

Optimizing Neural Networks for Mobile Devices

Radboud University, Nijmegen

Erdi Çallı

April 5, 2017

Abstract

Recent developments in Neural Networks(or Deep Learning) are promising. Some models are capable of accomplishing tasks as good as humans, or better. But we still lack the applications that are available to the public. The general opinion is, Neural Network models need expensive equipment. But that only applies to the training process, where the network learns from data. But using the trained model for inference is easier. There are also methods to reduce the computational complexity of a model. With these methods, we may be able to use cheap compute devices(i.e. Mobile Phones) for inference. Thus making some models available to public.

Contents

1	Introduction	3
1.1	Recent Studies	4
2	Methods	5
2.1	Dependencies	5
2.1.1	Tensorflow	5
2.2	Pruning	5
2.2.1	Pruning Individual Weights	6
2.2.2	Activation Based Pruning - Fully Connected Layers . .	7
2.2.3	Activation Based Pruning - Convolution and Deconvolutions	9
2.3	Efficient Structures	11
2.3.1	1D Convolutions	11
2.3.2	Bottleneck Architecture	12
3	Results	14
4	Discussion	15
5	Conclusion	16

Chapter 1

Introduction

Recent state of the art Deep Learning models are surpassing previous methods. Fields such as; computer vision, automatic speech recognition, natural language processing, speech recognition, and bioinformatics make use of these models. They use deep models consisting of many layers (e.g. 152 layers in [HZRS15]), many parameters in each layer, and as a result, a lot of Floating Point Operations to run Inference (e.g. 11.3×10^9 in [HZRS15]). In contrast, mobile devices have limited processing power and memory. Also the best practice is to provide a fluent user experience with low response time. Thus, we should change these models to provide a good user experience. There is research on methods to define optimized models or optimize a given model. These methods consist; pruning unimportant parameters, using less bits to represent parameters, or using less parameters by using more optimized structures.

In this research we are going run experiments to answer;

1. Which models are running slow in Mobile Devices?
2. Why these models are running slow?
3. Which methods can we use to optimize these models?
4. What is the trade off of using these methods?
5. Why an optimization technique is working or not on a model?
6. Can we define a more optimized model for the same task?
7. How can we combine different optimization techniques?

8. Are these optimized models efficient enough to run in Mobile Devices?

1.1 Recent Studies

Artificial Neural Networks (ANN) have several parameters such as number of hidden layers, number of neurons in a layer, or the structure of a layer. Until now, we have seen different combinations for these parameters. For example, [SZ14] introduces a model called VGGNet. VGGNet introduces more layers (16 to 19) than the previous models. They show how this parameter effects the accuracy. [HZRS15] introduces the residual connections. This new connection between layers is capable of stacking more layers than before. Training up to 152 layers, they show superior accuracy. [ZK16] compares having higher number of neurons in each layer to having more layers. Each combination resulting in a unique model with a different accuracy level. In contrast to all these, [SLJ⁺14] suggests something different. Having a good harmony within the network works better than having more parameters. Supporting that, [CPC16] does a detailed comparison of different models. They show that, increasing the number of hidden layers or the number of neurons in a layer does not necessarily increase the accuracy.

Following these, we think that, some models are over-parameterized. Meaning they contain parameters that they are not making use of. Therefore, they are making unnecessary computations with them.

Chapter 2

Methods

2.1 Dependencies

2.1.1 Tensorflow

In our research, we will strictly use Tensorflow [AAB⁺16].

What is tensorflow, why we chose it, what are the advantages of using it, what are the limitations that come with it

things to be explained: FC layer, Convolution Operation, what do we mean when we say neuron/node, activation, bias, training dataset

2.2 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, 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, $\mathbf{l}_1 = (l_{11}, l_{12}, \dots, l_{1N})$. \mathbf{l}_2 is the output of this operation consists of M values, $\mathbf{l}_2 = (l_{21}, l_{22}, \dots, l_{2M})$. Between these two layers, there is a weight matrix W with size $N \times M$. The operation, that we want to optimize is, $\mathbf{l}_2 = \mathbf{l}_1 W$.

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)

2.2.1 Pruning Individual Weights

With this subcategory of pruning algorithms we want to optimize the number of floating point operations by removing some values from W . Theoretically, it makes sense to remove individual scalars from W , and exclude operations related to them. This would ideally reduce the required number of floating point operations. But in our library of our choice, Tensorflow, matrix multiplication implementation `tf.matmul` do not consider such a change. It takes two fixed size matrices, and does the computations using all of their values. Tensorflow also has another matrix multiplication operation, `tf.sparse_tensor_dense_matmul`. This operation 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 about this method state;

- Will the SparseTensor A fit in memory if densified?
- Is the column count of the product large ($\gg 1$)?
- 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.

2.2.2 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.

Experiment Set-up

To verify the validity of this method, we set up a basic experiment. 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.

We should also try different optimizers here

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 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 more 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 push the weights to different directions, randomly. This resulted with the result closes to optimum, but not the exact solution that we were looking for.

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 $\text{rand}(\text{weights.shape}) * (1 - \text{var}(\text{weights}))$). 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.

2.2.3 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.

Experiment Set-up

To verify the validity of this method, 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 its 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 autoencoder should be capable of reducing this number when encoding. Similarly, converting the reduced matrix back to its original form with minimal loss while decoding.

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.

Also, the simplest autoencoder should be capable of keeping this number same among blocks while keeping the loss at 0.

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.

Vanilla 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.

Distorted Pruning

this doesn't change anything.

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

2.3 Efficient Structures

2.3.1 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 2.2.2 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 2.2.2, 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 calculated with Big-O notation(\mathcal{O}). 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 that 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 complexity of this operation can be expressed as $\mathcal{O}(N^2KL)$. 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. The complexity of these operations can be described as, $\mathcal{O}(NLP)$ and $\mathcal{O}(NPK)$. And summing them up, we can say $\mathcal{O}(NP(L+K))$. To determine if this decomposition is making things basic, we could basically try to define a meaningful P while trying to make sure that $\frac{NP(L+K)}{N^2KL} < 1$.

shall we cite Big \mathcal{O} ?

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

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

Experiment - MNIST Classification

Our first experiment is to apply this decomposition on a convolutional classifier for MNIST Dataset. This classifier is originally consisting of three Convolutional Layers and one Fully Connected layer. This configuration is defined in Table 2.2. By converting the

2.3.2 Bottleneck Architecture

[HZRS15] introduced residual connections with bottleneck blocks. To optimize the performance of their network, they have introduced the bottleneck blocks. These blocks start with a 1×1 convolution that scales down the number of features to half. Followed by a 3×3 convolution without changing the number of features, and following that with a 1×1 convolution to quadruple the number of features. They also introduce residual connections, but we will not talk about these now.

[HZRS15] compared the performance of their various network configurations on ImageNet validation dataset. From these comparisons, we have selected the 34-layer structure consisting of pairs of 3×3 blocks with the

layer name	output size	34-layer	50-layer
conv1	112×112	$7 \times 7, 64$, 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

Table 2.3: Comparison of bottleneck blocks (50-layer) with stacked 3×3 layers (34-layer).

54-layer structure consisting of bottleneck blocks. In table 2.3 we have compared these networks by their structure, required number of FLOPs, and their top-1 and top-5 errors on this dataset. This a clear demonstration of the effectiveness of bottleneck blocks.

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.
- [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.
- [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.

- [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.
- [SZ14] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. 09 2014.
- [ZK16] Sergey Zagoruyko and Nikos Komodakis. Wide residual networks. 05 2016.