Master's Thesis

Optimizing Neural Networks for Mobile Devices

Radboud University, Nijmegen

Erdi Çallı

June 7, 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.

1.1 Neural Networks

In this chapter, we will try to describe neural networks briefly. To keep things simple, we only are concerned with feed forward neural networks and supervised learning. 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 networks 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

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

denote the concepts that we just explained.

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

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} and l_k, with shapes $m_{k-1} \times 1$ and $m_k \times 1$, respectively. For these layers to be fully connected, the weight matrix $\mathbf{w}^{(k)}$, should be of shape $m_{k-1} \times m_k$. Most fully connected layers also include a bias term (m) for every node l_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

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

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;

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

$\psi^{(FC)}$ is one of the most basic building blocks of any Neural Network. Stacking 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.

$$\begin{aligned} o_0 &= x_n \\ o_1 &= \psi_1^{(FC)}(o_0) \\ o_2 &= \psi_2^{(FC)}(o_1) \\ &\ldots \\ o_K &= \psi_1^{(FC)}(o_{K-1}) \\ \hat{y} &= o_K \end{aligned}$$

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 ($\hat{y}$) and the corresponding 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 all our approximations are exactly the same as the output ($\hat{\mathbf{y}} = \mathbf{y}$), 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, we will use convolutional layers or convolution operation. Let's assume an image $i \in \mathcal{R}^{W \times H \times C}$. Convolution operation first creates a sliding window that goes through the image. We will call the contents of this sliding window with centered (i,j) as a patch ($p_{(i,j)} \in \mathbb{R}^{K \times K \times C}$). Convolution operation sees every element of this patch as a node. By multiplying a weight matrix $\mathbf{w}^{(k)} \in \mathbb{R}^{K \times K \times C \times O}$ to this patch of nodes, it creates a set of output nodes $o_{(k+1),(i,j)} \in \mathbb{R}^{1 \times O}$. Those output nodes represent the features belonging to the pixel at point (i,j). By performing this operation for every patch, we calculate the outputs of a convolutional layer.

A convolutional layer with kernel size 3 would combine the data from all neighboring pixels while calculating the output information. Therefore it's outputs can represent shapes. By adding another convolutional layer, we would combine information of neighboring shapes, therefore represent more complex shape information. By adding more convolutional layers, we can represent more and more complex shapes, or objects, or stuffs. Such networks are called convolutional neural networks (CNN).

The complexity of the matrix multiplication on a patch is $(KKIO)$. Complexity of applying this operation to every patch is $\mathcal{O}(WHK^2CO)$.

shall we provide more details?

1.1.6 Pooling

Pooling is a way of reducing the dimensionality of an image. Depending on the task, one may use from different pooling methods. Pooling methods create patches of size k that are distant from each other by s pixels in horizontally and vertically. s is called the stride of pooling and k is referred as the kernel size. For example, pooling an image with $W \times H \times C$ dimensions with strides of 2 would result with an image of dimensions $W/2 \times H/2 \times C$. Pooling methods do not change the amount of channels the input has. But they choose values for the channels based on the pooling method.

Average pooling averages the values within the patch per channel. Max pooling takes the maximum value in a channel within the patch. Global Average Pooling takes the whole image as 1 patch and applies method max/average pooling on that 1 patch. Therefore returns only a matrix of dimensions $1 \times 1 \times C$.

1.2 Efficient Operations

1.2.1 Separable Convolutions

As the name suggests, these operations 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. Pointwise convolution correlates these output channels with each other, or in other words mixes them, by applying a matrix multiplication. For example, let's assume we have an input with 3 channels, applying a Depthwise convolution with 4 filters to that, we would get 12 output channels. Then applying a Pointwise convolution to that, we correlate these 12 output channels to create new output channels.

To describe the complexity of this operation, let's assume that we have a separable convolution with kernel size N, input channels K, depthwise filters I and output channels L. First operation will be applying L filters with size $N \times N$ to K input channels, one by one. The number of operations we need for this operation is, IKN^2. Second operation will be multiplying $1 \times IK$ output with $IK \times L$ correlation matrix, requiring IKL floating point operations. In total we need $IK(N^2 + L)$ operations.

1.2.2 1D Convolutions

Convolution kernels could be decomposed into smaller operations. For example using SVD, we can decompose the convolution kernel into the multiplication for two smaller kernels, and by applying these kernels consequently, we can approximate the convolution operation with fewer floating point operations. Instead of doing separating learned convolutions, we are going to use the method introduced in [AP16]. This method forces the separability of convolution operation as a hard constraint.

Normally convolution operations are defined 2 dimensional ($2D$). That is, kernel sizes are ($2D$). For example convolutions we have used in Section

1.3.2 are $2D$, their kernel sizes are 3×3. With this method, we are aiming to construct a $N \times N$ convolution operation as a combination two convolution operations. The first convolution has kernel size $1 \times N$ and the following convolution has $N \times 1$, or vice versa. Looking back at the experiment we did in Section 1.3.2, instead of applying one 3×3 convolution, we are talking about applying one 1×3 convolution followed by a 3×1 convolution.

The amount of speed up can be approximated using the number of floating point operations. A convolution operation can be seen as a matrix multiplication applied to consequent subsections of an image. To visualize this, let's assume that we have a convolution operation with kernel size N, input channels K and output channels L. Assuming we are applying that operation to a $N \times N$ image patch with K channels, our operation can be simplified as multiplying this $N * N * K$ vector with $N * N * K \times L$ matrix. The number of floating point operations required to execute this operation are N^2KL. When composed as two $1D$ compositions, this operation will be represented with two $1D$ convolutions. First convolution is multiplying vector $N*K$ with matrix $N * K \times P$ and the second convolution is multiplying vector $N * P$ with matrix $N * P \times L$. As you can see we have introduced the variable P as the intermediate output of the first convolution. Number of floating point operations required for these operations are, NLP and NPK. And summing them up, we can say we need $NP(L + K)$ operations in total. To see if this decomposition is reducing the number of operations, we could basically try to define a P satisfying $\frac{NP(L+K)}{N^2KL} < 1$. Therefore,

$$1 \leq P < \frac{NKL}{L + K}$$

To show the validity of this method, we have conducted two experiments.

1.3 Pruning

Pruning aims to reduce the number of operations by deleting the parameters that has low or no impact in the result. Studies show that applying this method in an ANN is effective in reducing the model complexity, improving generalization, and they are effective in reducing the required training cycles. In our experiments we will try to reproduce these effects. To visualize these methods, and help with the explanation later, let's think of two fully connected layers, $\mathbf{l_1}$ and $\mathbf{l_2}$. $\mathbf{l_1}$ is the input of this operation and it consists of

N values, $\mathbf{l_1} = (l_{11}, l_{12}, ..., l_{1N})$. $\mathbf{l_2}$ is the output of this operation consists of M values, $\mathbf{l_2} = (l_{21}, l_{22}, ..., l_{2M})$. Between these two layers, there is a weight matrix W with size $N \times M$. The operation, that we want to optimize is, $\mathbf{l_2} = \mathbf{l_1}W$. To do so, we will look at 2 cases of pruning. One will be focusing on pruning individual weights, and the other will be focusing on removing unimportant rows and columns from $\mathbf{l_2}$, $\mathbf{l_1}$ and W.

maybe explain in more detail and give examples of pruning algorithms here. (e.g. Optimal Brain Damage, Second order derivatives for network pruning: Optimal Brain Surgeon, Optimal Brain Surgeon and general network pruning, SEE Pruning Algorithms-a survey from R. Reed)

1.3.1 Pruning Weights

This subcategory of pruning algorithms try to optimize the number of floating point operations by removing some individual values from W. Theoretically, it could benefit the computational complexity to remove individual scalars from W, by not performing operations related to those weights. But practically, in Tensorflow, matrix multiplication on dense matrices uses all of the values of it's inputs. In contrary, sparse matrix multiplication takes a sparse matrix and a dense matrix as inputs and outputs a dense matrix. To implement this method, we could convert W to a sparse tensor after pruning the weights. But, Tensorflow documentations explicitly state;

- Will the SparseTensor A fit in memory if densified?
- Is the column count of the product large ($>> 1$)?
- Is the density of A larger than approximately 15%?

"If the answer to several of these questions is yes, consider converting the SparseTensor to a dense one."

In our terms, SparseTensor A is corresponding to the pruned version of W. Since W was already dense before, we can assume that the answer to the first question is yes. The column count of our product is M which is much larger than 1 in some cases. Also we don't know anything about the density of pruned version of W. Looking at these facts, we are assuming that implementing this operation will be problematic. Instead of delving deeper into these problems to evaluate this method, we will move on to other methods.

add figure to show what happens when we prune

1.3.2 Activation Based Pruning - Fully Connected Layers

Activation based pruning, works by looking at individual values in layers, and prunes the layer and corresponding weight row/columns completely. To visualize this, we will assume that the fully connected layers we have defined are, trained to some extent, and activated using ReLU activations. With this definition, if we apply our dataset and count the number of activations in $\mathbf{l_1}$ and $\mathbf{l_2}$, we may realize that there are some neurons that are not being activated at all. By removing these neurons from the layers, we can reduce the number of operations. This removal operation is done by removing neurons based on their activations.

add figure to show what happens when we prune

1.3.3 Activation Based Pruning - Convolution and Deconvolutions

Put references for conv and deconv operations.

In theory, convolution operation is a matrix multiplication applied on a sliding window. Thus, counting the output feature activations of a convolution operation, we can apply activation based pruning.

1.3.4 Second Order Derivatives (Fischer Information Matrix)

1.4 Factorization

Using Factorization methods, we can decompose a matrix into smaller different matrices. Some factorization methods can be used to reduce the dimensionality of these smaller matrices while approximating the original matrix. This has interesting uses with Neural Networks. Assume that we have a matrix multiplication operation. We are multiplying a random input $\mathbf{X}$ with a fixed weight matrix $\mathbf{W}$. Let's say $\mathbf{X}$ is $K \times N$ and $\mathbf{W}$ is $N \times M$. The matrix multiplication of these two matrices has the complexity of KNM. If we can successfully decompose $\mathbf{W}$ to the composition of two matrices $\mathbf{O}$ and $\mathbf{P}$ with dimensions $N \times L$ and $L \times M$, respectively, we can rewrite our matrix multiplication operation as; $\mathbf{XW} \approx \mathbf{XOP}$. The new complexity of this operation would be $KNL + NLM = NL(K + M)$. If we can find a decomposition where L is sufficiently small that successfully approximates the matrix $\mathbf{W}$ and satisfies $NL(K + M) < KNM$, we can reduce the complexity of this matrix multiplication.

draw some figures explaining how this happens

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

1.4.1 SVD

Using SVD we can make this approximation. SVD decomposes the a matrix into 3 parts.

Talk what SVD does, what are the decomposed matrices, what are the singular values and how they are relevant to the approximation of $\mathbf{W}$. Show your experiments and results from when you ran the experiments.

1.4.2 Weight Sharing

Weight sharing assumes we have a limited set of weights, and when we are representing values, instead of representing the value itself, we represent the indices to weights. This operation is used in some papers successfully to reduce model size considerably. To be able to implement such a representation, we can use 2 potential functions of Tensorflow. One is `tf.gather` which selects the given indices from a given matrix. The other is `tf.embedding_lookup` which works with a bucket of values and returns the given keys/indices from the given bucket. However, both methods accept the indices as a matrix of 32-bit or 64-bit integers. Therefore storing indices instead of weights would not yield with any improvements in the model size. Without an implementation of these methods using low-bit integers, it is not possible to exploit their usefulness.

give some examples here, you're saying some papers.

1.4.3 Other Factorization Methods

1.5 Quantization

1.5.1 8-bit Quantization

1.5.2 n-bit Quantization

1.6 Efficient Structures

Some structures help neural networks represent more information using less parameters.

talk more about why some structures are more efficient, how they help with training speed, how they reduce the number parameters or number of floating point operations even while increasing the accuracy.

1.6.1 Inception Blocks

do the introduction to [SLJ+14] and how it is improved using [SVI+16]

1.6.2 Bottleneck Blocks

[HZRS15] introduced residual connections with bottleneck blocks. To optimize the performance of their network, they have introduced the bottleneck blocks. Bottleneck blocks contain 3 convolutions. First is a 1×1 convolution that scales down the input channels to half. Output is applied to a 3×3 convolution which doesn't change the number of channels, and following that with a 1×1 convolution to quadruple the number of input channels. As an example, we can look at the conv3_x block of 50-layer network described in Table 1.1.

try to reason why bottleneck blocks work. Luc says: what is the reasoning of this? why would one want to do this?

[HZRS15] compared the performance of various network configurations on ImageNet validation dataset. From these comparisons, we have selected the 34-layer network and the 54-layer network. The 34-layer network is consisting of pairs of 3×3 blocks. The 50-layer network is consisting of bottleneck

layer name	output size	34-layer	50-layer
input image	224×224		
conv1	112×112	7×7, 64, stride 2	
conv2_x	56×56	3×3 max pool, stride 2	
		$\begin{bmatrix} 3 \times 3, & 64 \\ 3 \times 3, & 64 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, & 64 \\ 3 \times 3, & 64 \\ 1 \times 1, & 256 \end{bmatrix} \times 3$
conv3_x	28×28	$\begin{bmatrix} 3 \times 3, & 128 \\ 3 \times 3, & 128 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, & 128 \\ 3 \times 3, & 128 \\ 1 \times 1, & 512 \end{bmatrix} \times 3$
conv4_x	14×14	$\begin{bmatrix} 3 \times 3, & 256 \\ 3 \times 3, & 256 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, & 256 \\ 3 \times 3, & 256 \\ 1 \times 1, & 1024 \end{bmatrix} \times 3$
conv5_x	7×7	$\begin{bmatrix} 3 \times 3, & 512 \\ 3 \times 3, & 512 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, & 512 \\ 3 \times 3, & 512 \\ 1 \times 1, & 2048 \end{bmatrix} \times 3$
	1×1	average pool, 1000-d fc, softmax	
FLOPs		3.6×10^9	3.8×10^9
top-1 error (%)		21.53	20.74
top-5 error (%)		5.60	5.25
top-1 error (%, **10-crop** testing)		24.19	22.85
top-5 error (%, **10-crop** testing)		7.40	6.71

Table 1.1: Comparison of bottleneck blocks (50-layer) with stacked 3×3 layers (34-layer).

blocks. In table 1.1 we have compared these networks by their structure, required number of FLOPs, and their top-1 and top-5 errors on this dataset. As we can see in the FLOPs, the networks have about 5% of difference in number of floating point operations. As [HZRS15] reports, this small increase in parameters is effecting accuracy of the model considerably.

But the main contribution of [HZRS15] is not the bottleneck architecture, but Residual Connections that we will see in another section.

1.7 Improving Network Efficiency

1.7.1 Residual Connections

[HZRS15] is also introducing a method called Residual Connections.

go into details of how this works using information given in [HZRS15] and [SIV16]

1.7.2 Batch Normalization

$$\begin{aligned} V &\approx HW \\ IV &\approx IHW \\ I &\in \mathbb{R}^{l \times m} \end{aligned} \tag{1.4}$$

1.8 Datasets

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 in theory.

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 auto-encoder.

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.

[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.

[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.

[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.

[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.

[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.