Master's Thesis

Optimizing Neural Networks for Mobile Devices

Radboud University, Nijmegen

Erdi Çallı

August 1, 2017

Abstract

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

Contents

Chapter 1

Introduction

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

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

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

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

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

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

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

1.1 Notations

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

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

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

1.2 Neural Networks

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

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

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

denote the concepts that we just explained.

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

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

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

1.2.1 Fully Connected Layers

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

Let us assume two consecutive layers, $l^{(k-1)} \in \mathbb{R}^{m^{(k-1)} \times 1}$ and $l^{(k)} \in \mathbb{R}^{m^{(k)} \times 1}$. For these layers to be fully connected, the weight matrix connecting them

would be defined as $w^{(k)} \in \mathbb{R}^{m^{(k-1)} \times m^{(k)}}$. This structure is represented in Figure 1.1.

Most fully connected layers also include a bias term ($b^{(k)} \in \mathbb{R}^{m^{(k)}}$). The output of a fully connected layer, $o^{(k)}$, would simply be calculated using layer function $\psi^{(FC)}$ as

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

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

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

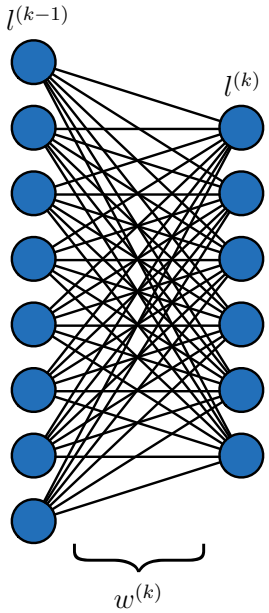


Figure 1.1: Graph representation of two fully connected layers, $l^{(k-1)}$ and $l^{(k)}$, connected by the weight matrix $w^{(k)}$.

1.2.2 Activation Function and Nonlinearity

By stacking fully connected layers, we can increase the depth of a neural network. By doing so we may be able to increase approximation quality of

the neural network. However, the $\psi^{(FC)}$ we have defined is a linear function. Therefore if we stack multiple fully connected layers using the current $\psi^{(FC)}$, we would end up with a linear model.

To achieve non-linearity, we apply *activation functions* to the results of ψ. There are many activation functions (such as *tanh* or *sigmoid*) but one very commonly used activation function is ReLU [NH10]. As shown in Figure 1.2, ReLU is defined as

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

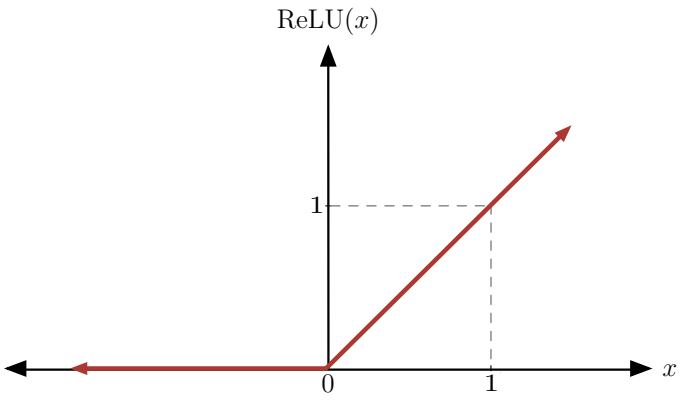


Figure 1.2: ReLU non linearity visualized.

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

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

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

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

$\psi^{(FC)}$ is one of the most basic building blocks of neural networks. By stacking building blocks in different types and configurations, we come up

with different neural network structures. The outputs of every layer, starting from the input are calculated as

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

1.2.3 Loss

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

Root Mean Square Error

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

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

Softmax Cross Entropy

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

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

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

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

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

1.2.4 Minimizing Loss

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

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

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

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

1.2.5 Convolutional Layer

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

Let us assume a 3 dimensional layer output $o^{(k-1)} \in \mathbb{R}^{\mathcal{H}_{k-1} \times \mathcal{W}_{k-1} \times m^{(k-1)}}$ where the dimensions $\mathcal{H}_{k-1}$ representing the length of the height dimension, $\mathcal{W}_{k-1}$ representing the length of the width dimension and $m^{(k-1)}$ representing number of nodes in that layer. Convolution operation first creates a sliding window of size $K \times K \times m^{(k-1)}$ that goes through height and width dimensions. The contents of this sliding window ~~would be~~ patches ($p^{(k-1)}_{(I,J)} \in \mathbb{R}^{K \times K \times m^{(k-1)}}$) where $0 < I \leq \mathcal{W}_k$ and $0 < J \leq \mathcal{W}_k$. By multiplying the weight matrix $w^{(k)} \in \mathbb{R}^{K \times K \times m^{(k-1)} \times m^{(k)}}$ to the patch $p^{(k-1)}_{(I,J)}$ centered at (I, J), we create the set of output nodes for that point $o^{(k)}_{(I,J)} \in \mathbb{R}^{1 \times m^{(k)}}$. While calculating the patches, we also make use of a parameter called stride, $s_k \in \mathbb{N}^+$. s_k defines the number of vertical and horizontal steps to take between each patch.

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

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

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

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

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

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

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

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

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

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

This direct reference is

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

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

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

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

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

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

where the weight and the bias are defined as

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

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

In other words, as shown in Figure 1.3, the output of layer is a set of vectors

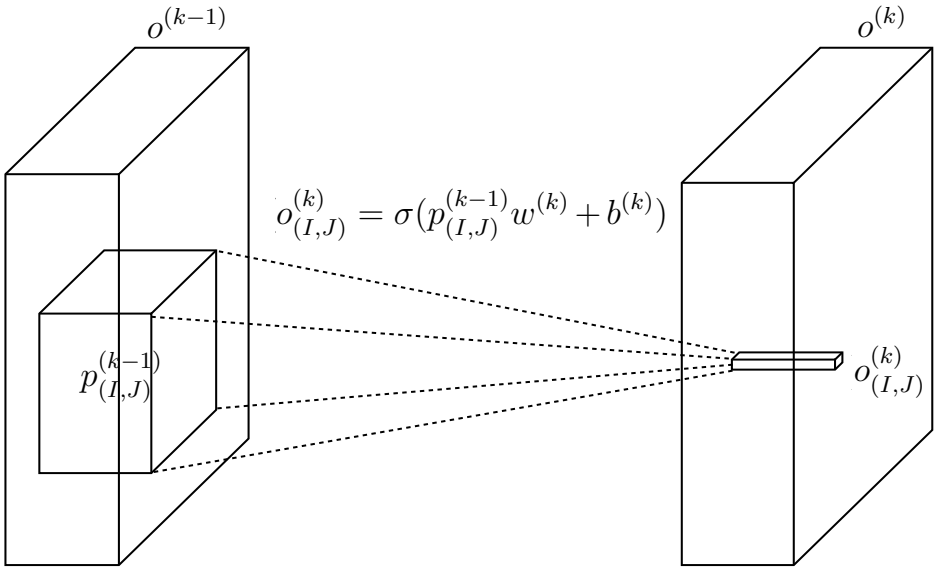


Figure 1.3: Convolution operation visualized.

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

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

1.2.6 Pooling

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

Max Pooling

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

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

Using this definition, max pooling can be defined as

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

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

Average Pooling

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

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

Using this definition, average pooling can be defined as

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

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

Global Pooling Methods

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

1.2.7 Deconvolution

Introduced by [ZKTF10], deconvolution operation aims to increase the dimensionality of an input. To do that, it basically transposes the convolution operation. Deconvolution operation creates patches of $p_{(I,J)}^{(k-1)} \in \mathbb{R}^{1\times1\times m^{(k-1)}}$ from the input, and applies a weight matrix of $w^{(k)} \in \mathbb{R}^{m^{(k-1)}\times K\times K\times m^{(k)}}$. In other words, it creates a $K \times K \times m^{(k)}$ output from every $1 \times 1 \times m^{(k-1)}$ patch and expands the height and width of the input.

1.2.8 Batch Normalization

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

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

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

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

1.2.9 Regularization

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

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

L1 Regularization

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

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

L2 Regularization

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

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

1.3 Datasets

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

1.3.1 MNIST

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

1.3.2 CIFAR10

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

1.3.3 ImageNet

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

Chapter 2

Methods

So far we have explained some neural network building blocks and some techniques for increased efficiency. In this chapter, we are going to introduce some methods to define models with reduced computational cost and some methods to reduce the computational cost of a defined model. After introducing each method, we are going to explain how we used them in our experiments.

2.1 Pruning

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

they prune the unimportant weights. They also use Fisher Information Metric to determine the number of bits to represent individual weights. Also, [Ree93] compiled many pruning algorithms.

In this study, we are going to look at two types of pruning methods, pruning connections and pruning nodes.

2.1.1 Pruning Connections

This type of pruning methods reduce the number of floating point operations by removing some connections. In other words, as seen in Figure 2.1, they remove individual values from weight matrices. In theory, removing such connections from a weight matrix (i.e. $w^{(k)}$) could benefit the computational complexity. However, in practice, we represent the connections between layers using dense weight matrices. To be able to *remove* weights in such a setting, we need to convert these dense weight matrices to sparse weight matrices. However, to our knowledge, operating with dense matrices is much faster than operating with sparse matrices, unless the sparse matrix is smaller than 90% of the weight matrix. Because of this limitation, we will not apply this method it to directly reduce the computational cost. However, we will make use of this method when we are investigating approximation methods in Section 2.2.

To determine the nodes to be pruned, we will look at one simple criteria.

Irrelevant Connections

One way to prune weights is to remove relatively irrelevant connections. To do so, we will set a threshold and remove the absolute values below that threshold. To determine this threshold we will make use of the mean and the variance of weight matrices. By finding a different threshold for different weight matrices, we will try to maximize the efficiency of this method.

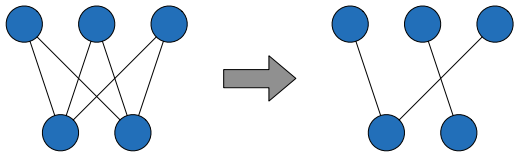


Figure 2.1: Pruning the weights of a fully connected layer. ~~Pruning the nodes of a fully connected layer.~~ The figure on the left shows the ~~weight~~ connections before pruning, and the figure on the right shows the ~~weight~~ connections after pruning.

2.1.2 Pruning Nodes

This type of pruning methods reduce the number of floating point operations by removing nodes from layers and all the weights connected to them, as seen in Figure 2.2. Let us assume two fully connected layers, k and $k+1$. The computational complexity of computing the outputs of these two layers would be $\mathcal{O}(\psi^{(FC)}_{(k+1)}(\psi^{(FC)}_{(k)}(o^{(k-1)})) = \mathcal{O}(m^{(k)}(m^{(k-1)}+m^{(k+1)})$. Assuming that we have removed a single node from layer k, the complexity would drop by $m^{(k-1)} + m^{(k+1)}$.

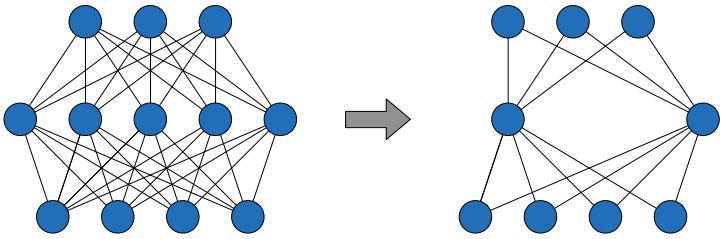


Figure 2.2: Pruning the nodes of a fully connected layer. The figure on the left shows the node structure before pruning, and the figure on the right shows the node structure after pruning.

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.

To determine the nodes to be pruned, we will look at two simple pruning criteria.

Activation Counts

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

Activation Variance

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

Training Cycles

Based on these criteria, as used by [HPTT16] and many others, we employ training cycles. First we initialize our models with random weights. After some initial training, we collect statistics based on the selected pruning criteria. Using these statistics, we prune the model. If we ~~have~~ successfully pruned any nodes, we go back to the training step and keep iterating over these steps until ~~there are no more~~ nodes to prune. The training cycles are illustrated in Figure 2.3.

2.1.3 Experiments

So far we have defined two pruning methods, pruning connections and pruning nodes. Since we cannot directly use pruning connections to reduce computational cost, we will focus on experimenting with pruning nodes. To be able to interpret our results, we will try to create some simple cases for which we can find the most optimum solution without using pruning. Knowing the most optimum solution for ~~a given~~ will ~~let us~~ evaluate the performance of different configurations. In these experiments we are aiming to find the configurations that can achieve the best results.

Fully Connected Networks

To experiment with fully connected networks, we chose to train a neural network to *predict* the summation of two inputs. As we have shown in Fig-

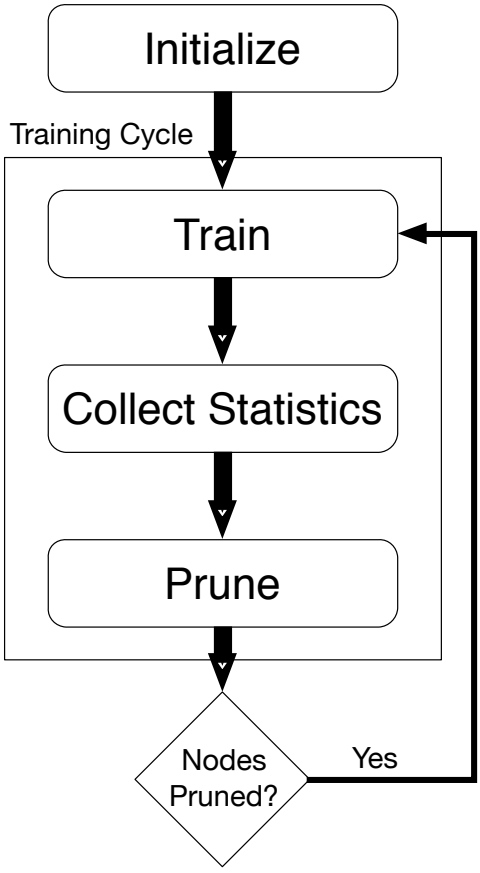
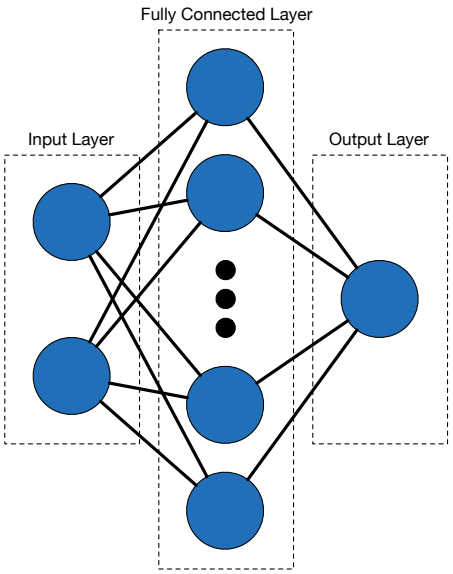


Figure 2.3: Training cycles we have defined.

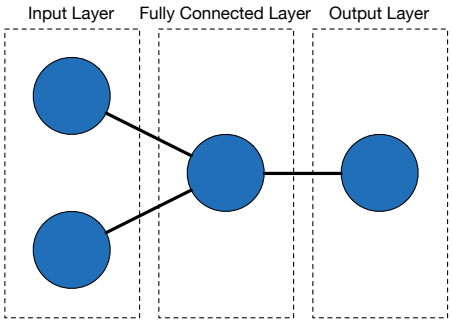
ure 2.4a, we have defined a neural network consisting of 2 input dimensions ($x_n \in \mathbb{R}^2$), one fully connected layer with 1000 nodes and one fully connected output with a single node ($y_n \in \mathbb{R}$). We have defined the expected output as the summation of two inputs, ($y_= x_{n,1} + x_{n,2}$).

Thanks to this definition, we precisely know the neural network structure that we're aiming for. As you can see in Figure 2.4b, the neural network architecture we're aiming for has only one node in its fully connected layer. If all of the weights are equal to 1 and all of the biases are equal to 0 in ~~this~~ such a setting, we can calculate the output with zero loss. To achieve such a setting, we are going to prune the nodes on that layer.

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



(a) Initial network structure.



(b) The most mathematically plausible pruned network.

Figure 2.4: (a) Neural network structure used to on the fully connected summation experiment, (b) the result we are trying to achieve.

Convolutional Neural Networks

To extend our pruning experiments to convolutional neural networks, we have trained an autoencoder on MNIST dataset.

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

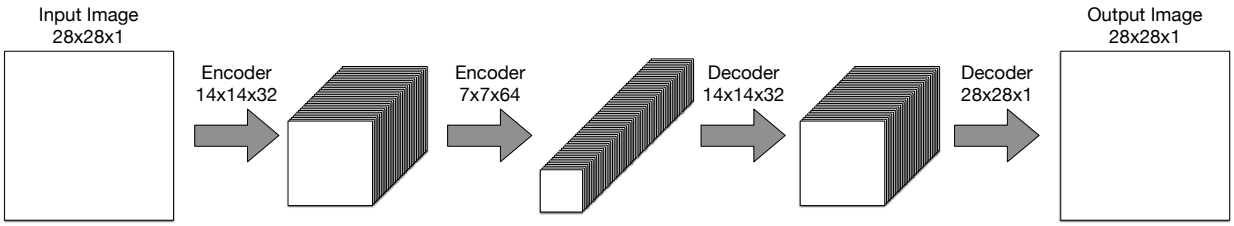
We use autoencoders because they have a clear baseline. In the baseline autoencoder, the dimensionality of the input would be equal to the output

dimensions of every layer. Assuming an input $x_n \in \mathbb{R}^{\mathcal{H}_0 \times \mathcal{W}_0 \times m^{(0)}}$, the baseline autoencoder would satisfy the following equation for every layer

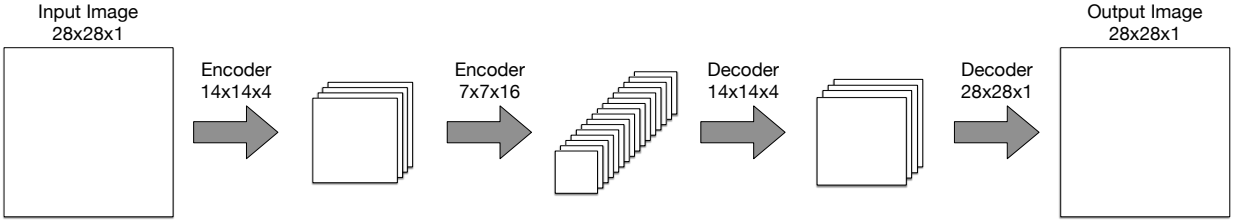
$$\mathcal{H}_k \mathcal{W}_k m^{(k)} = \mathcal{H}_0 \mathcal{W}_0 m^{(0)}$$

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

We have defined our autoencoder with two encoder and two decoder layers. Each encoder layer ($\psi_{(1)}^{(Conv)}$ and $\psi_{(2)}^{(Conv)}$) is running a convolution with kernel size 3 and stride of 2. After each encoding layer, we add bias, apply batch normalization and ReLU activation. Each decoding layer ($\psi_{(3)}^{(Deconv)}$ and $\psi_{(4)}^{(Deconv)}$) is running deconvolutions with kernel size of 3 and strides of two. After each, we add bias and apply batch normalization. The first decoding layer($\psi_{(3)}^{(Deconv)}$) is followed by ReLU activation and the last one is followed by *tanh* activation. We defined the loss as the root mean square of the input and the output of the network. The initial autoencoder configuration can be seen in Figure 3.1a and the baseline autoencoder configuration can be seen in Figure 2.5b.



(a) Initial autoencoder configuration.



(b) Baseline autoencoder that doesn't reduce the dimensionality.

Figure 2.5: Autoencoders configurations.

2.2 Approximation Methods

In this section, we are going to look at some ways to reduce the computational cost of fully connected layers and convolutional layers by approximating their results. We will look at two types of approximation methods, factorization and quantization.

2.2.1 Factorization

Factorization approximates a weight matrix as the product of smaller matrices. As explained by [ZZHS16], [DZB+14], [CS16], factorization has interesting uses with neural networks. Let us assume that we have a fully connected layer k. Using factorization, we can approximate $w^{(k)} \in \mathbb{R}^{m^{(k-1)} \times m^{(k)}}$ using two smaller matrices, $U_{w^{(k)}} \in \mathbb{R}^{m^{(k-1)} \times n}$ and $V_{w^{(k)}} \in \mathbb{R}^{n \times m^{(k)}}$. As shown in Figure 2.6, if we can find matrices such that $U_{w^{(k)}} V_{w^{(k)}} \approx w^{(k)}$, we can rewrite $\psi_{(k)}^{(FC)}$ as

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

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

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

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

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

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

$$n < \frac{m^{(k-1)} m^{(k)}}{m^{(k-1)} + m^{(k)}} \tag{2.1}$$

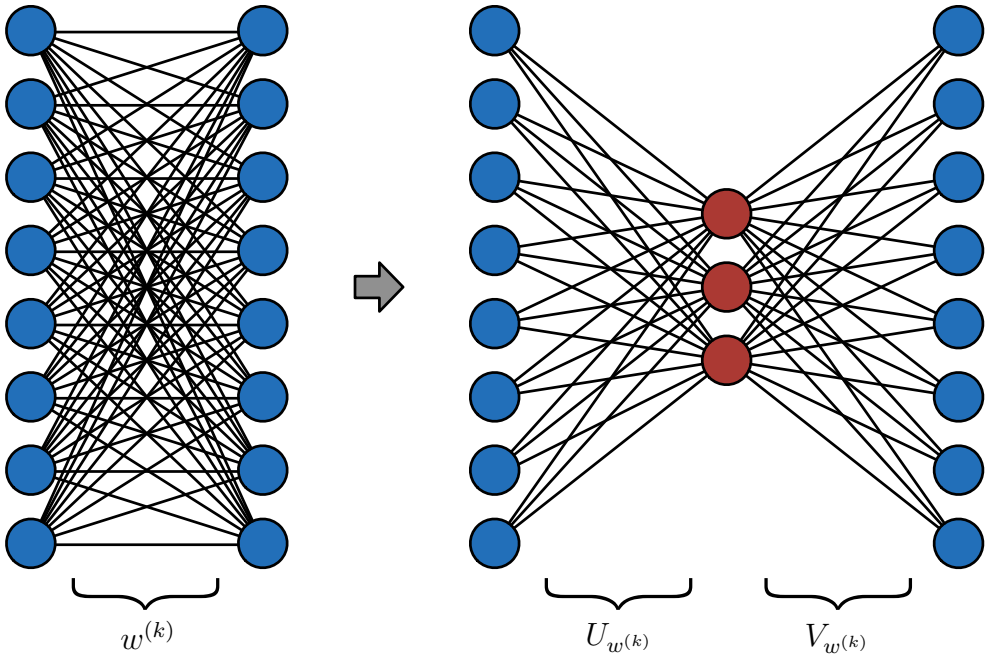


Figure 2.6: Factorization on fully connected layers.

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

SVD

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

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

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

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

where $U' \in \mathbb{R}^{m^{(k-1)} \times n}$, $V'^T \in \mathbb{R}^{n \times m^{(k)}}$, and $S' \in \mathbb{R}^{n \times n}$. By choosing a sufficiently small rank (n) satisfying Equation 2.1 and setting $U_{w^{(k)}} = U'S'$ and $V_{w^{(k)}} = V'^T$, we can approximate the weights, and reduce the complexity of a layer. [ZZHS16] applies this method to reduce the execution time of a network by 4 times and increase accuracy by 0.5%.

2.2.2 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 these bits are going to be represented with some error. Quantization is the process of representing values using less bits and some error. For example, [HMD15] uses 5-bits to represent decimals, instead of 32-bit floating point variables.

At a higher level, the computational complexity does not depend on number of bits. But if we dive deeper in the computer architecture, using less bits to represent variables provide some major advantages. As the number of bits gets smaller, the required cpu-cycles to perform an operation and the cost of transferring data from memory to cpu cache reduces. Moreover, it increases the amount of data that can fit into the cache. One 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.

2.2.3 Weight Clustering

Also as known as *hashing trick* or *feature hashing*, this method aims to represent weight matrices using a set of vaues. [NH92] trains their model with regular weight matrices. Once model is trained, they use clustering (i.e k-means) to find a set of weights($W' \in \mathbb{R}^a$) that approximate the model. Then they store the cluster index per weight in $d^{(k)} \in \mathbb{N}^{m^{(k-1)} \times m^{(k)}}$. By redefining $w_{i,j}^{(k)} = W'_{d_{i,j}^{(k)}}$, they perform weight clustering.

[CWT+15] uses a method called frequency sensitive hashing. Assuming that low-frequency weights have more importance than high-frequency weights, they cluster the weights in similar frequencies, then they create shared weights for these clusters, By doing so, they emphasize the low-frequency (high importance) features.

Please note that such methods do not necessarily reduce model complexity

~~by itself.~~ However, they reduce the model size by storing indices using less bits. In theory, such methods should also provide a lower rank in low rank decomposition.

2.3 Convolution Operation Alternatives

As we have shown in the first chapter, the computational cost of convolution operation is described as the multiplication of width, height, kernel size squared, input channels and output channels. This computational cost can be quite high when we are working with large images, or large kernels, or large input and output channels. Here we will look at some alternative methods to define convolution operations with lower computational cost.

2.3.1 Kernel Composing Convolutions

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

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

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

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

As [AP16] explained, their method forces the separability of the weight matrix as a hard constraint. By performing such an operation, they convert the computational complexity of a convolution operation to $\mathcal{O}(Kn(m^{(k-1)} + m^{(k)}))$. Suggesting that, similar to factorization methods, choosing a low rank satisfying Equation 2.1, we can define a convolution operation with less computational complexity.

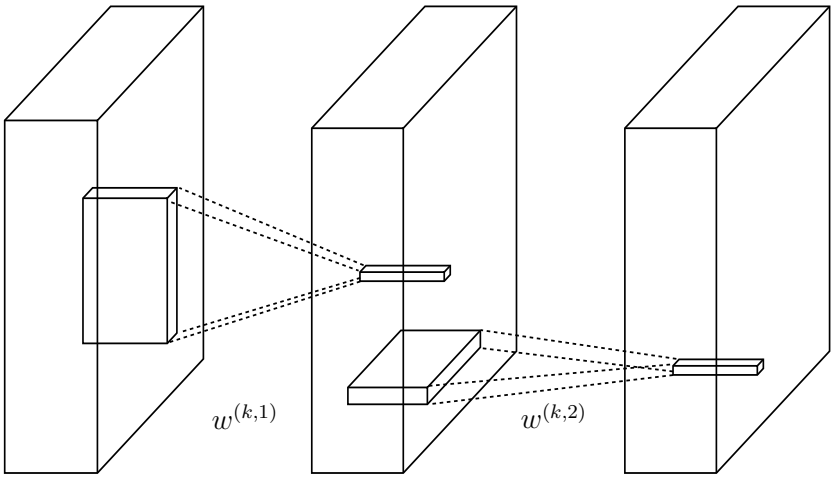


Figure 2.7: Kernel composing convolutions visualized

2.3.2 Separable Convolutions

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

Depthwise Convolution

Depthwise convolutions apply a separate convolution operation (referred to as inner convolution) on every input channel. Therefore, number of output channels of a depthwise convolution is the number of input channels times the number of output channels of these inner convolutions. In other words, it results with a number of output channels that is equal to (or folds of) the number of input channels. Unless defined otherwise, we use inner convolution operations with one output channel. Therefore the number of input channels are equal to the number of output channels for our depthwise convolutions. Please see Figure 2.8 for a visual explanation.

Formally, given a patch $p_{(I,J)}^{(k-1)} \in \mathbb{R}^{K \times K \times m^{(k-1)}}$, depthwise convolution has a single weight matrix $w^{(k,dw)} \in \mathbb{R}^{K \times K \times m^{(k-1)}}$. Let us assume that the

subscripts of patch $p_{(I,J)}^{(k-1)}$ and weight matrix $w^{(k,dw)}$ are described as $p_{(I,J),i}^{(k-1)} \in \mathbb{R}^{K\times K\times 1}$ and $w_i^{(k,dw)} \in \mathbb{R}^{K\times K\times 1}$.

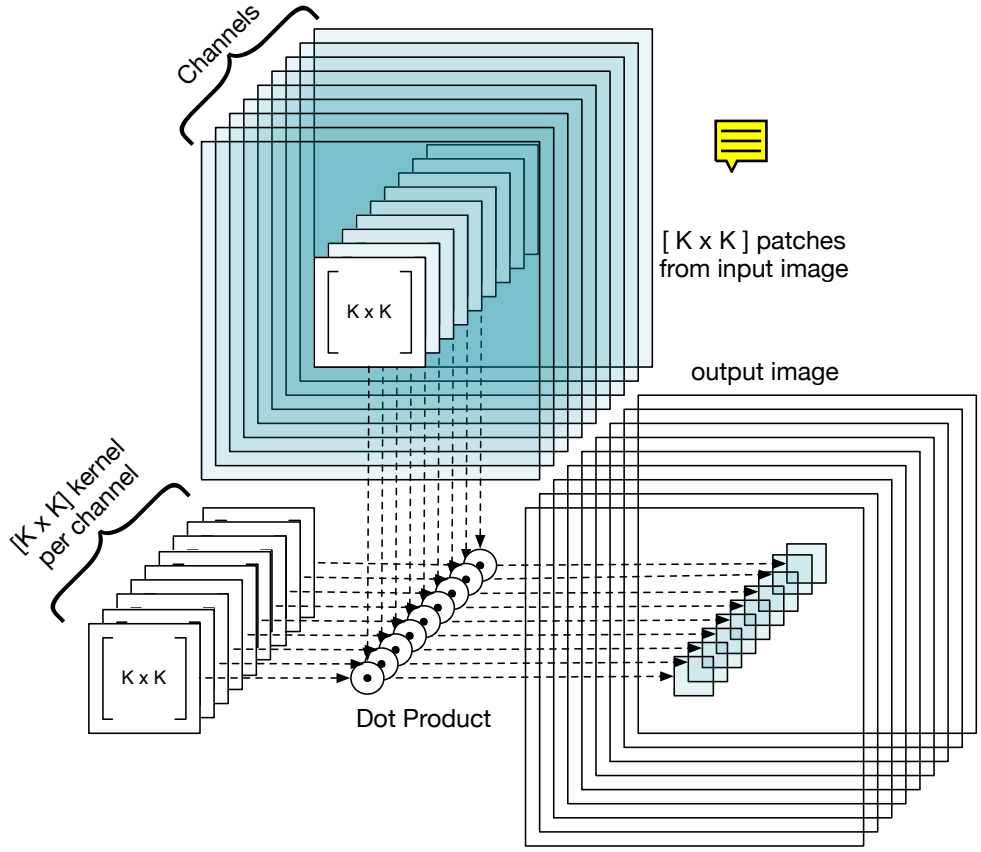


Figure 2.8: Depthwise convolution visualized

The depthwise convolution operation is defined as

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

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

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

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

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

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

Pointwise Convolution

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

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

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

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

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

The complexity of this operation is

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

2.3.3 Non-Linear Separable Convolutions

[HZC+17] proposed that, adding batch normalization and ReLU activations between depthwise and pointwise convolutions in separable convolutions would increase the model accuracy.

2.3.4 Experiments

In general, convolution operations are expensive. Here we experiment with alternative convolution operations to see which one is the better alternative. To see the differences between these operations, we try to compare them on

two tasks, one classifying MNIST dataset, the other classifying CIFAR-10 dataset. For these experiments we have defined a baseline model consisting of three convolutional layers followed by a fully connected layer. Each convolutional layer has kernel size 5. Convolutional layers are followed with bias, batch normalization, and ReLU activations. First two convolution layers are followed by a max pooling layer with kernel size 2 and strides of 2. The third convolutional layer is followed by a global average pooling layer. We have trained the model with SCE loss and momentum optimizer with momentum 0.9 and learning rate 0.1 divided by 10 in steps $[20000, 30000, 40000]$. We train this model for 50000 steps with batch size 128.

Overall, the layers and their outputs can be seen in Figure 2.9. By replacing second and third convolution operations with separable convolution, separable convolution with nonlinearity and kernel composing convolution, we obtain 3 alternative models for comparison.

We don't change the first convolution operation because using an alternative operation does not reduce the computational cost in case of 3 input channels.

We also use a few operations using large kernels (5×5). First, kernel decomposition only provides speed up for large kernels. Second, we try to ~~come up~~ with a small model that can train fast, so that we can run these experiments multiple times.~~s~~

2.4 Small Models

Another approach to creating neural networks with low computational cost is to design smaller models. These type of models would be designed to have small input dimensions or do aggressive dimensionality reduction in first layers, use alternative convolution operations, have fewer layers and/or have less number of features per layer. For example, [HZC+17] introduced a series of models with varying input dimensions and number of features. Here, we will define and train some small models for CIFAR-10 and ImageNet datasets. While defining our model, we will make use of the knowledge we have gained from the experiments we ran in Section 2.3. We will also use the methods we introduced in Sections 2.2 and 2.1 and try to make our models even smaller.

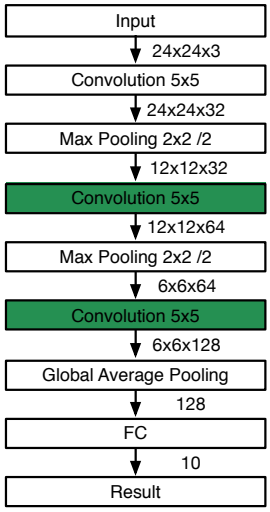


Figure 2.9: Baseline model visualised. Second and third (green) convolution operations are replaced with 3 other alternatives for comparison.

2.4.1 Models

Both models that we have defined were inspired by the [HZRS15] which introduces a connection that improves the performance of a neural network. Before going into the details of our models, we will explain the concepts introduced by [HZRS15].

Residual Networks

[HZRS15] introduced residual connections and they called the convolutional neural networks having these connections as residual networks (ResNet). They show that residual connections improve the network performance. According to their results, as the network gets deeper, the effect of adding residual connections increases. [HZRS15] trains neural networks up to 1000 layers.

However, using residual connections, [ZK16] achieves a better performance by increasing the number of features per layer, instead of adding more layers to ResNets. In other words, they show that having wider and shallower networks works better than having narrower and deeper networks. They use a ResNet with 16 wide layers to outperform the original 1000 layer

ResNet model in various tasks.

Since residual connections increase the performance of a given network with very little overhead, we will make use of them and see how they perform in small models.

Definition of Residual Connections

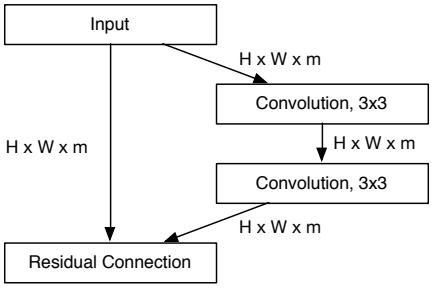
Let us assume *blocks* to be repeated groups of consecutive layers in a neural network and define them as we have defined layers, so that the output of a block is $o^{(b)}$ and the function(layers) that this block applies is $\psi_{(b)}$. We create a residually connected block when we add the input of a block to the output of the last layer in the block to calculate the input of the next block. Let us assume a block b with input $o^{(b-1)} \in \mathbb{R}^{m^{(b-1)}}$ and output $o^{(b)} \in \mathbb{R}^{m^{(b)}}$. We call this block residually connected when $o^{(b)} = o^{(b-1)} + \psi_{(b)}(o^{(b-1)})$.

Residually Connected Blocks

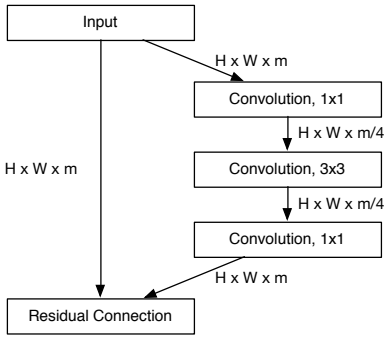
[HZRS15] introduced two types of residually connected blocks. The first is called a *residual block*, consisting of two convolution operations and a residual connection between the input and the output of the block. The second is called a *residual bottleneck block*, consisting of three convolution operations. First reducing number of channels with a one by one kernel, second applying a three by three kernel, third applying another one by one kernel to increase the number of dimensions. They have used residual blocks to train networks up to 34 layers. For networks having 50 or more layers, they have used the residual bottleneck block. Their 50 layer network using residual bottleneck blocks achieves 22.85% top-1 error rate on ImageNet dataset while their 34-layer network is achieving 25.3% top-1 error rate. For our models we chose to use the residual blocks. 1×1 convolutions in residual bottleneck blocks are relatively more expensive and we cannot reduce their cost using an alternative operation. Furthermore, residual bottleneck blocks are not a good fit for smaller networks because they do not expand the receptive field as aggressively as residual blocks.

Separable Residual Blocks

Except for the first convolutional operation, we have replaced every convolution layer with a separable convolution layer. The first layer has 3 input and 32 output channels. Therefore, for this layer the number of FLOPs for



(a) A residual block.



(b) A residual bottleneck block.

Figure 2.10: Residually connected blocks, visualized as proposed by [HZRS15]. ReLU non-linearity, batch normalization and identity mappings hidden for simplicity.

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

Following [HZRS15], [HZRS16] proposed that using full pre-activation residual connections increases the model performance. This type of residual connections residually connect the outputs before ReLU activation and batch normalization. We have illustrated these connections in Figure 2.11.

We apply the strides in the first depthwise convolution. We increase the number of channels in the first pointwise convolution. As shown in Figure 2.11

CIFAR-10 Model

For CIFAR-10, we define a very small model consisting of a convolutional layer followed by 6 separable residual blocks. The model is illustrated in

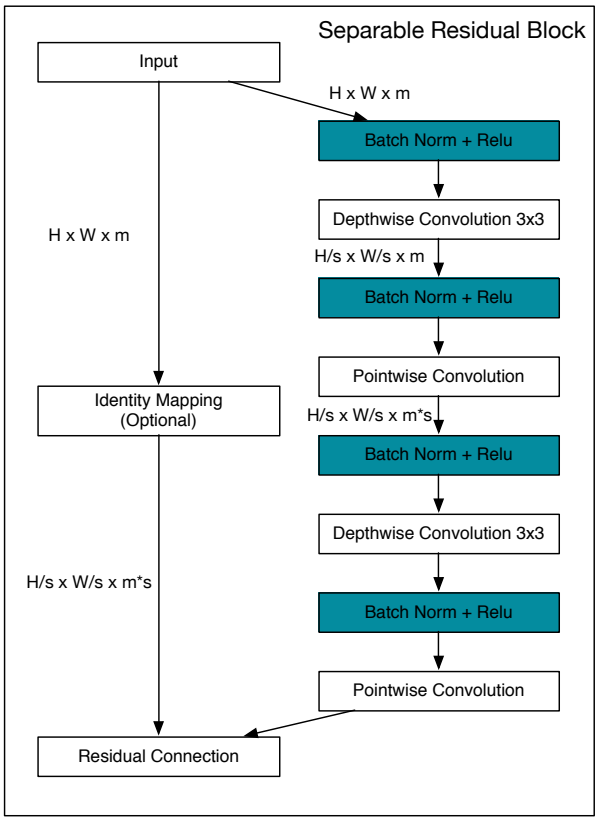


Figure 2.11: Full pre-activation separable residual block. H/s and W/s represents the strides applied to height and width dimensions respectively. m represents the output channels and $m * s$ represents multiplying number of channels by the strides. Dimensions are not shown if they do not change between two layers.

Figure 2.12. For identity mapping we have used average pooling with strides and kernel size of two. To multiply the number of channels, we have padded the feature matrices with zeros.

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

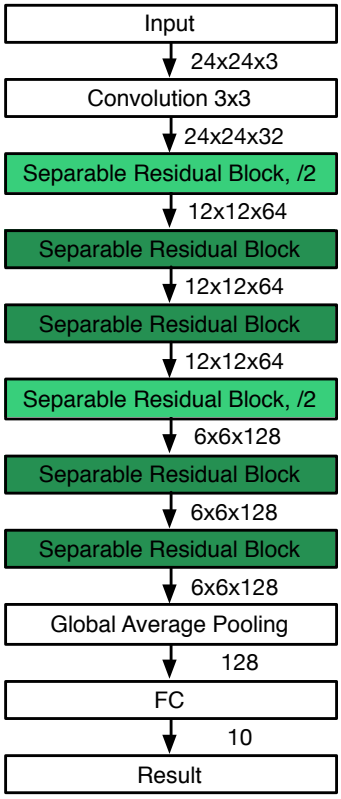


Figure 2.12: Model trained for CIFAR-10. Light green blocks reduce the image dimensionality by two while multiplying the number of features by 2.

trained our model using the training images for 80.000 steps with batch size 128.

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

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

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

~~For CIFAR 10 training, we make some changes in our model. Since we have defined our input as a 24×24 image, we can apply a total of three convolutions with stride of two. After these, the image dimensions become 3×3. After this point for our convolutions to make sense, we can not apply convolutions with strides of two until we apply global average pooling. Therefore, we remove the strides from green and pink blocks in Figure 2.13 and we do not multiply the number of channels by two.~~

ImageNet Model

We recreated the Resnet-34 using ~~model choices defined above.~~ The resulting model is shown in Figure 2.13. For identity mappings, we have used 1×1 convolutions with strides of two because [HZRS15] shows that they work better for ImageNet dataset.

Aggressive Dimensionality Reduction in ImageNet

In [HZRS15], models defined ~~with~~ ImageNet start with a 7×7 convolution with strides of two ~~and it is~~ followed by max pooling layer with strides of two and kernel size 2. If we think about this design choice, we see that the kernel size choice (7×7) is to cover the receptive field to minimize the loss of information caused by two consecutive layers with strides of two. From another point of view, those two layers decrease the image size from 224×224 to 56×56. This decreases the computational cost of future layers. When we apply such a convolution, the first convolutional layer becomes very complex compared to the rest of the network. To prevent that we propose a different first layer. We change this 7×7 convolution followed by a max pooling layer with two 3×3 convolutions Following that, before applying the max pooling layer, we apply a 3×3 depthwise convolution that multiplies the number of channels with two. Then we apply a 1×1 convolution (pointwise) to that. By doing so we reduce the complexity of these layers about four times.

We trained our model in ImageNet training dataset. We used gradient descent optimizer and learning rates 0.01, 0.001, 0.0001 for steps 0 to 150.000, 150.000 to 200.000 and 250.000 to 300.000 respectively. We have defined the loss with SCE of the truth and prediction, with an addition of L2 norm of weights multiplied by 0.001. We train our model using the training images with batch size 128.

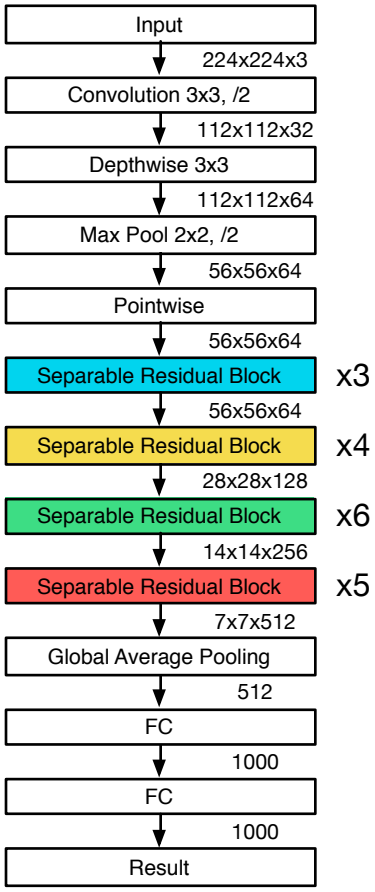


Figure 2.13: Resnet-34 recreated using separable residual blocks and proposed input layers. ReLU and Batch Normalization operations are hidden. Dimensionality is reduced in the first block of second (yellow), third(green) and fourth(red) repeated separable residual blocks.

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

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

After training these models, we try to apply the methods we have defined in Sections 2.1, and 2.2 to see how they perform with small models.

2.4.2 Pruning Small Models

To be able to prune nodes on these models, we have defined two new pruning routines.

Pruning Residual Connections

The addition operation in the residual block creates a one-to-one relationship between the output channels of different blocks. With the existence of such a relationship, it is not possible to prune a residual block's output nodes without pruning the former or latter blocks. Therefore, we group the residual blocks that are connected to each other without identity mappings and refer them to as directly connected (in Figure 2.13, residual blocks represented with the same colors are directly connected). Then we calculate the ~~indexes of~~ nodes to keep ~~from the output of~~ every residual block. We union the nodes to be kept in those layers and prune the remaining nodes from the inputs and outputs of the directly connected residual blocks. We also prune the pointwise convolutions within residual blocks separately.

Pruning Depthwise Convolutions

Since depthwise convolutions multiply their input channels by a number, their inputs and outputs have the same number of channels. Therefore, if ~~some~~ output features of the previous layer are pruned, same features from the input channels of the next layer should also be pruned.

2.4.3 Approximating Small Models

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

Pruning Connections

We set weight values that are smaller than a threshold to 0. Despite the fact that this operation does not change computational complexity, when

combined with factorization, it increases the number of zeros within the matrix. Therefore it helps with finding a lower rank.

Weight Clustering

We round the weights after the second decimal. This is a very basic method of weight clustering, however it may help finding a lower rank.

Factorization

We factorize the trained, pruned and rounded weights using SVD. We can not apply this method to depthwise convolution operation. So we only factorize the convolution, pointwise convolution and fully connected weights. To do that, we calculate U, S and V for each of the weight matrices. We lower the rank of the decomposition by one. Then, we calculate the approximation error using the L2 (euclidian) distance between the original matrix and the low rank weight matrices. To be able to scale that for the number of parameters in the matrix, we divide this distance to the number of parameters. We keep reducing the rank by 1 while the approximation error is above the predefined error threshold. When the error threshold is reached, we check if this low rank decomposition would actually reduce the complexity. If it does, we ~~factorize those layers~~. Otherwise we use the original weights.

Quantization

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

2.5 Experiments

In Table 2.1 we provide a summary of the experiments we ran.

Method	**Model**	**Dataset(s)**	**Variables**
Pruning Nodes	fully connected	summation	regularization, distortion, activation counts, activation variance
Pruning Nodes	autoencoder	MNIST	regularization, distortion, activation counts, activation variance
Convolution Alternatives	baseline model	CIFAR-10 MNIST	convolution, kernel composing, separable non-linear separable
Small Models	CIFAR-10 Model	CIFAR-10	Number of layers, number of features
Small Models	ImageNet Model	ImageNet	
Approximation Methods + Pruning Weights	CIFAR-10 Model	CIFAR-10	Error Threshold Pruning Threshold

Table 2.1: A summary of all experiments we ran.

Chapter 3

Results

3.1 Pruning

In this section we consider the configurations that have pruned the most nodes as our results. Since the combination of variables and parameters that we have used are very big, we chose not to report the configurations that are not working.

As we have explained in Chapter 2, we have pruned the nodes of ~~the~~ fully connected neural network trained to predict the summation of two floating point values, and ~~the~~ autoencoder trained on the MNIST dataset. We ran the experiments at least three times with each configuration to verify our results.

3.1.1 Fully Connected Networks

We have initialized and ~~ran~~ training cycles on the fully connected network. By applying distortions to remaining weights between training cycles, we have achieved the optimum result we have shown in Figure 2.4b, with only one node in the hidden layer.

Using distortions, both ~~activation count and activation variance~~ pruning criteria worked. For both criteria, we were able to achieve the optimum network structure using a fixed threshold of 0. In other words, for activation count criteria, we have pruned the nodes that were not being activated, and for activation variance ~~statistics~~, we have pruned the nodes that had zero variance in values.

The loss was almost zero for the training and test datasets. ~~However, even though~~ we have achieved the optimal shape, the model was overfitting for the mean and standard deviation parameters we set for the random number generator while generating the training dataset.

In the experiments that we did not apply distortions we could not find a configuration that achieves the optimal network structure. In most of the configurations we were unable to find any nodes to prune after the first training cycle. We could not see any difference between different regularization terms or pruning criteria.

3.1.2 Convolutional Neural Networks

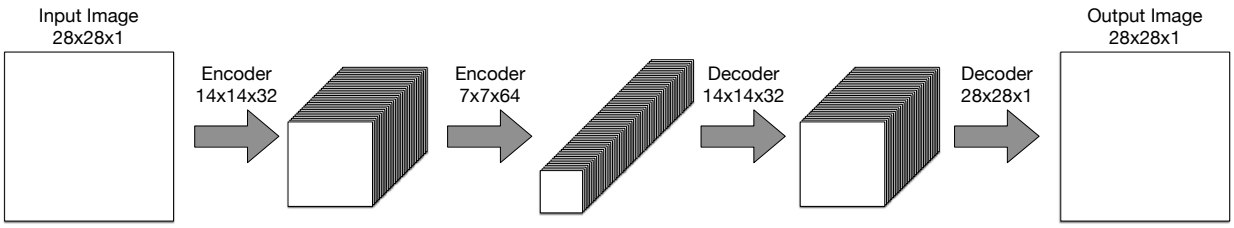
In this setting we did not see any improvement by using distortions. Using L2 regularization, compared to no regularization and L1 regularization, we have seen an improvement in the number of nodes pruned in every training cycle and the final result.

We were unable to find a good threshold for activation count criteria. However, doing a basic outlier selection on the activation variances, we were able to achieve the most optimum results. Given that x is the variance vector for per node given the dataset, by choosing a lower and upper boundaries as,

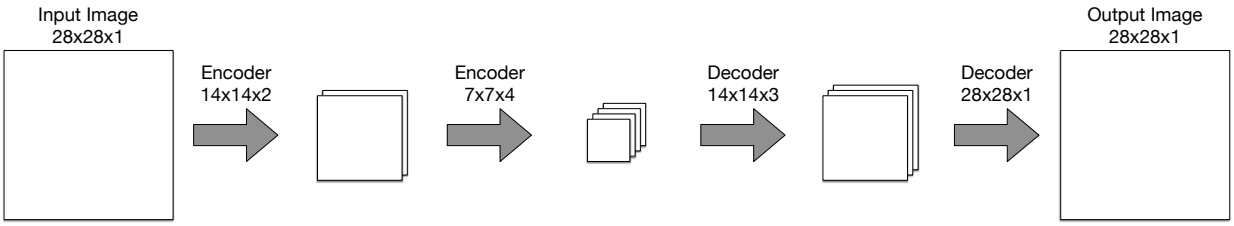
$$mean(x) - 2 * var(x) < x < mean(x) + 2 * var(x)$$

and removing the nodes that are ~~not~~ outside these boundaries, we were able to find the most optimum solutions.

The most ideal case was with no distortion, l2 regularization (with $\lambda = 0.01$) and pruning nodes based on activation variance criteria given above. Using this setting, we have pruned the autoencoder from $1-32-64-32-1$ to $1-2-4-3-1$ nodes per layer, which is a much better result compared to the baseline we have defined. It took us 10 ± 4 training cycles to achieve this result. The loss we have achieved is very close to 0. The encoder we have achieved is shown in Figure 3.1.



(a) Initial autoencoder configuration.



(b) Resulting autoencoder configuration.

Figure 3.1: Pruned autoencoder compared to initial autoencoder.

3.2 Convolution Operation Alternatives

We ran experiments to see which operation is a cheaper alternative to convolution operation. We ran our experiments 10 times to validate our results.

3.2.1 MNIST

In our experiments with MNIST dataset, we have not seen a comparable difference between experiments with different operations. All of the experiments have resulted with $99 \pm 0.3\%$ top-1 accuracy, with no clearly visible difference.

3.2.2 CIFAR-10

The results of our experiments in CIFAR-10 dataset are given in Table 3.1. As emphasized in the table, separable convolution operation with non-linearity performed slightly better than the rest of the operations.

Model	Mean Accuracy	Max Accuracy
Convolution (Baseline)	81.84	82.56
Kernel Composing Conv.	81.98	82.51
Separable Convolution	82.11	82.53
Separable Convolution with non-linearity	**82.16**	**82.75**

Table 3.1: Top-1 accuracy results for alternative convolution operations in CIFAR-10 test dataset. Best results are emphasized with bold fonts.

Model	# params	# layers	# ops	top-1 accuracy (%)
Our Model	**0.12**	27	$\sim$ **13M**	91.10
ResNet-20	0.27	20	$\sim$ 40M	**91.25**

Table 3.2: Our small model compared with ResNet-20 from [HZRS15].

3.3 Small Models

In this section we will present the results of our experiments on small models. We show how small models perform compared to large ones and see how pruning and approximation methods work with them.

3.3.1 Models

CIFAR-10

The model we have defined for this task (see Figure 2.12), has achieved a maximum of 91.1% top-1 classification accuracy on CIFAR-10 test dataset in 10 training sessions. Compared to ~~the [HZRS15] model, ResNet-20,~~ our model has performed slightly worse in terms of top-1 classification accuracy. However, in terms of floating point operations and number of parameters, our model is about 2 times smaller in terms of number of parameters, and requires 4 times less floating point operations to perform an inference.

ImageNet

The model we have defined for this task (see Figure 2.13), has achieved a maximum of 63 % top-1 classification accuracy on ImageNet test dataset. ~~We only had a three chances to experiment with this model.~~ Compared to

the original model, ResNet-34 from [HZRS15], our model has performed very poorly.

3.3.2 Pruning Small Models

Using the pruning criteria we have defined for autoencoders, we have tried to prune the CIFAR-10 model. However, we were unable to prune a significant amount of nodes and recover the accuracy ~~at the same time.~~ 

3.3.3 Approximating Small Models

We ran the approximation tool we have defined on our best model (91.1 % top-1 accuracy) from our CIFAR-10 experiments. Using various pruning thresholds and error thresholds, we could not find any factorization that would make this model faster. 

3.3.4 Quantization

We have quantized the CIFAR-10 model, converting 32-bit floating point operations to 8-bit floating point operations. However, in our benchmarks, we have seen that this method has slowed down the inference speed by almost half. We haven't seen any significant change in the model accuracy.

Chapter 4

Discussion

Here we will try to criticize our decisions in model and method selection and some of our results. We will also talk about the problems we have faced during this research.

Model Selection for Pruning Nodes

To select the best nodes to prune, we tried to make use of some pruning criteria. However, in our experiments, we have seen that randomly pruning nodes and leaving out some of them also produces similar results. This makes us question the integrity of our methods and models for given task.

Also, in our experiments, we started with very large initial models for the given problem. We believe that this may have left a false impression. As it did in our experiments, pruning a model would not reduce the model complexity by 1000 times for every model. As we have seen with small models, pruning will help us reduce the computational cost, depending on the model size and the problem definition.

Model Approximation

Similar to our pruning experiments, when we applied factorization on large models, we were able to reduce the complexity significantly. But when the model is compact enough, the gains from these methods become trivial.

Operation Comparison

In Section 2.3, we have defined a neural network to compare convolution, separable convolution and kernel composing convolution operations. Before our experiments, we have tried to find the best settings for some parameters, such as learning rate, regularization constant and optimizer. We think that using the same settings may have influenced our results. Especially because the number of parameters change considerably when we use separable convolutions, instead of convolutions.

Training for ImageNet

Compared to MNIST and CIFAR-10, ImageNet is a very large dataset. Using CIFAR-10, we were able to search the parameter space for small models that perform well. However, since training a model for ImageNet takes days with the available equipment, we were unable to search the parameter space for small models.

Using Tensorflow

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

- We do not implement lower level operations (such as convolutions). It gives us the opportunity to focus on higher level implementations, such as pruning, or factorization.
- Most of the operations are highly optimized for many platforms and devices. If we were to implement a model in C++, we'd have to implement it twice, one for training in GPU and another for running in the mobile device.
- Tensorflow provides the necessary tools to deploy models on mobile devices.

And it comes with some disadvantages, such as:

- When we started our work, Tensorflow was in version 0.10. By the date we write this, it is on 1.2. There have been 4 major releases that we had to modify our codebase for.

- Not all operations are properly implemented. For example, before version 1.2, Tensorflow implementation of separable convolutions were not very well optimized. They were as fast as convolution operations. Before that we could only hope that they would optimize their implementation.

- It is difficult to implement operations (e.g. ef operator [AYN+17]) or play around with existing ones. The documentation describing C++ internals and build procedures (as of Tensorflow 1.2) are not good enough.

- Tensorflow does not provide tools to implement low-bit variables (e.g. a 2-bit integer). So it is not possible to implement some methods that make use of variable width decimals. This limitation makes some methods impossible to use or useless. For example it is not possible to use methods that represent weights using variable width decimals. Also, storing low bit weight indices in combination with a small global weight array to reduce the model size is useless. Since we can not use low bit integers to represent these indices, our model size does not shrink at all.

Chapter 5

Conclusion

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

In our experiments we have observed that the models using separable convolutions with non-linearity result with a slightly better accuracy compared to models using convolution or kernel compositing convolution operations. We also saw that kernel composing convolution operation is a good alternative to convolution operation, only if the kernel size is larger than 3. However, since most state of the art models use convolutions with kernel size 3, kernel composing convolutions are not compatible for comparison.

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

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

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

Bibliography

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

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

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

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

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

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

[CWT+15] Wenlin Chen, James T Wilson, Stephen Tyree, Kilian Q Weinberger, and Yixin Chen. Compressing convolutional neural networks. *arXiv preprint arXiv:1506.04449*, 2015.

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

[DZB+14] Emily L Denton, Wojciech Zaremba, Joan Bruna, Yann LeCun, and Rob Fergus. Exploiting linear structure within convolutional networks for efficient evaluation. In *Advances in Neural Information Processing Systems*, pages 1269–1277, 2014.

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

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

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

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

[HMD15] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

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

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

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

[HZC+17] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto,

and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[VWB16] Andreas Veit, Michael J Wilber, and Serge Belongie. Residual networks behave like ensembles of relatively shallow networks. In *Advances in Neural Information Processing Systems*, pages 550–558, 2016.

[ZK16] Sergey Zagoruyko and Nikos Komodakis. Wide residual networks. 05 2016.

[ZKTF10] Matthew D Zeiler, Dilip Krishnan, Graham W Taylor, and Rob Fergus. Deconvolutional networks. In *Computer Vision and*

Pattern Recognition (CVPR), 2010 IEEE Conference on, pages 2528–2535. IEEE, 2010.

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