove bias, add batch normalization properly and redo your experiments.

2.2 Pruning Experiments

2.2.1 Fully Connected Layers

To gain more insight on activation based pruning, we ran some experiments with Fully Connected Summation model.

Vanilla Pruning

First we have implemented the very basic idea of pruning unused activations. To do so, we defined training cycles based on the method defined in [HPTT16]. In each training cycle, 1) we have trained the model for some epochs, 2) we lock the weights, 3) feed the training data to the network and count the activations for each neuron in the hidden layer, 4) prune the neurons that have less than or equal to the activation threshold, 5) go back to

Layer	Configuration	Output
Input Image		$32 \times 32 \times 3$
Random Crop		$24 \times 24 \times 3$
Random Flip	left to right	$24 \times 24 \times 3$
Random Brightness	$[-63, 63]$	$24 \times 24 \times 3$
Random Contrast	$[0.2, 1.8]$	$24 \times 24 \times 3$
Normalization	`per_image_standardization`	$24 \times 24 \times 3$
Convolution	$N = 5, \text{strides} = 1, \text{padding} = SAME$	$24 \times 24 \times 64$
Add Bias		$24 \times 24 \times 64$
Batch Normalization		$24 \times 24 \times 64$
ReLU		$24 \times 24 \times 64$
Max Pool	$N = 3, \text{strides} = 2, \text{padding} = SAME$	$12 \times 12 \times 64$
Convolution	$N = 5, \text{strides} = 1, \text{padding} = SAME$	$12 \times 12 \times 64$
Add Bias		$12 \times 12 \times 64$
Batch Normalization		$12 \times 12 \times 64$
ReLU		$12 \times 12 \times 64$
Max Pool	$N = 3, \text{strides} = 2, \text{padding} = SAME$	$6 \times 6 \times 64$
FC Layer	$(6 * 6 * 64) \times 384$	1×384
Add Bias		1×384
Batch Normalization		1×384
ReLU		1×384
FC Layer	384×192	1×192
Add Bias		1×192
Batch Normalization		1×192
ReLU		1×192
FC Layer	192×10	1×10
Add Bias		1×10

Table 2.3: Network configuration for CIFAR-10 Classifier, output of every row is applied as the input of next.

step 1 if some neurons were pruned, stop otherwise.

When tested with 0 activation threshold, after the first training cycle, this method did not to prune anymore weights. In our experiments, we have pruned approximately 950 weights out of 1000. This result is promising but at the same time, it's not close enough to the result we were expecting. We delved deeper into the source of this issue.

We should try different optimizers and make the beginning of the case about why we decided to distort weights.Tell that we have checked the gradients and seen that they were mostly in one direction (+).

Distorted Pruning

When we inspected the gradients of weights, we have seen that most of them were in the positive direction. In our case, this trend in gradients is not helping with the understanding of which neurons are necessary, and which are not. This trend can also be understood as, the feature representation is shared among different hidden neurons.

talk about what does "all gradients are in the positive direction" mean for feature representation

To prevent shared feature representation, we have decided to distort the weights using random values. This allowed some weights to become unused, therefore getting closer to the optimum result.

the results were in a form not resembling the real solution. maybe because floating point numbers not adding up perfectly, but the result is almost the same in terms of our loss. the exact values of weights and biases are:

```
w1: [[ 0.74285096], [ 0.64994317]]
b1: [ 7.80925274]
w2: [[-6.75151157]]
b2: [ 7.80925274]
```

So since our random values are between -1 and 1, these values are actually okay.

talk about how you decide on the amount of distortion (currently $rand(weights.shape) * (1 - var(weights))$). Talk about what changed when we introduced

Regularized Distorted Pruning

Since the solution we found is only resembling our result under some boundaries, we have decided to add an l1 regularizer to our loss. By doing so we are aiming to push the high bias and w2 values closer to 0. But it doesn't really make any difference when used with Moment Optimizer.

2.2.2 Convolutional Layers

To verify the validity of this method, we ran experiments with MNIST autoencoder.

Activation Based Pruning

As we did in the Fully Connected Layers, we have pruned the connections that are not being activated. In these experiments we have seen that the network has been pruned insignificantly. After applying this method, we have achieved a network consisting of $[16, 64, 22]$ output channels for blocks Encoder 1, Encoder 2 and Decoder 1 respectively.

Applying Distirtions

test if this actually changes anything.

we can also check activation probabilities and make a decisions based on this data

Applying Regularizers

explain why you chose this regularizer, tell the effect of using it

Pruning Outliers

explain why you decided to prune the "outliers" from activations, how you decide on what are outliers ($mean - 2 * std$) and how this effects the final solution

Regularized and Distorted Outilier Pruning

explain your results when you combined these methods.

2.3 Efficient Operations

2.3.1 1-D Convolutions

Chapter 3

Results

Chapter 4

Discussion

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.

[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 *Ad-*

vances 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.

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

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

[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.

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

[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.

[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.